

Reversing .NET

Part I – An introduction

By Kwazy Webbit [RETeam]
October, 2005

Introduction

Every few years, something new hits the streets. The latest craze is .NET. Microsoft's been plugging it like crazy, essentially claiming it to be the best thing since sliced bread. Some programmers have started to develop in it, which in turn means reverse engineers have been wondering about what goes on underneath the hood. So far there has been a lack of good information on the subject, and this paper is meant to fill that void. It is part of a joint project between #Cracking4Newbies (on EFNet), and the Reverse Engineering Team (<http://www.reteam.org>). This is the first part of a series, relating to an increasingly difficult series of ReverseMe's created by x-Bi0dESC. Perhaps I will be writing more of them myself, and perhaps I will only be on the sidelines, watching other people in C4N join in on the project. We'll see...

What is .NET, really?

To most programmers, the concept of .NET is still something of an enigma. It's been one of Microsoft's latest buzzwords, and it has been used in products ranging from ASP.NET to Visual Studio .NET. The aim of this paper is to demystify the concept as much as possible, so that it is no longer regarded by Reverse Engineers as a black box.

The basic concept of .NET is quite simple. Many of you may already be familiar with Java and it's JVM¹, which are similar in functionality to .NET in many ways. A program made for the .NET framework is not compiled directly to machine code (as is the case with most traditional languages, e.g. C++), but is instead compiled to an Intermediate Language (from here on known as IL). This IL is comparable to the bytecode Java programs are compiled to. Doing this has several advantages from a programming perspective (although it gives a small penalty to the execution speed). Java's reason for doing so is to allow a Java program to run on different Operating Systems and even different processors. This was not the main goal for .NET, but they chose the same design approach.

The main advantage of IL for .NET programmers is the fact that identifiers² still exist in the compiled program. This allows a programmer to make different parts of an application in different languages, while still keeping them able to see eachothers classes and functions, to use them appropriately.

A major advantage of IL to reverse engineers is the exact same thing. Because this means that a .NET program is only a bytecode representation of its source, while still having all it's identifiers intact. Also, because IL is a somewhat higher level code than real processor code, high level language structures can easily be identified (e.g. 'if-then-else' constructs). Knowing this, tools can be developed to reconstruct a .NET program back to a .NET source language **of your choice**.

A good tool to do just this has already been developed. It is called Reflector³, and was made by Lutz Roeder. It will allow you to load a .NET executable, and then select a target language which it should decompile to. In this introductory paper, I will show you how to reverse a simple practice crackme using Reflector. Note that this introduction is just that, it does not show how to break a realistic protection, instead it will merely show you the very basics of how to deal with a compiled .NET program.

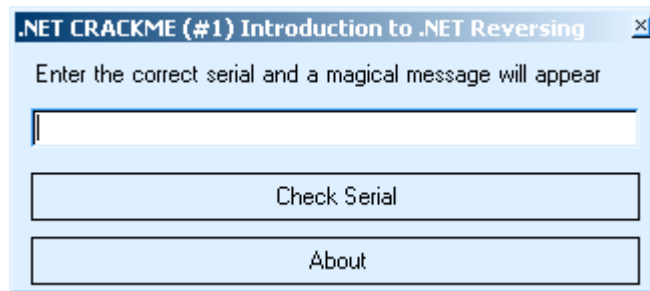
1 Java Virtual Machine

2 Such as class names, function names, and variable names

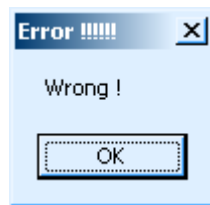
3 <http://www.aisto.com/roeder/dotnet/>

A concrete example

A nice small example program made by x-Bi0dESC for this paper is ideal for showing some basic concepts about reverse engineering a .NET file. As with most targets, it is good to first simply run the program. This might show us what to look for when we are looking from 'the other side'. It looks something like this:



It displays a message box when you have entered the wrong serial:

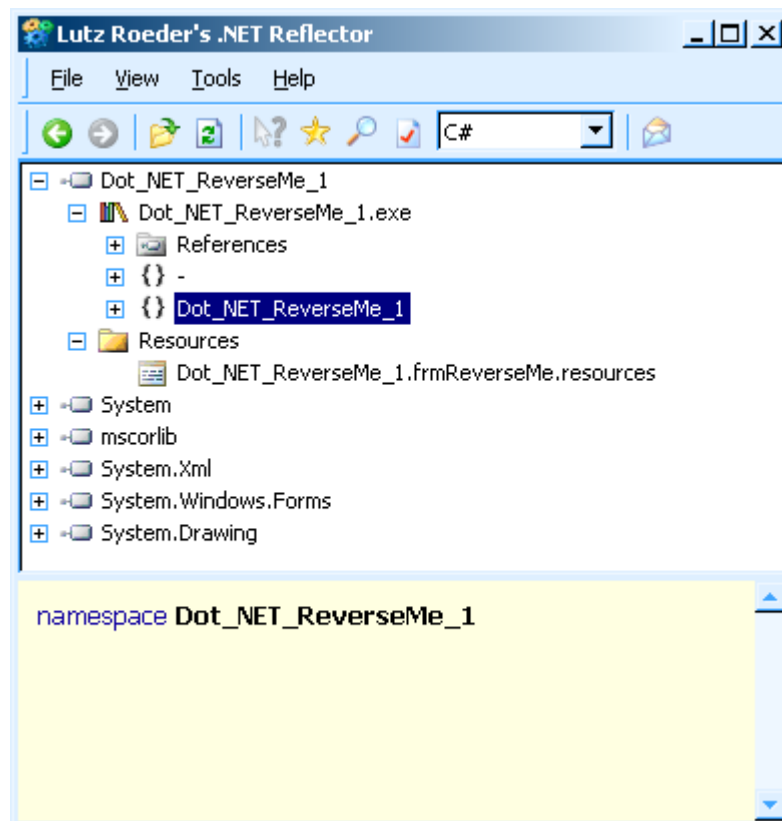


This completes our quick overview from the program's **outside**.. Now let's see what it is like on the **inside**⁴...

4 Figured out the reason the program is called 'Reflector' yet?

A moment of reflection

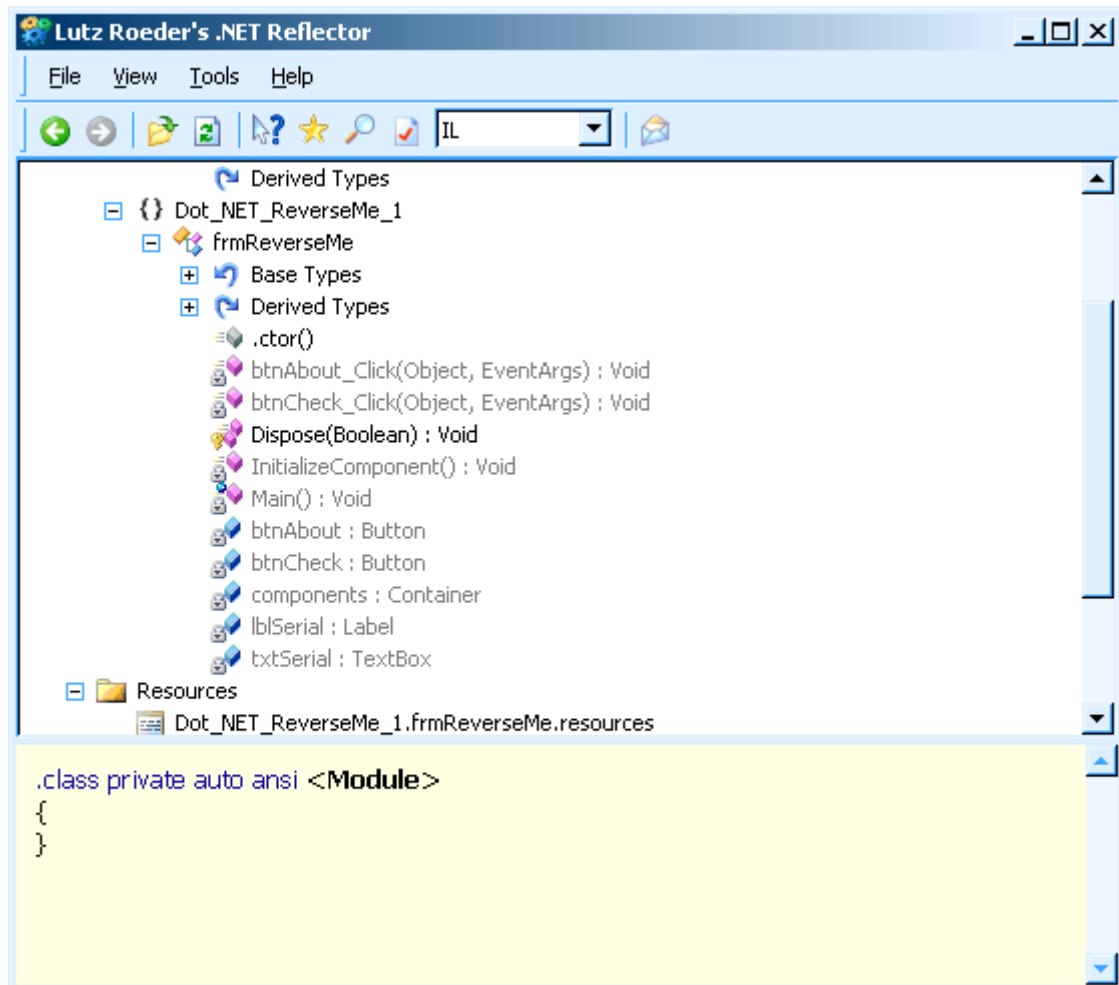
Let's open up reflector, our tool for this task. You will see the main window. Let's select C# as the main language (the dropdown box of languages is in the main toolbar), since most of us come from a C/C++/Java/C# background and find that easiest to read. If you prefer something like Delphi, VB or even ILAsm the decision is of course up to you. The next step is to open up the target program through the File->Open menu, in this case Dot_NET_ReverseMe_1.exe by x-Bi0dESC. Once you have opened it, your main window will look like this:



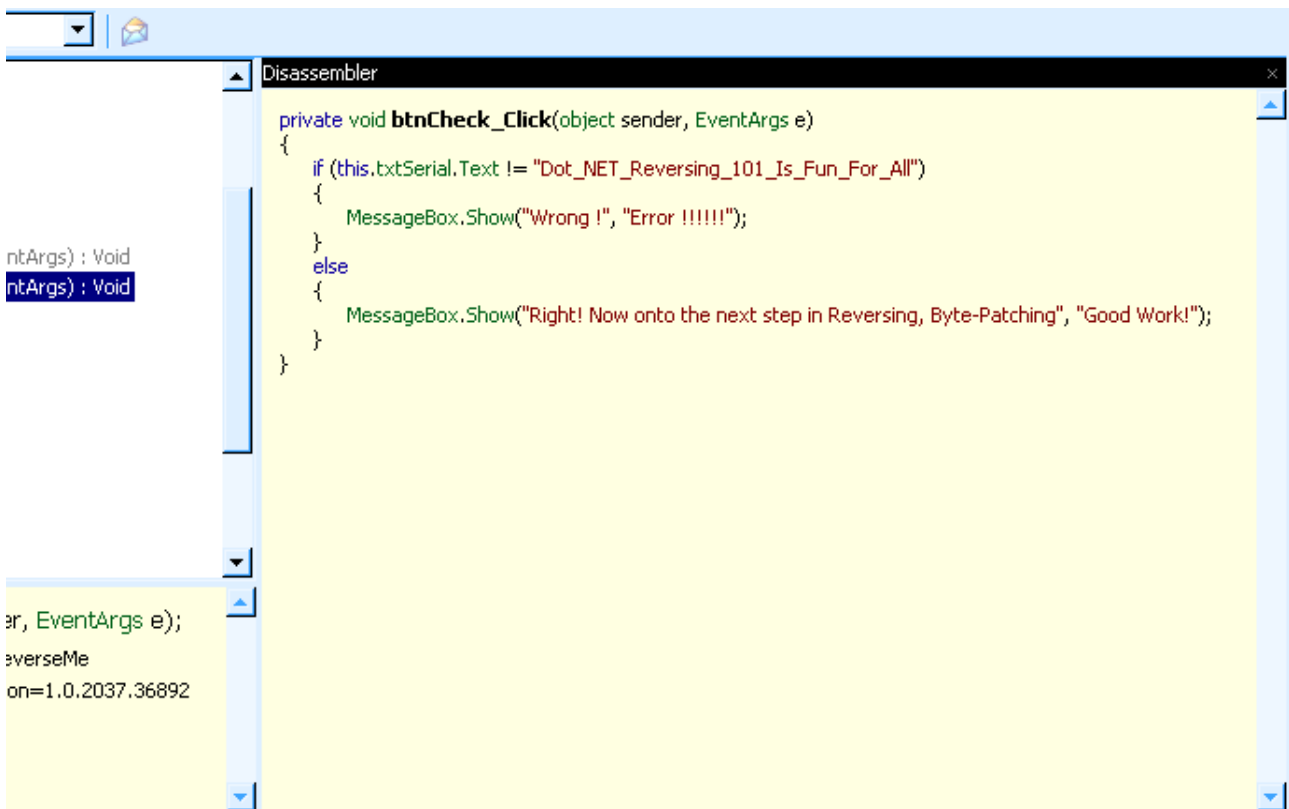
The target program has been analyzed and placed in a tree structure, allowing you to open the parts you are currently interested in, one at a time. In our case we are interested in the ReverseMe, which is already expanded in the screenshot above. As you can see, it opens up into code and resources. The resources are similar to the ones in other windows programs, and irrelevant to us at this point. The code part consists of 'References', which are similar to the 'imports' used in regular PE files. All we are really interested in at the moment, however, is the code in Dot_NET_ReverseMe_1. Let's see what code is really inside our target!

