

WHEN MANAGED-CODE BASED ATTACKS AREN'T ENOUGH, THAT IS TO SAY LET'S REVERSE DXTORY. (TONYWEB 2012)



Introduction

It's been a while since reversing of .NET applications began. I still remember the first tutorials on the subject and the first targets for which changing a few bytes with an hex editor was enough to fully remove the restrictions from.

So much has changed: developers of both software and protections made the reversing process increasingly complex and time-consuming; however, from the reverse engineering side, capable and willing individuals wrote increasingly powerful tools that allow us to continue focusing on those few bytes to patch. ☺

Occasionally, however, Reflector decompiled code allows us to remove only a portion of the limitations targets we are working on exhibit, and we are therefore forced to get our hands dirty with native code to complete our mission. One of these programs is Dxtory, which is currently at version 2.0.110.

Warning. *Although the tutorial tries to be as detailed as possible, it's clear that not all of the subjects will be treated in a comprehensive manner. Therefore it is assumed that the reader has some experience or is at least familiar at reversing both .NET applications and native ones.*

Ladies and gentlemen ... the victim.

Dxtory is basically a program that lets you take snapshots or video clips from Direct-X applications, primarily games, even at full screen mode. Allegedly, unlike other applications with the same purpose, it doesn't slow down the gameplay experience and therefore it guarantees better performances and results.

A detailed description of the program and its download links are available from the manufacturer website: <http://dxtory.com/v2-home-en.html>

What we need to start

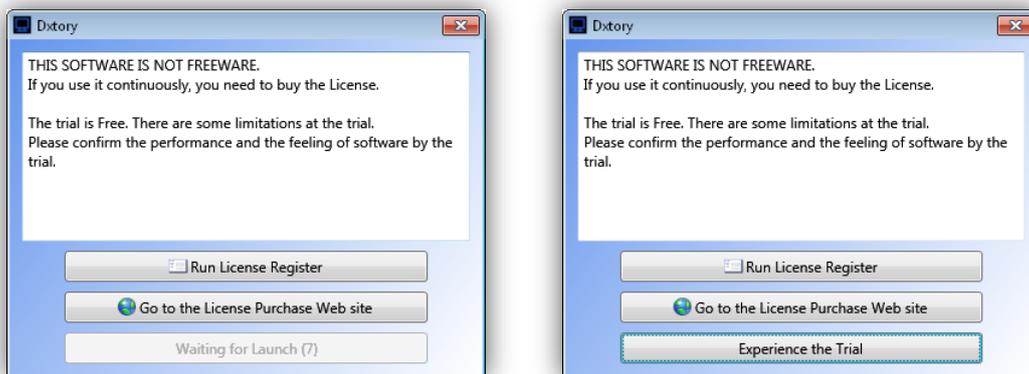
Below is a list of the tools we'll use:

- RedGate Reflector and its Reflexil plugin (at v.1.3 at the time of writing)
- De4Dot (to deobfuscate our .NET executable, <https://github.com/Oxd4d/de4dot>)
- CFF Explorer
- Your favorite hex editor (I'll use the free HxD editor)
- Mono.Cecil and Public Key Injector (should be attached to this tutorial)
- Reter Decompiler
- OllyDbg 1.10 with Multimate Assembler plugin
- A DDS image viewer (optional, I used IrfanView version 4.25)
- A small Direct-X app which you can test the program onto. I found a nice and tiny one, FractalDemo.exe, here: <http://www.defmacro.org/ramblings/fractal.html>
- A brain in working order, as usual ;)

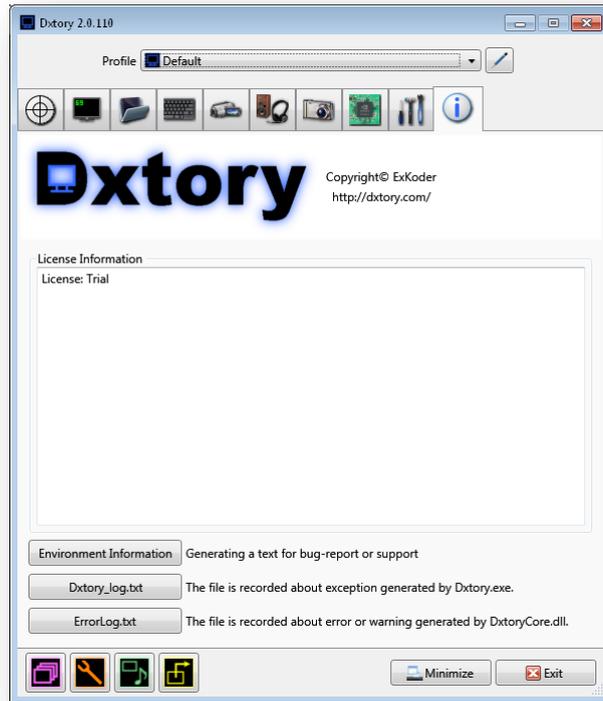
Now we should be ready to go. Fasten your seatbelts and let's start ☺

First program run and first impressions

Once installed and run, the program shows us immediately a welcome NAG:



After the countdown, let's click on 'Experience the Trial' so we can see the main window, where the trial status is clearly stated (*License: Trial*):



When we close the application we are redirected to author web site: clear invite to purchase the product. Main limitation, however, is not the NAG nor the redirection but the occurrence of a logo overlay (*watermak*) on the captures: here's an example of a frame, produced by FractalDemo.exe, registered through Dxtory:



So our final goal will be to have “clean” videos that is without those annoying overlays; but let's proceed step by step and launch *ProtectionID* in order to identify our enemy.

```

-[ ProtectionID v0.6.4.1 JULY]-
(c) 2003-2011 CDKILLER & TippeX
Build 07/22/11-02:48:07
Ready...
Scanning -> C:\Program Files\Dxtory2.0.110\Dxtory.exe
File Type : 32-Bit Exe (Subsystem : Win GUI / 2), Size : 535040 (082A00h) Byte(s)
[File Heuristics] -> Flag : 0000000000001001101000000110000 (0x0004D030)
[!] dotFuscator detected !
[CompilerDetect] -> .NET
[.Net Info -> v 2.5 | Flags : 0x0000000B -> COMIMAGE_FLAGS_ILONLY | COMIMAGE_FLAGS_32BITREQUIRED |
COMIMAGE_FLAGS_STRONGNAMESIGNED |
[.] Entrypoint (Token) : 0x06000643
[.] Metadata RVA : 0x00025A94 | Size : 0x0002C894 (182420)
[.] Metadata->Version 1.1 -> v4.0.30319
[.] Flags : 0x0 | Streams : 0x5 (5)
- Scan Took : 0.187 Second(s) [0000000BBh tick(s)]

```

The output shows us that the target is obfuscated with *dotFuscator* and gives us useful information about the executable like entry-point token, required .NET framework version and the presence of a StrongName (we'll come to it later)¹. Well, I think we can start rolling ☺

It starts

To simplify the analysis we can immediately fire up *de4dot*, the awesome tool by *0xd4d*, which will take care of the deobfuscation for us:

```

>de4dot Dxtory.exe

de4dot v1.4.1.3405 Copyright (C) 2011 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Dotfuscator 000:0:2:5.0.2500.0 ([...]Dxtory.exe)
Cleaning ...\Dxtory.exe
Renaming all obfuscated symbols
Saving ...\Dxtory-cleaned.exe

```

Usually, files cleaned by *de4dot* are already runnable, so let's copy *Dxtory-cleaned.exe* in original folder and try to launch it; soon after clicking the 'Experience the Trial' button the program shows us this error message:



We rush to read the log and we find:

```

=== 2012/01/29 ===
[Module]
Dxtory-cleaned.exe 2.0.110

[Process]
ID: 8184

```

¹ This is a beta version of ProtectionID (at the time of writing).

```
[Thread]
MainThread (1)

[Error Message]
'c' field specified was not found.

[StackInfo]
  at System.Reflection.CustomAttribute.GetCustomAttributes(RuntimeModule decoratedModule, Int32 decoratedMetadataToken, Int32 pcaCount, RuntimeType attributeFilterType, Boolean mustBeInheritable, IList derivedAttributes, Boolean isDecoratedTargetSecurityTransparent)
  at System.Reflection.CustomAttribute.GetCustomAttributes(RuntimePropertyInfo property, RuntimeType caType)
  at System.Reflection.RuntimePropertyInfo.GetCustomAttributes(Type attributeType, Boolean inherit)
  at GClass56.smethod_5(Object[] object_0)
  at GClass51.method_3()
  at Class4.method_1()
  at GClass90.GClass90_Startup(Object sender, StartupEventArgs e)
```

Several times Oxd4d (de4dot author) explained us that, when cleaned files have problems accessing resources, it's convenient to keep original symbol names using the "--dont-rewrite" option:

```
>de4dot --dont-rewrite Dxtory.exe

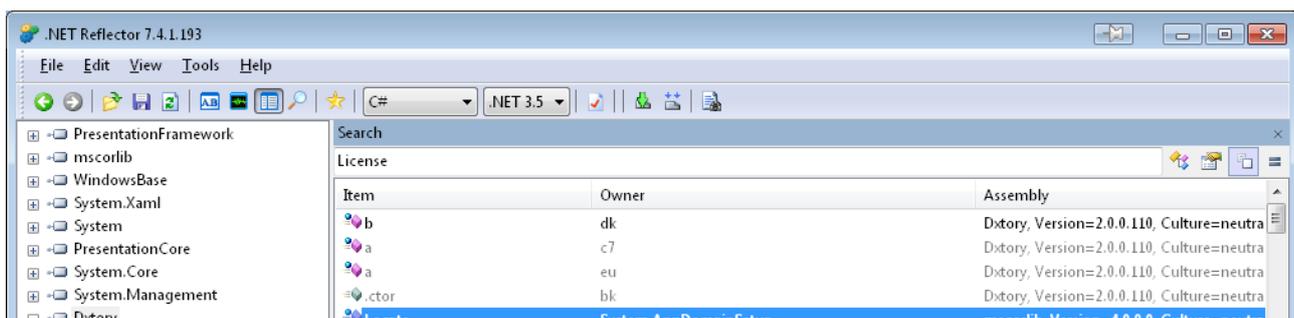
de4dot v1.4.1.3405 Copyright (C) 2011 de4dot@gmail.com
Latest version and source code: https://github.com/Oxd4d/de4dot

Detected Dotfuscator 000:0:2:5.0.2500.0 (...\\Debug\Dxtory.exe)
Cleaning ...\\Debug\Dxtory.exe
Saving ...\\Debug\Dxtory-cleaned.exe
```

This way the resulting file will start without any problem (just be sure to disable .NET framework StrongName verification²). ☺

Patching phase 1: managed world from the inside with Reflector

Now we can begin opening our cleaned executable in Reflector and actually getting our hands dirty; on the main window (info tab) we can clearly see the 'License: Trial' string and that string seems a very good starting point. So, once target is opened in Reflector, search for the 'License' string :



We would find four occurrences. Let's check them. ;)

The first one:

```
public static void b()
{
    string str = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location) + @"\";
    JumpList list = new JumpList();
    if (!File.Exists(dt.a + @"register.dat"))
    {
        list.JumpItems.Add(a(str + "LicReg.exe", d9.b("String_LicenseRegister")));
    }
    list.JumpItems.Add(a(str + "RawCapConv.exe", d9.b("String_RawCapConv")));
}
```

² It's enough to set the *AllowStrongNameBypass* DWORD to 1 (see <http://msdn.microsoft.com/en-us/library/cc713694.aspx>)

```

list.JumpItems.Add(a(str + "AVIFix.exe", d9.b("String_AVIFix")));
list.JumpItems.Add(a(str + "AVIMux.exe", d9.b("String_AVIMux")));
list.JumpItems.Add(a(str + "DxtoryVideoSetting.exe", d9.b("String_VideoSetting")));
JumpList.SetJumpList(Application.Current, list);
}

```

seems not so interesting because it just verifies the existence of a license file and shows/hides the link for registration accordingly.

Let's move to the next one:

```

public static bool a()
{
    if (c)
    {
        MessageBox.Show(d9.b("Msg_ExpireLicense"), "Dxtory", MessageBoxButton.OK,
        MessageBoxImage.Asterisk);
        return true;
    }
    return false;
}

```

This routine is more interesting for us: it shows us the expiration condition ('c' variable): let's go ahead and check also the other two to gather as more information as possible.

The third routine, as we can guess, prints a complete resume of the program and the system it's installed on (that's the output given by 'Environment Information' button shown on about tab):

```

public static string a()
{
    string str2;
    long num5;
    StringBuilder builder = new StringBuilder();
    builder.AppendLine("[Dxtory]");
    Assembly entryAssembly = Assembly.GetEntryAssembly();
    Version version = entryAssembly.GetName().Version;
    builder.AppendLine("Version: " + string.Format("{0}.{1}.{2}", version.Major, version.Minor,
version.Revision));
    builder.AppendLine("UID: " + c.a().ToString());
    builder.AppendLine();
    builder.AppendLine("[Dxtory Files]");
    builder.Append("InstallPath: ");
    builder.AppendLine(Path.GetDirectoryName(Assembly.GetEntryAssembly().Location));
    foreach (string str in Directory.GetFiles(Path.GetDirectoryName(entryAssembly.Location)))
    {
        a(builder, str);
    }
    builder.AppendLine();
    builder.AppendLine("[System Information]");
    if (br.a(out str2))
    {
        builder.AppendLine("CPU: " + str2);
    }
    else
    {
        builder.AppendLine("CPU: Unknown");
    }
    :
    :

    builder.AppendLine("[AudioCodec]");
    foreach (IAudioCodec codec2 in c0.c().GetAudioCodec())
    {
        builder.AppendLine(codec2.ToString());
    }
    builder.AppendLine();
    builder.AppendLine("[LicenseInformation]");
    builder.Append(c0.c().GetLicenseInformation());
    string licenseUserID = c0.c().GetLicenseUserID();
    if (licenseUserID != "")
    {
        SHA1Managed managed = new SHA1Managed();
        managed.Initialize();
        builder.AppendLine("LicVerify: " + ec.a(managed.ComputeHash(ec.c(licenseUserID))));
    }
    builder.AppendLine();
}

```

```

foreach (IProfile profile in c0.c().GetProfileManager().GetProfile())
{
builder.AppendLine("=====");
builder.AppendLine();
a(builder, profile);
}
builder.AppendLine("=====");
string str4 = builder.ToString();
byte[] bytes = Encoding.Unicode.GetBytes(str4.Replace("\r\n", ""));
SHA1Managed managed2 = new SHA1Managed();
managed2.Initialize();
string str5 = ec.a(managed2.ComputeHash(bytes));
return (str4 + "VerifyHash: " + str5 + "\r\n");
}

```

Look, in the last part, there are references to registration status and likely to a SHA-1 based integrity check.

And last but not least let's see the fourth routine:

```

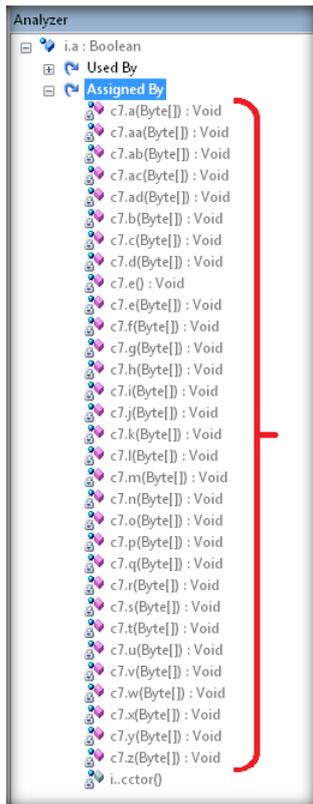
public bk()
{
this.a = new cv();
this.b = !i.a;
StringBuilder builder = new StringBuilder();
if (i.a)
{
builder.AppendLine(d9.b("String_License") + ": " + d9.b("String_Registered"));
if (i.c == DateTime.MinValue)
{
builder.AppendLine(d9.b("String_Expire") + ": " + d9.b("String_Unlimited"));
}
else
{
builder.AppendLine(d9.b("String_Expire") + ": " + i.c.ToString("yyyy/MM/dd"));
}
this.d = i.b;
}
else
{
builder.AppendLine(d9.b("String_License") + ": " + d9.b("String_Trial"));
this.d = "";
}
this.c = builder.ToString();
this.e = new eq();
}

```

And without any doubt this is the clearer one :D

We immediately grasp that the registration status is stored into *i.a* property and that, if *i.c* equals *DateTime.MinValue*, it will result in an unlimited license.

The question now is: where is this property set? To answer this question let's analyze *i.a* with the Reflector Analyze command:



If we inspect these routines one by one, almost all have the same structure (that's why they are "grouped" in the picture above). Below, as an example, we'll look at one of them:

```

private static void ab(byte[] A_0)
{
    string str;
    d1 d = new d1();
    if (d.b(A_0, out str))
    {
        bool flag;
        string str2 = dt.a + cb.a("XoA7ZbrhQgD8BjndpWTmGM4PiqeyfcB04");
        cj cj = d.e(str2, out str);
        i.a = cj != null;
        if (i.a)
        {
            try
            {
                i.b = cj.n(cb.a("S4ANZQ"));
                i.c = cj.k(cb.a("R4AxZa/rwgy8A7J"));
                if (i.c < DateTime.Now)
                {
                    i.a = false;
                    File.Delete(str2);
                    c = true;
                }
            }
            catch
            {
            }
        }
        byte[] buffer = new byte[] {
            0xcd, 0x1d, 0x3b, 0x2d, 0xd8, 0x12, 0x87, 0xdb, 0x18, 0x2a, 0x26, 0xdf, 0x62, 0x31, 0x41, 0xd4, 0x18, 0x5c
        };
        ds ds = new ds();
        ds.a(i.e);
        ds.a(buffer, buffer);
        d.a(buffer, 0, 20, out flag);
        if (flag)
        {
            Interlocked.Increment(ref c7.d);
        }
    }
}

```

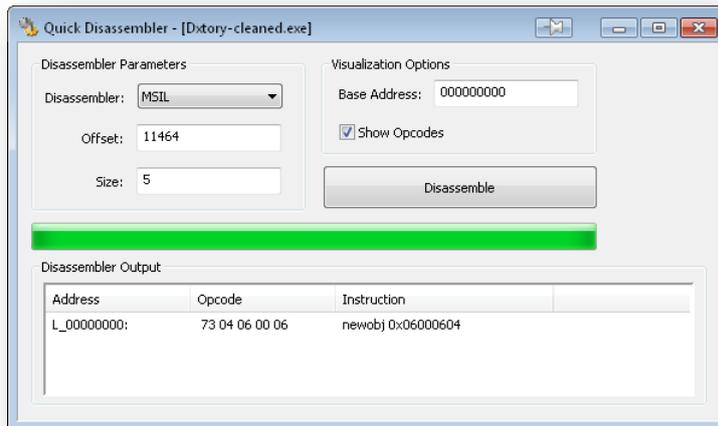
VA	00413258
RVA	00013258
File Offset	00011458

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00011450	4F	95	00	03	21	00	00	01	1B	30	05	00	E8	00	00	00
00011460	B3	00	00	11	73	04	06	00	06	0A	06	02	12	01	6F	FD
00011470	05	00	06	39	D3	00	00	00	7E	F8	06	00	04	72	B3	20

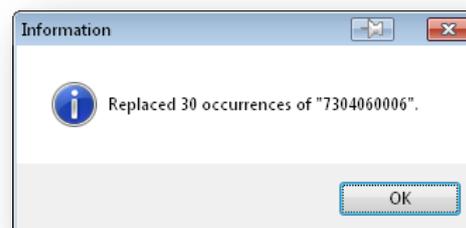
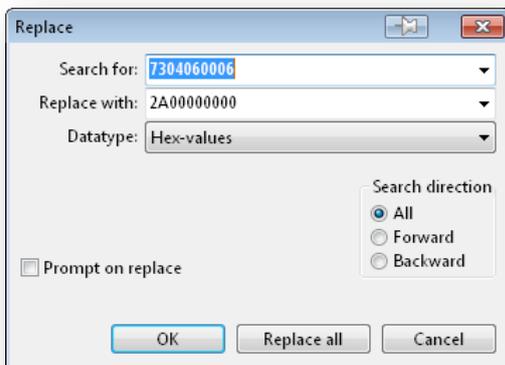
The aim of these routines is mainly evaluating our registration status to determine if we are indeed registered and/or if our trial period has expired: we'll take care to avoid any kind of control about real program status.

With the aid of Reflexil we can get method RVA: **78424**. In this version of the plugin the address is in decimal notation (in contrast with previous versions), so we need to convert it in hex (**0x13258**) before using it in CFF Explorer.

