

Eazfuscator.NET Documentation

_

Copyright © 2017 Gapotchenko

Table of Contents

What Is Eazfuscator.NET?

1. Introduction

Definition of Obfuscation Why .NET Applications Need Obfuscation In Theory In Practice When to Use Obfuscation Drawbacks of The Obfuscation 2. Quick Start

3. How Does Eazfuscator.NET Work? Obfuscation Techniques Symbol Renaming String Encryption Constant Literals Pruning Overload Induction Class Hierarchy Linerization XML Documentation Filter XAML Renaming Optimization Techniques Merging of String Literal Duplicates Sealing of Terminal Classes String Compression Code Optimizations 4. Advanced Features

About Advanced Features

Declarative Obfuscation Using Custom Attributes System.Reflection.ObfuscateAssemblyAttribute System.Reflection.ObfuscationAttribute .NET Compact Framework, Silverlight, Windows Store and .NET **Core Projects Indirect Declarative Obfuscation Obfuscation Attribute Priorities Conditional Obfuscation Type Members Options are Combinable Diagnostics** Symbol Names Encryption **Advanced Symbol Renaming Options** Symbol Renaming with Printable Characters **Type Renaming Patterns Advanced String Encryption Options Code Control Flow Obfuscation Assemblies Merging** Introduction Instructions Tuning Internalization **Custom Parameters for Merging Assemblies Embedding Introduction** What's the point for embedding when we have merging (or vice versa)? Instructions Tuning Troubleshooting **Resource Encryption**

Introduction Instructions Compression Selective Resource Encryption **Options are Combinable Serialization Tuning Overview Binary Serialization and Obfuscation Self-Interoperability** Non-stable Self-Interoperable Serialization Stable Self-Interoperable Serialization **Debugging** Introduction to Debugging After Obfuscation How It Works **Possible Security Risks Tuning Debug Renaming PEVerify Integration Probing Paths About Probing Paths** How to Define Probing Paths? **Script Variables**

5. Sensei Features

About Sensei Features Code Inlining Protected Private Visibility Custom Attributes Removal Design-Time Usage Protection Overview How It Works Instructions Tuning Resource Sanitization Introduction Instructions Minification Selective Resource Sanitization Options are Combinable 6. Virtualization

Introduction How to Use Code Virtualization How to Use Data Virtualization 7. Troubleshooting

My application is not working properly after obfuscation. Why does it happen? **Troubleshooting Features** Stack Trace Decoding **Inspection-Friendly Obfuscation Preserving the Original Names Disabling ILDASM Suppression** About InternalsVisibleToAttribute Solution #1. Do not use InternalsVisibleToAttribute at all Solution #2. Swap with EditorBrowsable attribute Solution #3. Hide the warning Solution #4. Ignore the attribute "Option Strict Off" Compatibility for VB.NET Introduction **Compatibility Mode Instructions Nonintrusive Debugging** Introduction **Sample Scenario** Warnings and Errors **Disabling Warnings Treat Warnings as Errors** Long-Term Compatibility **Compatibility Version** Demanding the Specific Version of Eazfuscator.NET Error Codes Knowledge Base EF-1099: Unable to load input assembly, reflection load failed EF-3035: Assembly or part of it is already obfuscated

8. Best Practices

Introduction General Best Practices Keeping the Balance Human Factors Keeping It Simple The Paralysis of Simplicity 9. Deployment

About Eazfuscator.NET Deployment Microsoft Installer (MSI) NuGet Package Manager Command Line Interface Glossary

Bibliography



What Is Eazfuscator.NET?

Eazfuscator.NET is an obfuscator and optimizer for .NET platform.

The main purpose of obfuscator is to protect *intellectual property* of the software.

Key features:

- Easy to use as 1-2-3
- Automatic <u>code protection</u> with variety of supported <u>obfuscation techniques</u>
- Automatic <u>optimizations</u>
- Can obfuscate any 100% managed .NET assembly
- Provides easy to use GUI interface as well as classical command line interface
- Microsoft Visual Studio integration. Supported versions are Microsoft Visual Studio 2005 2017 including Express editions
- Supports automatic builds

Supported platforms and technologies:

- .NET Framework versions 2.0 4.7
- <u>XAML with intelligent renaming of symbols</u>
- Enterprise-grade technologies: ClickOnce, VSTO, VSIX, MEF, Entity Framework, ASP.NET and many others
- .NET Standard versions 1.0 2.0
- .NET Core versions 1.0 2.0
- Universal Windows Platform (UWP)
- Windows Store applications for Windows 8 8.1 and Windows Phone 7 8.1 platforms
- Silverlight 2 5
- XNA applications for Windows, Xbox 360 and Zune platforms
- .NET Compact Framework versions 2.0 and 3.5

Why Eazfuscator.NET so outstanding among others?

- To protect your *intellectual property* you need to perform just several mouse clicks do not waste your precious hours to manually enter all the complex settings and make killing-hard decisions as you do with the most other obfuscators
- Eazfuscator.NET has as its object to <u>automatically protect</u> intellectual property to the maximum possible extent but without breaking .NET assembly functionality
- Eazfuscator.NET applies <u>code optimizations</u> to deliver the best performance to your applications. Furthemore, Eazfuscator.NET is built to deliver the best runtime performance from ground up. It does not use dirty tricks, hacks and incompatible techniques that break application reliability, performance and satisfaction of your customers.
- The history of obfuscation technology knows the score:
 - Every single obfuscator before Eazfuscator.NET required a lot of manual configuration, tuning and integration efforts.
 - Every single obfuscator after Eazfuscator.NET tends to claim it does nearly everything out of the box.

How to use it?

See Quick Start guide.



Chapter 1. Introduction

Table of Contents

Definition of Obfuscation Why .NET Applications Need Obfuscation In Theory In Practice When to Use Obfuscation Drawbacks of The Obfuscation

Definition of Obfuscation

Obfuscated code is a code that is (usually intentionally) very hard to read and understand. Some programming languages and technologies are more prone to obfuscation than others.

There are also programs known as obfuscators that may operate on source code, object code, or both, for the purpose of deterring *reverse engineering*.

Obfuscating code to prevent *reverse engineering* is typically done to manage risks that stem from unauthorized access to source code. These risks include loss of *intellectual property*, ease of probing for application vulnerabilities and loss of revenue that can result when applications are reverse engineered, modified to circumvent metering or usage control and then recompiled. Obfuscating code is, therefore, also a compensating control to manage these risks. The risk is greater in computing environments such as Java and Microsoft's .NET which take advantage of "Just-in-Time" (JIT) compilation technology that allow developers to deploy an application as intermediate code rather than code which has been compiled into native machine language before being deployed. [WikiObCode]

Eazfuscator.NET is an obfuscator that operates on .NET object code entitled as <u>*CIL*</u>.



Why .NET Applications Need Obfuscation

In Theory

Traditionally applications were compiled to the native code of the target CPU. During this translation all the information about the source code was lost. However it is still possible to infer the program operation using debugging and hacking tools but it's difficult, time-consuming and very expensive craft.

In contrast, .NET applications are compiled to the *CIL* code. *CIL* code contains a lot of additional metadata which allows to achieve better interoperability and robustness of the application. But *CIL* metadata allows to infer program operation much easier at the same time. Using specially designed tools such as decompilers it is even possible to retrieve source code of the .NET application. So when .NET application published without being obfuscated it is equivalent to releasing its source code with all painful consequences.

Obfuscation removes redundant *CIL* metadata and scrambles and encrypts the rest. So after obfuscation it is much harder to dig in into obfuscated .NET application. However it is still possible to infer the program operation using specially designed tools but it is difficult, time-consuming and expensive now just as in case of the native code.

In Practice

Let's perform some practical job to undercover the way your .NET application can be decompiled. First of all, we need some sample application to dig in. Let me introduce one of superstar applications with ambitious title "My Precious Idea" as a sample. I should admit that this application is based on the Mike's Gold port of the vintage "Space Invaders" game. This C# application was compiled in Release configuration with Microsoft Visual Studio. Here is the look of sample application when it is running:



Really nice, isn't it? To decompile the application we will use <u>.NET Reflector</u> tool. Then just open "My Precious Idea.exe" file with .NET Reflector. Here is what can be seen after opening:



As you can see in the screenshot above, all application sources can be easily obtained just with several mouse clicks! If your product is not obfuscated then hackers have a huge possibilities to tamper with product features. Unsavory competitors may even copy & paste parts of your code for use in their own products.



When to Use Obfuscation

You may use obfuscation in freeware, shareware or any kind of commercial software when you want to protect your intellectual property.

Obfuscation of open source software has no sense because application source code is always available.



Drawbacks of The Obfuscation

- Exception stack traces in obfuscated assemblies will contain obfuscated symbol names instead of real symbol names. This fact makes difficult to resolve possible application issues. However it is easy to overcome this problem by using stack trace decoding feature.
- Obfuscator may break assembly functionality in case of usage of <u>reflection</u> techniques in the obfuscated application. However Eazfuscator.NET tries to minimize such failures by heuristic detection of *reflection* usage patterns and thoroughly analyzing an assembly item before applying any obfuscation transformations to it. But because of heuristic nature of analysis it is not 100% reliable.

More information about possible problems and their solutions can be found at the <u>chapter about troubleshooting</u>.



Chapter 2. Quick Start

Are you ready to protect your *intellectual property*? Now it is as simple as never was before.

First of all, <u>download</u> and install Eazfuscator.NET. If you install Eazfuscator.NET for the first time then it's highly recommended to restart Visual Studio if it was running during installation.

Then, launch Eazfuscator.NET Assistant from Visual Studio menu. To do that, go to Tools \rightarrow Eazfuscator.NET Assistant menu item and click:



Such floating window will appear:



As you can see, the window above consists of two main zones — green and red. Green zone is responsible for protection and red zone is responsible for protection removal. Whenever you want to protect the project you can drag and drop it to the green zone.



After you drop the project onto protection zone the progress window will appear. This window is shown at the picture below. You should close it when protection is completed.



When you return to Visual Studio it will ask you to reload modified project. Click Reload button.

e Modif	ication Detected
<u>^</u>	The project 'Me.MyPreciousIdea' has been modified outside the environment.
	Press Reload to load the updated project from disk.
	Press Ignore to ignore the external changes. The changes will be used the next time you open the project.
	Reload Ignore

Once protection had been applied to the project, it will be obfuscated during every build in Release configuration.

That's all!

Next you may find useful to read the chapter about the best practices.

Note



Quick Start guide covers just the most common usage pattern.



Chapter 3. How Does Eazfuscator.NET Work?

Table of Contents

Obfuscation Techniques Symbol Renaming String Encryption Constant Literals Pruning Overload Induction Class Hierarchy Linerization XML Documentation Filter XAML Renaming Optimization Techniques Merging of String Literal Duplicates Sealing of Terminal Classes String Compression Code Optimizations

Obfuscation Techniques

Eazfuscator.NET works on the <u>CIL</u> level so any 100% managed .NET assembly can be obfuscated. Automatic obfuscation of satellite assemblies is fully supported. Assemblies with embedded native code are not supported by Eazfuscator.NET. Obfuscated assemblies produced by Eazfuscator.NET can work on alternative .NET runtime implementations such as <u>Mono</u>.

