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

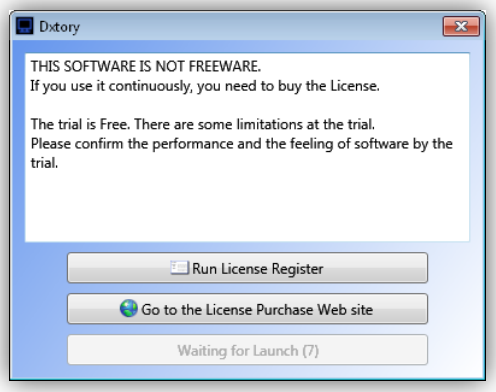
Below is a list of the tools we'll use:

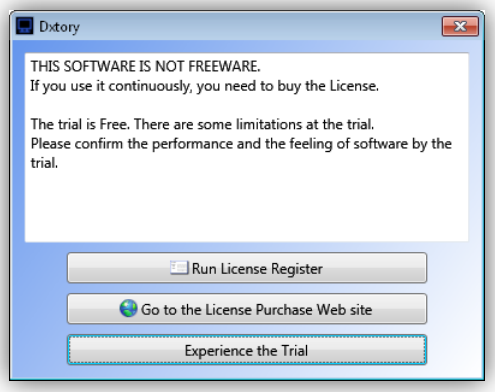
- RedGate Reflector and its Reflexil plugin (at v.1.3 at the time of writing)
- De4Dot (to deobfuscate our .NET executable, https://github.com/0xd4d/de4dot)
- CFF Explorer
- Your favorite hex editor (I'll use the free HxD editor)
- Mono.Cecil and Public Key Injector (should be attached to this tutorial)
- 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 ☺

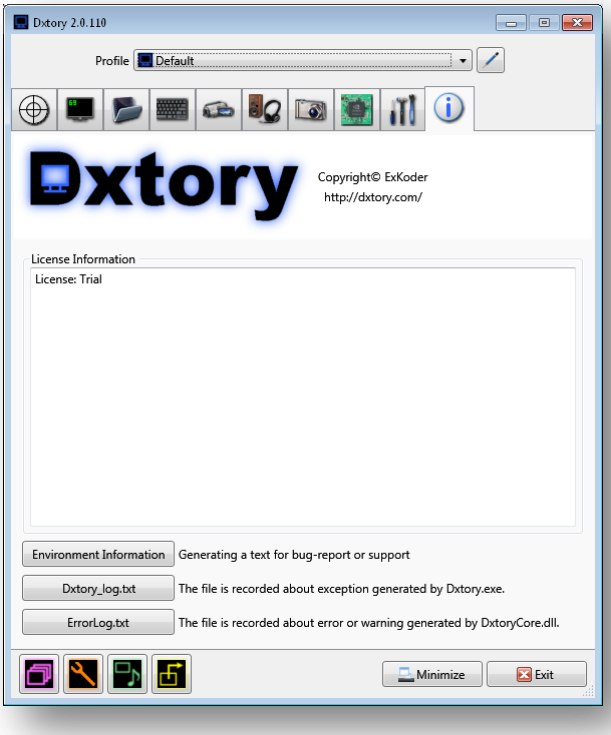
First program run and first impressions

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 : 00000000000001001101000000110000 (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)]
```

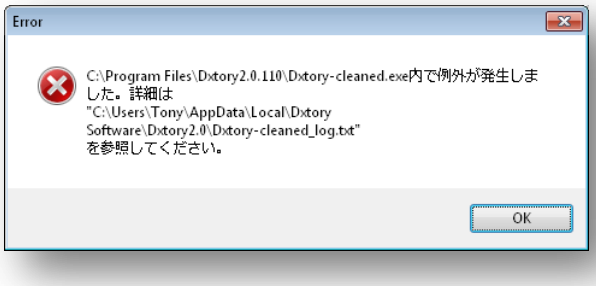
The output shows us that the target is obfuscated with *dotFuscator* and gives us useful information about the executable like entry-point token, required .NET framework version and the presence of a StrongName (we'll come to it later)[1]. Well, I think we can start rolling ☺

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)
```

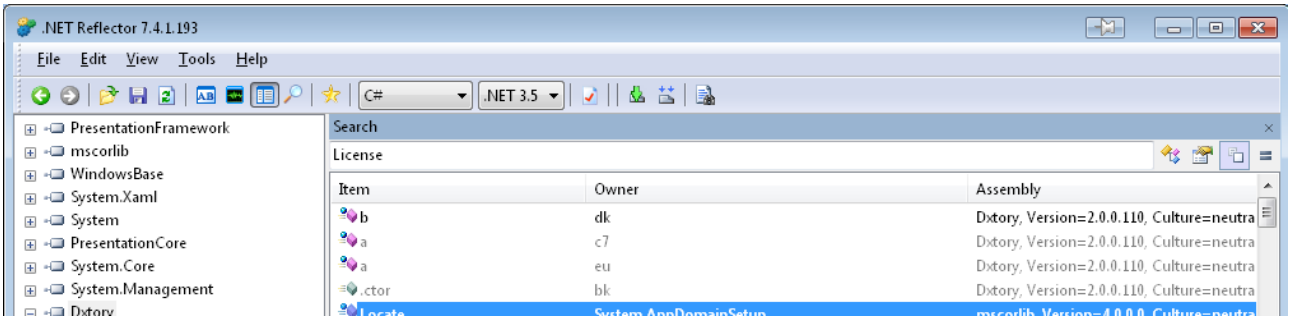
Several times 0xd4d (de4dot author) explained us that, when cleaned files have problems accessing resources, it's convenient to keep original symbol names using the "--dont-rename" option:

```
>de4dot --dont-rename 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 (…\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 verification[2]). ☺

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.

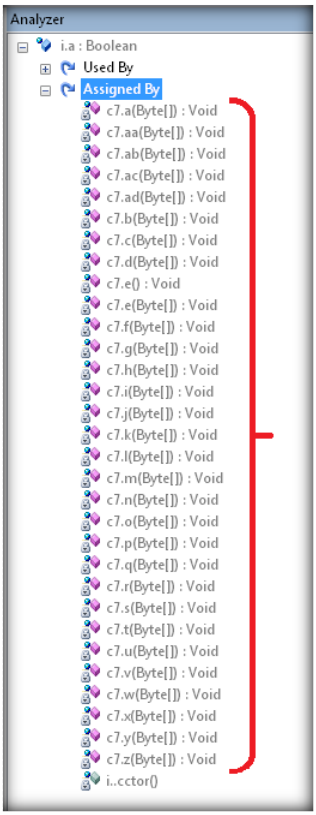
And last but not least let's see the fourth routine:

```
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:

c7.ab(Byte[]) : Void

```
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("XoA7ZbrrhQgD8BjJndpWTmGM4PIqeyfcB04");
      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/riwgY8A7J"));
            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, 0
         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);
      }
   }
}
```

c7.ab(Byte[]) : Void

```
.method private hidebysig static void ab(uint8[] A_0) cil managed
{
   .maxstack 5
   .locals init (
      [0] class d1 d,
      [1] string str,
      [2] string str2,
      [3] class cj cj,
      [4] uint8[] buffer,
      [5] class ds ds,
      [6] bool flag)
   L_0000: newobj instance void d1::.ctor()
   L_0005: stloc.0
   L_0006: ldloc.0
   L_0007: ldarg.0
   L_0008: ldloca.s str
   L_000a: callvirt instance bool d1::b(uint8[], string&)
   L_000f: brfalse L_00e7
   L_0014: ldsfld string dt::a
   L_0019: ldstr "XoA7ZbrrhQgD8BjJndpWTmGM4PIqeyfcB04"
   L_001e: call string cb::a(string)
   L_0023: call string [mscorlib]System.String::Concat(string, string)
   L_0028: stloc.2
```

Sebastien LEBRETON's Reflexil v1.3

Method definition

Instructions | Variables | Parameters | Exception Handlers | Overrides | Attributes

@Code type: IsIL
@Code type: IsNative
@Code type: IsRuntime
@Managed: IsManaged

Convention: Default

RVA: 78424 = 0x13258

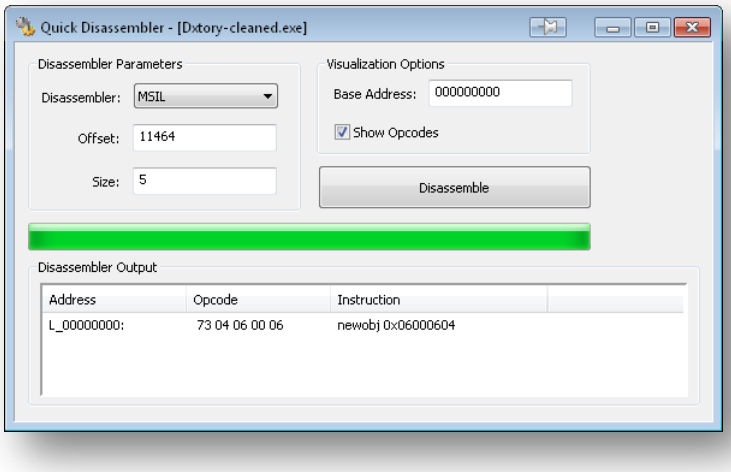
VA	00413258
RVA	00013258
File Offset	00011458

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	B1	00	00	11	73	04	06	00	06	0A	06	02	12	01	6F	FD

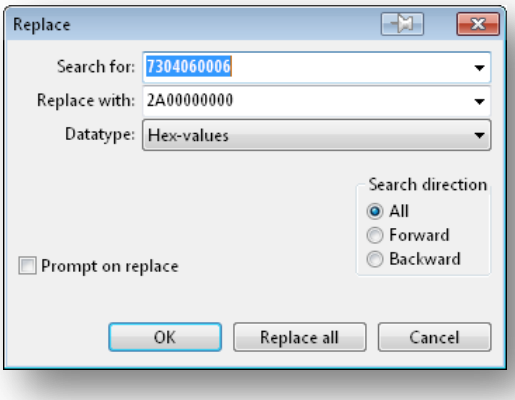
The aim of these routines is mainly evaluating our registration status to determine if we are indeed registered and/or if our trial period has expired: we'll take care to avoid any kind of control about real program status.

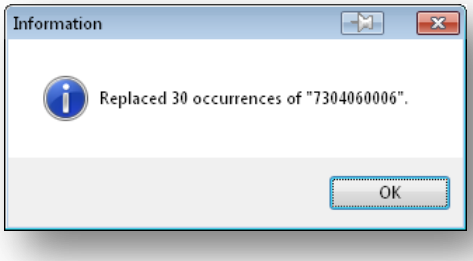
With the aid of Reflexil we can get method RVA: **78424**. In this version of the plugin the address is in decimal notation (in contrast with previous versions), so we need to convert it in hex (**0x13258**) before using it in CFF Explorer.

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`**



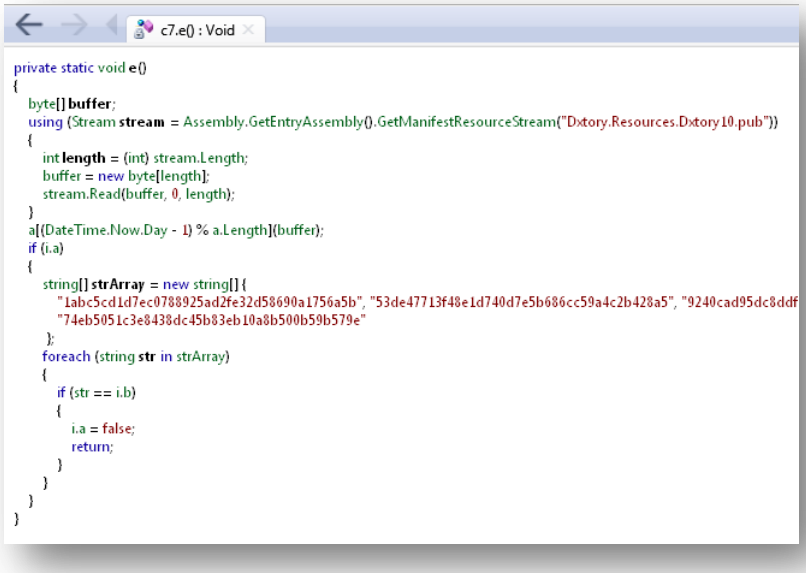
That instruction is important since it's a distinctive mark of these routines which gives us the opportunity to patch almost all of them with a single search and replace operation: in detail we'll replace the *d1* object creation with a ret (opcode 2A) ;). Open up our favorite hex editor and execute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

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



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

from:

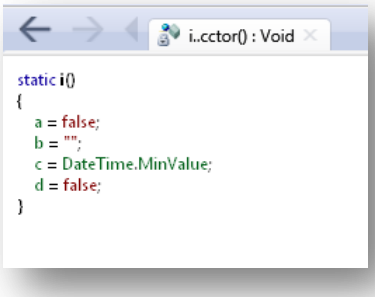
```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii

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:



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:

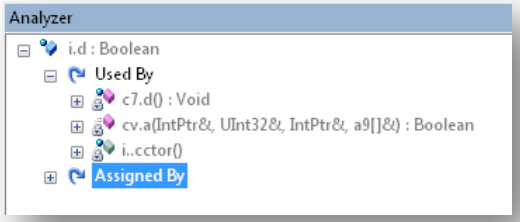
```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii

0001CC90                                  86 16 80 E3            †€ã
```

to:

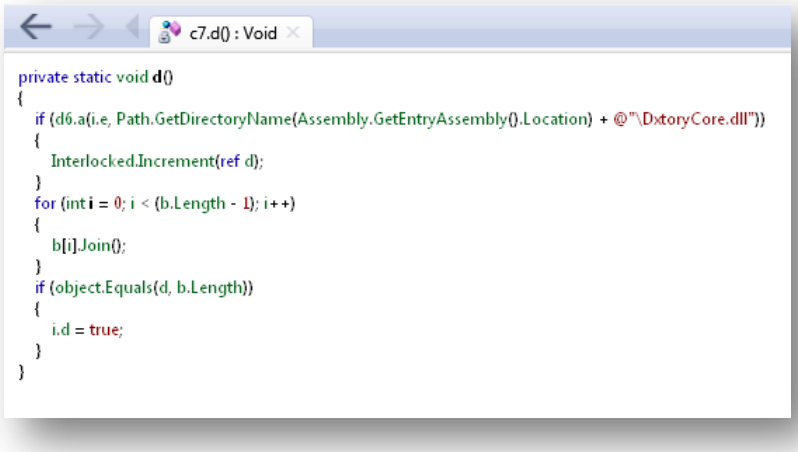
```
 Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii

0001CC90                                    17 80                €
```

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

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

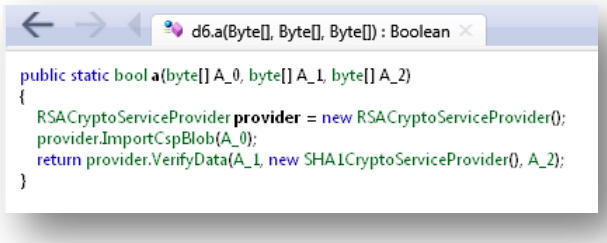


Let's try to understand its meaning by studying how it's used. Look at the first routine:



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:

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:

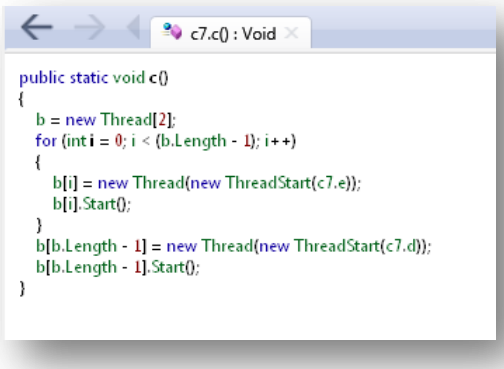


But why do we care so much about this verification? The answer is really simple: to complete our work we'll need to alter that DLL (we'll talk about it later) so, we have to avoid this check. However, we are still trying to understand the *i.d* role, therefore let's come back focusing on it.

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:

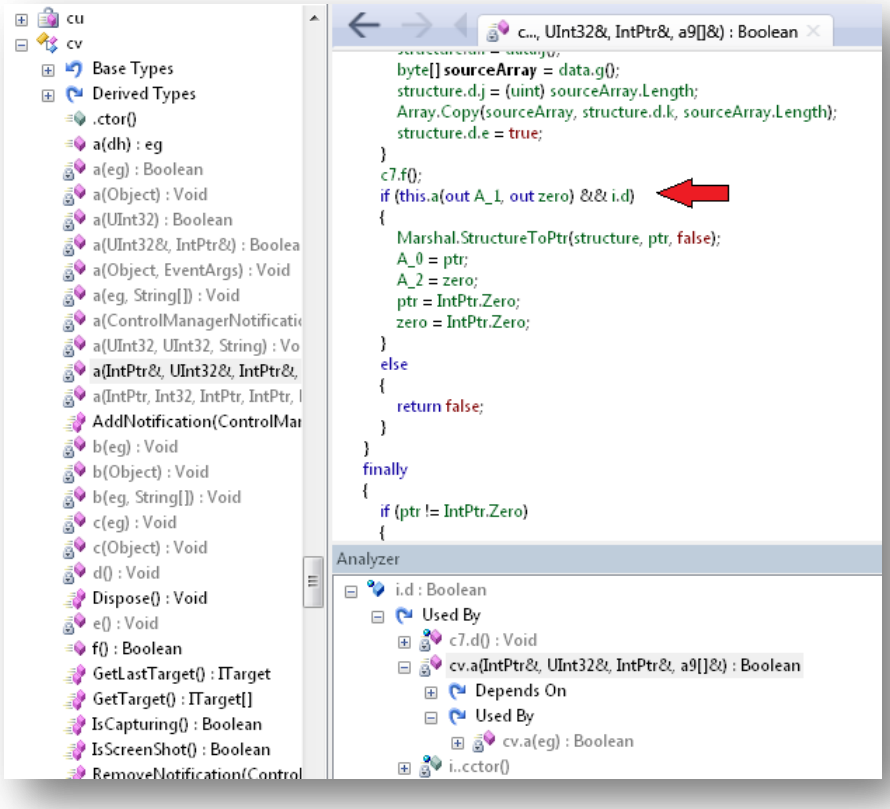
```
if ( object.Equals(d, b.Length) )
```

compares the c7.d value with the length of the *b* thread array, threads that are in charge of verifying product integrity[3]:



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:



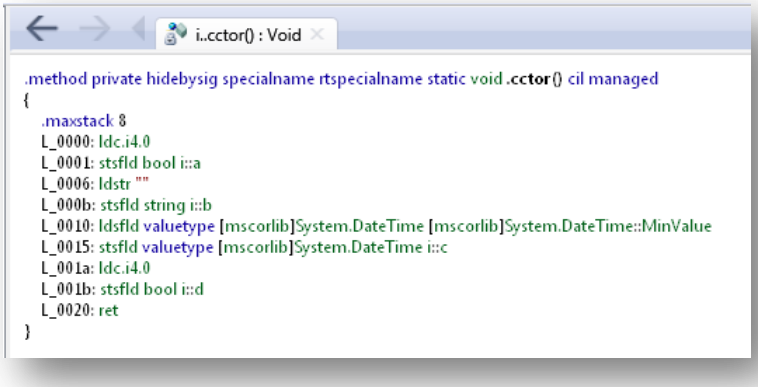
A value of false for *i.d* imply a return value of *false* for the whole routine! But, if we look at the caller[4] (`cv.a(eg) : Boolean`), we'll soon understand we can't accept that result ☺

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

[4] Here too through Analyze command.

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):

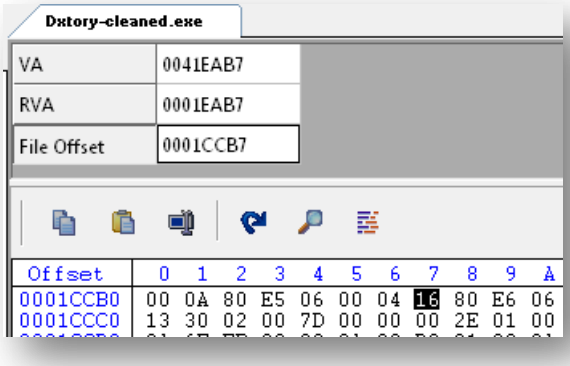


The IL view allows us to find immediately the address to patch: the *ldc.i4.0* that we need to change into *ldc.i4.1* is at 001a offset from the beginning of the routine; therefore we find the byte to alter at:

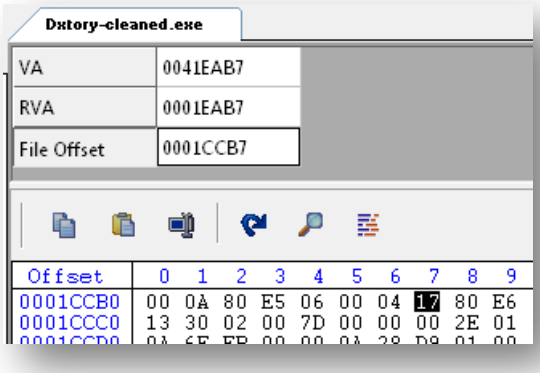
```
(RVA) + (Tiny Header) + (offset) = 0x1EA9C + 1 + 0x1a = 0x1EAB7
```

We change it from 0x16 to 0x17.

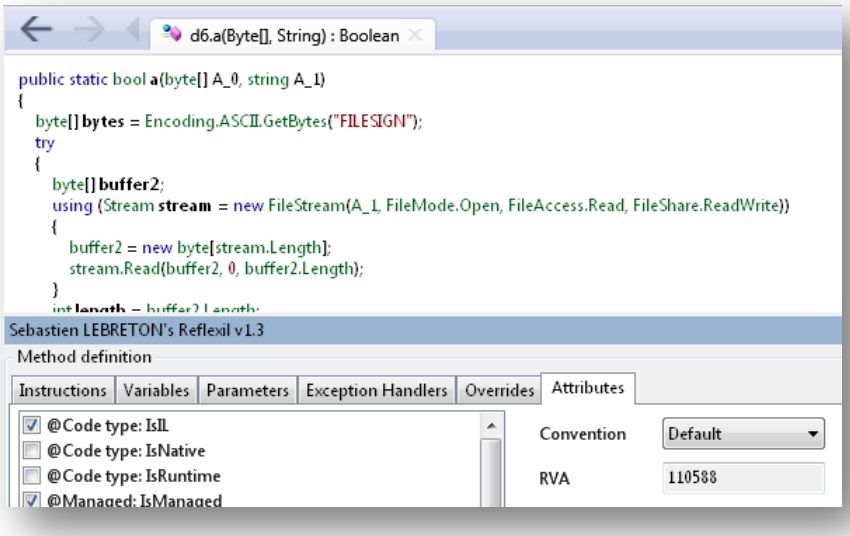
Before



After



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:

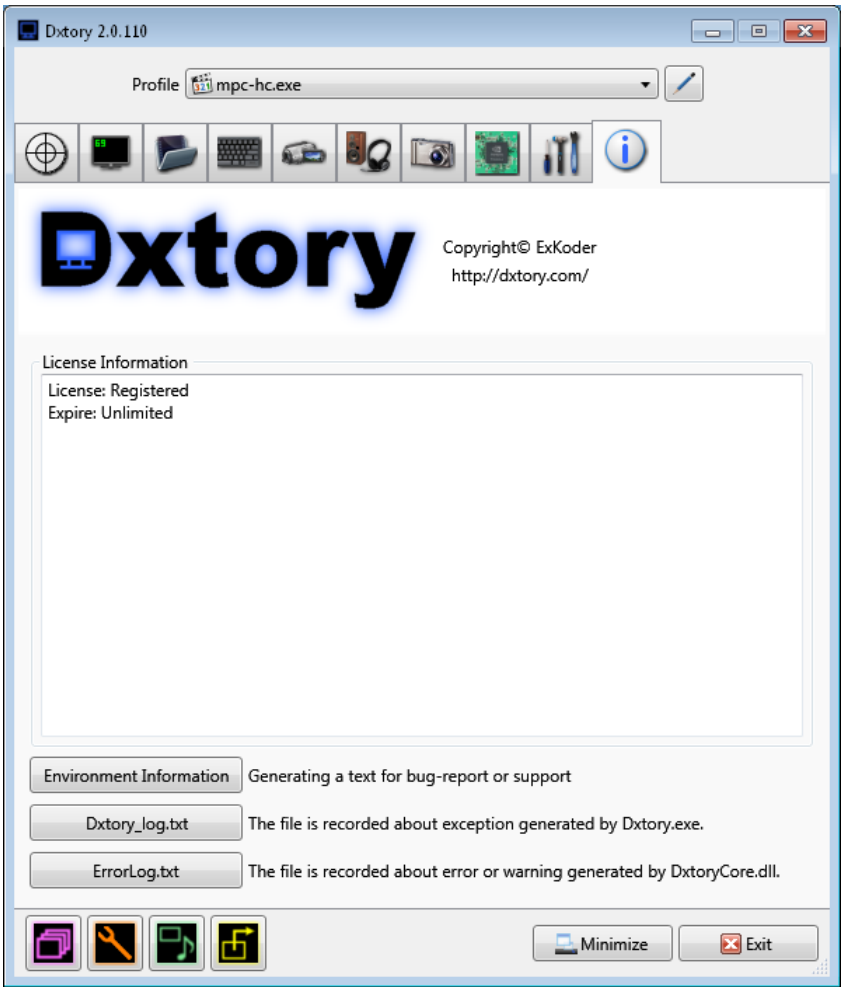
```
 Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00019200                        28 F1 02 00 0A                  (ñ..
```

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                  *...
```

