

# .NET Reverse Engineering Tutorial Episode 1

Version 1.0  
November 2006



## 1. Forewords

We begin another chapter in the ongoing ARTeam series of Reverse Engineering tutorials. Today's topic will go over .NET Reverse Engineering. This topic has been hardly touched upon, thus giving me room to add some information to the reader.

*BEE Seeing YA.*

MDMA



---

*Editor: MaDMAn\_H3rCuL3s*



## 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. .NET Reverse Engineering Tutorial Episode 1 .....	3
1.1. Abstract.....	3
1.2. Targets.....	3
1.3. How to reverse the application .....	3
1.3.1 Preparation .....	3
1.3.2 Disassembling the target .....	3
1.4. References.....	13
1.5. Conclusions .....	13
1.6. Greetings .....	13
Document History .....	14



# 1. .NET Reverse Engineering Tutorial Episode 1

## 1.1. Abstract

This target is protected by a license check. We are greeted at the beginning with a nag asking for the license file. As a trial user you would get one. The scope of this tutorial isn't to make the program act like a full version, but rather patch around the nag at startup. So if the patch does indeed render it full, this is not my responsibility. You are encouraged to delete this program from your hard drive upon full reversing.

## 1.2. Targets

To keep the tutorial on a current level, you can find the target at the following place. Since programs are usually regularly updated this will keep this tutorial valid for some time to come.

- StreamPatrol v2.0

<http://arteam.accessroot.com/tools/StreamPatrol-2.0.0.zip>

## 1.3. How to reverse the application

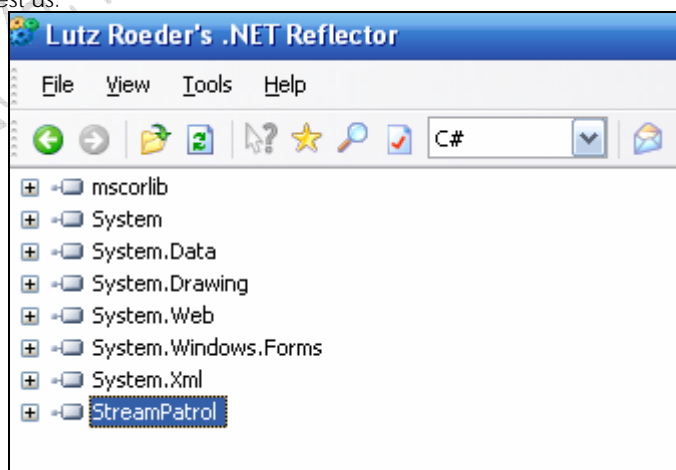
### 1.3.1 Preparation

We need the following tools to help us along the way.

1. [Reflector](#)
2. IDA (Google it)
3. FlexHEX (Google it)

### 1.3.2 Disassembling the target

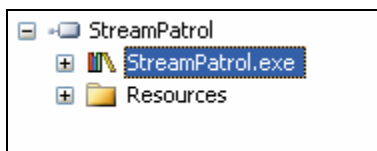
Today's target will be StreamPatrol. The fix we will discuss is not 100 percent, which I will talk about later on. To start we get the victim program from link we saw before. We will load up the victim into Reflector and look for anything that might interest us.



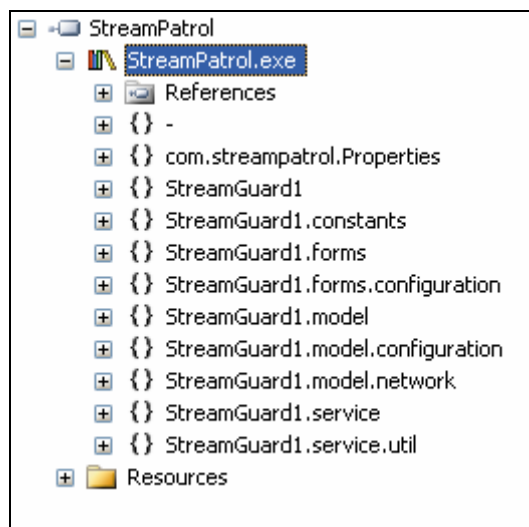
*We see the target is loaded up.*



We then expand the target using the "+" sign.

