

Obfuscation and .NET

Dr. Richard Wiener, Editor-in-Chief, JOT, Associate Professor of Computer Science, University of Colorado at Colorado Springs

Assemblies generated under .NET may be decompiled into easily recognized source code that is either identical or similar to the original source code. Individuals or companies deploying .NET generated assemblies that are targeted for client machines may unwittingly be distributing their source code – in most cases an unintended consequence. Since source code is generally considered a valuable intellectual asset, measures should be taken to prevent decompilation into easily recognized source code.

Obfuscator software is aimed at making decompilation into easily recognized source code very difficult. This article examines the issue of obfuscation under .NET and reviews two obfuscator products. The test suite used to evaluate the two obfuscator products are applications taken from the forthcoming book by Richard Wiener entitled *Modern Software Development Using C#.NET* (to be published by Course Technology in late 2005).

The Nature of .NET Assemblies

Assemblies in .NET consist of two major components: metadata and intermediate language code. The reflection capabilities of .NET languages and the metadata features of an assembly that include its classes and associated method signatures, fields, properties and events make it possible to reverse engineer and retrieve this intellectual property. The intermediate language code provides information regarding the details of each method implementation. This enables well constructed decompilers to reveal the details of proprietary algorithms that you would typically not want users to have access to.

To illustrate this, consider the following class, in Listing 1, designed to perform generic sorting using the simple and relatively inefficient selection-sort algorithm.

Listing 2 contains the decompiled source code obtained using the respected and widely used Lutz Roeder's C# .NET Decompiler (<http://www.aisto.com/roeder/dotnet/>). The decompiled language chosen is C#, the same as the originating language.

Listing 3 contains the decompiled source code in Visual Basic .NET obtained from the assembly produced by the original C# source code.

Although not perfect, the Roeder .NET Decompiler serves as a simple language translator since it effectively converts C# source code to Visual Basic source code (something that the author of this article would not dare attempt because of his lack of familiarity with Visual Basic).

Listing 1 – Original C# source code for generic sorting

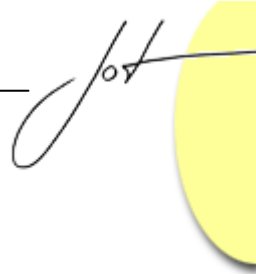
```
using System;
using System.Collections.Generic;

namespace GenericSorting {

    public class Sorting {
        public static void SelectionSort<T>(T [] data,
            int size) where T : IComparable {
            for (int outerIndex = size - 1; outerIndex >= 1;
                outerIndex--) {
                // Find the largest value in data
                T largest = data[0];
                int indexLargest = 0;
                for (int innerIndex = 1;
                    innerIndex <= outerIndex;
                    innerIndex++) {
                    if (data[innerIndex].CompareTo(largest) > 0) {
                        // found value larger than largest
                        largest = data[innerIndex];
                        indexLargest = innerIndex;
                    }
                }
                /* Interchange data[indexLargest] with
                 data[outerIndex];
                */
                T temp = data[indexLargest];
                data[indexLargest] = data[outerIndex];
                data[outerIndex] = temp;
            }
        }
    }

    public class Application {

        public static void Main() {
            double[] myData =
                { 2.5, -1.3, -0.75, 1.5, 2.4, -0.5, 0.75, 1.0 };
            Sorting.SelectionSort<double>(myData,
                myData.Length);
            for (int i = 0; i < myData.Length; i++) {
                Console.Write(myData[i] + " ");
            }
            Console.WriteLine();
        }
    }
}
```



```
        Console.ReadLine();
    }
}
```

