

Primer on Reversing .NET Applications

Version 1.2
October 2006

Table of Contents

1. **Reversing .NET and a License File Check** by GooglePlex
2. **Patching a Licence Check of a .NET Application** by zyzygy
3. **Natively Patching MSIL .NET code** by Shub-Nigurath
4. **Serial Fishing in .NET (Live Debugging)** by zyzygy

Editor: Shub-Nigurath

1. Forewords

Finally we have time to publish a decent basic tutorial on .NET. I should thanks zyzygy and GooglePlex who submitted to us their tutorials about .NET.

I thought to merge them into a unique tutorial, and also thought to apply to it the new template we are going to use for our tutorials.

The world of .NET applications has seen a tremendous improvement and finally instruments are mature enough to successfully patch applications. Of course this is a first tutorial for which we will try to introduce the procedures and instruments one can use to decompile and patch applications.

Several extremely good tutorials has been published on this subject but mainly for laziness we didn't have one available..till now ☺

Anyway I added at the end a complete or rich at least list of references for further reading and improving your understanding of this world. Do not underestimate the difficulties you might find.

Ok, time to go! GooglePlex and zyzygy will describe how to patch two applications using decompilation, modifying the MSIL source code (few MSIL details will be given) and then recompiling the new program. I will show instead at the end on another application how to do the same without recompilation, using the MSIL bytecode specifications. zyzygy at the end again, will use *PeBrosePro* to live debugging a .NET application.

*Have phun,
Shub*

Disclaimers

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 document 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, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible damages the companies holding rights on those programs. The scope of this tutorial 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

Verification	2
1. Reversing .NET and a License File Check, GooglePlex	3
1.1. Abstract.....	3
1.2. How to crack this nut	3
1.2.1 Preparation	3
1.2.2 Checking out the target.....	3
1.2.3 Opening the target in .NET Reflector.....	4
1.2.4 The disassembling	7
1.2.5 The reassembling.....	8
1.2.6 The testing	8
1.3. References.....	9
1.4. Conclusions	9
1.5. Greetings	9
2. Patching a License Check of a .NET Application, zyzygy.....	10
2.1. Tools Required	10
2.2. Essay	10
2.3. Assemble.....	16
2.4. Greetings	16
3. Natively Patching MSIL .NET code, Shub-Nigurath.....	17
3.1. Abstract.....	17
3.1.1 Targets:.....	17
3.2. Reversing the Version A.....	17
3.2.1 Analyzing the loader of the application	17
3.2.2 How the program runs.....	18
3.2.3 Analyzing the real Application	19
3.2.4 Patching the application using MSIL bytecode specifications.....	21
3.2.5 Patching the application.....	22
3.2.6 Coding a .NET Oraculum	24
3.3. Reversing the Version B	26
3.3.1 Decompiling the program with IDA	27
3.4. Final Remarks.....	31
4. Serial Fishing in .NET (Live Debugging), zyzygy.....	32
4.1. Tools	32
4.2. Essay	32
5. Further readings	37
5.1. References.....	37
Document History	38

1. Reversing .NET and a License File Check, GooglePlex

1.1. Abstract

The purpose of this tutorial is to learn how to reverse a .NET-application, which also is protected by a license file check.

The application that we are going to reverse is EngiSSol's 2D Frame Analysis Dynamic Edition (or just 2D Truss – you'll see later why).

The tools used for this are Lutz Roeder's .NET Reflector, Microsoft .NET SDK, Notepad, and - of course – brains. No OllyDbg, though. It can't reverse .NET-applications ;{

1.2. How to crack this nut

1.2.1 Preparation

This time, I've done some preparations for you. I've had opened our target in PeiD and found out that it was a .NET application. This forces us to use other software than good ol' Olly, but it really makes our job easier (thanks, Microsoft) because we now can decompile our executable to IL – Intermediate Language – which is much more readable than assembly. We can also recompile the IL to EXE (thanks again, Microsoft).

1.2.2 Checking out the target

It is always a good idea to see how the program reacts because the program's reactions to your inputs are often important flags to search for.

So, let's try to fire up our target and see what happens (Figure 1):

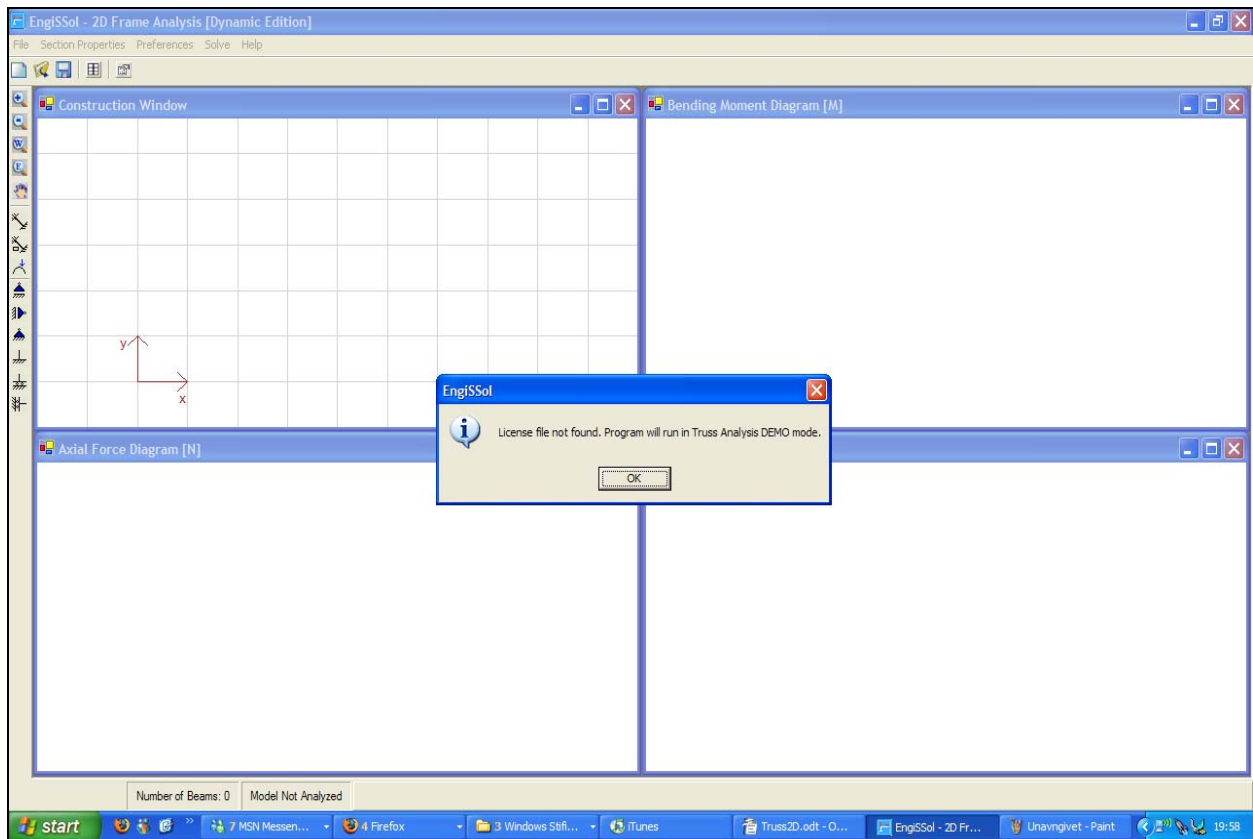


Figure 1 - Oh no! We've got no license!

Hmm...see the title? "2D Frame Analysis [Dynamic Edition]"? And now it will switch to Truss Analysis DEMO? Seems like it's a multiple-version application...

1.2.3 Opening the target in .NET Reflector

Let's open our target in .NET Reflector – Reflector might ask about a default list but just pick the newest you've got. Go to the tree "Frame2D" → "Frame2D.exe" → "Pframe" (see Figure 2):

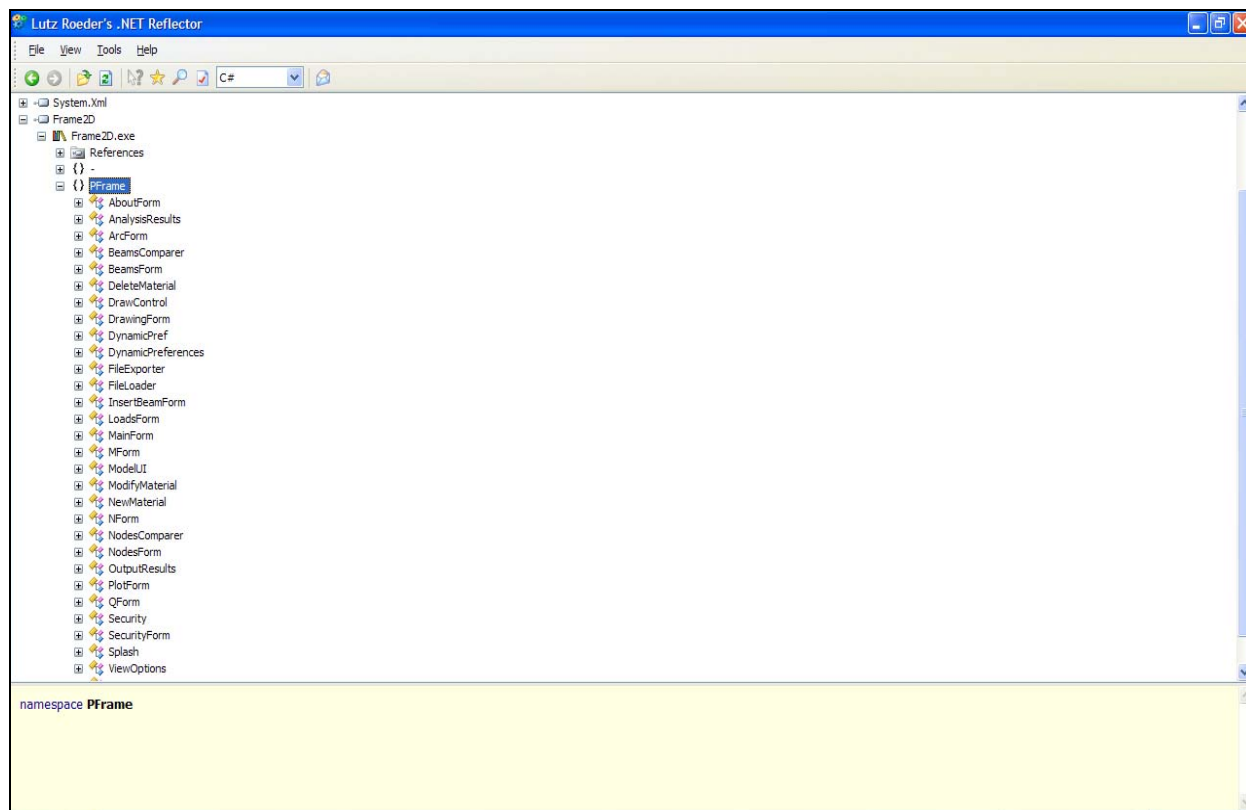


Figure 2 - Our target in .NET Reflector

Lovely.

.NET is neat because the executable is never REALLY compiled – all the source code can be read. And yes, Reflector does the job ;).

You can see the source from IL through C# and Visual Basic to Delphi. Quite nice because you can easier analyze you through what the program does if you know just a little high-level language. Just too bad it can't dump it in high-level language ;).

Now, since our license file check initialized in sync with the application, it is rather interesting to find the main form. Go to "MainForm" in Reflector and expand the tree.

It is stuffed with things that happen within our MainForm. Check out "DemoFrame() : void" - right click and select "Disassembler". If it asks you to manually resolve stuff, just skip it.

You'll see this (I've chosen Visual Basic as the high-level language, but you can alter that in the toolbar) (Figure 3):

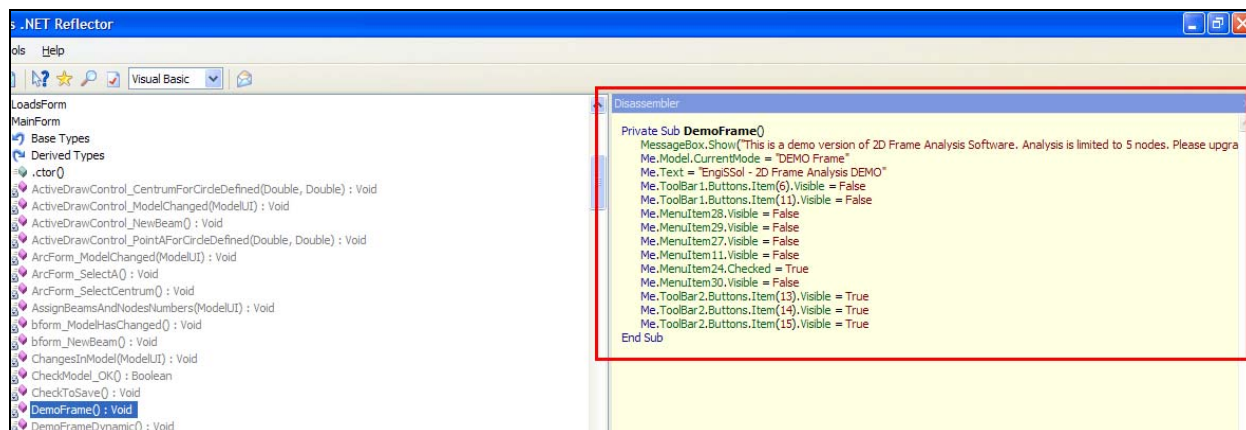


Figure 3 - The code of the procedure DemoFrame()

If you investigate further, there are also procedures called like "DemoFrameDynamic" and "DemoTruss". As the illustration suggests, the procedures tell the program which features that are unlocked and which are not. This confirms my theory about a multi-version program ;).