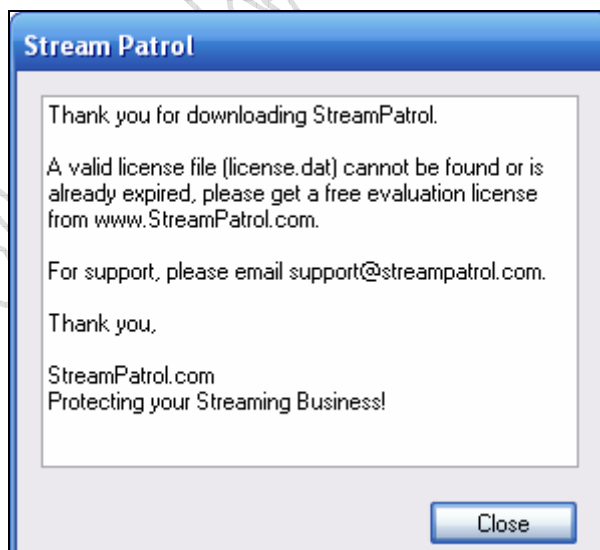


*And then we expand the "StreamPatrol.exe"*



*And we see what the exe has for us to view.*

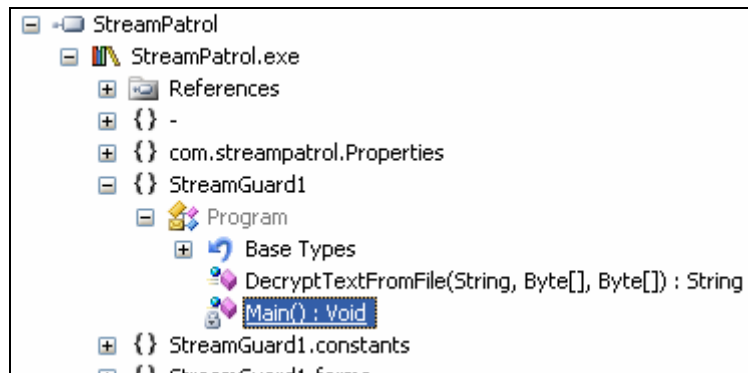
If we were to run the exe outside any kind of disasm or debug engine, we would be greeted by this:



*A nasty nag!!!!*

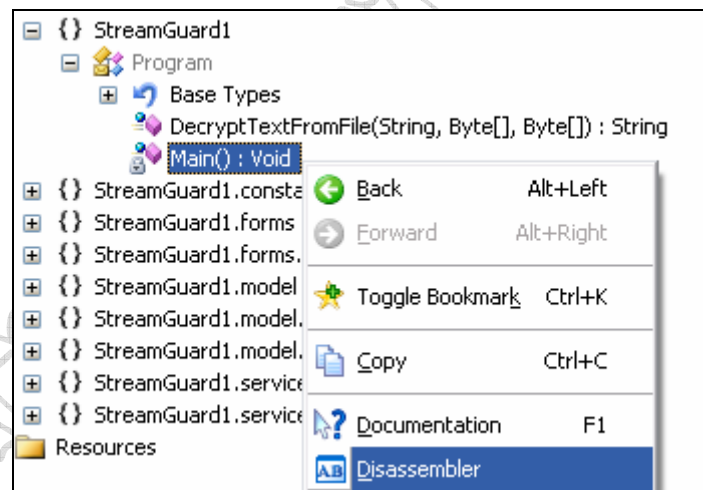
The author has led us right to the check. Look closely at the nag, its looking for "license.dat".





*We see the Main() : Void*

We then right click it then we will disassemble it.



