Title:

Reverse Engineering: Anti-Cracking Techniques

Date:

April 12th 2008

Website:

http://www.astalavista.com

Author:
Nicolaou George
Mail:
ishtus@astalavista.com

Author:
Charalambous Glafkos
Mail:
glafkos@astalavista.com

Table of Contents

Introduction

This paper is a guide into better understanding most of the approaches a reverse engineer can follow in order to achieve his goal. Additionally, it includes a number of advices on how to better protect your software against tracing its sensitive information, like serial key checks and authentication procedures. This paper is not about changing anyone's ideals; this paper is about people that believe that reverse engineering can create a safer world. So if you are not one of those individuals then stop reading, for this is not for you.

Note that this paper might not cover the wide range of techniques used by reverse engineers so if you feel that something is missing, please do not hesitate to email with your suggestions.

TODO

Subjects to cover:

- PE packers and crypt tools
- Online checks
- Malware analysis
- x64 reverse engineering
- Discovering and exploiting vulnerabilities

Any other suggestions are welcome. If you feel that you have something to contribute and/or offer, do not hesitate to email.

Reverse Engineering Tools

A number of reverse engineering tools are available over the net, a number of them are free and others need purchasing. Some of the most advanced disassembling and debugging tools out there are:

- OllyDBG [http://www.ollydbg.de/] (Version 2 expected soon)
- IDA Pro Disassembler and Debugger [http://www.hex-rays.com/]
- W32Dasm [http://www.google.com] (Old, but you will be amazed with some of its functions)
- SoftICE (Stopped being supported from April 2006)
- WinDbg [http://www.microsoft.com/whdc/devtools/debugging/default.mspx]

Additionally, a number of other tools can be used as well. The names of the tools and their description are listed below:

- PROTECTiON iD [http://pid.gamecopyworld.com/]
  Used for scanning windows system executables for known packer/encryprtor signatures and identifying the compiler of the program [http://en.wikipedia.org/wiki/Executable_compression]
- Import REConstructor [http://www.google.com/]
  Used for repairing damaged import table (IAT) of executables
- System Internals [http://technet.microsoft.com/en-us/sysinternals/default.aspx]
  Programs like FileMon, RegMon can be used to monitor the program's behavior. An alternative approach to this is a sandbox that provides information for all program activities.

Reverse Engineering Approaches

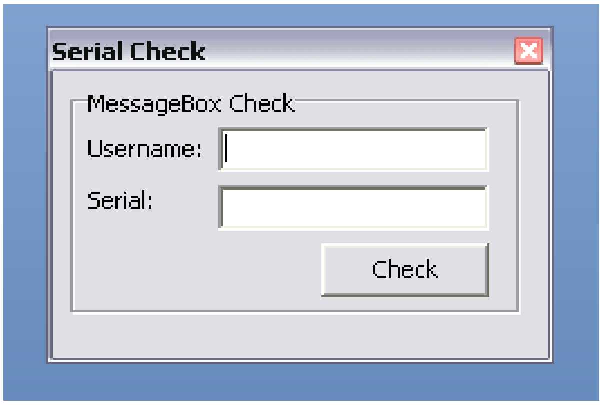
We will begin looking into the approaches a reverse engineer uses. The preferred debugger used in this section will be a modified version of OllyDBG, the original version will do as well.

Example Software

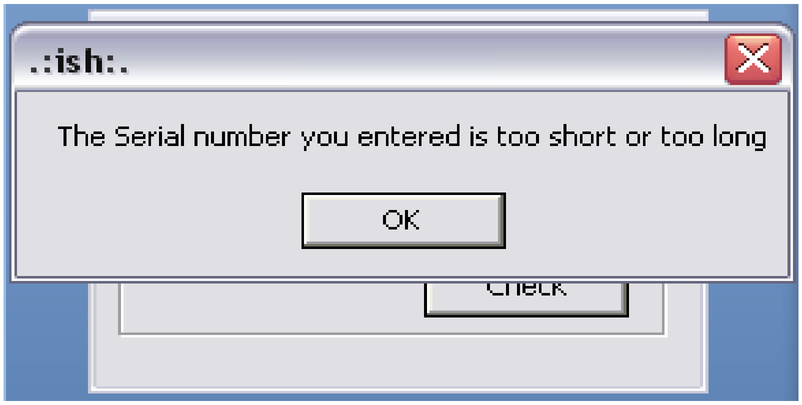
Program Name: Example.v1.0.exe (Serial Check)
Md5sum: 4c78179f07c33e0f9ec8f2b0509bd069
Compiler: Borland Delphi

Program Analysis

To begin with, we need to analyze the program functionality in order to determine our approach and better understand how it works.



As you can see, the program form is simple. The main functionality is a username and serial check. Our first step is to insert random data inside the Text boxes, click “Check” and observe the program response.



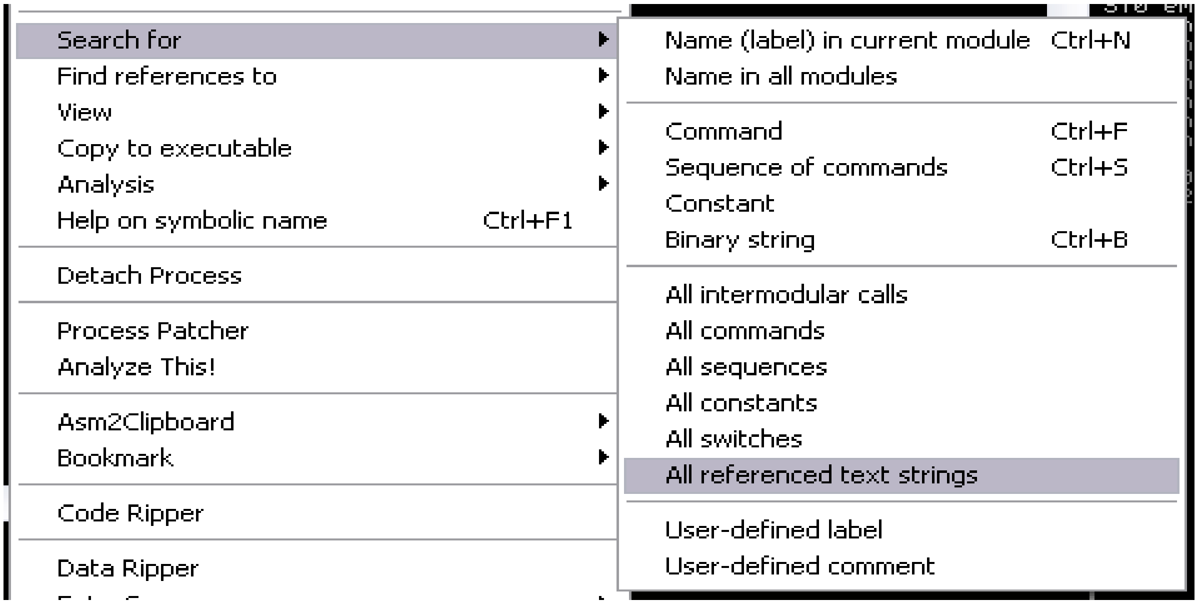
The Serial number you entered is too short or too long

The result gives us a hint that before the serial check algorithm we should expect a function that checks if the Serial string length within the given boundaries by the programmer.

Next, we move to the stage of disassembling and debugging the application in order to gather more information regarding the way it works. What is going to follow is a number of approaches a reverse engineer might use and some suggestions on hardening your software.

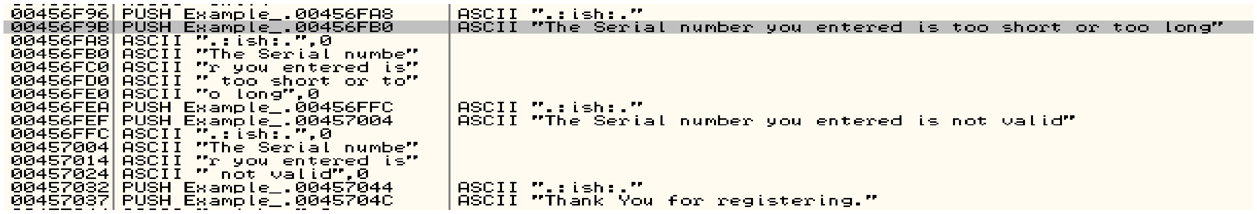
Approach No1 (String References)

Step 1:
Right Click > Search for > All referenced text strings



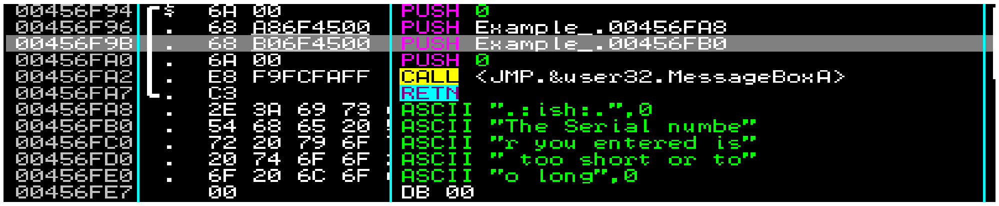
Step 2:

As you can see, the message text string easily links to the dialog box. By double clicking on the string, you get transferred directly to the dialog procedure



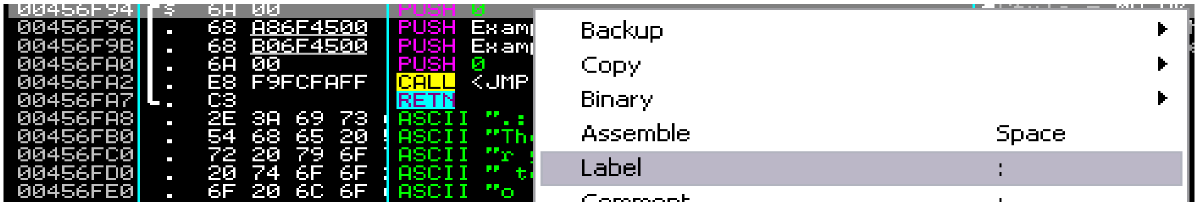
```
00456F96 PUSH Example_.00456FA8       ASCII ".:ish:."
00456F9B PUSH Example_.00456FB0       ASCII "The Serial number you entered is too short or too long"
00456FA8 ASCII ".:ish:.",0
00456FB0 ASCII "The Serial numbe"
00456FC0 ASCII "r you entered is"
00456FD0 ASCII " too short or to"
00456FE0 ASCII "o long",0
00456FEA PUSH Example_.00456FFC       ASCII ".:ish:."
00456FEF PUSH Example_.00457004       ASCII "The Serial number you entered is not valid"
00456FFC ASCII ".:ish:.",0
00457004 ASCII "The Serial numbe"
00457014 ASCII "r you entered is"
00457024 ASCII " not valid",0
00457032 PUSH Example_.00457044       ASCII ".:ish:."
00457037 PUSH Example_.0045704C       ASCII "Thank You for registering."
```

Step 3:
Although the program Serial Check is coded with a level of difficulty, a reverse engineer with little experience can trace where this function is called and patch the program flow



```
00456F94  $  6A 00          PUSH 0
00456F96  .  68 A86F4500    PUSH Example_.00456FA8
00456F9B  .  68 B06F4500    PUSH Example_.00456FB0
00456FA0  .  6A 00          PUSH 0
00456FA2  .  E8 F9FCFAFF    CALL <JMP.&user32.MessageBoxA>
00456FA7  .  C3             RETN
00456FA8  .  2E 3A 69 73    ASCII ".:ish:.",0
00456FB0  .  54 68 65 20    ASCII "The Serial numbe"
00456FC0  .  72 20 79 6F    ASCII "r you entered is"
00456FD0  .  20 74 6F 6F    ASCII " too short or to"
00456FE0  .  6F 20 6C 6F    ASCII "o long",0
00456FE7     00             DB 00
```

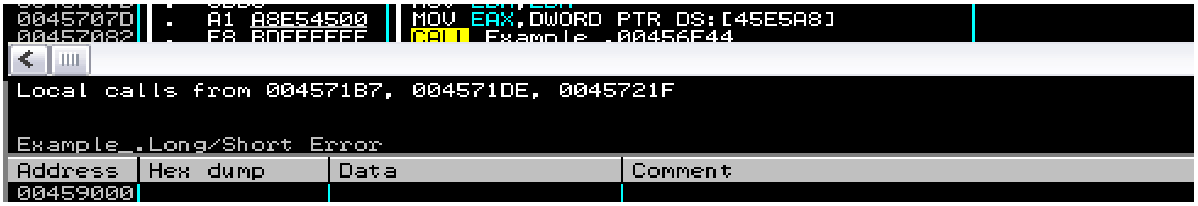
Step 4:
For now, we set a Label on the start of this function (for easier reference). We do that by:
Right Click > Label



```
00456F94 $ 6A 00        PUSH 0
00456F96 . 68 A86F4500  PUSH Examp
00456F9B . 68 B06F4500  PUSH Examp
00456FA0 . 6A 00        PUSH 0
00456FA2 . E8 F9FCFAFF  CALL <JMP
00456FA7 . C3           RETN
00456FA8 . 2E 3A 69 73  ASCII ".:
00456FB0 . 54 68 65 20  ASCII "Th
00456FC0 . 72 20 79 6F  ASCII "r
00456FD0 . 20 74 6F 6F  ASCII " t
00456FE0 . 6F 20 6C 6F  ASCII "o
```

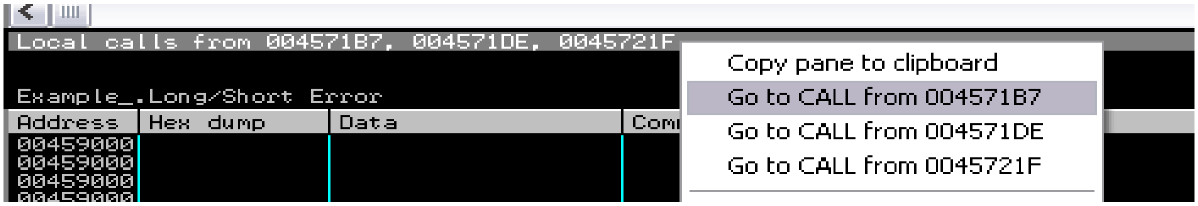
And set a Label: “Long/Short Error”

Step 5:
As you can see at the bottom, this function is called from 3 different addresses which are fairly close to each other.