At that RVA we find the routine beginning and, skipping the 12-byte-fat-header (in green in the picture), we reach the "real" code: 73 04 06 ... First instruction is highlighted: **newobj 0x06000604**



That instruction is important since it's a distinctive mark of these routines which gives us the opportunity to patch almost all of them with a single search and replace operation: in detail we'll replace the *d1* object creation with a *ret* (opcode 2A ;). Open up our favorite hex editor and execute this substitution:



Let's save the altered file and re-decompile it (or simply press F5, refresh, on keyboard if we applied the modifications to the file already opened in Reflector). Look again for *i.a* references and see by yourself that the majority of the aforementioned routines have a *ret* as the first instruction. The search and replace operation missed *c7.e()* and *i..ctor()* constructor.

c7.e() routine has also the purpose of ascertaining if our registration status is invalid ...

```

private static void e()
{
    byte[] buffer;
    using (Stream stream = Assembly.GetEntryAssembly().GetManifestResourceStream("Dxtory.Resources.Dxtory10.pub"))
    {
        int length = (int) stream.Length;
        buffer = new byte[length];
        stream.Read(buffer, 0, length);
    }
    a[(DateTime.Now.Day - 1) % a.Length](buffer);
    if (i.a)
    {
        string[] strArray = new string[] {
            "1abc5cd1d7ec0788925ad2fe32d58690a1756a5b", "53de47713f48e1d740d7e5b686cc59a4c2b428a5", "9240cad95dc8ddf
            "74eb5051c3e8438dc45b83eb10a8b500b59b579e"
        };
        foreach (string str in strArray)
        {
            if (str == i.b)
            {
                i.a = false;
                return;
            }
        }
    }
}
}

```

... therefore it will be treated like the other ones. Let's find its RVA: **0x12D8C** and, directly in CFF Explorer, do change

from:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00010F90										28	FA	00	00	0A			(ú...

to:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00010F90											2A	00	00	00	00	

and save the file.

We only have the constructor left. Here's how it appears in Reflector:

```

static i()
{
    a = false;
    b = "";
    c = DateTime.MinValue;
    d = false;
}

```

Very good, there's really little to do here 😊. Our variable *a* should be clearly set to true. Variable *c* is already set to a suitable value (did you remember *MinValue* mean *Unlimited*?).

Again thanks to Reflexil plugin we can read the method RVA: 125596 (0x1EA9C) and we can apply the obvious modification (we'll initialize the property to true changing an *ldc.i4.0* – equivalent to 0x16 – with an *ldc.i4.1* that is 0x17)

from:

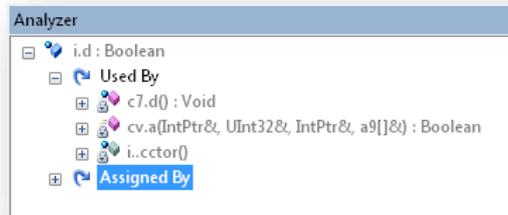
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
0001CC90													86	16	80	E3	+Eä

to:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
0001CC90													17	80			e

Just for the record, here a tiny header is used (green highlighted).

And what about the *d* variable? Well, it's worth analyzing it:



Let's try to understand its meaning by studying how it's used. Look at the first routine:

```
private static void d()
{
    if (d6.a(i.e, Path.GetDirectoryName(Assembly.GetEntryAssembly().Location) + @"\DxtoryCore.dll"))
    {
        Interlocked.Increment(ref d);
    }
    for (int i = 0; i < (b.Length - 1); i++)
    {
        b[i].Join();
    }
    if (object.Equals(d, b.Length))
    {
        i.d = true;
    }
}
```

Hmmm ... a reference to *DxtoryCore.dll* in main executable folder. Enter *d6.a* routine just to clear up our ideas on what's happening here:

```

d6.a(Byte[], String) : Boolean
public static bool a(byte[] A_0, string A_1)
{
    byte[] bytes = Encoding.ASCII.GetBytes("FILESIGN");
    try
    {
        byte[] buffer2;
        using (Stream stream = new FileStream(A_1, FileMode.Open, FileAccess.Read, FileShare.ReadWrite))
        {
            buffer2 = new byte[stream.Length];
            stream.Read(buffer2, 0, buffer2.Length);
        }
        int length = buffer2.Length;
        if (length >= 4)
        {
            int num2 = BitConverter.ToInt32(buffer2, length - 4);
            int num3 = (length - num2) - 4;
            if ((num3 >= 0) && ((num3 + bytes.Length) <= (length - 4)))
            {
                for (int i = 0; i < bytes.Length; i++)
                {
                    if (bytes[i] != buffer2[num3 + i])
                    {
                        return false;
                    }
                }
                byte[] destinationArray = new byte[num2 - bytes.Length];
                Array.Copy(buffer2, num3 + bytes.Length, destinationArray, 0, num2 - bytes.Length);
                int num5 = BitConverter.ToInt32(buffer2, 60);
                buffer2[num5 + 0x58] = 0;
                buffer2[num5 + 0x59] = 0;
                buffer2[num5 + 90] = 0;
                buffer2[num5 + 0x5b] = 0;
                for (int j = num3; j < buffer2.Length; j++)
                {
                    buffer2[j] = 0;
                }
                return a(A_0, buffer2, destinationArray);
            }
        }
        return false;
    }
    catch
    {
        return false;
    }
}

```

It's quite clear that a sort of check on DLL bytes is performed (note the hard-coded FILESIGN string) and, if we dig into the highlighted routine (the only one that would allow a 'true' value to be returned), we have confirmation of an integrity check:

```

d6.a(Byte[], Byte[], Byte[]) : Boolean
public static bool a(byte[] A_0, byte[] A_1, byte[] A_2)
{
    RSACryptoServiceProvider provider = new RSACryptoServiceProvider();
    provider.ImportCspBlob(A_0);
    return provider.VerifyData(A_1, new SHA1CryptoServiceProvider(), A_2);
}

```

But why do we care so much about this verification? The answer is really simple: to complete our work we'll need to alter that DLL (we'll talk about it later) so, we have to avoid this check. However, we are still trying to understand the *i.d* role, therefore let's come back focusing on it.

Only if a certain condition - this time on the *c7.d* (integer variable) – passes, our *i.d* property will be set to *true*. The mentioned condition:

```
if ( object.Equals(d, b.Length) )
```

compares the *c7.d* value with the length of the *b* thread array, threads that are in charge of verifying product integrity³:

```

c7.c() : Void
public static void c()
{
    b = new Thread[2];
    for (int i = 0; i < (b.Length - 1); i++)
    {
        b[i] = new Thread(new ThreadStart(c7.e));
        b[i].Start();
    }
    b[b.Length - 1] = new Thread(new ThreadStart(c7.d));
    b[b.Length - 1].Start();
}

```

The two thread-bound functions are just two of the latest seen routines. Therefore we can safely guess that *i.d* gets set if and only if both verifications get executed and passed with a positive outcome (i.e. product has not been corrupted).

But that's just the first place where our *i.d* is used. It also plays a role into this other piece of code:

The screenshot shows a Visual Studio window with a code editor and an analyzer window. The code editor displays a function call: `if (this.a(out A_1, out zero) && i.d)`. A red arrow points to the `i.d` variable. The analyzer window shows a tree view for `i.d : Boolean` with the following structure:

```

Analyzer
├── i.d : Boolean
│   └── Used By
│       ├── c7.d() : Void
│       └── cv.a(IntPtr&, UInt32&, IntPtr&, a9[]&) : Boolean
│           ├── Depends On
│           └── Used By
│               ├── cv.a(eg) : Boolean
│               └── i.cctor()

```

A value of `false` for *i.d* imply a return value of `false` for the whole routine! But, if we look at the caller⁴ (`cv.a(eg) : Boolean`), we'll soon understand we can't accept that result 😊

³ We can easily find `c7.c()` routine applying Analyze command on `c7.d`.

⁴ Here too through Analyze command.

```

private bool a(eg A_0)
{
    IntPtr ptr;
    uint num;
    IntPtr ptr2;
    a9[] aArray;
    if ((this.l != null) || (this.m != null))
    {
        return false;
    }
    this.l = A_0;
    if (this.a(out ptr, out num, out ptr2, out aArray)) ←
    {
        bool flag;
        try
        {
            if (br.DxCore_StartCapture(this.l.GetProcessID(), ptr, num, ptr2, (uint) aArray.Length, aArray))
            {
                this.e();
                this.a(ControlManagerNotificationCode.IsCapturing, null, null);
                flag = true;
            }
            else
            {
                goto Label_0083;
            }
        }
        finally
        {
            if (ptr != IntPtr.Zero)
            {
                Marshal.FreeHGlobal(ptr);
            }
            if (ptr2 != IntPtr.Zero)
            {
                Marshal.FreeHGlobal(ptr2);
            }
        }
        return flag;
    }
Label_0083:
    this.l = null;
    return false;
}

```

The red arrow indicates the routine where the famous *i.d* is used! If that routine doesn't return *true*, the more than explanatory *DxCore_StartCapture(...)* method doesn't get a chance to be invoked and this will undoubtedly affect the program functionality, right? :P Then, there are no doubts anymore: *i.d* must be *true*!

Let's start with the constructor (we already know its address: 0x1EA9C):

```

.method private hidebysig specialname rtspecialname static void .cctor() cil managed
{
    .maxstack 8
    L_0000: ldc.i4.0
    L_0001: stsfld bool i::a
    L_0006: ldstr ""
    L_000b: stsfld string i::b
    L_0010: ld sfld valueType [mscorlib]System.DateTime [mscorlib]System.DateTime::MinValue
    L_0015: stsfld valueType [mscorlib]System.DateTime i::c
    L_001a: ldc.i4.0
    L_001b: stsfld bool i::d
    L_0020: ret
}

```

The IL view allows us to find immediately the address to patch: the *ldc.i4.0* that we need to change into *ldc.i4.1* is at 001a offset from the beginning of the routine; therefore we find the byte to alter at:

$$(RVA) + (Tiny\ Header) + (offset) = 0x1EA9C + 1 + 0x1a = 0x1EAB7$$

We change it from 0x16 to 0x17.

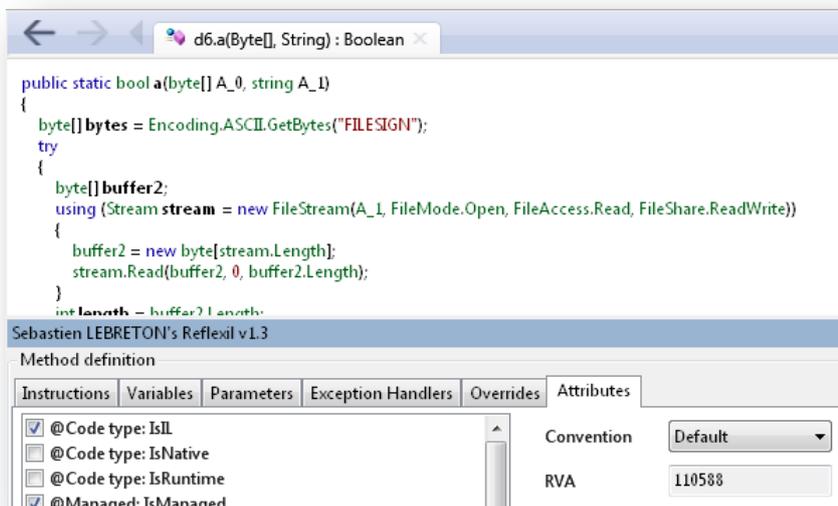
Before

Dxtory-cleaned.exe	
VA	0041EAB7
RVA	0001EAB7
File Offset	0001CCB7
Offset	0 1 2 3 4 5 6 7 8 9 A
0001CCB0	00 0A 80 E5 06 00 04 16 80 E6 06
0001CCC0	13 30 02 00 7D 00 00 00 2E 01 00
0001CCD0	01 6F EB 00 00 01 80 80 81 82 83

After

Dxtory-cleaned.exe	
VA	0041EAB7
RVA	0001EAB7
File Offset	0001CCB7
Offset	0 1 2 3 4 5 6 7 8 9
0001CCB0	00 0A 80 E5 06 00 04 17 80 E6
0001CCC0	13 30 02 00 7D 00 00 00 2E 01
0001CCD0	01 6F EB 00 00 01 80 80 81 82

Let's save the file and switch to the *d6.a* routine. A word of warning here: both previously seen routines have *d6.a* as their name. We'll patch the first overload (usually it's preferable to alter the deeper routine but, since the latter is called only from its homonymous, we can save some processing time directly patching the caller):



We resort to our trusted CFF Explorer and reach the RVA 110588 = 0x1AFFC (remember to skip the fat header) in order to apply the following change:

from:

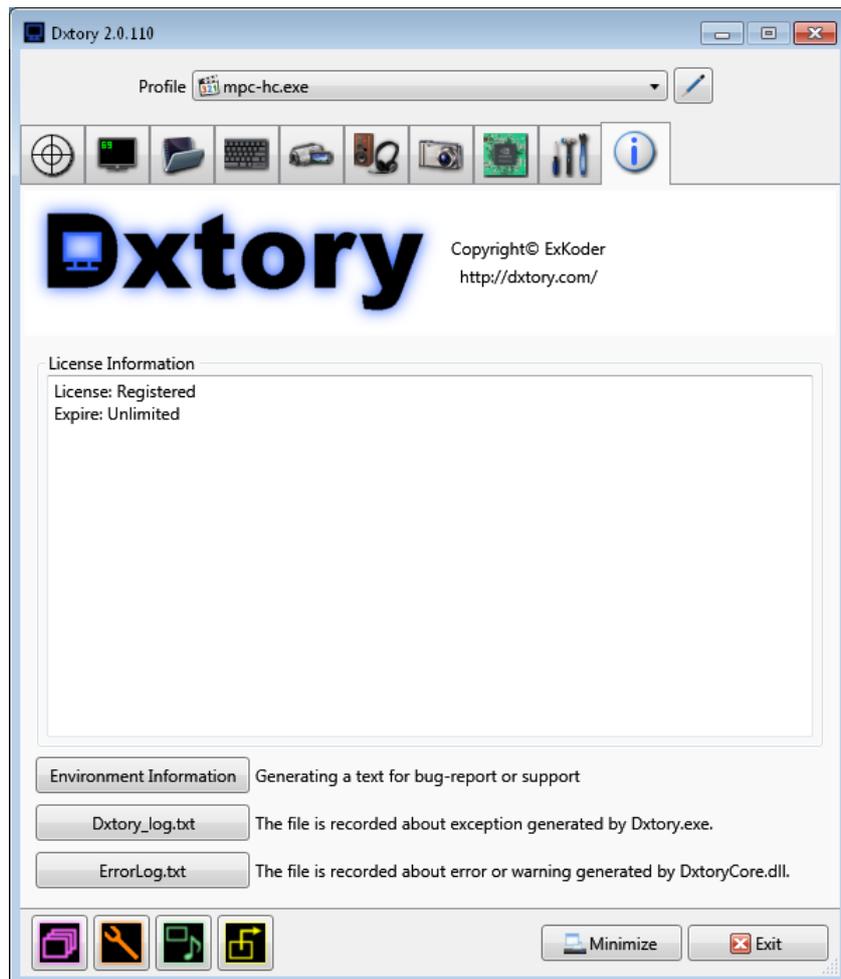
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00019200									28	F1	02	00	0A				(ñ..

to:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00019200									17	2A	00	00	00				*...

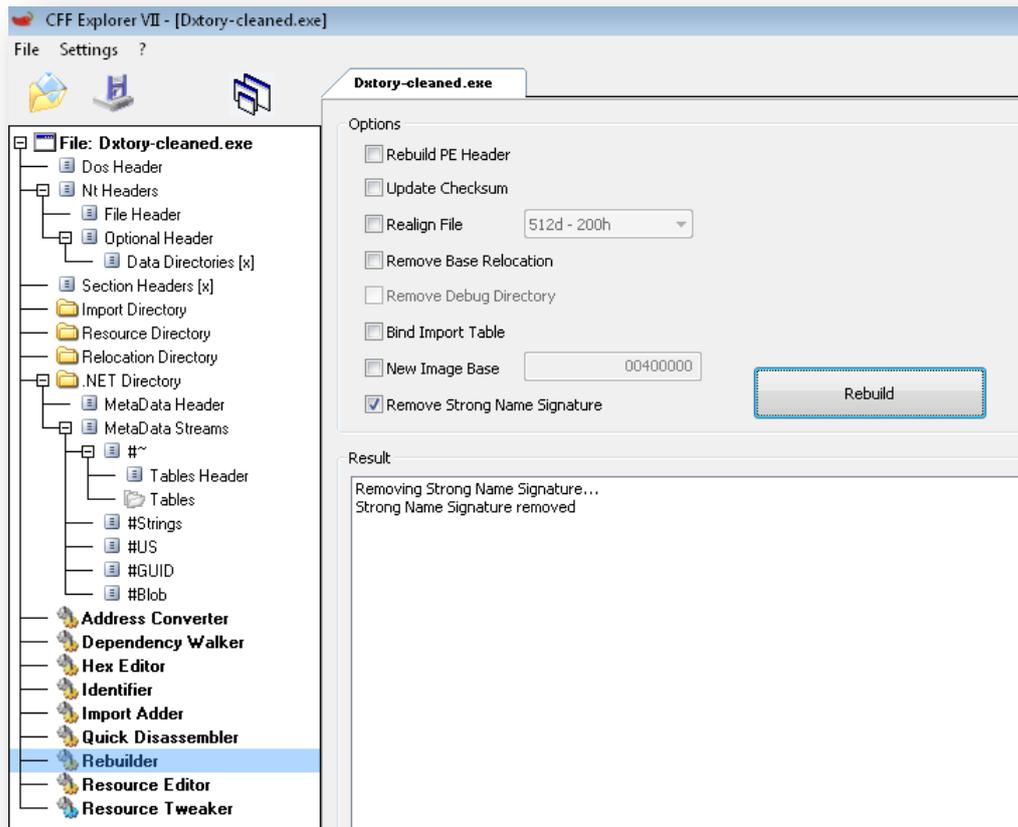
Basically we patch the routine so it will always return *true* ;)

If we launch now the program, the starting nag and the author site redirection disappear and the about tab shows we are fully licensed ;)

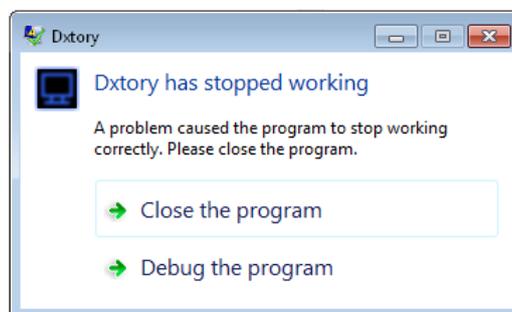


Make our executable run on every system

Remember, however, that the only reason the application runs correctly is, because of our request, .NET framework has ignored StrongName validation. To make our program run on every system (with or without strongname validation enabled) we should remove the now invalid StrongName, simply choosing the "Remove Strong Name Signature" CFF Rebuilder option.



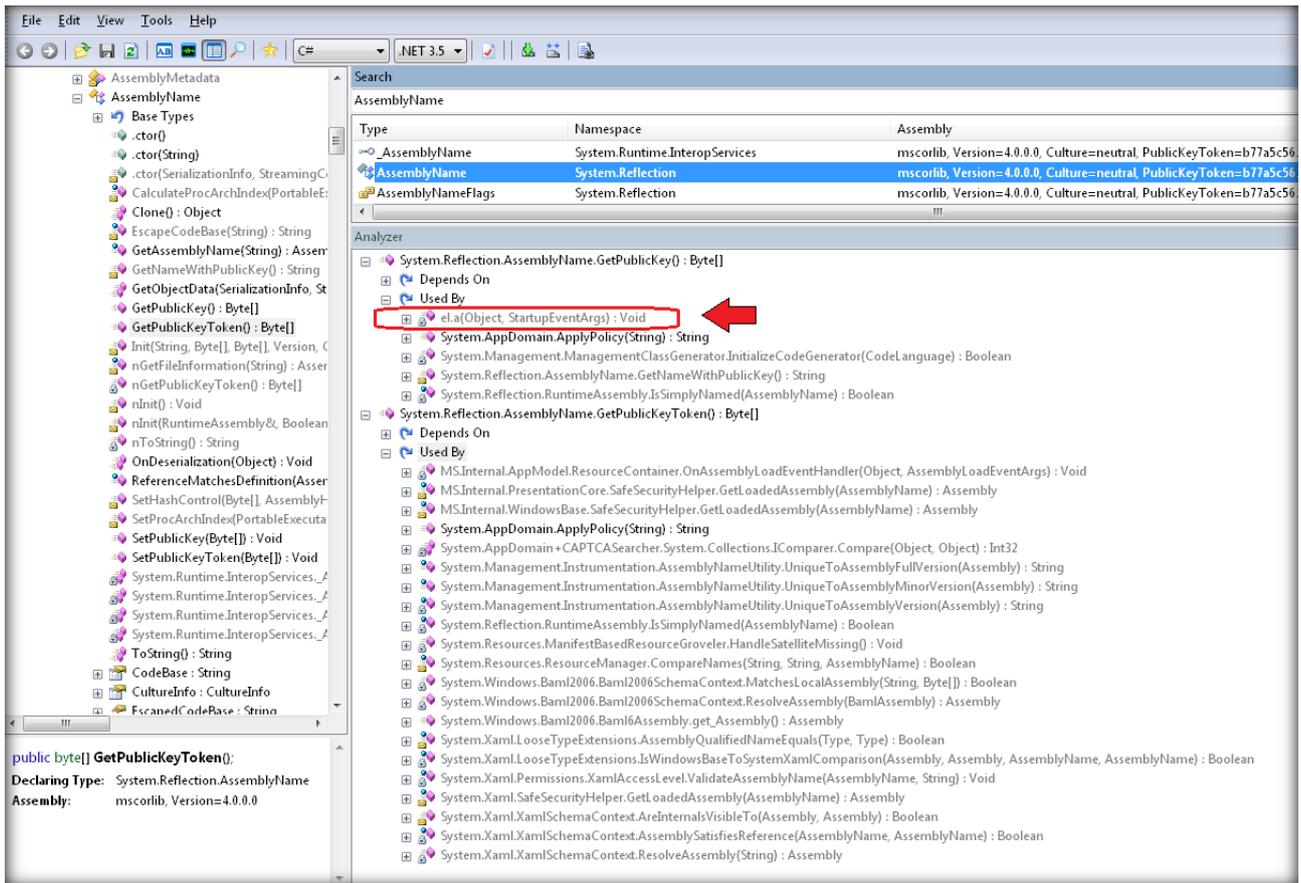
Let's save the executable and run again the program and ... bang!