Eazfuscator.NET uses several techniques to obfuscate the code.

Eazfuscator.NET automatically selects appropriate obfuscation methods for every item in .NET assembly. Not all the items can be obfuscated without breaking assembly functionality and so Eazfuscator.NET thoroughly analyzes an item before applying obfuscation transformations to it. Eazfuscator.NET has as its object to protect *intellectual property* to the maximum possible extent but without breaking assembly functionality. If *intellectual property* safety is endangered then Eazfuscator.NET produces a warning message but almost never overobfuscates at cost of breaking assembly functionality.

Let's overview main obfuscation techniques used by Eazfuscator.NET.

Symbol Renaming

Symbol renaming is the most powerful obfuscation method. Classes, methods, properties, fields and method's parameters get renamed with a randomly generated or <u>encrypted</u> title whenever it is applicable. New name usually consists of unprintable or chaotic Unicode characters.

Example 3.1. Source code retrieved with decompiler before symbol renaming

```
private void MainForm_Load(object sender, EventArgs e)
    {
        if (this.DesignMode)
            return;
        this.Icon =
Icon.ExtractAssociatedIcon(Assembly.GetEntryAssembly().Location);
        if (this._UseStartupShutdownEffects)
        {
            this.Opacity = 0;
            this._FadeDirection = true;
            this._LargeFadeStep = true;
            this._OpacityTimer.Start();
        }
        else
        {
            Application.DoEvents();
            Point point1 = this.Location;
            this.Location = new Point(Int32.MinValue,
Int32.MinValue);
            this.Show();
            Application.DoEvents();
            this.Location = point1;
        }
    }
}
```

Example 3.2. Source code retrieved with decompiler after symbol renaming

```
{
            this.Opacity = 0;
            this. = true;
            this. = true;
            this. .Start();
        }
        else
        {
            Application.DoEvents();
            Point point1 = this.Location;
            this.Location = new Point(Int32.MinValue,
Int32.MinValue);
            this.Show();
            Application.DoEvents();
            this.Location = point1;
        }
    }
}
```

There are <u>advanced options</u> available for symbol renaming feature.

String Encryption

String is one of the most widely used data type in applications. At the same time, unencrypted string values allow to easily infer the program operation with an extreme brutality. To avoid that, every string value gets <u>encrypted</u> during obfuscation.

Example 3.3. Source code retrieved with decompiler before string encryption

```
class LicenseManager
{
    internal bool CanRun()
    {
        if
    (LicenseContainer.Get("License").IsValidFor(_CurrentCustomer))
            return true;
        else
            return false;
    }
}
```
}

Example 3.4. Source code retrieved with decompiler after string encryption

```
class LicenseManager
{
    internal bool CanRun()
    {
        if (LicenseContainer.Get( .
    (-2942637)).IsValidFor(_CurrentCustomer))
        return true;
        else
            return false;
    }
    ...
}
```

There are <u>advanced options</u> available for string encryption feature.

Constant Literals Pruning

Constant literals pruning removes redundant meta information from the obfuscated .NET assembly whenever it is possible. This information is often not necessary at the runtime, and it is useful at compile time only. However its presence in the compiled assembly can lead to additional weakness for decompilation.

Example 3.5. Source code retrieved with decompiler before constant literals pruning

```
class LicenseManager
{
    public enum Decision
    {
        Allow,
        Deny,
    }
}
```

```
UnrestrictedDeveloperMode,
        Lock
    }
    internal Decision MakeDecision()
    {
        if (CanRun())
            return Decision.Allow;
        if (IsDeveloperSite())
            return Decision.UnrestrictedDeveloperMode;
        int int1 = UnlicensedRunCount;
        if (int1 > MaxUnlicensedRunCount)
            return Decision.Lock;
        int1++;
        UnlicensedRunCount = int1;
        return Decision.Deny;
    }
}
```

Example 3.6. Source code retrieved with decompiler after constant literals pruning

```
class LicenseManager
{
    public enum Decision
    {
    }
    internal Decision MakeDecision()
    {
        if (CanRun())
            return 0;
        if (IsDeveloperSite())
            return 2;
        int int1 = UnlicensedRunCount;
        if (int1 > MaxUnlicensedRunCount)
            return 3;
        int1++;
        UnlicensedRunCount = int1;
        return 1;
    }
```

Overload Induction

Overload induction is a complementary obfuscation method to <u>symbol renaming</u> technique. Formally, overload induction algorithm minimizes the number of unique symbol names in the obfuscated assembly. As a result, obfuscated classes, fields, properties and methods may have the same name as long as it doesn't violate symbol resolution rules used by .NET runtime. Such symbol names sameness makes prying intruder absolutely entangled.

Example 3.7. Source code retrieved with decompiler before symbol renaming using overload induction

```
class MainForm
{
    private bool _UseStartupShutdownEffects;
    private bool _FadeDirection;
    private bool _LargeFadeStep;
    private System.Windows.Forms.Timer _OpacityTimer;
    private void MainForm_Load(object sender, EventArgs e)
    {
        if (this.DesignMode)
            return;
        this.Icon =
Icon.ExtractAssociatedIcon(Assembly.GetEntryAssembly().Location);
        if (this._UseStartupShutdownEffects)
        {
            this.Opacity = 0;
            this. FadeDirection = true;
            this._LargeFadeStep = true;
            this._OpacityTimer.Start();
        }
        else
        {
            Application.DoEvents();
            Point point1 = this.Location;
```

}

```
this.Location = new Point(Int32.MinValue,
Int32.MinValue);
    this.Show();
    Application.DoEvents();
    this.Location = point1;
    }
  }
  private int CalculateSize()
  {
    return this.Width * 2 / 3;
  }
....
}
```

Example 3.8. Source code retrieved with decompiler after symbol renaming using overload induction

Note



In this listing unprintable characters in symbol names were replaced with visible ones (A, B, C, ...) to show up names' uniqueness distribution.

```
this.Opacity = 0;
            this.B = true;
            this.C = true;
            this.D.Start();
        }
        else
        {
            Application.DoEvents();
            Point point1 = this.Location;
            this.Location = new Point(Int32.MinValue,
Int32.MinValue);
            this.Show();
            Application.DoEvents();
            this.Location = point1;
        }
    }
    private int A()
    {
        return this.Width * 2 / 3;
    }
}
```

Class Hierarchy Linerization

Class hierarchy linearization is a complementary obfuscation method to <u>symbol</u> renaming technique. This method is applied to the class names only. During obfuscation all the information about class namespaces is irreversibly destroyed when it's possible. After the obfuscation, all obfuscated classes are linearly located at the large root namespace, so all the information about class affiliation with application's subsystems is pruned.

Example 3.9. Source code retrieved with decompiler before symbol renaming using class hierarchy linearization

```
namespace MyPreciousIdea
{
    namespace Licensing
    {
        class License
    }
}
```

{ ... } class Manager { ... } } namespace UI { class MainForm { ... } class SettingsForm { ... } class LicensingForm { ... } } ... }

Example 3.10. Source code retrieved with decompiler after symbol renaming using class hierarchy linearization





XML Documentation Filter

Some .NET languages provide an easy way to automatically create XML documentation for projects. You can automatically generate an XML skeleton for your types and members, and then provide summaries, descriptive documentation for each parameter, and other remarks. With the appropriate setup, the XML documentation is automatically emitted into an XML file with the same name as your project and the .xml extension. This file is located in the same directory as the output .exe or .dll file of your project.

Everything seems good until it is discovered that XML documentation file contains descriptions not only for publicly visible classes and members, but for private items too.

Eazfuscator.NET stops that threat by instantly applying XML documentation filter on every obfuscation run. XML documentation for all non-public classes, methods, fields, properties and events is automatically pruned so that essential knowledge about component internals does not leak to the rest of the world anymore. This feature is essential for component developers and publishers.

XAML Renaming

XAML markup language is used by WPF, Silverlight, Windows Store applications to define elements, events, data bindings and other aspects of user interface. XAML renaming is the process of simultaneous renaming of related symbols in code and XAML during obfuscation.

Eazfuscator.NET finds all connections between XAML and code:

public partial class (MainWindow): Window public (MainWindow)) { InitializeComponent(); } private void Button_Click_1(object sender, RoutedEventArgs e) { MessageBox.Show("OK"); } } <Window x:Class="WpfApp1.MainWindow" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="A Sample" Height="350" Width="525"> <Grid> <Button Content="Ipsum Lorem" HorizontalAlignment="Left" Margin="17,17,0,0" VerticalAlignment="Top" Width="75" Click= "Button Click 1")> </Grid> </Window>

Then, the related symbols get renamed. Synchronously and consistently in code and XAML:



As a result, XAML renaming delivers higher obfuscation coverage and ensures that all related items are accordingly and correctly processed in XAML and code.



Optimization Techniques

Eazfuscator.NET performs several code and metadata optimizations during obfuscation.

Merging of String Literal Duplicates

Merging of string literal duplicates is a complementary optimization method to <u>string encryption</u> obfuscation technique. Formally, this optimization ensures that all encrypted string values are unique in one .NET assembly. Such kind of optimization also widely known as string pooling.

Example 3.11. Hypothetical string table before encryption

String Values
Abracadabra
Siam
Foo
Abracadabra
Foo
Bar

Example 3.12. Hypothetical string table after encryption with duplicates merging

Original String Values	Encrypted String Values
Abracadabra	}S£tP)€_9€[]
Siam	@!€NayfI*
Foo	!>@"buY]
Bar	E)€a£J

Sealing of Terminal Classes

Sealed class is a specially marked class which can not be inherited. .NET runtime uses a knowledge about such classes to perform an optimization on virtual method calls, such optimization has considerable impact on overall application performance. Therefore Eazfuscator.NET analyzes the classes of .NET assembly and marks all not inheritable ones as sealed when it is possible. However, this optimization does not enhance obfuscation; information about class inheritance can be used by a hacker as an additional information to infer the program operation. But in this case the risk is lower than performance benefits.

Example 3.13. Source code retrieved with decompiler before sealing of terminal classes

```
class Shape
{
    ...
}
class Circle : Shape
{
    ...
}
```

Example 3.14. Source code retrieved with decompiler after sealing of terminal classes

```
class Shape
{
    ...
}
sealed class Circle : Shape
{
    ...
}
```

String Compression

String compression is an automatic size optimization which results in smaller assemblies. Large strings are compressed during obfuscation. The compressed strings are uncompressed on demand during the run time. The decompression algorithm is incredibly fast, so there is no performance penalty observed — the actual decompression speed comes very close to the speed of a simple memory copy.

String compression is a part of string encryption technique and is on by default.

Code Optimizations

Eazfuscator.NET applies code optimizations to deliver the best performance to your applications. .NET compilers such as C#, VB.NET and JIT already do a pretty decent job in this area. But what they all do not do is high-level optimizations.

High-level optimization is a fresh trend in optimization technology and *Eazfuscator.NET establishes itself as the first tool* to deliver this technology to the wide .NET user base. The best way to briefly describe high-level optimization is to start thinking as developer thinks: we all know that there are some methods and code patterns which are faster than others. What Eazfuscator.NET does is this: it finds the slow code and swaps it with faster one. Eazfuscator.NET uses a preciously brewed knowledge base of common and efficient code patterns that you can meet in every .NET application.