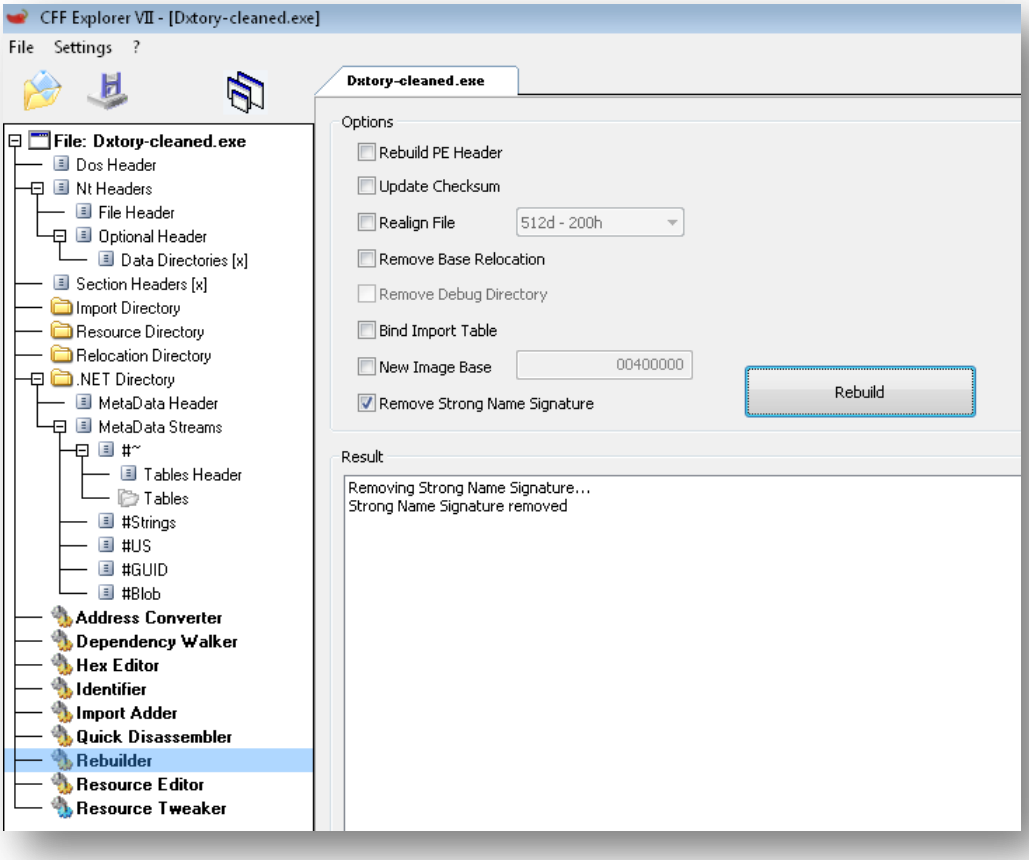
Basically we patch the routine so it will always return *true* ;)

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

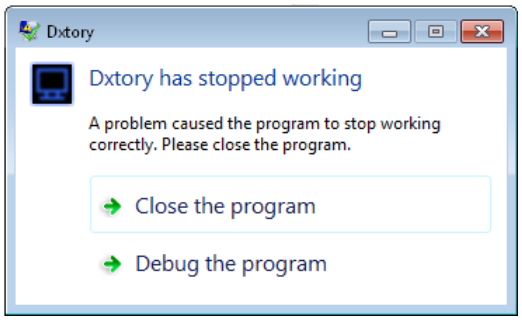


Make our executable run on every system

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.



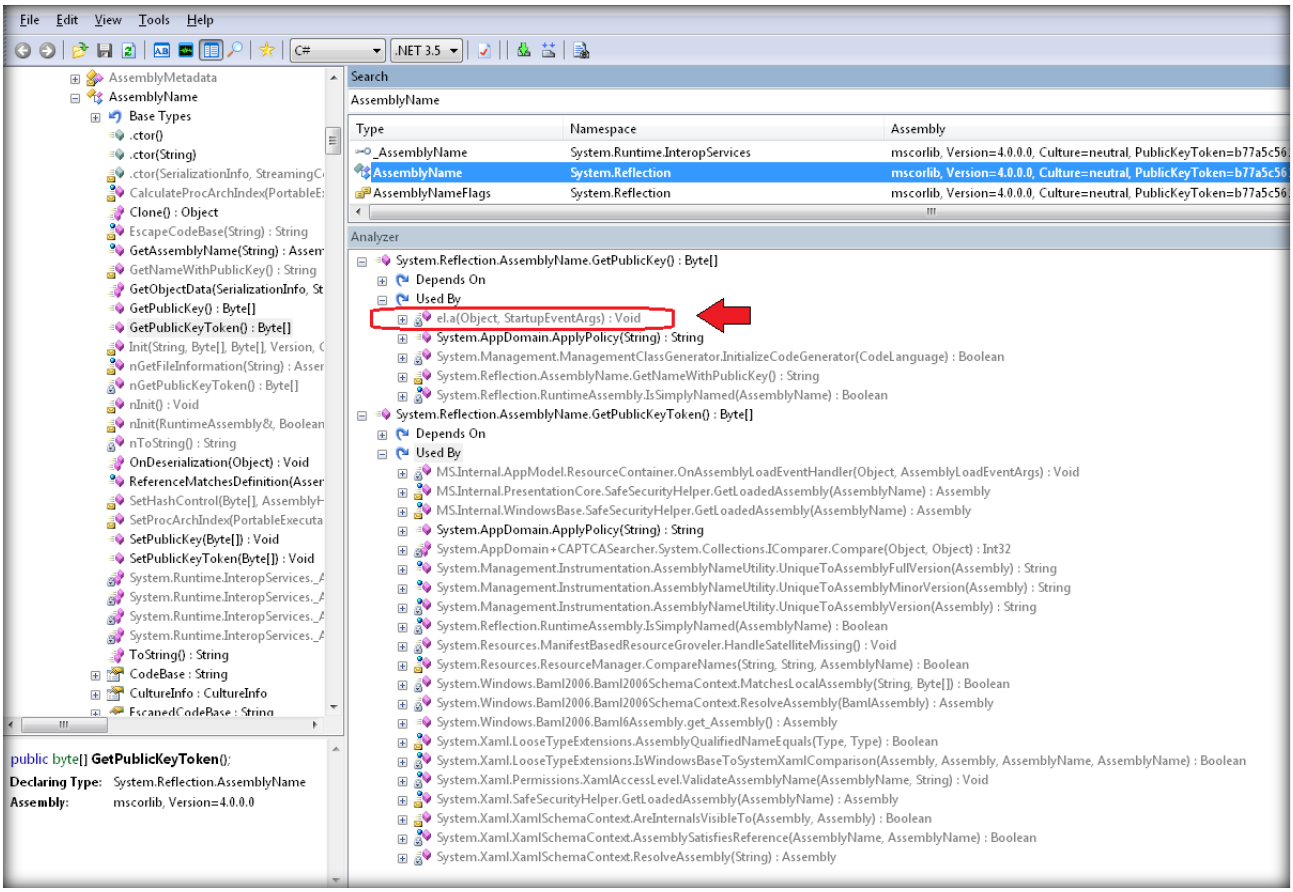
Let's save the executable and run again the program and … bang!



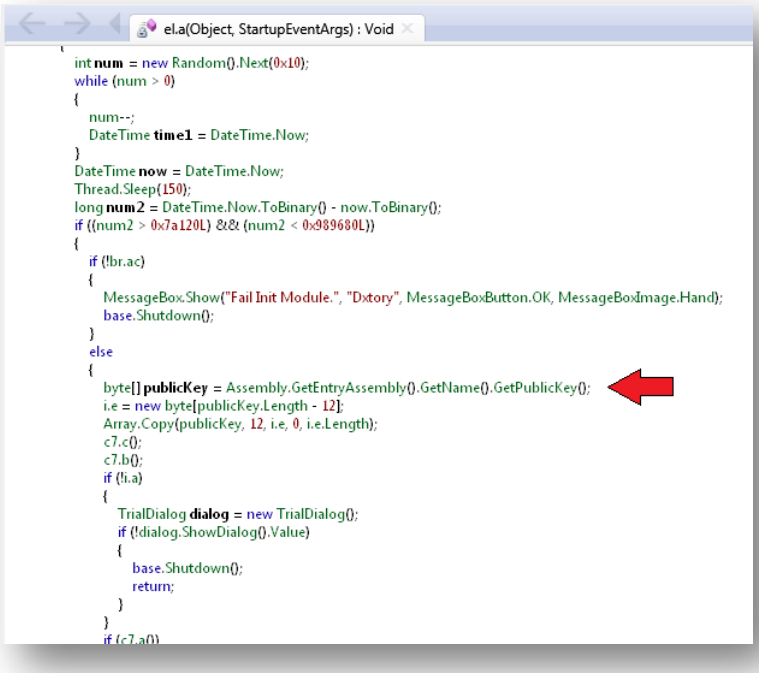
Rush another time towards the log folder (with the hope to find some useful error stack trace :P) but this time app crashes without writing anything. The problem showed up once we removed the Strong Name so we'll look in Reflector if the program uses PublicKey or PublicKeyToken[5].

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 .

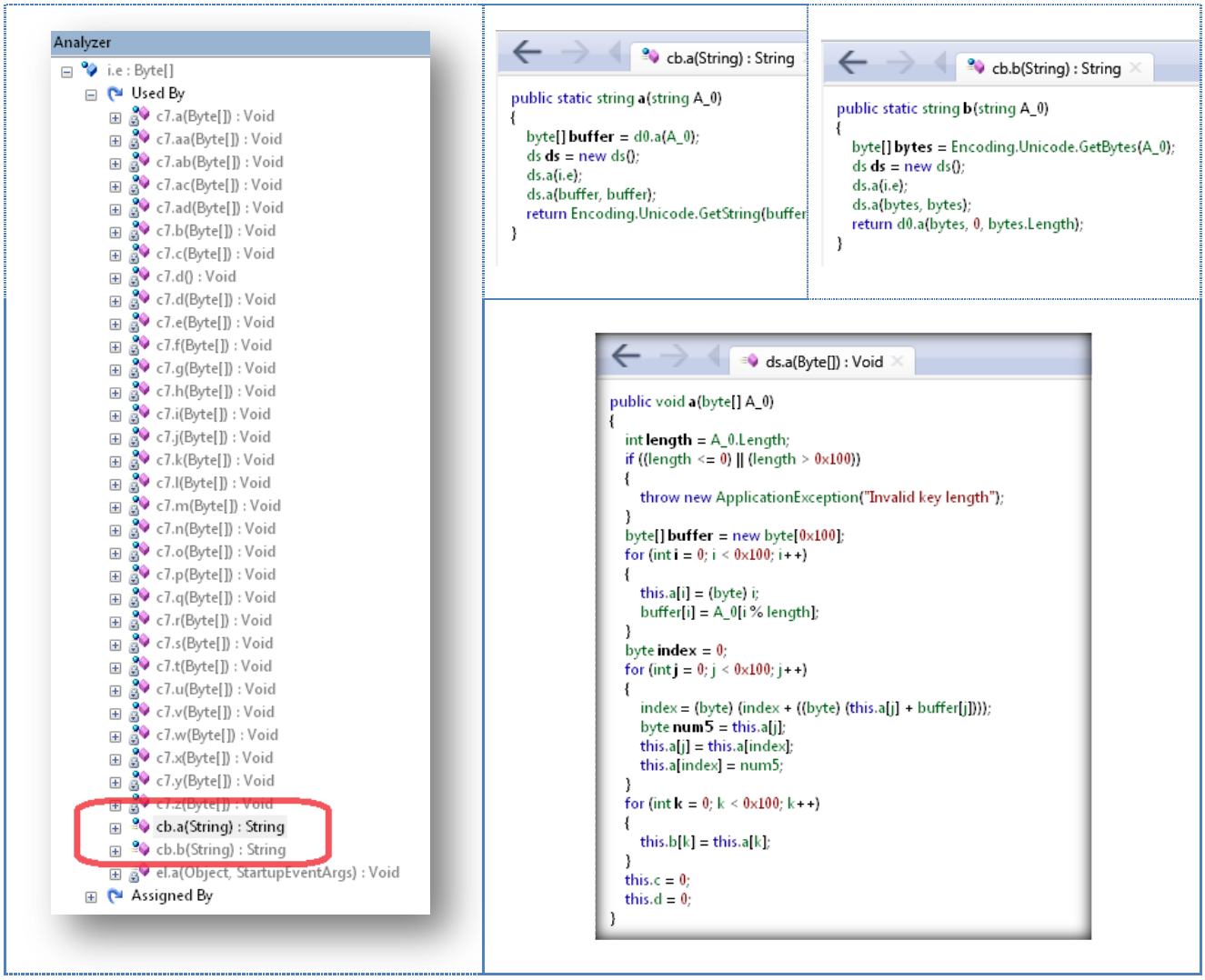


GetPublicKey has only one reference (see the highlighted routine in the picture). Let's go there :P



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 routines[6]:

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

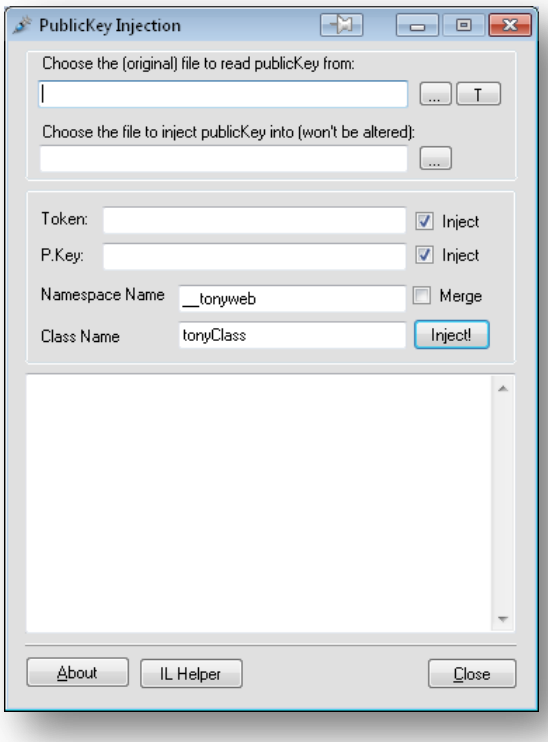


A quick way to solve this problem is to inject the right PublicKey and force the code to put the original injected public key into *i.e* and not the one read from the assembly (which is now *null* because of the Strong Name removal).

