Reversing .NET

Part III – Advanced Patching

By Kwazy Webbit [RETeam]
November, 2005

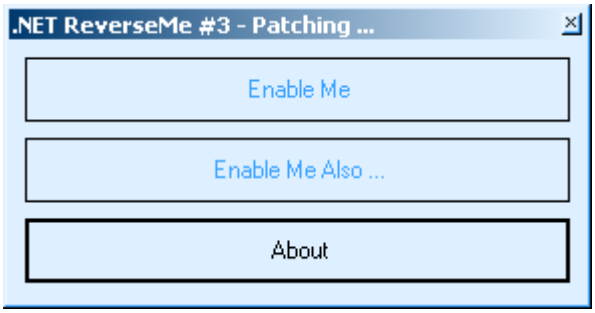
Introduction

The time has come to leave the baby steps behind and get busy with some more complicated (and thus realistic) problems. This chapter will demonstrate some more advanced patching techniques you will encounter in real situations. Our example target will be the third reverseme created by x-Bi0dESC to help with this course. Because the subject involves more advanced patching, the Reflector tool will no longer suffice (it is really not suited for use in patching, but more for understanding a program), instead we will switch to using IDA for analysis. IDA is a powerful disassembler that will give us a lot of useful information (like the location in file, the code bytes representing instructions, and a lot of other advanced analysis information ;-))

Running the program

As always, we run the program and encounter the first obstacles of the program, two nag screens:

After which we reach the main window, which contains two more protections to defeat:

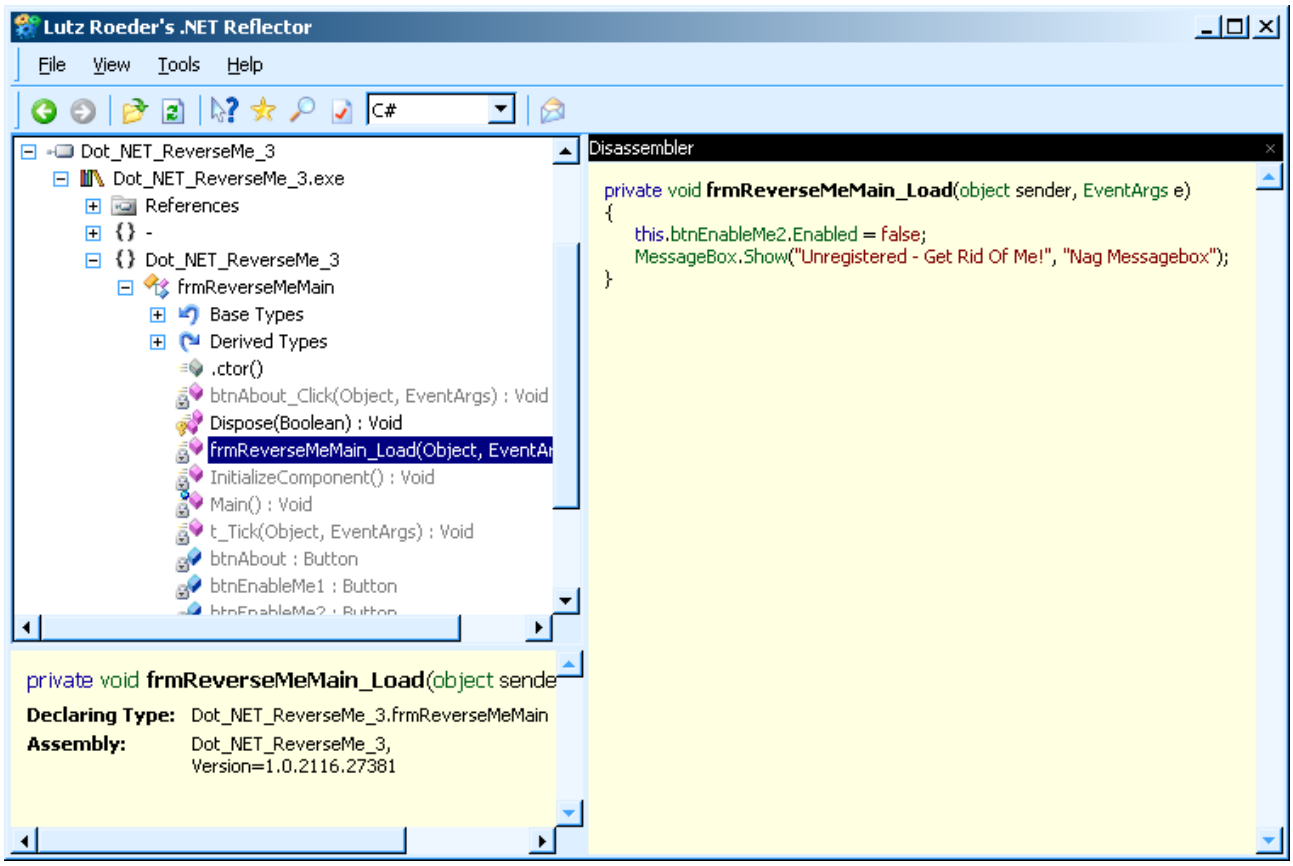


These are all different problems, with one thing in common: They all require patching of the executable, as you might have gathered given the name of this chapter. We will start by removing the Messagebox nag screen and work our way through the reverseme one step at a time.