Rush another time towards the log folder (with the hope to find some useful error stack trace :P) but this time app crashes without writing anything. The problem showed up once we removed the Strong Name so we'll look in Reflector if the program uses `PublicKey` or `PublicKeyToken`⁵.

Switch to Reflector and search for the `AssemblyName` class. Once we located it, look for references to its `getPublicKey()` and `getPublicKeyToken()` methods:

⁵ For a complete strong name description refer to the official documentation and/or to the available tutorials. See, for example, <http://tuts4you.com/download.php?view.1197> and <http://tuts4you.com/download.php?view.2701>.



`GetPublicKey` has only one reference (see the highlighted routine in the picture). Let's go there :P

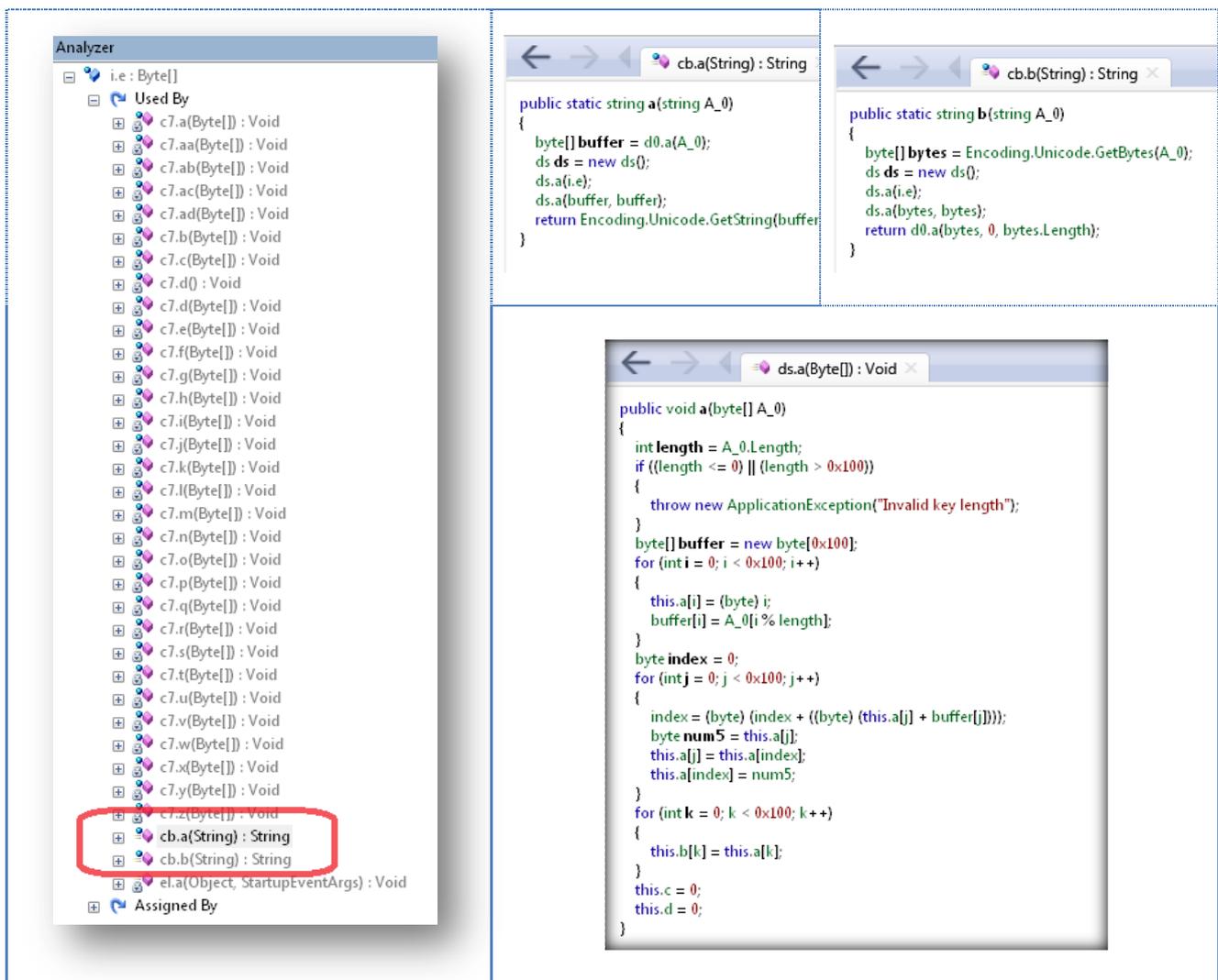
```

ela(Object, StartupEventArgs) : Void
{
    int num = new Random().Next(0x10);
    while (num > 0)
    {
        num--;
        DateTime time1 = DateTime.Now;
    }
    DateTime now = DateTime.Now;
    Thread.Sleep(150);
    long num2 = DateTime.Now.ToBinary() - now.ToBinary();
    if ((num2 > 0x7a1201) && (num2 < 0x989680))
    {
        if (!br.ac)
        {
            MessageBox.Show("Fail Init Module.", "Dxtory", MessageBoxButtons.OK, MessageBoxIcon.Hand);
            base.Shutdown();
        }
        else
        {
            byte[] publicKey = Assembly.GetEntryAssembly().GetName().GetPublicKey();
            i.e = new byte[publicKey.Length - 12];
            Array.Copy(publicKey, 12, i.e, 0, i.e.Length);
            c7.c();
            c7.b();
            if (!i.a)
            {
                TrialDialog dialog = new TrialDialog();
                if (!dialog.ShowDialog().Value)
                {
                    base.Shutdown();
                    return;
                }
            }
        }
    }
    if (c7.a())
}

```

The code reads the `PublicKey` from assembly metadata and copies it into `i.e` property which, in turn, is referred in some kind of string decryption routines⁶:

⁶ And, probably, that's the reason for the crash.

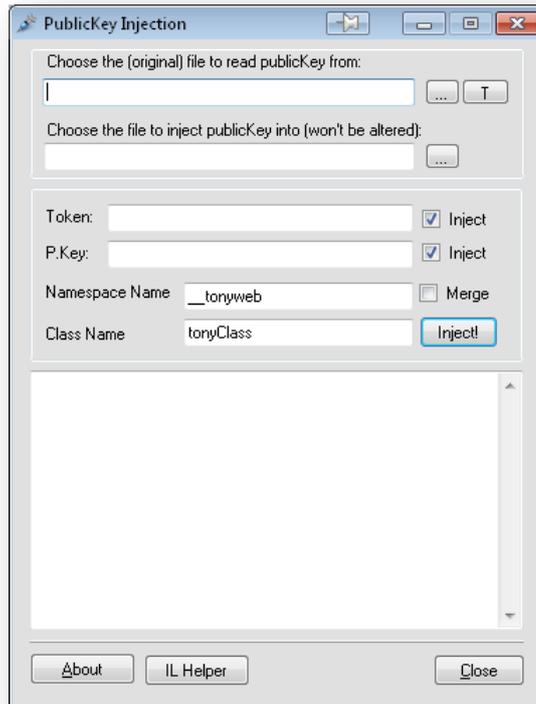


A quick way to solve this problem is to inject the right PublicKey and force the code to put the original injected public key into *i.e* and not the one read from the assembly (which is now *null* because of the Strong Name removal).

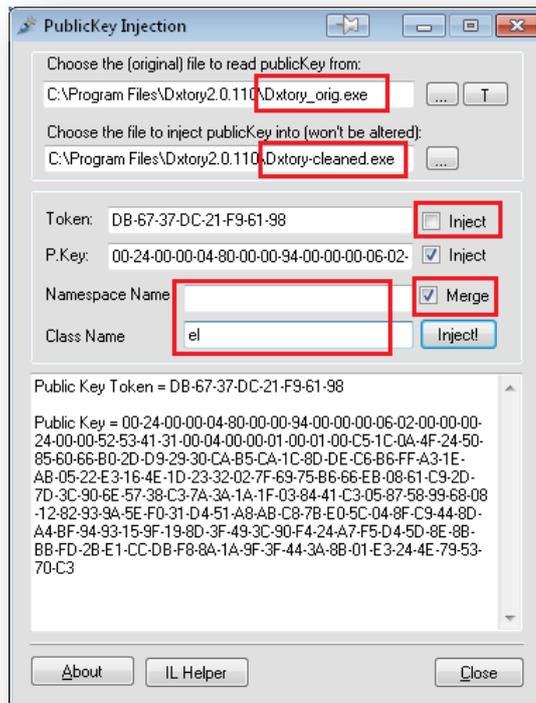
To do this kind of modifications we can use *Mono.Cecil* library which allows us, in a really simple way, to manipulate our assemblies adding, removing or simply changing methods, classes, namespaces, variables, strings, ecc. Suffice to say that the same Reflexil plugin uses this library to perform his services :D

I "stole" the idea to embed the original PublicKey from whoknows (thank you mate!) and coded the little tool you should find attached to this tutorial: let's start it!⁷ ;)

⁷ As Administrator if we have no writing rights on main app folder (Vista/7/2008).

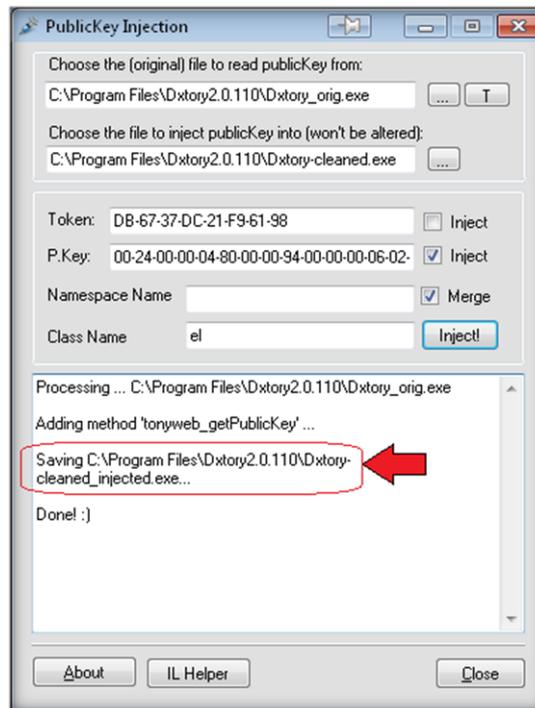


This tool can inject methods which return byte arrays matching the PublicKey and PublicKeyToken of a reference assembly. Suppose we worked this far on the Dxtory-cleaned.exe file and we have the original target renamed as Dxtory_orig.exe. Drag Dxtory_orig.exe onto the first text field (or click on the ellipsis to browse the file system); the Token and PublicKey (P.Key) text fields should fill up automatically. Choose our Dxtory-cleaned.exe as the destination file and uncheck the 'Inject' checkbox near Token (in this target we don't need it).

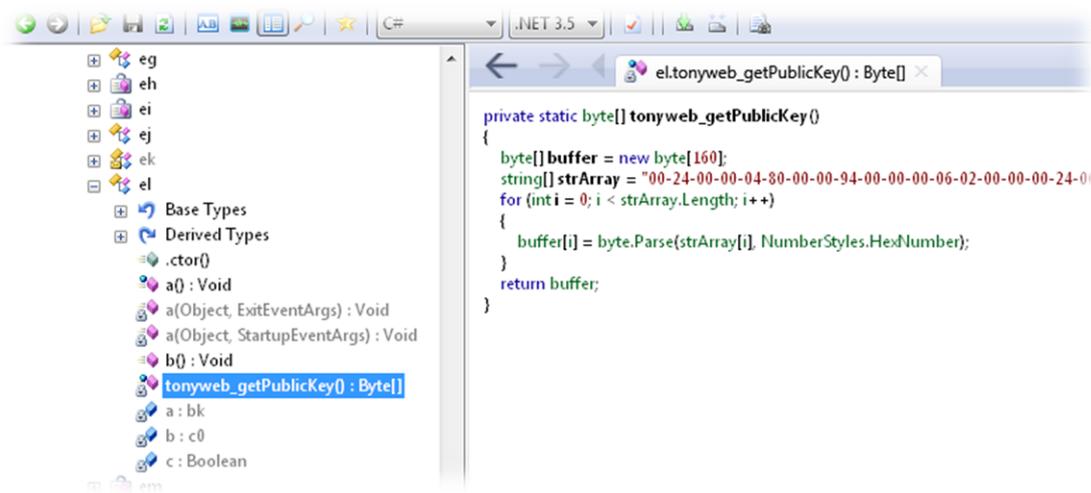


We can place the new methods in a new class or “merge” them into existing types: for convenience, we inject the Dxtory assembly original PublicKey in the same class where the method that requires it resides in: *el* (default namespace).

Click on Inject! button and, if all goes well, we’ll get a new Dxtory-cleaned_injected.exe file with a *tonyweb_getPublicKey()* method into the selected class and namespace.

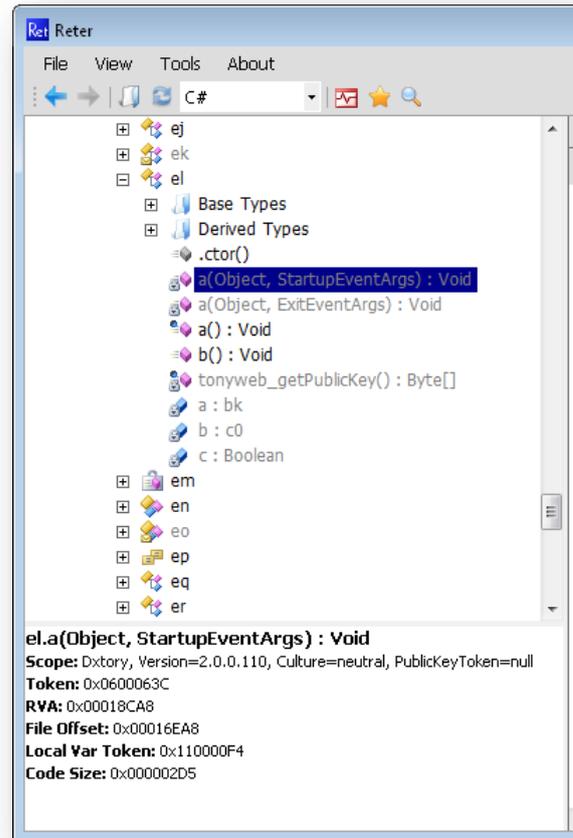
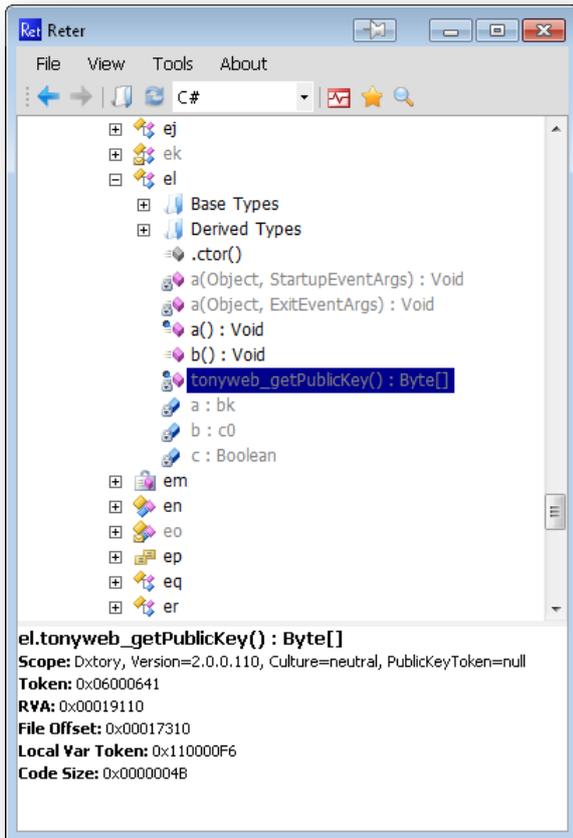


If we open the resulting file in Reflector we can actually see the routine.

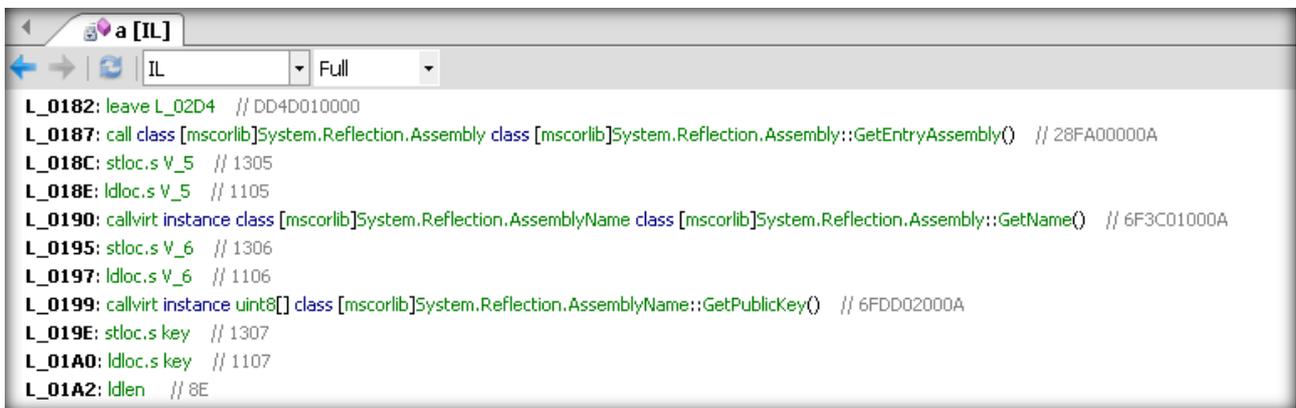


Now we have to use this new method to read the public key instead of allowing the program reading it from assembly metadata.

To do this, open our new file in Reter Decompiler (an excellent decompiler written by yck1509) and collect the needed information on both the added method and his future caller ...



... and on the code we plan to alter:



What we really want is, in the following code

```
byte[] publicKey = Assembly.GetEntryAssembly().GetName().GetPublicKey();
i.e = new byte[publicKey.Length - 12];
Array.Copy(publicKey, 12, i.e, 0, i.e.Length);
```

the line

```
byte[] publicKey = Assembly.GetEntryAssembly().GetName().GetPublicKey();
```

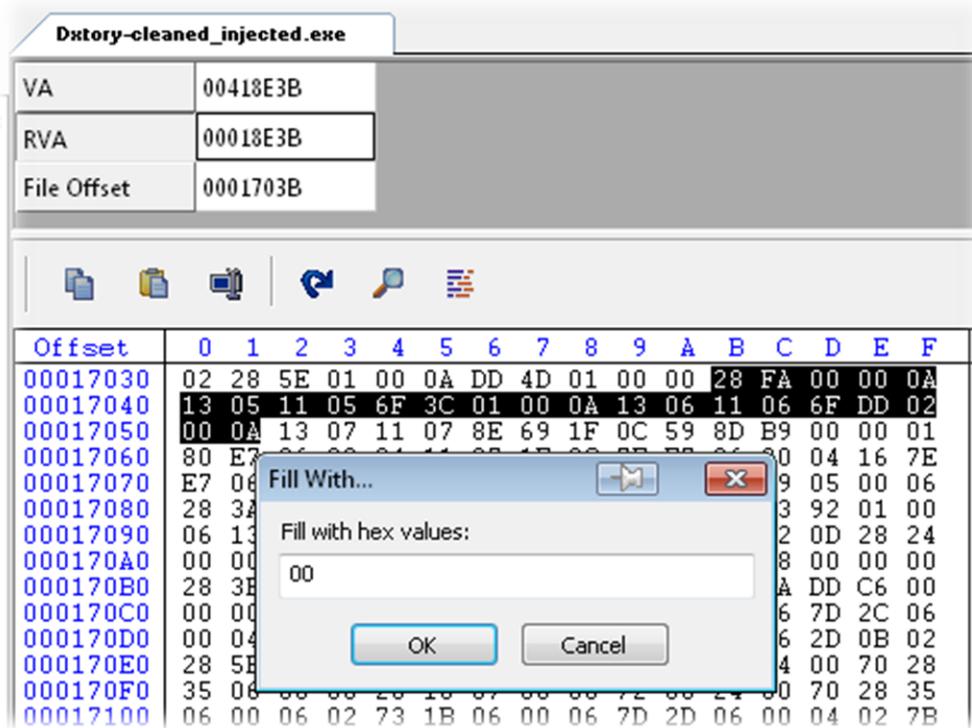
becoming

```
byte[] publicKey = el.tonyweb_getPublicKey();
```

To do that we just have to calculate the patch places and apply the following changes.