Listing 2 – Decompiled source using Lutz Roeder's Reflector Decompiler

```
namespace GenericSorting {

    public class Application {
        // Methods
        public Application();
        public static void Main();
    }

    public class Sorting {
        // Methods
        public Sorting();
        public static void SelectionSort<T>(T[] data, int
size)
            where T: IComparable;
    }
}

public static void SelectionSort<T>(T[] data, int size)
    where T: IComparable {
    for (int num1 = size - 1; num1 >= 1; num1--) {
        T local1 = data[0];
        int num2 = 0;
        for (int num3 = 1; num3 <= num1; num3++) {
            if (data[num3].CompareTo(local1) > 0) {
                local1 = data[num3];
                num2 = num3;
            }
        }
        T local2 = data[num2];
        data[num2] = data[num1];
        data[num1] = local2;
    }
}

public static void Main() {
    double[] numArray1 = new double[8]
        { 2.5, -1.3, -0.75, 1.5, 2.4, -0.5, 0.75, 1 } ;
    Sorting.SelectionSort<double>(numArray1,
numArray1.Length);
    for (int num1 = 0; num1 < numArray1.Length; num1++) {
        Console.Write(numArray1[num1] + " ");
    }
    Console.WriteLine();
}
```

```

    }
    Console.ReadLine();
}

```

Listing 3 – Decompilation into Visual Basic .NET

```

Public Shared Sub SelectionSort(Of T As IComparable)(ByVal
    data As T(),
    ByVal size As Integer)
    Dim num1 As Integer = (size - 1)
    Do While (num1 >= 1)
        Dim local1 As T = data(0)
        Dim num2 As Integer = 0
        Dim num3 As Integer = 1
        Do While (num3 <= num1)
            If (data(num3).CompareTo(local1) > 0) Then
                local1 = data(num3)
                num2 = num3
            End If
            num3 += 1
        Loop
        Dim local2 As T = data(num2)
        data(num2) = data(num1)
        data(num1) = local2
        num1 -= 1
    Loop
End Sub

```

Although the decompiled C# source code is not identical to the original it is close enough to reveal all its essential details. The namespace, class names and method names and their signatures were decompiled perfectly. These are the features that define the architecture of the software system. The details of method *SelectionSort* reveal the algorithm used.

For users who prefer Delphi Pascal, the Lutz Roeder's decompiled code in Listing 4 provides the details of the generic *SelectionSort* procedure in Delphi Pascal.

Listing 4 – Decompiled SelectionSort in Delphi Pascal

```

procedure Sorting.SelectionSort<T>(data: T[]; size: Integer)
    where T:
    IComparable;
begin
    num1 := (size - 1);
    while ((num1 >= 1)) do
    begin
        local1 := data[0];
        num2 := 0;
        num3 := 1;
        while ((num3 <= num1)) do
        begin

```



```
        if (data[num3].CompareTo(local1) > 0) then
        begin
            local1 := data[num3];
            num2 := num3
        end;
        inc(num3)
    end;
    local2 := data[num2];
    data[num2] := data[num1];
    data[num1] := local2;
    dec(num1)
end;
end;
```

Moderate size applications containing dozens of classes and thousands of lines of source code have been decompiled with equal success using the Roeder's .NET decompiler.

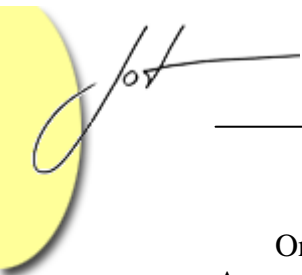
As a teacher of computer science, I often post my solutions (.NET assembly) on my university website for major projects that are assigned to my students. These solutions provide an additional specification to the students about how the system they are to design and implement should function. A major part of these projects is defining the architecture of the solution (class names, their features and interrelationships). The ability for students to decompile my posted assembly and reverse engineer not only the architecture of my solution but even its fine details requires that I run my assembly through a competent obfuscator before posting it to my website. This obfuscator should make the architecture and details of my solution very difficult to decipher.

Clearly the same need exists for the commercial deployment of .NET assemblies.

The Nature of Obfuscation

An ideal obfuscator mangles the large features (namespaces, class names, method signatures and fields) and small features (method details and in particular string values defined as fields and within a method) of your assembly without changing its functionality. The more aggressive the mangling, the more likely that the obfuscated assembly will not run the same way as the original assembly. It is essential that an obfuscator keep the functionality of the software totally intact while making the original source code unrecognizable if the obfuscated assembly is decompiled.

There are several well known problem areas that a well designed obfuscator must allow the user to deal with. When runtime type identification is used in an application to determine whether an object is of a particular type, the class name of the type being sought is used. If the obfuscator has mangled this class name, the dynamic type identification will fail in the obfuscated assembly. The user of the obfuscator must be given the option of selecting features that are not to be mangled. These include class, field and method names. The same issue exists when dynamic class loading is performed within the assembly or serialization or remoting is used.