The Messagebox

The least complex form of a nag screen is probably the messagebox. It is so simple because it usually only requires one API call to create and handle its functionality (in the standard Win32 API this function is aptly named 'MessageBox'). Let's open up the program in Reflector first, and try to

find the messagebox (I open it in Reflector first, because that supports high level syntax like C# making the code much more readable and easier to do the first analysis). The messagebox creation is fairly easy to find, so I won't bore you with a step by step walkthrough (read the first two chapters for the basics):

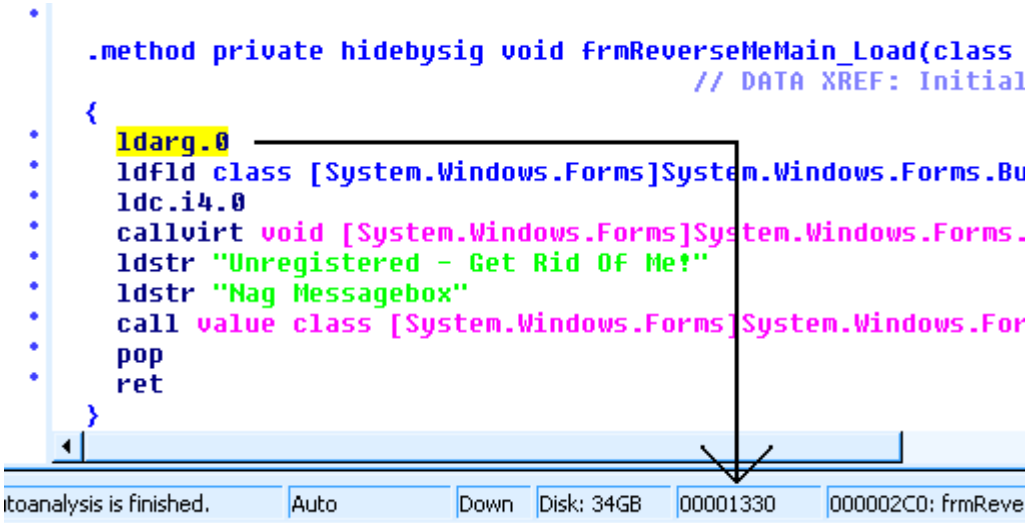


Here we find it in the '*frmReverseMeMain_Load(...)*' call, the function to create a messagebox is (in C#) apparently called '*MessageBox.Show(...)*', so we will have to remove that code. But wait, what's that line above it? It seems we have inadvertently run into another protection, disabling one of the buttons. Let's open up the program in IDA, and do some further analysis now that we know what function to look for. IDA's analysis is as follows:

```
.method private hidebysig void frmReverseMeMain_Load(class System.Object sender,
                                                     class [mscorlib]System.EventArgs e)
{
      ldarg.0
      ldfld class [System.Windows.Forms] System.Windows.Forms.Button
                        Dot_NET_ReverseMe_3.frmReverseMeMain::btnEnableMe2
      ldc.i4.0
      callvirt void [System.Windows.Forms]
                        System.Windows.Forms.Control::set_Enabled(bool)
      ldstr "Unregistered - Get Rid Of Me!"
      ldstr "Nag Messagebox"
      call value class [System.Windows.Forms] System.Windows.Forms.DialogResult
            [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(class
            System.String, class System.String)
      pop
      ret
}
```

While we're here, let's also remove the code that disables the button. To remove code, we simply need to replace it with code that does nothing. The 'no operation' (NOP) in .NET is 00h (where it is

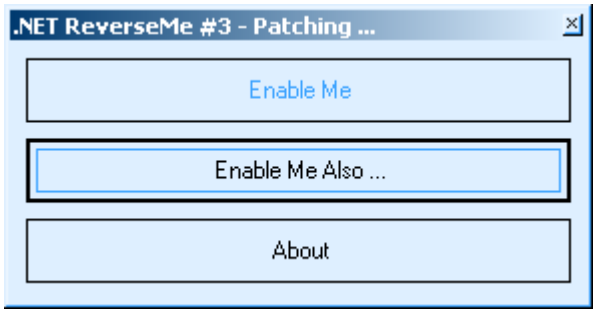
90h in regular x86 opcodes), and we can use it to overwrite code we do not want any longer. In the status bar, IDA will show the file offset as an easy reference:



however, it might be useful to see the byte representation of the code we are seeing. This is disabled by default, but we can enable it easily through the IDA options (set 'Options->General->Number of opcode bytes' to the maximal amount of bytes you wish to see, 8 will do). This will produce lines like the following:

```
72 2D 01 00 70              ldstr "Nag Messagebox"
```