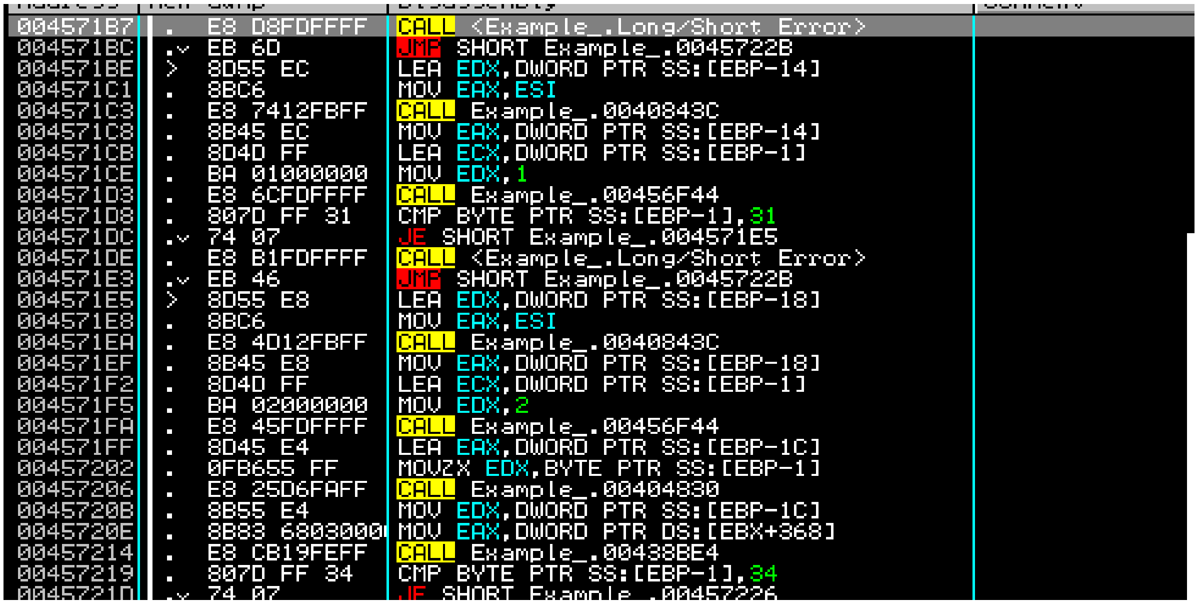
```
0045707D . A1 A8E54500  MOV EAX,DWORD PTR DS:[45E5A8]
00457082 . E8 BDFFFFFF  CALL Example_.00456F44

Local calls from 004571B7, 004571DE, 0045721F

Example_.Long/Short Error
Address   Hex dump   Data   Comment
00459000
```

Step 6:
We trace back to the first occurred call (004571B7)
Right Click > Go to CALL from 004571B7



```
Local calls from 004571B7, 004571DE, 0045721F

Example_.Long/Short Error
Address   Hex dump   Data   Comm
00459000
00459000
00459000
00459000
```

And we have successfully landed inside the serial check procedure algorithm



```
004571B7   .   E8 D8FDFFFF   CALL <Example_.Long/Short Error>
004571BC   .v  EB 6D         JMP SHORT Example_.0045722B
004571BE   >   8D55 EC       LEA EDX,DWORD PTR SS:[EBP-14]
004571C1   .   8BC6          MOV EAX,ESI
004571C3   .   E8 7412FBFF   CALL Example_.0040843C
004571C8   .   8B45 EC       MOV EAX,DWORD PTR SS:[EBP-14]
004571CB   .   8D4D FF       LEA ECX,DWORD PTR SS:[EBP-1]
004571CE   .   BA 01000000   MOV EDX,1
004571D3   .   E8 6CFDFFFF   CALL Example_.00456F44
004571D8   .   807D FF 31    CMP BYTE PTR SS:[EBP-1],31
004571DC   .v  74 07         JE SHORT Example_.004571E5
004571DE   .   E8 B1FDFFFF   CALL <Example_.Long/Short Error>
004571E3   .v  EB 46         JMP SHORT Example_.0045722B
004571E5   >   8D55 E8       LEA EDX,DWORD PTR SS:[EBP-18]
004571E8   .   8BC6          MOV EAX,ESI
004571EA   .   E8 4D12FBFF   CALL Example_.0040843C
004571EF   .   8B45 E8       MOV EAX,DWORD PTR SS:[EBP-18]
004571F2   .   8D4D FF       LEA ECX,DWORD PTR SS:[EBP-1]
004571F5   .   BA 02000000   MOV EDX,2
004571FA   .   E8 45FDFFFF   CALL Example_.00456F44
004571FF   .   8D45 E4       LEA EAX,DWORD PTR SS:[EBP-1C]
00457202   .   0FB655 FF     MOVZX EDX,BYTE PTR SS:[EBP-1]
00457206   .   E8 25D6FAFF   CALL Example_.00404830
0045720B   .   8B55 E4       MOV EDX,DWORD PTR SS:[EBP-1C]
0045720E   .   8B83 6803000  MOV EAX,DWORD PTR DS:[EBX+368]
00457214   .   E8 CB19FEFF   CALL Example_.00438BE4
00457219   .   807D FF 34    CMP BYTE PTR SS:[EBP-1],34
0045721D   .v  74 07         JE SHORT Example_.00457226
```

Suggestions (Approach No1)

In order to avoid tracing sensitive program functions through looking up string references, a programmer could follow the steps:

- Store strings in global variables or better inside arrays and then reference to them when needed.
  Pseudo Code Example:

```
array[] myMsges = {'The Serial number you entered is too short or too long',
                   'The Serial number you entered is not valid',
                   'Thank You for registering.'}

//Code omitted

function registrationCheck():

if(invalid_length) then
sendMessage(myMsges[0])

if(invalid_serial) then
sendMessage(myMsges[1])

if(valid_serial) then
sendMessage(myMsges[2])
```

  Additionally, the programmer could encrypt the strings inside the array and decrypt them when they are needed (there is no need for an advanced encryption, just a simple algorithm)

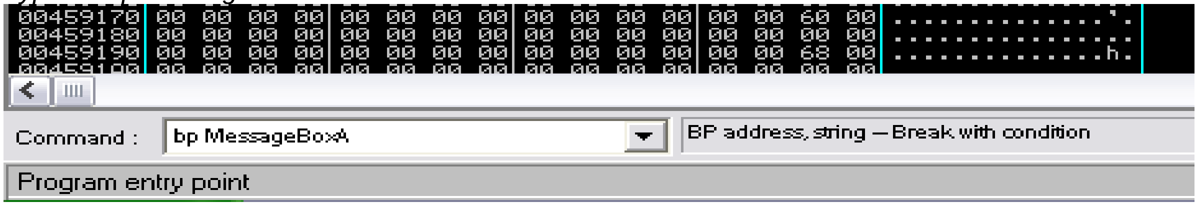
```
//This can be done separately.
//Let’s assume that the result of this code will be: ‘dkg$2 kF2 gkfoaplk’

string thank_you = ‘Thank You for registering’

for(each letter in thank_you) do
add_5_to_ascii_value(letter)
print thank_you


//program serial check
If(valid_serial) then
sendMessage(decrypt(‘dkg$2 kF2 gkfoaplk’))
```

- Store strings inside a file or registry

Approach No2 (Breakpoint on windows API)

In this approach we will make use of a breakpoint on MessageBoxA API. Some programs might use MessageBoxW, MessageBoxExA or MessageBoxExW.

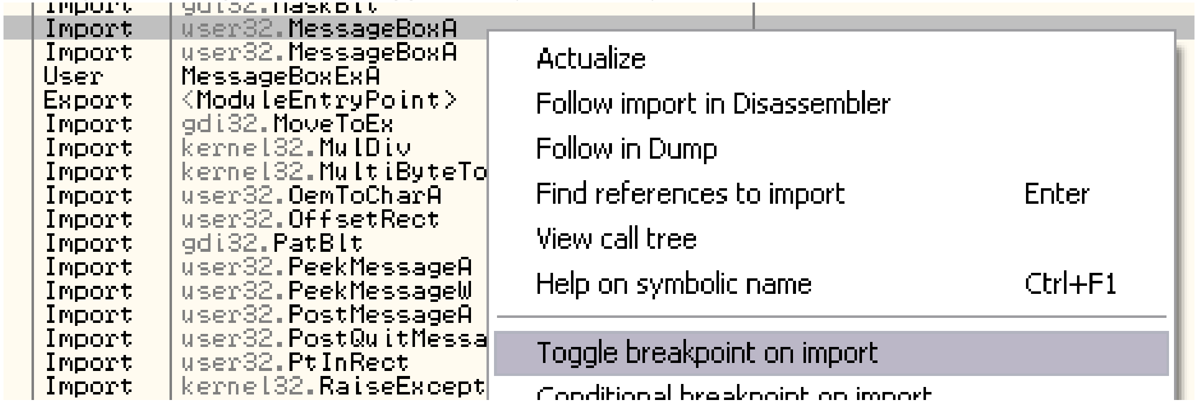
Step 1:
(Using Ollydbg’s Command Bar plug-in)
Type in “bp MessageBoxA” and then Hit enter



```
00459170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 00 ..............`.
00459180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00459190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 68 00 ..............h.
```

Command : bp MessageBoxA
BP address, string — Break with condition
Program entry point

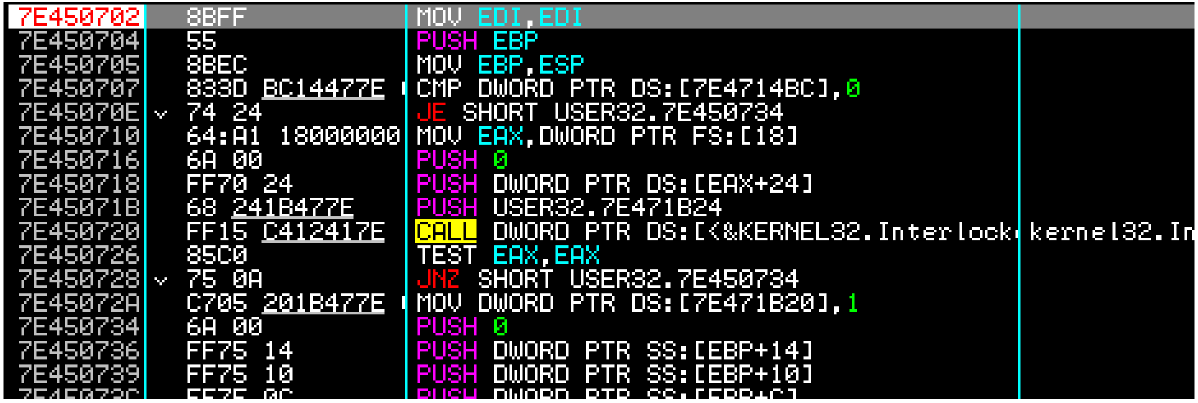
(Using Ollydbg’s Names window)
Press Alt+E to switch to “Executable Modules” list > Select your executable and click Ctrl+N > Find MessageBoxA > Right Click > “Toggle breakpoint on import”



```
Import  user32.MessageBoxA
Import  user32.MessageBoxA
User    MessageBoxExA
Export  <ModuleEntryPoint>
Import  gdi32.MoveToEx
Import  kernel32.MulDiv
Import  kernel32.MultiByteTo
Import  user32.OemToCharA
Import  user32.OffsetRect
Import  gdi32.PatBlt
Import  user32.PeekMessageA
Import  user32.PeekMessageW
Import  user32.PostMessageA
Import  user32.PostQuitMessa
Import  user32.PtInRect
Import  kernel32.RaiseExcept
```