One of the side-effects of obfuscation is the difficulty of debugging obfuscated code. An exception that is generated and reported by a user will typically include mangled method and class names making it almost impossible to identify the root cause. Providing a clearly labeled map file in the obfuscation tool is essential to the user in interpreting debugger output from the obfuscated assembly.

Hackers often search deployed assemblies for strings that contain keywords such as “Password” or “Enter password”. By locating such strings, hackers attempt to circumvent the password protection embedded in the product that they are hacking. Some obfuscators provide the option of string encryption. Although this may introduce a small performance penalty because of the need to decrypt strings at runtime, the overhead associated with such string encryption and decryption is often negligible.

Some obfuscator products include the ability to statically analyze the application and determine the parts that are not being used. This includes unused types, unused methods, and unused fields. This could be of great benefit if memory footprint is a concern.

Some obfuscators provide control of flow obfuscation. Control flow obfuscation typically introduces false conditional statements and other misleading constructs in order to confuse decompilers. Some obfuscators destroy the code patterns that decompilers use to recreate source code. The trick is to confuse the decompiler without changing the functionality of the obfuscated assembly.

Incremental obfuscation allows the developer to make changes to the original sources after releasing an obfuscated assembly and then provide a patch to the user that reflects the changes to the original application while preserving the name-mapping used in the original release. In order to accomplish this, a map file must be saved and later used to ensure that the renaming is preserved when making changes and re-releasing the obfuscated assembly. Some obfuscator products support this useful capability.

Finally, some obfuscators enable the user to embed watermarks such as user names and registration codes into the internal binary structures within the assembly. Watermarking can assist in tracking distribution of the product on a per-executable basis if it is illicitly distributed or obtained.

Several well known obfuscator products were sought and obtained for this review. Only two survived the rigorous tests that each obfuscator was subjected to. The obfuscators that failed generally produced assemblies that would not run or did not provide sufficient features to be of use for deploying commercial obfuscated assemblies.

Only .NET 2005 (beta 1) assemblies were used in all the tests.



The two products that survived these tests and will be the subject of this review are:

1. Dotsfuscator Professional Edition, Version 3 (RC)
<http://www.preemptive.com/>
PreEmptive Solutions
26250 Euclid Avenue
Suite 503
Cleveland, Ohio 44132
Voice: 216.732.5895
General Email: information@preemptive.com
2. (2) Salamander, RemoteSoft .NET Explorer
<http://www.remotesoft.com/>
Dr. Huihong Luo
Tel: 1-510-579-2752
Sales information:
sales@remotesoft.com

Dotfuscator – Professional Edition, Version 3 (RC)

This is an outstanding product in every respect. It is obvious why Microsoft has chosen to bundle this product (a light version) with .NET 2003 and more recently .NET 2005 (beta). Assemblies can be obfuscated within Visual Studio using the Tools menu or using the stand-alone GUI version. Both provide equivalent functionality. I used the stand-alone GUI version for most of my testing.

A detailed on-line user's guide provides detailed information about the use of the product. This user's guide is well written and useful. The only item that I found missing from this user's guide was information about how to detect the watermark that the user may optionally embed in the obfuscated assembly. An e-mail to Preemptive Solutions got me an answer within 10 minutes.

Dotfuscator is a full-feature obfuscator product. It supports all of the facilities described in the previous section. Of particular interest is the "overload induction engine". Using consecutive letters of the alphabet, Dotfuscator attempts to legally overload these letters when transforming identifier names from the original to obfuscated assembly. Clearly this obfuscated assembly deserves a grade of "A"!

Consider its work on obfuscating the assembly produced by Listing and observe the "a's". A portion of the decompiled obfuscated assembly is given in Listing 5.

