Reversing .NET

Part II – Byte Patching

By Kwazy Webbit [RETeam]
October, 2005

Introduction

Having covered the basics in the previous chapter, we now know how to look inside a .NET program. This is all we need to do for some very basic protections, like the hardcoded serial crackme we looked at. A lot of times, however, we will need to do more than just look. We will need to change the program as well. This is known as patching (as you might well know), and is useful for both cracking and other reverse engineering (adding functionality for example). There are some difficulties in doing this, namely finding the right place in the file and figuring out what bytes to put there. The target for this chapter has once again been made by x-Bi0dESC and can be found in the attachment that comes with this paper.

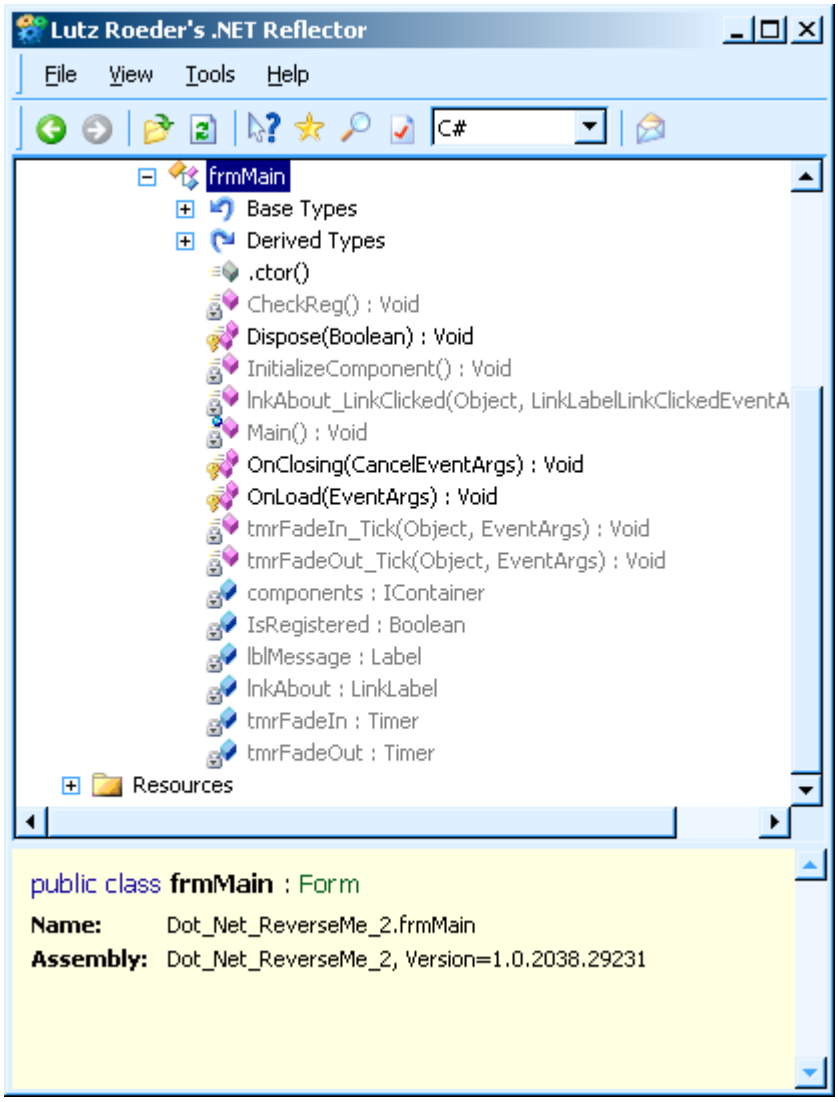
The outside

Let's run the program to see what we're dealing with. We see a window, with a simple message:

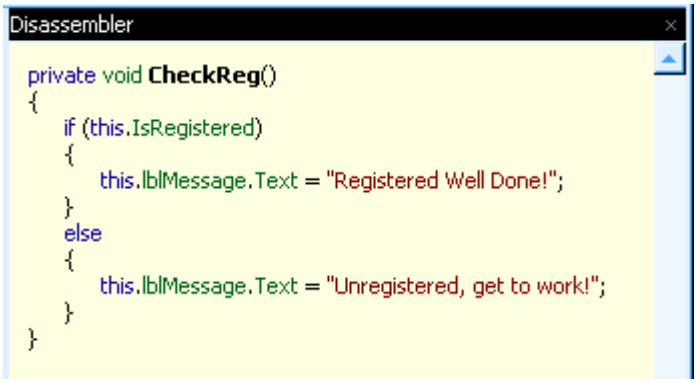
As you can see there is no button to re-check, and no field to enter any information. Not very interesting to us, so let's move right along to the interesting part, the inside...