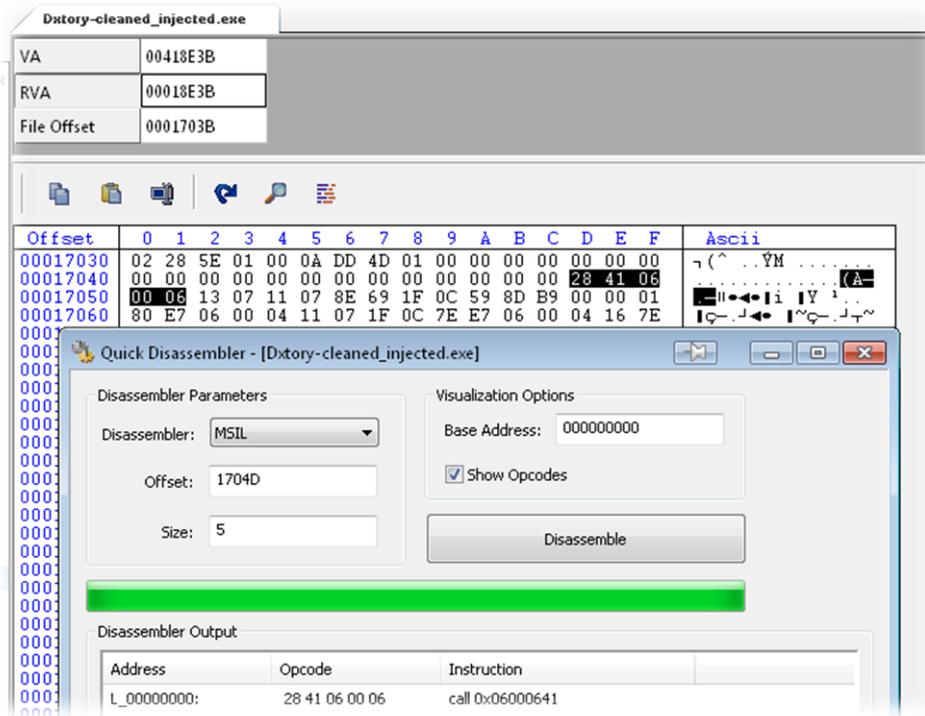
RVA: <routine_rva> + <fat_header_len> + <code_offset> = 0x00018CA8 + 0C + 0187 = 0x18E3B
File Offset: 0001703B

Let's nop beforehand the instructions we need to replace:

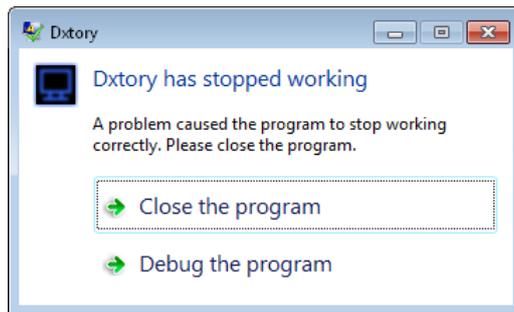


And prepare the new call:

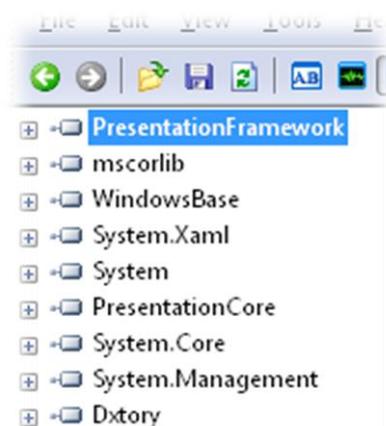
op-code for call instruction:	0x28
token of the routine to call:	0x06000641
bytes to write:	28 41 06 00 06



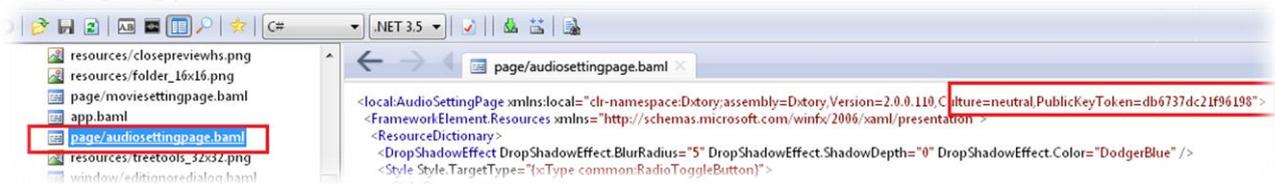
Save the file and try again running the exe and BAM!:



Damn, it crashes again! ☹️ And no log written once more. Think with me, the issue must still be related to Strong Name removal. Browsing the program in Reflector we have certainly seen references to classical WPF (Windows Presentation Foundation) namespaces like *PresentationFramework* and *System.Xaml*.



In that framework, UI and resource description files – baml/xaml – refer to assemblies through their complete names, therefore including public key token (thank you to Kurapica for this hint!) . If we browse main executable resources from Reflector we can see, for example:



Open up our hex editor and then execute this replacement:

```
Find: ", PublicKeyToken=db6737dc21f96198"
Replace: " "
```

Note the replacement with a string of blanks of the same length!



Now **launch** again our executable and finally ... **WoooooW it works !**

The quest for the watermark

Although the application now looks licensed, its features are practically equivalent to the trial version⁸: the watermark is still there on the captures. :-/

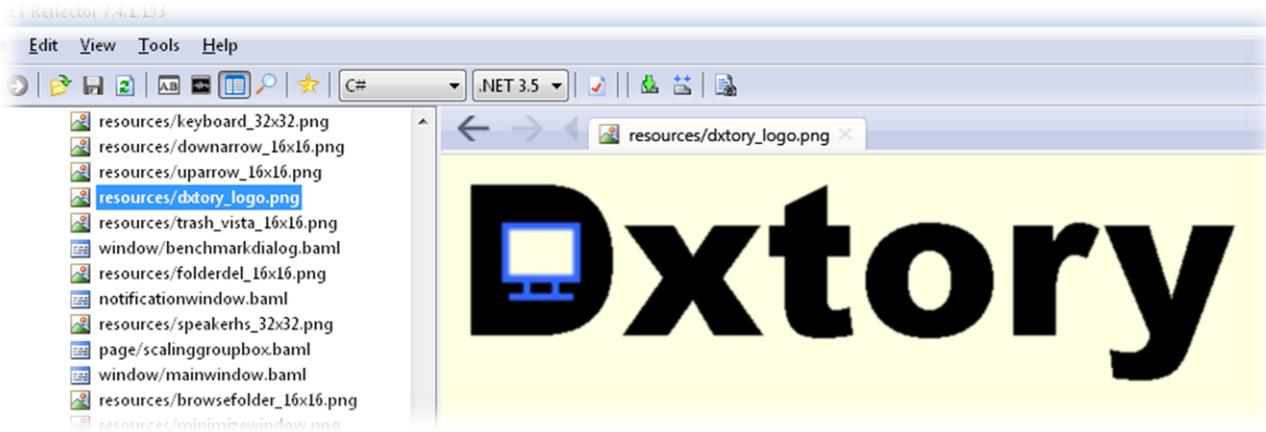
The watermark picture must be located somewhere and used by the program during video recording. Most likely we already spotted two suspect *png* files in main application folder: *Src16x9_Dest4x3.png* and *Src16x10_Dest4x3.png* both representing a sort of watermark mask.



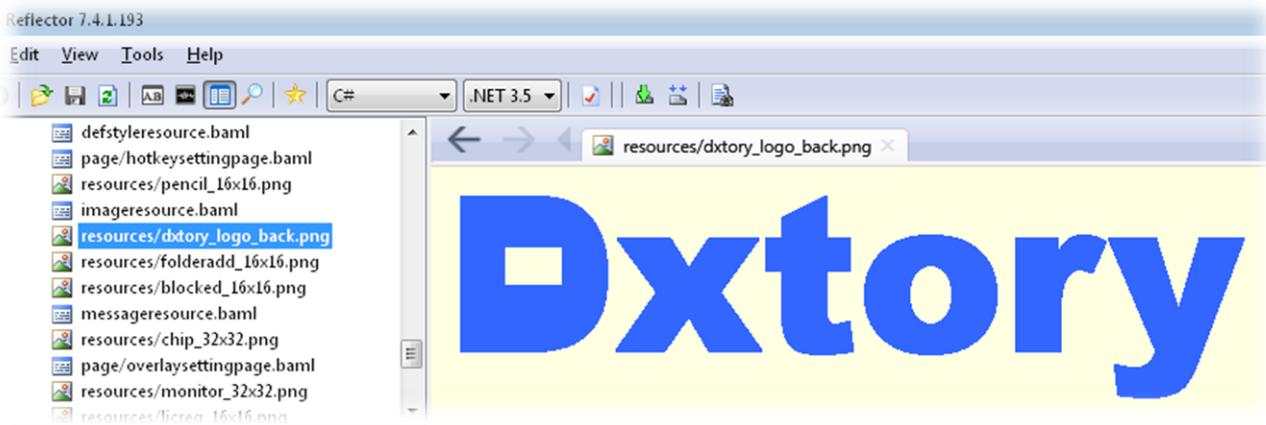
But that would be too easy :D Renaming or moving away these pictures from executable folder doesn't make any difference, the watermark still stands (ehh, clearly those two files have the only purpose to divert our attention ☺).

Now our first thoughts are towards the resources: the program has to retrieve somewhere the image it places in overlay. Open the file in .NET Reflector and sift through resources; two specific items attract our attention:

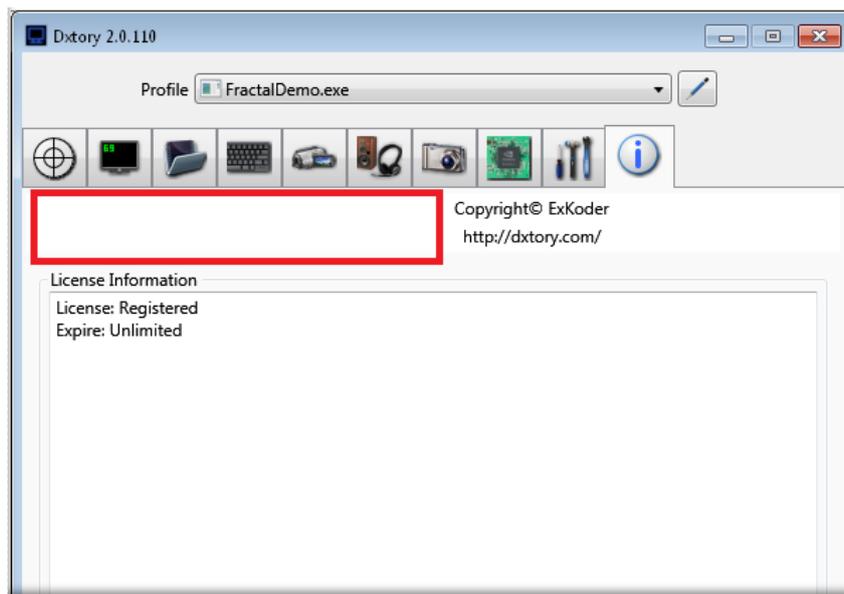
⁸ It's clear we removed some limitations like NAG and home redirection.



and



Yet, even if you remove the suspicious dxtory_logo.png and dxtory_logo_back.png we don't touch at all the watermark, rather we just end up ruining the user interface.

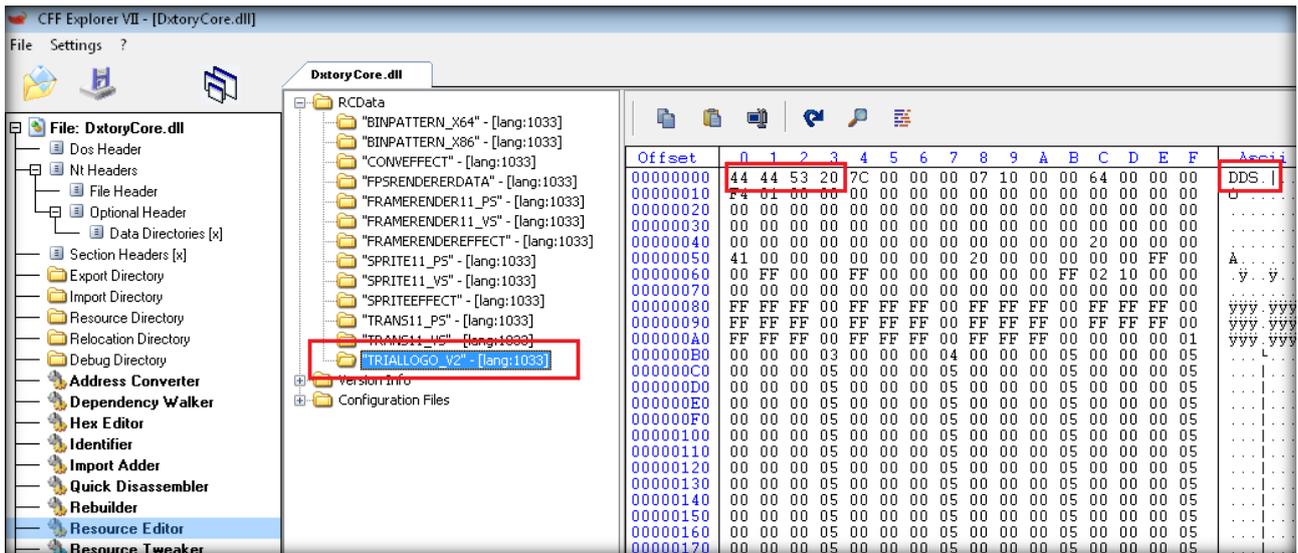


Besides these pictures, there is nothing more suspicious among resources and, even looking into the code, we find no obvious references to the watermark. We just have to look better around: what is available?

Patching phase 2: All together in the native world

Discover the resource

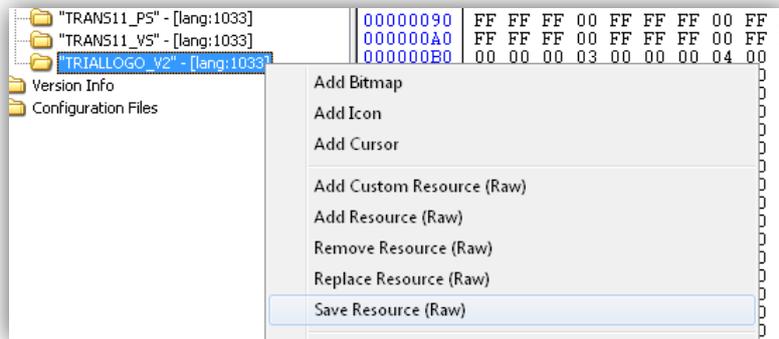
Recall what we saw earlier: the main .NET executable takes care of checking the DxcCore.dll file integrity; this is a very important evidence: our first suspect is therefore that file. ☺ Let's analyze its resources (always with the great CFF Explorer) and we soon have this view in front of us:



Stood out instantly a name: **TRIALLOGO_V2**.

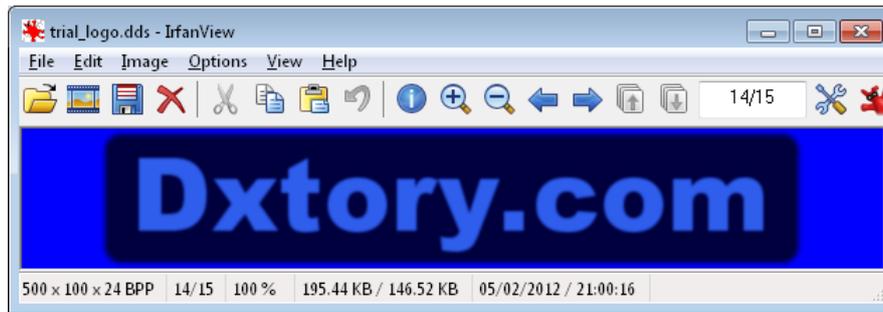
It's easy guessing this is a DDS (Direct Draw Surface) resource type; since I never worked with Direct-X I knew nothing about them: on MSDN we can find a lot of information and DDS header is described in great detail. There you can find the offsets of picture's height and width and, in the first tries, I simply tried to zero-out picture dimensions ... but let's stay calm and don't move too far from the main topic.

If we save that resource:

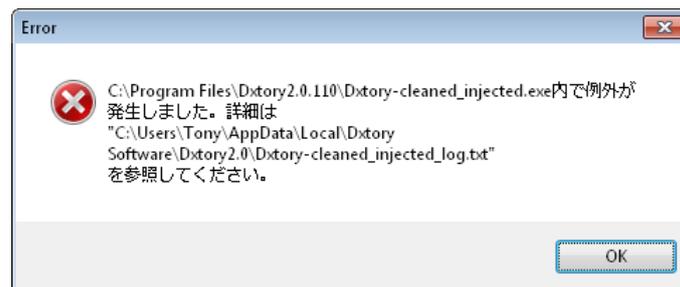


and open it with a viewer like IrfanView we can confirm we finally found the watermark we're looking for⁹ ;)

⁹ I changed to blue Irfanview main window background color only to point out the image semi-transparent background.