*And then look to the right side to see our disasm.*



```

Disassembler

[STAThread]
private static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Version version1 = Environment.Version;
    LogService.LogInfo(string.Concat(new object[] { "Starting the application with .net framework version: ", version1.Major, ".", version1.Minor, ".", version1.Revision }));
    string text1 = "dOjiz+CFbuH8VD1HPxF8xDd+yzvpaRC";
    string text2 = "SH2RLBBWwIw=";
    TripleDES edes1 = TripleDES.Create();
    edes1.Key = Convert.FromBase64String(text1);
    edes1.IV = Convert.FromBase64String(text2);
    string text3 = "";
    bool flag1 = false;
    License license1 = null;
    try
    {
        text3 = Program.DecryptTextFromFile("license.dat", edes1.Key, edes1.IV);
        license1 = new License(text3);
        flag1 = license1.IsValid();
    }
    catch (Exception)
    {
        flag1 = false;
    }
    if (flag1)
    {
        FormSummary summary1 = new FormSummary();
        summary1.StartPosition = FormStartPosition.CenterScreen;
        summary1.License = license1;
        Application.Run(summary1);
    }
    else
    {
        FormNoLicense license2 = new FormNoLicense();
        license2.StartPosition = FormStartPosition.CenterScreen;
        Application.Run(license2);
    }
}

```

*And a few things will pop out right away.*

First off we will learn a bit about Reflector and what we should be doing here. If we look close we see

```

License Check

try
{
    text3 = Program.DecryptTextFromFile("license.dat", edes1.Key, edes1.IV);
    license1 = new License(text3);
    flag1 = license1.IsValid();
}
catch (Exception)
{
    flag1 = false;
}

```

So we see we get something set here, like "Flag1". So we could easily set the flag, but more problems could arise from this, as other members could use the same function. So let's check to see if it does. If you go to the "flag1 = License1.IsValid();" and the "IsValid" should be underlined. If you click it you should get here:

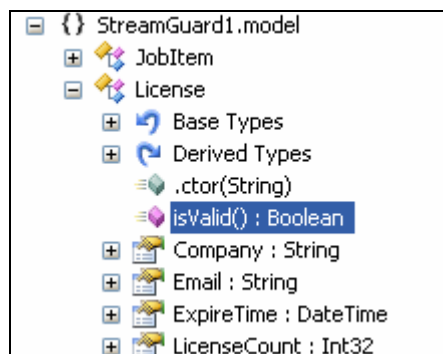


```
Disassembler

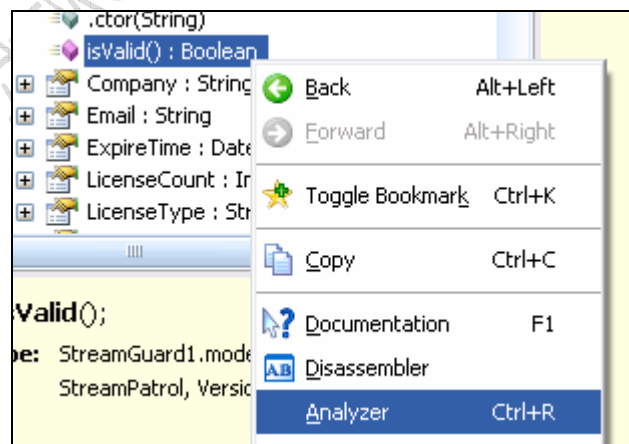
public bool isValid()
{
    if ((this.startTime < DateTime.Now) && (DateTime.Now < this.expireTime))
    {
        return true;
    }
    return false;
}
```

*And we now conclude that the "isValid" flag is set true or false.*

Now we will check to see what other functions might use this, if it turns out that others use it, this will become a bad place to patch. So we look left and we appear to be in the function "isValid"