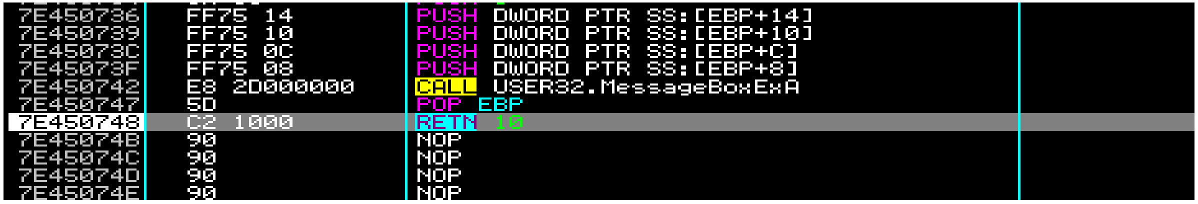
Actualize
Follow import in Disassembler
Follow in Dump
Find references to import Enter
View call tree
Help on symbolic name Ctrl+F1
Toggle breakpoint on import
Conditional breakpoint on import

Step 2:
Run the program > Insert random data > Press Check
Now you break at MessageBoxA API inside User32



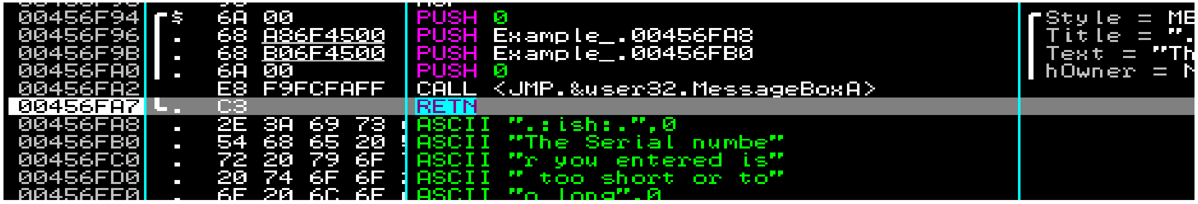
```
7E450702    8BFF             MOV EDI,EDI
7E450704    55               PUSH EBP
7E450705    8BEC             MOV EBP,ESP
7E450707    833D BC14477E    CMP DWORD PTR DS:[7E4714BC],0
7E45070E  v 74 24            JE SHORT USER32.7E450734
7E450710    64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7E450716    6A 00            PUSH 0
7E450718    FF70 24          PUSH DWORD PTR DS:[EAX+24]
7E45071B    68 241B477E      PUSH USER32.7E471B24
7E450720    FF15 C412417E    CALL DWORD PTR DS:[<&KERNEL32.Interlock  kernel32.In
7E450726    85C0             TEST EAX,EAX
7E450728  v 75 0A            JNZ SHORT USER32.7E450734
7E45072A    C705 201B477E    MOV DWORD PTR DS:[7E471B20],1
7E450734    6A 00            PUSH 0
7E450736    FF75 14          PUSH DWORD PTR SS:[EBP+14]
7E450739    FF75 10          PUSH DWORD PTR SS:[EBP+10]
```

Step 3:
Execute program until return (Ctrl+F9 or F8 until the end)



```
7E450736    FF75 14          PUSH DWORD PTR SS:[EBP+14]
7E450739    FF75 10          PUSH DWORD PTR SS:[EBP+10]
7E45073C    FF75 0C          PUSH DWORD PTR SS:[EBP+C]
7E45073F    FF75 08          PUSH DWORD PTR SS:[EBP+8]
7E450742    E8 2D000000      CALL USER32.MessageBoxExA
7E450747    5D               POP EBP
7E450748    C2 1000          RETN 10
7E45074B    90               NOP
7E45074C    90               NOP
7E45074D    90               NOP
7E45074E    90               NOP
```

Step 4:
Step outside the function (F8)



```
00456F94  ┌$  6A 00          PUSH 0                              ┌Style = ME
00456F96  │.  68 A86F4500    PUSH Example_.00456FA8              │Title = ".
00456F9B  │.  68 B06F4500    PUSH Example_.00456FB0              │Text = "Th
00456FA0  │.  6A 00          PUSH 0                              │hOwner = N
00456FA2      E8 F9FCFAFF    CALL <JMP.&user32.MessageBoxA>
00456FA7  └.  C3             RETN
00456FA8   .  2E 3A 69 73    ASCII ".:ish:.",0
00456FB0   .  54 68 65 20    ASCII "The Serial numbe"
00456FC0   .  72 20 79 6F    ASCII "r you entered is"
00456FD0   .  20 74 6F 6F    ASCII " too short or to"
00456FE0   .  6F 20 6C 6F    ASCII "o long",0
```

As you can see we ended up in the same place we did in *Approach No1, Step 3.*

Suggestions (Approach No2)

In order to avoid tracing your program through setting breakpoints using API breakpoints a programmer should limit their uses. Code your programs with the minimum of API calls; create your own message boxes instead of using API's.

Approach No3 (Stack Tracing)

Another interesting approach a reverse engineer can use is “stack tracing”. Stack tracing, is the technique of tracing back your steps through the stack.
When the “CALL <procedure>” instruction is executed by the CPU, the value of the Instruction Pointer (EIP), plus the number of bytes until the next instruction, is pushed inside the stack. When the procedure finishes and the “RETN” instruction is reached the processor pops the value from the stack and returns to the previews function.

Let’s assume:

Offset	Opcode
1	PUSH 0
2	CALL 0xF
3	TEST EAX,EAX

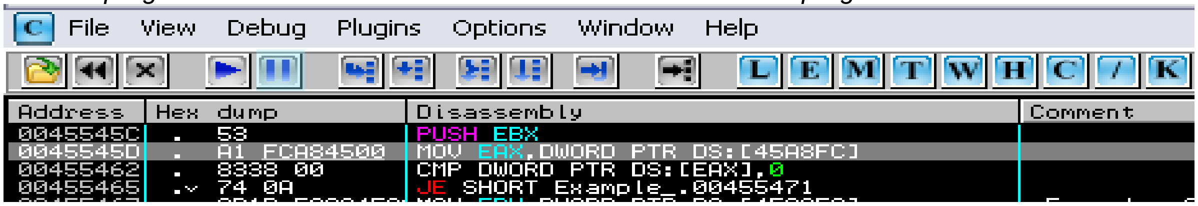
When “*CALL 0xF*” runs, the value of offset 3 is pushed inside the stack

Offset	Opcode
F	MOV EAX,1
10	RETN
11	NOP

When RETN runs the value of offset 3 POPs from the stack and placed into EIP

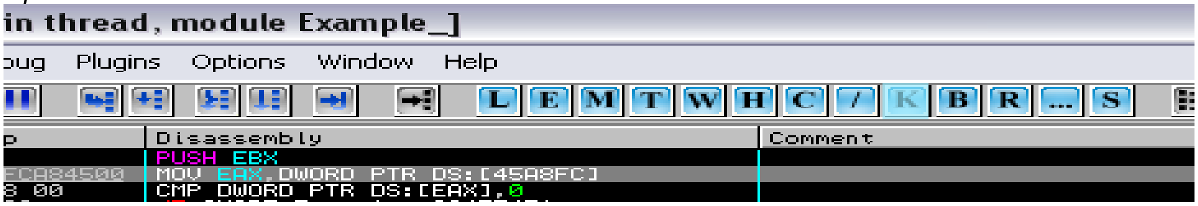
Step 1:

Run the program > Enter random data > Click Confirm > Pause the program



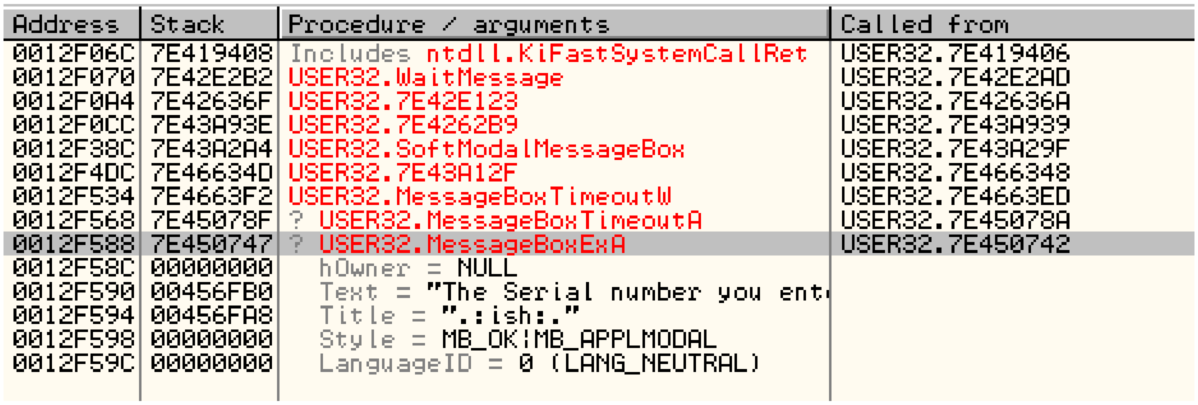
```
Address   Hex dump        Disassembly                          Comment
0045545C  .   53          PUSH EBX
0045545D  .   A1 FCA84500 MOV EAX,DWORD PTR DS:[45A8FC]
00455462  .   8338 00     CMP DWORD PTR DS:[EAX],0
00455465  .v  74 0A       JE SHORT Example_.00455471
```

Step 2:
Open the “Call stack” window



```
p           Disassembly                      Comment
            PUSH EBX
FCA84500    MOV EAX,DWORD PTR DS:[45A8FC]
8 00        CMP DWORD PTR DS:[EAX],0
```

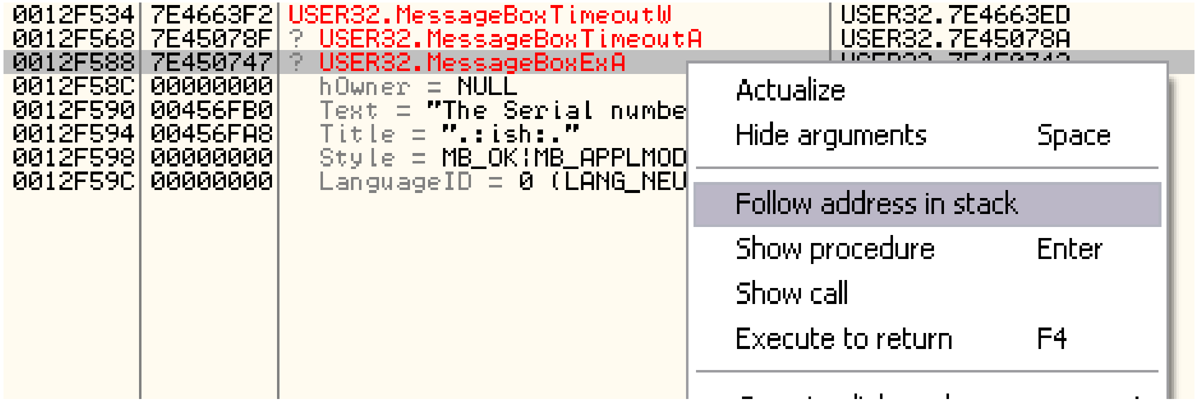
There are a number of functions calling each other. The function we can use to trace into the main program registration routine is MessageBoxExA but that is not efficient. We need to see what calls that function.



```
Address  Stack    Procedure / arguments                     Called from
0012F06C 7E419408 Includes ntdll.KiFastSystemCallRet       USER32.7E419406
0012F070 7E42E2B2 USER32.WaitMessage                        USER32.7E42E2AD
0012F0A4 7E42636F USER32.7E42E123                           USER32.7E42636A
0012F0CC 7E43A93E USER32.7E4262B9                           USER32.7E43A939
0012F38C 7E43A2A4 USER32.SoftModalMessageBox                USER32.7E43A29F
0012F4DC 7E46634D USER32.7E43A12F                           USER32.7E466348
0012F534 7E4663F2 USER32.MessageBoxTimeoutW                 USER32.7E4663ED
0012F568 7E45078F ? USER32.MessageBoxTimeoutA               USER32.7E45078A
0012F588 7E450747 ? USER32.MessageBoxExA                    USER32.7E450742
0012F58C 00000000   hOwner = NULL
0012F590 00456FB0   Text = "The Serial number you ent
0012F594 00456FA8   Title = ".:ish:."
0012F598 00000000   Style = MB_OK|MB_APPLMODAL
0012F59C 00000000   LanguageID = 0 (LANG_NEUTRAL)
```