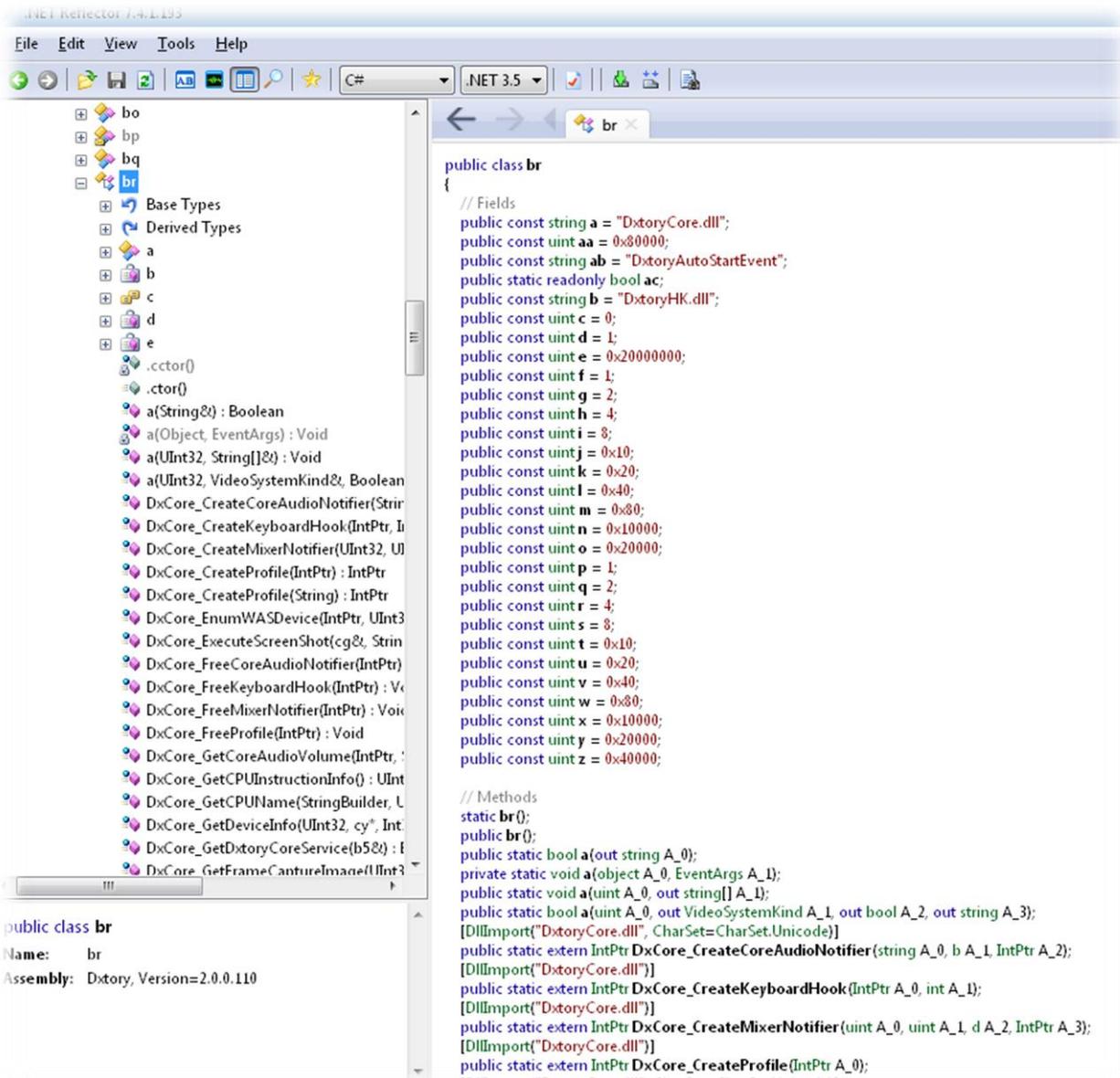
We might consider making a backup of the DLL, removing the resource and try starting the application. If we did such a modification, though, we would realize quickly we won't solve the matter so easily:



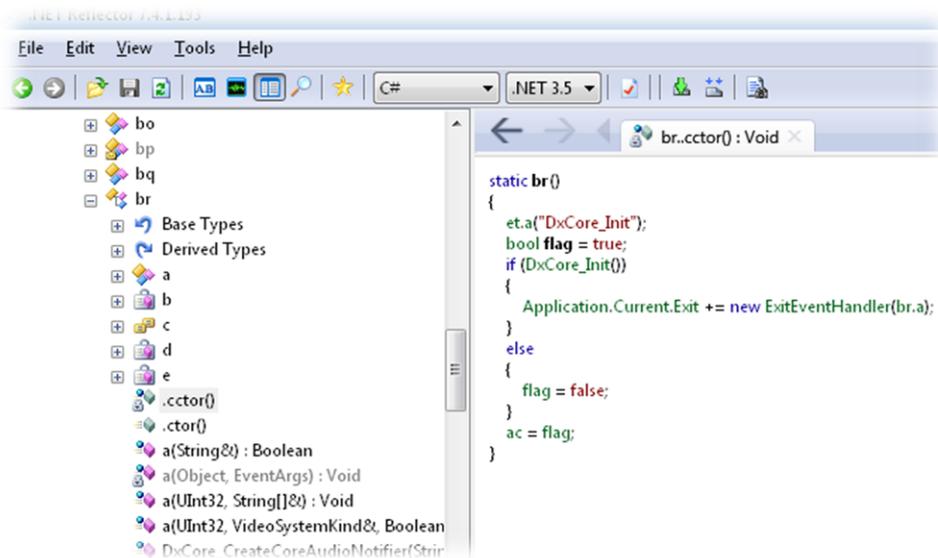
Since the program ask us to look at the log so kindly, we open it and find:

```
=== 2012/02/05 ===  
[Module]  
Dxtory-cleaned_injected.exe 2.0.110  
  
[Process]  
ID: 8140  
  
[Thread]  
MainThread (1)  
  
[Error Message]  
The type initializer for 'br' threw an exception.  
  
[StackInfo]  
at e1.a(Object A_0, StartupEventArgs A_1)
```

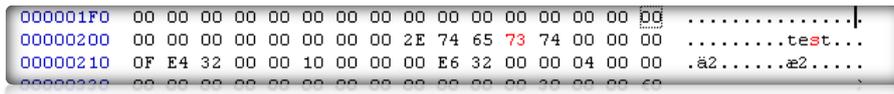
A quick look in Reflector makes us understand that *br* class is the 'interface' towards the program injected DLL(s) like *Dxcore.dll* and *DxtoryHK.dll*.



The error message states there was a problem during class initialization. Since the instance constructor is “empty”, all must be done into the static constructor:



It's quite clear, from the few instructions we see, that something went wrong during the DLL initialization. A doubt rises. Restore original DLL and try to just change a single byte (like one of the characters of first section name):

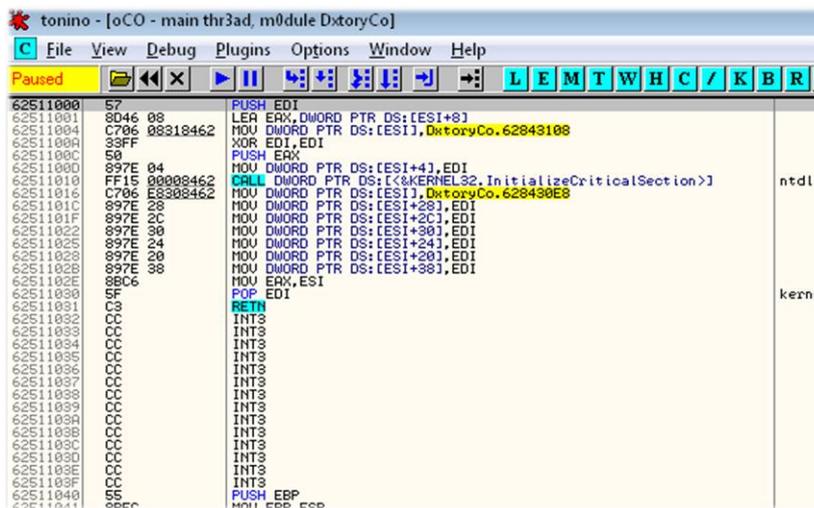


Save the file, start again the program and BAM: same error!

For quite a while I kept looking for some other integrity check in the .NET code (remember we've already patched one of them previously) but I simply wasted my time. Only after I finished the ideas, I choose to take a look at the native code.

OllyDbg to the rescue

Start by opening *original* DLL with Olly; wait for LOADDLL.EXE work to be done and soon we shall see the beginning of code section¹⁰.



¹⁰ Note that the addresses will be almost certainly different on your system.

If we open it also in CFF Explorer, we can get entry-point RVA: **0031D80F**

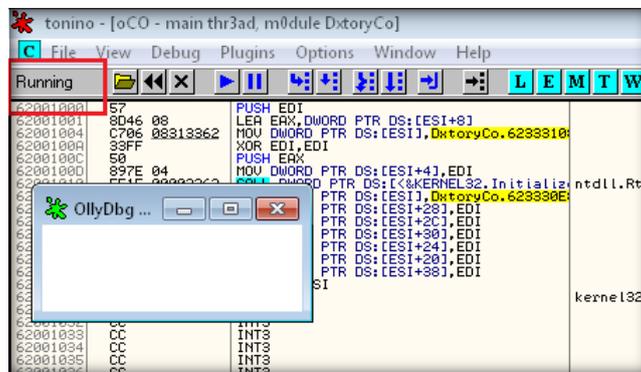
SizeOfUninitializedData	00000134	Dword	00000000	
AddressOfEntryPoint	00000138	Dword	0031D80F	.text
BaseOfCode	0000013C	Dword	00001000	

which, added to current ImageBase, allows us to reach the DLL entry-point:

```

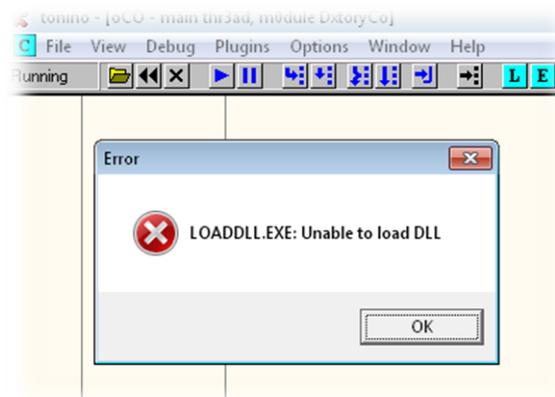
6282D807  E8 F71B0000  CALL DxtoryCo.6282D80B
6282D80E  C3          RETN
6282D80F <ModuleEntryPoint> 8BFF       MOV EDI,EDI
6282D811  55          PUSH EBP
6282D812  8BEC       MOV EBP,ESP
6282D814  837D 0C 01  CMP DWORD PTR SS:[EBP+0],1
6282D818  75 05      JNZ SHORT DxtoryCo.6282D81F
6282D81A  E8 E4110000 CALL DxtoryCo.62831A0C
6282D81F  FF75 08    PUSH DWORD PTR SS:[EBP+8]
6282D822  8B4D 10    MOV ECX,DWORD PTR SS:[EBP+10]
6282D825  8B55 0C    MOV EDX,DWORD PTR SS:[EBP+C]
6282D828  E8 ECFEFFFF CALL DxtoryCo.6282D719
6282D82D  59          POP ECX
6282D82E  5D          POP EBP
6282D82F  C2 0C00   RETN 0C
6282D832  8BFF       MOV EDI,EDI
  
```

Press F9 (run) and see that the DLL “runs”:

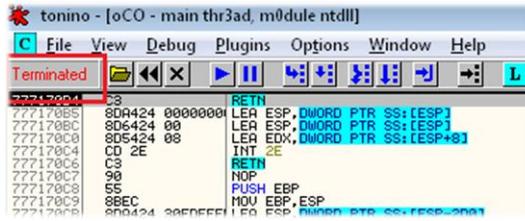


Ok, all seems fine till here: Olly allows us to work flawlessly with the executable. Try now opening the *modded* DLL (the one with the altered text section name for example) just out of curiosity: did we already delete it? :P Mine, renamed as *DxtoryCore_.dll*, was still in the recycled bin :D

We can open the DLL in Olly, but if we run it (F9) here’s what happens:

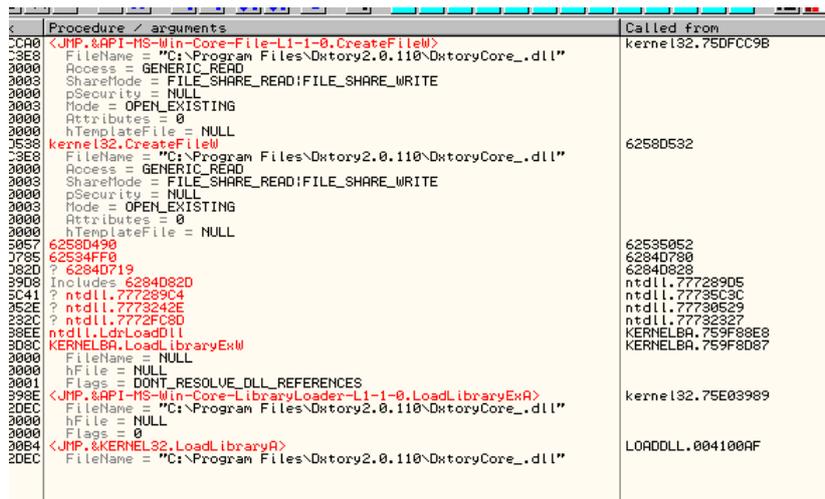
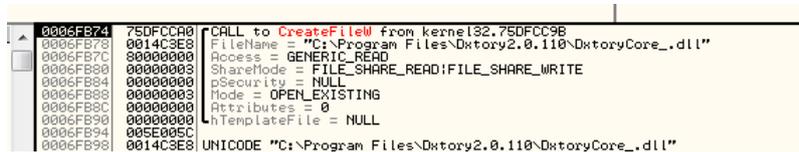
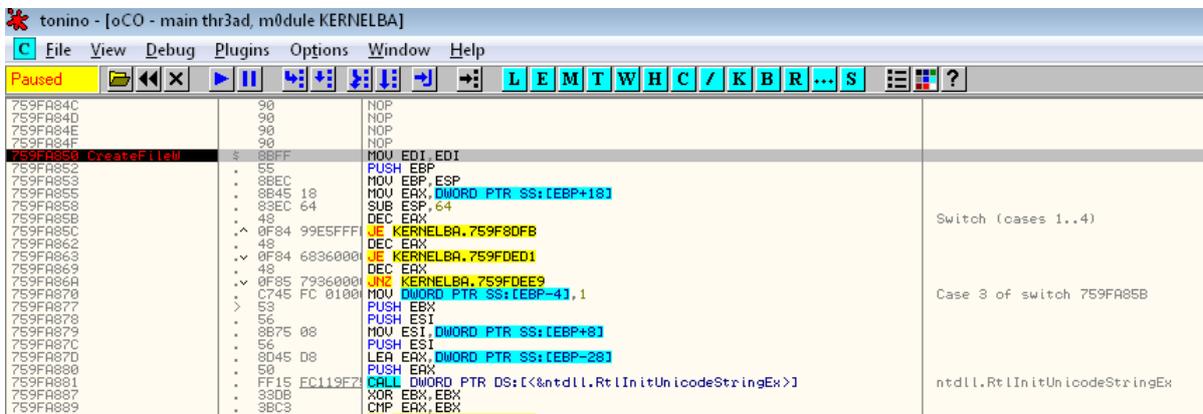


At this point we have no more choices: if we click on OK we can’t expect anything other than this:



The logical conclusion is that the integrity check we were looking for is inside the same DLL: that's the reason why we could not find anything in .NET code ☺

Well, now our new goal is to find *where* this integrity check gets executed ;) In order to state that the file is corrupted, the code undoubtedly has to "read" it and, to do that, has to "open" it beforehand ... when we're at the 'Terminated' state (see picture), put a couple of breakpoints on *CreateFileA* and *CreateFileW* API and reload the DLL (CTRL+F2).



CreateFileW - break 1

The first break, consequence of the loadll.exe request to load our DLL, is already suspicious: as we can see from the call stack no "proprietary" piece of code gets executed, but there are a number of "calls" coming from un-owned memory regions.

Apart from that we have no particular clue that DLL code gets executed. Let's move on, only for the moment, to the next break by pressing F9:

```

j8, 75A240E9, NotifyMountMgr+0F4, 75A245F3, DeleteVolumeMountPointW+0A4, De
0006ED74 75DFCC9B CALL to CreateFileW from kernel32.75DFCC9B
0006ED78 0006EF1C FileName = "C:\Windows\system32\rsaenh.dll"
0006ED7C 80000000 Access = GENERIC_READ
0006ED80 00000001 ShareMode = FILE_SHARE_READ
0006ED84 00000000 pSecurity = NULL
0006ED88 00000003 Mode = OPEN_EXISTING
0006ED8C 00000080 Attributes = NORMAL
0006ED90 00000000 hTemplateFile = NULL
0006ED94 0006E09C UNICODE "C:\Windows\system32\rsaenh.dll"
0006ED98 0006EF1C
0006ED9C 0006E0D0
0006EDA0 752A92D0 RETURN to 752A92D0 from kernel32.CreateFileW
0006EDA4 0006EF1C UNICODE "C:\Windows\system32\rsaenh.dll"
0006EDAC 80000000
0006EDB0 00000001
0006EDB4 00000003
0006EDB8 00000000
0006EDBC 00000000
0006EDC0 00000001
0006EDC4 00000000
0006EDC8 00000000
0006EDCC 00000001
0006EDD0 00000000
0006EDD4 00000000
0006EDD8 00000001
0006EDE0 0006EE64
0006EDE4 0006EE60
0006EDB4 752A3F87
0006EDB8 752A3E88
0006EDBC 752A447A
0006EDC0 6258D0EB
0006EDC4 6258D64B
0006EDC8 0178C57C
0006EDCC 0188C594
0006EDD0 00000000
0006EDD4 0000000C
0006EDD8 017C0020
0006EDE0 013F0000
0006EDE4 003CC600
0006EDE8 003CC600
0006EDEC 003CC600
0006EDF0 62535057
0006EDF4 6284D785
0006EDF8 62530000
0006EDFC 00000001
0006ED00 00000000
0006ED04 6284D82D
0006ED08 777289D8
0006ED0C 62530000
0006ED10 00000001
0006ED14 00000001
0006ED18 00000000
0006ED1C 00000000
0006ED20 77735C41
0006ED24 7773052E
0006ED28 7773232C
0006ED2C 759F8DEE
0006ED30 759F8D8C
0006ED34 00000000
0006ED38 00000000
0006ED3C 00000001
0006ED40 75E0398E
0006ED44 001C2DEC
0006ED48 00000000
0006ED4C 00000000
0006ED50 004100B4
0006ED54 001C2DEC

```

If we look now at the call stack we could already find something interesting ☺

Address	Stack	Procedure / arguments	Called from	Frame
0006ED74	75DFCC9B	<JMP.&API-MS-Win-Core-File-L1-1-0.CreateFileW>	kernel32.75DFCC9B	0006ED9C
0006ED78	0006EF1C	FileName = "C:\Windows\system32\rsaenh.dll"		
0006ED7C	80000000	Access = GENERIC_READ		
0006ED80	00000001	ShareMode = FILE_SHARE_READ		
0006ED84	00000000	pSecurity = NULL		
0006ED88	00000003	Mode = OPEN_EXISTING		
0006ED8C	00000080	Attributes = NORMAL		
0006ED90	00000000	hTemplateFile = NULL		
0006EDA0	752A92D0	kernel32.CreateFileW	752A92CA	0006ED9C
0006EDA4	0006EF1C	FileName = "C:\Windows\system32\rsaenh.dll"		
0006EDAC	80000000	Access = GENERIC_READ		
0006EDB0	00000001	ShareMode = FILE_SHARE_READ		
0006EDB4	00000000	pSecurity = NULL		
0006EDB8	00000003	Mode = OPEN_EXISTING		
0006EDBC	00000080	Attributes = NORMAL		
0006EDC0	00000000	hTemplateFile = NULL		
0006EDD0	752A907B	752A920F	752A907B	0006E0D0
0006EDD8	00000001	Arg1 = 00000001		
0006EDDC	0006EF1C	Arg2 = 0006EF1C		
0006EDE0	0006EE64	Arg3 = 0006EE64		
0006EDE4	0006EE60	Arg4 = 0006EE60		
0006EDB4	752A3F87	752A9005	752A3F82	0006EEB0
0006EDB8	752A3E88	752A9005	752A46E1	0006F138
0006EDBC	752A447A	752A9005	752A647A	0006F1F4
0006EDC0	6258D0EB	? advapi32.CryptAcquireContextW	DxtoryCo.6258D0EB	0006F258
0006EDC4	6258D64B	DxtoryCo.6258D64B	DxtoryCo.6258D64B	0006F890
0006EDC8	0178C57C	Arg2 = 0178C57C		
0006EDCC	0188C594	Arg3 = 0188C594		
0006EDD0	00000000	Arg4 = 00000000		
0006EDD4	0000000C	Arg5 = 0000000C		
0006EDD8	017C0020	Arg6 = 017C0020		
0006EDE0	013F0000	Arg7 = 013F0000		
0006EDE4	003CC600	Arg8 = 003CC600		
0006EDE8	003CC600	Arg9 = 003CC600		
0006EDFC	62535057	DxtoryCo.62535057	DxtoryCo.62535052	0006F8FC
0006ED00	6284D785	DxtoryCo.62534FF0	DxtoryCo.6284D780	0006FC08
0006ED04	62530000	Arg1 = 62530000		
0006ED08	00000001	Arg2 = 00000001		
0006ED0C	00000000	Arg3 = 00000000		
0006ED10	6284D82D	DxtoryCo.6284D719	DxtoryCo.<ModuleEntryPoint>+19	0006FC48
0006ED14	777289D8	DxtoryCo.<ModuleEntryPoint>	ntdll.777289D5	0006FC54
0006ED18	62530000	Arg1 = 62530000		
0006ED1C	00000001	Arg2 = 00000001		
0006ED20	00000000	Arg3 = 00000000		
0006ED24	77735C41	? ntdll.777289C4	ntdll.77735C3C	0006FC74
0006ED28	7773052E	? ntdll.7773242E	ntdll.77730529	0006FD68
0006ED2C	7773232C	? ntdll.7772FC8D	ntdll.77732327	0006FED4
0006ED30	759F8DEE	ntdll.LdrLoadDll	KERNELBA.759F8DE8	0006FF08
0006ED34	759F8D8C	KERNELBA.LdrLoadLibraryExW	KERNELBA.759F8D87	0006FF40
0006ED38	00000000	hFile = NULL		
0006ED3C	00000000	hFile = NULL		
0006ED40	75E0398E	Flags = DONT_RESOLVE_DLL_REFERENCES	kernel32.75E03989	0006FF60
0006ED44	001C2DEC	<JMP.&API-MS-Win-Core-LibraryLoader-L1-1-0.LoadLibraryExA>		
0006ED48	00000000	hFile = NULL		
0006ED4C	00000000	hFile = NULL		
0006ED50	004100B4	Flags = 0	LOADDLL.004100AF	0006FF80
0006ED54	001C2DEC	<JMP.&KERNEL32.LoadLibraryA>		
0006ED58	001C2DEC	FileName = "C:\Program Files\Dxtory2.0.110\DxtoryCore_.dll"		

The *CryptAcquireContextW* routine, symptom of some cryptographic operations in progress, catches our attention; obviously very suspicious. Let's follow the highlighted address: **6258D0EB**.