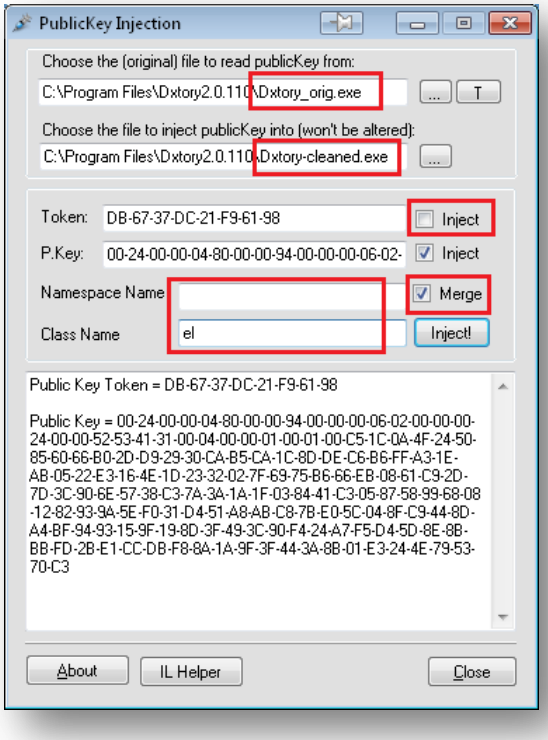
To do this kind of modifications we can use *Mono.Cecil* library which allows us, in a really simple way, to manipulate our assemblies adding, removing or simply changing methods, classes, namespaces, variables, strings, ecc. Suffice to say that the same Reflexil plugin uses this library to perform his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

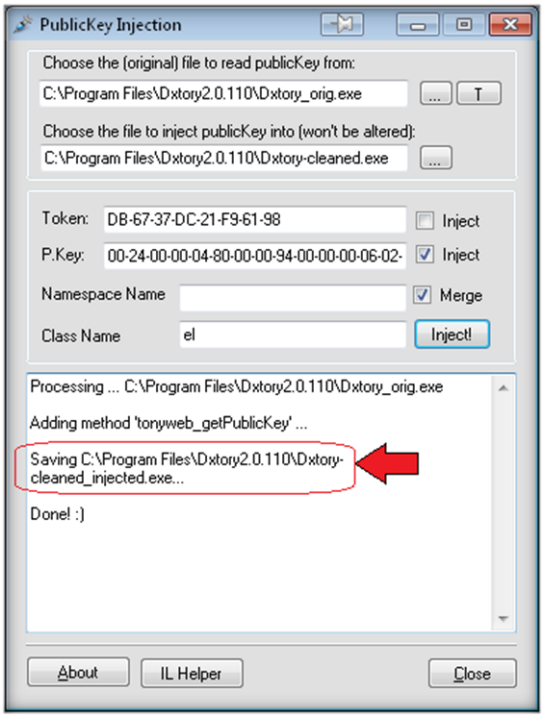


This tool can inject methods which return byte arrays matching the PublicKey and PublicKeyToken of a reference assembly. Suppose we worked this far on the Dxtory-cleaned.exe file and we have the original target renamed as Dxtory_orig.exe. Drag Dxtory_orig.exe onto the first text field (or click on the 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).

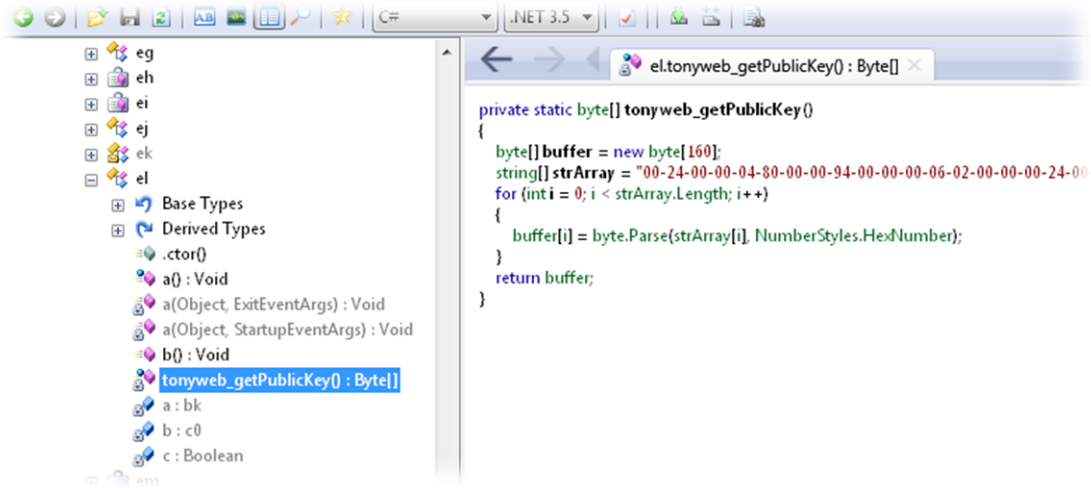


We can place the new methods in a new class or "merge" them into existing types: for convenience, we inject the Dxtory assembly original PublicKey in the same class where the method that requires it resides in: *el* (default namespace).

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.

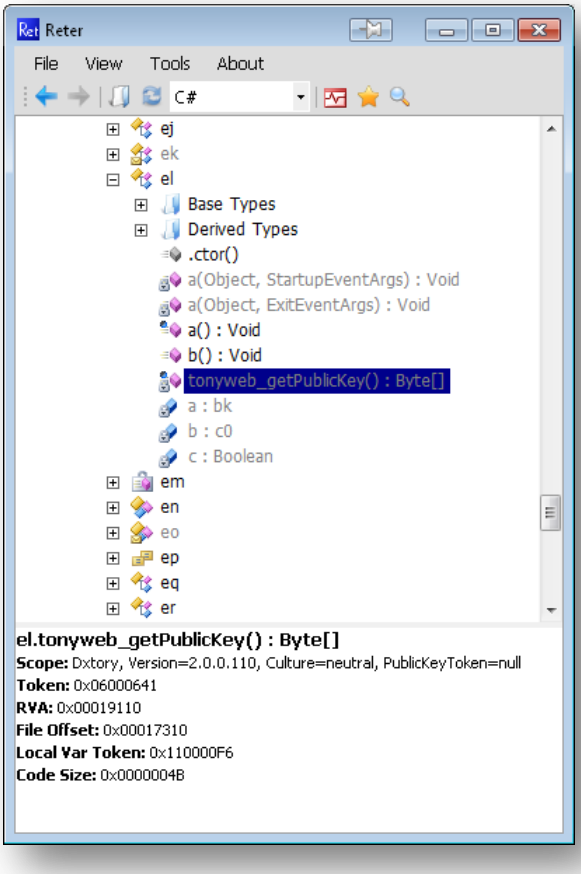


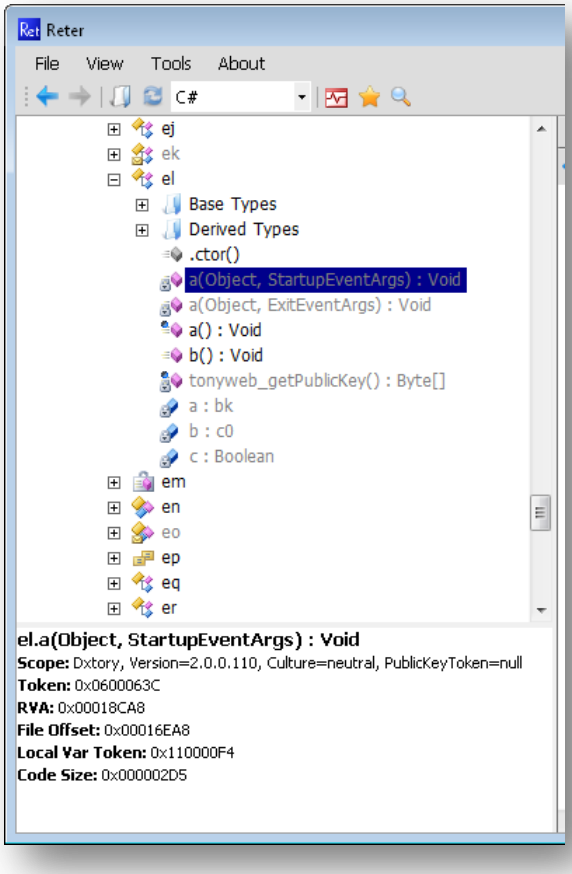
If we open the resulting file in Reflector we can actually see the routine.



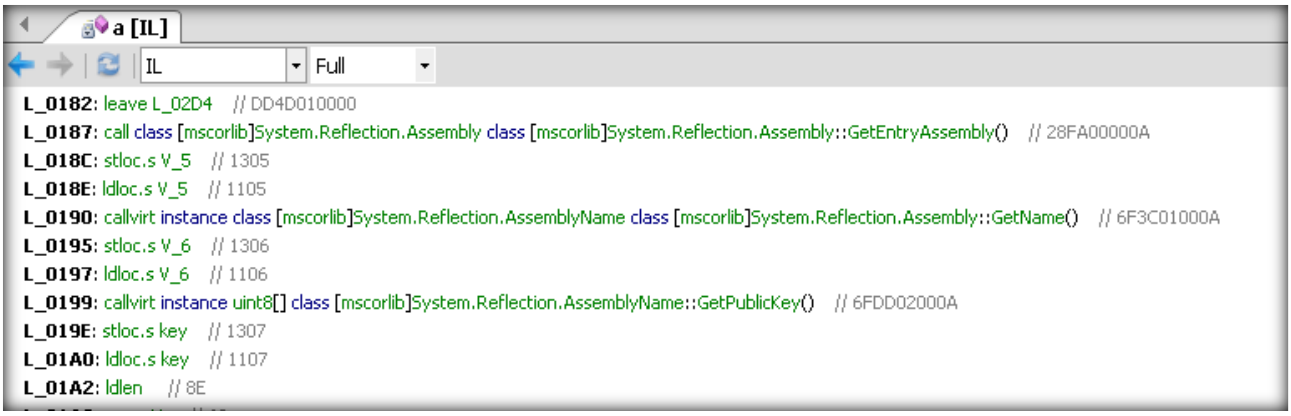
Now we have to use this new method to read the public key instead of allowing the program reading it from assembly metadata.

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();
```

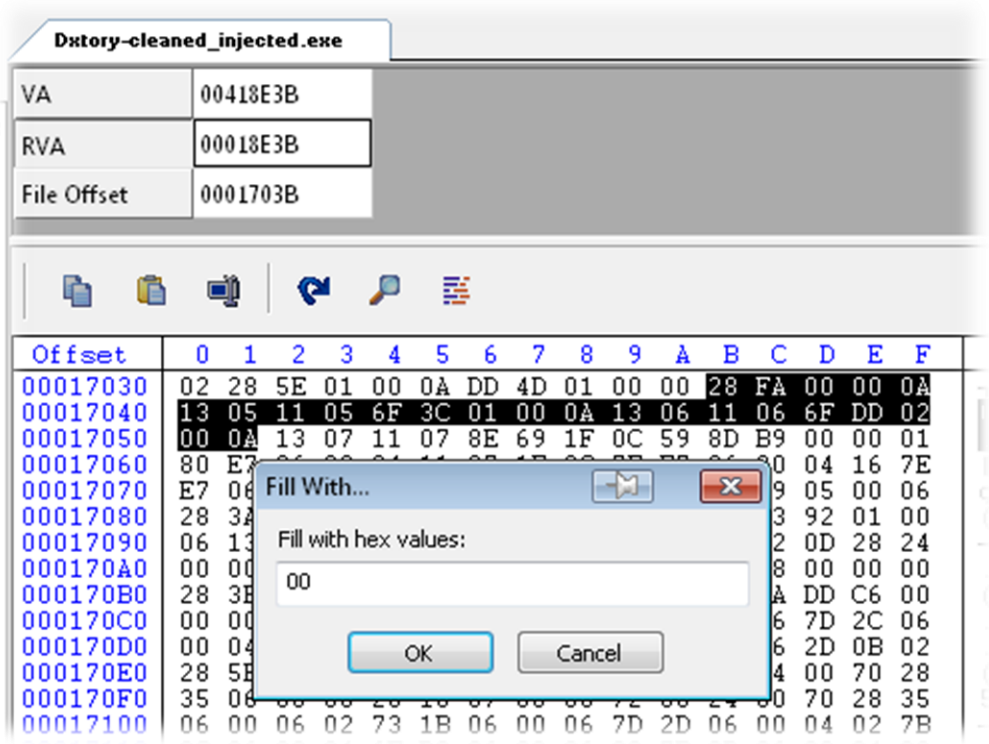
becoming

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

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

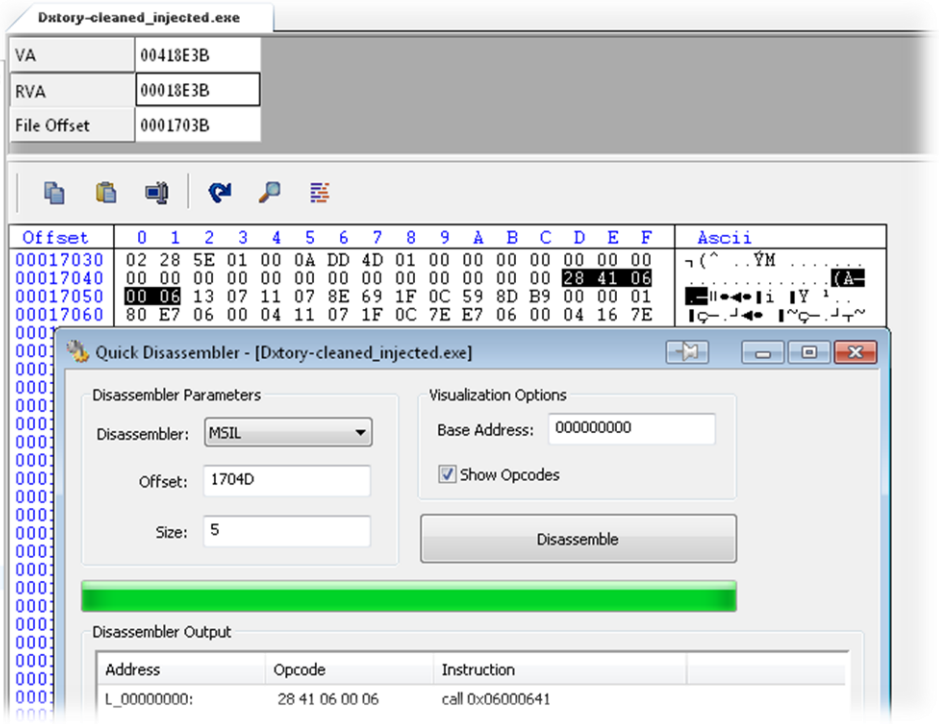
RVA: <routine_rva> + <fat_header_len> + <code_offset> = 0x00018CA8 + 0C + 0187 = 0x18E3B
File Offset: 0001703B

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

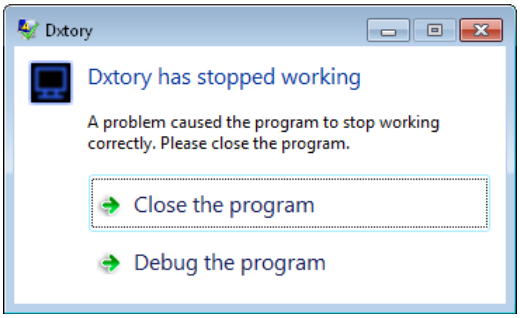


And prepare the new call:

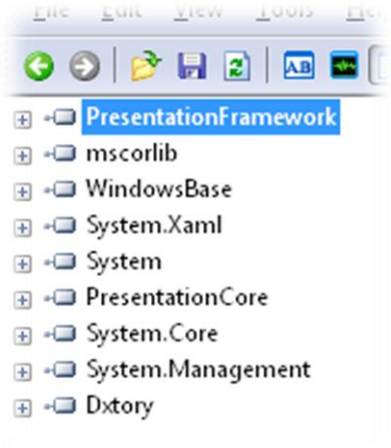
op-code for call instruction:	0x**28**
token of the routine to call:	0x**06000641**
bytes to write:	**28 41 06 00 06**



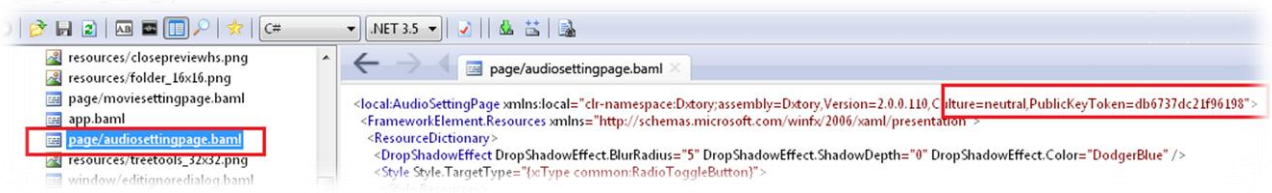
Save the file and try again running the exe and BAM!:



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.*



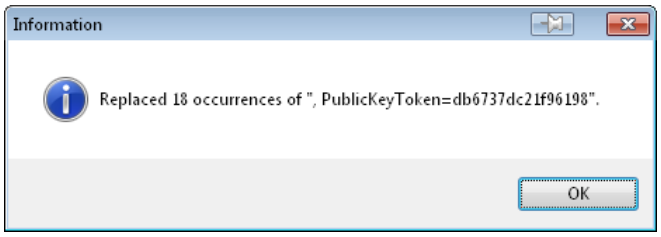
In that framework, UI and resource description files – baml/xaml – refer to assemblies through their complete names, therefore including public key token (thank you to Kurapica for this hint!) . If we browse main executable resources from Reflector we can see, for example:



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 version[8]: the watermark is still there on the captures. :-/

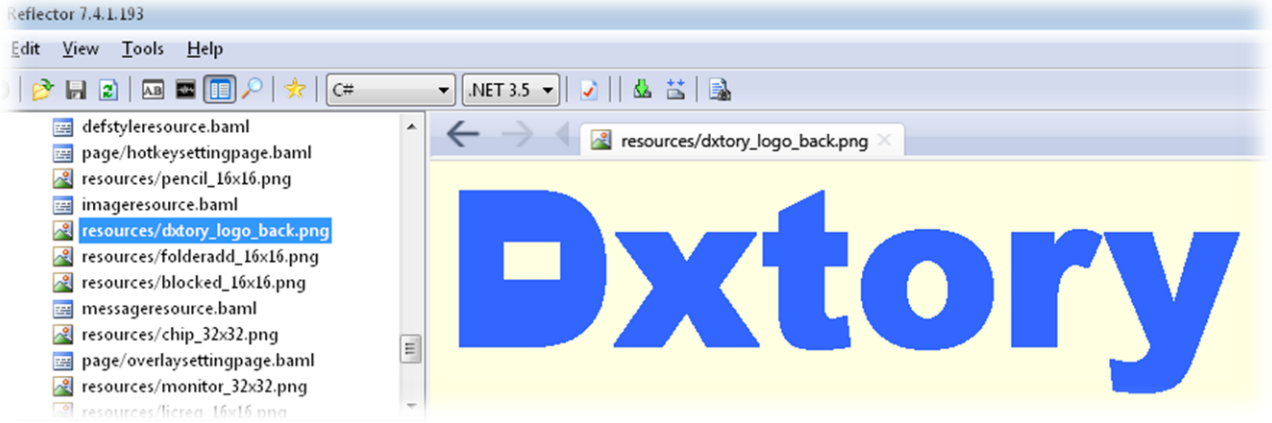
The watermark picture must be located somewhere and used by the program during video recording. Most likely we already spotted two suspect *png* files in main application folder: *Src16x9_Dest4x3.png* and *Src16x10_Dest4x3.png* both representing a sort of watermark mask.

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 (eheh, clearly those two files have the only purpose to divert our attention ☺).

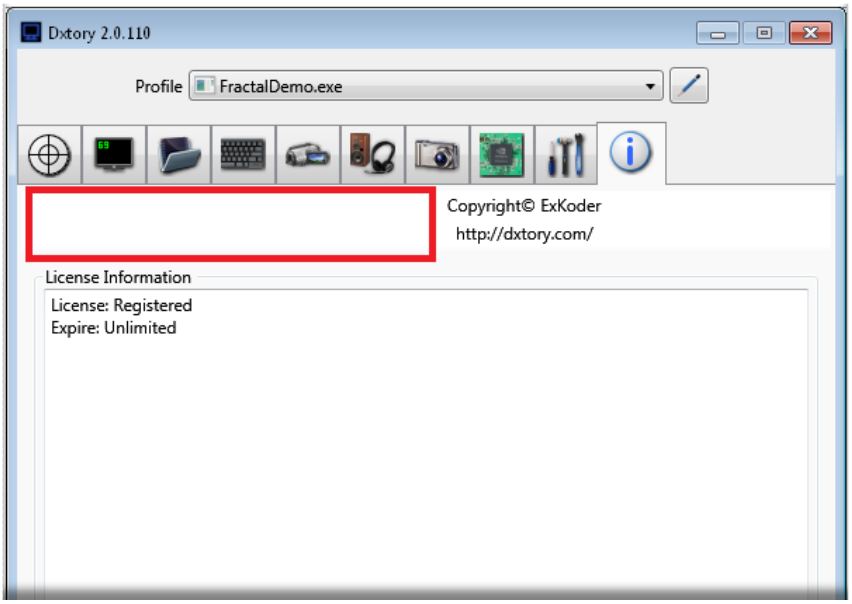
Now our first thoughts are towards the resources: the program has to retrieve somewhere the image it places in overlay. Open the file in .NET Reflector and sift through resources; two specific items attract our attention:

[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.

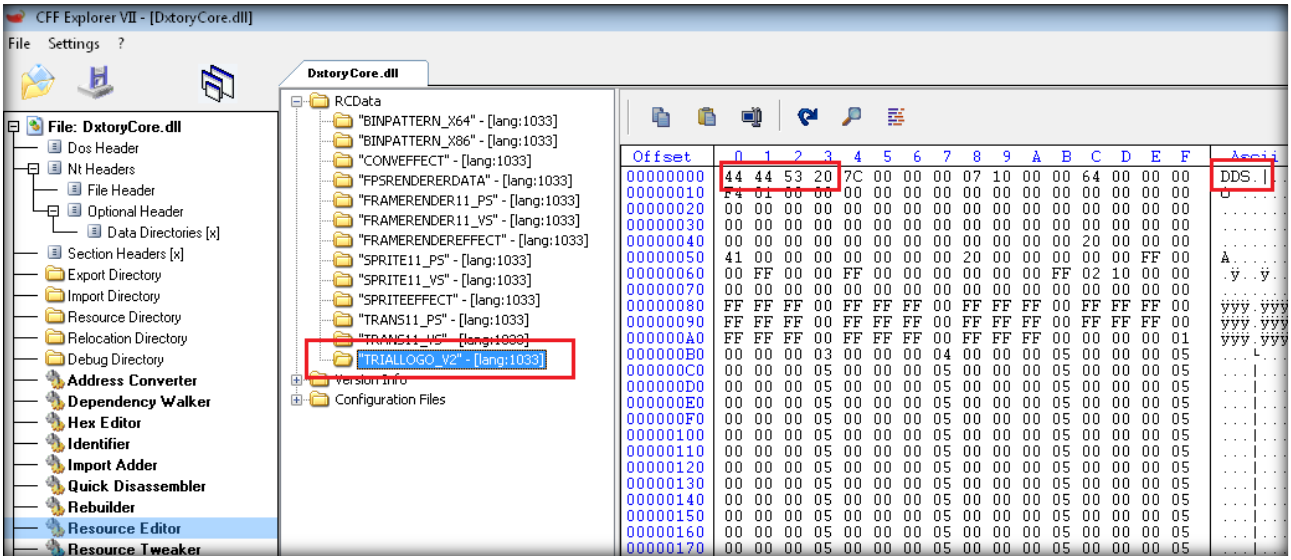


Besides these pictures, there is nothing more suspicious among resources and, even looking into the code, we find no obvious references to the watermark. We just have to look better around: what is available?

Patching phase 2: All together in the native world

Discover the resource

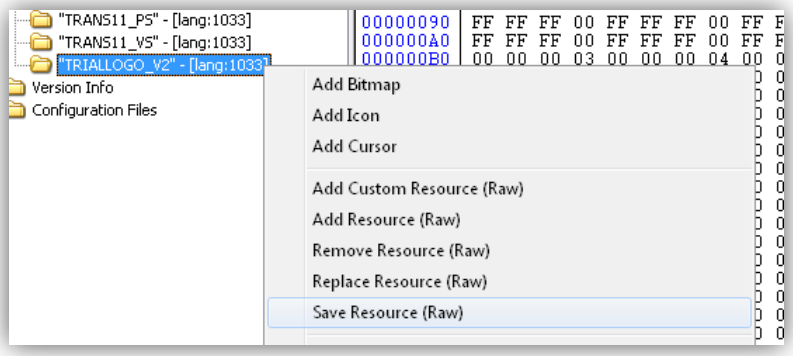
Recall what we saw earlier: the main .NET executable takes care of checking the Dxcore.dll file integrity; this is a very important evidence: our first suspect is therefore that file. ☺ Let's analyze its resources (always with the great CFF Explorer) and we soon have this view in front of us:



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

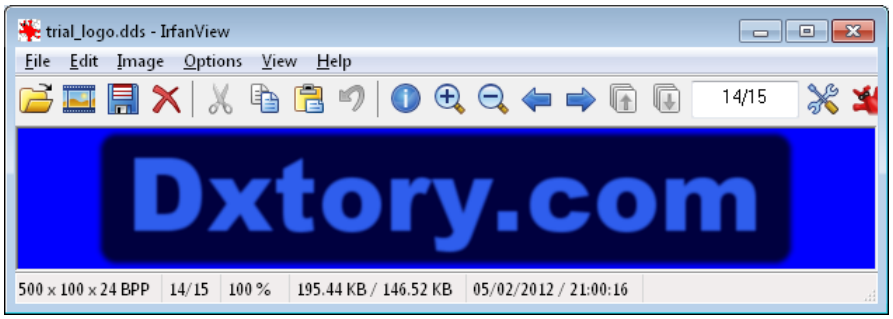
It's easy guessing this is a DDS (Direct Draw Surface) resource type; since I never worked with Direct-X I knew nothing about them: on MSDN we can find a lot of information and DDS header is described in great detail. There you can find the offsets of picture's height and width and, in the first tries, 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:

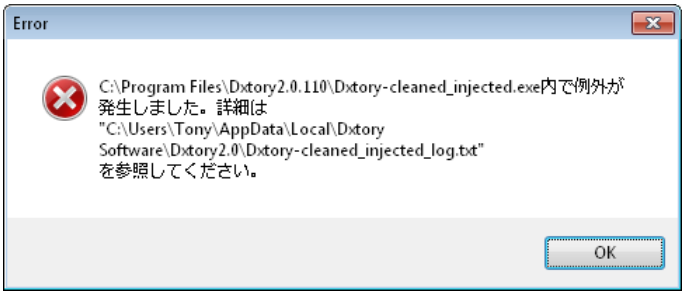


and open it with a viewer like IrfanView we can confirm we finally found the watermark we're looking for[9] ;)

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



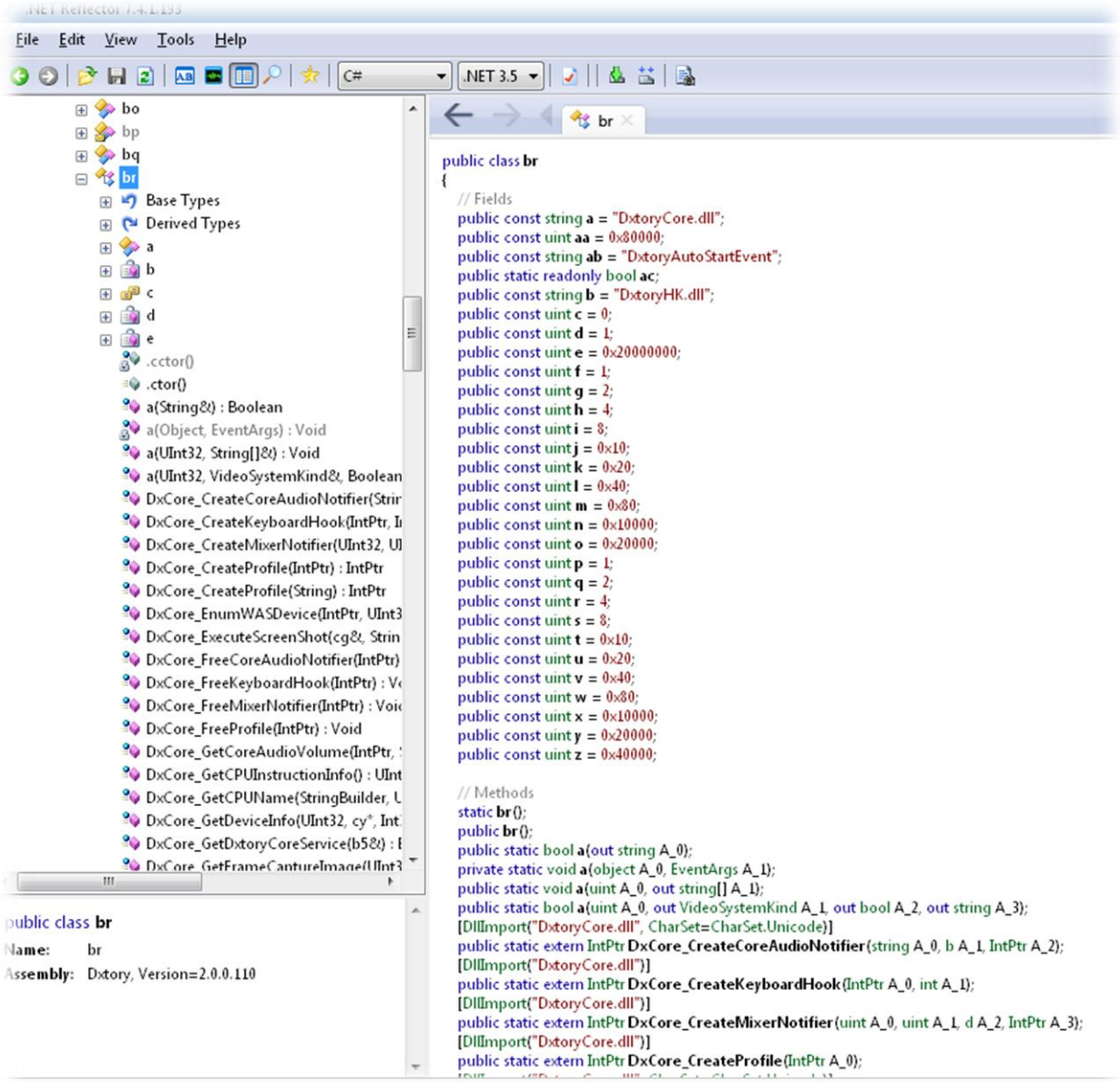
We might consider making a backup of the DLL, removing the resource and try starting the application. If we did such a modification, though, we would realize quickly we won't solve the matter so easily:



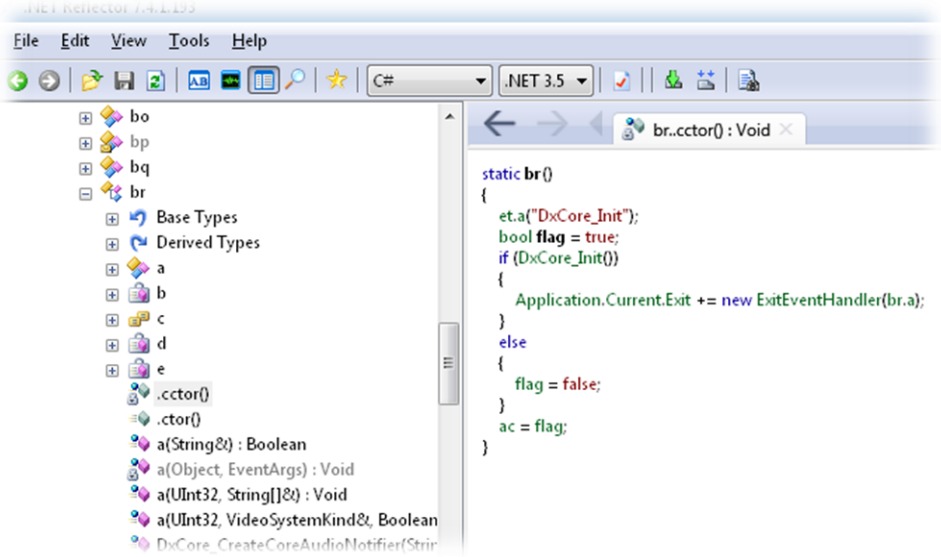
Since the program ask us to look at the log so kindly, we open it and find:

```
=== 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 el.a(Object A_0, StartupEventArgs A_1)
```

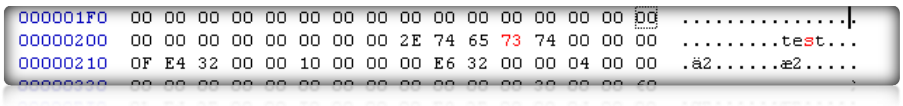
A quick look in Reflector makes us understand that *br* class is the 'interface' towards the program injected DLL(s) like Dxcore.dll and DxtoryHK.dll.



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):

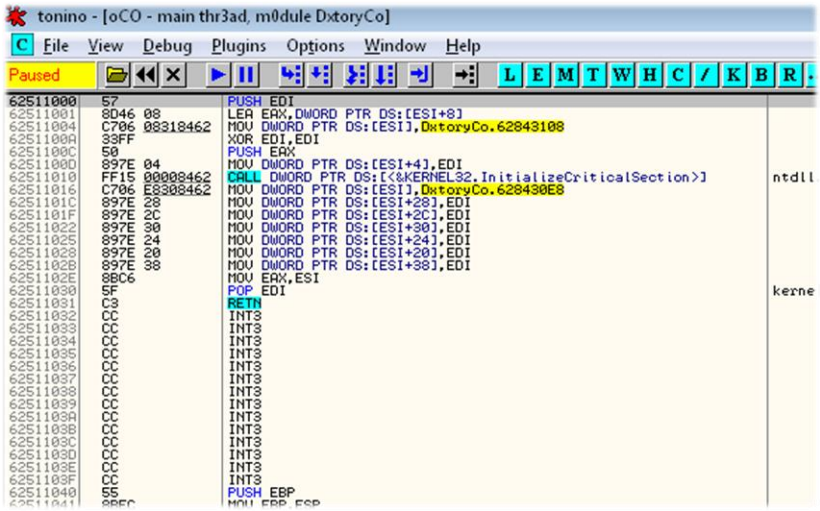


Save the file, start again the program and BAM: same error!

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 section[10].

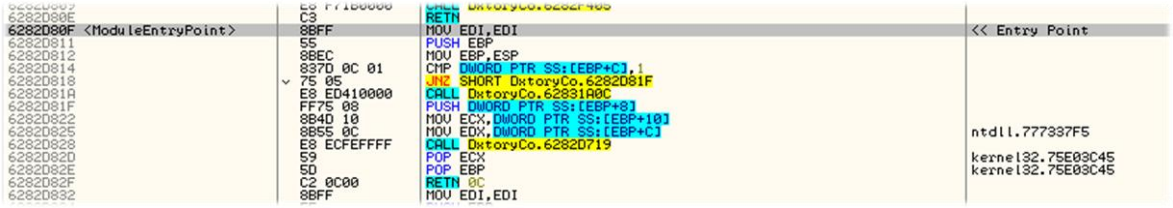


[10] Note that the addresses will be almost certainly different on your system.

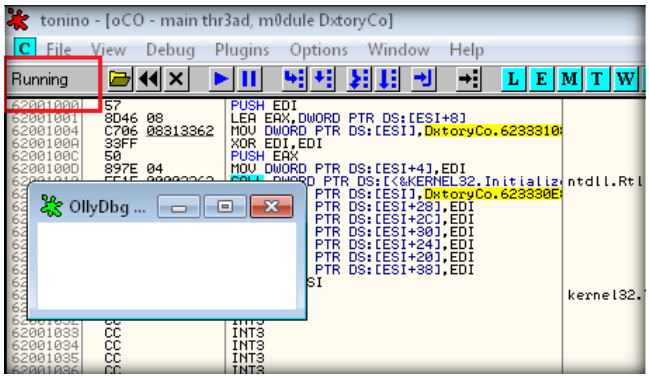
If we open it also in CFF Explorer, we can get entry-point RVA: **0031D80F**

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:

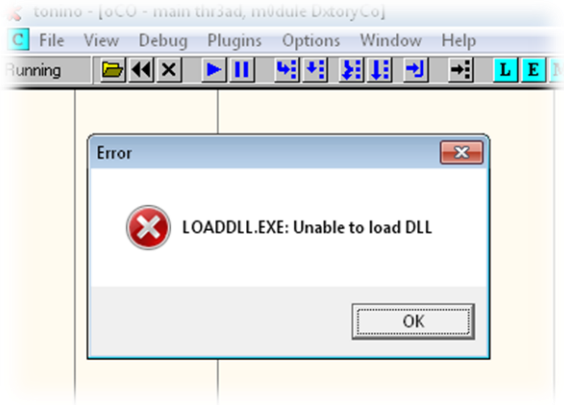


Press F9 (run) and see that the DLL "runs":

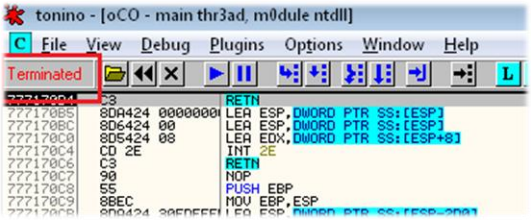


Ok, all seems fine till here: Olly allows us to work flawlessly with the executable. Try now opening the *modded* DLL (the one with the altered text section name for example) just out of curiosity: did we already delete it? :P Mine, renamed as *DxtoryCore_.dll,* was still in the recycled bin :D

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

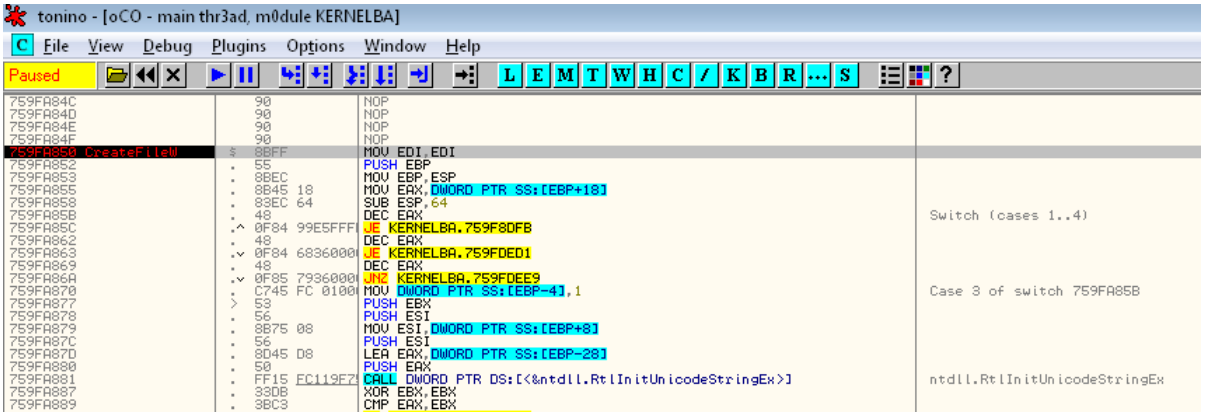


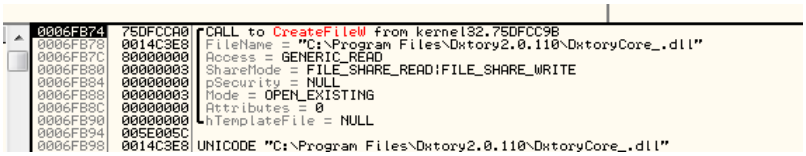
At this point we have no more choices: if we click on OK we can't expect anything other than this:

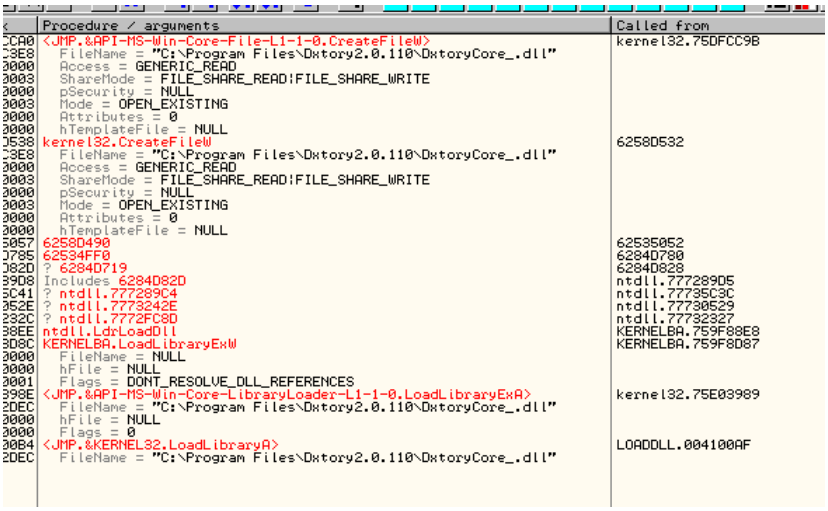


The logical conclusion is that the integrity check we were looking for is inside the same DLL: that's the reason why we could not find anything in .NET code ☺

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).



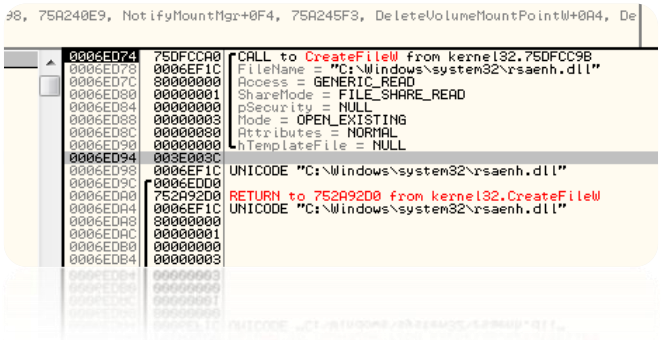




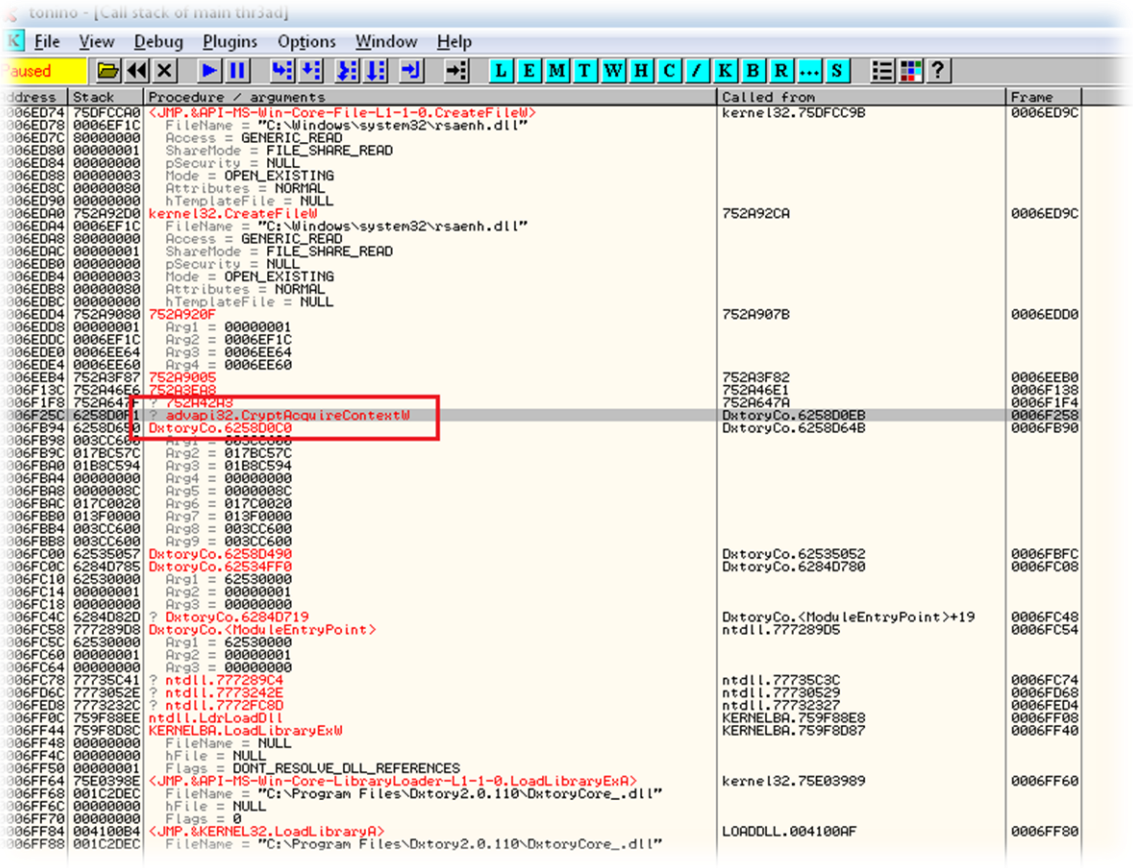
CreateFileW - break 1

The first break, consequence of the loaddll.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.

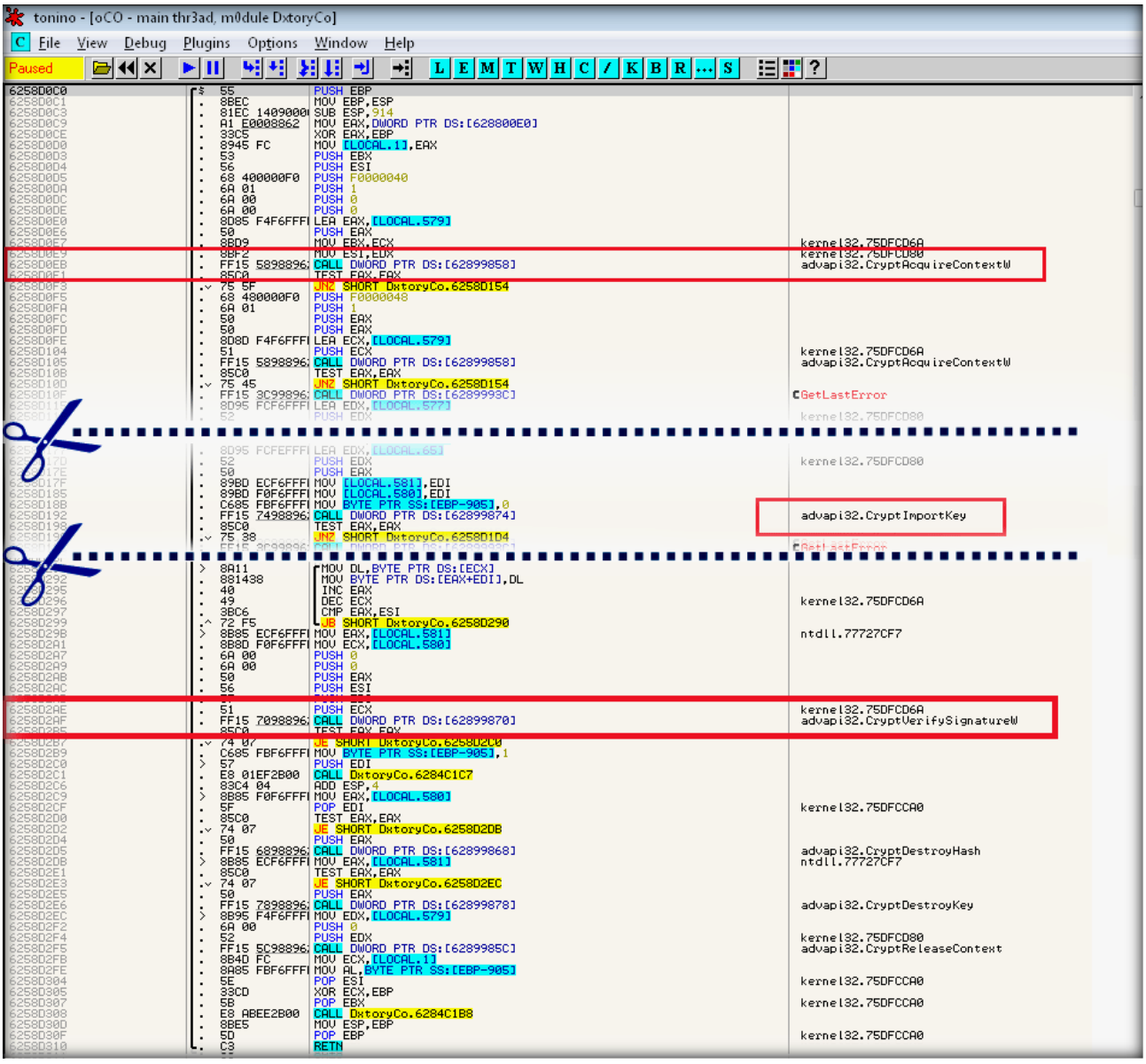
Apart from that we have no particular clue that DLL code gets executed. Let's move on, only for the moment, to the next break by pressing F9:



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

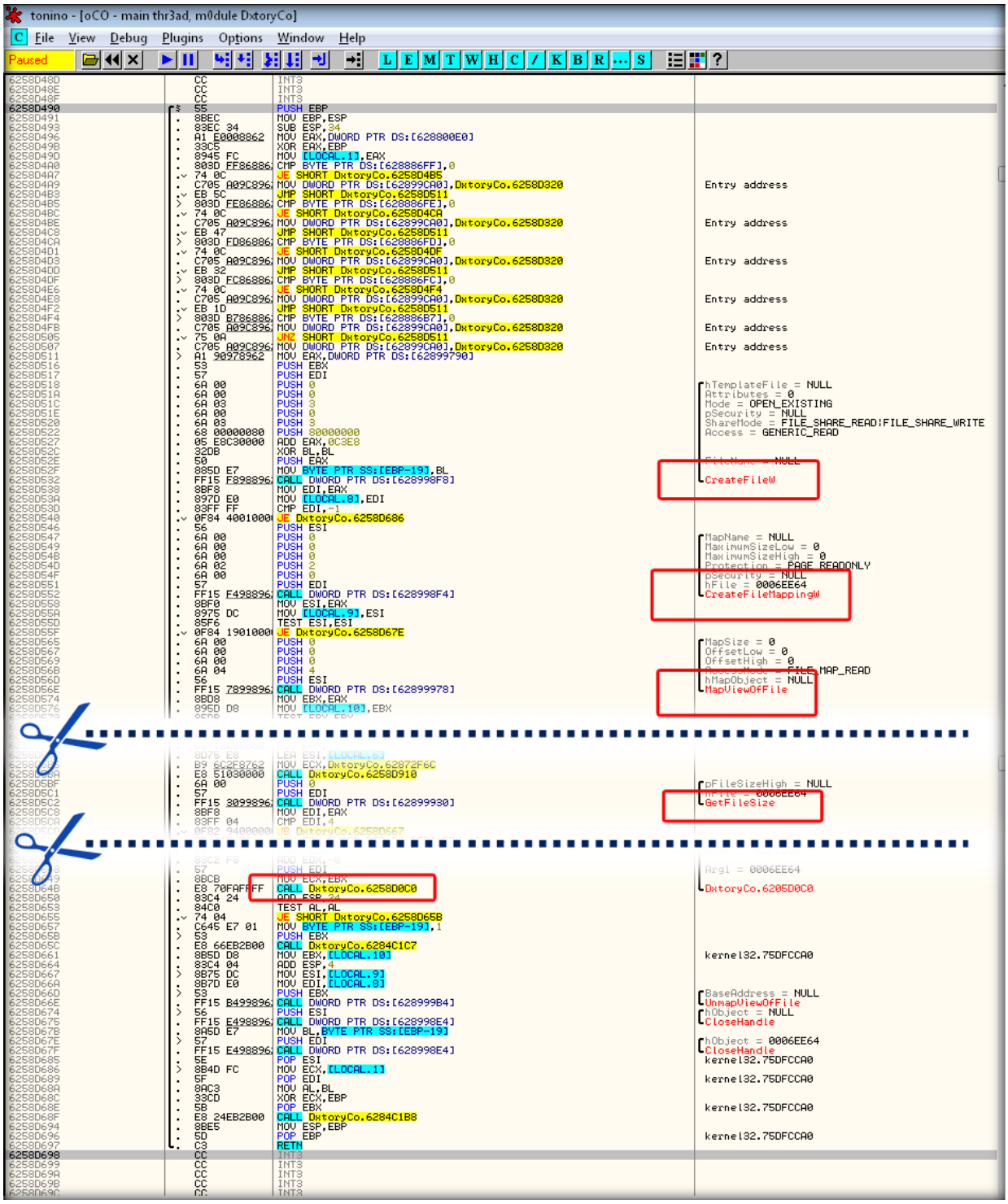


The *CryptAcquireContextW* routine, symptom of some cryptographic operations in progress, catches our attention; obviously very suspicious. Let's follow the highlighted address: **6258D0EB**.



It’s quite obvious that this routine takes care of checking a "digital signature" ... did it remind us of something? Didn’t we already stumbled upon something like that? Of course! ☺ If you do mind, we met a local reference to the FILESIGN string in .NET code, nearby a sanity check of the DLL ;) We will return to this routine shortly but I guess you already know what to do if the result of that *CryptVerifySignatureW* doesn’t please us :D

With an eye on the call stack, follow this routine’s caller: **6258D64B.**



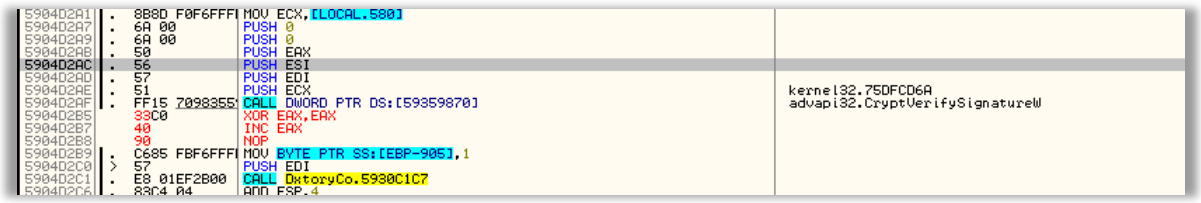
Now we're talking business! Note, in the first place, that the CreateFileW call you see here is, in fact, the one which made our debugger break for the first time (refer back to the picture at page 32), the one we skipped because of its not-so-clear implications: *my fault! Sorry*.

From the image above we can clearly see an integrity check is performed: once having the file loaded (i.e. the same DLL), it gets mapped into memory to be read and analyzed. Next, the code asks for the file size and finally it calls the following routine:

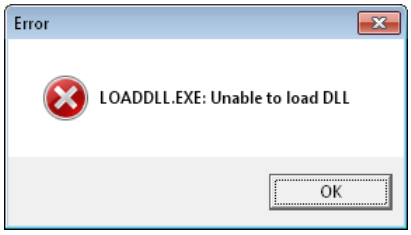
```
CALL 6258D0C0
```

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. ☺

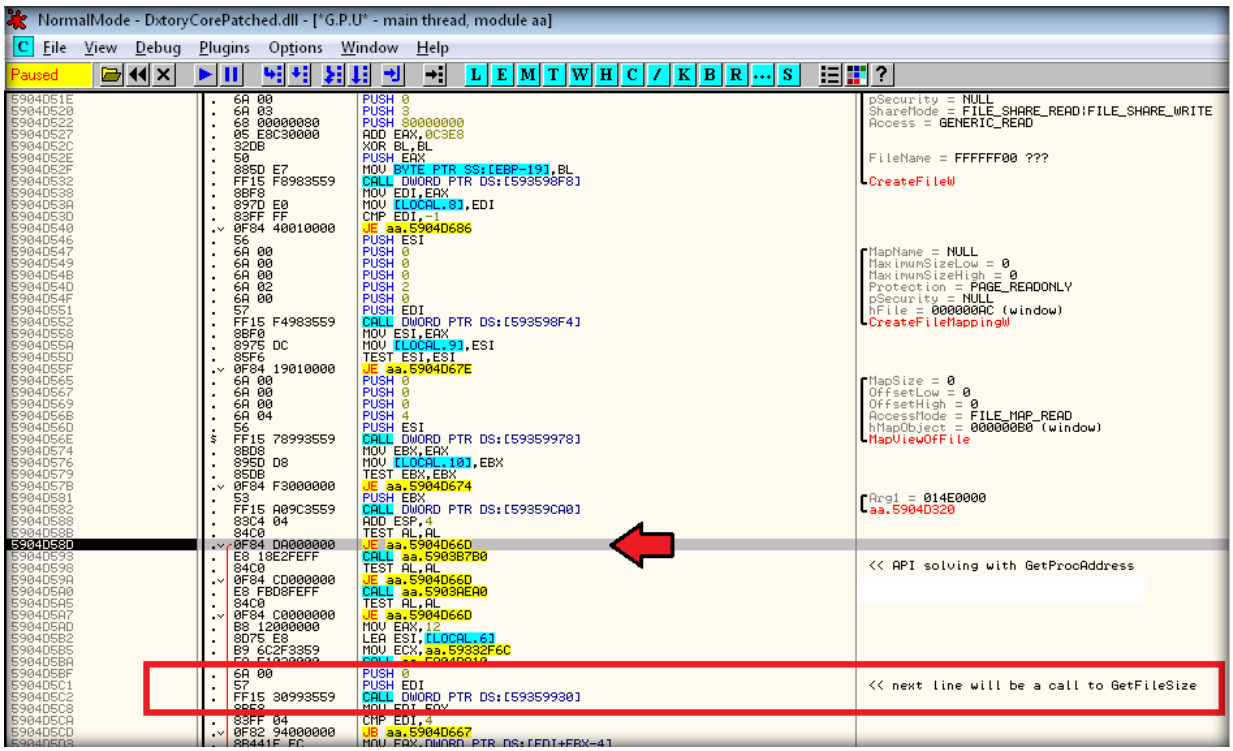
Maybe, our first idea could be waiting for the second break on CreateFileW, then entering the signature verification routine and directly *patching* the *VerifySignature* outcome, something like the following:



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.

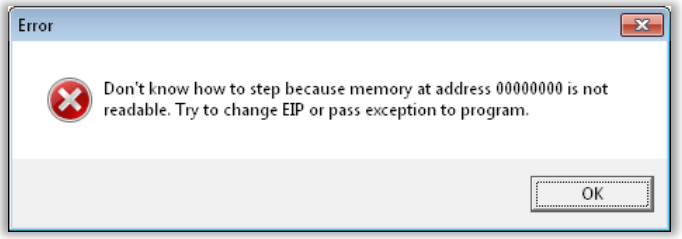


Breakpoint again CreateFileW and CTRL+F2. If we return to DLL code from the first break on CreateFileW, we have to trace just a few instructions to see that a number of routines “fail” (the first one is the call above the red arrow)



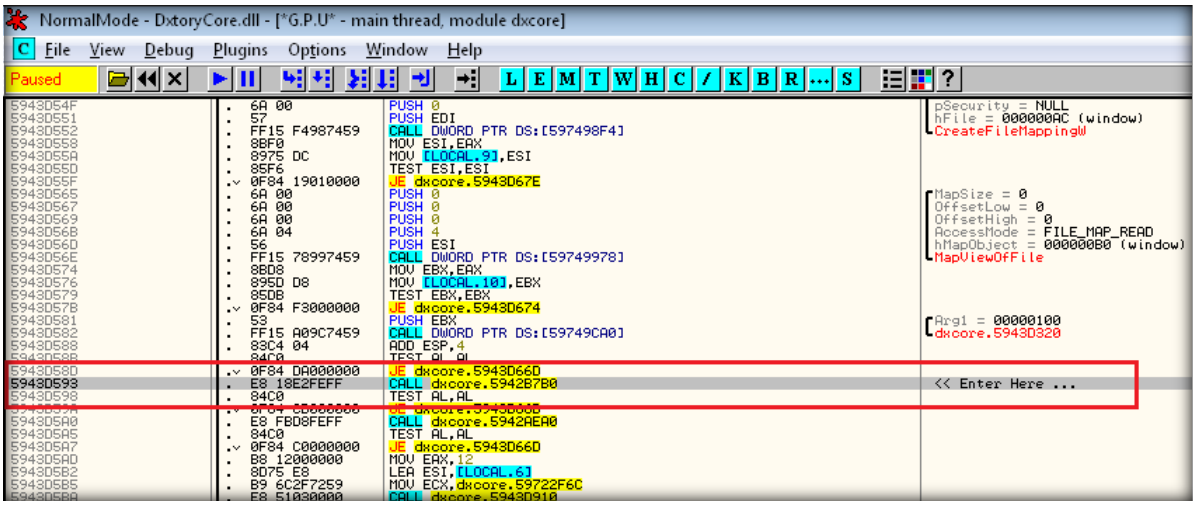