*Then we right click this function and do like below:*

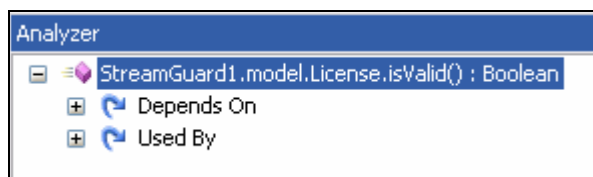


*Then we right click it, but instead choose "Analyzer" and look right:*

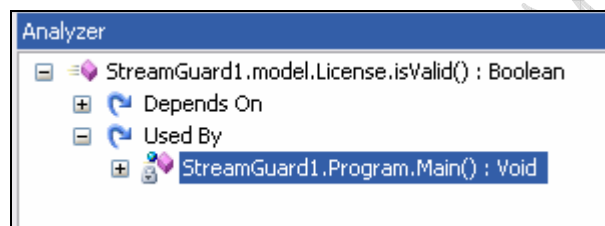




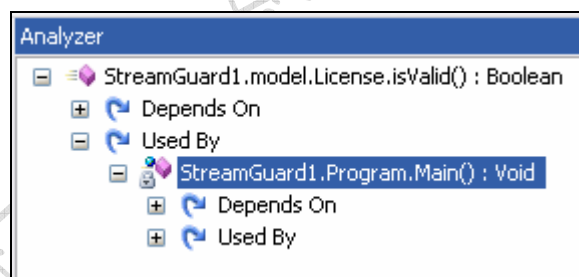
*Expand this:*



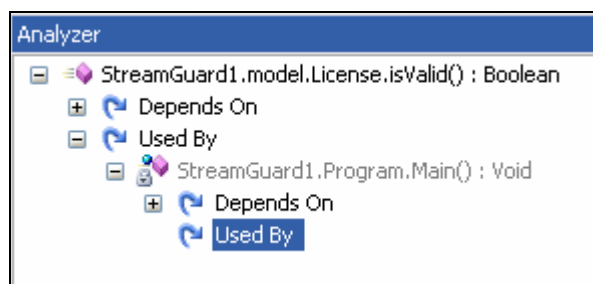
*Expand this:*



*Expand this:*



*Expand this:*

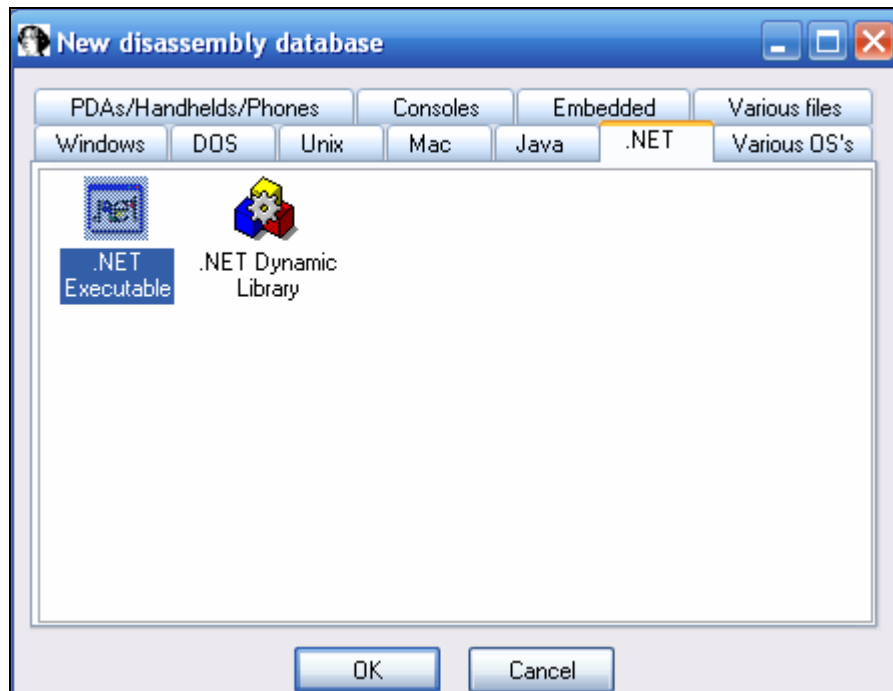


*We see nothing but our main function uses it.*

Now we can get into our patching. First we will analyze with IDA to get the exact opcode we are using. So start up IDA and load up the exe into it.







*Just in case you others don't know how to load up .net exe's into IDA.*

get_minimal_supported_configuration	0000C200
F get_ContactList	0000C270
F set_ContactList	0000C280
D StreamGuard1.Program	0000C570
F Main	0000C580 P
F DecryptTextFromFile	0000C6A0
D StreamGuard1.model.configuration.StreamSet...	0000C700

*We double click the "StreamGuard1.Program" since we got this from Reflector.*

We scroll down a bit until we find our place from before.

```

• ldloc.s 6
• callvirt bool StreamGuard1.model.License::IsValid()
• stloc.s 5
• leave.s loc_C656
}
catch [mscorlib]System.Exception {
• pop
• ldc.i4.0
• stloc.s 5
• leave.s loc_C656
}

```

*This is the exact place we want to patch.*

If we refer to our Opcode manual, "PUSH 0" in ASM. So we can conclude that "false.isValid", if we PUSH 1, we get



we will see that the command "ldc.i4.0" = since we are going to PUSH 0 for true.isValid. So let's refer to our opcode

manual real fast to get the hex number associated with the command.