At first glance, high-level optimization is very similar to a well-known *peephole optimization* approach. But the main difference is that the classical peephole optimization works only on a small window of target machine instructions, while high-level optimization works at the *application-wide level* and considers control and data flows as well as the *sacred* knowledge about specific frameworks such as LINQ, MEF and others.

Let's take a look at example.

Example 3.15. The slow code

```
[Flags]
enum RunOptions
{
    None = 0x00,
```

The code above uses Enum.HasFlag method to check whether PrepareDatabase flag is set. Being sweet in syntax, the code has astonishingly bad performance due to boxing operations that are invisibly generated by C# compiler.

Example 3.16. The fast code. Produced by Eazfuscator.NET after optimizing the slow code

```
public void Run(RunOptions options)
{
    if ((options & RunOptions.PrepareDatabase) ==
RunOptions.PrepareDatabase)
        InitializeDatabase();
    ...
}
```

As you can see, Eazfuscator.NET emitted functionally equivalent code. The result of optimization is 500x speed improvement of condition evaluation over original slow code.

The optimizer is on by default and works behind the scenes during obfuscation.

To get information about advanced obfuscation algorithms please read the <u>next</u> <u>chapter</u>.



Chapter 4. Advanced Features

Table of Contents

About Advanced Features Declarative Obfuscation Using Custom Attributes System.Reflection.ObfuscateAssemblyAttribute System.Reflection.ObfuscationAttribute .NET Compact Framework, Silverlight, Windows Store and .NET Core **Projects Indirect Declarative Obfuscation Obfuscation Attribute Priorities Conditional Obfuscation Type Members Options are Combinable Diagnostics** Symbol Names Encryption **Advanced Symbol Renaming Options** Symbol Renaming with Printable Characters **Type Renaming Patterns Advanced String Encryption Options Code Control Flow Obfuscation Assemblies Merging** Introduction Instructions Tuning Internalization **Custom Parameters for Merging Assemblies Embedding** Introduction What's the point for embedding when we have merging (or vice versa)? **Instructions** Tuning

Troubleshooting Resource Encryption Introduction Instructions **Compression Selective Resource Encryption Options are Combinable Serialization Tuning Overview Binary Serialization and Obfuscation Self-Interoperability** Non-stable Self-Interoperable Serialization Stable Self-Interoperable Serialization **Debugging** Introduction to Debugging After Obfuscation How It Works **Possible Security Risks Tuning Debug Renaming PEVerify Integration Probing Paths About Probing Paths** How to Define Probing Paths? **Script Variables**

About Advanced Features

You may want to get more control on obfuscation process when you become more familar with obfuscation. Advanced features of Eazfuscator.NET allow you to achieve this.

To start with advanced features it is recommended to read about <u>declarative</u> <u>obfuscation</u>.



Declarative Obfuscation Using Custom Attributes

The .NET Framework since version 2.0 provides two custom attributes designed to make it easy to change obfuscation behavior. Using these two attributes it is possible to override default decisions made by Eazfuscator.NET during obfuscation. Also these attributes can be used to configure advanced Eazfuscator.NET features. It is assumed that you have a knowledge about custom attributes and how to apply them in your development language.

System.Reflection.ObfuscateAssemblyAttribute

This attribute can be applied to an assembly to tell obfuscator how to treat it. Setting the AssemblyIsPrivate property to false tells obfuscator to treat an assembly as a library. Setting the AssemblyIsPrivate property to true tells obfuscator to treat an assembly as an executable. The difference is how Eazfuscator.NET renames and seals public types and their public members. In case of an executable all public types and their public members are considered terminal so they gets renamed. In case of a library, those types and members may be used by other assemblies, thus they do not get renamed.

The default obfuscator behavior is to treat all .dll assemblies as libraries and all .exe and .com assemblies as executables. If the assembly has another file extension (nor .dll neither .exe/.com) then Eazfuscator.NET treats such assemblies depending on the entry point presence. If the assembly has the entry point then it is treated as an executable, otherwise it is treated as a library.

System.Reflection.ObfuscationAttribute

This attribute is the main troubleshooter of the reflection related problems that may occur during obfuscation. As it was mentioned above Eazfuscator.NET has no formal reflection scenarios analysis engine and it uses heuristic algorithms instead. Decisions produced by those algorithms may be wrong in some rare cases so you may need to override them by using System.Reflection.ObfuscationAttribute

System.Reflection.ObfuscationAttribute can be applied to a type. Possible feature property values are "all" (by default), "renaming" and "properties

renaming". So if you want to disable renaming of your class you may write something like an example below (C#):

Example 4.1. Disabling class renaming

If you want to disable renaming of your class and all its members you may write:

Example 4.2. Disabling class and its members renaming

If you want to disable renaming of the properties in your class you may write:

Example 4.3. Disabling class properties renaming

```
[System.Reflection.ObfuscationAttribute(Feature = "properties
renaming")]
class MyOneThousandAndThirdClass
{
    ...
}
```

Sometimes it may be useful to disable just single or several properties in a class. In order to do that you may write:

Example 4.4. Disabling single class property renaming

```
class MyOneThousandAndFourthClass
```

```
{
  [System.Reflection.ObfuscationAttribute(Feature =
  "renaming")]
  public string DisplayName
  {
    get;
    set;
  }
}
```

In some rare cases you may want to disable the renaming of properties for anonymous types. In order to do that, you should apply the following attribute at the assembly level:

Example 4.5. Disabling the renaming of anonymous type properties

```
using System.Reflection;
[assembly: Obfuscation(Feature = "anonymous type properties
renaming", Exclude = true)]
```

Another appliance of System.Reflection.ObfuscationAttribute is to configure Eazfuscator.NET features. You may find more information about this at the following places:

- Advanced symbol renaming options
- Advanced string encryption options
- Code control flow obfuscation
- Assemblies merging
- Assemblies embedding
- <u>Resource encryption</u>
- <u>Debugging</u>

.NET Compact Framework, Silverlight, Windows Store and .NET Core Projects

There are no System.Reflection.ObfuscateAssemblyAttribute and System.Reflection.ObfuscationAttribute attributes available in .NET Compact Framework, Silverlight and WinRT runtimes. So if you want to use declarative obfuscation you must define corresponding attributes in your assembly. The easiest way to do this is to add ready-to-use file ObfuscationAttributes.cs (for C#) or ObfuscationAttributes.vb (for VB.NET) to your project. These files can be found at Start Menu \rightarrow Eazfuscator.NET \rightarrow Eazfuscator.NET Code Snippets menu item. Alternatively they can be found at C:\Program Files\Eazfuscator.NET\Code Snippets path.

Note



The path may differ depending on the installation options and operating system. For example, this path may look like C:\Program Files (x86)\Eazfuscator.NET\Code Snippets on 64-bit operating systems.

Indirect Declarative Obfuscation

Sometimes it may not be possible to access the class definition directly to apply a custom attribute for obfuscation tuning. If latter is the case then indirect declarative obfuscation comes to the rescue. It allows to indirectly tune the obfuscation by using custom attributes defined at the assembly level.

Let's take a closer look on an example. Suppose there is a class MyNamespace.ResourceClass1 and its declaration can not be changed because it was automatically generated by a tool. So how to disable the renaming of that class during obfuscation? The solution is to use indirect declarative obfuscation. In order to do that, you should apply the following attribute at the assembly level (C#):

Example 4.6. Indirectly disable the renaming of a class

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type
MyNamespace.ResourceClass1: renaming", Exclude = true,
ApplyToMembers = false)]
```

Please note, the feature string starts with "Apply to type" expression in the sample above. That expression signals Eazfuscator.NET that feature should be redirected to MyNamespace.ResourceClass1 class. There is a semicolon after the class name; just next to it, there is a real feature name "renaming". So, the above sample code is virtually equivalent to the code shown below (C#):

```
namespace MyNamespace
{
   [System.Reflection.ObfuscationAttribute(Feature =
"renaming", Exclude = true, ApplyToMembers = false)]
   class ResourceClass1
   {
    ...
   }
}
```

The class name in indirect expression can contain mask characters such as * and ?, so one indirect expression can match several classes. * stands for several any characters, ? stands for any character.

As for example, it's possible to disable all obfuscation features for all classes inside a given namespace. The following attribute should be applied at the assembly level in order to do that (C#):

Example 4.7. Indirectly disable all obfuscation features for a given namespace

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type
SomeExcludedNamespace.*: all", Exclude = true, ApplyToMembers =
true)]
```

A more powerful configuration syntax is available in <u>conditional obfuscation</u>.

Obfuscation Attribute Priorities

Introduction

Sometimes you may want to define obfuscation attributes in several places.

For example, the given assembly may be configured by two files. One file, ObfuscationSettings.cs, defines the obfuscation directives directly related to the given assembly. Another file, CommonObfuscationSettings.cs, defines the obfuscation directives common to all obfuscated assemblies in a whole project. That file is shared among all obfuscated assemblies in a project.

Then you decide that <u>string encryption</u> is not needed for most assemblies, so you just disable it at the common level in CommonObfuscationSettings.cs file:

```
[assembly: Obfuscation(Feature = "string encryption", Exclude =
true)]
```

Some time after, it turns out that a particular assembly ContosoEngine.dll should have string encryption enabled. ObfuscationSettings.cs file defined in ContosoEngine assembly looks like this:

```
[assembly: Obfuscation(Feature = "string encryption", Exclude =
false)]
```

Please note that both ObfuscationSettings.cs and CommonObfuscationSettings.cs files are included in compilation of ContosoEngine assembly as source code files. So both directives find their ways into resulting compiled ContosoEngine.dll file.

What happens when Eazfuscator.NET sees conflicting obfuscation directives in input assembly? The exact semantics depends on a directive. Eazfuscator.NET may just take the least permissive directive for features like string encryption. Another strategy is to take the very first directive and abandon the rest (note that C# and many other .NET compilers do not guarantee the order of custom attributes in resulting compiled assembly). Nevertheless this is not what you want when it comes to several configuration sources where you want to have **priorities**. E.g. directives defined in ObfuscationSettings.cs file should have higher priority than those defined in CommonObfuscationSettings.cs file.

So how to achieve the prioritization of obfuscation attributes? Just follow

priority syntax described below.

Priority Syntax

Priority is defined by a numeric prefix:

[Obfuscation(Feature = "1. <some feature here>")]

The idea behind priority prefixes is to form a natural list:

```
[Obfuscation(Feature = "1. <some high priorty feature here>")]
[Obfuscation(Feature = "2. <medium priorty feature>")]
[Obfuscation(Feature = "3. <a feature with the lowest
priority>")]
```

where the first item has the highest priority, the second is less important and so on. Just like your typical TODO list.

Note



Priority prefix can only contain a natural number (a positive integer). Zero and negative priorities are not allowed.