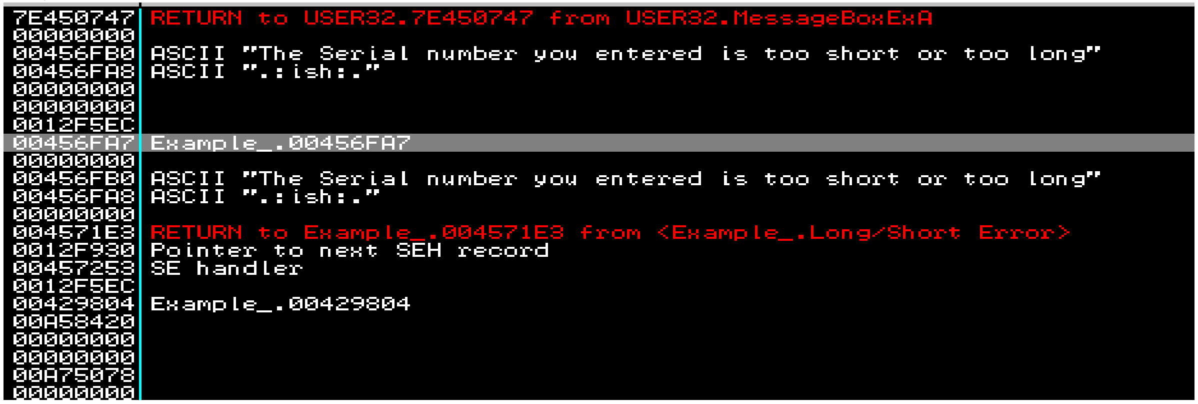
Step 3:

Right Click > Follow address in stack



```
0012F534 7E4663F2 USER32.MessageBoxTimeoutW                 USER32.7E4663ED
0012F568 7E45078F ? USER32.MessageBoxTimeoutA               USER32.7E45078A
0012F588 7E450747 ? USER32.MessageBoxExA
0012F58C 00000000   hOwner = NULL
0012F590 00456FB0   Text = "The Serial numbe
0012F594 00456FA8   Title = ".:ish:."
0012F598 00000000   Style = MB_OK|MB_APPLMOD
0012F59C 00000000   LanguageID = 0 (LANG_NEU
```

Step 4:



```
7E450747 RETURN to USER32.7E450747 from USER32.MessageBoxExA
00000000
00456FB0 ASCII "The Serial number you entered is too short or too long"
00456FA8 ASCII ".:ish:."
00000000
00000000
0012F5EC
00456FA7 Example_.00456FA7
00000000
00456FB0 ASCII "The Serial number you entered is too short or too long"
00456FA8 ASCII ".:ish:."
00000000
004571E3 RETURN to Example_.004571E3 from <Example_.Long/Short Error>
0012F930 Pointer to next SEH record
00457253 SE handler
0012F5EC
00429804 Example_.00429804
00A58420
00000000
00000000
00A75078
00000000
```

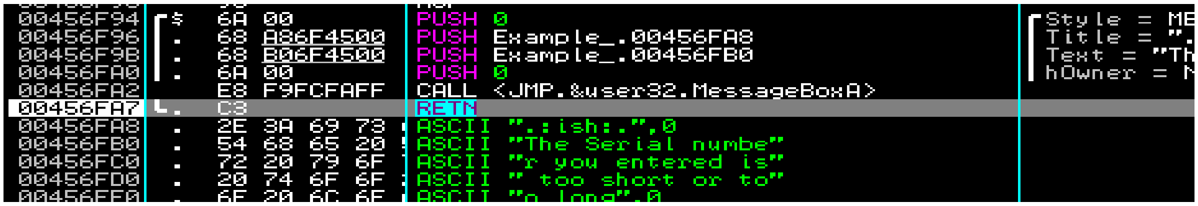
RETURN to USER32.7E450747 from USER32.MessageBoxExA

It is obvious that USER32.7E450747 is MessageBoxA

(If you don't know why, look at the code inside user32.dll)

Therefore the function we look for is located at:

Example_.00456FA7 (highlighted above)



```
00456F94  ┌$  6A 00        PUSH 0                                 ┌Style = ME
00456F96  │.  68 A86F4500  PUSH Example_.00456FA8                 │Title = ".
00456F9B  │.  68 B06F4500  PUSH Example_.00456FB0                 │Text = "Th
00456FA0  │.  6A 00        PUSH 0                                 │hOwner = N
00456FA2  │.  E8 F9FCFAFF  CALL <JMP.&user32.MessageBoxA>
00456FA7  └.  C3           RETN
00456FA8   .  2E 3A 69 73  ASCII ".:ish:.",0
00456FB0   .  54 68 65 20  ASCII "The Serial numbe"
00456FC0   .  72 20 79 6F  ASCII "r you entered is"
00456FD0   .  20 74 6F 6F  ASCII " too short or to"
00456FE0   .  6E 20 6C 6F  ASCII "o long",0
```

Suggestions (Approach No3)

Avoiding stack tracing is a hard technique. One might argue that we could do so by replacing all the sensitive procedure “CALL” and “RETN” instructions inside your program with “JMP”. This is called “Binary Code Obfuscation”.

Code Obfuscation is the technique of transforming the original program binary code thus rendering it unreadable and harder to analyze by static disassembly. Although this confuses reverse engineers, it doesn’t protect the software; it only delays the code analysis.

The basic idea behind CO is to combine Data and Code sections. Additionally, obfuscation replaces the following OPCODES in order to avoid disassembly and stack tracing:

- Replace of CALL with PUSH, POP, RET and JMP. And replace JMP with PUSH and RET. For example:

Original Code:	Obfuscated Code:
PUSH 0 *CALL* 7E450747	*PUSH 0* PUSH EIP + <bytes to next instruction> *JMP 7E450747*

Original Code:	Obfuscated Code:
MOV EBX,1 *RETN*	*POP EAX* *JMP EAX*

Original Code:	Obfuscated Code:
JMP 00456F94	*PUSH 00456F94* *RETN*

- Replace JMP branches with conditional branches (e.g.: JE, JNZ, JL) that are always satisfied. Additionally, this way you can confuse reversers and lead them to a junk code section.

Original Code:	Obfuscated Code:
JMP 00456F94	*MOV EAX, 1* *CMP EAX, 0* *JE <JUNK_CODE>* *JNE 00456F94*

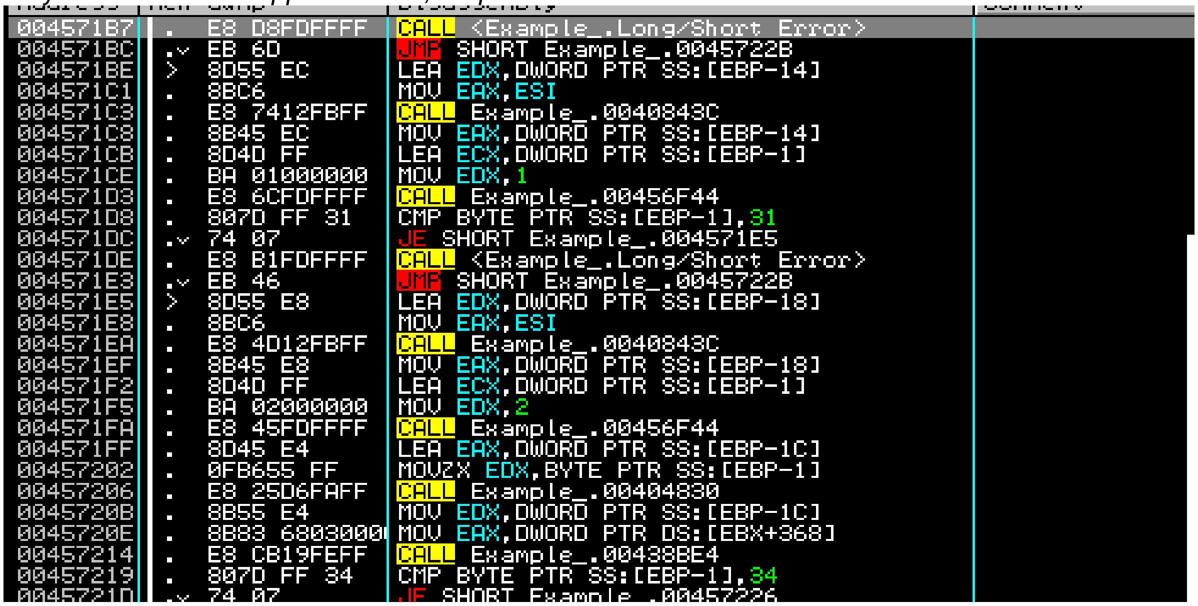
- Add partial instructions at unreachable areas.

- Avoid using direct references to offsets (e.g.: JMP 00456F94). Use simple calculations to obfuscate that offset and then call it. For example:

  MOV EAX, 00456000 ; *EAX =* *00456000*
  ADD EAX, 00000F94 ; *EAX =* 00456F94
  JMP EAX ; *JMP* 00456F94

Binary Code Patching:

As you can see from *Approach No1, Step 6:*

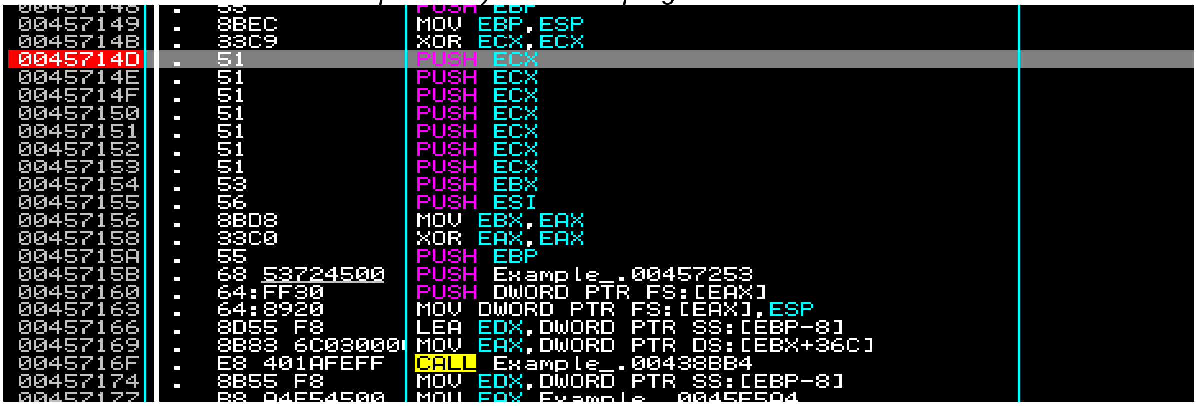


```
004571B7   .  E8 D8FDFFFF   CALL <Example_.Long/Short Error>
004571BC   .v EB 6D         JMP SHORT Example_.0045722B
004571BE   >  8D55 EC       LEA EDX,DWORD PTR SS:[EBP-14]
004571C1   .  8BC6          MOV EAX,ESI
004571C3   .  E8 7412FBFF   CALL Example_.0040843C
004571C8   .  8B45 EC       MOV EAX,DWORD PTR SS:[EBP-14]
004571CB   .  8D4D FF       LEA ECX,DWORD PTR SS:[EBP-1]
004571CE   .  BA 01000000   MOV EDX,1
004571D3   .  E8 6CFDFFFF   CALL Example_.00456F44
004571D8   .  807D FF 31    CMP BYTE PTR SS:[EBP-1],31
004571DC   .v 74 07         JE SHORT Example_.004571E5
004571DE   .  E8 B1FDFFFF   CALL <Example_.Long/Short Error>
004571E3   .v EB 46         JMP SHORT Example_.0045722B
004571E5   >  8D55 E8       LEA EDX,DWORD PTR SS:[EBP-18]
004571E8   .  8BC6          MOV EAX,ESI
004571EA   .  E8 4D12FBFF   CALL Example_.0040843C
004571EF   .  8B45 E8       MOV EAX,DWORD PTR SS:[EBP-18]
004571F2   .  8D4D FF       LEA ECX,DWORD PTR SS:[EBP-1]
004571F5   .  BA 02000000   MOV EDX,2
004571FA   .  E8 45FDFFFF   CALL Example_.00456F44
004571FF   .  8D45 E4       LEA EAX,DWORD PTR SS:[EBP-1C]
00457202   .  0FB655 FF     MOVZX EDX,BYTE PTR SS:[EBP-1]
00457206   .  E8 25D6FAFF   CALL Example_.00404830
0045720B   .  8B55 E4       MOV EDX,DWORD PTR SS:[EBP-1C]
0045720E   .  8B83 6803000  MOV EAX,DWORD PTR DS:[EBX+368]
00457214   .  E8 CB19FEFF   CALL Example_.00438BE4
00457219   .  807D FF 34    CMP BYTE PTR SS:[EBP-1],34
0045721D   .v 74 07         JE SHORT Example_.00457226
```

This is the actual algorithm that determines whenever the serial code inserted is valid or not and informs the user of his “mistake” to properly validate his registration.