### 3.40 ldc.<type> – load numeric constant

Format	Assembly Format	Description
20 <int32>	ldc.i4 num	Push num of type int32 onto the stack as int32.
21 <int64>	ldc.i8 num	Push num of type int64 onto the stack as int64.
22 <float32>	ldc.r4 num	Push num of type float32 onto the stack as F.
23 <float64>	ldc.r8 num	Push num of type float64 onto the stack as F.
16	ldc.i4.0	Push 0 onto the stack as int32.
17	ldc.i4.1	Push 1 onto the stack as int32.
18	ldc.i4.2	Push 2 onto the stack as int32.
19	ldc.i4.3	Push 3 onto the stack as int32.
1A	ldc.i4.4	Push 4 onto the stack as int32.

*Our lovely opcode reference manual.*

We have found our command in the manual, and see the Opcode for our command is 16. So let us think for at least 5 minutes here. The command we are looking at is a PUSH 0 in ASM.

16	ldc.i4.0	Push 0 onto the stack as int32.
----	----------	---------------------------------

This will set our flag to false. If we were to somehow set the flag true, by say "PUSH 1" then that would be cool. Let us refer to our manual for the command.

17	ldc.i4.1	Push 1 onto the stack as int32.
----	----------	---------------------------------

The trick I use (which is what I was shown by my homie) If you highlight the .net command, then switch to Hex view you can see the exact Hexadecimal number you need. So highlight the "ldc.i4.0" then switch to hex view.

```
catch [mscorlib]System
pop
ldc.i4.0
stloc.s 5
leave.s loc_C656
```