Complete Example

So here is how the aforementioned configuration files should be defined to make the prioritization work according to the task described in <u>introduction</u>. The content of CommonObfuscationSettings.cs file:

```
[assembly: Obfuscation(Feature = "2. string encryption", Exclude
= true)]
```

The content of ObfuscationSettings.cs file for ContosoEngine assembly:

```
[assembly: Obfuscation(Feature = "1. string encryption", Exclude
= false)]
```



Conditional Obfuscation

Conditional obfuscation is an extension feature to <u>indirect declarative</u> <u>obfuscation</u>. Conditional obfuscation allows to process the types in a bulk according to their natural properties such as visibility (public, internal, protected, private), subtype (class, struct, enum, delegate) and others.

This functionality is achieved by when conditional clause which can be specified in an obfuscation attribute. So the full conditional attribute notation has the following form:

```
[assembly: Obfuscation(Feature = "Apply to type [name mask] when
[condition]: [feature]")]
```

where [condition] is a string defining the condition for a match.

Condition is defined as a boolean predicate with Pascal syntax. Quick example of an conditional attribute is shown below.

Example 4.8. Indirectly disable renaming of all internal enums and their members

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type * when internal
and enum: renaming", Exclude = true, ApplyToMembers = true)]
```

Please take a closer look at internal and enum predicate in the sample above. What it says is this: the result of predicate is true when internal variable equals to true and enum variable equals to true too; otherwise the result of predicate is false. The values of internal and enum variables are calculated for every type in the assembly, describing the natural properties of a CLR type in boolean form.

The list of available variables is presented in the table below.

Table 4.1. The list of available variables for conditional obfuscation of types

Variable	Description
abstract	true if the type is abstract; otherwise, false
anonymous	true if the type is anonymous; otherwise, false
class	true if the type is a class; otherwise, false
delegate	true if the type is a delegate; otherwise, false
enum	true if the type is an enumeration; otherwise, false
generic	true if the type is generic; otherwise, false
interface	true if the type is an interface; otherwise, false
internal	true if the type is internal; otherwise, false
nested	true if the type is nested; otherwise, false
private	true if the type is private; otherwise, false. Applies to nested types only; false if the type is not nested
protected	true if the type is protected; otherwise, false. Applies to nested types only; false if the type is not nested
public	true if the type is public; otherwise, false
sealed	true if the type is sealed; otherwise, false
serializable	true if the type is serializable; otherwise, false
static	true if the type is static; otherwise, false
struct	true if the type is a structure; otherwise, false

Predefined constants can be used in expressions as well. The list of available constants is presented in the table below.

Table 4.2. The list of available constants for conditional obfuscation

Constant	Description
false	false value
true	true value

Built-in functions can be used in expressions too. The list of available functions is presented in the table below.

Function	Description
inherits(' <i>type_name</i> ')	Returns true if the type inherits another type sp parameter; otherwise, false. Inherits function inherited types including interfaces. Type name either the full type name or a mask for a bulk ma
extends(' <i>type_name</i> ')	Returns true if the type extends another type sp parameter; otherwise, false. Extends function (type, but all implemented and inherited interface Type name parameter can contain either the full for a bulk match
has_attribute(' <i>type_name</i> ')	Returns true if the type or member has a custon <i>type_name</i> parameter; otherwise, false. Type not contain either the full type name or a mask for a Example: has_attribute(' <i>System.ComponentModel.Dis</i>

Table 4.3. Built-in functions for conditional obfuscation

Variables, functions and constants can be combined by operators. They have the standard Pascal precedence. The list of available operators is presented in the table below.

Table 4.4. The list of available operators for conditional obfuscation

Operator	Description	Priority
not	Unary operator for boolean negation	Highest
and	Binary operator for boolean and operation	Medium
or	Binary operator for boolean or operation	Lower
=	Binary operator for boolean equal operation	Lowest
<>	Binary operator for boolean not equal operation	Lowest

The precedence of operations can be changed by parentheses.

Let's take a look on examples.

Example 4.9. Disable renaming of all types except enums

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type * when not enum:
renaming", Exclude = true)]
```

Example 4.10. Disable renaming of all internal nested and serializable types together with their members

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type * when internal
and (nested or serializable): renaming", Exclude = true,
ApplyToMembers = true)]
```

Example 4.11. Disable renaming of all types except internal nested and serializable types

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type * when not
(internal and (nested or serializable)): renaming", Exclude =
true)]
```

Example 4.12. Disable renaming of all interfaces in Contoso.Core.Services namespace

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type
Contoso.Core.Services.* when interface: renaming", Exclude =
true)]
```

Example 4.13. Disable renaming of all classes derived from System.IDisposable interface. Renaming of members for matched classes is disabled too

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type * when class and
extends('System.IDisposable'): renaming", Exclude = true,
ApplyToMembers = true)]
```

As you can see the conditions can have any complexity and can be freely defined to achieve your specific goals.

Type Members

It was described how to tune the obfuscation of types in a bulk way in the <u>section above</u>. But what about type members such as methods, fields, properties and others? Sometimes it may be beneficial to process them in a bulk way too.

The declarative attribute for conditional obfuscation of type members has the following form:

```
using System.Reflection;
[Obfuscation(Feature = "Apply to member [name mask] when
[condition]: [feature]")]
class Sample1
{
....
}
```

[name mask] is a pattern which selects the members according to their names. [condition] allows to specify a boolean predicate to select members according to their properties. The rules are all the same as for types; the only difference is a set of variables which can be used in a boolean predicate. The list of available variables is presented below.

Table 4.5. The list of available variables for conditional obfuscation of typemembers

Variable	Description
abstract	true if the member is abstract; otherwise, false
const	true if the member defines a literal constant; otherwise, false. Applies to fields only; false if the member is not a field
constructor	true if the member is a constructor; otherwise, false
event	true if the member is an event; otherwise, false
field	true if the member is a field; otherwise, false
generic	true if the member is generic; otherwise, false. Applies to methods only; false if the member is not a method
internal	true if the member is internal; otherwise, false
method	true if the member is a method; otherwise, false
private	true if the member is private; otherwise, false
property	true if the member is a property; otherwise, false
protected	true if the member is protected; otherwise, false
public	true if the member is public; otherwise, false
readonly	true if the member is read-only; otherwise, false. Applies to fields and properties only; false if the member is neither a field nor a property
static	true if the member is static; otherwise, false
virtual	true if the member is virtual; otherwise, false

The information on this topic is extremely bare, so let's take a relaxed look on some real-life samples.

Example 4.14. Disable renaming of public properties

Example 4.15. Disable renaming of all methods

Example 4.16. Disable renaming of internal fields

```
using System.Reflection;
[Obfuscation(Feature = "Apply to member * when field and
internal: renaming", Exclude = true)]
class ContosoHeadquarters
{
....
}
```

Options are Combinable

<u>Conditional obfuscation of types</u> and <u>type members</u> can be easily combined to achieve specific goals in an elegant and powerful way.

Just take a look at the samples below.

Example 4.17. Disable renaming of property Contoso in type Acme. Services

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type Acme.Services:
apply to member Contoso when property: renaming", Exclude =
true)]
```

Example 4.18. Disable renaming of all public properties in all types

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type *: apply to
member * when public and property: renaming", Exclude = true)]
```

Example 4.19. Disable renaming of all public properties in types defined in MyNamespace

```
using System.Reflection;
```

```
[assembly: Obfuscation(Feature = "Apply to type MyNamespace.*:
apply to member * when public and property: renaming", Exclude =
true)]
```

Example 4.20. Disable renaming of all internal events in all public types

```
using System.Reflection;
```

```
[assembly: Obfuscation(Feature = "Apply to type * when public:
apply to member * when event and internal: renaming", Exclude =
true)]
```

Diagnostics

How do you know which classes or members conditional obfuscation applies to?

Sometimes it may be useful to get the full list of classes and members that are targeted by specific conditional obfuscation statement. To achieve that, you can use log feature as shown below:

Example 4.21. Log all affected members of all public classes

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type * when public:
apply to member *: log")]
```

Note



Logging does not affect obfuscation in any way. It just dumps the list of items at Eazfuscator.NET's console output. To see this list in Visual Studio, please take a look at Output Window with View \rightarrow Output (**Ctrl+W,O**) right after obfuscation.


Symbol Names Encryption

Symbol names encryption is a complementary feature to <u>symbol renaming</u> technique. Encryption feature is useful in production scenarios when it's necessary to resolve possible issues with your product. Such issues are very often reported via log files and error stack traces.

But as you might know, symbol names are renamed with randomly generated titles and become irreversibly lost after obfuscation. This makes it nearly impossible to analyze stack traces because it's hard to establish a correlation between error stack trace and original source code. Symbol names encryption can be used to overcome this problem. It encrypts obfuscated symbol names instead of random generation.

Symbol names encryption technology uses symmetrical crypto algorithm underneath. Used crypto algorithm is AES with 256 bits key strength. Cryptographic key for the algorithm is derived from the password. Symbol names encryption produces printable ASCII characters in encrypted symbol names, so error dumps can be easily transfered with E-mail or some other kind of textual error reporting.

By default, symbol names encryption is not used during obfuscation of the assembly.

To enable symbol names encryption you should apply specially formed attribute to your assembly. In order to do that you can use the instructions below.

Instructions on enabling symbol names encryption

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

using System;

```
using System.Reflection;
```

```
[assembly: Obfuscation(Feature = "encrypt symbol names with
password XXXXXX", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="encrypt symbol names with
password XXXXXX", Exclude:=False)>
```

Note

Change XXXXXX with your password. Keep the password in secret.



Passwords with a greater length are more preferable than short ones. Longer passwords have a better informational entropy thus greatly improving cryptographic strength of the encrypted data. It's suggested to have a password which at least consists of 8 characters.

When symbol names encryption is enabled on your project then you able to use <u>stack trace decoding feature</u>.

Tip

If your product or solution consists of several projects then you most likely want to give them all the same encryption password. In order to do that globally, you can create CommonObfuscationSettings.cs (or .vb) file that is shared among all the projects in the solution. Note that Microsoft Visual Studio supports adding of a project file as a reference, so you can add a reference to the same global CommonObfuscationSettings.* file in several projects. Please also note, that a reference to global CommonObfuscationSettings.* file may be added just to one project and then just drag and dropped to all other projects.



Advanced Symbol Renaming Options

Symbol Renaming with Printable Characters

<u>Symbol renaming</u> algorithm uses unprintable unicode characters by default. But sometimes it may be useful to use printable ASCII characters instead. In order to do that you can use the instructions below. Alternatively you may use <u>symbol</u> <u>names encryption</u> for the same purpose.

Instructions on enabling printable characters for symbol renaming

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "rename symbol names with
printable characters", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="rename symbol names with
printable characters", Exclude:=False)>
```

Note



Please note that printable characters in symbol names can be controlled at the assembly level only. For example, it is impossible to use printable characters for some specific class or method; it is possible to do this just for a whole assembly.

Type Renaming Patterns

Eazfuscator.NET removes the namespaces of renamed types by default. This can lead to some issues when badly written code tries to get a namespace of an renamed type via reflection.

Let's see on example what kind of flawed code can suffer from the absence of namespaces.