There is a number of ways reversers use in order to successfully patch the code and control its flow. Before we do that, we have to analyze the actual code and understand where our actual goal lies at.

Step 1:
Scroll up and set a breakpoint near or at the function start (To set a breakpoint select the instruction you would like to break on and then press F2) > Run the program

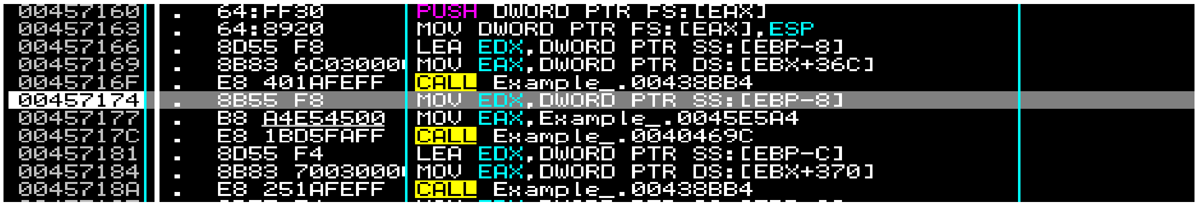


```
00457148  .  55           PUSH EBP
00457149  .  8BEC         MOV EBP,ESP
0045714B  .  33C9         XOR ECX,ECX
0045714D  .  51           PUSH ECX
0045714E  .  51           PUSH ECX
0045714F  .  51           PUSH ECX
00457150  .  51           PUSH ECX
00457151  .  51           PUSH ECX
00457152  .  51           PUSH ECX
00457153  .  51           PUSH ECX
00457154  .  53           PUSH EBX
00457155  .  56           PUSH ESI
00457156  .  8BD8         MOV EBX,EAX
00457158  .  33C0         XOR EAX,EAX
0045715A  .  55           PUSH EBP
0045715B  .  68 53724500  PUSH Example_.00457253
00457160  .  64:FF30      PUSH DWORD PTR FS:[EAX]
00457163  .  64:8920      MOV DWORD PTR FS:[EAX],ESP
00457166  .  8D55 F8      LEA EDX,DWORD PTR SS:[EBP-8]
00457169  .  8B83 6C03000 MOV EAX,DWORD PTR DS:[EBX+36C]
0045716F  .  E8 401AFEFF  CALL Example_.00438BB4
00457174  .  8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
00457177  .  B8 A4E54500  MOV EAX,Example_.0045E5A4
```

Step 2:
Step each instruction and try to understand what this code is for.

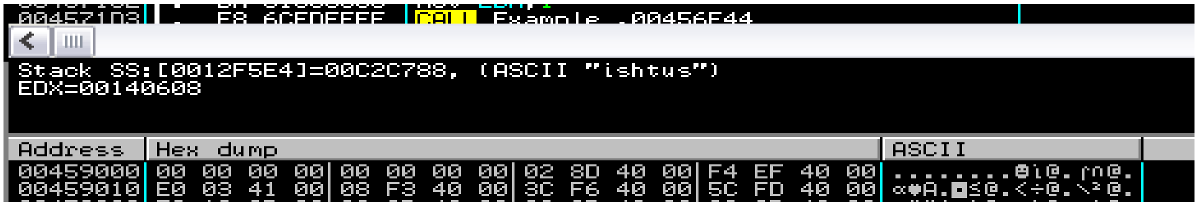
As you can see in the images below, the CALL instruction at offset 0045716F returns the pointer of the string given by the user inside the "Username:" text box.

Code Section:



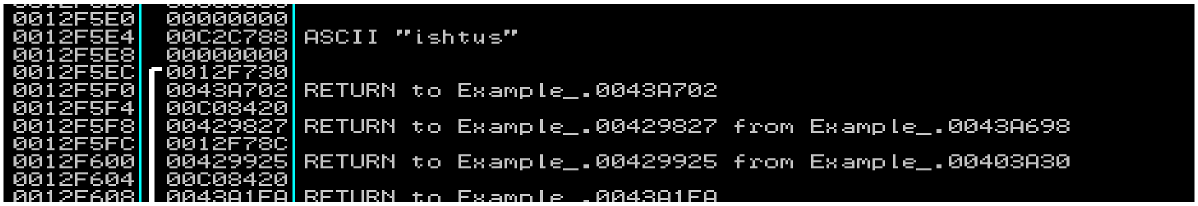
```
00457160  .  64:FF30      PUSH DWORD PTR FS:[EAX]
00457163  .  64:8920      MOV DWORD PTR FS:[EAX],ESP
00457166  .  8D55 F8      LEA EDX,DWORD PTR SS:[EBP-8]
00457169  .  8B83 6C03000 MOV EAX,DWORD PTR DS:[EBX+36C]
0045716F  .  E8 401AFEFF  CALL Example_.00438BB4
00457174  .  8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
00457177  .  B8 A4E54500  MOV EAX,Example_.0045E5A4
0045717C  .  E8 1BD5FAFF  CALL Example_.0040469C
00457181  .  8D55 F4      LEA EDX,DWORD PTR SS:[EBP-C]
00457184  .  8B83 7003000 MOV EAX,DWORD PTR DS:[EBX+370]
0045718A  .  E8 251AFEFF  CALL Example_.00438BB4
```

Current Instruction:



```
00457183  .  E8 6CFDFFFF  CALL Example_.00456F44

Stack SS:[0012F5E4]=00C2C788, (ASCII "ishtus")
EDX=00140608

Address   Hex dump                                         ASCII
00459000  00 00 00 00|00 00 00 00|02 8D 40 00|F4 EF 40 00  ........☻ì@.ôï@.
00459010  E0 03 41 00|08 F3 40 00|3C F6 40 00|5C FD 40 00  à♥A.◘ó@.<ö@.\ý@.
```

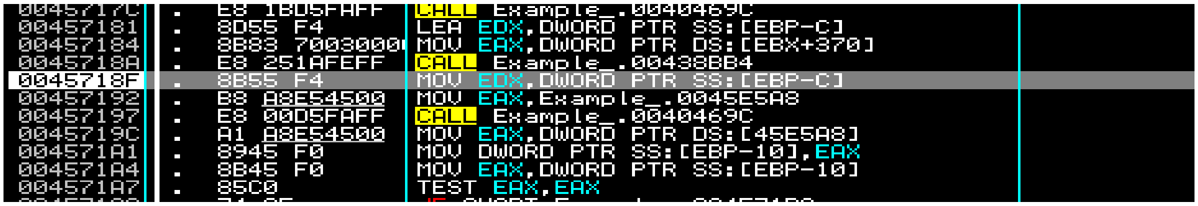
Stack:



```
0012F5E0   00000000
0012F5E4   00C2C788  ASCII "ishtus"
0012F5E8   00000000
0012F5EC  ┌0012F730
0012F5F0   0043A702  RETURN to Example_.0043A702
0012F5F4   00C08420
0012F5F8   00429827  RETURN to Example_.00429827 from Example_.0043A698
0012F5FC   0012F78C
0012F600   00429925  RETURN to Example_.00429925 from Example_.00403A30
0012F604   00C08420
0012F608   0043A1FA  RETURN to Example_.0043A1FA
```

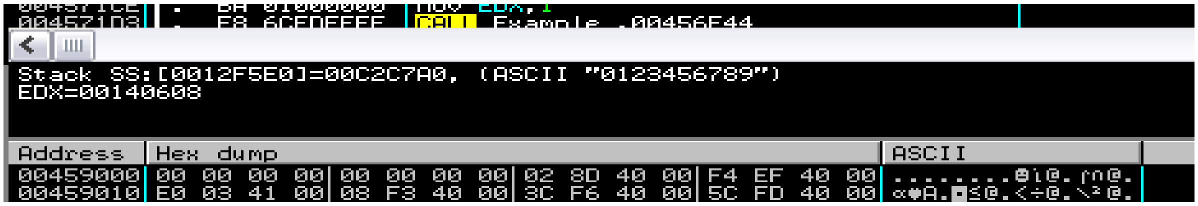
The following CALL instruction at offset 0045718F returns the pointer of the string given by the user inside the “Serial:” text box.

Code Section:



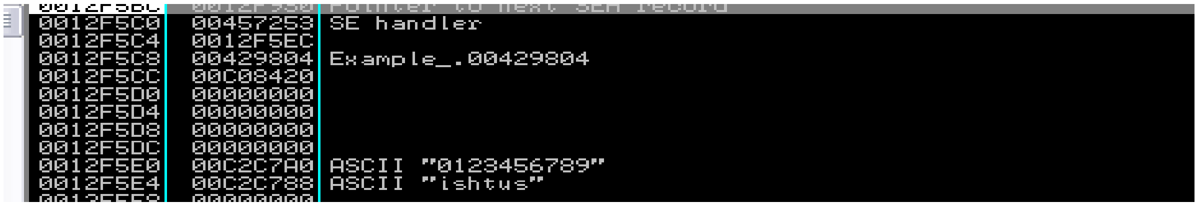
```
0045717C  .  E8 1BD5FAFF   CALL Example_.0040469C
00457181  .  8D55 F4       LEA EDX,DWORD PTR SS:[EBP-C]
00457184  .  8B83 7003000  MOV EAX,DWORD PTR DS:[EBX+370]
0045718A  .  E8 251AFEFF   CALL Example_.00438BB4
0045718F  .  8B55 F4       MOV EDX,DWORD PTR SS:[EBP-C]
00457192  .  B8 A8E54500   MOV EAX,Example_.0045E5A8
00457197  .  E8 00D5FAFF   CALL Example_.0040469C
0045719C  .  A1 A8E54500   MOV EAX,DWORD PTR DS:[45E5A8]
004571A1  .  8945 F0       MOV DWORD PTR SS:[EBP-10],EAX
004571A4  .  8B45 F0       MOV EAX,DWORD PTR SS:[EBP-10]
004571A7  .  85C0          TEST EAX,EAX
```

Current Instruction:



```
004571CE  .  BA 01000000   MOV EDX,1
004571D3  .  E8 6CFDFFFF   CALL Example_.00456F44

Stack SS:[0012F5E0]=00C2C7A0, (ASCII "0123456789")
EDX=00140608

Address   Hex dump                                         ASCII
00459000  00 00 00 00|00 00 00 00|02 8D 40 00|F4 EF 40 00  ........☻ì@.ôï@.
00459010  E0 03 41 00|08 F3 40 00|3C F6 40 00|5C FD 40 00  à♥A.◘ó@.<ö@.\ý@.
```

Stack:

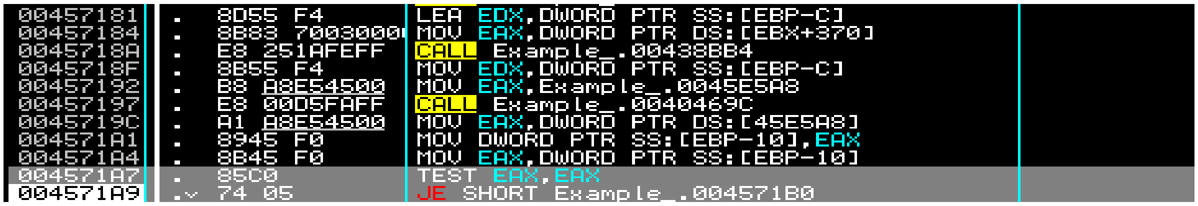


```
0012F5BC   0012F950  Pointer to next SEH record
0012F5C0   00457253  SE handler
0012F5C4   0012F5EC
0012F5C8   00429804  Example_.00429804
0012F5CC   00C08420
0012F5D0   00000000
0012F5D4   00000000
0012F5D8   00000000
0012F5DC   00000000
0012F5E0   00C2C7A0  ASCII "0123456789"
0012F5E4   00C2C788  ASCII "ishtus"
```

The following code loads the serial number given by the user into EAX, then checks if it is equal to null.