*Now we switch to hex view.*

```
00 00 06 13
26 16 13 05
07 11 07 17
```

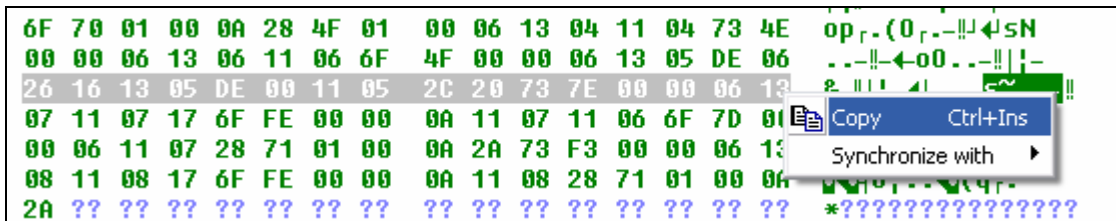
*Our opcode is 16.*

So now like the tutorials scope, we

will patch the executable instead of Disasm

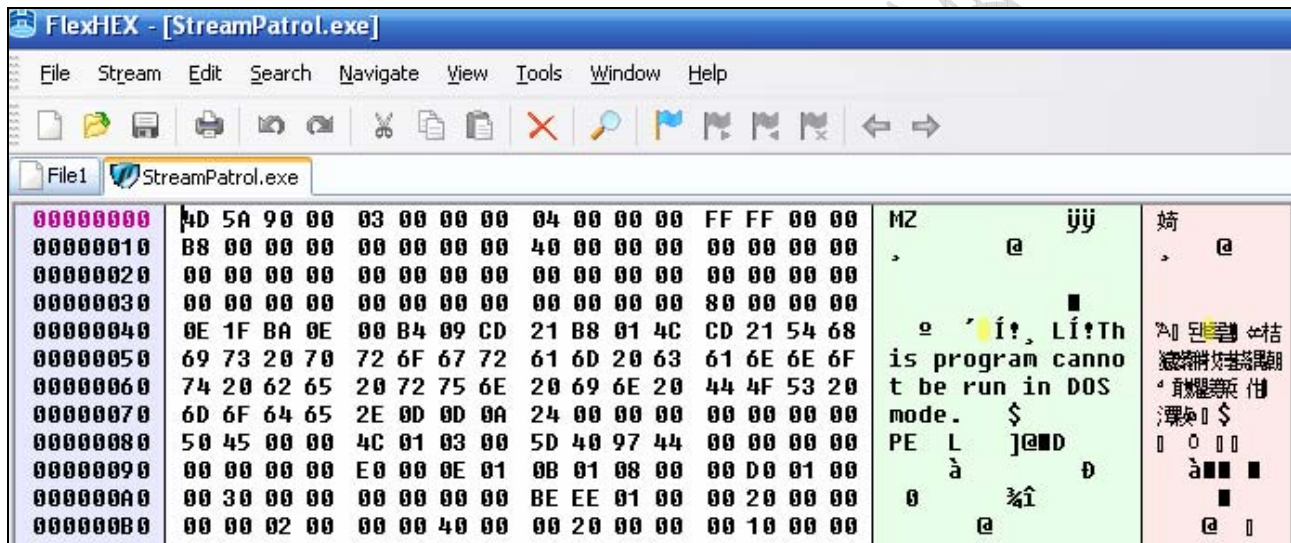


then Asm. But just to be safe, let's do something to ensure we don't mess up. In IDA, in the Hex View, where our "16" is at, let's binary copy the whole line so when we search for it, we don't mess it up.



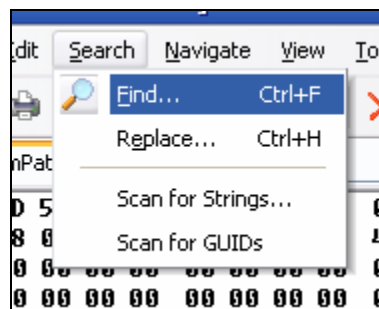
*There. Now we have a bigger byte signature to find.*

So, now to patch the executable, let's fire up any hex editor. My hex editor of choice is FlexHEX. So I will show pictures from that. Any hex editor will work (should?).



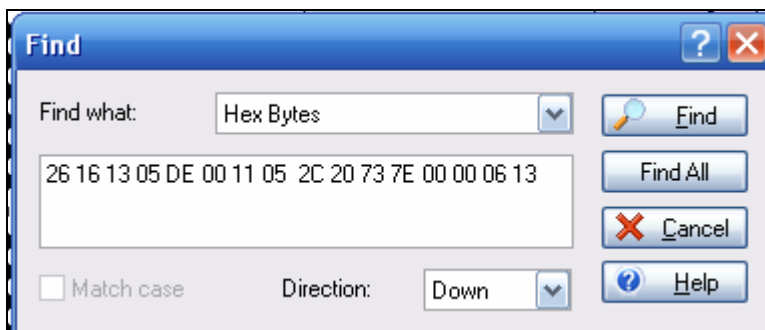
*I love this hex editor ☺.*

Now we will do a binary search for our opcodes.



*And then the next picture pops up.*





*We hit the "Find" button.*

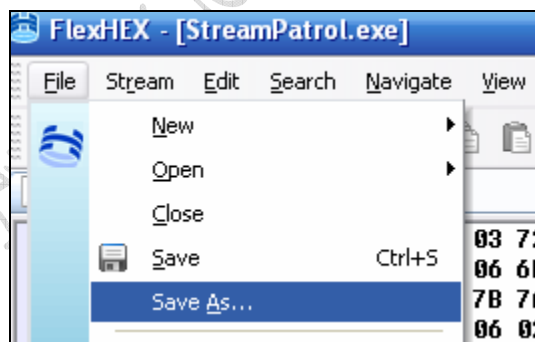
0000CE90	01 00 0A 09	6F 70 01 00	0A 28 4F 01	00 06 13 04	op (0
0000CEA0	11 04 73 4E	00 00 06 13	06 11 06 6F	4F 00 00 06	sN o0
0000CEB4	13 05 DE 06	26 16 13 05	DE 00 11 05	2C 20 73 7E	p & p , s~
0000CEC0	00 00 06 13	07 11 07 17	6F FE 00 00	0A 11 07 11	op

*This is the only instance we find. So yeah!*

Let's edit it, and make the 16 -> 17.