Example 4.22. Example code which fails with NullReferenceException when the given type has no namespace

```
bool SampleMethod(Type type)
{
    if (type != null &&
    type.Namespace.StartsWith("System.Data"))
        return true;
    return false;
}
```

As you can see in the sample above, the method can fail with NullReferenceException when type.Namespace property returns null indicating that the given type has no namespace. This issue can be easily fixed if you have access to the source code, but sometimes the flawed code has the binary form only.

To workaround possible problems, a custom type renaming pattern can be defined for an assembly, for a type or for a group of types. The examples below show the possible definitions.

Example 4.23. Add 'b' namespace to all renamed types in assembly

```
using System;
using System.Reflection;
```

```
[assembly: Obfuscation(Feature = "type renaming pattern 'b'.*",
Exclude = false)]
```

Example 4.24. Add 'b' namespace to a class

```
using System;
using System.Reflection;
namespace App
{
  [Obfuscation(Feature = "type renaming pattern 'b'.*",
Exclude = false)]
  class Class1
  {
   ...
  }
}
```

Example 4.25. Add 'b' namespace to a group of renamed classes. All classes with 'Impl' name ending are affected

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type *Impl: type
renaming pattern 'b'.*", Exclude = false)]
```



Of course you are free to choose any namespace in a pattern.



Advanced String Encryption Options

<u>String encryption</u> is a technique which encrypts every string in the assembly. It is always turned on by default.

However in some situations you may prefer to turn this feature off. In order to do that you can use the instructions below.

Instructions on disabling string encryption

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "string encryption",
Exclude = true)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
```

```
<Assembly: Obfuscation(Feature:="string encryption",
Exclude:=True)>
```

Note



Please note that string encryption can be controlled at the assembly level only. For example, it is impossible to disable string encryption for some specific class or method; it is possible to do this just for a whole assembly.



Code Control Flow Obfuscation

Code control flow obfuscation allows to make IL code more entangled. Decompilers often crash on such code, so the code may be considered as much better protected.

By default, code control flow obfuscation feature is not used during obfuscation of the assembly.

To enable code control flow obfuscation feature you should apply a specially formed attribute to your assembly. In order to do that you can use the instructions below.

Instructions on enabling control flow obfuscation

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "code control flow
obfuscation", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="code control flow
obfuscation", Exclude:=False)>
```



Assemblies Merging

Introduction

Assemblies merging allows to merge several assemblies into one. This may be beneficial from the deployment and security points of view.

By default, assemblies merging is not used during obfuscation of the assembly.

Instructions

To enable assemblies merging you should apply specially formed attribute(s) to your assembly. In order to do that you can use the instructions below.

Instructions on enabling assemblies merging

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "merge with XXXXX.dll",
Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="merge with XXXXX.dll",
Exclude:=False)>
```

Note

Change XXXXXX.dll with the file name of the assembly you want to merge with.

Tip

Eazfuscator.NET <u>automatically finds</u> the assembly path when only the file name is supplied.

If you prefer to specify the exact file path to assembly then you can use <u>script variables</u>:

```
[assembly: Obfuscation(Feature = @"merge
with $(InputDir)\Lib\AssemblyToMerge.dll",
Exclude = false)]
```

Tip

If you want to merge with several assemblies then just add several attributes:

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "merge with
Assembly1.dll", Exclude = false)]
[assembly: Obfuscation(Feature = "merge with
AnotherAssembly2.dll", Exclude = false)]
```

Note



Usage of assemblies merging may lead to some side effects which may make obfuscation to fail. If such is the case then use the principle of the least common denominator – merge just those assemblies which do not cause obfuscation failure. <u>Assemblies embedding</u> can be used in conjunction or as an alternative to assemblies merging.

Tuning

The satellite assemblies are not merged by default. You may prefer to change that by instructing Eazfuscator.NET to automatically merge them for you. In order to do that, please read the notes below.

The full notation of a custom attribute for assembly merging has the following form:

```
[assembly: Obfuscation(Feature = "merge with [flags]
XXXXXX.dll", Exclude = false)]
```

where [flags] is an optional enumeration of flags separated by spaces.

The list of available flags is presented in the table below.

 Table 4.6. The list of flags for assembly merging attribute

Flag	Description
satellites	Enables automatic merging of satellite assemblies
internalization=auto	Instructs Eazfuscator.NET to automatically decide which public merged types should be internalized. This is the default setting. The typical decision is to internalize a given type. At the same time the type internalization can be inhibited, for example, by the fact that some kinds of WPF controls cannot have internal visibility
internalization=none	Disables the internalization of public merged types
internalization=full	Instructs Eazfuscator.NET to internalize all public merged types

Let's take a look on examples.

Example 4.26. Merge with assembly and its satellite assemblies

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "merge with [satellites]
XXXXXX.dll", Exclude = false)]
```

Example 4.27. Merge with assembly and its satellite assemblies; do not internalize public merged types

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "merge with [satellites
internalization=none] XXXXXX.dll", Exclude = false)]
```

Internalization

Internalization changes the visibility of merged classes from public to internal. By default, classes from merged assemblies are automatically internalized in order to improve obfuscation coverage.

That's fine for the most scenarios but sometimes you may want to change that for some specific classes. Please follow the instructions below to achieve it.

Instructions on disabling internalization for specific class

- 1. Open the source code of a class
- 2. Add a custom attribute as shown below (C#):

```
using System;
using System.Reflection;
[Obfuscation(Feature = "internalization", Exclude = true)]
public class YourClass
```

{ }

For Visual Basic .NET:



Important

<u>Conditional obfuscation</u> is not available for this feature.

Custom Parameters for Merging

Sometimes you may need to pass custom parameters for assembly merging. For example, you may prefer to control class internalization yourself or use some tricky merging feature.

Historically Eazfuscator.NET relied on <u>ILMerge utility</u> in the past. Now it is equipped with its own compatible merger since version 4.1. The new merger understands most of ILMerge configuration parameters and can be configured in the very same way. Please refer to <u>ILMerge documentation</u> for the list of available parameters.

Instructions on passing custom parameters to merger

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "ilmerge custom parameters:
<parameters>", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="ilmerge custom parameters:
<parameters>", Exclude:=False)>
```

Note



Change <parameters> with the parameters you want to pass to merger. Eazfuscator.NET passes */internalize /ndebug* parameters by default when no attribute defined. If you do not want to pass any parameters to merger then change <parameters> with *none* string.



Assemblies Embedding

Introduction

Assemblies embedding allows to embed assembly's dependencies into assembly itself. This may be beneficial from the deployment and security points of view.

Assemblies embedding is similar to merging. The main difference is that the assemblies are not merged into single assembly when they are embedded. They just get encrypted and packed as the assembly resources. As a result, there is a single assembly at the output and it contains the packed dependencies at the same file.

What's the point for embedding when we have merging (or vice versa)?

Assemblies merging delivers the best performance for the resulting assemblies. They can be NGEN'ed, they work in all constrained environments (Windows Phone, Compact Framework etc.). File mappings and JIT'ted code can be cached by the operating system for such assemblies, bringing the blinding fast application startups. Assembly merging definitely rocks.

The only downside of merging is that it's not always possible to apply it without breaking the application. So this is the point where assemblies embedding comes to the rescue.

Embedded assemblies are easy goals to achieve and they work out of the box. Downsides? Well, they are present. Embedded assemblies can not be NGEN'ed, they do not work in some constrained environments (Xbox, Windows Phone and Compact Framefork). The extraction of embedded assemblies during the application load is a performance penalty (penalty is pretty small, so it's unlikely you are able to notice it).

Assemblies embedding brings some benefits as well. The embedded assemblies are encrypted, so this is a securty hardening against the hackers. Embedded assemblies are compressed, bringing the size reduction of the resulting assembly. And of course assemblies embedding is the easiest way to achieve single-file deployment, making your application to consist of a single .exe (or .dll) file.

Instructions

To enable assemblies embedding you should apply specially formed attribute(s) to your assembly. In order to do that you can use the instructions below.

Instructions on enabling assemblies embedding

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "embed XXXXXX.dll", Exclude
= false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="embed XXXXXX.dll",
Exclude:=False)>
```

Note



Change XXXXXX.dll with the file name of the assembly you want to embed.

Important

It is recommended to obsuscate the embedded assemblies.

Tip

Eazfuscator.NET automatically finds the assembly path when only the file name is supplied.

If you prefer to specify the exact file path to assembly then you can use <u>script variables</u>:

```
[assembly: Obfuscation(Feature = @"embed
$(InputDir)\Lib\AssemblyToEmbed.dll",
Exclude = false)]
```

Tip

If you want to embed several assemblies then just add several attributes:

```
[assembly: Obfuscation(Feature = "embed
Assembly1.dll", Exclude = false)]
[assembly: Obfuscation(Feature = "embed
AnotherAssembly2.dll", Exclude = false)]
```

Tuning

Embedded assemblies are compressed and encrypted by default. You may prefer to turn off the compression, encryption or them both. In order to do that, please read the notes below.

The full notation of a custom attribute for assembly embedding has the following form:

```
[assembly: Obfuscation(Feature = "embed [flags] XXXXX.dll",
Exclude = false)]
```

where [flags] is an optional enumeration of flags separated by spaces.

The list of available flags is presented in the table below.

Table 4.7. The	list of flags for	assembly embeddi	ng attribute
rubic m/t rine	mot or mage for	abbembly embedai	ng utti ibutt

Flag	Description
no_compress	Disables the compression
no_encrypt	Disables the encryption
no_satellites	Disables automatic embedding of satellite assemblies
load_from_file	Instructs Eazfuscator.NET to load the embedded assembly from file instead of memory during the obfuscated assembly run-time. This can be used to preserve a meaningful value of Location property from System.Reflection.Assembly type.
immediate_load	Instructs Eazfuscator.NET to load the embedded assembly immediately on a module start. This may be required to satisfy the technologies that rely on a custom assembly resolution mechanism. An example of such technology is WPF which uses XmlnsDefinitionAttribute to locate the assemblies at runtime. Please note that Eazfuscator.NET automatically detects the most of affected technologies and applies the flag automatically when required.

Let's take a look on examples.

Example 4.28. Embed assembly without compression and encryption

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "embed [no_compress no_encrypt]
XXXXXX.dll", Exclude = false)]
```

Example 4.29. Embed assembly without encryption; compression is enabled

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "embed [no_encrypt]
XXXXXX.dll", Exclude = false)]
```

Example 4.30. Embed assembly; compression and encryption are enabled

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "embed XXXXXX.dll", Exclude =
false)]
```

Example 4.31. Embed own satellite assemblies; compression and encryption are enabled

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "embed satellites", Exclude =
false)]
```

Troubleshooting

While assembly embedding is the most non-intrusive way to link the assembly, some rare issues may occur. Possible issues are described in this chapter together with corresponding solutions to avoid them.

Possible Issue #1: Location Property of System.Reflection.Assembly Class

Issue summary. Location property of System.Reflection.Assembly class is often used to find the paths of files near the assembly. While not being a correct solution, this works for most deployment scenarios.

What may go wrong? First of all, Location property can have a completely unexpected value when assembly shadow copying is used, thus breaking the intended application logic. Secondly, Location property has a null value when a corresponding assembly is embedded.

Solution. Use EscapedCodeBase property instead. This property always has a correct value in all deployment scenarios. Please take a look at the sample below.