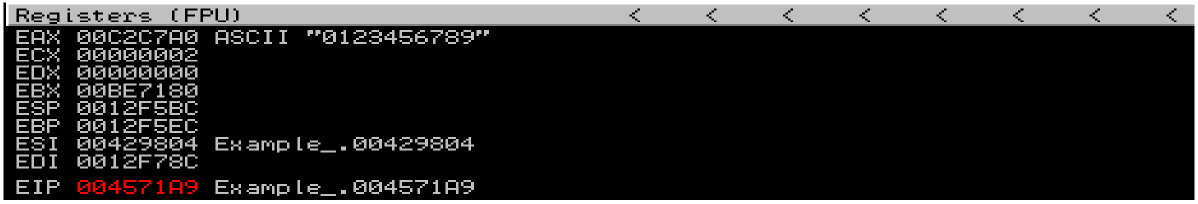
Apparently the value pointed by 0045E5A8 (see offset 0045719C) is the given serial ASCII value which eventually is loaded into EAX at offset 004571A4.

```
If (EAX == null) {
 //do something
}
```



```
00457181  .  8D55 F4       LEA EDX,DWORD PTR SS:[EBP-C]
00457184  .  8B83 7003000  MOV EAX,DWORD PTR DS:[EBX+370]
0045718A  .  E8 251AFEFF   CALL Example_.00438BB4
0045718F  .  8B55 F4       MOV EDX,DWORD PTR SS:[EBP-C]
00457192  .  B8 A8E54500   MOV EAX,Example_.0045E5A8
00457197  .  E8 00D5FAFF   CALL Example_.0040469C
0045719C  .  A1 A8E54500   MOV EAX,DWORD PTR DS:[45E5A8]
004571A1  .  8945 F0       MOV DWORD PTR SS:[EBP-10],EAX
004571A4  .  8B45 F0       MOV EAX,DWORD PTR SS:[EBP-10]
004571A7  .  85C0          TEST EAX,EAX
004571A9  .v 74 05         JE SHORT Example_.004571B0
```

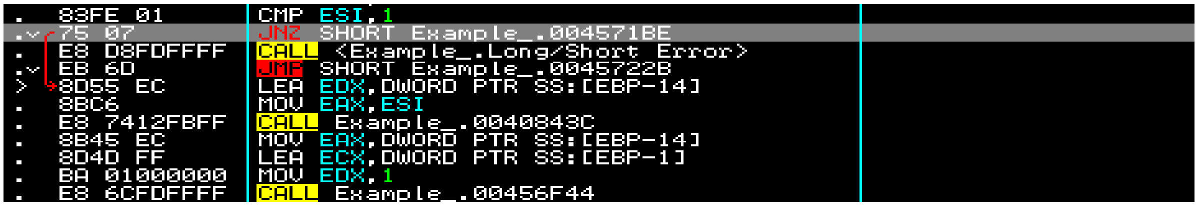
Registers Window:



```
Registers (FPU)
EAX 00C2C7A0 ASCII "0123456789"
ECX 00000002
EDX 00000000
EBX 00BE7180
ESP 0012F5BC
EBP 0012F5EC
ESI 00429804 Example_.00429804
EDI 0012F78C

EIP 004571A9 Example_.004571A9
```

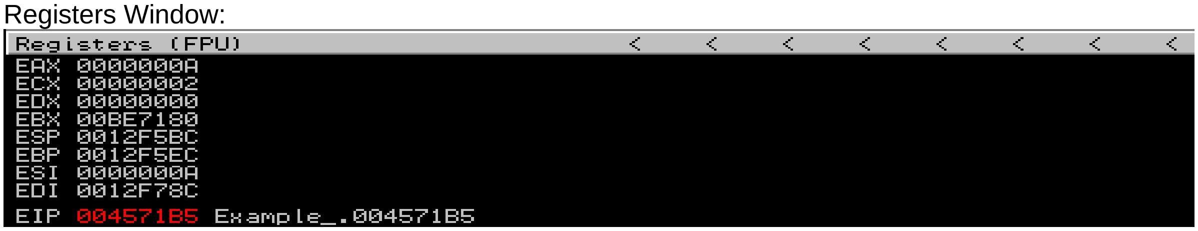
As you can see there is another length check. This time ESI holds the current length of our serial (in our case its 0xA Hexadecimal = 10 Decimal) which is compared with the hexadecimal number 0x01 which is equal to decimal 1. If the length of our serial is equal to one, then "Long/Short Error" is called (see Approach No1, Step 4)

Code Section:



```
.    83FE 01        CMP ESI,1
.v   75 07          JNZ SHORT Example_.004571BE
.    E8 D8FDFFFF    CALL <Example_.Long/Short Error>
.v   EB 6D          JMP SHORT Example_.0045722B
>    8D55 EC        LEA EDX,DWORD PTR SS:[EBP-14]
.    8BC6           MOV EAX,ESI
.    E8 7412FBFF    CALL Example_.0040843C
.    8B45 EC        MOV EAX,DWORD PTR SS:[EBP-14]
.    8D4D FF        LEA ECX,DWORD PTR SS:[EBP-1]
.    BA 01000000    MOV EDX,1
.    E8 6CFDFFFF    CALL Example_.00456F44
```

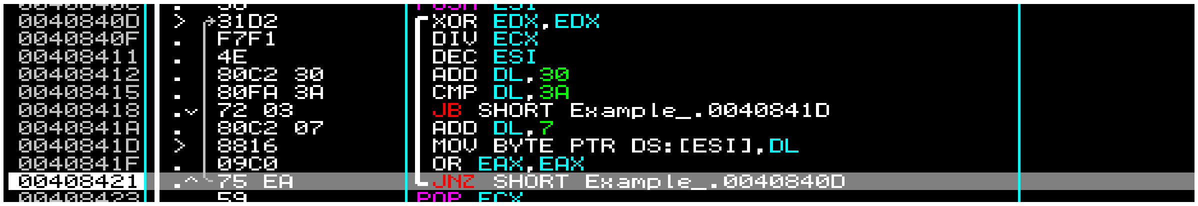
Registers Window:



```
Registers (FPU)
EAX 0000000A
ECX 00000002
EDX 00000000
EBX 00BE7180
ESP 0012F5BC
EBP 0012F5EC
ESI 0000000A
EDI 0012F78C

EIP 004571B5 Example_.004571B5
```

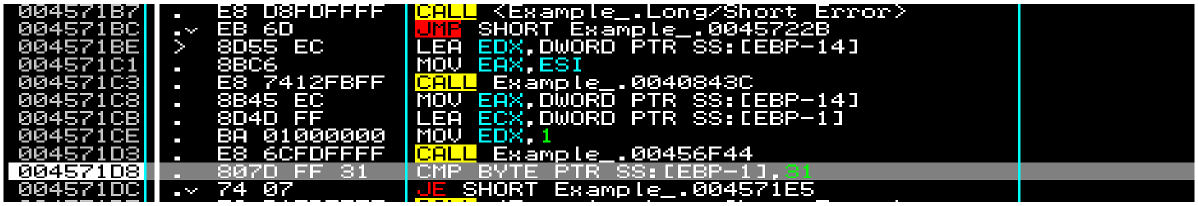
The following highlighted code compares the first string character from the serial decimal length (10) with the ASCII value 0x31, which is equal to "1". For those who are wondering how the length was converted into an ASCII string you can follow the call at the offset 004571C3 then have a look at the following loop:

```
0040840D  |> /31D2        /XOR EDX,EDX
0040840F  |.  |F7F1       |DIV ECX
00408411  |.  |4E         |DEC ESI
00408412  |.  |80C2 30    |ADD DL,30
00408415  |.  |80FA 3A    |CMP DL,3A
00408418  |.  |72 03      |JB SHORT Example_.0040841D
0040841A  |.  |80C2 07    |ADD DL,7
0040841D  |>|8816         |MOV BYTE PTR DS:[ESI],DL
0040841F  |.  |09C0       |OR EAX,EAX
00408421  |.^\75 EA       \JNZ SHORT Example_.0040840D
```



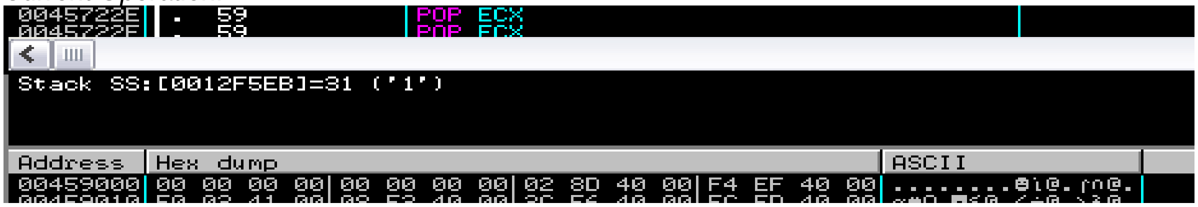
```
                                PUSH ESI
0040840D  >  31D2               XOR EDX,EDX
0040840F  .  F7F1               DIV ECX
00408411  .  4E                 DEC ESI
00408412  .  80C2 30            ADD DL,30
00408415  .  80FA 3A            CMP DL,3A
00408418  .v 72 03              JB SHORT Example_.0040841D
0040841A  .  80C2 07            ADD DL,7
0040841D  >  8816               MOV BYTE PTR DS:[ESI],DL
0040841F  .  09C0               OR EAX,EAX
00408421  .^ 75 EA              JNZ SHORT Example_.0040840D
00408423  .  59                 POP ECX
```

Code Section:



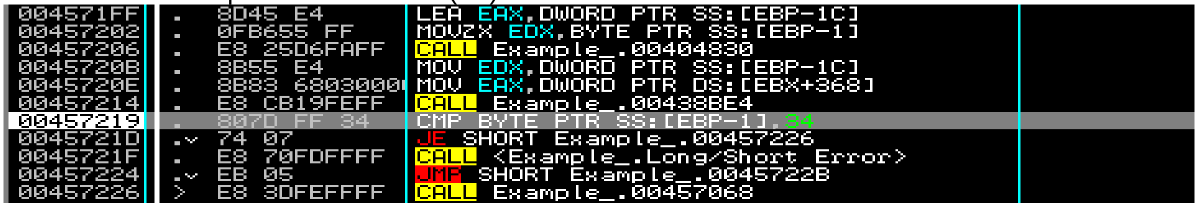
```
004571B7  .   E8 D8FDFFFF   CALL <Example_.Long/Short Error>
004571BC  .v  EB 6D         JMP SHORT Example_.0045722B
004571BE  >   8D55 EC       LEA EDX,DWORD PTR SS:[EBP-14]
004571C1  .   8BC6          MOV EAX,ESI
004571C3  .   E8 7412FBFF   CALL Example_.0040843C
004571C8  .   8B45 EC       MOV EAX,DWORD PTR SS:[EBP-14]
004571CB  .   8D4D FF       LEA ECX,DWORD PTR SS:[EBP-1]
004571CE  .   BA 01000000   MOV EDX,1
004571D3  .   E8 6CFDFFFF   CALL Example_.00456F44
004571D8  .   807D FF 31    CMP BYTE PTR SS:[EBP-1],31
004571DC  .v  74 07         JE SHORT Example_.004571E5
```

Current Operation:



```
0045722E  .   59            POP ECX
0045722F  .   59            POP ECX

Stack SS:[0012F5EB]=31 ('1')

Address   Hex dump                                          ASCII
00459000  00 00 00 00|00 00 00 00|02 8D 40 00|F4 EF 40 00  ........@ì@.ﾛï@.
```

The following does the same thing like above, but for the second number. In this case, the second number must be equal to 0x34 ASCII (“4”).



```
004571FF  .   8D45 E4       LEA EAX,DWORD PTR SS:[EBP-1C]
00457202  .   0FB655 FF     MOVZX EDX,BYTE PTR SS:[EBP-1]
00457206  .   E8 25D6FAFF   CALL Example_.00404830
0045720B  .   8B55 E4       MOV EDX,DWORD PTR SS:[EBP-1C]
0045720E  .   8B83 6803000  MOV EAX,DWORD PTR DS:[EBX+368]
00457214  .   E8 CB19FEFF   CALL Example_.00438BE4
00457219  .   807D FF 34    CMP BYTE PTR SS:[EBP-1],34
0045721D  .v  74 07         JE SHORT Example_.00457226
0045721F  .   E8 70FDFFFF   CALL <Example_.Long/Short Error>
00457224  .v  EB 05         JMP SHORT Example_.0045722B
00457226  >   E8 3DFEFFFF   CALL Example_.00457068
```

So the code of this program until now should look like this:

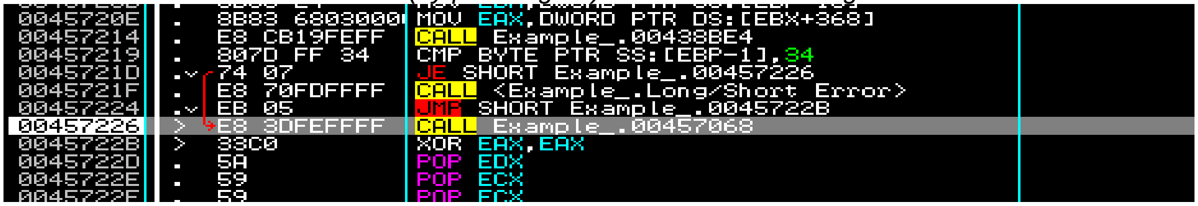
```
char first = getChar(length,1,?); //Get first character
if (first  != ‘1’) {
        char second = getChar(length,2,?); //Get second character
        if(second != ‘4’) {
                //continue with serial check
        }
        else {
                sendLongShortError();
        }

}
else {
        sendLongShortError();
}
```

Note: the character “?” shows an unknown value which most likely is the data type the returned value is stored in. (Delphi compiler).