The inside

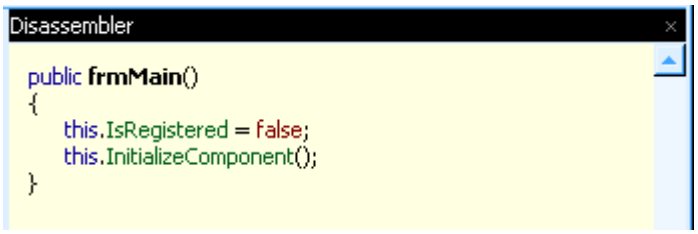
Opening the program in Reflector shows us the structure of the program:



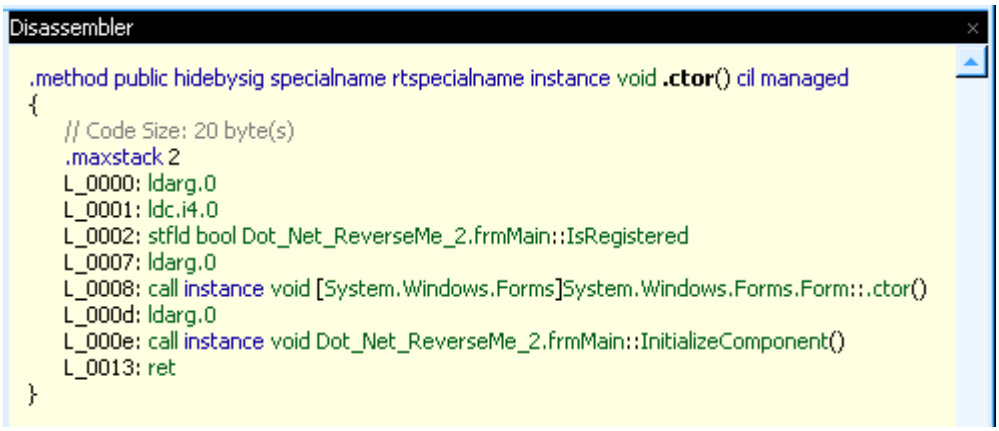
This gives us several bits of interesting information, one of which is the fact that the class contains a boolean called *isRegistered*. Another thing we see is that there is a function called *CheckReg()*. Opening the *CheckReg()* function (double-click on the function), we see that our suspicions were indeed correct:



Now it is time to take a little step aside and discuss a function who's name might not strike you as very interesting at first. Its name is *.ctor()*. As everyone knows that has done C++, Java, C#, or any other Object Oriented programming, classes have a constructor to initialize the values of its class members. In .NET the class constructor is not named after the class (in this case *frmMain*), but it is named *.ctor()* instead (an abbreviation for constructor). Knowing this makes the function a lot more interesting all of a sudden, because we have a member variable called isRegistered that determines if the program is registered. There's a good chance it is being initialized in the constructor, so let's open up *.ctor()* and see:



Indeed, there is no hidden registration method, it is just permanently disabled. We will have to change the program to make it registered. To do so, we will need to go a bit lower level, so let's start by switching to ILAsm as our language:



This is a direct representation of the internal bytecode, and this is the level we will have to work in to be able to make useful changes to the program. Internally, .NET is essentially a stack machine. For those of you not familiar with the term, this means it relies on the stack rather than using registers. For instance, moving a value from location A to B means the value from A is pushed to the stack, then it is popped from the stack into B. Other systems could for example allow you to directly transfer from A to B, or use a register for temporary storage. We will need some reference to understand the often cryptic meanings of these Assembler instructions. I personally have a copy of a book called Inside Microsoft .NET IL Assembler, which contains a listing of all the instructions and their use. For those of you who do not have a book like this, I would recommend checking out the MSDN page on this subject. Now that we all have a good instruction reference, we can start looking at the code and start forming a plan to get the program registered.