```
using System;
class Program
{
    static string GetEulaPath()
    {
        var assembly = typeof(Program).Assembly;
        // string location = assembly.Location; // Please do not
    use this. This is a flawed approach
        string location = new
Uri(assembly.EscapedCodeBase).LocalPath; // <-- Use this instead
        return Path.Combine(Path.GetDirectoryName(location),
    "EULA.rtf");
    }
}
```

Alternative workaround is to use load_from_file flag for the embedded assembly.

Possible Issue #2: Custom Assembly Loading

Issue summary. Some applications, libraries and technologies use custom assembly loading mechanisms that go beyond locating assemblies by strong names. They typically rely on some domain-specific properties of an assembly. It may be special assembly name convention, specific custom attribute applied to an assembly etc.

What may go wrong? A particular property of a custom assembly loading mechanism is that it looks for an assembly either at the list of loaded assemblies or at the probing paths for the current AppDomain. This is where Assemblies Embedding may become a breaking change for the custom algorithm – the assembly is no longer there, it is embedded.

Solution. Use immediate_load flag for the embedded assembly. This will instruct Eazfuscator.NET to immediately load the assembly at the module startup.



Resource Encryption

Introduction

Resource encryption feature allows to encrypt and optionally compress the embedded resources of an assembly.

Instructions

To enable resource encryption you should apply an attribute to your assembly. In order to do that you can use the instructions below.

Instructions on enabling resource encryption

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "encrypt resources",
Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="encrypt resources",
Exclude:=False)>
```

Compression

Assembly resources are not compressed by default. If you want to achieve smaller size of an output assembly then you may consider to turn on the resource compression. The [compress] flag turns on the compression when specified, as shown in the sample below.

Example 4.32. Encrypt and compress all resources

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "encrypt resources [compress]",
Exclude = false)]
```

Selective Resource Encryption

Sometimes it may be beneficial to encrypt just some resources while leaving the others intact. The [exclude] flag can be used in order to do that, as shown in the sample below.

Example 4.33. Encrypt all resources except .png files

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "encrypt resources", Exclude =
false)]
[assembly: Obfuscation(Feature = "encrypt resources [exclude]
*.png", Exclude = false)]
```

It may be profitable to go other way around by explicitly specifying just those resources that should be encrypted. This technique is shown in the sample below.

Example 4.34. Encrypt secret.txt and all .sql resources; the others are left intact

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "encrypt resources secret.txt",
```

```
Exclude = false)]
[assembly: Obfuscation(Feature = "encrypt resources *.sql",
Exclude = false)]
```

Options are Combinable

The given options can be combined in a free way giving you the power to choose the best combination for performance, security and possibly obscurity to mislead the hacker.

If you are not sure which combination to choose then just go with a simplest one: encrypt all resources.

If you know what you are doing then you can end up with something like that:

Example 4.35. Advanced resource encryption configuration

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "encrypt resources", Exclude =
false)]
[assembly: Obfuscation(Feature = "encrypt resources [exclude]
License.txt", Exclude = false)]
[assembly: Obfuscation(Feature = "encrypt resources [exclude]
CommandLineOptions.txt", Exclude = false)]
[assembly: Obfuscation(Feature = "encrypt resources [compress]
*.dat", Exclude = false)]
[assembly: Obfuscation(Feature = "encrypt resources [compress]
*.dat", Exclude = false)]
[assembly: Obfuscation(Feature = "encrypt resources [compress]
*.sql", Exclude = false)]
```



Serialization Tuning

Overview

Serialization can be defined as the process of storing the state of an object to a storage medium. There are two main kinds of serialization available in .NET:

- XML serialization
- Binary serialization

This chapter covers the binary serialization.

Binary serialization is often used in application state persistence, offline caches and <u>remoting</u> communication. .NET platform has rich services that allow to perform binary serialization of objects within several lines of code. All serializable classes and structures are marked with [Serializable] (or <Serializable()> in VB.NET) custom attribute, so .NET runtime is aware about serialization-ability of every class and structure.

Binary Serialization and Obfuscation

.NET serialization services use *reflection* to retrieve the data of the serializable objects. That means that obfuscator should take some precautions when it tries to obfuscate serializable elements. Eazfuscator.NET uses the safest approach by default — all serializable classes, structures and fields are automatically excluded from symbol renaming. This guarantees that obfuscation has the minimal impact on application functionality and interoperability. At the same time, this approach has one drawback — all serializable elements are too obviously visible during *reverse engineering*.

Self-Interoperability

Sometimes absolute interoperability of binary serialization is not required for an application. It's a very common situation when application serializes and deserializes the objects by *itself*, so no other applications are meant to have an access to the serialized data. If latter is the case then it is possible to improve the obfuscation of serializable elements in your application by using self-

interoperable serialization.

Non-stable Self-Interoperable Serialization

If serialized data do not leave the boundaries of the application process then nonstable binary serialization can be used instead of fully interoperable one. This kind of serialization is called *non-stable* because the names of serializable elements are changed on every obfuscation of the application. Technically, nonstable serialization is achieved by enabling the renaming of serializable elements, so they are not excluded from symbol renaming process anymore.

To enable non-stable self-interoperable serialization you should apply specially formed attribute to your assembly. In order to do that you can use the instructions below.

Instructions on enabling non-stable self-interoperable serialization

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "rename serializable
symbols", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="rename serializable
symbols", Exclude:=False)>
```

Stable Self-Interoperable Serialization

Stable serialization should be used when the serialized data can leave the boundaries of the application process. This kind of serialization is called *stable* because the names of serializable elements stay the same between the obfuscations of the application. Some obfuscator vendors use the term *incremental obfuscation* when they want to say that symbol names remain the same between several obfuscations. Technically, stable serialization is achieved by encrypting the names of serializable elements with a password. The encryption algorithm is the same as in <u>symbol names encryption</u>.

To enable stable self-interoperable serialization you should apply specially formed attribute to your assembly. In order to do that you can use the instructions below.

Instructions on enabling stable self-interoperable serialization

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "encrypt serializable
symbol names with password 'XXXXXX'", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="encrypt serializable symbol
names with password 'XXXXXX'", Exclude:=False)>
```

Note

Change XXXXXX with your password. Keep the password in secret.



Passwords with a greater length are more preferable than short ones. Longer passwords have a better informational entropy thus greatly improving cryptographic strength of the encrypted data. It's suggested to have a password which at least consists of 8 characters.

Tip



If you use <u>symbol names encryption</u> and want to use the same password for the stable serialization then apply "encrypt serializable symbol names with password" instead of "encrypt serializable symbol names with password 'XXXXXX'" token at the custom attribute shown above.


Debugging

Introduction to Debugging After Obfuscation

Assembly can be easily debugged in Debug project configuration when no obfuscation takes place.

But you may want to debug your assembly in Release configuration when obfuscation does take place. There are not much reasons to do so, however sometimes it may be a life-saver. That's why debugging after obfuscation is supported by Eazfuscator.NET.

The debugging feature of Eazfuscator.NET is turned off by default to allow faster builds and better optimizations. If you want to enable debugging then please follow the instructions below:

Instructions on enabling debug support for an output assembly

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "debug", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="debug", Exclude:=False)>
```

Note



Debugging experience may slightly suffer if used together with <u>code control flow obfuscation feature</u>.

How It Works

Eazfuscator.NET changes assembly contents during obfuscation, so the debugging information gets out of sync when no special provisions are made. If the input assembly contains an applied "debug" attribute that was discussed above then Eazfuscator.NET takes care of debugging information and transforms it according to the applied assembly changes. That's why the debug information is always in sync when debug support is on for a given assembly.

The debug information is stored in .pdb file which is located near the assembly. For example, MyAssembly.dll may have a corresponding MyAssembly.pdb file. When you start a debug session the debugger tries to find the .pdb files for all loaded assemblies. If the right .pdb file is found then you are able to set breakpoints and watch variables in debugger.

Possible Security Risks

.pdb files store the following information:

- The names of source files (including their full paths)
- Line numbers
- Associations between IL instruction offsets, line numbers and source files

The names of source files can be used to find the original class names when corresponding .pdb file is available to a reverse-engineer. So please use .pdb files with care — they can weaken the protection strength.

The security of .pdb files can be improved by using secure debugging.

Tuning

The debugging directive provides a basic experience by default. This is enough to quickly step through your code and spot exceptions. You may also opt in to the improved security and readability of debugging information. In order to do that, please read the notes below.

The full notation of a custom attribute for debugging has the following form:

```
[assembly: Obfuscation(Feature = "debug [flags]", Exclude =
false)]
```

where [flags] is an optional enumeration of flags separated by spaces.

The list of available flags is presented in the table below.

Table 4.8. The list of flags for debugging directive

Flag	Description
secure	Activates secure debugging
relative_file_paths	Instructs to produce <u>relative file paths</u> in resulting .pdb file
nonintrusive	Allows to perform <u>nonintrusive debugging sessions</u> to catch hard-to-reproduce bugs sensitive to time, size or other nonlinear factors

Secure Debugging

Debug information can be a <u>weak point</u> in security of an obfuscated application. But what if you want to get the line numbers and file names without compromising the security? Well, this goal is easily achievable. All you have to do is to supply [secure] flag to an obfuscation attribute like so in C#:

```
[assembly: Obfuscation(Feature = "debug [secure]", Exclude =
false)]
```

or in VB.NET:

```
<<u>Assembly</u>: Obfuscation(Feature:="debug [secure]",
Exclude:=False)>
```

What happens then? Eazfuscator.NET starts to encrypt source file names in .pdb file in the very same way as it does for other items such as class and method names. So, whenever you have <u>symbol names encryption</u> set up for your assembly it will be also applied to the content of .pdb file.

Tip

If you do not want to apply <u>symbol names encryption</u> to the whole assembly then just provide a password as shown below:



Change XXXXXX with your password. Keep the password in secret.

Obviously, the secure debugging makes stepping through the code impossible as a debugger no longer knows how to find the source files: their names are now encrypted and debugger has no means to decrypt them. This is a little sacrifice for the big benefit: all logged exceptions will contain the full information you need in an encrypted and safe form. Class names, method names, argument names, file names and line numbers are all there to help you to precisely locate the problematic code. And now you can safely distribute .pdb files to your customers without the risk of security breach.

Tip



Secure debugging is essential feature when you want to distribute . pdb files to your customers as a sole part of your product. Smaller applications surely can live without it. Applications of a larger scale, notably business-oriented ones, may have the hard times unless secure debugging is employed for the obfuscated code.

Relative File Paths

.pdb file stores the full paths of source files by default. That's an excellent idea for an interactive debugger because it can find the source files easily. However this can be an overhead when you only need stack traces or secure debugging. That's why you may prefer to save some storage bits by using relative file paths. They are shorter, as well as a resulting .pdb file. Stack traces get shorter too, their readibility improves. If you use automated error reporting then you will benefit from smaller workloads. To use relative file paths, please supply [relative_file_paths] flag to an obfuscation attribute like so in C#:

```
[assembly: Obfuscation(Feature = "debug [relative_file_paths]",
Exclude = false)]
```

or in VB.NET:

```
<Assembly: Obfuscation(Feature:="debug [relative_file_paths]",
Exclude:=False)>
```

Debug Renaming

Debug renaming is a special renaming technique for debugging of obfuscated applications. The result of this technique is the presence of human readable symbol names in output assemblies. Such symbol names allow to instantly watch variables and stack traces by an unaided eye.

Debug renaming works by applying _x_ prefix to every renamed class and class member. The name part after prefix equals to original item name.

Instructions on enabling debug renaming for an output assembly

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
```

```
[assembly: Obfuscation(Feature = "debug renaming", Exclude =
false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="debug renaming",
Exclude:=False)>
```

Caution



Debug renaming is exclusively a debugging feature. Never leave this feature enabled for production assemblies, otherwise original symbol names can leak to the outside world.