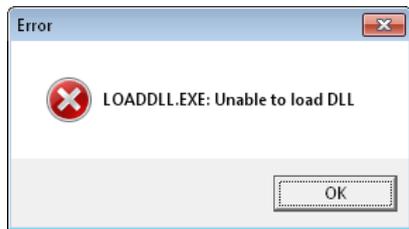
That is the “crypto-routine” we saw previously, the one where the *signature* gets calculated and verified: no doubts we don’t want it returning zero. ☺

Maybe, our first idea could be waiting for the second break on CreateFileW, then entering the signature verification routine and directly *patching* the *VerifySignature* outcome, something like the following:

```

5904D2A1 . 8B8D F0F6FFFI MOV ECX, [LOCAL:580]
5904D2A7 . 6A 00          PUSH 0
5904D2A9 . 6A 00          PUSH 0
5904D2AB . 50            PUSH EAX
5904D2AC . 56            PUSH ESI
5904D2AD . 57            PUSH EDI
5904D2AE . 51            PUSH ECX
5904D2AF . FF15 70983555 CALL DWORD PTR DS:[59359870]
5904D2B5 . 33C0         XOR EAX, EAX
5904D2B7 . 40            INC EAX
5904D2B8 . 90            NOP
5904D2C0 . > C685 FB6FFFI MOV BYTE PTR SS:[EBP-905], 1
5904D2C0 . 57            PUSH EDI
5904D2C1 . E8 01EF2B00 CALL DxtoryCo.5930C1C7
5904D2C6 . 83C4 04      ADD ESP, 4
  
```

and, at a first glance, if we press F9 all would seem OK. Nevertheless if we persist the changes and reopen the patched DLL with Olly we soon realize that DLL patching too won’t be so easy or quick.



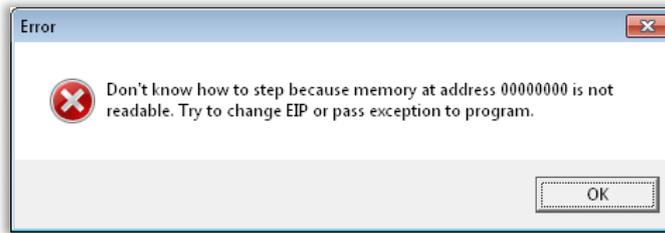
Breakpoint again CreateFileW and CTRL+F2. If we return to DLL code from the first break on CreateFileW, we have to trace just a few instructions to see that a number of routines “fail” (the first one is the call above the red arrow)

```

NormalMode - DxtoryCorePatched.dll - [*G.P.U* - main thread, module aa]
File View Debug Plugins Options Window Help
Paused
5904D51E . 6A 00          PUSH 0
5904D520 . 6A 03          PUSH 3
5904D522 . 68 00000080   PUSH 80000000
5904D527 . 05 E3C30000   ADD EAX, 0C3E3
5904D52C . 32DB         XOR BL, BL
5904D52E . 50            PUSH EAX
5904D52F . 8B5D E7       MOV BYTE PTR SS:[EBP-19], BL
5904D532 . FF15 F9983559 CALL DWORD PTR DS:[593598F8]
5904D538 . 8BF8         MOV EDI, EAX
5904D53A . 897D E0       MOV [LOCAL:83], EDI
5904D53D . 83FF FF       CMP EDI, -1
5904D540 . 0F84 40010000 JE aa.5904D686
5904D546 . 56            PUSH ESI
5904D547 . 6A 00          PUSH 0
5904D549 . 6A 00          PUSH 0
5904D54B . 6A 00          PUSH 0
5904D54D . 6A 02          PUSH 2
5904D54F . 6A 00          PUSH 0
5904D551 . 57            PUSH EDI
5904D552 . FF15 F4983559 CALL DWORD PTR DS:[593598F4]
5904D558 . 8BF0         MOV ESI, EAX
5904D55A . 8975 DC       MOV [LOCAL:93], ESI
5904D55D . 85F6         TEST ESI, ESI
5904D55F . 0F84 19010000 JE aa.5904D67E
5904D565 . 6A 00          PUSH 0
5904D567 . 6A 00          PUSH 0
5904D569 . 6A 00          PUSH 0
5904D56B . 6A 04          PUSH 4
5904D56D . 56            PUSH ESI
5904D56E . FF15 78993559 CALL DWORD PTR DS:[59359978]
5904D574 . 8BD8         MOV EBX, EAX
5904D576 . 895D D8       MOV [LOCAL:103], EBX
5904D579 . 850B         TEST EBX, EBX
5904D57B . 0F84 F3000000 JE aa.5904D674
5904D581 . 53            PUSH EBX
5904D582 . FF15 A09C3559 CALL DWORD PTR DS:[59359CA0]
5904D588 . 83C4 04      ADD ESP, 4
5904D58A . 3400         TEST AL, AL
5904D58D . 0F84 DA000000 JE aa.5904D66D
5904D593 . E8 18E2FEFF CALL aa.5903B7B0
5904D598 . 84C0         TEST AL, AL
5904D59A . 0F84 C0000000 JE aa.5904D66D
5904D5A0 . E8 FB08FEFF CALL aa.5903AE00
5904D5A5 . 84C0         TEST AL, AL
5904D5A7 . 0F84 C0000000 JE aa.5904D66D
5904D5AD . B5 12000000   MOV EAX, 12
5904D5B2 . 8D75 E8       LEA ESI, [LOCAL:63]
5904D5B5 . B9 6C2F3359 MOV ECX, aa.59332F6C
5904D5B8 . 66000000     MOV EAX, 0
5904D5BF . 6A 00          PUSH 0
5904D5C1 . 57            PUSH EDI
5904D5C2 . FF15 30993559 CALL DWORD PTR DS:[59359930]
5904D5C8 . 51            PUSH ECX
5904D5CA . 83FF 04      CMP EDI, 4
5904D5CD . 0F82 94000000 JE aa.5904D667
5904D5D0 . 8B41 F0      MOV EAX, DWORD PTR DS:[EDI+FB*4]
  
```

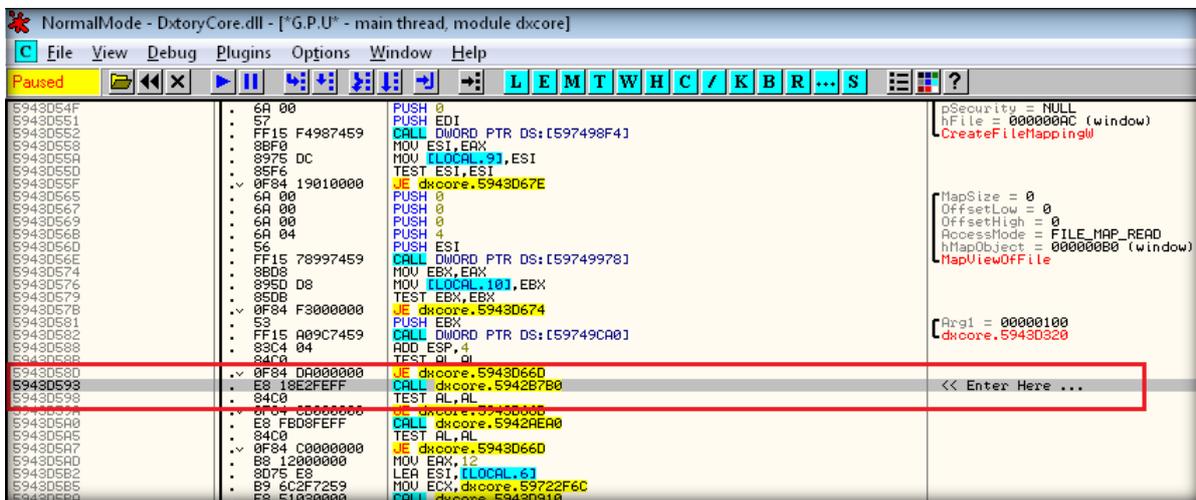
Even if we invert the Zero flag and continue stepping we would see the next call failing as well and also the one which follows it. It’s interesting to note that the *GetFileSize* call, we saw so clearly before - backtracing from the second break on *CreateFileW* - now is not even visible. Moreover if we alter the code flow, so we

can reach it through stepping, we'll see that it won't even be resolved (keeps its 00000000 value), making Olly show the familiar message (reported below):

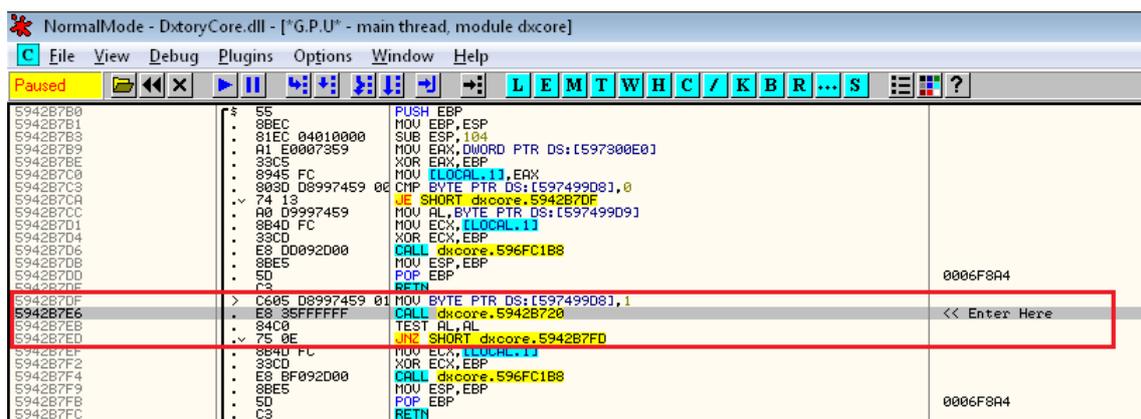


It's worth noting also that other APIs like CreateFileW/CreateFileMappingW, stored in target IAT, are instead available since from the beginning.

If we trace again the *original* DLL we quickly realize that its integrity is a necessary condition for correct import resolution. To be completely sure of that, it suffices digging a bit more into the code.



Following the highlighted call we'll find ourselves here:

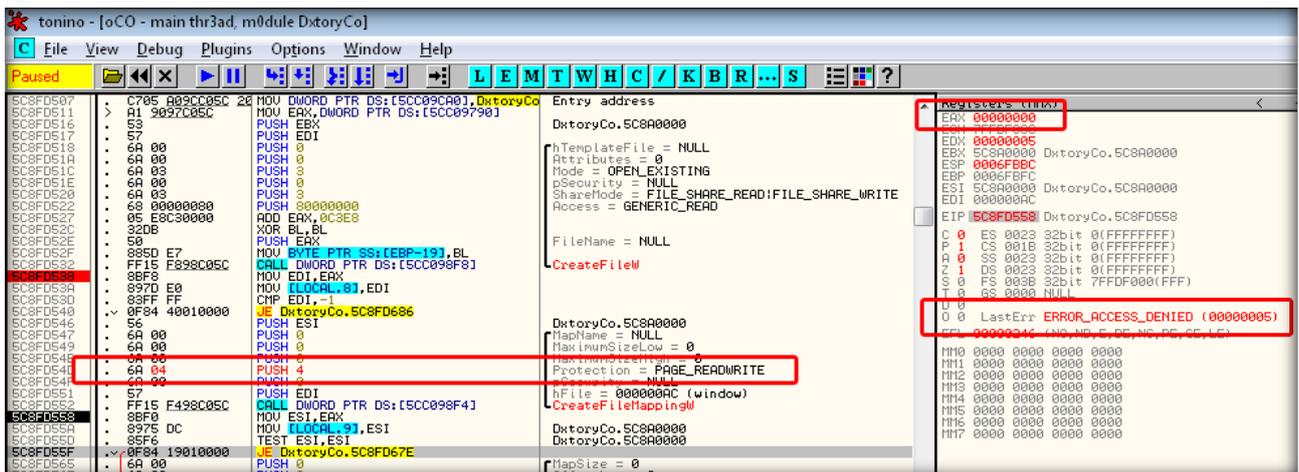


A TEST AL, AL instruction is always intriguing, right? :P Let's enter the call so we are here:



Ah, ha! The DLL names of a few system libraries (and API functions, as we would have seen if we traced a bit more) are “decrypted” through some calculations on the executable derived memory-map ... that’s why everything fucked up when we altered the code :D

Alright, this isn’t the first time we encounter such a situation. Recall, for example, JohnWho papers about inline patching AsProtect and make the memory map writable so we can hide our patches before the real check takes place. That way, however, is not feasible: if we change to READWRITE the file mapping type we will soon be greeted by a cold ERROR_ACCESS_DENIED ☹



I really don’t know why the call will fail ... probably this is due to our being in DLL initialization phase – DLL_PROCESS_ATTACH – where some kind of writing lock is in place. Though this outcome catches us, in no way it shall make us desist from our aim. That’s a valid idea! We just need to think better on how to put it into practice ;)

We need to find a way so the code “can see” an untouched view of the file. We just saw we can’t directly alter the view ... but why not just copying this view in a place we have full control on, conveniently hide our modifications and “pass” this new “view” of the file to the checking routines? ☺ That should work indeed!

Right, we need a memory area big enough to copy the whole content of our DLL file: we can ask for that the operating system through a call to VirtualAlloc! But, wait! VirtualAlloc import is one of those that aren’t yet resolved at DLL initialization and, even if we would think of looking for its address, the absence of GetProcAddress prevents us from retrieving it in an easy way. LoadLibraryW would make it possible to recover GetProcAddress and/or VirtualAlloc addresses navigating into kernel32.dll export table but, before getting involved in something I never tried, I prefer to look better around.

Stop at the first CreateFileW (or put an EB FE – an infinite loop – at the DLL entry point) and reload the DLL in Olly; once inside the code, search for all intermodular calls:

```

SEAR1937 CALL DWORD PTR DS:[&KERNEL32.GetGener kernel32.GetGener
SEAR193B CALL DWORD PTR DS:[&KERNEL32.GetStartu kernel32.GetStartupInfol
SEAR1943 CALL DWORD PTR DS:[&KERNEL32.GetStdHan kernel32.GetStdHandle
SEAR1948 CALL DWORD PTR DS:[&KERNEL32.GetStdHan kernel32.GetStdHandle
SEAR1953 CALL DWORD PTR DS:[&KERNEL32.GetString kernel32.GetStringTypeW
SEAR1958 CALL DWORD PTR DS:[&KERNEL32.GetString kernel32.GetStringTypeW
SEAR1963 CALL DWORD PTR DS:[&KERNEL32.GetSystem kernel32.GetSystemTimeAsFileTime
SEAR1968 CALL DWORD PTR DS:[&KERNEL32.GetTime kernel32.GetTickCount
SEAR1973 CALL DWORD PTR DS:[&KERNEL32.HeapCreat kernel32.HeapCreate
SEAR1978 CALL DWORD PTR DS:[&KERNEL32.HeapDestro kernel32.HeapDestroy
SEAR1983 CALL DWORD PTR DS:[&KERNEL32.HeapFree kernel32.HeapFree
SEAR1988 CALL DWORD PTR DS:[&KERNEL32.Initialize kernel32.InitializeCriticalSectionAndSpinCount
SEAR1993 CALL DWORD PTR DS:[&KERNEL32.Initialize kernel32.InitializeCriticalSectionAndSpinCount
SEAR1998 CALL DWORD PTR DS:[&KERNEL32.Initialize kernel32.InitializeCriticalSectionAndSpinCount
SEAR2003 CALL DWORD PTR DS:[&KERNEL32.Initialize kernel32.InitializeCriticalSectionAndSpinCount

```

While scrolling, the highlighted allocation functions catch our attention; their purpose is to allocate memory too so they can therefore be useful for our job. But, here too, wait a bit! Where's *HeapAlloc*? Just for about a second I thought it was one of those dynamically resolved imports too but, instead, scrolling a bit below I found it¹¹ exported by ntdll.

```

SEAR1A6B CALL DWORD PTR DS:[&KERNEL32.QueryPerf kernel32.QueryPerformanceCounter
SEAR1A73 CALL DWORD PTR DS:[&KERNEL32.RaiseExcep kernel32.RaiseException
SEAR1A78 CALL DWORD PTR DS:[&KERNEL32.RaiseExcep kernel32.RaiseException
SEAR1A83 CALL DWORD PTR DS:[&KERNEL32.HeapAllo ntdll.RtlAllocateHeap
SEAR1A88 CALL DWORD PTR DS:[&KERNEL32.HeapAllo ntdll.RtlAllocateHeap
SEAR1A93 CALL DWORD PTR DS:[&KERNEL32.DecodePoi ntdll.RtlDecodePointer
SEAR1A98 CALL ESI ntdll.RtlDecodePointer
SEAR1AA3 CALL DWORD PTR DS:[&KERNEL32.DecodePoi ntdll.RtlDecodePointer
SEAR1AA8 CALL DWORD PTR DS:[&KERNEL32.DecodePoi ntdll.RtlDecodePointer

```

Right, now we have all the basic building blocks: we are ready to start!

Let's start dancing

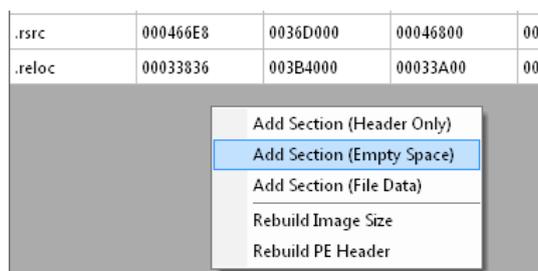
Now we have all the needed information collected, we'll proceed as we planned.

We will:

- Intercept the creation of the read-only file map in memory;
- Create a memory area (a heap to be precise) we have full control on;
- Save the address of the original memory map so we can, before the end¹², "return it back" to original code so the latter can free up the memory allotted (i.e. *UnMapViewOfFile, CloseHandle*);
- Copy the content of this view into our new heap hiding both code flow redirections and real patches;
- Pass our "cleaned copy" to the control routines so the file will be considered valid and untouched.

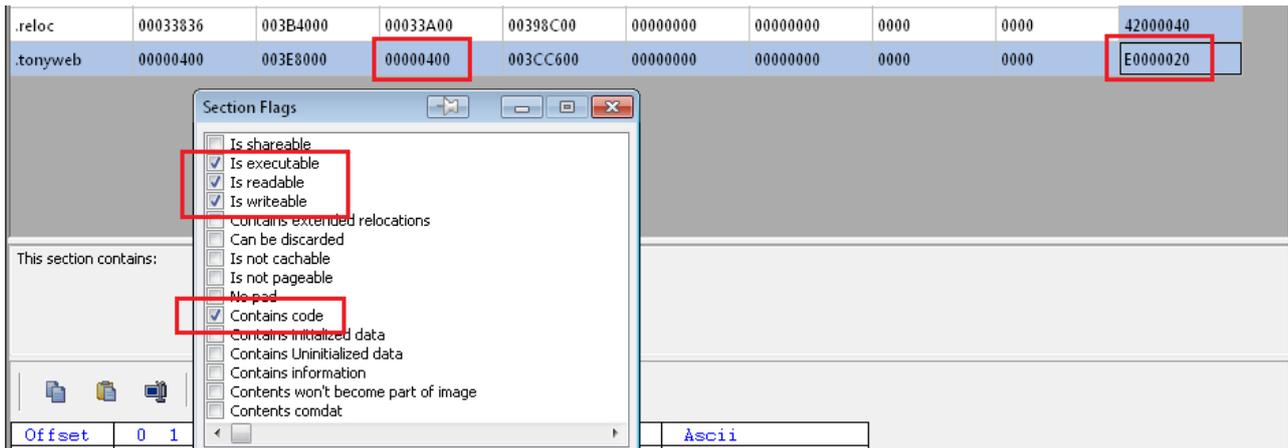
First of all let's add some space for our code-injection. Usually, when there's the need to inject some simple code, I'll use already available code-caves but, when the code to write is complex and/or needs some extensive debugging (like the one we are going to write) I'll prefer adding a completely new PE section.

As always fire up our CFF Explorer and perform the operation (choose a size of 400h, it should be big enough for our purposes):

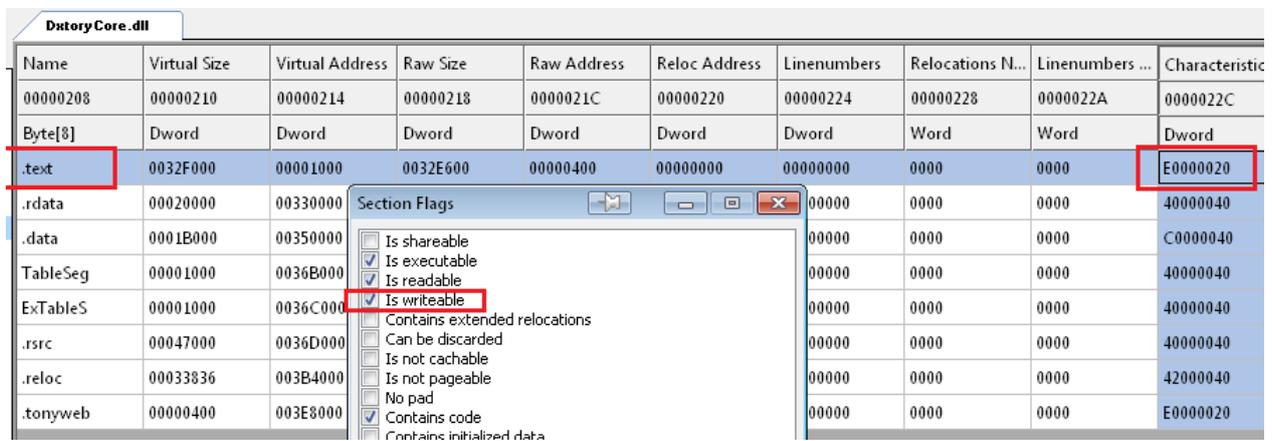


¹¹ I'm on Windows 7 (x86)

¹² We'll put a redirection also before *UnMapViewOfFile* call. There, we'll deal with heap destruction before restoring the original code flow.