Most likely, the serial number you inserted does not have the valid length of 14 characters. Therefore you can press F9 and type the serial again.

Step 3:

Follow the call at offset 00457226 (by pressing F7) as shown in the image below



```
0045720E  .  8B83 6803000 MOV EAX,DWORD PTR DS:[EBX+368]
00457214  .  E8 CB19FEFF  CALL Example_.00438BE4
00457219  .  807D FF 34   CMP BYTE PTR SS:[EBP-1],34
0045721D  .v 74 07        JE SHORT Example_.00457226
0045721F  .  E8 70FDFFFF  CALL <Example_.Long/Short Error>
00457224  .v EB 05        JMP SHORT Example_.0045722B
00457226  >  E8 3DFEFFFF  CALL Example_.00457068
0045722B  >  33C0         XOR EAX,EAX
0045722D  .  5A           POP EDX
0045722E  .  59           POP ECX
0045722F  .  59           POP ECX
```

Step 4:

Let's have a look at the code below

```
00457068  /$  53                  PUSH EBX
00457069  |.  56                  PUSH ESI
0045706A  |.  57                  PUSH EDI
0045706B  |.  55                  PUSH EBP
0045706C  |.  83C4 F8             ADD ESP,-8
0045706F  |.  BB 01000000         MOV EBX,1
00457074  |.  BE ACE54500         MOV ESI,Example_.0045E5AC
00457079  |>  8BCE                /MOV ECX,ESI
0045707B  |.  8BD3                |MOV EDX,EBX
0045707D  |.  A1 A8E54500         |MOV EAX,DWORD PTR DS:[45E5A8]
00457082  |.  E8 BDFEFFFF         |CALL Example_.00456F44
00457087  |.  43                  |INC EBX
00457088  |.  46                  |INC ESI
00457089  |.  83FB 0F             |CMP EBX,0F
0045708C  |.^ 75 EB               \JNZ SHORT Example_.00457079
0045708E  |.  A1 A4E54500         MOV EAX,DWORD PTR DS:[45E5A4]
00457093  |.  894424 04           MOV DWORD PTR SS:[ESP+4],EAX
00457097  |.  8B4424 04           MOV EAX,DWORD PTR SS:[ESP+4]
0045709B  |.  85C0                TEST EAX,EAX
0045709D  |.  74 05               JE SHORT Example_.004570A4
0045709F  |.  83E8 04             SUB EAX,4
004570A2  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
004570A4  |>  85C0                TEST EAX,EAX
004570A6  |.  7E 16               JLE SHORT Example_.004570BE
004570A8  |.  BB 01000000         MOV EBX,1
004570AD  |>  8B15 A4E54500       /MOV EDX,DWORD PTR DS:[45E5A4]
004570B3  |.  0FB6541A FF         |MOVZX EDX,BYTE PTR DS:[EDX+EBX-1]
004570B8  |.  03EA                |ADD EBP,EDX
004570BA  |.  43                  |INC EBX
004570BB  |.  48                  |DEC EAX
004570BC  |.^ 75 EF               \JNZ SHORT Example_.004570AD
004570BE  |>  BB 0E000000         MOV EBX,0E
004570C3  |.  B8 ACE54500         MOV EAX,Example_.0045E5AC
004570C8  |.  BA BCE54500         MOV EDX,Example_.0045E5BC
004570CD  |>  0FB608              /MOVZX ECX,BYTE PTR DS:[EAX]
004570D0  |.  890A                |MOV DWORD PTR DS:[EDX],ECX
004570D2  |.  83C2 04             |ADD EDX,4
004570D5  |.  40                  |INC EAX
004570D6  |.  4B                  |DEC EBX
004570D7  |.^ 75 F4               \JNZ SHORT Example_.004570CD
004570D9  |>  803D ACE54500>      /CMP BYTE PTR DS:[45E5AC],7B
004570E0  |.  74 07               |JE SHORT Example_.004570E9
004570E2  |.  BF 01000000         |MOV EDI,1
004570E7  |.  EB 46               |JMP SHORT Example_.0045712F
004570E9  |>  8BC5                |MOV EAX,EBP
```

```
004570EB  |.  B9 0A000000          |MOV ECX,0A
004570F0  |.  99                   |CDQ
004570F1  |.  F7F9                 |IDIV ECX
004570F3  |.  0FB605 ADE545>       |MOVZX EAX,BYTE PTR DS:[45E5AD]
004570FA  |.  3BD0                 |CMP EDX,EAX
004570FC  |.  75 06                |JNZ SHORT Example_.00457104
004570FE  |.  830424 02            |ADD DWORD PTR SS:[ESP],2
00457102  |.  EB 2B                |JMP SHORT Example_.0045712F
00457104  |>  BB 0C000000          |MOV EBX,0C
00457109  |.  BE ADE54500          |MOV ESI,Example_.0045E5AD
0045710E  |>  0FB606               |/MOVZX EAX,BYTE PTR DS:[ESI]
00457111  |.  B9 0A000000          ||MOV ECX,0A
00457116  |.  33D2                 ||XOR EDX,EDX
00457118  |.  F7F1                 ||DIV ECX
0045711A  |.  8BCA                 ||MOV ECX,EDX
0045711C  |.  83F9 0E              ||CMP ECX,0E
0045711F  |.  73 0A                ||JNB SHORT Example_.0045712B
00457121  |.  83F9 01              ||CMP ECX,1
00457124  |.  76 05                ||JBE SHORT Example_.0045712B
00457126  |.  E8 05FFFFFF          ||CALL <Example_.thank you>
0045712B  |>  46                   ||INC ESI
0045712C  |.  4B                   ||DEC EBX
0045712D  |.^ 75 DF                |\JNZ SHORT Example_.0045710E
0045712F  |>  83FF 01              |CMP EDI,1
00457132  |.^ 75 A5                \JNZ SHORT Example_.004570D9
00457134  |.  8B0424               MOV EAX,DWORD PTR SS:[ESP]
00457137  |.  83E8 02              SUB EAX,2
0045713A  |.  75 05                JNZ SHORT Example_.00457141
0045713C  |.  E8 A7FEFFFF          CALL <Example_.invalid number>
00457141  |>  59                   POP ECX
00457142  |.  5A                   POP EDX
00457143  |.  5D                   POP EBP
00457144  |.  5F                   POP EDI
00457145  |.  5E                   POP ESI
00457146  |.  5B                   POP EBX
00457147  \.  C3                   RETN
```

This code runs the actual serial checking. As you can see when you analyze the code while debugging, there is a number of jumps that lead you away from the desired call, which is located at offset 0045713C. Usually there are a number of approaches towards reaching your desired result. Those involve patching, analyzing, reconstructing or even ripping (the assembly) the code. In this software there are a limited number of approaches. As you can see, the above code only deals with checking the serial key and invoking the appropriate message to inform the user for his success or failure to validate his user/serial identity.

The following approaches might not apply in the real world, but they provide a basic and simple idea on how reversers work.

Approach No1 (Branch Patching)

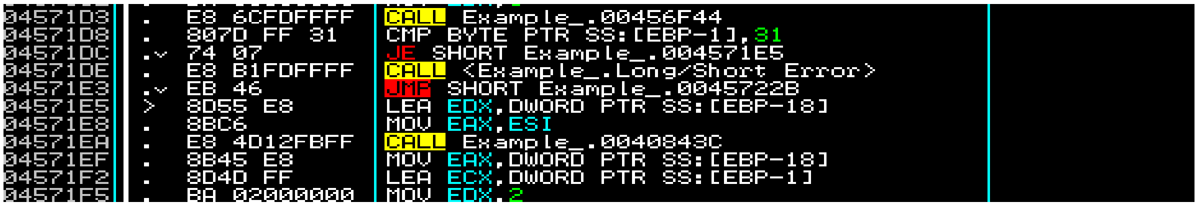
One way of patching the program flow is by modifying the conditional branches.

There are a number of places where the serial validation algorithm determines that the serial given by the user is invalid. Those are:

Check No1:

As shown in the binary analysis above, the function converts the serial length into a string ASCII data type then takes the first letter and compares it with the hex value 0x31 which is equal to ASCII character '1'

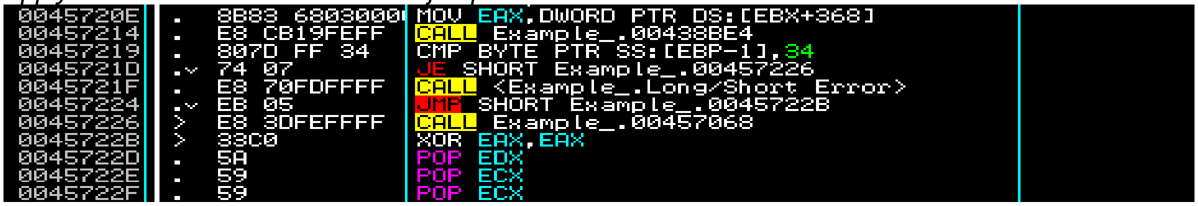
A simple patch can be placed by:
Double click on the opcode at offset 004571DC >
replace "JE SHORT 004571E5"
with "JMP SHORT 004571E5"
Therefore then the CALL at 004571DE is never called



```
04571D3  .   E8 6CFDFFFF   CALL Example_.00456F44
04571D8  .   807D FF 31    CMP BYTE PTR SS:[EBP-1],31
04571DC  .v  74 07         JE SHORT Example_.004571E5
04571DE  .   E8 B1FDFFFF   CALL <Example_.Long/Short Error>
04571E3  .v  EB 46         JMP SHORT Example_.0045722B
04571E5  >   8D55 E8       LEA EDX,DWORD PTR SS:[EBP-18]
04571E8  .   8BC6          MOV EAX,ESI
04571EA  .   E8 4D12FBFF   CALL Example_.0040843C
04571EF  .   8B45 E8       MOV EAX,DWORD PTR SS:[EBP-18]
04571F2  .   8D4D FF       LEA ECX,DWORD PTR SS:[EBP-1]
04571F5  .   BA 02000000   MOV EDX,2
```

Check No2:

Apply the same with the conditional jump at offset 0045721D

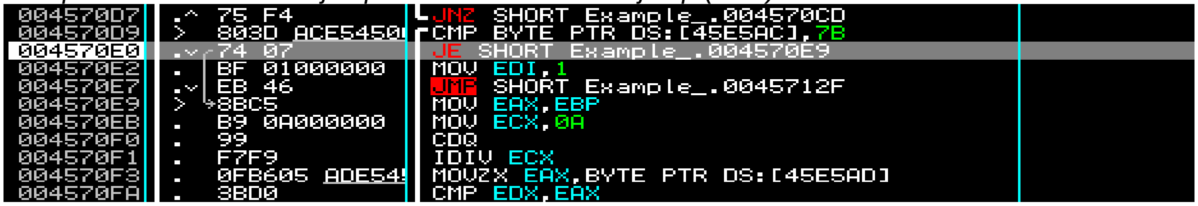


```
0045720E  .   8B83 6803000 MOV EAX,DWORD PTR DS:[EBX+368]
00457214  .   E8 CB19FEFF  CALL Example_.00438BE4
00457219  .   807D FF 34   CMP BYTE PTR SS:[EBP-1],34
0045721D  .v  74 07        JE SHORT Example_.00457226
0045721F  .   E8 70FDFFFF  CALL <Example_.Long/Short Error>
00457224  .v  EB 05        JMP SHORT Example_.0045722B
00457226  >   E8 3DFEFFFF  CALL Example_.00457068
0045722B  >   33C0         XOR EAX,EAX
0045722D  .   5A           POP EDX
0045722E  .   59           POP ECX
0045722F  .   59           POP ECX
```

Therefore, the code in Step2, Binary code patching changes into:

```
char first = getChar(length,1,?); //Get first character
if(true) {  //This is always true
        char second = getChar(length,2,?); //Get second character
        if(true) {  //This is always true
                //continue with serial check
        }
        else {
                sendLongShortError();      //This is never called
        }
}
else {
        sendLongShortError();      //This is never called
}
```

Check No3:
Also patch that conditional jump into an unconditional jump (JMP)