PEVerify Integration

PEVerify tool allows to check assemblies to determine whether MSIL code and associated metadata meet type safety requirements. .NET verification has the special meaning in security constrained environments such as Windows Phone, WinRT and Silverlight, where running the unverifiable code can produce unexpected results, sometimes rendering the application unworkable.

Eazfuscator.NET modifies metadata and MSIL assembly code during obfuscation. So it may be profitable to ensure that the output assembly still meets type safety requirements.

PEVerify.exe is distributed as a part of .NET SDK which is installed together with Microsoft Visual Studio and can be invoked manually. But Eazfuscator.NET provides a much better option: an ability to automatically invoke PEVerify tool after obfuscation. Please use the instructions below in order to use the latter feature.

Instructions on turning on the PEVerify tool for an output assembly

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "PEVerify", Exclude =
false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
```

Imports System.Reflection

```
<Assembly: Obfuscation(Feature:="PEVerify", Exclude:=False)>
```



Probing Paths

About Probing Paths

Probing paths is a set of places at the file system where Eazfuscator.NET can search for dependencies of input assembly. Eazfuscator.NET is smart enough to deduct these paths from installed assemblies and from project settings, however it might be neccessary to define probing paths manually when some complex scenario is involved.

How to Define Probing Paths?

There are two ways to define probing paths: by declarative obfuscation attributes or by command line option.

Define probing paths by declarative obfuscation attributes (the recommended way)

To define a probing path you should apply an attribute to your assembly. In order to do that you can use the instructions below.

Instructions on defining the probing path by declarative obfuscation attribute

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = @"assembly probing path
C:\Example\Lib")]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="assembly probing path
C:\Example\Lib")>
```

Note



Change C:\Example\Lib with the directory name you want to be probed for assembly dependencies.

Tip

It is recommended to use relative directory paths with <u>script variables</u>:

```
[assembly: Obfuscation(Feature = @"assembly
probing path $(InputDir)\Lib")]
```

Tip

If you want to define several probing paths then just add several attributes:



Define probing paths by command line option

There is a <u>command line</u> option --probing-paths "C:\Example\Lib" which can be specified to achieve this functionality. If you want to define several

probing paths then please use semicolon as a list separator: --probing-paths
"C:\Example\Lib1;C:\Example\Lib2".

Note



Change C:\Example\Lib with the directory name you want to be probed for assembly dependencies.

Warning



Do not put a trailing slash at the end of the path, otherwise the operating system will interpret it as an escape symbol.



Script Variables

Eazfuscator.NET provides support for script variables that can be used to configure <u>probing paths</u>, <u>assemblies merging</u> and <u>assembly embedding</u>. The list of available script variables is presented in the table below.

Variable	Description	Example value
\$(InputDir)	Directory path of the input assembly	C:\Dev\Project1\bin\Release
\$(ProjectDir)	Directory path of the MSBuild project	C:\Dev\Project1
<pre>\$(SolutionDir)</pre>	Directory path of the MSBuild solution	C:\Dev
\$(ConfigurationName)	Name of MSBuild project configuration	Release

Table 4.9. The list of available script variables

Environment variables can be referenced as well. Please use pattern %VARIABLE% to reference a corresponding environment variable.



Chapter 5. Sensei Features

Table of Contents

About Sensei Features Code Inlining Protected Private Visibility Custom Attributes Removal Design-Time Usage Protection Overview How It Works Instructions Tuning Resource Sanitization Introduction Instructions Minification Selective Resource Sanitization Options are Combinable

About Sensei Features

Sensei () is a Japanese word that is used to show respect to someone who has achieved a certain level of mastery in an art form or some other skill. Obfuscation is not exception.

Sensei obfuscation features are powerful and demanding. With a great power comes great responsibility. Not all of them are designed for a common everyday usage. Still, you may *want* them one day. Especially when you work on a licensing code or want to improve obfuscation coverage even further.



Code Inlining

Method bodies can be inlined to their call sites during obfuscation. Please take a look at example (C#):

Example 5.1. Before obfuscation

```
using System;
using System.Reflection;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Inlining test");
        SecretMethod();
    }
    [Obfuscation(Feature = "inline", Exclude = false)]
    static void SecretMethod()
    {
        Console.WriteLine("Secret");
    }
}
```

Example 5.2. After obfuscation

```
using System;
using System.Reflection;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Inlining test");
        Console.WriteLine("Secret");
    }
}
```

Code inlining brings obvious security benefits:

- Once method is inlined, it's no longer a subject of hacker's special attention
- Call site gets larger as it takes inlined instructions of the method. This makes code analysis a harder task for an intruder

Code inlining may be useful in such scenarios as licensing checks and knowhow algorithms.

Instructions on enabling method inlining

- 1. Open the source code of a method you want to inline
- 2. Add a custom attribute as shown below (C#):

```
using System;
using System.Reflection;
class YourClass
{
    [Obfuscation(Feature = "inline", Exclude = false)]
    void YourMethod()
    {
    }
}
```

For Visual Basic .NET:

```
Imports System
Imports System.Reflection
Class YourClass
    <Obfuscation(Feature:="inline", Exclude:=False)>
    Sub YourMethod()
    End Sub
End Class
```



Protected Private Visibility

.NET languages offer a few keywords for visibility control between assemblies, classes and members. For example, C# has public, internal, protected, protected internal and private access modifiers.

Sometimes you may need a special access level: protected private. It exists in C++ but absent in C# and VB.NET. It corresponds to <u>FamANDAssem</u> visibility scope in terms of .NET CLR.

This is indeed a rare requirement. Let's take a look on specific example. Suppose you have a DLL assembly written in C# that does XML serialization for some entities:

Example 5.3. Original code

```
using System;
using System.Xml.Serialization;
// This class is used by System.Xml.Serialization.XmlSerializer.
public class Card
{
    public string ID
    {
        get;
        set;
    }
    protected virtual void Validate()
    {
    }
}
// This class is used by System.Xml.Serialization.XmlSerializer.
public class VerticalCard : Card
{
    public int Height
    {
        get;
        set;
    }
```

```
protected override void Validate()
{
    if (Height <= 0)
        throw new Exception("Vertical card height should be
a positive number greater than zero.");
    }
}</pre>
```

See that Validate method? It won't be renamed after obfuscation despite the wish of developer. But why? The answer is: because assembly is *DLL*, the class is *public* and Validate method is *protected*. This means that Validate method can be reached by a third-party assembly and Eazfuscator.NET leaves its name intact.

That's ok for most situations.

Still, let's imagine that one picky developer decides to rename Validate method whatever it costs. Potential workarounds are:

- Make Validate methods private. Won't work because it wouldn't be possible to override Validate method in VerticalCard class
- Make Card and VerticalCard classes internal. Won't work because XmlSerializer works on public classes only
- Make Validate methods internal. Will work but will break the visibility borders inside the assembly. This may be unfeasible if you work in a team with established responsibility borders between its members
- Make Validate methods protected private. Will perfectly work, but only in C++. C# and VB.NET have no corresponding access modifier
- Move validation away to a separate set of classes. Will work, but it requires code refactoring. It may be risky for a large code base and thus not always suitable

So we are stuck if our code is in C# or VB.NET.

Fortunately Eazfuscator.NET can change the visibility of class members to FamANDAssem level. This is the exact same thing as protected private in C++. Having that, we can now solve the dilemma (C#):

Example 5.4. Modified code to allow family and assembly visibility for specified methods

```
using System;
using System.Xml.Serialization;
using System.Reflection;
// This class is used by System.Xml.Serialization.XmlSerializer.
public class Card
{
    public string ID
    {
        get;
        set;
    }
    [Obfuscation(Feature = "family and assembly visibility",
Exclude = false)]
    protected virtual void Validate()
    {
    }
}
// This class is used by System.Xml.Serialization.XmlSerializer.
public class VerticalCard : Card
{
    public int Height
    {
        get;
        set;
    }
    [Obfuscation(Feature = "family and assembly visibility",
Exclude = false)]
    protected override void Validate()
    {
        if (Height <= 0)</pre>
            throw new Exception("Vertical card height should be
a positive number greater than zero.");
    }
}
```

Once that in place, Validate methods will be renamed and will no longer be

visible to other assemblies.

Instructions on changing visibility to FamANDAssem level for a class member

- 1. Open the source code of a class member that should have a visibility change
- 2. Add a custom attribute as shown below (C#):

```
using System;
using System.Reflection;
class YourClass
{
   [Obfuscation(Feature = "family and assembly visibility",
Exclude = false)]
   protected void YourMethod()
   {
   ...
   }
}
```

For Visual Basic .NET:

Tip



If you change visibility for a virtual method then it is beneficial to ensure that the whole inheritance hierarchy has a corresponding change. This will improve renaming coverage during obfuscation.

Tip



It may be a good idea to turn on <u>code verification</u> for your assembly when visibility changes are applied to ensure that generated code conforms to industrial standard.



Custom Attributes Removal

.NET framework provides a set of custom attributes that allows to describe meta properties of a given class, field, property or method.

For example, Windows Forms and WPF visual designers use System.ComponentModel.DescriptionAttribute to find the textual descriptions for editable class properties. There are other use cases and they are numerous.

Eazfuscator.NET automatically prunes excessive meta attributes whenever possible. However you may prefer to remove all custom attributes with given conditions in some scenarios to achieve better obfuscation coverage.

Instructions on using the custom attributes removal

Suppose we want to remove System.ComponentModel.DescriptionAttribute from every class member of the assembly. Please follow the instructions below to achieve that.