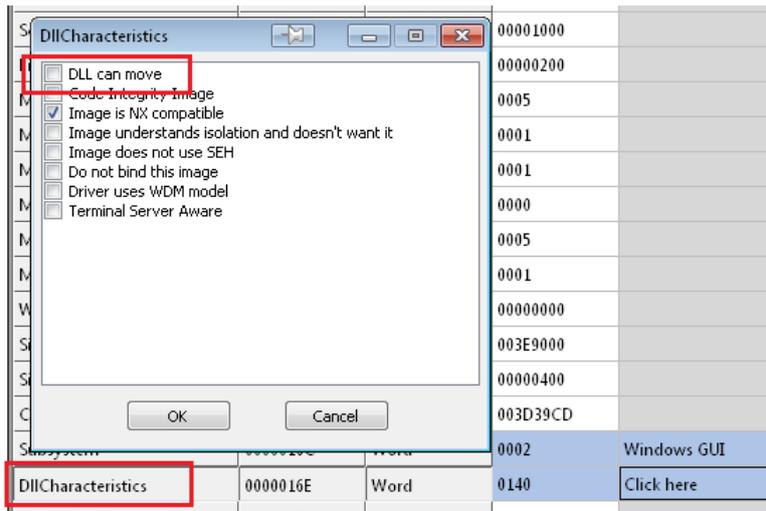
Since we'll have to alter the original code too, let's take advantage of this file modification step and make the .text section writable:



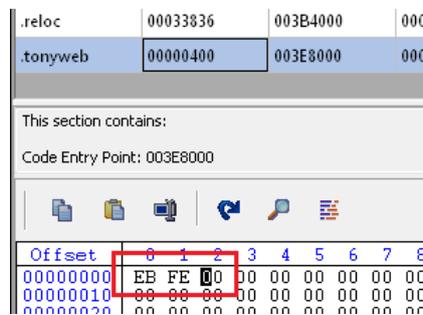
Change also the entry-point to the new section's start address:



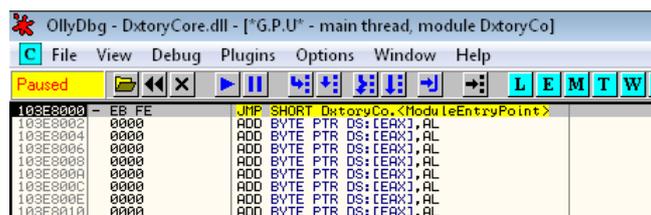
Moreover, in order to avoid some address handling annoyances during code-injection phases (the DLL can be loaded at different addresses among reloads) remove – only for now – the DLL Can Move flag ...



Lastly, for convenience, assemble an infinite loop at the beginning of the new *.tonyweb* section so we can break (clicking on the pause command ) directly on our code.



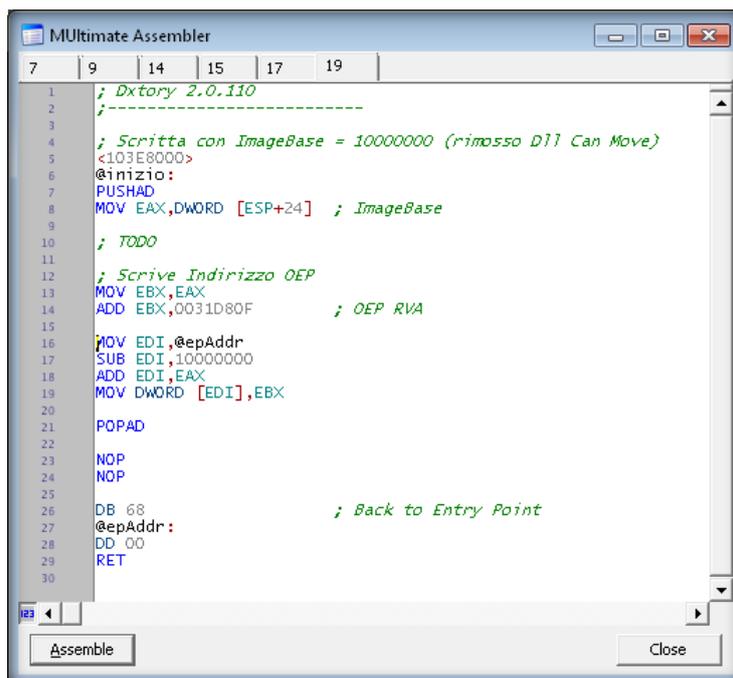
Pheew! Ok now we are ready. Save the DLL and load it in Olly. Once *Running* is written in Olly status window, click on the pause button, choose OK at the warning about entry-point location (outside the *.text* section), et voila! Here's our new section:



At this point start up the useful plugin called *Multimate Assembler* and begin writing the needed injection.

Code Injection: ready, camera ... action!

Start writing the code from the beginning of the section (on my system at address 0x103E8000), place the classic PUSHAD / POPAD pair and prepare the jump towards the "original" entry point.



If we now click on “Assemble” button we can trace the assembled code and actually reach the EP. Note that, since we’re working with a DLL, we need to account for code relocation so we can’t use absolute addresses: instead we have to use RVA. Though using labels (like ‘@epAddr’) is fairly useful they are always referred as absolute addresses by the plugin: therefore we need to subtract the imagebase at the time of writing (sub <r32>,1000000) and then add the run-time one. Since the latter is a particularly useful information and we’ll often need to refer to it, we’ll save it in a specific location in our injection.

Our main goal consists in making the DLL run despite the added section and the other modifications we applied. According to what we stated above we have to create an *heap* sized after the DLL file: so collect the needed information, particularly the import addresses (RVA) and the library disk size.

Start with imports, looking for their locations in IAT:

103300E8	75E0450E	8Edu	kernel32.WideCharToMultiByte
103300EC	75E010BC	70du	kernel32.GetEnvironmentStrings
103300F0	75E03E82	60du	kernel32.HeapCreate
103300F4	75DF2301	60du	kernel32.HeapDestroy
103300F8	75DF000F	60du	kernel32.HeapFree
103300FC	75DFBA60	11du	kernel32.GetTickCount
10330100	75DF0004	60du	kernel32.HeapReAlloc

The first two we get are the following:

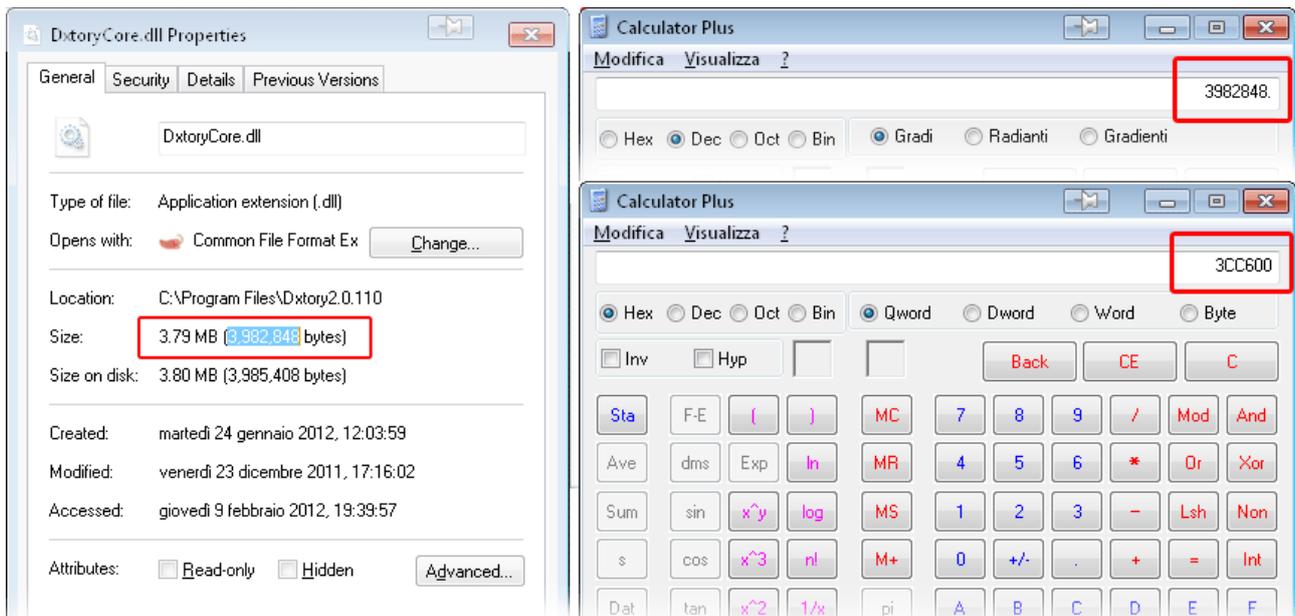
HeapCreate,	RVA: 3300F0
HeapDestroy,	RVA: 3300F4

Then look for *HeapFree* so we find the other ones:

10330070	75DF2381	10du	kernel32.TerminateProcess
10330074	75DFC0CF	30du	kernel32.GetCurrentProcess
10330078	75E0ED38	87du	kernel32.UnhandledExceptionFilter
1033007C	75E03081	80du	kernel32.SetUnhandledExceptionFilter
10330080	75DF3E88	20du	kernel32.IsDebuggerPresent
10330084	75DF8D08	31du	kernel32.HeapFree
10330088	77222006	10du	ntdll.RtlAllocateHeap
1033008C	75E076B5	A0du	kernel32.IsProcessorFeaturePresent
10330090	75E01400	10du	kernel32.WriteFile
10330094	75E01E46	F4du	kernel32.GetStdHandle

HeapFree,	RVA: 330084
RtlAllocateHeap,	RVA: 330088

We only miss the file size, but that is a baby game: a couple of clicks plus the Windows calculator are more than enough:



With these information we are ready to write the first part of our injection that deals with heap creation and allocation:

```

MULTIMATE ASSEMBLER
7 | 9 | 14 | 15 | 17 | 19 |
5 | <103E8000>
6 | @inizio:
7 | PUSHAD
8 | MOV EAX,DWORD [ESP+24] ; ImageBase
9 |
10 | ; Salva ImageBase corrente
11 | MOV EBX,@baseAddr
12 | SUB EBX,10000000
13 | ADD EBX,EAX
14 | MOV DWORD [EBX],EAX
15 |
16 | ; Recupera HeapCreate (VirtualAlloc non esiste ancora ...
17 | MOV EDI,EAX
18 | ADD EDI,3300F0
19 |
20 | MOV ESI,EAX ; SALVA ImageBase
21 | PUSH EAX
22 |
23 | ; Chiama HeapCreate
24 | PUSH 0 ; maxSize
25 | PUSH 003CC600 ; initialSize
26 | PUSH 0 ; Indirizzo scelto da lui
27 | CALL DWORD [EDI]
28 |
29 | ; Salva heap allocato
30 | MOV EBX,@heapCreateAddr
31 | SUB EBX,10000000
32 | ADD EBX,ESI
33 | MOV DWORD [EBX],EAX
34 |
35 | @call_HeapAlloc:
36 | POP EDX ; ImageBase
37 | PUSH EDX
38 |
39 | MOV EDI,EDX
40 | ADD EDI,0330088 ; offset di HeapAlloc
41 | PUSH 003CC600 ; Size
42 | PUSH 8 ; Zero-Memory
43 | PUSH EAX ; Heap Obj
44 | CALL DWORD [EDI] ; Heap Alloc
45 |
46 | POP EDX ; ImageBase
47 |
48 | ; Salva risultato heapAlloc
49 | MOV EBX,@heapAddr
50 | SUB EBX,10000000
51 | ADD EBX,EDX
52 | MOV DWORD [EBX],EAX
53 |
54 | MOV EAX,ESI ; Ri-recupera ImageBase
55 |
56 | ; Scrive Indirizzo OEP
57 | MOV EBX,EAX
58 | ADD EBX,0031D80F ; OEP RVA
59 |
60 | MOV EDI,@epAddr
61 | SUB EDI,10000000
62 | ADD EDI,EAX
63 | MOV DWORD [EDI],EBX
64 |
65 | POPAD
66 |
67 | DB 68 ; Back to Entry Point

```

Of course we reserve some space (after the jump to EP) to save a number of addresses we'll need throughout our code:

```

56 | MOV EBX,EAX
57 | ADD EBX,0031D80F ; OEP RVA
58 |
59 | MOV EDI,@epAddr
60 | SUB EDI,10000000
61 | ADD EDI,EAX
62 | MOV DWORD [EDI],EBX
63 |
64 | POPAD
65 |
66 | DB 68 ; Back to Entry Point
67 | @epAddr:
68 | DD 00
69 | RET
70 |
71 | @baseAddr:
72 | DD 00
73 |
74 | @heapCreateAddr:
75 | DD 00
76 |
77 | @mapViewAddr:
78 | DD 00
79 |
80 | @heapAddr:
81 | DD 00

```



```

MULTIMATE ASSEMBLER
7 | 9 | 14 | 15 | 17 | 19 |
@heapAddr:
DD 00
;===== CAVE 1: MapViewOfFile =====
@cave1:
PUSHAD
MOV ESI,EAX ; Indirizzo Buffer MapViewOfFile

; Recupera ImageBase
DB 68
@baseAddr_cave1:
DD 00
POP EAX

; Salva indirizzo reale della MapViewOfFile per poi ripristinarlo
MOV EBX,@mapViewAddr
SUB EBX,10000000
ADD EBX,EAX
MOV DWORD [EBX],ESI

; Indirizzo dell'HEAP da sostituire
MOV EBX,@heapAddr
SUB EBX,10000000
ADD EBX,EAX
MOV EBX,DWORD [EBX]

MOV EDI,@heapAddr_cave1
SUB EDI,10000000
ADD EDI,EAX
MOV DWORD [EDI],EBX ; completa PUSH+POP

;
PUSH EBX ; Salva indirizzo da patchare

; === Esegue la copia ===
MOV ECX,0F3180 ;003CC600 / 4 <<<<<<
MOV EDI,EBX
REP MOVSB

; === TODO: Camuffare le modifiche ===

POP ESI ; Recupera buffer da patchare

@ritorno_ad_originale:
; Prepara ritorno al flusso originale
MOV EBX,05D579 ; dest
ADD EBX,EAX

MOV EDI,@jmpBackFromCave1
SUB EDI,10000000
ADD EDI,EAX
MOV DWORD [EDI],EBX ; completa PUSH+RET

POPAD

DB 68
@heapAddr_cave1:
DD 00
POP EAX

MOV EBX,EAX ; orig
MOV DWORD PTR SS:[EBP-28],EBX ; orig

NOP
NOP
NOP
NOP

DB 68
@jmpBackFromCave1:
DD 00
RET

```

Here we're making a copy of MapViewOfFile returned buffer into our created heap whose address we save into @heapAddr_cave1. Before executing the original code (i.e. the instructions we replaced with the jump to this cave and now marked with 'orig' remark) and return to the original code-flow, we make sure heap address is stored into EAX register.

With that done, start hiding our patches. First, let's find out those we absolutely need to cover: the changes we did to the physical file. From command prompt do a comparison (at bytes' level) between original and modified DLL, thanks to the fc command.

```

C:\Program Files\Dxtory2.0.110>fc /b "DxtoryCore - Orig.dll" DxtoryCore.dll

00000116: 07 08          ; Num of Sections
00000138: 0F 00          ; EP
00000139: D8 80
0000013A: 31 3E

00000161: 80 90          ; Size Of Image
0000016E: 40 00          ; Dll Can Move
0000022F: 60 E0          ; .text section writable

00000320: 00 2E          ; '.'
00000321: 00 74          ; 't'
00000322: 00 6F          ; 'o'
00000323: 00 6E          ; 'n'
00000324: 00 79          ; 'y'
00000325: 00 77          ; 'w'
00000326: 00 65          ; 'e'
00000327: 00 62          ; 'b'
00000329: 00 04          ; VSize
0000032D: 00 80          ; Vaddr
0000032E: 00 3E
00000331: 00 04          ; RSize
00000335: 00 C6          ; RAddr
00000336: 00 3C
00000344: 00 20          ; Flags
00000347: 00 E0
FC: DXTORYCORE.DLL longer than DxtoryCore - Orig.dll

```

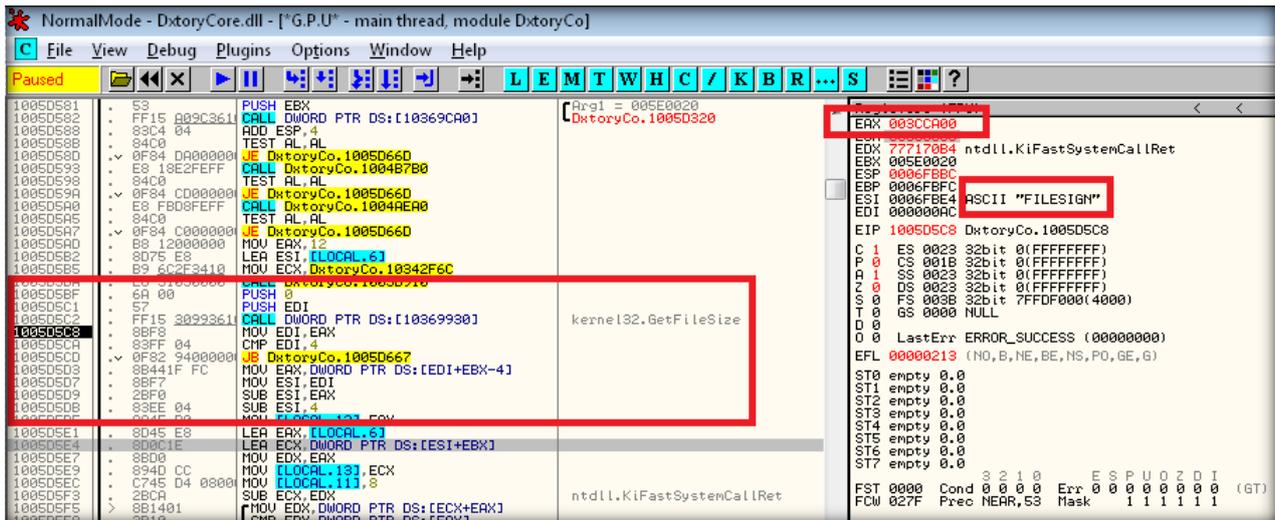
Then, add to the cave the code to hide these changes (note we use file offsets here):

```

MULTIMATE ASSEMBLER
7 | 9 | 14 | 15 | 17 | 19 |
; === TODO: Camuffare le modifiche ===
POP ESI ; Recupera buffer da patchare
PUSH EAX ; ImageBase
@Hide_Changes_In_Heap:
; Num Of Sections
MOV EAX,ESI
ADD EAX,116
MOV BYTE [EAX],7
; Size Of Image
MOV EAX,ESI
ADD EAX,161
MOV BYTE [EAX],80
; Nuova sezione
MOV EDI,ESI
ADD EDI,00000320
MOV ECX,30h
XOR EAX,EAX
REP STOS BYTE [EDI]
; Sezione .text writable
MOV EAX,ESI
ADD EAX,0000022F
MOV BYTE [EAX],60
; EntryPoint
MOV EAX,ESI
ADD EAX,00000138
MOV BYTE [EAX],0F
INC EAX
MOV BYTE [EAX],D8
INC EAX
MOV BYTE [EAX],31
;
POP EAX ; Recupera ImageBase
@ritorno_ad_originale:
; Prepara ritorno al flusso originale
MOV EBX,00000000

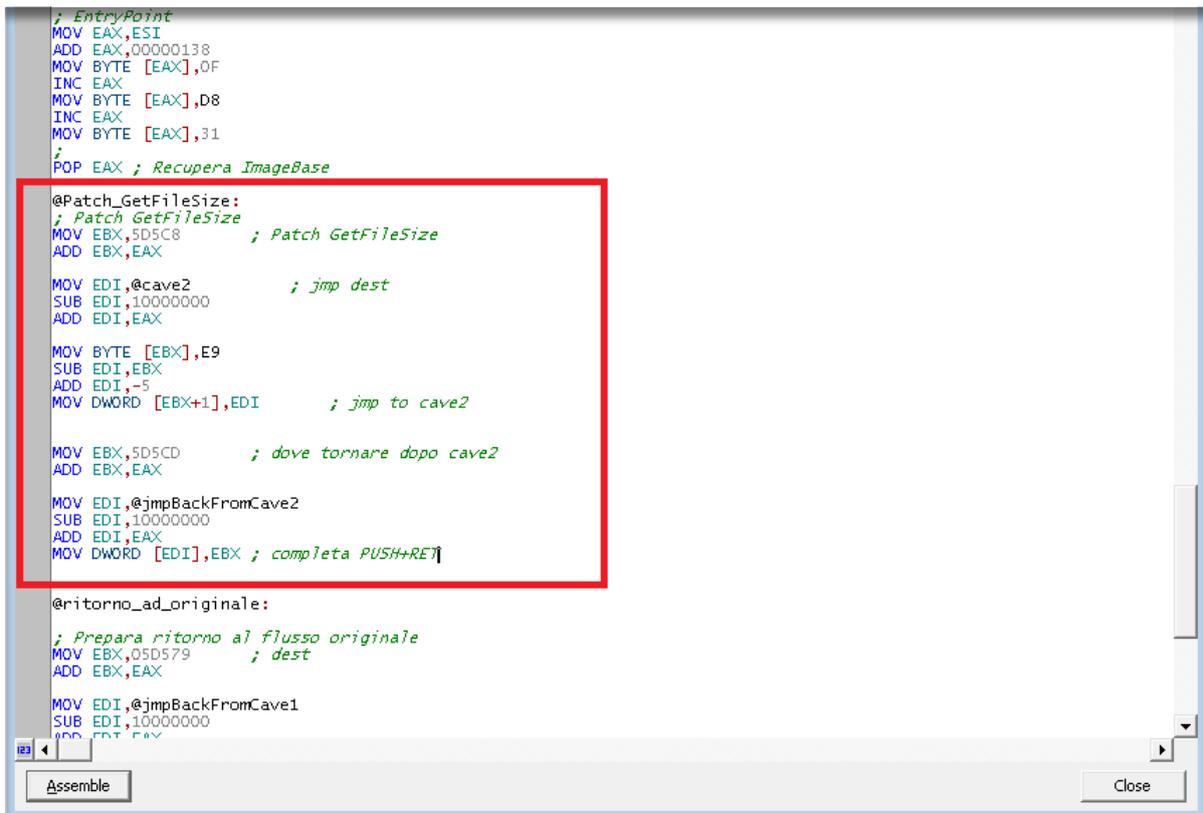
```

Even with that hiding in place we're not yet ready to run our DLL. It's clear we did a good job: to demonstrate this we can see imports get correctly resolved (in the example the GetFileSize is clearly visible):



Nevertheless, as highlighted in the picture above, we need to adjust the `GetFileSize` return value so it will not account for our new `.tonyweb` section (this is an important step because following routines would use that value to know when to stop reading the buffer; since we replaced the original buffer with a fixed-length heap – sized after the original DLL – we should avoid the program reading beyond heap dimensions).