Seen the message from the illustration before? Yup, that's the second warning you get in the "demo" version of the application! We're close.

Scroll further down. What do we see? A procedure called "MainForm_Load(Object, EventArgs) : void". Nice! Disassemble that! See Figure 4:

```
Private Sub MainForm_Load(ByVal sender As Object, ByVal e As EventArgs)
    Me.MachineName = Environment.MachineName
    ModelUI.Directory = Environment.CurrentDirectory
    Me.Directory = Environment.CurrentDirectory
    Try
        If Not File.Exists((Me.Directory & "\\license.dat")) Then
            MessageBox.Show("License file not found. Program will run in Truss Analysis DEMO mode.", "EngiSSol", MessageBoxButtons.OK, MessageBoxIcon.Asterisk)
            Me.DemoTruss
        Else
            Dim stream1 As New FileStream((Me.Directory & "\\license.dat"), FileMode.Open)
        End If
    End Try
End Sub
```

Figure 4 - Isn't that our first warning? :)

Oh yes indeed. It checks if the license.dat file exists or not. If it doesn't exist it shows the warning and loads the demo version! See Figure 5:

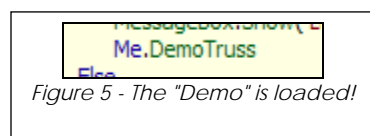


Figure 5 - The "Demo" is loaded!

If you dig deeper into the code, you'll find that the application decrypts the content of the license file to a string that starts with e.g. "Frame Dynamic" depending on the license. It then runs the appropriate procedure and therefore the version of the program.

We could low-level patch the application by altering all references to the license check procedure but it will take too long. We could also reverse the decryption procedure to make our own keygen (only KINGS make keygens – and we've got the code somewhere :D), but that will also take too long.

Instead, we will remove the demo-nag and make it jump immediately to the best program version. It requires no license file ;).

So, when still focused on our little nag, change language to IL. I suggest you print the source, because it's WAY longer than the high-level language.

Now we need to locate our patches.

This seems interesting (Figure 6):

```
L_0020: ldarg.0
L_0021: ldftd string PFrame.MainForm::Directory
L_0026: ldstr "\\license.dat"
L_002b: call string string::Concat(string, string)
L_0030: call bool [mscorlib]System.IO.File::Exists(string)
L_0035: brtrue.s L_0055
L_0037: ldstr "License file not found. Program will run in Truss Analysis DEMO mode."
L_003c: ldstr "EngiSSol"
L_0041: ldc.i4.0
L_0042: ldc.i4.s 64
L_0044: call [System.Windows.Forms]System.Windows.Forms.DialogResult [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string, string, [Sy
L_0049: pop
L_004a: ldarg.0
L_004b: callvirt instance void PFrame.MainForm::DemoTruss()
L_0050: leave L_01f8
```

Figure 6 - The source in IL

I've tried to "translate" the IL to some more understandable to know where to patch. These "translations" are probably not 100 % correct, since I have no experience in this language:

L_0021:	ldftd string PFrame.MainForm::Directory	Push the application's path
L_0026:	ldstr "\\license.dat"	Push the license file name
L_002b:	call string string::Concat(string, string)	Call the strings above
L_0030:	call bool [mscorlib]System.IO.File::Exists(string)	Check if the file exists via mscorlib.dll
L_0035:	brtrue.s L_0055	If true, go to line 0055, else...
L_0037:	ldstr "License file not found. Program will run in Truss Analysis DEMO mode."	...push the warning
L_003c:	ldstr "EngiSSol"	Push the warning's title

L_0041:	ldc.i4.0	Push 0
L_0042:	ldc.i4.s 64	Push 8
L_0044:	call [System.Windows.Forms]System.Windows.Forms.DialogResult [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string, string, [System.Windows.Forms]System. Windows.Forms.MessageBoxButtons, [System.Windows.Forms]System.Windows.Forms.Message BoxIcon)	Call the warning with the strings above
L_0049:	pop	POP
L_004a:	ldarg.0	Load argument at index 0
L_004b:	callvirt instance void PFrame.MainForm::DemoTruss()	Call the procedure DemoTruss() in MainForm
L_0050:	leave L_01f8	End of this procedure, leave to line 01f8

So...we could just NOP the operation about checking the presence of license.dat and just jump directly to the best version of the program, FrameDynamic(). Yup, Let's do that.

1.2.4 The disassembling

OK, let's start to disassemble the application. Take a backup of the application, just in case. Go to the folder where you installed the .NET SDK kit and open the folder "bin". Open the program "ildasm.exe".

Now, open the target ("File → Open") and dump it ("File → Dump") into the Microsoft.NET-folder in your Windows-folder (e.g. "C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727") and name it something unique. Just dump with the suggested options.

There you go, everything you saw in .NET Reflector is now in plain text!

Now, open your .IL-file in notepad (NOTE: NOTEPAD!!! Wordpad and others will just mess up the file due to formatting; the original file structure must be preserved for assembly later).

Search for "MainForm_Load" since it's rather unique. It's the second hit that is our procedure, which is as shown in illustrations 3, 4, and 5 above.

Now delete the demo-warning message in line L_0044 (but keep the line number or else it'll confuse the assembler) and change "instance void Pframe.MainForm::DemoTruss()" in line L_004b to "instance void Pframe.MainForm::FrameDynamic()" - see Figure 7:

```
+ "Analysis DEMO mode."
IL_003c: ldstr      "Engissol"
IL_0041: ldc.i4.0
IL_0042: ldc.i4.s  64
IL_0044:
IL_0049: pop
IL_004a: ldarg.0
IL_004b: callvirt   instance void PFrame.MainForm::FrameDynamic()
IL_0050: leave     IL_01f8
```

Figure 7 - The patched part of the application

Save the file.

1.2.5 The reassembling

Now you need to open the command prompt and go to the Microsoft.NET-folder in your Windows-folder (e.g. "C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727"). Now you type: `ilasm <yourfilename>.il`
If everything goes well, you should have compiled a new EXE file in the directory where you dumped the application ;).

No icon though. Whatever. Not needed ;).

Copy your freshly compiled EXE file to the application's installation directory – the EXE file doesn't need to be called the same as the original (I've tested it).

Let's confirm our changes! Open Reflector, open your new EXE file, and go to the procedure we saw on illustrations 3 and 5.

This is the altered procedure (Visual Basic again) (Figure 8):

```
Me.Directory = Environment.CurrentDirectory
Try
    If Not File.Exists((Me.Directory & "license.dat")) Then
        Me.FrameDynamic
    Else
        Dim stream1 As New FileStream((Me.Directory & "license.dat"
```

Figure 8 - The error message has vanished :D

The stupid nag has disappeared and the procedure jumps directly to the launch procedure for the best program version – but only if the license file doesn't exist. Lovely!

1.2.6 The testing

Now open the application for a test run!

Figure 9 shows what I got:

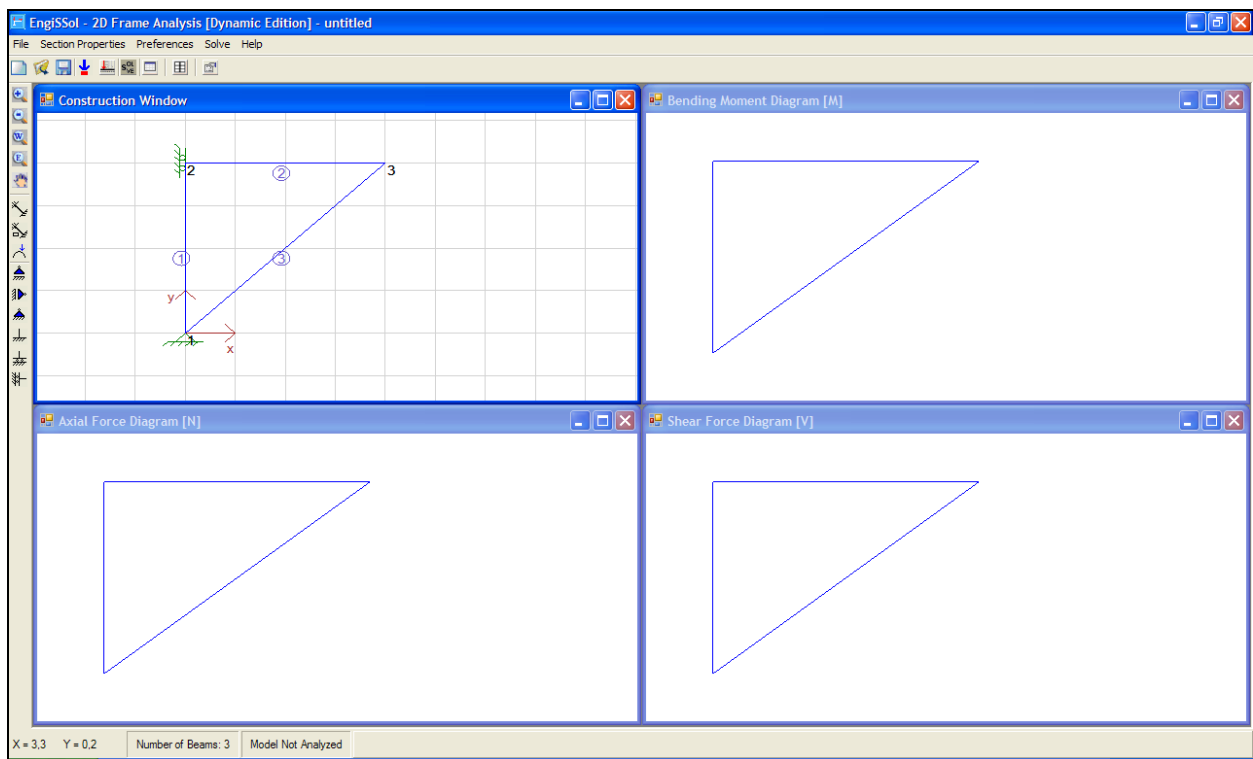


Figure 9 - ;) - working like a charm

It works! Hurray!

1.3. References

No references this time.

1.4. Conclusions

In this tutorial, we have learned how to disassemble, alter, and reassemble .NET applications where in this particular case; we have skipped a license file check.

It turn out that the full source code is available through disassembling, even in many high-level languages.

Remember: If you like this program, please buy it. The cracking is just for fun.

Hope you enjoyed my tutorial! If you have any suggestions, questions or corrections, do not hesitate to PM me at the forums at <http://arteam.accessroot.com>

1.5. Greetings

I wish to thank all the ARTeam members (especially Shub-Nigurrath who offered me to make a standard ARTeam release – this one!) and those who read the first version of this tutorial and gave it comments!

A really special greeting for Lena151, because Lena's tuts taught me the art of reverse engineering.

2. Patching a License Check of a .NET Application, zyzygy

2.1. Tools Required

1. Lutz Reflector (<http://www.aisto.com/roeder/DotNet/>)
2. Microsoft .NET Framework (version depends on the version the target is built on)
3. Text Editor- I would choose Ultra Edit
4. Common Sense
5. iArt (target, <http://www.ipodsoft.com/>)

2.2. Essay

I will try to be as descriptive as possible and since this is my first graphic article, bear with me.

The target is a very good application, which collects all the album arts and lyrics for your iPod.

There are 2 ways to get this program registered, however I shall opt for the tougher one as it will describe more changes and hence we will get acquainted with the language.

Load the application into Lutz Reflector. The approach remains the same as in Win32 platform. We first get to the probable function and modify it to our needs.



Figure 10 - Initial view into Reflector

Figure 10 says a lot itself. We now click on `IsLicensed` or `mdlLicense.IsLicensed`

The function is too big so I shall be pasting only the important areas, but you should browse it (see Figure 11).

```
If (text1 Is Nothing) Then
    mdlLicense.DbgMsg("Demo mode")
    UserName = "Demo"
    mdlMain.UserRegister = "Demo"
    flag1 = True
Else
    UserName = text1
    flag1 = False
End If
mdlLicense.DbgMsg("Check spec license")
If ((mdlMain.MyHDID = 459421) AndAlso mdlLicense.CheckOtherLicense) Then
    flag2 = True
    goto Label_04B1
End If
mdlLicense.DbgMsg("Generating Serial")
num4 = ((num4 + 3000) + 0)
```

Figure 11 - Details of `IsLicensed` function

@Label_04B1

```
End Try
Label_04B1:
  If (num5 <> 0) Then
    ProjectData.ClearProjectError
  End If
  Return flag2
End Function
```

Figure 12 - Label_04B1

This function return Boolean and according to Figure 10 it should return true. The first instance where it sets to true is in Figure 13:

```
mdlLicense.DbMsg("Check spec license")
If ((mdlMain.MyHDID = 459421) AndAlso mdlLicense.CheckOtherLicense) Then
  flag2 = True
  goto Label_04B1
End If
```

Figure 13 - first instance of true being set

If you look at Figure 13 then it compares to the Hardware ID with something and the return value of CheckOtherLicense. If both are true then the flag2 sets true and we jump to the exit with it as registered. Now we shall open up mdlLicense.CheckOtherLicense

```
Public Shared Function CheckOtherLicense() As Boolean
  Dim flag1 As Boolean
  Try
    Dim text1 As String
    Dim num1 As Integer = FileSystem.FreeFile
    FileSystem.FileOpen(num1, Path.Combine(Application.StartupPath, "iTAN.id"), OpenMode.Input)
    FileSystem.Input(num1, text1)
    FileSystem.FileClose(New Integer() { num1 })
    If (Conversion.Val(text1) = 32591213) Then
      flag1 = True
    End If
  Catch exception2 As Exception
    ProjectData.SetProjectError(exception2)
    Dim exception1 As Exception = exception2
    flag1 = False
    ProjectData.ClearProjectError
  End Try
  Return flag1
End Function
```

Figure 14 - CheckOtherLicense function

It returns Boolean. And we must ensure that it returns true always. So we must set flag1=true always. So we can set the "If" condition true. We can just make it true but for practice we shall change the "if" condition so that it always returns true.