Listing 5 – Decompiled Dotfuscated Assembly from Listing 1 (Generic Sorting)

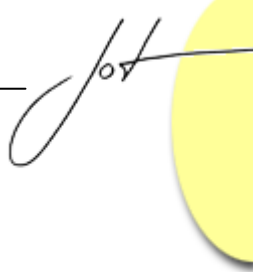
```

public static void a()
{
    int num2 = 11;
    double[] numArray1 = new double[8] { 2.5, -1.3, -0.75,
1.5, 2.4,
                                -0.5, 0.75, 1 }
;
    c.a<double>(numArray1, numArray1.Length);
    int num1 = 0;
    while (true)
    {
        if (num1 >= numArray1.Length)
        {
            break;
        }
        Console.Write(numArray1[num1] + a("\ucceb\uceed",
num2));
        num1++;
    }
    Console.WriteLine();
    Console.ReadLine();
}

internal static b.a a = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x40, 0xCD,
0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xF4, 0xBF,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xE8, 0xBF, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xF8, 0x3F,
    0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x33, 0x03, 0x40, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xE0, 0xBF,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xE8, 0x3F, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x3F
};

public static void a<a>(a[] A_0, int A_1) where a: IComparable
{
    int num1 = A_1 - 1;
Label_004D:
    if (num1 < 1)
    {
        return;
    }
    a local1 = A_0[0];
    int num2 = 0;
    int num3 = 1;
    while (true)
    {
        if (num3 <= num1)
        {
            if (A_0[num3].CompareTo(local1) > 0)

```

```
        {
            local1 = A_0[num3];
            num2 = num3;
        }
    }
else
{
    a local2 = A_0[num2];
    A_0[num2] = A_0[num1];
    A_0[num1] = local2;
    num1--;
    goto Label_004D;
}
num3++;
}
}
```

Of particular benefit is the Dotfuscator option to encrypt string values within an assembly.

Consider the simple C# application given in Listing 6 that prompts the user for a password. The password itself is stored as a field of class *PassWordApplication* (not a good practice but done here to dramatize the value of string encryption).

Listing 6 – C# Application that Prompts for Password

```
using System;

namespace PasswordApp {

    public class PassWordApplication {

        // Fields
        private String password = "Hi, my name is Paul.";

        public static void Main() {
            PassWordApplication app =
                new PassWordApplication();
            Console.WriteLine("Enter password: ");
            String pass = Console.ReadLine();
            if (pass.Equals(app.password)) {
                Console.WriteLine("Correct password.");
            } else {
                Console.WriteLine("Incorrect password.");
            }
            Console.ReadLine();
        }
    }
}
```

Listing 7 presents the decompiled source listing, again using Reflector, after the source in Listing 6 is obfuscated without string encryption.

Listing 7 – Decompiled Version of Listing 6

```
public a() {
    this.a = "Hi, my name is Paul.";
}

public static void a() {
    a a1 = new a();
    Console.WriteLine("Enter password: ");
    string text1 = Console.ReadLine();
    if (!text1.Equals(a1.a))
    {
        Console.WriteLine("Incorrect password.");
    }
    else
    {
        Console.WriteLine("Correct password.");
    }
    Console.ReadLine();
}
```

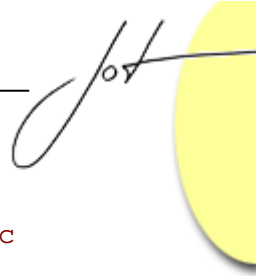
Clearly the decompiled source code informs the user not only about the section of code that requests a password but in this case the password itself. There would not be much value in having password protection in this case or obfuscation.

As mentioned before, Dotfuscator includes an option for encrypting strings within the assembly. After exercising this option, the new decompiled source code is shown in Listing 8.