The entire function should be removed (except of course, the '*ret*' instruction), which means all code from 1330h up to (but **not** including) 134Ch should be filled with 00s. This is a task for any hexeditor, I personally used Ultraedit for the job. Knowing the bytes to replace comes in handy here, even if it is merely to comfort oneself that the right code is indeed being overwritten. In any case, a backup is a good idea. Saving the program and running it confirms that we did the right thing, the messagebox has disappeared and one of the buttons is no longer disabled:



So far so good. Next, let's enable the other button, since it is (presumably) an easier task than the second nag screen.

The second button

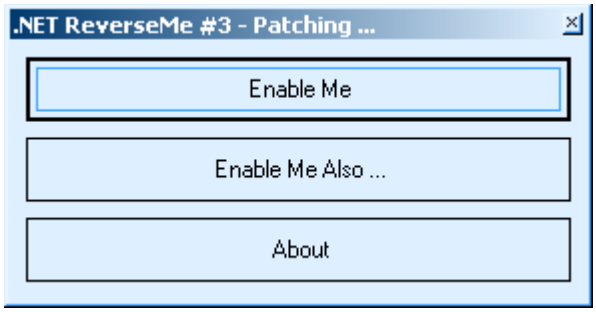
Again, it is easier to find one's way using Reflector, so we'll start our analysis there. In the '*initializeComponent*' function, we find some interesting code regarding the button:

```
this.btnEnableMe1.Enabled = false;
this.btnEnableMe1.FlatStyle = FlatStyle.Flat;
this.btnEnableMe1.Location = new Point(8, 8);
this.btnEnableMe1.Name = "btnEnableMe1";
this.btnEnableMe1.Size = new Size(0x110, 0x20);
this.btnEnableMe1.TabIndex = 0;
this.btnEnableMe1.Text = "Enable Me";
```

The first line is (of course) what concerns us, so let's find the exact location and size of it, in IDA. We find it file offset 10E7h:

```
02                  ldarg.0
7B 01 00 00 04      ldfld class
                          [System.Windows.Forms] System.Windows.Forms.Button
                          Dot_NET_ReverseMe_3.frmReverseMeMain::btnEnableMe1
16                  ldc.i4.0
6F 15 00 00 0A      callvirt void
                          [System.Windows.Forms]
                          System.Windows.Forms.Control::set_Enabled(bool)
```

We could assume a button is created enabled by default (something of which I am not sure), but without much extra effort we can do it properly and explicitly enable the button by replacing `ldc.i4.0` with `ldc.i4.1` (`this.btnEnableMe1.Enabled = true;`). The bytecode for this instruction is *17* so we will simply overwrite the *16* at location 10EDh to enable the button. The result is as expected:

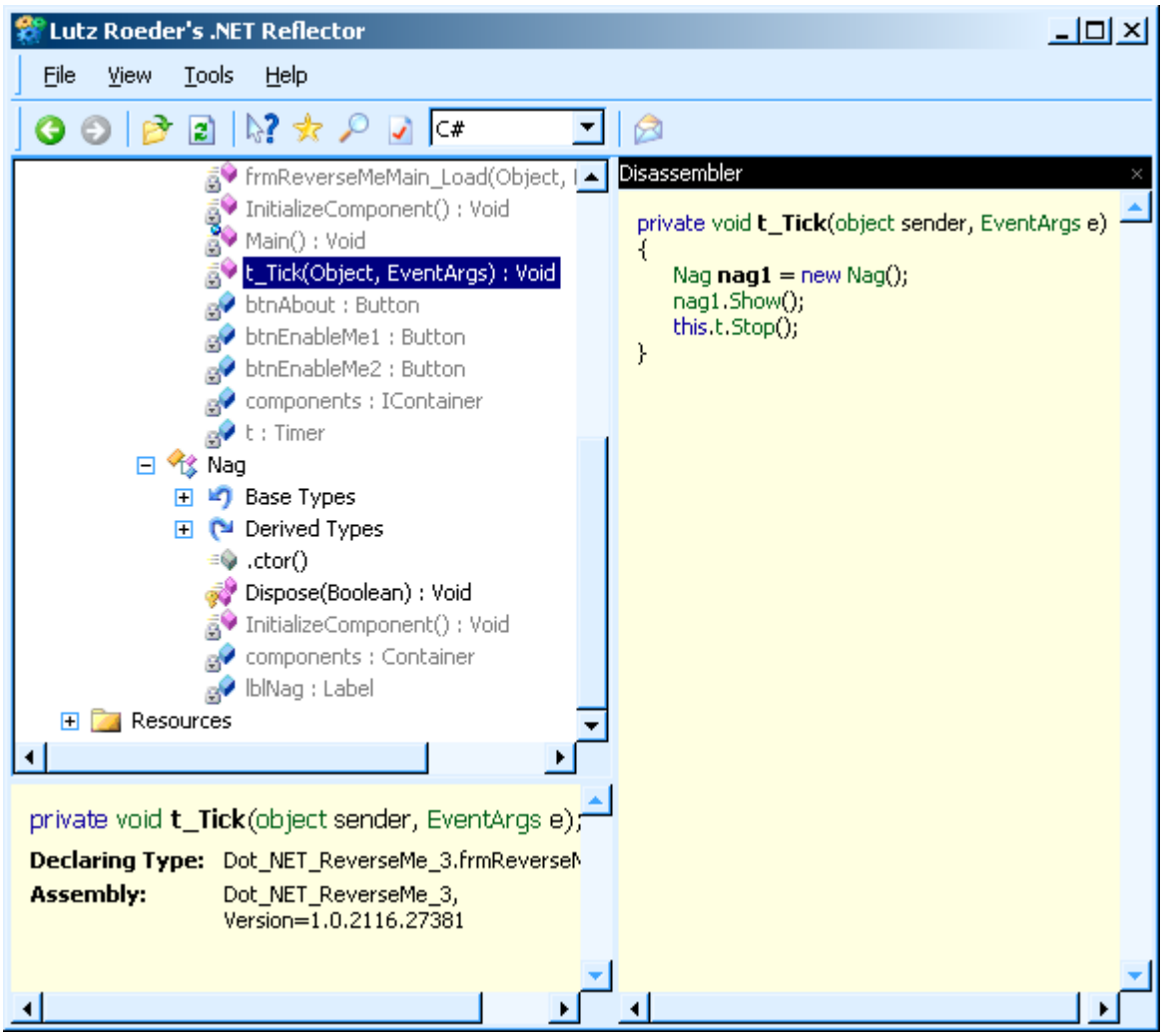


The last task remaining is the small window showing up as a nag screen. The reason this is presumably the most difficult task is because this is a non-standard window, so it probably consists of more code than the other obstacles.

The real nag screen

Looking in Reflector we see that Nag is actually a separate class in the program, with its own attributes and functions. Rather than taking it apart from the inside, we should stop it being created altogether. Because this is a separate class, it will probably be created in a '*new Nag()*' call, which

we will have to find and disable. We find it in the t_Tick function:



In the screenshot you can also see the 'Nag' class on the left. We could disable the nagscreen by removing the first line in this function, however the '*t_Tick*' method is automatically called by a timer, and it stops the timer on the first call (*this.t.Stop()*), so we should go back even further and even stop the timer from ever being created. We find its creation in '*initializeComponent*', at the line

```
this.t = new Timer(this.components);
```

This puts us in a bit of a tricky spot, because if we were to simply remove this line, we would be in trouble when other parts of the code reference this timer. These other lines will assume *t* is a successfully created timer and will fail if we remove its creation. These related lines are:

```
this.t.Enabled = true;
this.t.Interval = 0x3e8;
this.t.Tick += new EventHandler(this.t_Tick);
```

Here we find a much nicer way to deal with the timer. Let's just disable it, causing it never to be started, thus never to be called, thus never creating the nag screen. All other references to the timer will still work, because it is still created as before, it is simply not in use. I'm sure you understand the principle now of changing code (which is good, as it was the purpose of this paper ;-)), so I'll be brief. The relevant code is at offset 11B6h:

```
02              ldarg.0
7B 03 00 00 04  ldfld class
```

```
                        [System.Windows.Forms] System.Windows.Forms.Timer
                        Dot_NET_ReverseMe_3.frmReverseMeMain::t
17                  ldc.i4.1
6F 1E 00 00 0A      callvirt void
                        [System.Windows.Forms]
                        System.Windows.Forms.Timer::set_Enabled(bool)
```

In which we will have to replace the *17* with a *16* (`ldc.i4.0`) to disable instead of enable the timer. As we run the program now, we no longer get any nag screens, and both buttons are enabled. Hurray, we have successfully patched this program to enable and disable parts and functions as we desired.

Conclusion

Patching is useful in many cases, and its power should not be underestimated. It can be used to remove a nag screen protection, bypass a difficult protection to render it useless, and even to add entirely new features to a program. The techniques discussed in this chapter should help you start patching programs on your own, and will be the last chapter dealing with patching in a general sense (perhaps there will be a later chapter about adding code, for example). I do not know yet what the next chapter will be about (or if there will even be one), but in the meantime have fun exploring the world of .NET reversing by yourself. It's always fun to experiment...

– Kwazy Webbit