And in IsLicensed function if you notice:

```
Dim timer1 As DateTime
Dim num4 As Long
If (StringType.StrCmp(UserName, "CheckLicense2", True) = 0) Then
  UserName = ""
ElseIf mdlLicense.CheckLicense2 Then
  goto Label_041B
End If
ProjectData.ClearProjectError
```

Figure 15 - return test after CheckLicense2

This is present right in the start of the function, so we need to ensure that it doesn't go anywhere unwanted.

Let's summarize what we have to do.

1. In Figure 15 we will jump from either of the if/else if conditions to Figure 13 location.
2. At Figure 13, we shall ask it to compare 459421 with 459421 and also cause the return value of the other function to be true.
3. This will cause the flag2 to be set to true always and we will be jumping to the end (Figure 12).
4. Here it checks for num5 for a nonzero value. I don't know if this has any need but we shall still remove the "if" condition and cause it always to execute since the name suggests ClearProjectError.
5. We are done! ☺

There is a much easier way to do ask it register by always returning true but as an exercise I am following the other method.

Load the program in ILDASM from the MS .NET Framework SDK. And then dump it. You should get a lot of files. Load the .il file into your text editor. Search for IsLicensed function.

We will first make the change in the "if" condition of Figure 13.

Note: While changing any opcode, you need to bother only about the correctness of the opcode the mnemonics that remain of the old opcode can be left as they are. Once you have assembled the modified il file, the new mnemonics will take their place. You just have to change the opcodes!

```

.method /*06000000*/ public static bool
    IsLicensed([opt] string UserName) cil managed
// SIG: 00 01 00 00
{
    .param [1]/*080000416*/ = ""
    // Method begins at RVA 0xa2578
    // Code size      1213 (0x4bd)
    .maxstack 6
    .locals /*110000204*/ init (int32 V_0,
        bool V_1,
        int32 V_2,
        valuetype [mscorlib/* 230000001 */]System.DateTime/* 010000003 */ V_3,
        bool V_4,
        string V_5,
        string V_6,
        int32 V_7,
        int64 V_8,
        string V_9,
        class [mscorlib/* 230000001 */]System.Exception/* 010000070 */ V_10,
        int32 V_11,
        int32 V_12,
        int32[] V_13,
        valuetype [mscorlib/* 230000001 */]System.DateTime/* 010000003 */ V_14,
        int32 V_15)

```

Parameter

Variables used in the function, not parameters!

Figure 16 - IsLicences MSIL view

Now our friend has used the hex value for 459421 which is 0x7029D. How do we get there? Search for it of course. Search for Check spec license, which is right above the "if" condition and you shall be there.

```

IL_00ed: /* 0B | (70)00EAED */ stloc.1
IL_00ed: /* 72 | (70)00EAED */ ldstr    "Check spec license" /* 7000EAED */
IL_00f2: /* 28 | (06)00043A */ call     void iArt.net.mdlLicense/* 0200003A */:::DbgMsg(string) /* 0600043A */
IL_00f7: /* 7E | (04)000292 */ ldsfld    int64 iArt.net.mdlMain/* 0200003B */:::MyHDID /* 04000292 */
IL_00fc: /* 21 | 9D02070000000000 */ ldc.i8    0x7029d
IL_0105: /* 33 | 0F */ bne.un.s    IL_0116

IL_0107: /* 28 | (06)00043F */ call     bool iArt.net.mdlLicense/* 0200003A */:::CheckOtherLicense() /* 0600043F */
IL_010c: /* 2C | 08 */ brfalse.s    IL_0116

IL_010e: /* 17 | */ ldc.i4.1
IL_010f: /* 13 | 04 */ stloc.s     V_4
IL_0111: /* DD | 9B030000 */ leave     IL_04b1

```

Figure 17 - MSIL details

Note: .NET Framework is a stack based framework. So the parameter is pushed on the stack and then processed accordingly.

@ IL_00F7 the HDID is loaded on the stack followed by the number. Now we shall change the code at IL_00F7 to this (Figure 18):

```

IL_00ec: /* 0B | */ stloc.1
IL_00ed: /* 72 | (70)00EAED */ ldstr    "Check spec license" /* 7000EAED */
IL_00f2: /* 28 | (06)00043A */ call     void iArt.net.mdlLicense/* 0200003A */:::DbgMsg(string) /* 0600043A */
IL_00f7: /* 21 | 9D02070000000000 */ ldc.i8    0x7029d
IL_00fc: /* 21 | 9D02070000000000 */ ldc.i8    0x7029d

```

Figure 18 - MSIL details

Simply change:

```
ldsfld    int64 iArt.net.mdlMain
```

to

```
ldc.i8      0x7029d
```

And save.

Reason: The number 0x7029d as indicated is of type i8 (integer of 8 bytes). So we have to put a similar data type for it get compared!

This will cause the comparison of same numbers which will always result to true.

Next we shall move towards changing the return value of CheckOtherLicense() to true.

Search for the name and then we shall be setting the flag true at both cases.

Inside CheckOtherLicense(), see Figure 19

```
IL_0034: /* 11 | 04          */ ldloc.s   V_4
IL_0036: /* 28 | (0A)0000F6 */ call     void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.FileSystem/*
IL_003b: /* 08 |          */ ldloc.2
IL_003c: /* 28 | (0A)000023 */ call     float64 [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.Conversion
IL_0041: /* 23 | 000000D0D6147F41 */ ldc.r8   32591213.
IL_004a: /* 33 | 04          */ bne.un.s   IL_0050
```

Figure 19 - CheckOtherLicense() MSIL detail

This compares the float value to the 32591213. So we simple replace IL_003B and IL_003C with ldc.r8 32591213.

Change:

```
IL_003b: /* 08 |          */ ldloc.2
IL_003c: /* 28 | (0A)000023 */ call     float64 [Microsoft.VisualBasic]
```

to

```
IL_003b: /* 23 | 000000D0D6147F41 */ ldc.r8   32591213.
```

And replace the unused offsets with nop, I don't think you need to do this because they get assembled and a new address is compiled (but I am did it just as a backup, though logically there seems to be no reason). Something like Figure 20:

```
IL_0036: /* 28 | (0A)0000F6 */ call     void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic
IL_003b: /* 23 | 000000D0D6147F41 */ ldc.r8   32591213.
IL_003c: /* 00 |          */ nop
IL_003d: /* 00 |          */ nop
IL_003e: /* 00 |          */ nop
IL_003f: /* 00 |          */ nop
IL_0040: /* 00 |          */ nop
IL_0041: /* 23 | 000000D0D6147F41 */ ldc.r8   32591213.
IL_004a: /* 33 | 04          */ bne.un.s   IL_0050
```

Figure 20 - patched code

Now there we need to set the flag in the exception region as well to true.

```
catch [mscorlib/* 23000001 */]System.Exception/* 01000070 */
{
    IL_0052: /* 25 |          */ dup
    IL_0053: /* 28 | (0A)000040 */ call     void [Microsoft.VisualBasic/*
    IL_0058: /* 0D |          */ stloc.3
    IL_0059: /* 16 |          */ ldc.i4.0 Change this to ldc.i4.1
    IL_005a: /* 0A |          */ stloc.0
    IL_005b: /* 28 | (0A)000041 */ call     void [Microsoft.VisualBasic/*
    IL_0060: /* DE | 00          */ leave.s   IL_0062
} // end handler
```

Figure 21

Reason: As said earlier .NET is a stack based compiler. Thereby for setting the values, it uses stack. Now ldc.i4.0 indicates that it will load the value 0 and stloc.0 indicates that it will store the value on location 0 of the evaluation stack.

So we change it to ldc.i4.1.

Now we have ensured that flag2 will always be set to true. We now will make the changes in Figure 15 to ensure that under any circumstances the code will jump only to Figure 13.

To reach to the proper place, just search for the appropriate strings near by the code.

IL_0000:	/* 02		*/ ldarg.0	
IL_0001:	/* 72		*/ ldstr	"CheckLicense2" /* 7000EA35 */
IL_0006:	/* 17		*/ ldc.i4.1	
IL_0007:	/* 28		*/ call	int32 [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.CompilerS
IL_000c:	/* 16		*/ ldc.i4.0	
IL_000d:	/* 33		*/ bne.un.s	IL_0018 ← Jump to Else If
IL_000f:	/* 72		*/ ldstr	"" /* 70000015 */
IL_0014:	/* 10		*/ starg.s	UserName
IL_0016:	/* 28		*/ br.s	IL_0022
IL_0018:	/* 28		*/ call	bool iArt.net.mdllLicense/* 0200003A */::CheckLicense2() /* 0600043D */
IL_001d:	/* 3A		*/ brtrue	IL_041b

Figure 22

The boxed codes are the locations from where we loose the control of execution. What we want is set the flag2=true in any case. So rather than causing them to these jumps to unwanted locations, we shall make them jump to the location in Figure 13. We find that the location of the flag2=true if condition is at IL_00ED, if you have made changes as specified in this article.

br.s indicates short jump. br indicates far jump.

brtrue indicates far jump if true.

IL_0000:	/* 02		*/ ldarg.0	
IL_0001:	/* 72		*/ ldstr	"CheckLicense2" /* 7000EA35 */
IL_0006:	/* 17		*/ ldc.i4.1	
IL_0007:	/* 28		*/ call	int32 [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.CompilerS
IL_000c:	/* 16		*/ ldc.i4.0	
IL_000d:	/* 33		*/ bne.un.s	IL_0018 ← String Compare
IL_000f:	/* 72		*/ ldstr	"" /* 70000015 */
IL_0014:	/* 10		*/ starg.s	UserName
IL_0016:	/* 2B		*/ br.s	IL_0022
IL_0018:	/* 28		*/ call	bool iArt.net.mdllLicense/* 0200003A */::CheckLicense2() /* 0600043D */
IL_001d:	/* 3A		*/ brtrue	IL_041b ← Else If
IL_0022:	/* 28		*/ call	void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.CompilerSe

Figure 23

I shall explain you the changes that are needed to be done.

```

Dim time1 As DateTime
Dim num4 As Long
If (StringType.StrCmp(UserName, "CheckLicense2", True) = 0) Then
    UserName = ""
ElseIf mdllLicense.CheckLicense2 Then
    goto Label_041B
End If
ProjectData.ClearProjectError

```

Figure 24

After due experimentation, I conclude that the "if" condition always fails and the else if condition is always executed. I used to force the jump to the Figure 13 location and it used to get registered to iArtDemo. Next I tried a logic such that neither the "if" or else if conditions are executed and there I struck gold. I could register to the name I chose to.

```

IL_0000: /* 02 |          */ ldarg.0
IL_0001: /* 72 | (70)00EA35 */ ldstr    "CheckLicense2" /* 7000EA35 */
IL_0006: /* 17 |          */ ldc.i4.1
IL_0007: /* 28 | (0A)000007 */ call     int32 [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.CompilerSe

IL_000c: /* 16 |          */ ldc.i4.0
IL_000d: /* 33 | 09          */ bne.un.s  IL_0018 ← IL_0018 indicates the ELSE IF location. Instead I force it to jump to
                                                    IL_0022. This means that the ELSE IF condition is embedded in the IF
                                                    and will not get executed, which is what we want *if* we want the app to
                                                    get register to our name

IL_000f: /* 72 | (70)000015 */ ldstr    "" /* 70000015 */
IL_0014: /* 10 | 00          */ starg.s   UserName
IL_0016: /* 2B | 0A          */ br.s     IL_0022 ← This jump is immaterial to us, now.

IL_0018: /* 28 | (06)00043D */ call     bool iArt.net.mdlLicense/* 0200003A */::CheckLicense2() /* 0600043D */
IL_001d: /* 3A | F9030000 */ brtrue   IL_041b ← End if, we cause it to jump to IL_00ED

IL_0022: /* 28 | (0A)000041 */ call     void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.CompilerSe

```

Figure 25

Modified Code:

```

IL_000c: /* 16 |          */ ldc.i4.0
IL_000d: /* 33 | 09          */ bne.un.s  IL_0022 ← Changed from IL_0018 to IL_0022, thereby
                                                    embedding the else if

IL_000f: /* 72 | (70)000015 */ ldstr    "" /* 70000015 */
IL_0014: /* 10 | 00          */ starg.s   UserName
IL_0016: /* 2B | 0A          */ br.s     IL_0022

IL_0018: /* 28 | (06)00043D */ ldc.i4.1
IL_001d: /* 3A | F9030000 */ brtrue   IL_041b

IL_0022: /* 28 | (0A)000041 */ call     void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic

```

Figure 26

Now for the final patch. This I firmly believe that there seems to be no use but here it goes anyway to serve as a backup. We need to remove the "if" condition in Figure 12. After you are through with so much of code modification, you can easily figure out the necessary changes.

```

IL_04a7: /* FE1A |          */ rethrow
IL_04b1: /* 11 | 04          */ ldloc.s   V_4
        .try IL_0000 to IL_046c filter IL_046c handler IL_0481 to IL_04b1
        // HEX: 01 00 00 00 00 00 00 00 6C 04 00 00 81 04 00 00 30 00 00 00 6C 04 00 00
IL_04b3: /* 11 | 0B          */ ldloc.s   V_11
IL_04b5: /* 2C | 05          */ brfalse.s IL_04bc

IL_04b7: /* 28 | (0A)000041 */ call     void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic.CompilerSe
IL_04bc: /* 2A |          */ ret
} // end of method mdlLicense::IsLicensed

```

Figure 27

brfalse indicates that 0 is already on the stack and it will be compared to the value of the variable on the stack.

ldloc means that it will load the value on the stack from the location(V_11).