Listing 8 – Decompiled Version of Listing 6 Using String Encryption

```
public a() {
    int num1 = 5;
    this.a =
a("\u006ad\u009eb1\u0094b3\u00c1b7\u009ab9\u00d2bb\u00adb\u00a7c1\u00e4c3\u00afc
5\u00bbc7\u00eac9\u009ccb\u00afc\u00a5cf\u00bed1\u00fad3", num1);
}

public static void a()
{
    int num1 = 13;
    a a1 = new a();
}
```



```

Console.WriteLine(a("\uf3b5\ud6b7\uceb9\uccbd\ue0bf\ub2c1\ua5c
3\ub5c5\ubbc7\ubdc9\ua3cb\ubccd\ub4cf\ue8d1\uf4d3", num1));
    string text1 = Console.ReadLine();
    if (!text1.Equals(a1.a)) {
Console.WriteLine(a("\uffb5\ud6b7\ud3bb\uccbd\ub2bf\ua7c1\ua7c
3\ub2c5\ue8c7\ubac9\uadcb\ubdcd\ua3cf\ua5d1\ubbd3\ua4d5\ubcd7\
uf4d9", num1));
    }
    else
    {

Console.WriteLine(a("\uf5b5\ud7b7\uc8b9\ucebb\ua3bf\ub6c1\ue4c
3\ub6c5\ua9c7\ub9c9\ubfcb\ub9cd\ubfcf\ua0d1\ub0d3\uf8d5",
num1));
    }
    Console.ReadLine();
}

```

Clearly the string encrypted version is much better protected. It is not at all obvious that the decompiled code relates in any way to password solicitation. It should be clear that the string encryption feature of Dotfuscator is invaluable and indispensable.

Preserving features prior to obfuscation is of fundamental importance. As indicated earlier, this is essential before obfuscating assemblies that perform dynamic class loading or utilize runtime type identification. There are other motivations for preserving features (the use of serialization and remoting). Dotfuscator provides an outstanding graphical user interface that allows the user to select for preservation a class name or individual fields or methods within a class. Each of these features is presented in a tree that shows the fine-grained features of the class (class name, method names and field names). Associated with each feature is a checkbox that when selected prevents that particular feature's name from being mangled. After extensive testing I am pleased to report that this important feature works exactly as advertised.

As a useful security measure, Dotfuscator allows the user to embed watermarking within the obfuscated assembly. Using a command-line utility, *Premark*, the watermark, if present, can be retrieved from the obfuscated assembly.

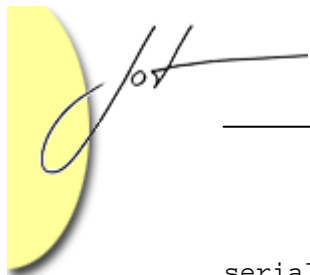
Dotfuscator produces map files in XML format. A sample map file, in this case associated with the generic sorting application presented in Listing 1, is shown in Listing 9.