Even if we invert the Zero flag and continue stepping we would see the next call failing as well and also the one which follows it. It’s interesting to note that the *GetFileSize* call, we saw so clearly before - backtracing from the second break on *CreateFileW* - now is not even visibile. Moreover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

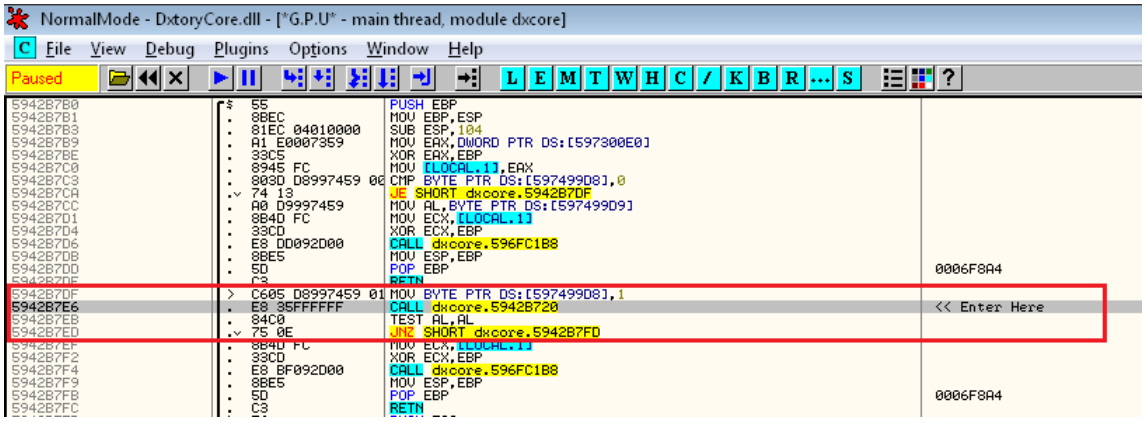


It's worth noting also that other APIs like CreateFileW/CreateFileMappingW, stored in target IAT, are instead available since from the beginning.

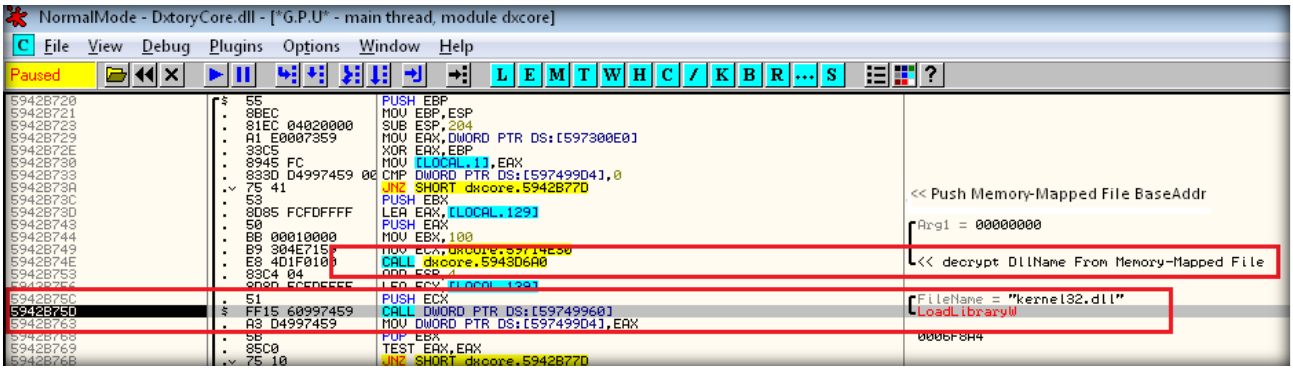
If we trace again the *original* DLL we quickly realize that its integrity is a necessary condition for correct import resolution. To be completely sure of that, it suffices digging a bit more into the code.



Following the highlighted call we'll find ourselves here:

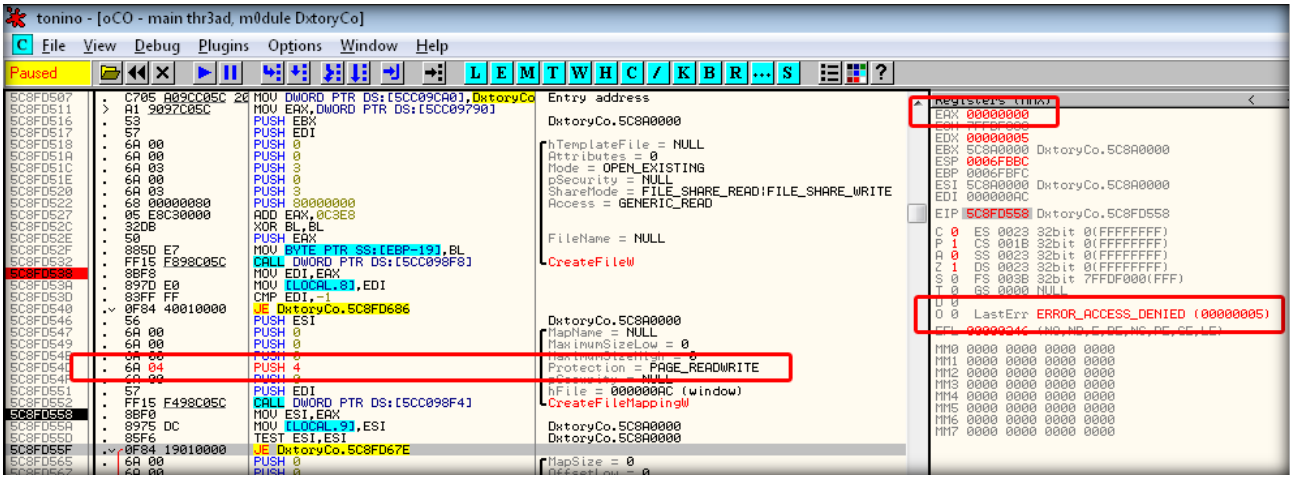


A `TEST AL,AL` instruction is always intriguing, right? :P Let's enter the *call* so we are here:



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 ☹

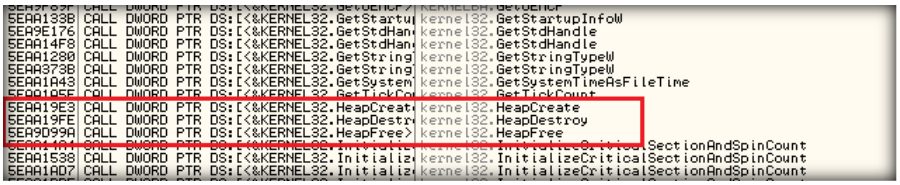


I really don’t know why the call will fail … probably this is due to our being in DLL initialization phase – DLL_PROCESS_ATTACH – where some kind of writing lock is in place. Though this outcome catches unprepared us, in no way it shell make us desist from our aim. That’s a valid idea! We just need to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

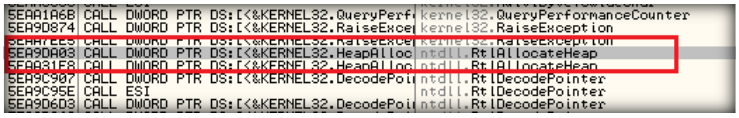
We need to find a way so the code “can see” an untouched view of the file. We just saw we can’t directly alter the view … but why not just copying this view in a place we have full control on, conveniently hide our modifications and “pass” this new “view” of the file to the checking routines? ☺ That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



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 it[11] exported by ntdll.



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

Let's start dancing

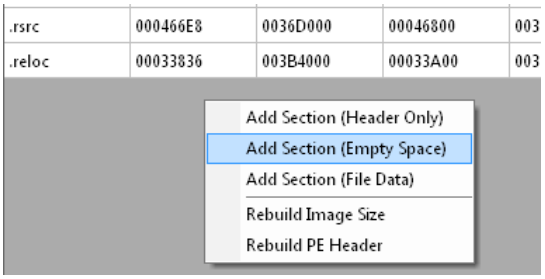
Now we have all the needed information collected, we'll proceed as we planned.

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 end[12], "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.

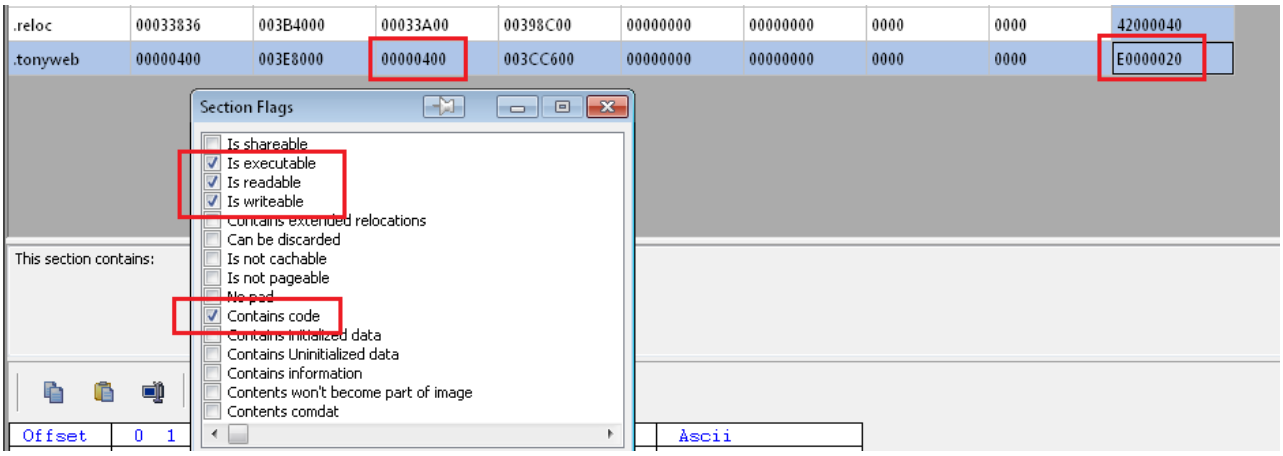
First of all let's add some space for our code-injection. Usually, when there's the need to inject some simple code, I'll use already available code-caves but, when the code to write is complex and/or needs some extensive debugging (like the one we are going to write) I'll prefer adding a completely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

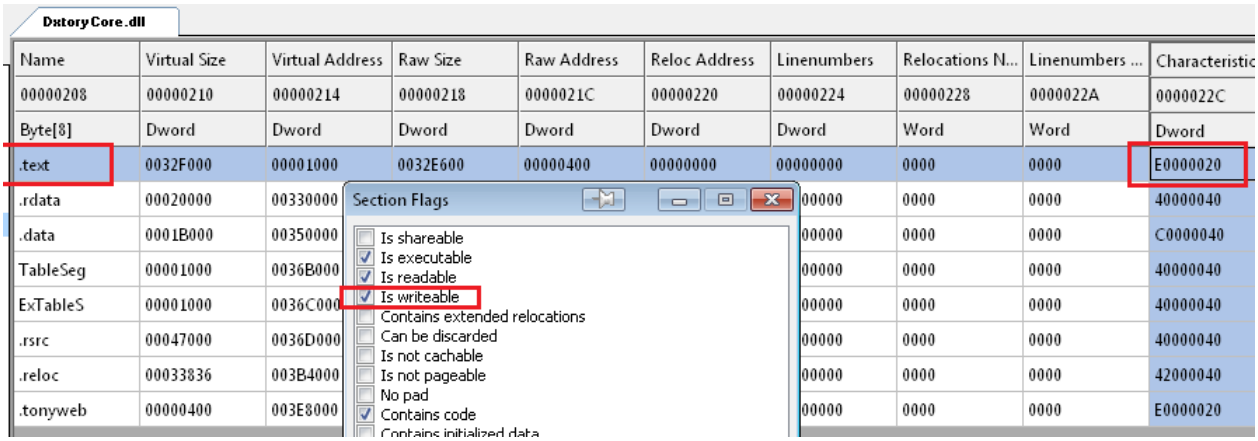


[11] I'm on Windows 7 (x86)

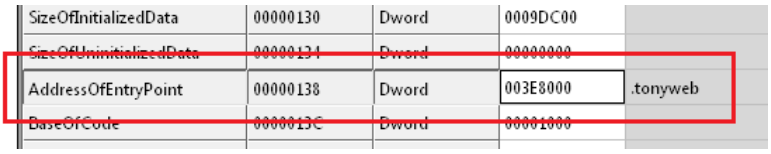
[12] We'll put a redirection also before UnMapViewOfFile call. There, we'll deal with heap destruction before restoring the original code flow.



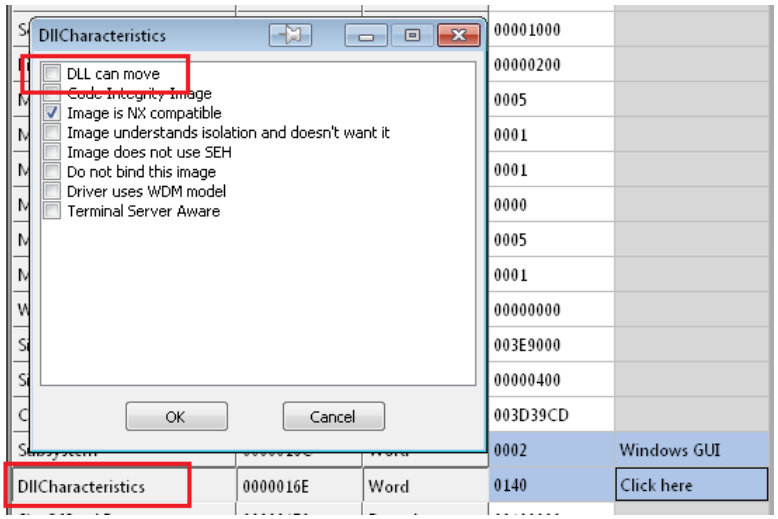
Since we'll have to alter the original code too, let's take advantage of this file modification step and make the .text section writable:



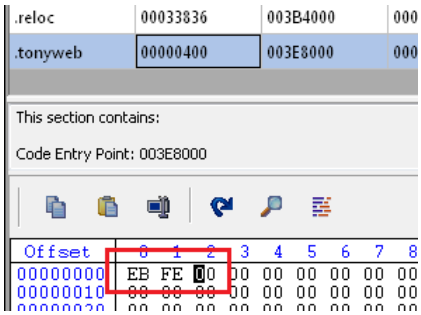
Change also the entry-point to the new section's start address:



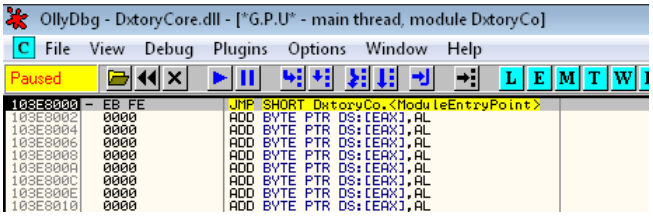
Moreover, in order to avoid some address handling annoyances during code-injection phases (the DLL can be loaded at different addresses among reloads) remove – only for now – the DLL Can Move flag …



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.



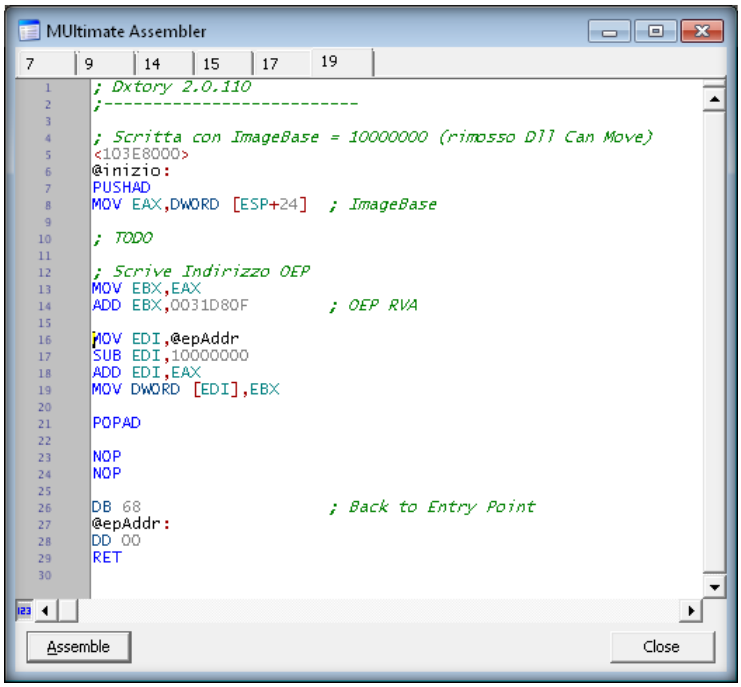
Pheeew! Ok now we are ready. Save the DLL and load it in Olly. Once *Running* is written in Olly status window, click on the pause button, choose OK at the warning about entry-point location (outside the .text section), et voila! Here's our new section:



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

Code Injection: ready, camera … action!

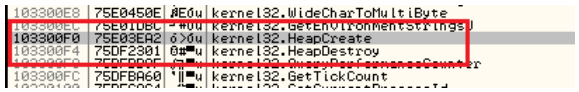
Start writing the code from the beginning of the section (on my system at address 0x103E8000), place the classic PUSHAD / POPAD pair and prepare the jump towards the "original" entry point.



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.

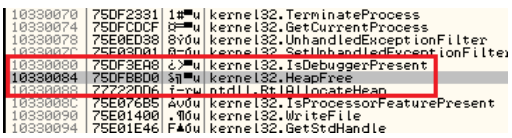
Start with imports, looking for their locations in IAT:



The first two we get are the following:

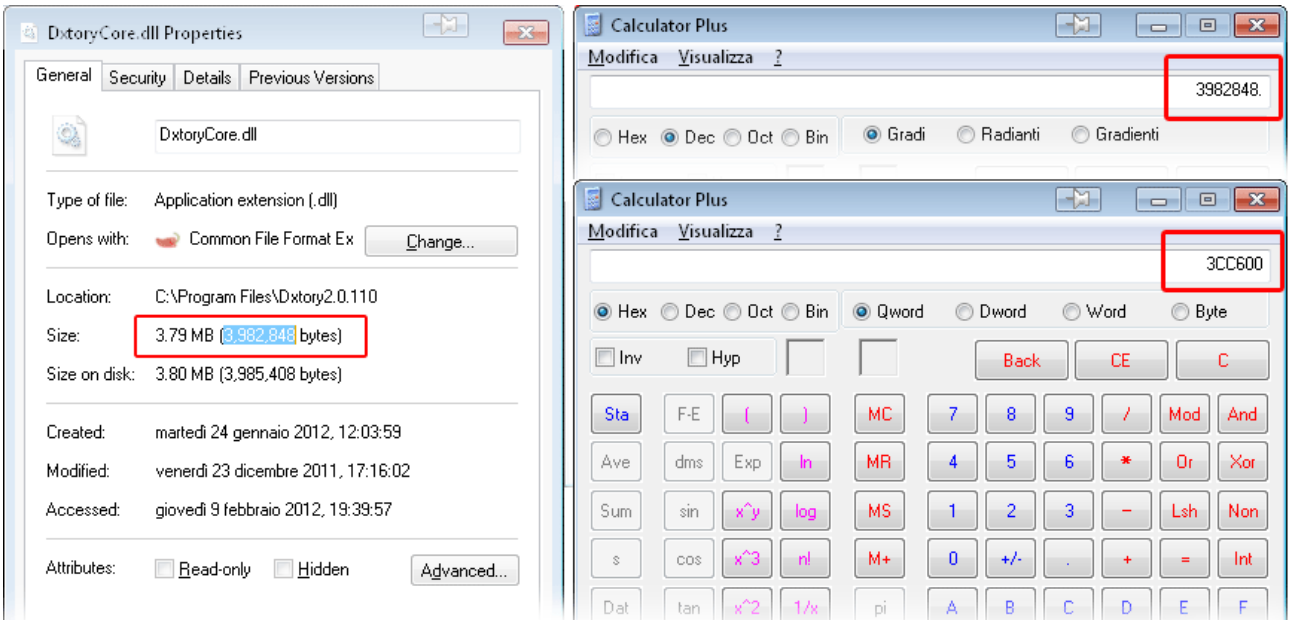
```
HeapCreate,        RVA: 3300F0
HeapDestroy,       RVA: 3300F4
```

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

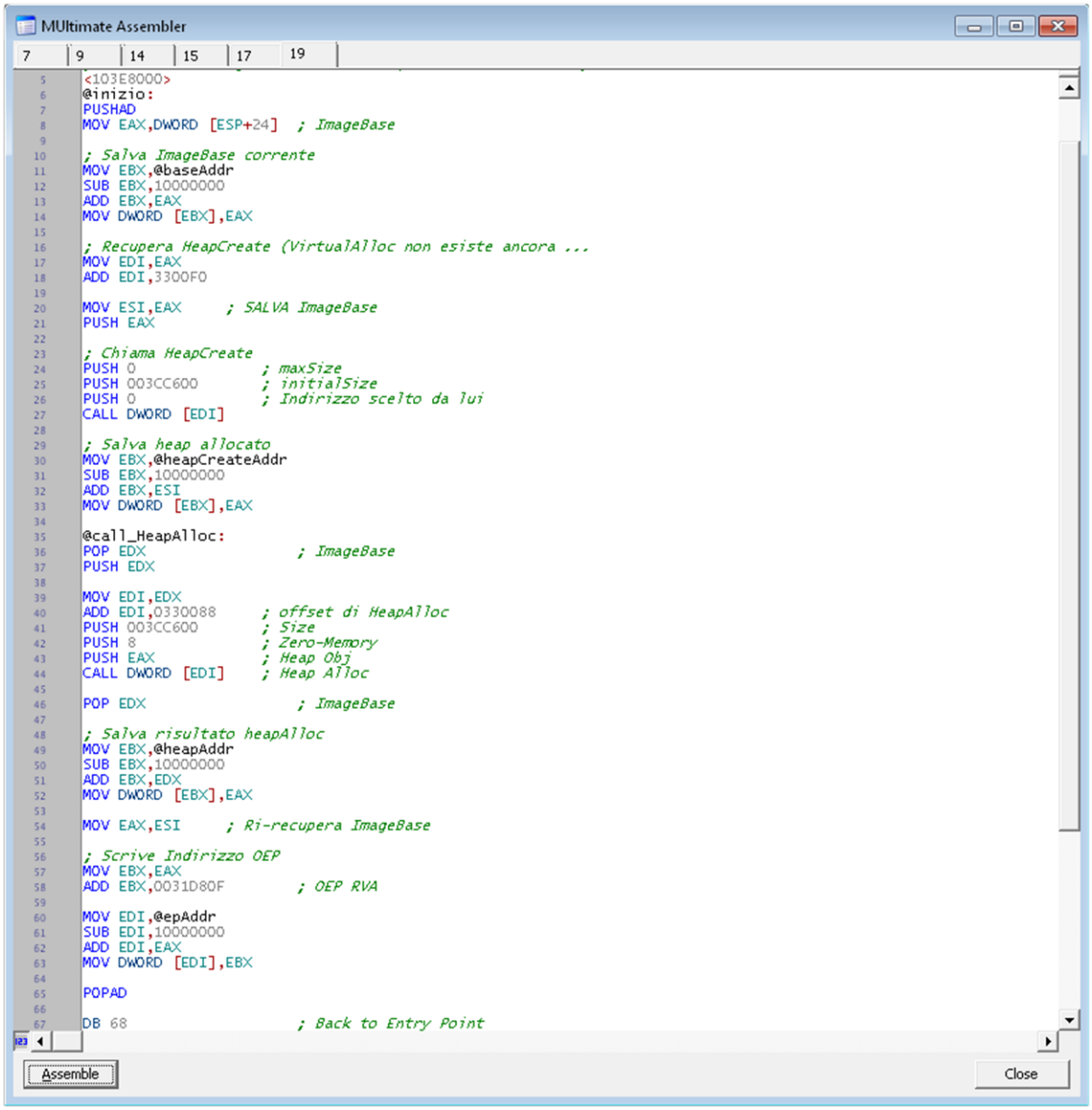