So we will remove ldloc.s V_11 and brfalse.s IL_04bc and replace it with nops like so:

```

IL_04b3: /* 00 |          */ nop
IL_04b4: /* 00 |          */ nop
IL_04b5: /* 00 |          */ nop
IL_04b6: /* 00 |          */ nop

IL_04b7: /* 28 | (0A)000041 */ call     void [Microsoft.VisualBasic/* 23000002 */]Microsoft.VisualBasic
IL_04bc: /* 2A |          */ ret

```

Figure 28

2.3. Assemble

Now all the necessary changes have been done. We shall now assemble it. Open your command prompt. Make sure your IL and other files are located in a folder say in C:\ilcode. I found certain glitches when assembling files which are located in folders which have white spaces in their names.

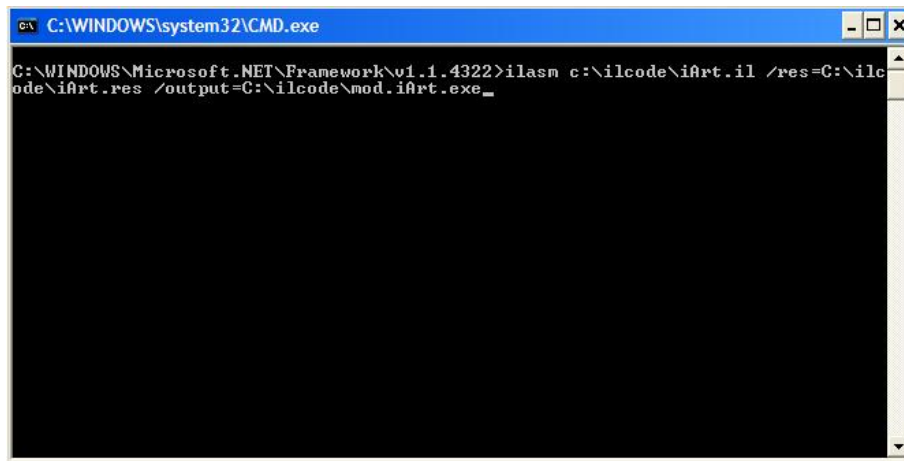


Figure 29 – reassembling the new code

And press return. Your code should get compiled. Now copy the file in the directory it was installed and you have the licensed version.

The easier way of patching is to return value true regardless of the value in flag2.

2.4. Greetings

This application has been coded nicely. I had tons of fun reversing it and learnt a lot while writing the article. This should clear any doubts while approaching to reverse engineer a .net application.

Greets to everybody I know and the author of this program, for the wonderful program, I would buy it if I had an iPod!

3. Natively Patching MSIL .NET code, Shub-Nigurrath

3.1. Abstract

The target of this tutorial is PoolAimer, a quite old application, one of the first .NET applications but interesting for educational purposes, because it's a native .NET application without any additional protection. The tutorial will cover versions 1.0.1783.42357 and 1.0.1796.1250, two subsequent releases of the same program: the first not using any obfuscation trick the second using a simple obfuscation trick (now not that much used), which prevented using either decompilation or Lutz Reflector. This second case will be handled using IDA, which can successfully decompile .NET applications using a different technique.

3.1.1 Targets:

Much probably these two specific version are no more existing on the web, or has been updated so I uploaded to our site two mirrors of the original distributions.

Version A - PoolAimer version 1.0.1783.42357:

http://arteam.accessroot.com/tools/highstakespoolaimer_V_1.0.1783.42357.exe

Version B - PoolAimer version 1.0.1796.1250:

http://arteam.accessroot.com/tools/highstakespoolaimer_1.0.1796.1250.exe

I will call Version A the first version, not using any obfuscation method, and Version B the second version, a little more protected.

3.2. Reversing the Version A

3.2.1 Analyzing the loader of the application

If you install the program you will see the presence of the main program (PoolAimer.exe). It is written with C++ and packed with UPolyX v0.5* (this is what PEiD reports, easily unpacked).

This is the loader of the main program; also the size of this little program is not too much, even considering that is compressed.

Developers of the program also stripped away the linker information but it's anyway a C++ program which loads the *.rsm file, also installed into the same folder.

The PoolAimer.rsm is a .NET executable but it's not a completely fixed .NET application, you cannot launch it directly! The role of the C++ loader is of rebuilding the real .NET application from the *.rsm file and launch it.

It indeed does some modifications in memory before calling the _CoreExeMain (which starts the execution of the .NET code, see following note).

Note: the trick that allows .NET programs contained into old PE format is simple. The system (e.g. the program loader) as well as the PE format was not suitable for new .NET requirements. Developers at Microsoft thought to insert a little trick: the .exe containing the .NET program is a real .exe which entry point just calls the system API `_CoreExeMain` (if you try to open a .NET program with Olly you will only land at the beginning of that API). This special API simply starts the CLR Common Language Runtime sub system and starts executing MSIL code he finds into the calling exe. Doing this way the system thinks to load a normal .exe and doesn't worry if all the real instructions are .NET or anything else [1].

if you place a BP 0x7000122A in PoolAimer.exe you will land here:

```

70001218 . 50          PUSH EAX          ; /pOldProtect
70001219 . 6A 04       PUSH 4           ; |NewProtect = PAGE_READWRITE
7000121B . 68 E0000000 PUSH 0E0        ; |Size = E0 (224.)
70001220 . 8B8D D0FBFFF MOV ECX,DWORD PTR SS:[EBP-430] ; |
70001226 . 83C1 18     ADD ECX,18       ; |
70001229 . 51         PUSH ECX        ; |Address
7000122A . FF15 00200070 CALL DWORD PTR DS:[<&KERNEL32.VirtualPro>; \VirtualProtect
```

Here you can see that "poldProtect" address (registry EAX) points to part of the PE header of PoolAimer.rsm.

Further investigating, you will easily see that in the following instructions the loader writes something modifying the PE header fields of the just loaded PoolAimer.rsm.
for example here:

```
70001248    . 8D8D CCFBFFFF LEA ECX,DWORD PTR SS:[EBP-434]
```

it writes some information...

Good, if we go further tracing the execution we will see that the page protections are replaced back:

The first part terminates here,

```
700012AC    .- FFA5 E8FEFFFF JMP DWORD PTR SS:[EBP-118] ; mscoree._CorExeMain
```

where PoolAimer.exe passes the execution to the entry point of the repaired .NET application, _CorExeMain.

We have then now a repaired .NET executable ready to be analyzed. Go into memory the map of OllyDbg (ALT-M), look the loaded module called PoolAi_1, this is PoolAimer.rsm.

Dump each section one by one, double clicking on it, the right click -> Backup -> Save Data to file.

When you have done each one join all them into a unique file (respecting the order).
Rename the resulting file as PoolAimer.rsm.exe.

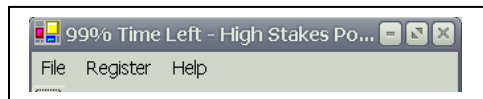
The resulting application needs some other PE header fixing but it's all here.. the .NET programs have a little IAT, essentially only the address of _CoreExeMain.

But at the point where we dumped the program it was only a blob in memory so IAT fixing is required. The only think it's still required is a fixing of the PE Header values.

Why such a loader has been used for? The real result of this loader is to prevent simple decompilation using **ilasm**, so previous approaches will not work. Anyway you have to follow above steps if you want to get rid of the loader or want to decompile it. I will anyway go over the loader approaching the problem from another point of view.

3.2.2 How the program runs

An important thing is always to see how a program runs and how the protection/limitations appear.
This is how this program looks:



The Menu Register is really important; it's added to the main form.
If you press it you will get the following dialog:

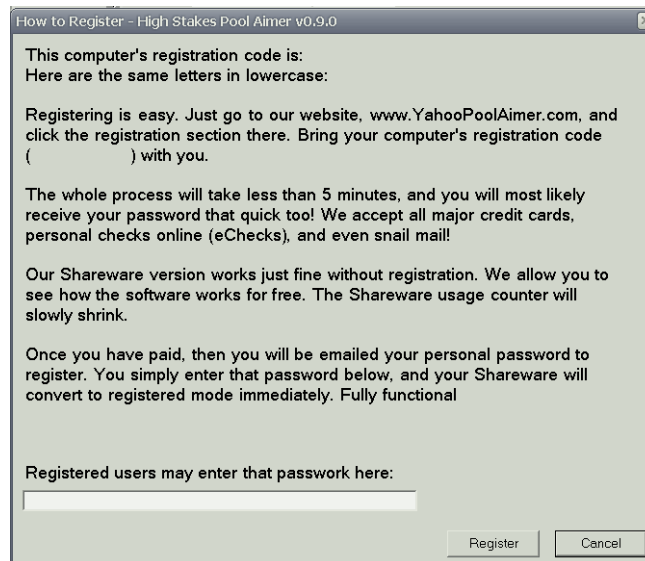


Figure 30 - Registration nag

3.2.3 Analyzing the real Application

Fortunately the modifications did to PoolAimer.rsm (and repaired by the loader) are not enough to fool Reflector then I can decompile the source code with it.

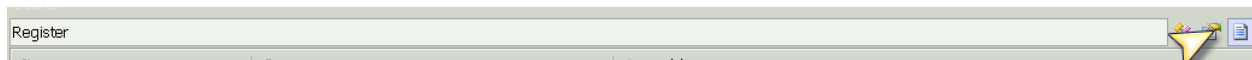
First of all I suggest installing all the Reflector Add-ins. Add-ins are available here:

<http://www.aisto.com/incoming/Reflector/AddIns/>

There are plenty, a check on each might be interesting (remember that starting from Reflector 4.2 the plugin architecture has changed then you probably will have to reload all the plugins you eventually had before and also for this reason some not updated plugins will not work), because some are really interesting stuffs¹

Note: Reflector is not the only tool available. Reflector is a free program but there's at least other two: Anakrino (not updated since ages), but has been the first to appear on the market, and Reflector's Salamander[23], which a really interesting tool, especially because it is mini-deployment tool which allows you to link .NET assemblies together into a single file, and to deploy your application without installation of the whole Microsoft .NET Framework.

First of all I used the search function and searched for the string "Register":



The results are the following; of course I'm interested only in those called from PoolAimer:

A	a.a	PoolAimer, Version=1.0.1783.42357, Culture=neutral, PublicKeyToken=null
.ctor	a.A	PoolAimer, Version=1.0.1783.42357, Culture=neutral, PublicKeyToken=null
A	a.A	PoolAimer, Version=1.0.1783.42357, Culture=neutral, PublicKeyToken=null
A	A.A	PoolAimer, Version=1.0.1783.42357, Culture=neutral, PublicKeyToken=null
A	A.A	PoolAimer, Version=1.0.1783.42357, Culture=neutral, PublicKeyToken=null
GetFormat	System.Windows.Forms.DataFormats	PoolAimer, Version=1.0.1783.42357, Culture=neutral, PublicKeyToken=null

I found this result particularly interesting, into the method A.A:

```
this.C.set_Index(1);
this.C.set_Text("Register");
this.C.Click += new EventHandler(this.I);
```

¹ For example a really interesting plugin is **DebReflector** (<http://www.felicepollano.com/category/7.aspx>), a debugger for .NET applications working from Reflector or **FileGenerator**, a plugin that decompile into a chose language a whole assembly and creates required project files for VC.NET (<http://www.denisbauer.com/NETTools.aspx>)

This is where the handler of the menu "Register" is set. The method associated to this handling is I, of the same class (due to this.I).

```
private void I(object A_0, EventArgs A_1)
{
    string text1;
    if (((this.A.ShowDialog() == DialogResult.OK) && this.A.A().A(out text1)) &&
    this.A.A().A(text1))
    {
        this.C.set_Visible(false);
        this.set_Text("High Stakes Pool Aimer v0.9.0");
    }
}
```

Well what this function does? It gets the text1 from the dialogbox using function A.A().A(out text1) and calls the function this.A.A().A(text1), which prototype a.a.A(String) is the following one:

```
public bool A(string A_0)
{
    A_0 = A_0.Trim();
    A_0 = A_0.ToUpper();
    return (A_0 == this.I);
}
```

This function takes the string A_0, erases the spaces (Trim) and converts in Uppercase. The it is compared with the serial number stored in the property I of the class A (I know naming algorithm of Reflector is a little confusing!) which is a string:

```
private string I;
```

If you are a C++ programmer and want to see how this looks like with Managed C++ (MCP, the C++ extension compatible with .NET) you can install the add-in Reflector.McppLanguage.dll which adds the MCP to the list available decompilation languages. The same method becomes:

```
public: System::Boolean A(System::String __gc * A_0)
{
    A_0 = A_0->Trim();
    A_0 = A_0->ToUpper();
    return (A_0 == this->I);
}
```

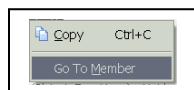
Which is crystal clear for a C++ programmer, and not only.

The really handy feature of Reflector is that allows browsing the different functions with hyperlinks! A not often known function of Reflector is the code navigation and references finding: in the method tree if you press CTRL-R (Analyze) or right click on a method, Reflector opens a browse navigation panel, reporting the "Depends on" and "Used by" relations of the selected function.

For example Figure 31 reports the result for function a.a.A(String), identified above as the place where registration is checked.



Figure 31 - Browse Navigation of function a.a.A(String)



Analysis of Figure 31 shows that the serial number check is done also by the constructor of the A.A object and by a.a.A(String) and already identified A.A.I(). You can then use the context menu on the name of the "Used by" function and go to the implementation.