Listing 9 – Map file associated with Obfuscation of Listing 1

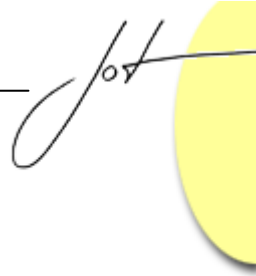
```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE dotfuscatorMap SYSTEM
"http://www.preemptive.com/dotfuscator/dtd/dotfuscatorMap_v1.1.dtd">
<dotfuscatorMap version="1.1">
  <header>
    <timestamp>2005-04-05T09:32:20</timestamp>

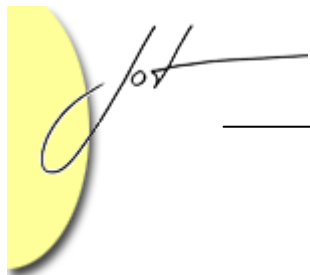
```



```
<product version="3.0.1920.16311" user="Dr. Richard Wiener"
serial="(removed)">Dotfuscator(tm) Professional Edition</product>
</header>
<mapping>
  <module>
    <name>Sorting.exe</name>
    <type>
      <name>&lt;PrivateImplementationDetails&gt;{F458E8E5-
6075-4944-A3B6-2E98BF906274}</name>
      <newname>b</newname>
      <methodlist />
      <fieldlist>
        <field>
          <signature>&lt;PrivateImplementationDetails&gt;{F458E8E5-6075-4944-A3B6-
2E98BF906274}/__StaticArrayInitTypeSize=64</signature>
            <name>$$method0x6000007-1</name>
            <newname>a</newname>
          </field>
        </fieldlist>
      </type>
      <type>
        <name>&lt;PrivateImplementationDetails&gt;{F458E8E5-
6075-4944-A3B6-2E98BF906274}/__StaticArrayInitTypeSize=64</name>
        <newname>b/a</newname>
        <methodlist />
        <fieldlist />
      </type>
      <type>
        <name>GenericSorting.Application</name>
        <newname>a</newname>
        <methodlist>
          <method>
            <signature>void()</signature>
            <name>.ctor</name>
          </method>
          <method>
            <signature>void()</signature>
            <name>Main</name>
            <newname>a</newname>
          </method>
        </methodlist>
        <fieldlist />
      </type>
      <type>
        <name>GenericSorting.Sorting</name>
        <newname>c</newname>
        <methodlist>
          <method>
            <signature>void()</signature>
            <name>.ctor</name>
          </method>
          <method>
            <arity>1</arity>
```



```
        <signature>void(!!0[], int32)</signature>
        <name>SelectionSort</name>
        <newname>a</newname>
    </method>
</methodlist>
<fieldlist />
</type>
<type>
    <name>Sorting.Properties.Resources</name>
    <newname>e</newname>
    <methodlist>
        <method>
            <signature>void()</signature>
            <name>.ctor</name>
        </method>
        <method>
<signature>System.Globalization.CultureInfo()</signature>
        <name>get_Culture</name>
        <newname>a</newname>
    </method>
    <method>
<signature>System.Resources.ResourceManager()</signature>
        <name>get_ResourceManager</name>
        <newname>b</newname>
    </method>
    <method>
<signature>void(System.Globalization.CultureInfo)</signature>
        <name>set_Culture</name>
        <newname>a</newname>
    </method>
</methodlist>
<fieldlist>
    <field>
<signature>System.Globalization.CultureInfo</signature>
        <name>_resCulture</name>
        <newname>b</newname>
    </field>
    <field>
<signature>System.Resources.ResourceManager</signature>
        <name>_resMgr</name>
        <newname>a</newname>
    </field>
</fieldlist>
</type>
<type>
    <name>Sorting.Properties.Settings</name>
    <newname>d</newname>
    <methodlist>
        <method>
```



```
        <signature>void()</signature>
        <name>.ctor</name>
    </method>
    <method>
        <signature>void()</signature>
        <name>.ctor</name>
    </method>
    <method>

<signature>Sorting.Properties.Settings()</signature>
    <name>get_Value</name>
    <newname>a</newname>
    </method>
</methodlist>
<fieldlist>
    <field>
        <signature>object</signature>
        <name>m_SyncObject</name>
        <newname>b</newname>
    </field>
    <field>

<signature>Sorting.Properties.Settings</signature>
    <name>m_Value</name>
    <newname>a</newname>
    </field>
</fieldlist>
</type>
</module>
</mapping>
<statistics>
    <statisticline>
        <description>Total Classes</description>
        <statistic>6</statistic>
    </statisticline>
    <statisticline>
        <description>Total Methods</description>
        <statistic>11</statistic>
    </statisticline>
    <statisticline>
        <description>Total Fields</description>
        <statistic>5</statistic>
    </statisticline>
    <statisticline>
        <description>Total Classes Renamed</description>
        <statistic>6</statistic>
        <statistic>100.00 %</statistic>
    </statisticline>
    <statisticline>
        <description>Total Methods Renamed</description>
        <statistic>6</statistic>
        <statistic>54.55 %</statistic>
    </statisticline>
</statisticline>
</statisticline>
```



```
<description>Total Fields Renamed</description>
<statistic>5</statistic>
<statistic>100.00 %</statistic>
</statisticline>
<statisticline>
  <description>Methods Renamed to 'a'</description>
  <statistic>5</statistic>
  <statistic>45.45 %</statistic>
</statisticline>
<statisticline>
  <description>Methods Renamed to 'b'</description>
  <statistic>1</statistic>
  <statistic>9.09 %</statistic>
</statisticline>
</statistics>
</dotfuscatorMap>
```

After spending several days testing and using Preemptive Solutions Dotfuscator (<http://www.preemptive.com/>), I can report that this product presents an outstanding role-model in product design and implementation. Its user-interface is simple and clean providing for a high degree of usability. Although I did not choose to do so, many users will appreciate its integration into Visual Studio. Its many important fine-grained features most important of which include its overload induction engine (heavy overloading of single-character identifiers), string encryption option, fine-grained control of features that are to be preserved, ability to provide the customer incremental patches, simple and useful map file and its ability to embed water marks make this full-featured product first in its class.