The code

Opening the code section will lead us into the following tree:

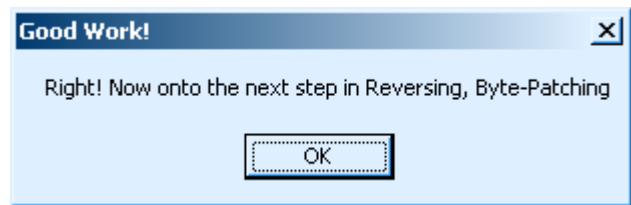


Reflector has detected a form (a dialog) called `frmReverseme` and detected several GUI elements and functions in it. Here you can clearly see that all the functions and GUI elements still have recognizable names, making it much easier for us to understand what they're for. We can see functions like `'btnAbout_Click(...)'` which presumably is what is being called when the `'btnAbout'` button has been clicked, and the `'btnCheck_Click(...)'` which is called when the `'btnCheck'` button has been clicked. The latter one is, of course, the one we are interested in. We want to know what the program does once we press that `'check'` button. Luckily, this can be decompiled into the source code of our choice (C# in my case). This is done by double-clicking on the function name. Reflector will show the disassembly in a new part of the window, in the language you chose. In this example:



```
private void btnCheck_Click(object sender, EventArgs e)
{
    if (this.txtSerial.Text != "Dot_NET_Reversing_101_Is_Fun_For_All")
    {
        MessageBox.Show("Wrong !", "Error !!!!!");
    }
    else
    {
        MessageBox.Show("Right! Now onto the next step in Reversing, Byte-Patching", "Good Work!");
    }
}
```

Here you can see, clear as day, what the function does. It will also easily allow us to defeat the protection in this program. I'm sure we can all agree that it doesn't take an experienced C# programmer to see that the serial of this program should be "Dot_NET_Reversing_101_Is_Fun_For_All". If we try this in the running program, it will show a new message box saying:



Which will probably be the next part of this course about Reversing .NET.

Summary

With this new (full) rewrite of this essay, I decided not to flood you with endless information about how .NET really works. Instead, I chose to only give you a quick summary, showing you a few things that would really matter while reversing. Hopefully this has helped you gain a little more understanding of how .NET works, as well as some information about how to deal with a .NET protection. The next papers in this series will surely teach you more in-depth knowledge than I have in this introductory chapter, but no man can stand firmly without a solid base.