For example a.a.A(String) is actually self-explanatory:

```
private void A(object A_0, EventArgs A_1)
{
```

```

        if (this.A.A(this.A.get_Text()))
        {
            this.A.A(this.A.get_Text());
            MessageBox.Show(this, "Thank you for your Registration! We hope you enjoy our
software!\r\nPlease restart the application.", "Registration - " + this.A);
        }
        else
        {
            MessageBox.Show(this, "Invalid Registration Code, please verify your code and reenter
it.", "Registration - " + this.A);
        }
    }
}

```

3.2.4 Patching the application using MSIL bytecode specifications

Well the action is to transform the identified code to something more responsive to any serial address we might enter into the application ☺

We want to convert the above function to something like:

```

public bool A(string A_0)
{
    A_0 = A_0.Trim();
    A_0 = A_0.ToUpper();
    return (A_0 == A_0);
}

```

which always returns a good result!

Practically the important thing is that the function returns always true, but I have to do it considering the stack integrity. Any method is fine.

I said at the beginning that the particular protection used avoid recompiling the *.rsm program, because it will not work. This is a specific limitation but patching using recompilation is not always the best thing: there are several reasons:

1. if you recompile the whole program you will have to distribute the whole new executable and will be impossible to create a small simple byte patcher, like for the usual Win32 native programs.
2. if you get a protected or obfuscated application you will not usually be able to decompile. I will explain why later on, when patching Version B.
3. if you do not have all the required modules (libraries) you will not be able to recompile the program again
4. if you want to recompile the program you have to install much more software to do it.

Definitely is an option I don't like that much, given that there's an alternative.

What you need is the MSIL bytecodes specification and a few more patience. The result is the possibility to still distribute a byte patcher and not recompiling all the things and avoid above problems.

What I am going to do is to directly write the op codes of the new instructions in place of the previous, so as to really patch the .NET application. I must anyway underline that patching using this method has one disadvantage: you must respect the space you already have available, because you are not recompiling the code, all the offsets cannot be modified.

You can use the "Table 1: Opcode Encodings" available at this address:

<http://dotnet.di.unipi.it/EcmaSpec/PartitionIII/cont1.html#Table1OpcodeEncodings>

or even you can use Reflector once more: if you select MSIL like decompilation language, you can see the MSIL equivalent code and if you leave the mouse over each MSIL instruction Reflector shows a popup with the opcode and few explanation, but the arguments coding will not be shown by Reflector, so you cannot avoid reading the documents below specified, sorry!

For example the function we want to patch looks like this in MSIL:

```

.method public hidebysig instance bool A(string A_0) cil managed
{
    .maxstack 8
    L_0000: ldarg.1

```

```

L_0001: callvirt instance string string::Trim()
L_0006: starg.s A_0
L_0008: ldarg.1
L_0009: callvirt instance string string::ToUpper()
L_000e: starg.s A_0
L_0010: ldarg.1
L_0011: ldarg.0
L_0012: ldfld string a.a::I
L_0017: call bool string::op_Equality(string, string)
L_001c: ret
}

```

At this point I suggest using one of the most valuable reversing instruments I ever used (brain apart): pen and paper.

The first thing to get in your hand is the ECMA Specifications relative to CLI and particularly the Chapter 3 (Base Instructions) of the following section:

Common Language Infrastructure (CLI)
Partition III
CIL Instruction Set

<http://www.ecma-international.org/publications/files/ecma-st/ECMA-335.pdf>

this document is available at the page :

<http://www.ecma-international.org/publications/standards/Ecma-335.htm>

or you can download for an offline reading getting it here <http://dotnet.di.unipi.it/ecmaspec/ecmaspecs.zip>

Through this document you can find the equivalence opcode-MSIL instruction, useful to directly code the new MSIL instructions inside the method we want to change.

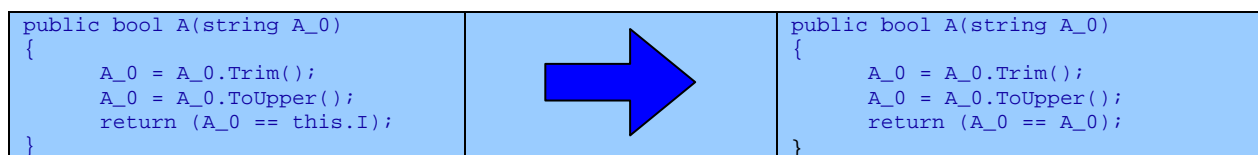
There are two methods one can follow at this point:

1. use pen and paper and write the new ILSDASM instructions then the new opcodes.
2. write a small equivalent C# program and compile it, then you can disassemble it and take the MSIDASM instructions the compiler generated for you. These instructions will be the new instructions.

Which one to use depends on your habits and on the size of the new code you must modify. Remember that the CLR is a stack based machine!

3.2.5 Patching the application

As I explained before the patch consist in doing the following changes:



I need to transform the IL routine and find the new opcodes. The current routine has these opcodes.

```

.method public hidebysig instance bool A(string A_0) cil managed
{
    // Code Size: 29 byte(s)
    .maxstack 8
    L_0000: ldarg.1                03
    L_0001: callvirt instance string string::Trim()    6F <ARGS>
    L_0006: starg.s A_0            10 01
    L_0008: ldarg.1                03
    L_0009: callvirt instance string string::ToUpper() 6F <ARGS>
    L_000e: starg.s A_0            10 01
    L_0010: ldarg.1                03
    L_0011: ldarg.0                02

```

```

L_0012: ldfld string a.a::I          7B <ARGS>
L_0017: call bool string::op_Equality(string, string) 28 <ARGS>
L_001c: ret                          2A
}

```

The hex sequence to find is 10-01-03-02-7B I must find where it is in the binary file, because the <ARGS> are coded in a specific way. Unfortunately there are two instances of this sequence, but the correct one is this:

```

0006 7D39 0000 042A 7603 6F97 0000 0A10 0103 6F7C 0400 0A10 0103 027B 3900 0004 2811 0000
0A2A 0000 0330 0300 1F00 0000 0100 0011 7E48 0200 0A0A 0206 729F 0000 7028 6000 0006 0206

```

This is the equivalent with the instructions bytecodes in bold:

```

00 00 04 2A 76 03 6F 97 00 00 0A 10 01 03 6F 7C 04 00 0A 10 01 03 02 7B 39 00 00 04 28 11 00 00 0A 2A 00 00

```

Please note that the method starts and finishes with two 00 00. You can exercise your MSIL skills trying to find why the <ARGS> are coded this way.

For example:

A_0.Trim()	L_0000: ldarg.1 L_0001: callvirt instance string string::Trim()	03 6F 97 00 00 0A
Part of return (A_0==this.I);	ldfld string a.a::I	7B 39 00 00 04

97-00-00-0A and 39-00-00-04 are two example of coded <ARG>.

From CIL Instruction Set specification at the § 4.2 it is reported the following description for **callvirt** method:

method is a metadata token (a methoddef, methodref or methodspec see Partition II) that provides the name, class and signature of the method to call.

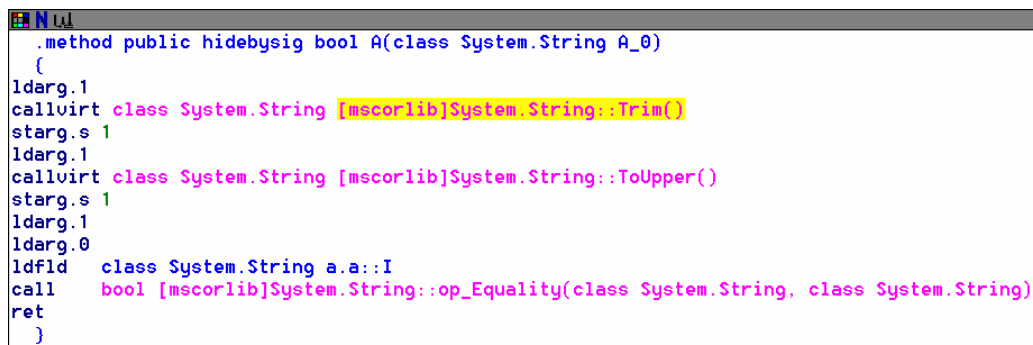
Instead the field of the instruction **ldfld** field:

field is a metadata token (a fieldref or fielddef see Partition II) that shall refer to a field member.

From the above definitions you can understand that the coding of the opcodes is absolutely not simple: the <ARGS> are essentially a coded the indexes of a table of object elements. The interesting thing that you can understand from this is that these bytes are position independent, the same instruction can be coded with the same bytes in the entire program.

Fortunately there's who can do it for us: IDA.

Figure 32 shows how the same method looks into IDA.



```

.method public hidebysig bool A(class System.String A_0)
{
    ldarg.1
    callvirt class System.String [mscorlib]System.String::Trim()
    starg.s 1
    ldarg.1
    callvirt class System.String [mscorlib]System.String::ToUpper()
    starg.s 1
    ldarg.1
    ldarg.0
    ldfld class System.String a.a::I
    call bool [mscorlib]System.String::op_Equality(class System.String, class System.String)
    ret
}

```

Figure 32 - IDA MSIL view of registration check routine



If you take care into the IDA Hex-View to select Synchronize with IDA-View, you can go with the cursor on the instruction you want decode and then switch to the Hex-View. IDA selects the whole set of bytes for that instruction.

So the method with all the bytes placed is the following:

```
.method public hidebysig instance bool A(string A_0) cil managed
{
    // Code Size: 29 byte(s)                                042A76 (method's signature)
    .maxstack 8
    L_0000: ldarg.1                                           03
    L_0001: callvirt instance string string::Trim()           6F 9700000A
    L_0006: starg.s A_0                                       10 01
    L_0008: ldarg.1                                           03
    L_0009: callvirt instance string string::ToUpper()       6F 7C04000A
    L_000e: starg.s A_0                                       10 01
    L_0010: ldarg.1                                           03
    L_0011: ldarg.0                                           02
    L_0012: ldfld string a.a::I                               7B 39000004
    L_0017: call bool string::op_Equality(string, string)    28 1100000A
    L_001c: ret                                              2A
}
```

It's time to pack the patch. The method as said before becomes.

```
.method public hidebysig instance bool A(string A_0) cil managed
{
    // Code Size: 29 byte(s)                                042A76 (method's signature)
    .maxstack 8
    L_0000: ldarg.1                                           03
    L_0001: callvirt instance string string::Trim()           6F 9700000A
    L_0006: starg.s A_0                                       10 01
    L_0008: ldarg.1                                           03
    L_0009: callvirt instance string string::ToUpper()       6F 7C04000A
    L_000e: starg.s A_0                                       10 01
    L_0010: ldarg.1                                           03
    L_0011: ldarg.1                                           03
    L_0017: call bool string::op_Equality(string, string)    28 1100000A
    nops                                                       00 00000000
    L_001c: ret                                              2A
}
```

In order to keep the offsets as said before, I inserted *nops* just before the final *ret*.

So the final byte patterns a patcher should switch are the following:

```
original:    02 7B 39000004 28 1100000A 2A
new bytes:   02 28 1100000A 00 00000000 2A
```

At this point any registration number you enter into the application will be accepted!

3.2.6 Coding a .NET Oraculum

At this point we are ready for something more complex. Do you remember my tutorial on writing Oraculums², for application, also called serial sniffer.

The idea is that the function `a.a.A(String)` already knows the correct serial number, actually it is compared with the one inserted by the user: `A_0` is the one inserted by the user, while `this.I` contains the correct one.

So why not modifying the function so as to make it showing a `MessageBox` with the correct serial number? This is what we will do here; it will also help you to understand a little more MSIL language.

Once more the `a.a.A(String)` is the following:

```
public bool A(string A_0)
{
    A_0 = A_0.Trim();
    A_0 = A_0.ToUpper();
    return (A_0 == this.I);
}
```

² "Guide on How to play with processes memory, write loaders and Oraculums", Shub-Nigurath of ARTeam, <http://tutorials.accessroot.com>

I want it to become something similar to the following:

```
public bool A(string A_0)
{
    MessageBox.Show(null, "Thank you for your Registration! ", "Registration: " + this.I);
}
```

How to use the equivalent MSIL code? I re-used the code already used into the application.

Into Reflector search for the word "MessageBox" using the "Type Search", you should get the results in Figure 33: the right one is the one used by PoolAimer.

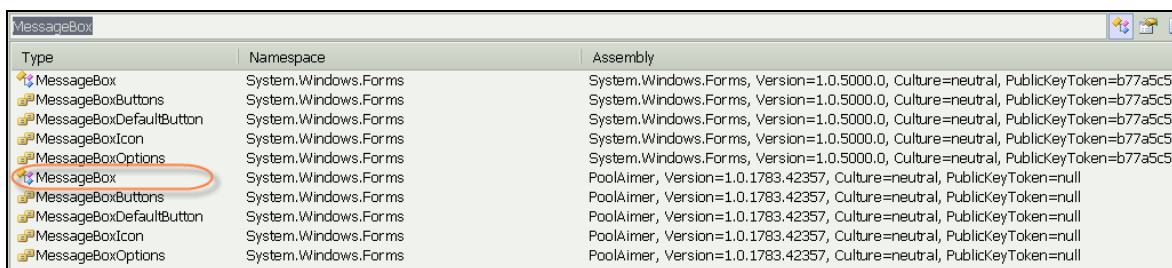


Figure 33 - MessageBox search results

Double click on the result circled in Figure 33 and then use the Analyze function (CTRL-R) already explained before to find the "Used by" results (Figure 34). As you can see there are several places where this API is already used then we can grab the MSIL code from one of them and adapt it to our needs.