Salamander RemoteSoft .NET Explorer, Version 2.0

Dr. Huihong Luo, the architect of the Salamander .NET Explorer, has been extremely helpful and responsive to questions sent during this review. I would expect customers to enjoy a high degree of support and satisfaction if they were to run into any problems while using the .NET Explorer.

The .NET Explorer product is both a capable decompiler and obfuscator. The focus in this review is on its obfuscation features.

Listing 10 shows the decompiled source code after obfuscating Listing 1 (the generic sort) using the Salamander .NET Explorer obfuscator (henceforth referred to as the Salamander obfuscator).

Listing 10 – Decompiled Source Code After Using the Salamander Obfuscator

```
public static void @abstract<T>(T[] localArray1, int num4)
    where T: IComparable
{
    for (int num1 = num4 - 1; num1 >= 1; num1--)
    {
        T local1 = localArray1[0];
```

```

        int num2 = 0;
        for (int num3 = 1; num3 <= num1; num3++)
        {
            if (localArray1[num3].CompareTo(local1) > 0)
            {
                local1 = localArray1[num3];
                num2 = num3;
            }
        }
        T local2 = localArray1[num2];
        localArray1[num2] = localArray1[num1];
        localArray1[num1] = local2;
    }
}

```

```

namespace @abstract {
    internal class @abstract
    {
        // Methods
        internal @abstract();
        public static CultureInfo @abstract();
        public static ResourceManager @abstract();
        public static void @abstract(CultureInfo);

        // Fields
        private static CultureInfo @abstract;
        private static ResourceManager @abstract;
    }

    public class @as : ApplicationSettingsBase
    {
        // Methods
        static @as();
        public @as();
        public static as @abstract();

        // Fields
        private static as @abstract;
        private static object @abstract;
    }
}

public static void @abstract()
{
    double[] numArray1 = new double[8] { 2.5, -1.3, -0.75,
1.5, 2.4,
0.75, 1 } ;
                                -0.5,
}

```




```
abstract.abstract<double>(numArray1, numArray1.Length);  
for (int num1 = 0; num1 < numArray1.Length; num1++)  
{  
    Console.WriteLine(numArray1[num1] + " ");  
}  
Console.WriteLine();  
Console.ReadLine();  
}
```

Here, C# names were chosen as the base-set for name mangling (the identifiers in the original C# source file converted to standard C# names). You may wish to compare Listing 10 with Listing 5.

The Salamander obfuscator does not support string encryption. After obfuscating and decompiling Listing 6, the source code is shown in Listing 11.

Listing 11 – Decompiled Version of Listing 6 Using Salamander Obfuscator

```
public @abstract()  
{  
    this.abstract = "Hi, my name is Paul.";  
}  
  
public static void @abstract()  
{  
    abstract abstract1 = new abstract();  
    Console.WriteLine("Enter password: ");  
    string text1 = Console.ReadLine();  
    if (text1.Equals(abstract1.abstract))  
    {  
        Console.WriteLine("Correct password.");  
    }  
    else  
    {  
        Console.WriteLine("Incorrect password.");  
    }  
    Console.ReadLine();  
}
```

This example once again highlights the importance of being able to achieve string encryption.

Salamander does not provide the same degree of control as Dotfuscator in determining which features are not to be transformed (mangled) in building the obfuscated assembly. Class names may be chosen for preservation, method names or all public members and all fields may be selected for preservation. For many purposes this is adequate. Identifier preservation is achieved using a context menu that appears when right-mouse clicking the feature that you wish to preserve.

The on-line documentation, although adequate, is less detailed than the documentation provided in Dotfuscator. In part, this is because Salamander contains fewer features.

The log file (Dotfuscator's equivalent of a map file) produced in connection with the generic sort is shown in Listing 12. It provides important and useful information to the user.