You can clearly see the heavy use of the stack in the code above. The C# line:

this.isRegistered=false;

has been translated into three stack-related lines:

ldarg.0
ldc.i4.0
stfld bool Dot_Net_ReverseMe_2.frmMain::isRegistered

Looking up these instructions in the reference, we find:

ldarg.0	Load argument 0 onto stack
ldc.i4.0	Push 0 onto the stack as **I4**.
stfld	Replace the value of *field* of the object *obj* with *val*

This essentially does (in a pseudo Object Oriented notation):

(arg0).isRegistered = 0;

To make it initialize to true, we simply need to change that to:

(arg0).isRegistered = 1;

Which means we will have to change the second instruction to

ldc.i4.1

This is really basic cracking, so I assume that for most of us this will all be very basic. The trick is however, that we are not in the standard environment. Let's look up the byte representations of these instructions in our reference. It says that *ldc.i4.0* is *0x16* in bytecode, and that *ldc.i4.1* is *0x17* in bytecode. Great, now we know what to replace. Here comes the second problem though, how do we find the spot where we want to replace code? Reflector shows us the code, but not the location in file where we would want to change a byte. I don't know of a tool currently available that will help you with this, so we'll do it the 'dirty' way instead. We will just translate a few surrounding instructions to bytecode, so that we have a long enough search string to find what we are looking for. Using the reference, we find that:

0x02	*ldarg.0*
0x16	*ldc.i4.0*
0x7D<T>	*stfld bool Dot_Net_ReverseMe_2.frmMain::isRegistered*

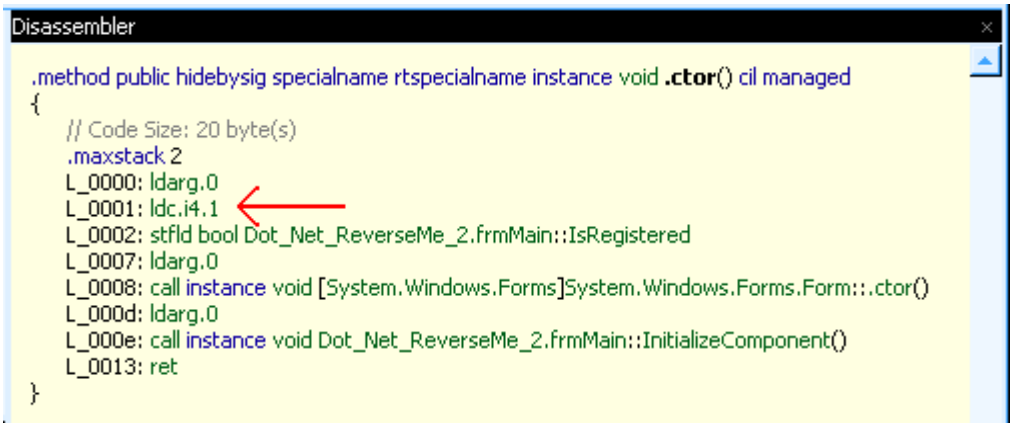
So let's do a search for *0x02167D* in your favourite hexeditor. Wonderful, only one hit! That means the search string was long enough to be unique, and we can make the changes. The offset in the file is *0x105D* so we will edit the *0x16* there and change it to our *0x17*.

Note: **Always** back up your original program before saving.

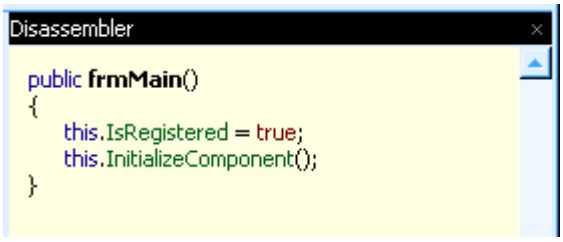
The moment of truth

Let's run the program and see if we've achieved the effect we were hoping for...

Great! We changed the program in such a way that it now thinks it's registered. The first .NET patch has been achieved! Out of curiosity let's take a peek inside again, and see what's changed. After opening the changed ReverseMe, while still in IL mode, we can see our change:



We can now leave the low level realm, and enjoy the much more human-readable C# version:



Conclusion

This is only the second chapter on reverse engineering .NET, and there are still many things unsaid. Nevertheless, you should be able to get started on your own small reverse engineering projects. The next chapter will be about more advanced patching techniques, dealing with nagscreens and disabled buttons, among others.

– Kwazy Webbit