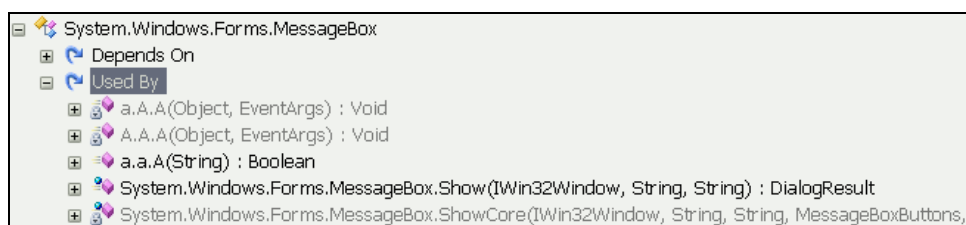


Figure 34 - MessageBox "Used by" details

We will use the member `a.A.A(Object, EventArgs) : Void` which contains something similar to our needs:

```
private void A(object A_0, EventArgs A_1)
{
    if (this.A.A(this.A.get_Text()))
    {
        this.A.A(this.A.get_Text());
        MessageBox.Show(this, "Thank you for your Registration! We hope you enjoy our software!\r\nPlease restart the application.", "Registration - " + this.A);
    }
    else
    {
        MessageBox.Show(this, "Invalid Registration Code, please verify your code and reenter it.", "Registration - " + this.A);
    }
}
```

The new piece of code we need, already adapted and translated to MSIL becomes:

```
14          ldnull
72 E9 09 00 70 ldstr "Thank you for your Registration! We hope you enjoy our software!\r\nPlease
restart the application."
72 AE 0A 00 70 ldstr "Registration - "
02          ldarg.0
7B 39 00 00 04 ldfld string a.A::I
28 10 00 00 0A call string string::Concat(string, string)
28 E3 1F 00 06 call System.Windows.Forms.DialogResult
System.Windows.Forms.MessageBox::Show(System.Windows.Forms.IWin32Window, string, string)
2A          ret
```

Shortly, the bold instructions are changed:

1. I used as first argument the instruction ldnull, which is a null pointer
2. re-used the same code already used by the application: given that the strings are called as references, as said above, changing one or creating one might be difficult.
3. I changed the instruction ldflld giving it the correct argument. I found it looking with IDA the opcodes of the original a.a.A(String), where a.a.:l was used as argument of same MSIL instruction.
4. I then filled with nops to keep the offsets unchanged and added a final ret at the end (filling nops must be before the final ret instruction)

Translated in pure bytes the result is:

Old bytes: 03 6F 97 00 00 0A 10 01 03 6F 7C 04 00 0A 10 01 03 02 7B 39 00 00 04 28 11 00 00 0A 2A
 New bytes: 14 72 E9 09 00 70 72 AE 0A 00 70 02 7B 39 00 00 04 28 10 00 00 0A 28 E3 1F 00 06 00 2A

Take your favourite Hex-Editor and change them.

Beware that in order to check the correctness of your changes, instead of running the program, it's safer to open the program into Reflector and see if it's able to decompile to C# the new code³. If the new program looks how you planned it to be, your program is ready for the real test.

In our cases once having done the above modifications this is how the a.a.A(String) method looks:

```
public bool A(string A_0)
{
    return (bool) MessageBox.Show(null, "Thank you for your Registration! We hope you enjoy our software!\nPlease restart the application.", "Registration - " + this.I);
}
```

Run the program, the result is shown in Figure 35.

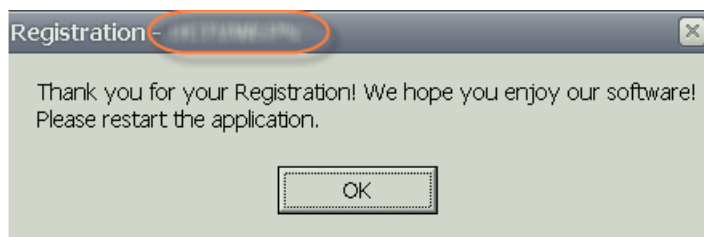


Figure 35 - MessageBox with your real serial number

Well done, your first .NET Oraculum is ready!

3.3. Reversing the Version B

The new version of the application cannot be handled by Reflector; you have to use IDA. Read the following sentence I took from a document (don't remember which one, sorry).

There are two basic types of MSIL (Microsoft intermediate language) disassemblers. ILDASM, Reflector and the like use the Reflection API to disassemble a .NET assembly. IDA does not use the reflection API, instead it examines the bytes of the .NET assembly to disassemble. The distinction between the two methods above is useful because many of the current crops of .NET protectors have the ability to stop disassembly via the reflection API. This is achieved by placing invalid meta data in the CLR (Common Language Runtime) header of the file. This meta data is ignored during the execution of the program so it runs fine.

This is exactly the protection that has been placed inside the new version of the program. I must say that this trick is nowadays not so spread because it's now public domain and easily bypassed.

Indeed I must correct myself: latest Reflector is able to decompile also this program, previous versions were not able. Good, but I would keep on with the hypothesis the Reflector is not able to decompile the program, so as to be able to show you how IDA manages .NET code.

³ You have to remove the previous version from the list of already loaded modules, using CTRL-F4 and then drag and drop the new one

3.3.1 Decompiling the program with IDA

Open IDA, give the program Version B and wait a few before the analysis is completed. I will show you two ways to patch the application, one is worst the other is equal to that of Version A, but are useful indeed to show how to do different patches to the application, to skip nags and so on. I think the educative content is important in either ways.

Now, search using ALT-T the string "Invalid Registration Code" the first occurrence found is the correct one, like shown in Figure 36.

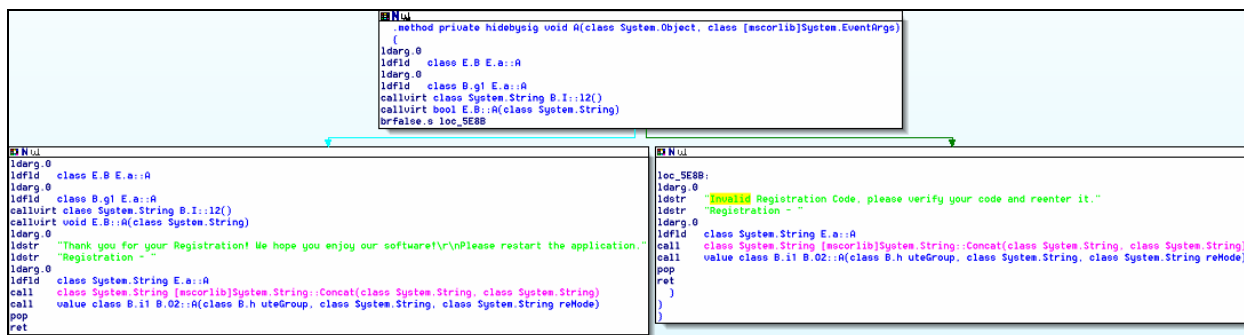


Figure 36 - Registration routine

The same code is the following:

```
.method private hidebysig void A(class System.Object, class [mscorlib]System.EventArgs)
// DATA XREF: sub_5B20+A3 r
{
    ldarg.0
    ldflld class E.B E.a::A
    ldarg.0
    ldflld class B.g1 E.a::A
    callvirt class System.String B.I::l2()
    callvirt bool E.B::A(class System.String)
    brfalse.s loc_5E8B
    ldarg.0
    ldflld class E.B E.a::A
    ldarg.0
    ldflld class B.g1 E.a::A
    callvirt class System.String B.I::l2()
    callvirt void E.B::A(class System.String)
    ldarg.0
    ldstr "Thank you for your Registration! We hope you enjoy our software!\r\nPlease restart the application."
    ldstr "Registration - "
    ldarg.0
    ldflld class System.String E.a::A
    call class System.String [mscorlib]System.String::Concat(class System.String, class System.String)
    call value class B.i1 B.O2::A(class B.h uteGroup, class System.String, class System.String reMode)
    pop
    ret
}

loc_5E8B:
// CODE XREF: sub_5E40+16 j
    ldarg.0
    ldstr "Invalid Registration Code, please verify your code and reenter it."
    ldstr "Registration - "
    ldarg.0
    ldflld class System.String E.a::A
    call class System.String [mscorlib]System.String::Concat(class System.String, class System.String)
    call value class B.i1 B.O2::A(class B.h uteGroup, class System.String, class System.String reMode)
    pop
    ret
}
```

The MSIL instruction **brfalse.s** exactly does what we need: we need to invert the jump, so transform into a **brtrue.s**, which accepts any wrong serial number. Going on with the opcodes like before, I then obtain:

```
brfalse.s   loc_5E8B           2C <ARGS>
ldarg.0     02
ldfld       class E.B E.a::A   7B <ARGS>
ldarg.0     02
ldfld       class B.g1 E.a::A   7B <ARGS>
callvirt    class System.String B.I::l2() 6F <ARGS>
```

Then what is required is to change **brfalse.s** (0x2C) with **brtrue.s** (0x2D).

Now we have also to patch the time limit which appears after 30 days of trial after the installation date.

Search as before the string "Your trial period has expired"

You will find the code of Figure 37, here reported too:

```
loc_57DF:                                     // CODE XREF: sub_5710+7D j
                                                // sub_5710+8B j ...
ldloc.2
brfalse.s   loc_5858
ldarg.0
ldfld       class B.K3 E.A::A
callvirt    void B.K3::a()
ldarg.0
callvirt    void E.A::x1()
ldarg.0
ldstr       "Your trial period has expired. Please register this application to continue
using it."
ldstr       "Registration - "
...
```

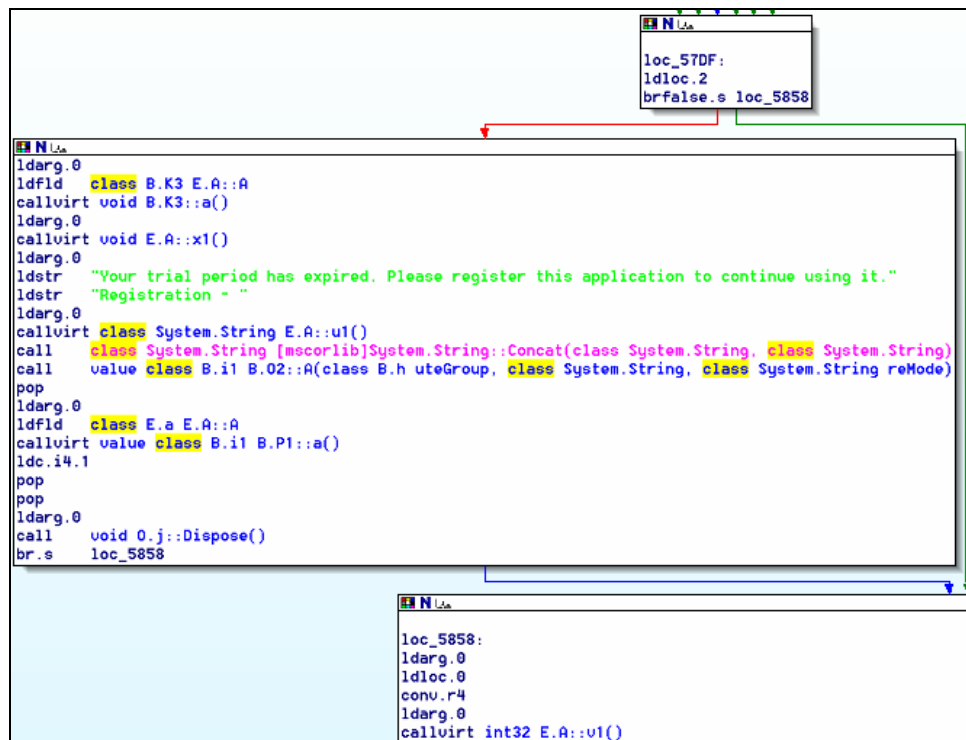


Figure 37 - Expired code

As you can see if you are using IDA to follow this tutorial there are several references to this label

```
Up j sub_5710+7D brfalse.s   loc_57DF
Up j sub_5710+8B brfalse.s   loc_57DF
Up j sub_5710+9F brfalse.s   loc_57DF
Up j sub_5710+A3 ble.s       loc_57DF
```

```
Up j sub_5710+AC bgt.s loc_57DF
```

What we need is to NOP all these jumps (MSIL nop is equal to 00), as shown in Figure 38,

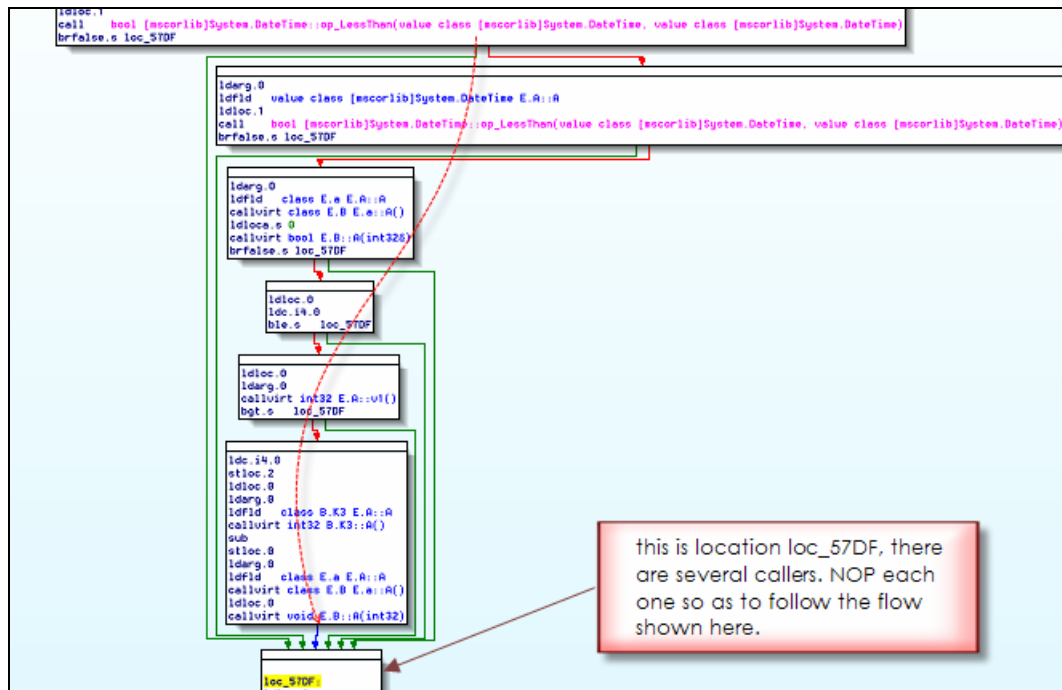


Figure 38 - Patch of time limit check routine

For example the first, at offset 6539 has the opcode 2C 50 becomes 00 00 and so on till you did all.