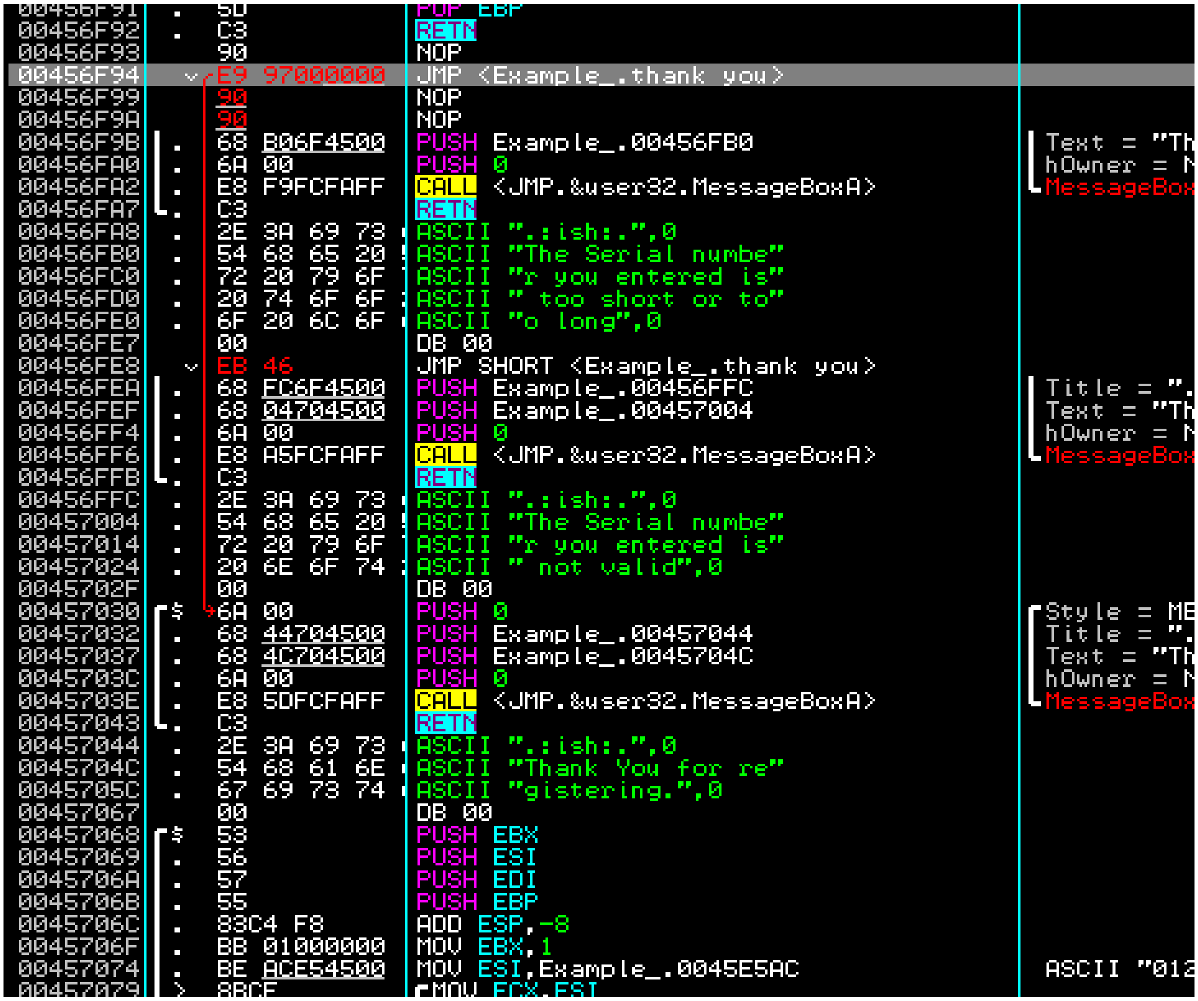


```
004570D7  .^ 75 F4            JNZ SHORT Example_.004570CD
004570D9  >  803D ACE54500   CMP BYTE PTR DS:[45E5AC],7B
004570E0  .v 74 07            JE SHORT Example_.004570E9
004570E2  .  BF 01000000      MOV EDI,1
004570E7  .v EB 46            JMP SHORT Example_.0045712F
004570E9  >  8BC5             MOV EAX,EBP
004570EB  .  B9 0A000000      MOV ECX,0A
004570F0  .  99               CDQ
004570F1  .  F7F9             IDIV ECX
004570F3  .  0FB605 ADE54!    MOVZX EAX,BYTE PTR DS:[45E5AD]
004570FA  .  3BD0             CMP EDX,EAX
```

In general, that should do it. Although there are a few bugs, I believe you understood the basic idea behind it.

Approach No2 (Replace functions)

A simpler approach is to alter the error message functions and point them at the success function. As shown below:



```
00456F91  .  5D               POP EBP
00456F92  .  C3               RETN
00456F93     90               NOP
00456F94   v E9 97000000      JMP <Example_.thank you>
00456F99     90               NOP
00456F9A     90               NOP
00456F9B  .  68 B06F4500      PUSH Example_.00456FB0             Text = "Th
00456FA0  .  6A 00            PUSH 0                             hOwner = N
00456FA2  .  E8 F9FCFAFF      CALL <JMP.&user32.MessageBoxA>     MessageBox
00456FA7  .  C3               RETN
00456FA8  .  2E 3A 69 73      ASCII ".:ish:.",0
00456FB0  .  54 68 65 20      ASCII "The Serial numbe"
00456FC0  .  72 20 79 6F      ASCII "r you entered is"
00456FD0  .  20 74 6F 6F      ASCII " too short or to"
00456FE0  .  6F 20 6C 6F      ASCII "o long",0
00456FE7     00               DB 00
00456FE8   v EB 46            JMP SHORT <Example_.thank you>
00456FEA  .  68 FC6F4500      PUSH Example_.00456FFC             Title = ".
00456FEF  .  68 04704500      PUSH Example_.00457004             Text = "Th
00456FF4  .  6A 00            PUSH 0                             hOwner = N
00456FF6  .  E8 A5FCFAFF      CALL <JMP.&user32.MessageBoxA>     MessageBox
00456FFB  .  C3               RETN
00456FFC  .  2E 3A 69 73      ASCII ".:ish:.",0
00457004  .  54 68 65 20      ASCII "The Serial numbe"
00457014  .  72 20 79 6F      ASCII "r you entered is"
00457024  .  20 6E 6F 74      ASCII " not valid",0
0045702F     00               DB 00
00457030  $  6A 00            PUSH 0                             Style = MB
00457032  .  68 44704500      PUSH Example_.00457044             Title = ".
00457037  .  68 4C704500      PUSH Example_.0045704C             Text = "Th
0045703C  .  6A 00            PUSH 0                             hOwner = N
0045703E  .  E8 5DFCFAFF      CALL <JMP.&user32.MessageBoxA>     MessageBox
00457043  .  C3               RETN
00457044  .  2E 3A 69 73      ASCII ".:ish:.",0
0045704C  .  54 68 61 6E      ASCII "Thank You for re"
0045705C  .  67 69 73 74      ASCII "gistering.",0
00457067     00               DB 00
00457068  $  53               PUSH EBX
00457069  .  56               PUSH ESI
0045706A  .  57               PUSH EDI
0045706B  .  55               PUSH EBP
0045706C  .  83C4 F8          ADD ESP,-8
0045706F  .  BB 01000000      MOV EBX,1
00457074  .  BE ACE54500      MOV ESI,Example_.0045E5AC          ASCII "012
00457079  >  8BCE             MOV ECX,ESI
```

Note: This will most likely not work if you, as a coder, are smart enough not to put everything inside one function.

Serial Generating

(known as keygening)

In this category, a “cracker” analyzes the program code and reconstructs the registration algorithm in such a way that instead of determining that the inserted serial is correct, it generates a correct serial key that will always be valid (without taking into consideration any external constrains). Some of the techniques used for constructing/reconstructing *Serial Generating* algorithms are:

Code Reconstructing

The careful analysis of an algorithm (usually by debugging) in order to understand the behavior of a function or set of functions in such a way that a reverser can transform the low-level assembly into a higher level programming language code (like C, C++, or as high as .NET and Java)

For example:

Low-level:

```
004570AD  |> /MOV EDX,DWORD PTR DS:[45E5A4]          ;  Load username string in EDX
004570B3  |   |MOVZX EDX,BYTE PTR DS:[EDX+EBX-1]     ;  Get letter in position EBX-1 (in each loop the pointer is incr by 1)
004570B8  |.  |ADD EBP,EDX                           ;  Add the hexadecimal ASCII value of the letter in EBP (UserCount)
004570BA  |. |INC EBX                                ;  Increase the pointer (EBX)
004570BB  |. |DEC EAX                                ;  Decrease the loop counter
004570BC  |.^ \JNZ SHORT Example_.004570AD           ;  Stop branching only when the loop counter reaches zero(0)
004570BE  |>  MOV EBX,0E
004570C3  |.   MOV EAX,Example_.0045E5AC
004570C8  |.   MOV EDX,Example_.0045E5BC
004570CD  |> /MOVZX ECX,BYTE PTR DS:[EAX]            ;  Get the ASCII char stored in memory at EAX (Serial string pointer)
004570D0  |.  |MOV DWORD PTR DS:[EDX],ECX            ;  Store it an array of integer (see next operation)?
004570D2  |.  |ADD EDX,4                             ;  Move 4 bytes to the right => An array of 32bit Integer values
004570D5  |.  |INC EAX                               ;  Move memory pointer one(1) byte to the right
004570D6  |.  |DEC EBX                               ;  Decrease loop counter
004570D7  |.^ \JNZ SHORT Example_.004570CD           ;  Stop branching when loop counter reaches zero(0)
004570D9  |>  CMP BYTE PTR DS:[45E5AC],7B            ;  Compare first character from the ASCII value with 0x7B ( "{" )
//Code omitted
```

High-Level (Java)

```
String username = getUsername();
int sum = 0;
for(int i = 0;i < username.length(); i++) {
        sum += username.charAt(i);
}

String serial = getSerial();
int[] array = new int[255];
if(serial.length()<=255) {     //Well, Java is safe but we don’t need exceptions popping around.
        for(int i = 0;i < serial.length(); i++) {
                array[i] = serial.charAt(i);
        }
}
```

```
If(serial.charAt(0) == '{') {
        //Code omitted
```

Code Ripping

This is the use of various techniques to copy the binary code of a program into another program or embed it inside a higher programming language that support direct assembly coding. This had nothing to do with *Code Reconstructing* since in *Code Ripping* the effort and time spend on debugging is reduced significantly.

For Example:

Low-level:

```
004570AD  |> /MOV EDX,DWORD PTR DS:[45E5A4]          ;  Load username string in EDX
004570B3  |   |MOVZX EDX,BYTE PTR DS:[EDX+EBX-1]     ;  Get letter in position EBX-1 (in each loop the pointer is incr by 1)
004570B8  |.  |ADD EBP,EDX                           ;  Add the hexadecimal ASCII value of the letter in EBP (UserCount)
004570BA  |. |INC EBX                                ;  Increase the pointer (EBX)
004570BB  |. |DEC EAX                                ;  Decrease the loop counter
004570BC  |.^ \JNZ SHORT Example_.004570AD           ;  Stop branching only when the loop counter reaches zero(0)
004570BE  |>  MOV EBX,0E
004570C3  |.   MOV EAX,Example_.0045E5AC
004570C8  |.   MOV EDX,Example_.0045E5BC
004570CD  |> /MOVZX ECX,BYTE PTR DS:[EAX]            ;  Get the ASCII char stored in memory at EAX (Serial string pointer)
004570D0  |.  |MOV DWORD PTR DS:[EDX],ECX            ;  Store it an array of integer (see next operation)?
004570D2  |.  |ADD EDX,4                             ;  Move 4 bytes to the right => An array of 32bit Integer values
004570D5  |.  |INC EAX                               ;  Move memory pointer one(1) byte to the right
004570D6  |.  |DEC EBX                               ;  Decrease loop counter
004570D7  |.^ \JNZ SHORT Example_.004570CD           ;  Stop branching when loop counter reaches zero(0)
004570D9  |>  CMP BYTE PTR DS:[45E5AC],7B            ;  Compare first character from the ASCII value with 0x7B ( "{" )
//Code omitted
```

High Level Rip:

```
//Code omitted
getUsername(username);
getPassword(password);
user_length := length(username);
pass_length := length(password);
asm
        @loop1:
        MOV EAX,user_length
        MOV EBX,1
        MOV EDX,&username
        MOVZX EDX,BYTE [EDX+EBX-1]
        ADD EBP,EDX
        INC EBX
        DEC EAX
        JNZ @loop1

        //Code omitted

end;
```

Other

The use of licensing services could increase the risks of reverse engineering and keygenning. I am neither against implementing 3rd party components into your software nor do I believe they are a security risk. What renders them a security risk is the weak implementation and the lacks of time spend understanding that software.