So we put a new cave, and therefore a new jump to it, where we subtract the new section raw size (0x400) from `EAX` before that value gets passed to any other routine.



This time our second cave (cave2, we place it after cave1 in the injection) is way easier than then the first one given that it simply needs to avoid the “wrong” size be passed along:

```

DB 68
@jmpBackFromCave1:
DD 00
RET

;===== CAVE 2 : GetFileSize

@cave2:
SUB EAX,400 ; Sottrae la dimensione della mia sezione .tonyweb
MOV EDI,EAX ; orig.
CMP EDI,4 ; orig.
DB 68
@jmpBackFromCave2:
DD 00
RET

NOP
NOP

```

Later, the DLL file gets reopened (do you remember the second CreateFileW?) in order to perform the crypto checks. This time, luckily for us, we need not to “sanitize” the read file content: it’s enough to patch the *VerifySignature* outcome. We choose to do that this way:

So add the following piece of code to our cave1:

```

MOV EDI,@jmpBackFromCave2
SUB EDI,10000000
ADD EDI,EAX
MOV DWORD [EDI],EBX ; completa PUSH+RET

@Patch_CryptVerifySignature:
; Patch CryptVerifySignature
; RVA: 5D245 ==> 80 01 MOV AL,1
; RVA: 5D248 ==> 07 -> 00
MOV EDI,5D2B5 ; Patch CryptVerifySignature
ADD EDI,EAX
MOV WORD [EDI],01B0 ; TEST EAX,EAX ==> MOV AL,1
ADD EDI,3 ; JE <07> ==> JE <00>
MOV BYTE [EDI],00

@ritorno_ad_originale:
; Prepara ritorno al flusso originale
MOV EBX,05D579 ; dest
ADD EBX,EAX

```

In the excitement don’t forget to get rid of the heap: insert another cave (the last one, cave3) where to jump after integrity checks get performed, but before resources get released. There, destroy the heap we created and prepare the return to the original flow, where the program can clean up the file map itself:

The red arrow shows the location where we'll put the jump to cave3; modify cave1 code to insert this new redirection:

```
@Patch_CryptVerifySignature:
; Patch CryptVerifySignature
; RVA: 5D245 ==> 80 01          MOV AL,1
; RVA: 5D248 ==> 07 -> 00
MOV EDI,5D2B5                ; Patch CryptVerifySignature
ADD EDI,EAX
MOV WORD [EDI],01B0         ; TEST EAX,EAX ==> MOV AL,1
ADD EDI,3
MOV BYTE [EDI],00          ; JE <07> ==> JE <00>

@Patch_UnmapView:
MOV EDI,005D667             ; src
ADD EDI,EAX

MOV EBX,@cave3             ; dest
SUB EBX,10000000
SUB EBX,EDI
ADD EBX,-5
ADD EBX,EAX

MOV BYTE [EDI],E9
MOV DWORD [EDI+1],EBX
MOV BYTE [EDI+5],90

@ritorno_ad_originale:
; Prepara ritorno al flusso originale
MOV EBX,05D579             ; dest
ADD EBX,EAX
MOV EDI,@impBackFromCave1

Assemble
```

then write the code for cave3, code we insert at the end of our injection:

```

MULTIMATE ASSEMBLER
7 | 9 | 14 | 15 | 17 | 19 |
;----- CAVE 3: Destroy
@Cave3:
PUSH EAX
PUSH EBX
CALL @delta3
@delta3:
POP EAX
; Calculate ImageBase in a cumbersome way
MOV EDI,EAX
MOV EBX,@delta3
SUB EBX,@inizio
SUB EDI,3E8000 ; Inizio sezione .tonyweb
SUB EDI,EBX ; ImageBase
; dest
MOV ESI,005D66D ; where to return after cave
ADD ESI,EDI
; completa PUSH+RET
MOV EBX,@jmpBackFromCave3
SUB EBX,@delta3
ADD EBX,EAX
MOV DWORD [EBX],ESI
@call_HeapDestroy:
; Chiamare HeapDestroy
MOV EBX,3300F4 ; HeapDestroy
ADD EBX,EDI
; get heap address to destroy
MOV ESI,@heapCreateAddr
SUB ESI,@delta3
ADD ESI,EAX
PUSH EAX ; Salva offset
PUSH DWORD [ESI]
CALL DWORD [EBX] ; HeapDestroy
@restore_MapViewOfFile:
POP EAX ; Recupera OFFSET
; Restore return address of MapViewOfFile ...
MOV ESI,@mapViewAddr
SUB ESI,@delta3
ADD ESI,EAX
;
POP EBX
POP EAX
MOV EBX, DWORD [ESI] ; Indirizzo originale MapViewOfFile
MOV ESI,DWORD PTR SS:[EBP-24] ; orig.
MOV EDI,DWORD PTR SS:[EBP-20] ; orig.
DB 68
@jmpBackFromCave3:
DD 00
RET
Assemble Close

```

Aside from the particularly ugly way to recover the ImageBase (I could have simply retrieved it from the dword I saved it into :P), the rest is pretty clear: we prepare the return address from the cave, destroy the created heap and place, where expected from the program (in EBX register), the address of the "original view" of the file.

Of course the more careful of you may ask: where is *HeapFree*? Why didn't we call it? Well, I read about heaps on MSDN and, probably wrongly, I ended up concluding that it's technically possible to omit it.¹³

Remarks

Processes can call *HeapDestroy* without first calling the *HeapFree* function to free memory allocated from the heap.

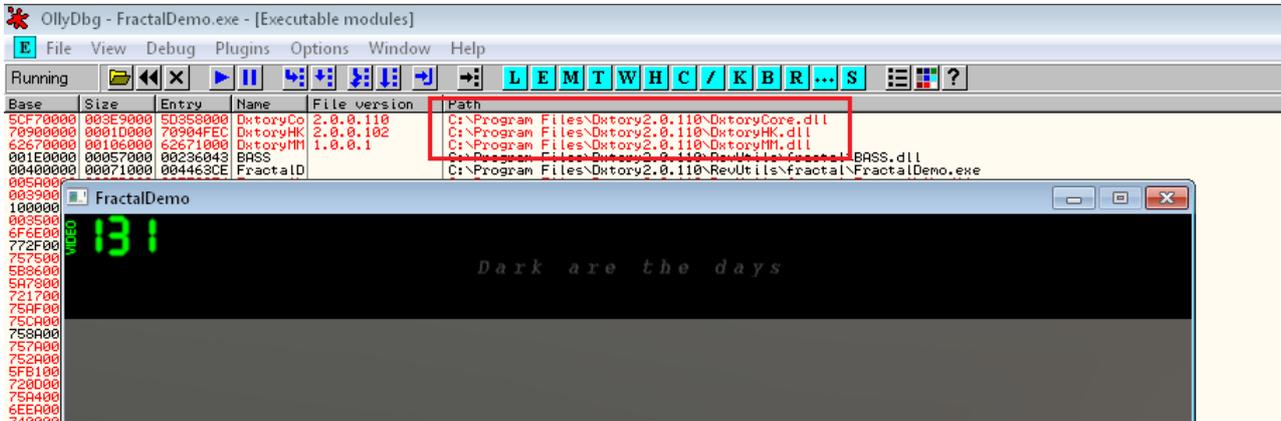
At this point our altered DLL should be considered perfectly uncorrupted by the program and should allow us to take screenshots and movie captures without any problem. Assemble the written code and save the changes, then remember to restore DLL Can Move flag and try to launch the program ... it works! Great! ☺

We're now ready to deal with the real issue: that annoying program-logo overlay.

¹³ (see <http://msdn.microsoft.com/en-us/library/windows/desktop/aa366700%28v=vs.85%29.aspx>).

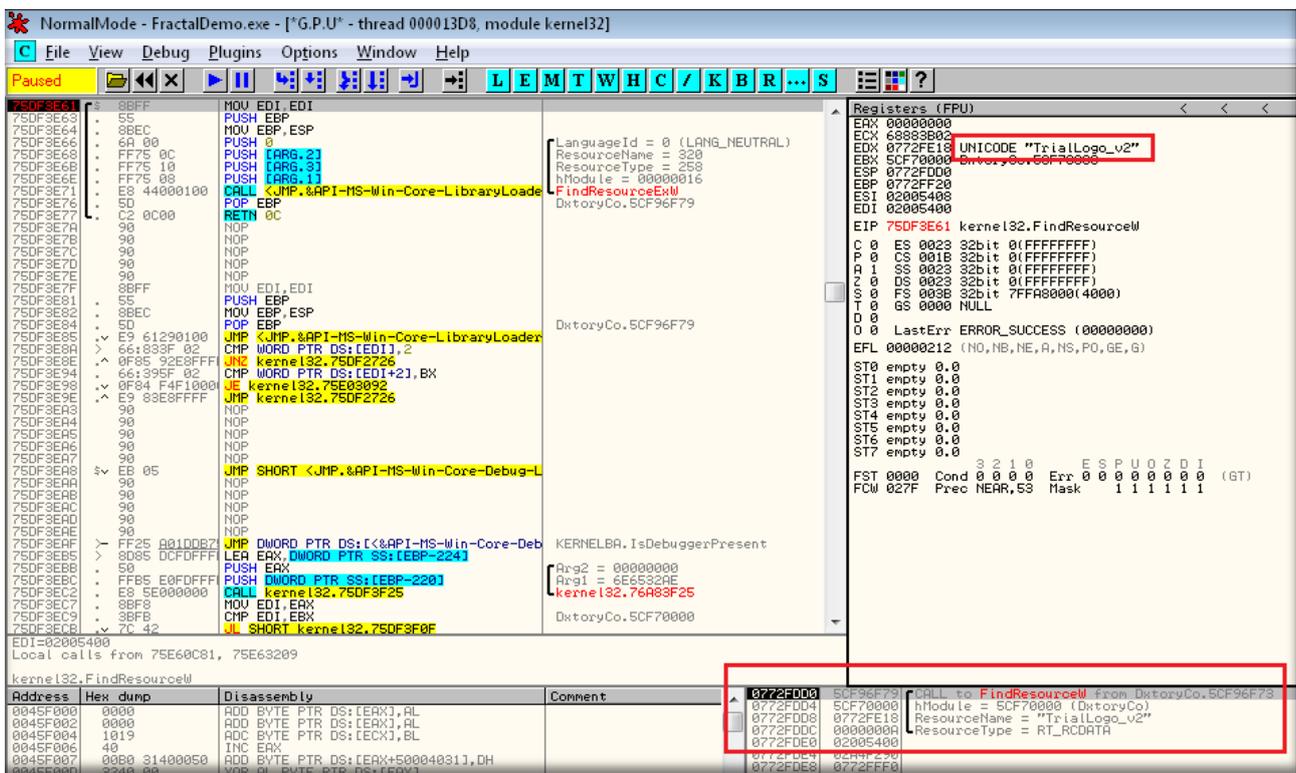
Removing the watermark

As DxtoryCore.dll is injected in every process the program can handle, we can easily debug our FractalDemo.exe and locate the exact point in time where the watermark resource gets retrieved. In fact, If we start FractalDemo.exe in Olly, we can observe how Dxtory DLLs get injected and we're able to see them among loaded modules:

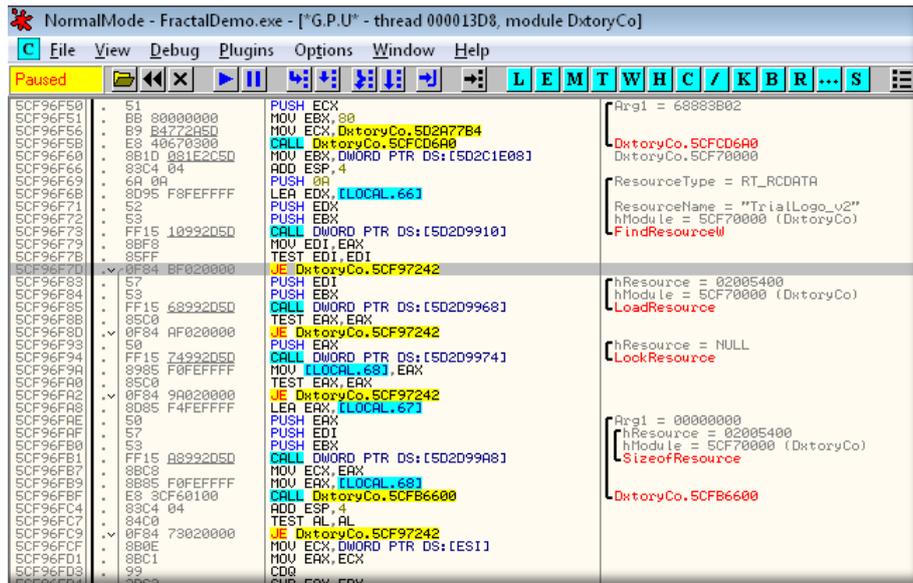


To find when and where the target loads the logo we run FractalDemo, put a BP on FindResourceW and start the video recording (pressing F12, by default).

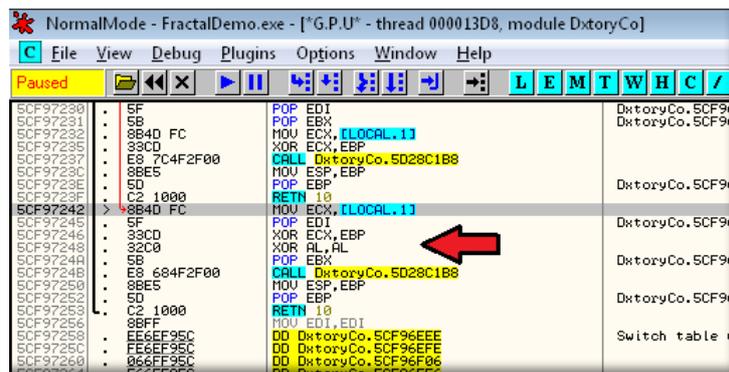
Once we press F12 on keyboard our breakpoint triggers:



Through the stack we can backtrace to the caller, reaching this piece of code:



just in the middle of the routine which deals with watermark loading and drawing. ☺ If that FindResourceW fails, the whole routine fails as well and the capture will not start:



We'll make the program think the resource is missing but we let the routine return 1 into EAX so the recording could take place normally. ☺

Thanks to the above pictures and the one below (where we get the current module ImageBase) we'll have no problem preparing the needed patches:

Address	Entry	Name	File version	Path
50358000	50358000	DxtoryCo	2.0.0.110	C:\Program Files\Dxtory2.0.110\DxtoryCore.dll
70904FEC	70904FEC	DxtoryHK	2.0.0.102	C:\Program Files\Dxtory2.0.110\DxtoryHK.dll
62671000	62671000	DxtoryMH	1.0.0.1	C:\Program Files\Dxtory2.0.110\DxtoryMH.dll
001E0000	00057000	BASE		C:\Program Files\Dxtory2.0.110\ReuUtils\FractalS...

Rva	5CF96F7B - 5CF70000 = 26F7B			5CF97248 - 5CF70000 = 27248		
Original	5CF96F7B	85FF	TEST EDI,EDI	5CF97248	. 32C0	XOR AL,AL
Change To	5CF96F7B	33FF	XOR EDI,EDI	5CF97248	B0 01	MOV AL,1

To complete our work and perform these changes remove the 'DLL Can Move' flag again, open the DLL in Olly and insert the following instructions into cave1:

```
@Patch_UnmapView:
MOV EDI,005D667      ; src
ADD EDI,EAX

MOV EBX,@cave3      ; dest
SUB EBX,10000000
SUB EBX,EDI
ADD EBX,-5
ADD EBX,EAX

MOV BYTE [EDI],E9
MOV DWORD [EDI+1],EBX
MOV BYTE [EDI+5],90

@patch_CheckRisorsa:
; RVA: 26F7B
MOV EDI,26F7B
ADD EDI,EAX
MOV BYTE [EDI],33

; RVA: 27248 ==> 80 01 MOV AL,1
MOV EDI,0027248
ADD EDI,EAX
MOV WORD [EDI],01B0

@ritorno_ad_originale:
; Prepara ritorno al flusso originale
MOV EBX,05D579      ; dest
ADD EBX,EAX
```

Assemble, save changes to file, restore 'DLL Can Move' flag and try capturing again:



Great! We did it ... we can finally enjoy this beautiful sunshine! ☺

As usually, after assuring all is working properly and there's nothing more to worry about, replace the two patched files with the original ones to restore initial program status.

Conclusions

In this tutorial we saw that, from time to time, .NET protections are not limited to managed scope but force us to play with native code too. I hope you managed to follow my wanderings (I added a lot of pictures just to simplify your understanding :D) and, above all, I hope someone somewhere had the opportunity to "learn" something from it since the writing of these pages took quite a bit of time. ☺

Greetings and Thanks

First of all I must say thank you to lena151: if it were not for her amazing tutorials I would never entered the world of reverse engineering. You're great Lena!

A warm greeting to the friends of UIC (quequero, sparpacillon, PnUic, Quake, phobos, tonymhz, ecc.), to Black@Storm team and forum guys (Kurapica, whoknows, revert, 0xd4d, romero, yck1509, kao, bball0002, CodeCracker, ecc., the .NET gurus), to ARTeam crew (Nacho_Dj, Shub Nigurrath, Ghandi, Nieylana, SunBeam, deroko, ...) and to all people I "encounter" everyday browsing tuts4you, eXeTools, Appznet, ecc. boards ... eventually I'll end up forgetting someone :D

Greetings to JeRRy (SnD) who was the first one who asked me a mini-tut for the previous revision of this target ... I'm a bit late, I know, but better late than never, right? :P Special thanks to sparpacillon who often wastes quite a bit of his free time chatting with me about reversing; he also accepted to read and make readable this tutorial's beta version ... thanks again mate!. A thank you also to Mr.eXoDia who made me enter the Armadillo Keygenning world: he shared with me what he knew and had discovered.

Thank you also to all the friends at the SnD Requester Board; wihtout a specific order: Baxter, quosego, willie, deepzero, Snake, Vepergen, apuromafo, MasterUploader, qpt, JohnWho, PeterPunk, Silence, DisArm (a real pity he decided to leave the board), ... even here I forgot someone for sure, don't hate me for that.

In conclusion, I thank all the persons I can, luckily for me, call friends: thank you all for your continued support and motivation that helps me advance, in small steps unfortunately, at/in our wonderful hobby. ☺

Finally, thanks to all of you who had the guts to reach the last page of this tutorial :P

*tonyweb,
February 2012*