Note: you can also use the program <http://www.crackmes.de/opencrackmes/Collectionists/BiW-Reversing/unsorted/dexil.zip> but doesn't cover all the MSIL instructions.

But now we must do a serious patch. This one is too bad, because patches the checks done by the program and not the source of these checks. The rule as always is to patch at the deepest level possible, you know..

Search with IDA "Register" and stop at the first result, we will find the place where the menu "Register" is created.

We find what follows:

```
ldstr    "Register"
callvirt void B.m2::A(class System.String )
ldarg.0
ldfld    class B.m2 b.A::C
ldarg.0
ldftn    void b.A::h(class System.Object , class [mscorlib]System.EventArgs )
newobj   void [mscorlib]System.EventHandler::ctor(class System.Object, native int)
```

Remember that the MSIL always is based on a stack logic so all the operator are placed on the stack before calling an API. The newobj instruction creates a menuhandler associated to the "Register" and the menu handler is b.A::h().

Follow the method b.A::h().

```
.method private hidebySig void h(class System.Object , class [mscorlib]System.EventArgs )
// DATA XREF: A+565 r
{
    .locals (class System.String V0)
    ldarg.0
```

```

    ldflld    class E.a E.A::A
    callvirt  value class B.i1 B.P1::a()
    ldc.i4.1
    bne.un.s  loc_1A5C
    ldarg.0
    ldflld    class E.a E.A::A
    callvirt  class E.B E.a::A()
    ldloca.s  0
    callvirt  bool E.B::A(class System.String& [out] )
    brfalse.s loc_1A5C
    ldarg.0
    ldflld    class E.a E.A::A
    callvirt  class E.B E.a::A()
    ldloc.0
    callvirt  bool E.B::A(class System.String )
    brfalse.s loc_1A5C
    ldarg.0
    ldflld    class B.m2 b.A::C
    ldc.i4.0
    callvirt  void B.m2::a(bool )
    ldarg.0
    ldstr     "High Stakes Pool Aimer v0.9.1"
    callvirt  void B.I::M2(class System.String )

loc_1A5C:                                     // CODE XREF: h+C j h+20 j ...
    ret
}

```

This function actually decides if to draw on not the menu "Register", like we have already seen disassembling Version A.

With IDA we can browse as well double clicking on the function name and then we finally land here:

```

.method public hidebysig bool A(class System.String) // CODE XREF: h+2E p
// sub_56A0+50 p ...
{
    ldarg.1
    callvirt class System.String [mscorlib]System.String::Trim()
    starg.s 1
    ldarg.1
    callvirt class System.String [mscorlib]System.String::ToUpper()
    starg.s 1
    ldarg.1
    ldarg.0
    ldflld    class System.String E.B::I
    call      bool [mscorlib]System.String::op_Equality(class System.String, class
System.String)
    ret
}

```

It's the function we found before for Version A; we can then do a similar patch!

```

.method public hidebysig bool A(class System.String ) // CODE XREF: h+2E p
// sub_56A0+50 p ...
{
    ldarg.1
    callvirt class System.String [mscorlib]System.String::Trim()
    starg.s 1
    ldarg.1
    callvirt class System.String [mscorlib]System.String::ToUpper()
    starg.s 1
    ldarg.1
    ldarg.1
    call      bool [mscorlib]System.String::op_Equality(class System.String, class
System.String)
    nop
    ret
}

```

The patch is then the following

Original code:
03 ldarg.1

```

02          ldarg.0
7B C2000004  ldfld    class System.String E.B::I
28 0B00000A  call     bool [mscorlib]System.String::op_Equality(class System.String, class
System.String)
2A          ret

```

New code:

```

03          ldarg.1
03          ldarg.0
28 0B00000A  call     bool [mscorlib]System.String::op_Equality(class System.String, class
System.String)
00 00000000  nops
2A          ret

```

Almost the same patch as before.

Original bytes: [offset: 6D56] 02 7B C2000004 28 0B00000A 2A

New bytes: [offset: 6D56] 03 28 0B00000A 00 00000000 2A

3.4. Final Remarks

I hope to have shown that there's another alternative to patch .NET applications, which doesn't involve decompilation and recompilation of the whole application. Sometimes it is faster, especially for applications not easily re-compilable and for easy protections. Furthermore IDA has a unique interesting feature when it comes to .NET applications: has its own decompilation engine, not using reflections .NET APIs, which is an excellent feature when you have to understand obfuscated code.

To exercise more try this nice Crackme: <http://www.crackmes.de/users/crackersixx/cscrackme/>

With this first primer we have not touched advanced protection techniques and obfuscations currently available with .NET, neither strong names; you shouldn't get the sensation that reversing .NET application is simple, or simpler than normal Win32 executables, neither believe that would be simpler than reversing classical compiled programs (ASM). Just on the one hand most developers still have to get accustomed and trained about advanced .NET protection features, and on the other hand .NET commercial applications (not web) are not too widespread, things will change...

4. Serial Fishing in .NET (Live Debugging), zyzygy

4.1. Tools

These the basic tools that will be used

1. PEBrowser Pro Interactive (www.smidgeonsoft.com), which is able to debug .NET applications!
2. Crackme 1 (included in the zip file, framework v1.1)
3. Basic Fundamentals of .NET Framework and IL.
4. Lutz Reflector

4.2. Essay

This article aims to throw some light on Live Debugging the .NET Framework. The scenario is similar to the Win32 environment.

For those who don't know, a .NET Application (called assembly) is always compiled at runtime and only those codes are compiled that need to be executed, thereby avoiding unnecessary memory consumption.

I have coded a very simple Crackme, to explain my idea. I have also applied this technique to real software for verification and it works. We shall be working directly with the x86 opcodes during the debugging session.

Let's start.

We need to know the function/method that is responsible to calculate the serial key. So we load this Crackme in the Reflector and see what it has in store for us.

Load up the application in Reflector. We directly corner the function responsible of the serial calculation.



Figure 39 - Locating the function for serial generation.

Well we have the serial calculation algorithm right over here. Our task is to serial fish. And since .NET makes our lives easier by giving the entire code away (remember that this application is not obfuscated), we only need to know the location where the final serial is being generated. Glancing at the image I guess it is not so tough to locate it.

We now have the information we need. We shall set a breakpoint on the last arithmetic operation on *num2* in the *btnCheck_Click* function. And then the serial will be ours. ☺

Start PeBrowserPro Interactive. And make the following settings. Some of them would be default; I got them from the site, so you may get it from there as well (Figure 40).

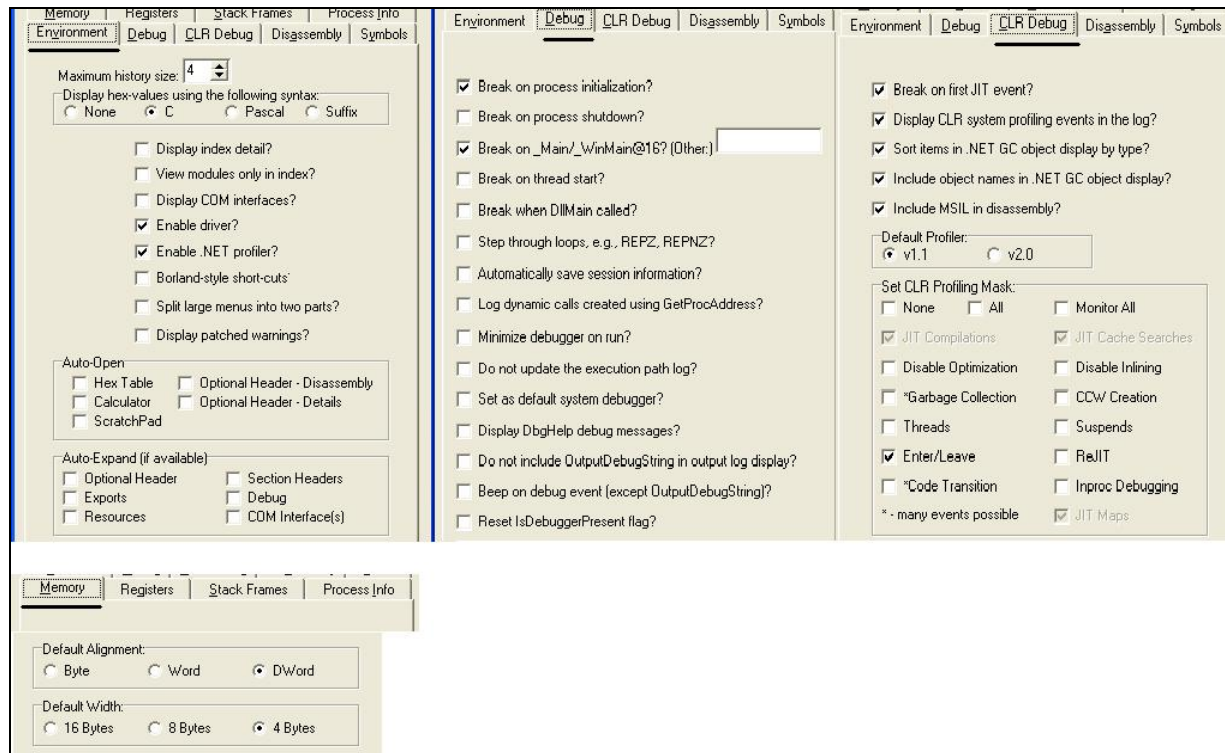


Figure 40 - Settings for the debugger

Once you are done with the settings, load the executable by pressing Ctrl + S / Start Debugging.

As soon as you press Ok, you see windows cluttering up. Well not to fear. They all make sense.

Click on our executable name: Crackme1 and then select the .NET Methods. There you shall see btnCheck_Click function. Open it and you will see the function in IL.

Now is the time when your IL knowledge comes in handy. For those who are unaware of it, I suggest you get a basic idea of it and then return (<http://msdn2.microsoft.com/en-us/library/812xyxy2.aspx>, opcode listing).

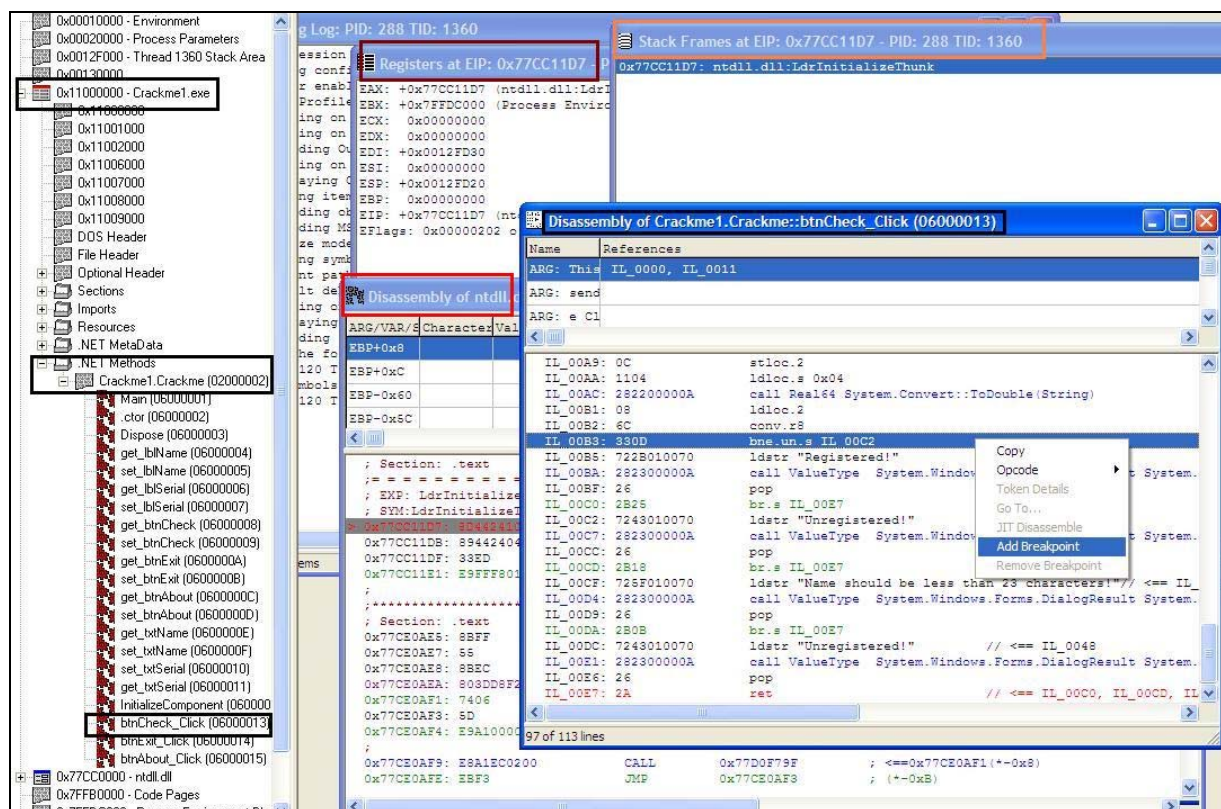


Figure 41 - Locating the code to set the breakpoint on.

I have tried to put as much information in this image as possible.