0000CEA0	11 04 73 4E	00 00 06 13	
0000CEB6	13 05 DE 06	26 17 13 05	
0000CEC0	00 00 06 13	07 11 07 17	

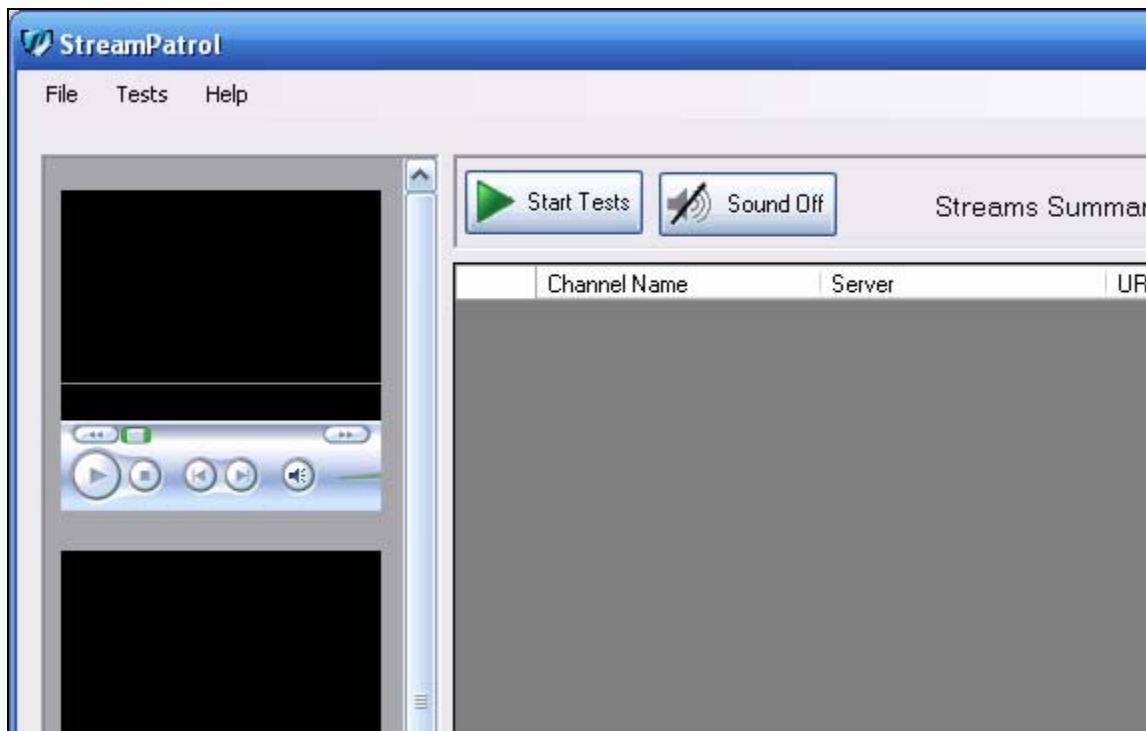
*Now we save changes.*



*Then save to whatever.*

Let's run the new patched executable...





*And the bitch runs!!!!*

Now the note about the program. If you were to click the "about" button, we would crash. This happens because the "license.dat" file is not even real. We make the program think it found it, but when you check "about" it actually reads from it. So, consider this a reversing tutorial. At least we had fun learning.

## 1.4. References

"Primer on Reversing .NET Applications", Shub-Niggurath, Googleplex, zyzygy, <http://tutorials.accessroot.com>

## 1.5. Conclusions

We have reached the end, and my hope is that you, the reader, learned a new technique. Instead of the old, Disassemble, re-assemble technique, we simply patched the target and saved ourselves more headaches. This technique will only work on "non-obfuscated" executables. If this program was obfuscated, then the simple patching would not be possible. We would have to rely on the decrypt, disassemble, patch, and re-assemble method.

## 1.6. Greetings



ARTeam, fly, shoooo, heXer, unpack.cn, PEdiy forum, SECTION-8, Like maybe one or two 0day groups, Anyone who has done any sort of chemical brain enhancement ( ☺ ), Anyone who makes their own chemical brain enhancers, especially the old HiVE dwellers (you know who you are), and of course.... YOU!

## Document History

- Version 1.0 first public release



BEE HAPPY!