Listing 12 – Salamander Log File after Obfuscating Listing 1

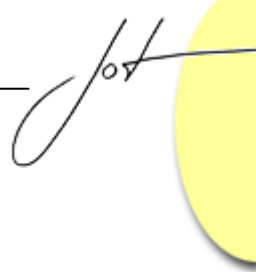
```
// Remotesoft MSIL Disassembler 1.1.5.
// Copyright (C) 2002 Remotesoft Inc. All rights reserved.
// http://www.remotesoft.com/dotexplorer/index.html

.module Sorting
.namespace GenericSorting
{
    .class public Sorting => as.abstract
    {
        .method public static void SelectionSort(!!0[] data, int32 size) =>
abstract
        .method public void .ctor()
    }

    .class public Application => base.abstract
    {
        .method public static void Main() => abstract
        .method public void .ctor()
    }
}

.namespace abstract
{
    .class private Resources => abstract.abstract
    {
        .field private static class
[mscorlib]System.Resources.ResourceManager _resMgr => abstract
        .field private static class
[mscorlib]System.Globalization.CultureInfo _resCulture => abstract
        .method assembly void .ctor()
        .method public static class
[mscorlib]System.Resources.ResourceManager get_ResourceManager() =>
abstract
        .method public static class
[mscorlib]System.Globalization.CultureInfo get_Culture() => abstract
        .method public static void set_Culture(class
[mscorlib]System.Globalization.CultureInfo value) => abstract

        .property class [mscorlib]System.Resources.ResourceManager
ResourceManager() : deleted
    }
}
```



```

    {
        .get class [mscorlib]System.Resources.ResourceManager
Sorting.Properties.Resources::'abstract'()
    }

    .property class [mscorlib]System.Globalization.CultureInfo Culture()
: deleted
    {
        .get class [mscorlib]System.Globalization.CultureInfo
Sorting.Properties.Resources::'abstract'()
        .set void Sorting.Properties.Resources::'abstract'(class
[mscorlib]System.Globalization.CultureInfo)
    }
}

.class public Settings => abstract.as
{
    .field private static class Sorting.Properties.Settings m_Value =>
abstract
    .field private static object m_SyncObject => abstract
    .method public static class Sorting.Properties.Settings get_Value()
=> abstract
    .method private static void .ctor()
    .method public void .ctor()

    .property class Sorting.Properties.Settings Value() : deleted
    {
        .get class Sorting.Properties.Settings
Sorting.Properties.Settings::'abstract'()
    }
}

}

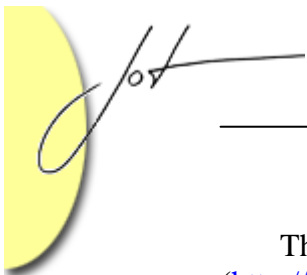
.class private <PrivateImplementationDetails>{F458E8E5-6075-4944-A3B6-
2E98BF906274} => base.as
{
    .class nested private sealed __StaticArrayInitTypeSize=64 => abstract
    {
    }

    .field assembly static valuetype
'<PrivateImplementationDetails>{F458E8E5-6075-4944-A3B6-
2E98BF906274}'/'__StaticArrayInitTypeSize=64' $$method0x6000007-1 =>
abstract
}

```

Salamander does not currently support the embedding of water marks in its obfuscated assemblies.

One of Salamander's major features, Protection, could not be evaluated since it does not currently work with .NET 2005 assemblies. Dr. Luo has indicated that this important capability will be working under .NET 2005 in several months.



The following claim is made on RemoteSoft's website (<http://www.remotesoft.com/salamander/protector.html>)

“Our protector is not an obfuscator, rather it converts the decompilable Microsoft Intermediate Language code (MSIL or CIL) of your assemblies into native format while keeping all .NET metadata intact, and thus it provides the same level of protection as native C/C++ code. Further more, it offers code, string and resource encryption, and therefore, it provides even better protection than native C/C++ code.”

I look forward to testing and later reporting more about this capability using the suite of .NET 2005 assemblies that were used in this review.

A table that summarizes the major features of Dotfuscator and Salamander .NET Explorer is shown below.

Major Features of Dotsfucator and Salamander .NET Explorer Obfuscators

Feature	Map file	Incremental obfuscation	Preservation of user-selected features	String encryption	Water marks	Integration With Visual Studio
Dotfuscator	x	x	x	x	x	x
Salamander	x	x	x			

About the author



Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises. His latest book, to be published by Course Technology in late 2005, is entitled *Modern Software Development Using C#.NET*.