```
HeapFree,          RVA: 330084
RtlAllocateHeap,   RVA: 330088
```

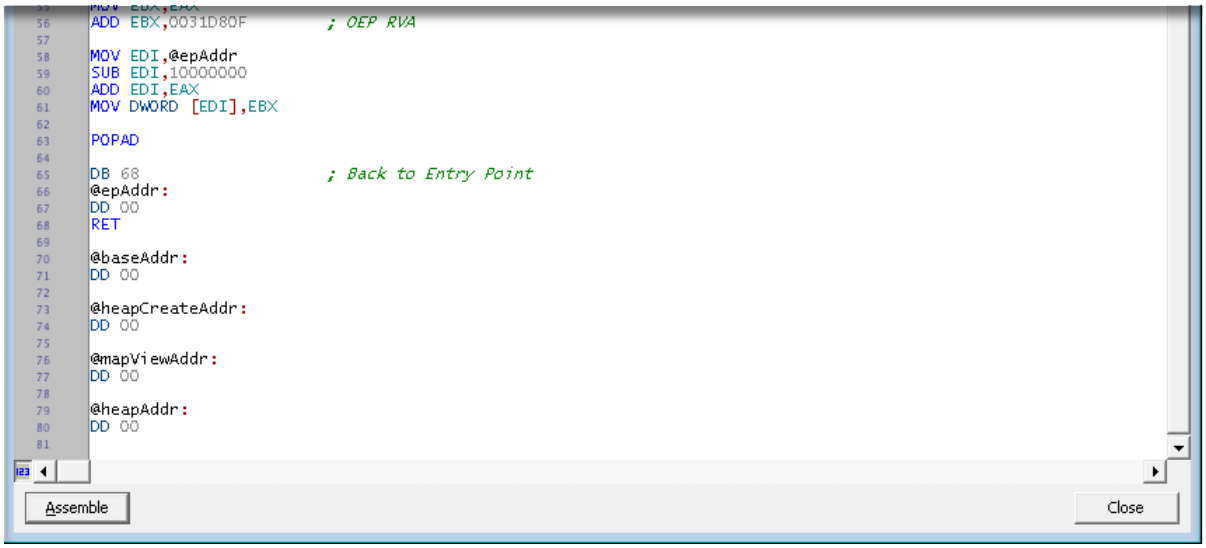
We only miss the file size, but that is a baby game: a couple of clicks plus the Windows calculator are more than enough:



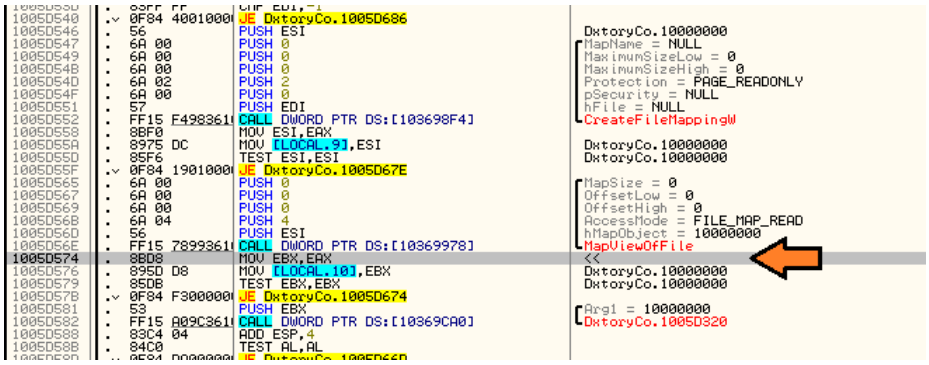
With these information we are ready to write the first part of our injection that deals with heap creation and allocation:



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



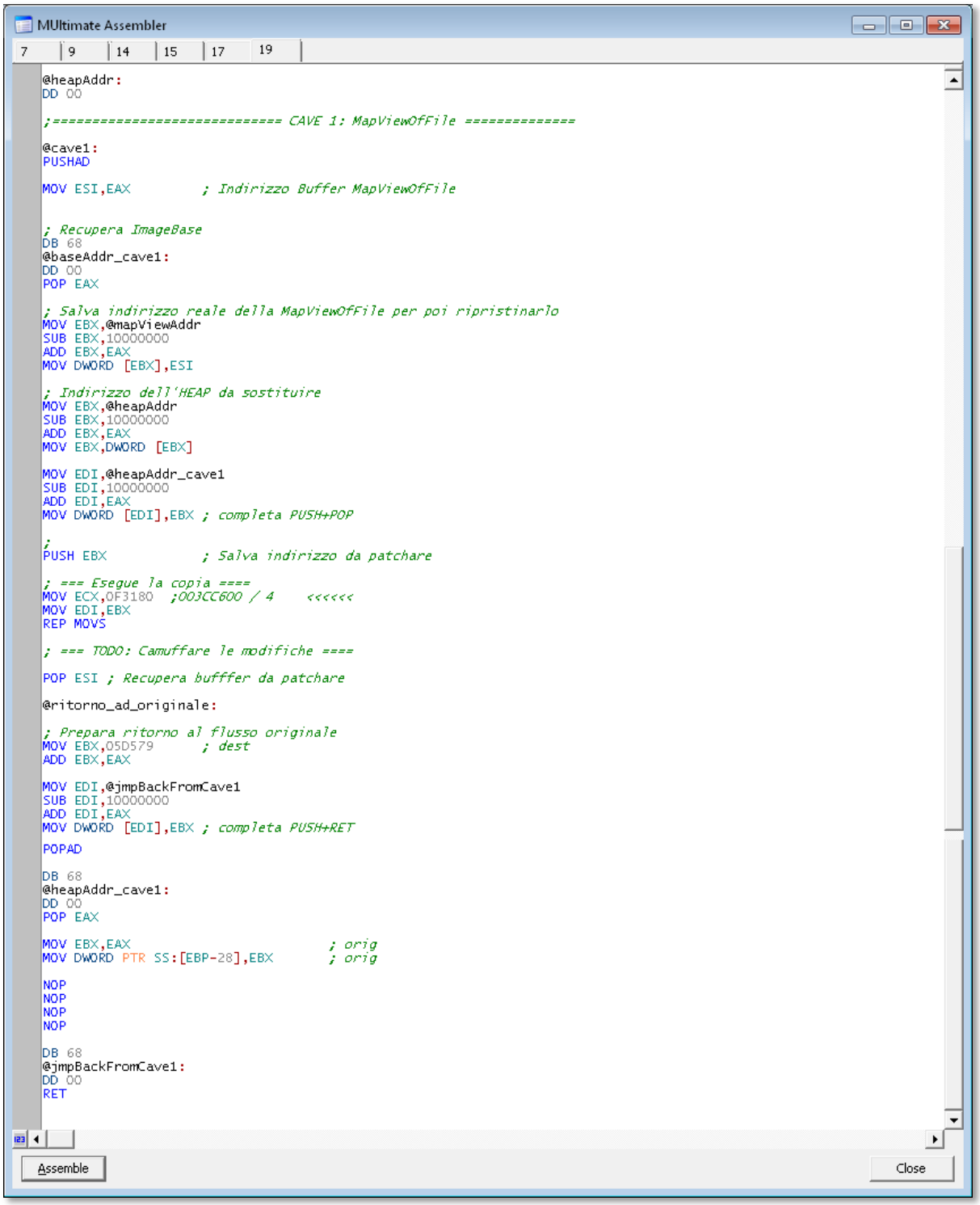
With that done, following our checklist, divert the execution flow after MapViewOfFile call and copy the DLL bytes into our heap. The picture below shows a good point …



… where we can assemble a jump to our cave:

```
MOV EAX,ESI     ; Ri-recupera ImageBase

; ImageBase per la Cave1
MOV EBX,@baseAddr_cave1
SUB EBX,10000000
ADD EBX,EAX
MOV DWORD [EBX],EAX


; Piazza le patch, ================

; Patch 1 (MapViewOfFile)
MOV EBX,EAX             ; jmp src
ADD EBX,05D574
MOV BYTE [EBX],E9

MOV EDI,@cave1          ; jmp dest
SUB EDI,10000000
ADD EDI,EAX

SUB EDI,EBX
ADD EDI,-5
MOV DWORD [EBX+1],EDI       ; Necessita di sezione .text writable

; Scrive Indirizzo OEP
MOV EBX,EAX
ADD EBX,0031D80F        ; OEP RVA
```

Note how we made the current run-time ImageBase available to the cave. Let's add the code for the cave (we'll call it Cave1) immediately below the jump to EP and the saved-addresses' "placeholders":

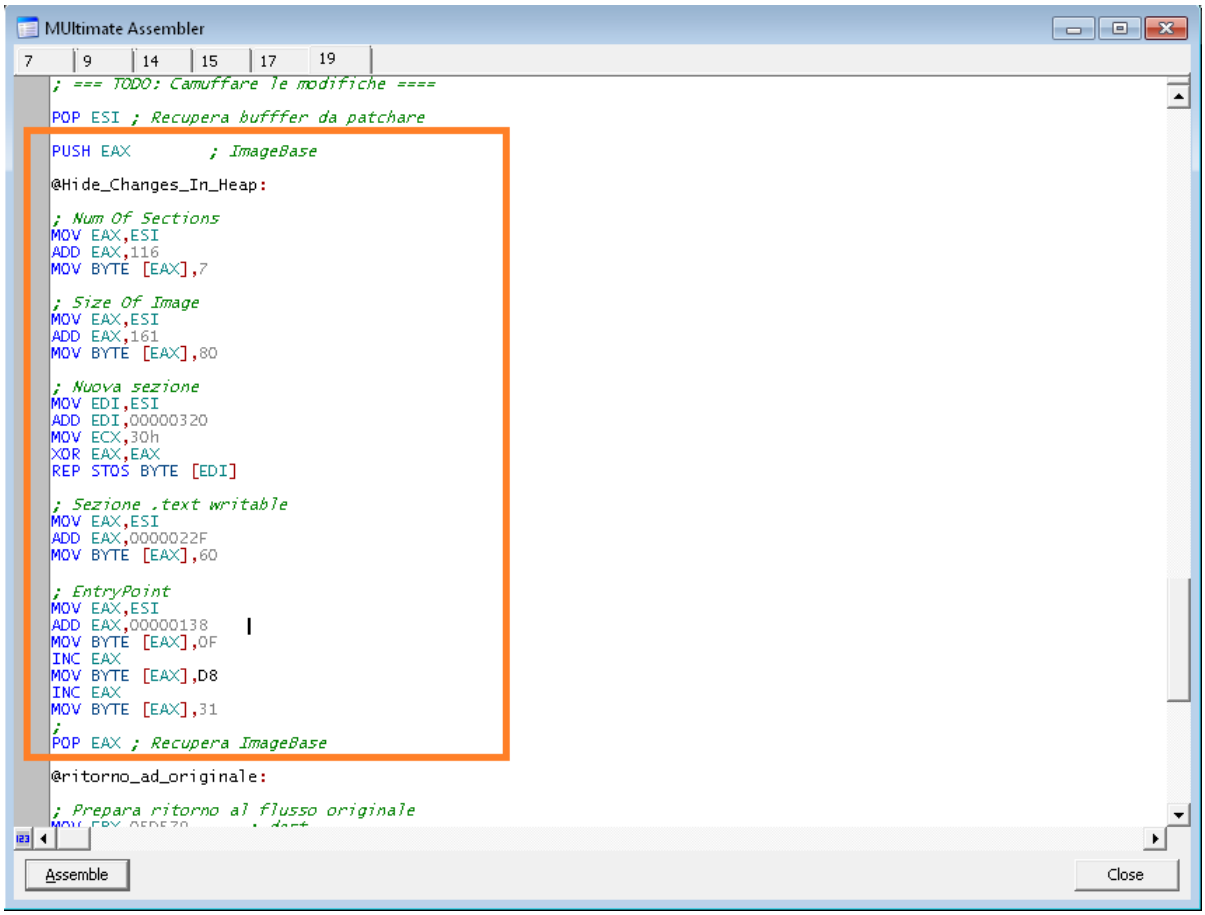


Here we're making a copy of MapViewOfFile returned buffer into our created heap whose address we save into `@heapAddr_cave1`. Before executing the original code (i.e. the instructions we replaced with the jump to this cave and now marked with '; orig' remark) and return to the original code-flow, we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

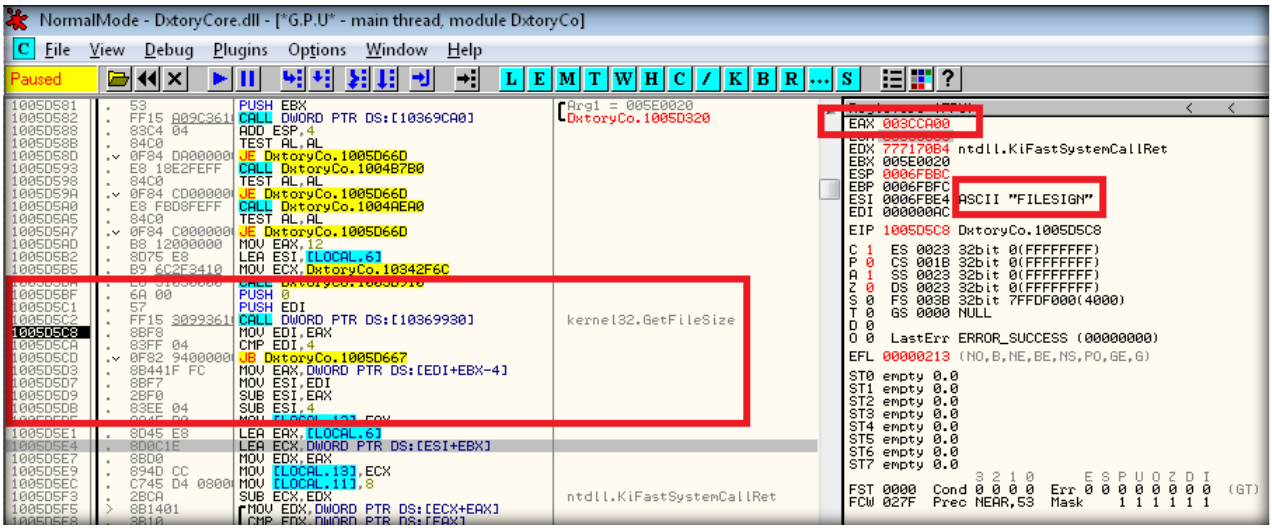
With that done, start hiding our patches. First, let's find out those we absolutely need to cover: the changes we did to the physical file. From command prompt do a comparison (at bytes' level) between original and modified DLL, thanks to the *fc* command.

```
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):

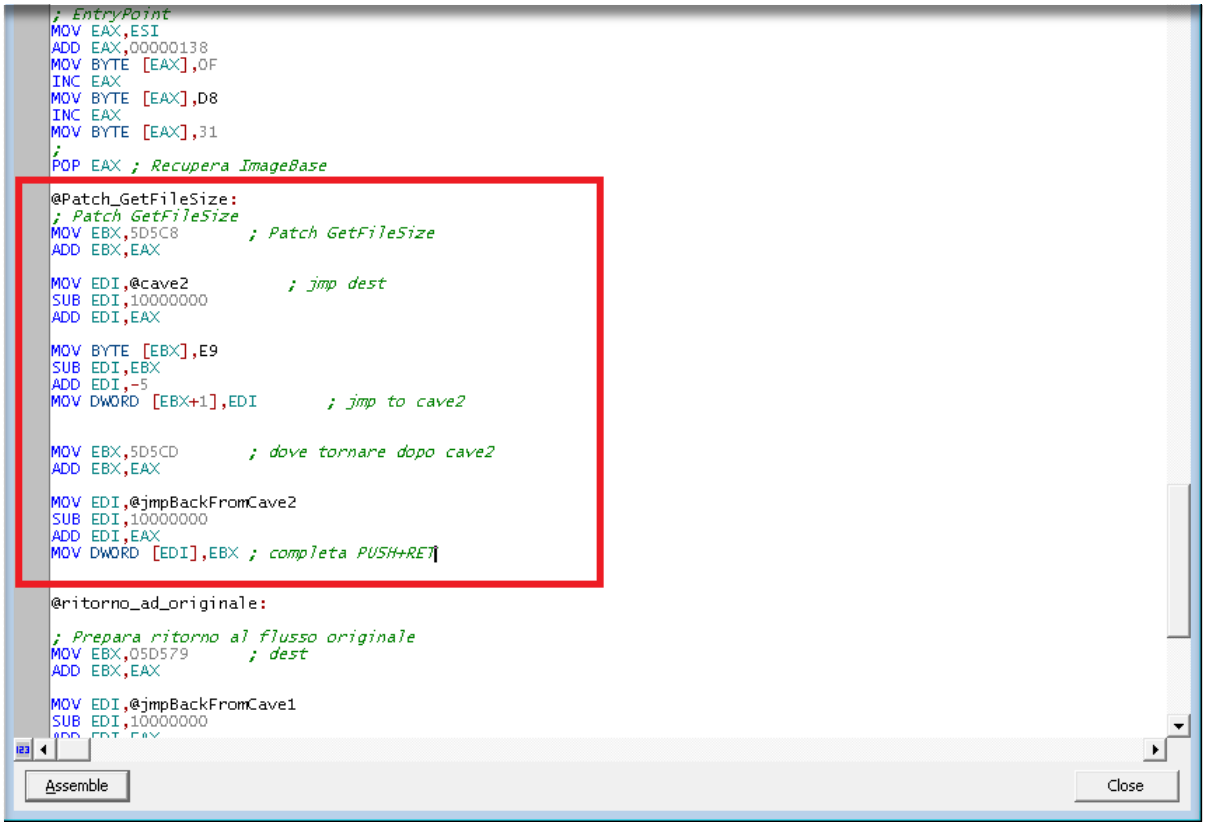


Even with that hiding in place we're not yet ready to run our DLL. It's clear we did a good job: to demonstrate this we can see imports get correctly resolved (in the example the GetFileSize is clearly visible):

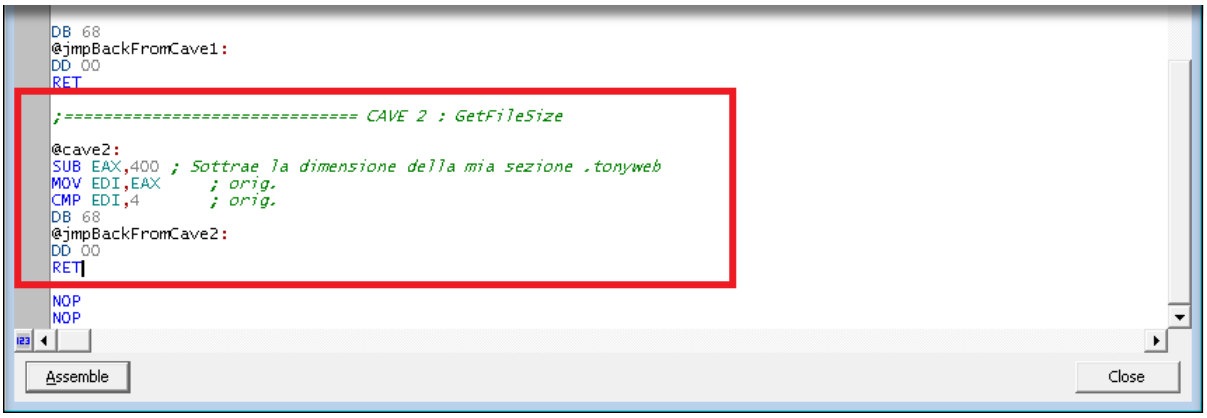


Nevertheless, as highlighted in the picture above, we need to adjust the GetFileSize return value so it will not account for our new .tonyweb section (this is an important step because following routines would use that value to know when to stop reading the buffer; since we replaced the original buffer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

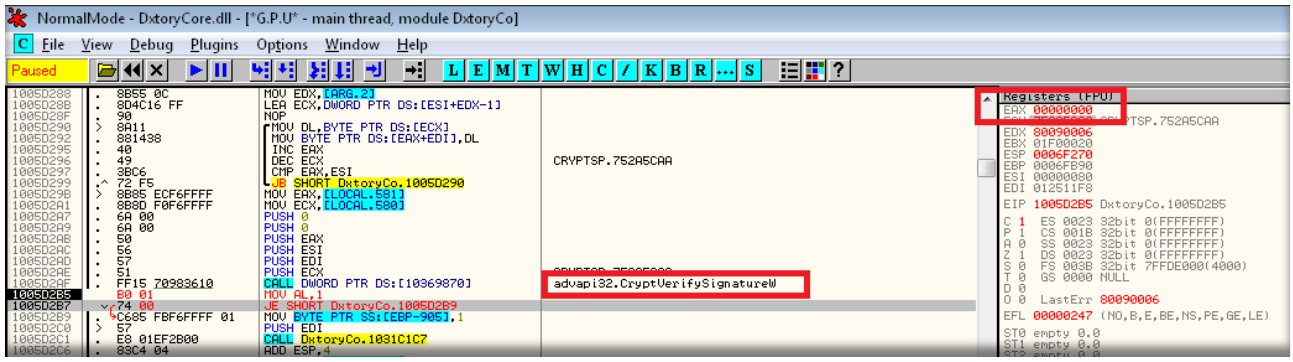
So we put a new cave, and therefore a new jump to it, where we subtract the new section raw size (0x400) from EAX before that value gets passed to any other routine.