- The disassembly window (black highlight) that you see is the IL form of the function.
- The disassembly window (red highlight) is the assembly form. We shall be debugging in that window. You see this because you had selected the option – *Break on process initialization*.
- The stack frame window (orange highlight) is similar to the call stack.
- The register window (brown highlight) is the register window where you see/edit the values of the registers.

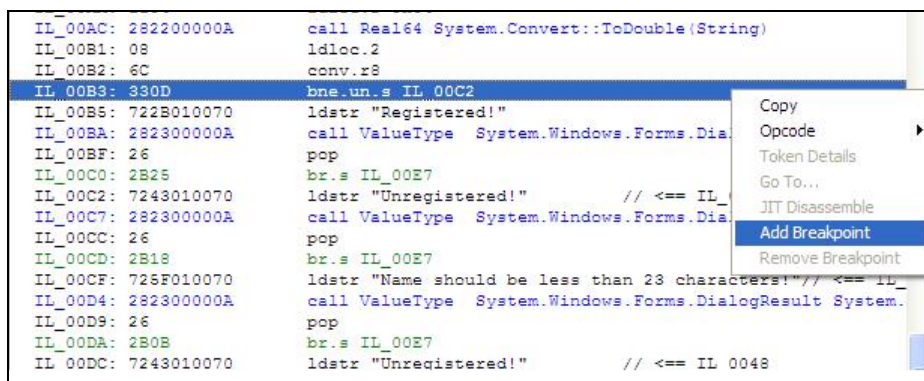


Figure 42 - Setting a breakpoint at the location where serial is compared.

We will set a breakpoint on the code as shown in the image because it is the location where the serial is being compared.

Note: I think there is a bug in PeBrowserPro. Sometimes the breakpoint doesn't get set though you set it. So confirm by right clicking on that line.

We are ready to roll!!!

The debugging keys are that of Microsoft Visual Studio. You may even change it from the configure option.

After you have set the breakpoint, press F5 and it will break here:

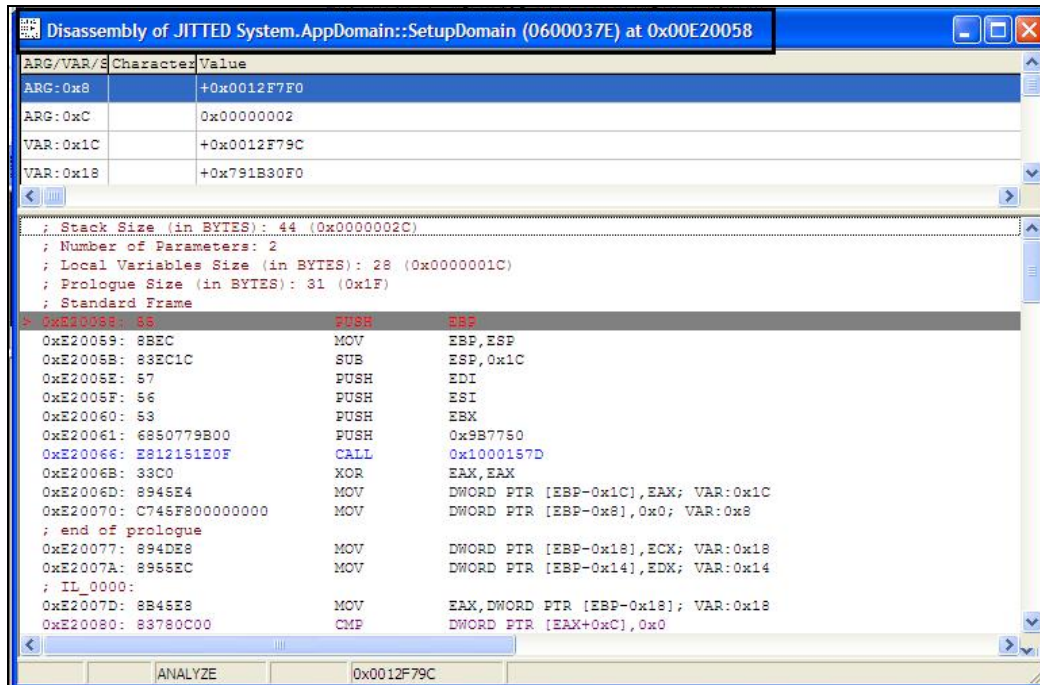


Figure 43 - Breakpoint at AppDomain.

You see this because you selected the option – *Break on First JIT event*

It breaks here because whenever a .NET Application is executed, it has to register itself with an AppDomain. What this means is that the execution is restricted/isolated to a memory, thereby minimizing the after effects of a massive crash.

Next it breaks on the Main() function of our executable. This is similar to the constructor (form_load). The next time you press F5 again, the application will load.

Enter your name and serial and then click *Check*. Upon clicking check, the application will break at the point we set the breakpoint on. How do we ascertain this fact? Right click on the line, where the execution has halted and select MSIL disassemble and you will be presented with the equivalent IL code. If you notice there will be a '>' sign indicating the currently executing code in the MSIL window. As you execute the pointer will move accordingly.

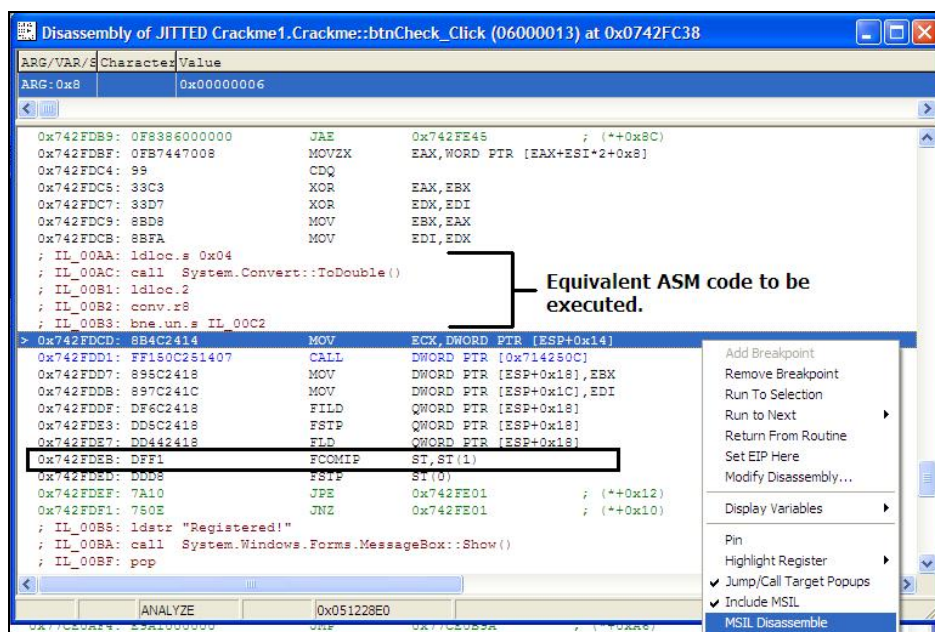


Figure 44 - Breakpoint at serial compare.

What we have here are the common assembly instructions so I shall leave them for you to interpret. We shall trace till 0X742FDEB and then look up in the register window. Then in the register window right click and select the floating point registers and you will see the code you entered and the correct code.

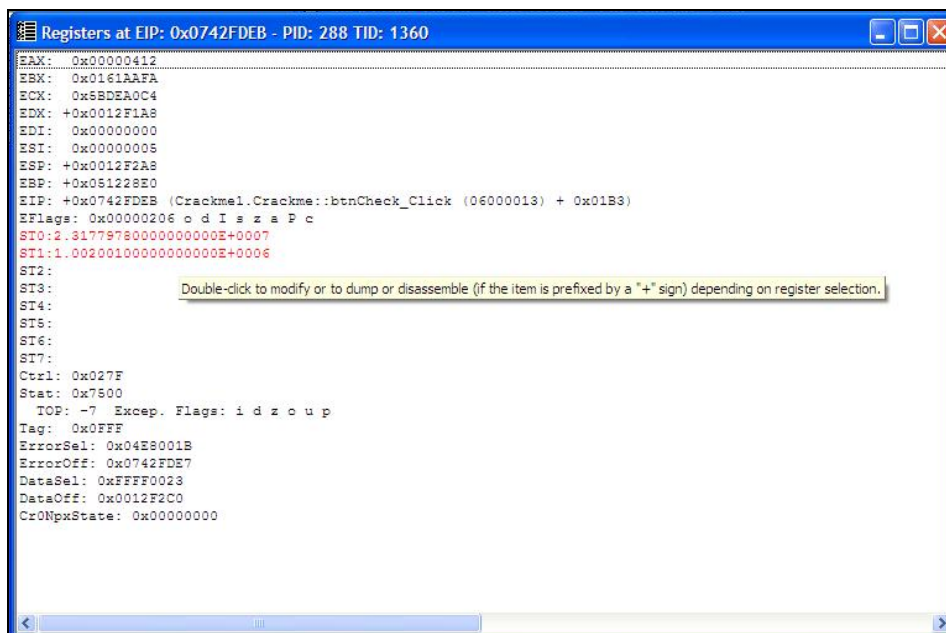


Figure 45 - Register Window

Now that you know the real code, enter it.

Well there you did it. Live debugged a .NET application. Note that I had made this simple but then with a .NET application all the details are given away as it is. For a complex algorithm all you need to figure is the right place to set breakpoints on to fetch the values.

5. Further readings

- Strong Names: here are some other good documents. For what about Strong Names, you can read [4, 5, 6, 7] to get used to the terms and concepts, then you can follow 3 approaches: decompilation and removing [8] byte level patching [9], automated tool [10]. Document [11, 12] is also interesting to see some protection measures starting to appear (not involving obfuscation or packing).
- MSIL: despite reading the ECMA standard, which is not meant to be a tutorial, it's more interesting to read the series of document about demystifying the Microsoft Intermediate Language (MSIL) [14, 15, 16].
- Cracking code: there are plenty of tutorials around the known forums, but here are some interesting tutorials I found [17, 18, 19, 20, 21, 22].

5.1. References

- [1] Table 1: Opcode Encodings" available at this address, <http://dotnet.di.unipi.it/EcmaSpec/PartitionIII/cont1.html#Table1OpcodeEncodings>
- [2] ECMA Standard 335, *Common Language Infrastructure (CLI) Partitions I to VI*, 4th Edition (2006), <http://www.ecma-international.org/publications/files/ecma-st/ECMA-335.pdf> also available offline at <http://dotnet.di.unipi.it/ecmaspec/ecmaspecs.zip>
- [3] ECMA-335 documentation page, <http://www.ecma-international.org/publications/standards/Ecma-335.htm>
- [4] Strong Names Explained, <http://www.codeproject.com/dotnet/StrongNameExplained.asp>
- [5] Understanding .NET Code Access Security, http://www.codeproject.com/dotnet/UB_CAS_NET.asp
- [6] Building Security Awareness in .NET Assemblies : Part 1 - Learn to break a .NET Assembly, <http://www.codeproject.com/dotnet/NeCoder01.asp>
- [7] Building Security Awareness in .NET Assemblies : Part 2 - Learn to protect your .NET assemblies from being tampered, <http://www.codeproject.com/dotnet/NeCoder02.asp>
- [8] Building Security Awareness in .NET Assemblies : Part 3 - Learn to break Strong Name .NET Assemblies, <http://www.codeproject.com/dotnet/NeCoder03.asp>
- [9] Removing strong-signing from assemblies at file level (byte patching), <http://www.codeproject.com/dotnet/StrongNameRemove20.asp>
- [10] SNRemove Tool, http://www.nirsoft.net/dot_net_tools/strong_name_remove.html
- [11] Securing .NET Assemblies, <http://www.codeproject.com/dotnet/AssemblyTest.asp>
- [12] Securing Managed Assemblies with Native EXE Interoperability, http://www.codeguru.com/Csharp/Csharp/cs_misc/security/article.php/c8309/
- [13] The .NET File Format, <http://pmode.net/USERS/116/Files/dotnetformat.htm>
- [14] Demystifying Microsoft Intermediate Language. Part 1 – Introduction, http://www.devcity.net/Articles/54/msil_1_intro.aspx
- [15] Demystifying Microsoft Intermediate Language. Part 2 - A Short Description of .NET Application, http://www.devcity.net/Articles/55/msil_2_dotnet.aspx
- [16] Demystifying Microsoft Intermediate Language. Part 3 – Debugging, http://www.devcity.net/Articles/57/msil_3_debug.aspx
- [17] Cracking code - Introduction, <http://www.atrevido.net/blog/PermaLink.aspx?guid=73204548-5970-46db-b7cf-76cd4c22c3b9>
- [18] Cracking code - Part 1, <http://www.atrevido.net/blog/PermaLink.aspx?guid=ec99e239-8917-48e3-bd4f-af866b730150>
- [19] Cracking code - Part 2: Other simple attacks, <http://www.atrevido.net/blog/PermaLink.aspx?guid=f2b7825e-1a8b-4beb-adf0-215011fd89e0>
- [20] Cracking code 3: Cracking an obfuscated .NET assembly, <http://www.atrevido.net/blog/PermaLink.aspx?guid=8315fa01-0286-47ce-a20b-fcc15eb297c3>
- [21] Cracking Code 4: Replacing a strong name, <http://www.atrevido.net/blog/PermaLink.aspx?guid=f772c18a-f389-4c28-bd6a-a30f4ccc84f5>
- [22] Cracking code 5.1: Increasing your configuration, <http://www.atrevido.net/blog/PermaLink.aspx?guid=92b5d25e-e53a-459c-b2c1-77aa26544880>
- [23] RemoteSoft Salamander, <http://www.remotesoft.com/>

Document History

- Version 1.0 first public release
- Version 1.1 added a full commented References list
- Version 1.2 added "Serial Fishing in .NET (Live Debugging)" by zyzygy