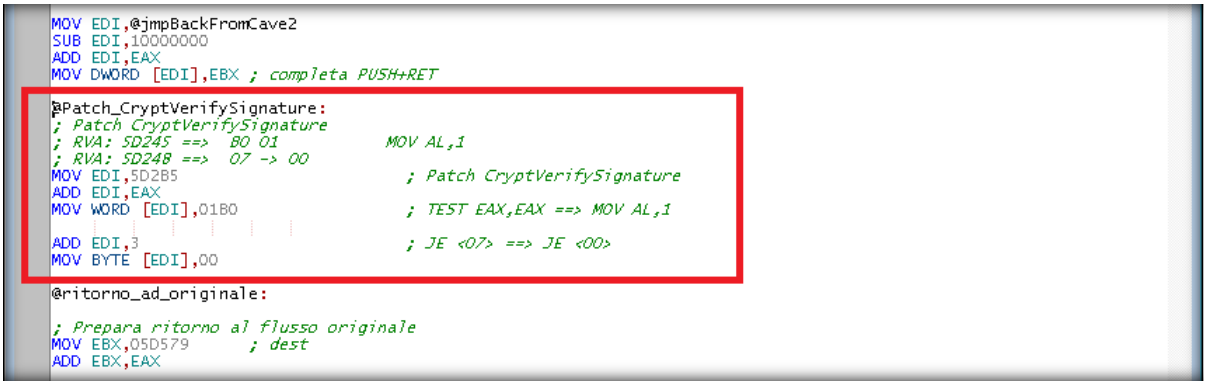
This time our second cave (cave2, we place it after cave1 in the injection) is way easier than then the first one given that it simply needs to avoid the “wrong” size be passed along:



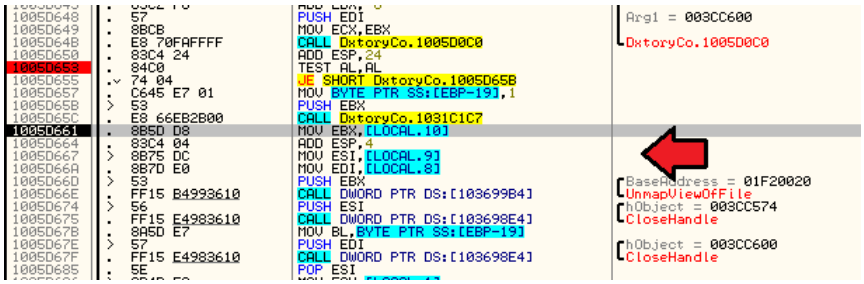
Later, the DLL file gets reopened (do you remember the second CreateFileW?) in order to perform the crypto checks. This time, luckily for us, we need not to "sanitize" the read file content: it's enough to patch the *VerifySignature* outcome. We choose to do that this way:



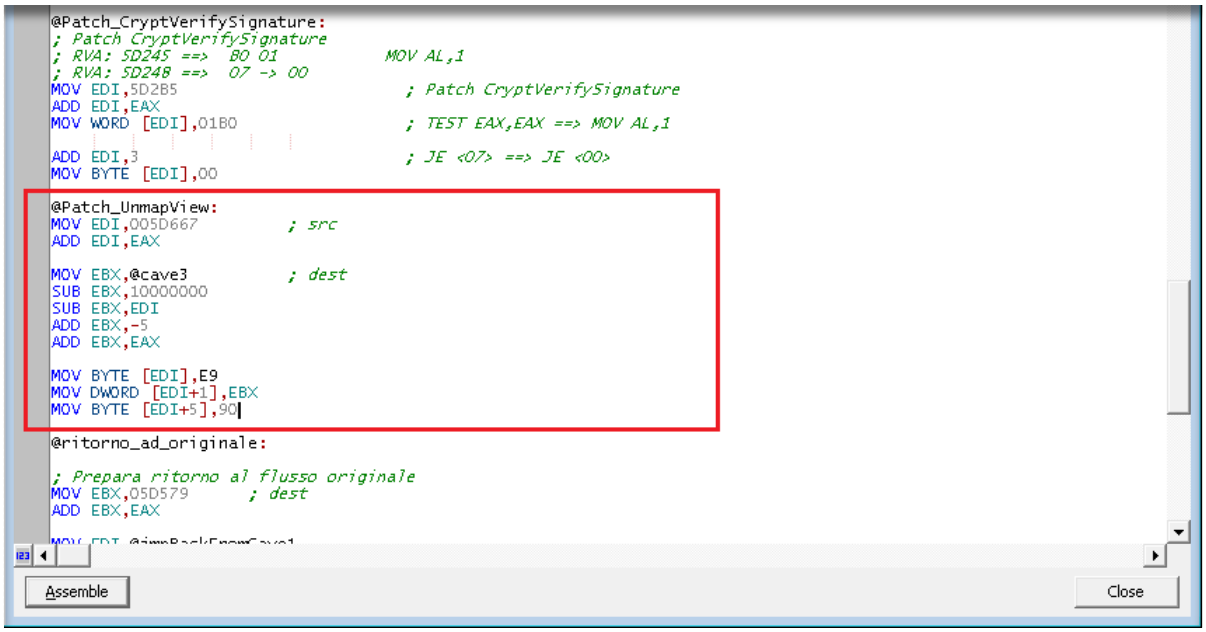
So add the following piece of code to our cave1:



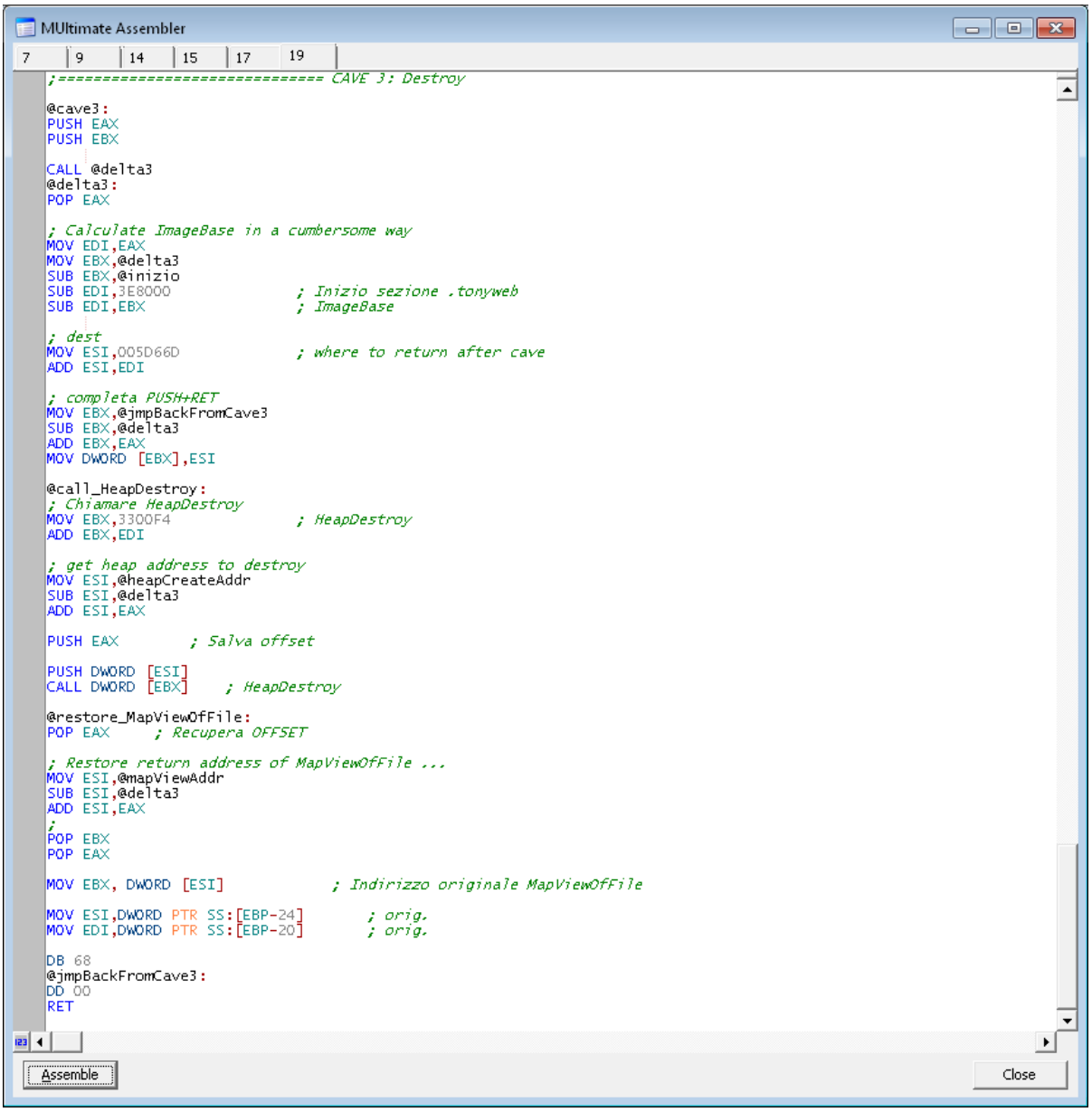
In the excitement don't forget to get rid of the heap: insert another cave (the last one, cave3) where to jump after integrity checks get performed, but before resources get released. There, destroy the heap we created and prepare the return to the original flow, where the program can clean up the file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



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



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



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.*

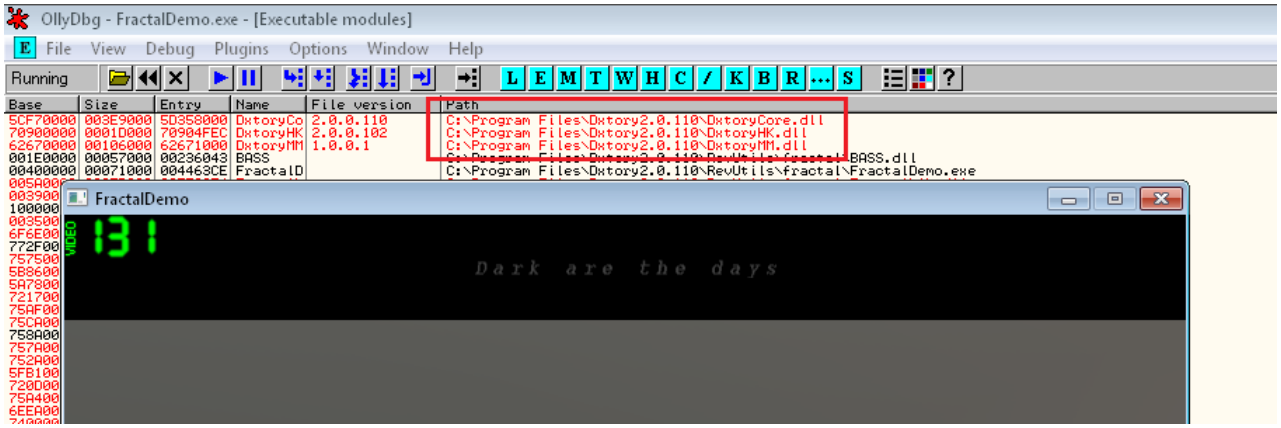
At this point our altered DLL should be considered perfectly uncorrupted by the program and should allow us to take screenshots and movie captures without any problem. Assemble the written code and save the changes, then remember to restore DLL Can Move flag and try to launch the program … it works!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

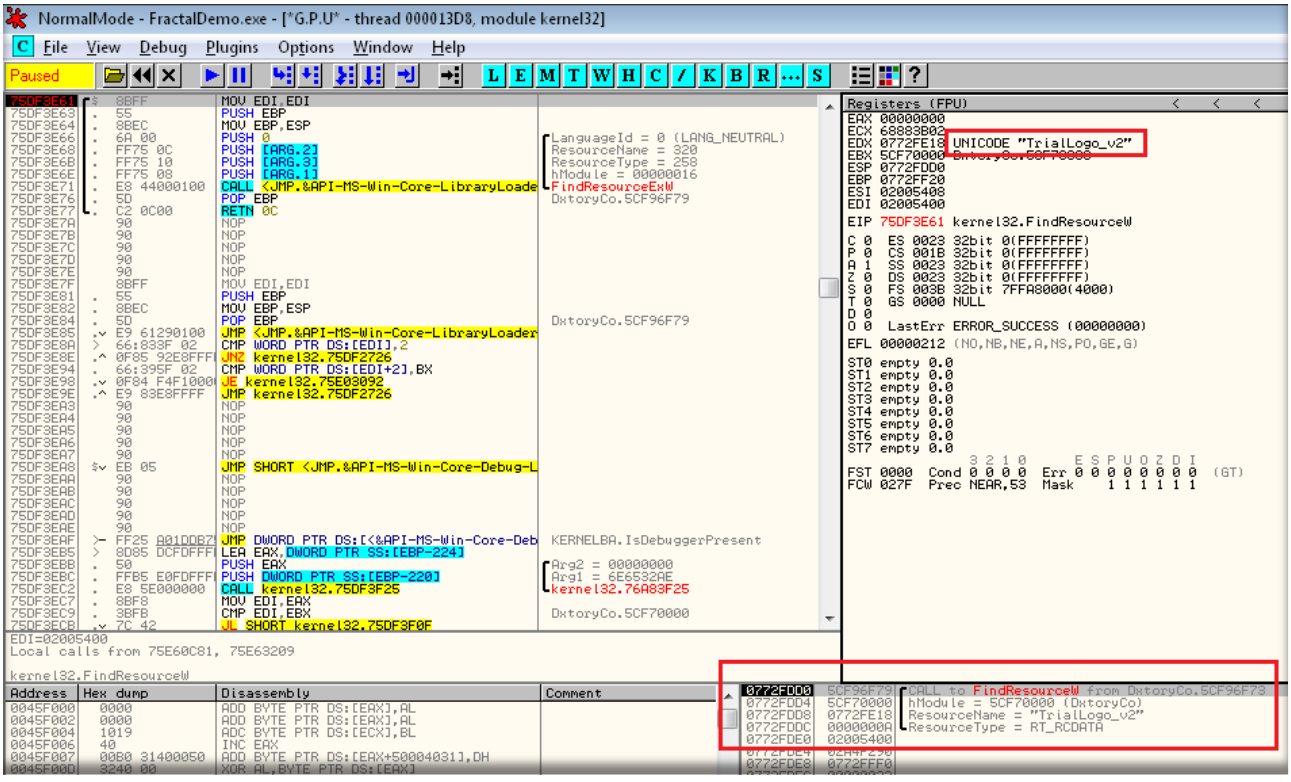
Removing the watermak

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:

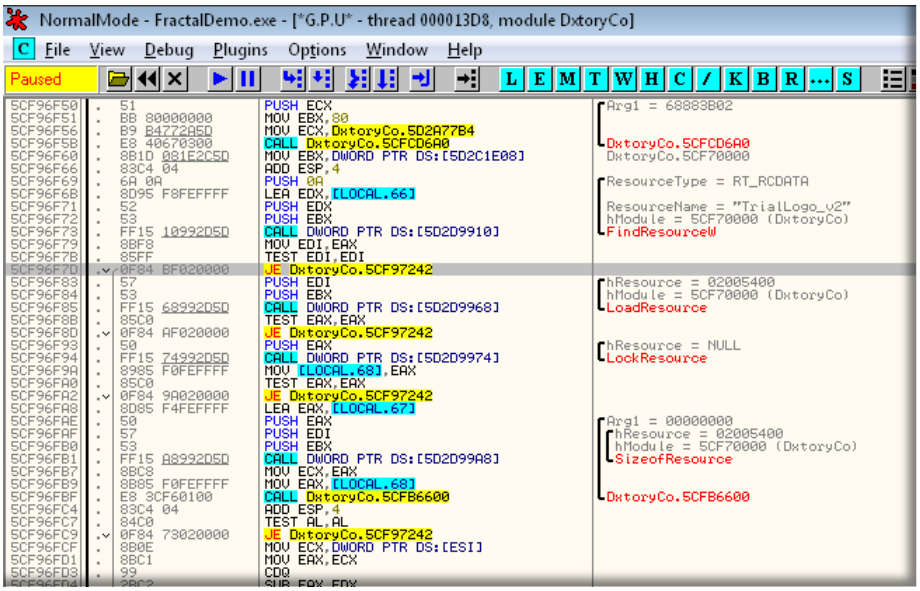


To find when and where the target loads the logo we run FractalDemo, put a BP on FindResourceW and start the video recording (pressing F12, by default).

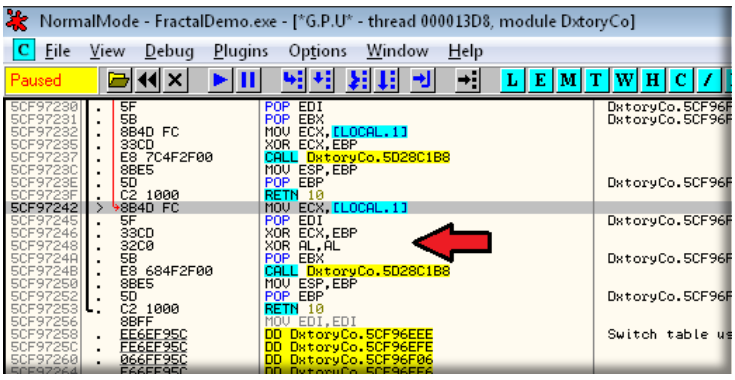
Once we press F12 on keyboard our breakpoint triggers:



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

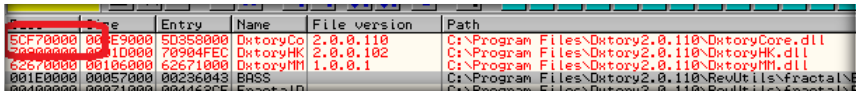


just in the middle of the routine which deals with watermark loading and drawing. ☺ If that FindResourceW fails, the whole routine fails as well and the capture will not start:



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:



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