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type *: apply to
member *: remove custom attribute
System.ComponentModel.DescriptionAttribute", Exclude =
false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="Apply to type *: apply to
member *: remove custom attribute
System.ComponentModel.DescriptionAttribute",
Exclude:=False)>
```

Tip

You can use specific conditions to define the scope of custom attributes removal. See <u>conditional obfuscation</u> for details.

Tip



You can specify any class name instead of System.ComponentModel.DescriptionAttribute. Patterns are allowed too, for example: *.DescriptionAttribute

Tip



Attribute literal at the end of the class name can be omitted, e.g. System.ComponentModel.Description will do the job too.



Design-Time Usage Protection

Overview

It is a common scenario when software developers use class libraries authored in-house. Such libraries are not made available to third-parties but they are extensively used throughout the application.

Eazfuscator.NET provides a way to protect such libraries from unsolicited usage by third-parties in design time.

How It Works

Eazfuscator.NET injects special checks into obfuscated assembly and shrinks public API surface when design-time usage protection is enabled. The injected checks ensure that components can only be instantiated at runtime context, thus effectively preventing their unsolicited usage in designer.

Component Designer Suppression

Let's take a look on example. Suppose the application has ContosoWindowsFormsControlLibrary assembly that defines ContosoUserControl UI component. When the solution is in Debug configuration and is not obfuscated, the developer is able to use Toolbox panel and play with ContosoUserControl in Visual Studio designer:



Let's switch the solution to Release configuration and build it with enabled obfuscation and design-time usage protection:



Please note that designer now shows "Design-time usage is not allowed" message. This is expected error message because the control library was obfuscated and protected from usage in designer.

Component designer suppression is automatically applied to Component Model and Windows Forms components defined in class library when design-time protection is on.

Public API Surface Shrink

Public API surface is a set of public classes and their members exposed by a class library. By default, Eazfuscator.NET preserves public API surface of class library so that it can be consumed by other modules. However not all data are needed in runtime. For example, method arguments can be renamed to obfuscated titles without loosing runtime functionality.

What Eazfuscator.NET does is essentially this: it automatically renames method

arguments to obfuscated equivalents when design-time protection is on. This process is called public API surface shrink. It allows to achieve better obfuscation coverage.

Instructions

Please follow the instructions below to enable design-time usage protection for your assembly:

Instructions on enabling design-time usage protection

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "design-time usage
protection", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="design-time usage
protection", Exclude:=False)>
```

Tuning

By default, component designer suppression and public API surface shrink are active when design-time usage protection is enabled for your assembly. You may prefer to turn off the component designer suppression or configure public API shrink options. In order to do that, please read the notes below.
The full notation of a custom attribute for design-time usage protection has the following form:

```
[assembly: Obfuscation(Feature = "design-time usage protection
[flags]", Exclude = false)]
```

where [flags] is an optional enumeration of flags separated by spaces.

The list of available flags is presented in the table below.

Table 5.1.	The list	of flags for	design-time	e usage p	rotection	attribute

Flag	Description		
no_cds	Disables the <u>component designer suppression</u>		
arguments=keep	Disables the method arguments renaming		
arguments=auto	Eazfuscator.NET automatically decides which arguments to rename during <u>public API surface shrink</u> . This is the default setting		
arguments=rename	All method arguments are renamed during public API surface shrink. Note that this seeting may cause troubles with optional parameters if they are referenced by names in source code		

Let's take a look on example.

Example 5.5. Enable design-time usage protection without component designer suppression. Rename all method arguments during public API surface shrink

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "design-time usage protection
[no_cds arguments=rename]", Exclude = false)]
```



Resource Sanitization

Introduction

Resource sanitization feature allows to sanitize and optionally minify the embedded resources of an assembly. Sanitization removes privacy disclosing information such as comments in XML and JSON files, EXIF tag/thumbnail headers in JPEG and PNG files etc.

Eazfuscator.NET supports a finite set of file types which can be sanitized: XML, XSD, XSLT, JSON, PNG and JPEG. All other file types are ignored and kept intact even when there is a directive that instructs to sanitize them.

Let's take a look on example.

Example 5.6. The original XML file

```
<request id="1">
<reference>REQ-D2867DBE</reference>
<destination>Contoso Headquarters</destination>
<!-- For the full list of types see
https://example.net/internal/docs/contoso-protocol-doc.html -->
<type>43</type>
</request>
```

Example 5.7. The sanitized XML file

```
<request id="1">
<reference>REQ-D2867DBE</reference>
<destination>Contoso Headquarters</destination>
<type>43</type>
</request>
```

As you can see, the XML comment was pruned during sanitization.

Instructions

To enable resource sanitization you should apply an attribute to your assembly.

In order to do that you can use the instructions below.

Instructions on enabling resource sanitization

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "sanitize resources",
Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="sanitize resources",
Exclude:=False)>
```

Minification

Assembly resources are not minified by default. If you want to achieve smaller size and better runtime performance of an output assembly then you may consider to turn on the resource minification.

The exact minification effect depends on a file type. For example, all the redundant whitespaces in .xml files are pruned when minification is on.

Example 5.8. The sanitized XML file

```
<request id="1">
<reference>REQ-D2867DBE</reference>
<destination>Contoso Headquarters</destination>
```

```
<type>43</type>
</request>
```

Example 5.9. The sanitized and minified XML file

```
<request id="1"><reference>REQ-D2867DBE</reference>
<destination>Contoso Headquarters</destination><type>43</type>
</request>
```

The [minify] flag turns on the minification when specified, as shown in the sample below:

Example 5.10. Sanitize and minify all resources

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "sanitize resources [minify]",
Exclude = false)]
```

Selective Resource Sanitization

Sometimes it may be beneficial to sanitize just some resources while leaving the others intact. The [exclude] flag can be used in order to do that, as shown in the sample below.

Example 5.11. Sanitize all resources except .png files

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "sanitize resources", Exclude =
false)]
[assembly: Obfuscation(Feature = "sanitize resources [exclude]
*.png", Exclude = false)]
```

It may be profitable to go other way around by explicitly specifying just those resources that should be sanitized. This technique is shown in the sample below.

Example 5.12. Sanitize secret.xml and all .jpg resources; the others are left intact

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "sanitize resources
secret.xml", Exclude = false)]
[assembly: Obfuscation(Feature = "sanitize resources *.jpg",
Exclude = false)]
```

Options are Combinable

The given options can be combined in a free way giving you the power to choose the best combination. If you are not sure which combination to choose then just go with a simplest one: sanitize all resources. If you know what you are doing then you can end up with something like that:

Example 5.13. Advanced resource sanitization configuration

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "sanitize resources", Exclude =
false)]
[assembly: Obfuscation(Feature = "sanitize resources [exclude]
*.png", Exclude = false)]
[assembly: Obfuscation(Feature = "sanitize resources [exclude]
*.jpg", Exclude = false)]
[assembly: Obfuscation(Feature = "sanitize resources [minify]
License.xml", Exclude = false)]
[assembly: Obfuscation(Feature = "sanitize resources [minify]
Help.xml", Exclude = false)]
```



Chapter 6. Virtualization

Table of Contents

Introduction How to Use Code Virtualization How to Use Data Virtualization

Introduction

Code Virtualization

Many of us consider particular pieces of code especially important. May it be a license code check algorithm implementation, an innovative optimization method, or anything else equally important so we would want to protect it by any means possible. As we know, the traditional obfuscation techniques basically do renaming of symbols and encryption, thus leaving the actual algorithms — cycles, conditional branches and arithmetics potentially naked to eye of the skilled intruder.

Here a radical approach may be useful: to remove all the .NET bytecode instructions from an assembly, and replace it with something completely different and unknown to an external observer, but functionally equivalent to the original algorithm during runtime — this is what the code virtualization actually is.

Eazfuscator.NET provides an implementation of custom virtual machine which works atop the .NET virtual machine, using a different virtual instruction set every time you obfuscate your application. This makes the code of a protected algorithm completely bullet-proof and hidden from others. All you need to hide your precious logic is to apply a special attribute to your methods or classes.

See how to use code virtualization

Data Virtualization

Not only the code, but data can be virtualized too. The virtualization changes the way the data are represented in memory and on disk. The resulting data representation is something completely different and unknown to an external observer, but functionally equivalent to the original algorithm during runtime.

See how to use data virtualization

Tip



Code and data always come together so it is beneficial to use both kinds of virtualization to achieve better protection results.



How to Use Code Virtualization

To enable the code <u>virtualization</u> you should apply a custom attribute to your method. In order to do that you can use the instructions below.

Instructions on enabling code virtualization

- 1. Open the source code of a method you want to virtualize
- 2. Add a custom attribute as shown below (C#):

```
using System;
using System.Reflection;
class YourClass
{
   [Obfuscation(Feature = "virtualization", Exclude =
false)]
   void YourMethod()
   {
   ...
  }
}
```

For Visual Basic .NET:

```
Imports System
Imports System.Reflection
Class YourClass
        <Obfuscation(Feature:="virtualization", Exclude:=False)>
        Sub YourMethod()
        End Sub
End Class
```

Note



Virtualization can significantly slow down the speed of code execution, so please use this feature wisely.

Applying Code Virtualization to Multiple Methods at Once

It may beneficial to apply the code virtualization to the whole class or assembly. The <u>conditional obfuscation</u> can be employed to achieve that.

Examples are provided below.

Example 6.1. Virtualize all methods of a class

```
using System.Reflection;
[Obfuscation(Feature = "Apply to member * when method or
constructor: virtualization", Exclude = false)]
class YourClass
{
....}
```

Example 6.2. Virtualize all methods in assembly

```
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type *: apply to
member * when method or constructor: virtualization", Exclude =
false)]
```



How to Use Data Virtualization

To enable the data <u>virtualization</u> you should apply a custom attribute to your field. In order to do that you can use the instructions below.

Instructions on enabling data virtualization

- 1. Locate the declaration of a field you want to virtualize
- 2. Add a custom attribute as shown below (C#):

```
using System;
using System.Reflection;
class YourClass
{
    [Obfuscation(Feature = "virtualization", Exclude =
false)]
    bool yourField;
}
```

For Visual Basic .NET:

```
Imports System
Imports System.Reflection
Class YourClass
    <0bfuscation(Feature:="virtualization", Exclude:=False)>
    Dim yourField As Boolean
```

End Class

Note



Virtualization can significantly slow down the speed of code execution, so please use this feature wisely.

Note



Only simple value types such as int, double etc can be virtualized.



Chapter 7. Troubleshooting

Table of Contents

My application is not working properly after obfuscation. Why does it happen? **Troubleshooting Features Stack Trace Decoding Inspection-Friendly Obfuscation** Preserving the Original Names **Disabling ILDASM Suppression** About InternalsVisibleToAttribute Solution #1. Do not use InternalsVisibleToAttribute at all Solution #2. Swap with EditorBrowsable attribute Solution #3. Hide the warning Solution #4. Ignore the attribute "Option Strict Off" Compatibility for VB.NET Introduction **Compatibility Mode** Instructions **Nonintrusive Debugging** Introduction **Sample Scenario** Warnings and Errors **Disabling Warnings Treat Warnings as Errors Long-Term Compatibility Compatibility Version** Demanding the Specific Version of Eazfuscator.NET Error Codes Knowledge Base EF-1099: Unable to load input assembly, reflection load failed EF-3035: Assembly or part of it is already obfuscated

My application is not working properly after obfuscation. Why does it happen?

The answer is the most probably your application uses *reflection*. Eazfuscator.NET analyzes assemblies for reflection scenarios but analysis algorithm is heuristic – that's why it is not 100% reliable at the moment. So assembly functionality depended on reflection may unintentionally suffer. Eazfuscator.NET is quite smart about data serialization, visualization and other reflection appliances but sometimes it may fail in its decisions.

Note



Theoretically, it is possible to create an obfuscator with 99.99% reliable reflection analysis engine but it is too difficult to solve this problem in a formal way. However Eazfuscator.NET is becoming smarter with each new release so maybe someday it will have a near-formal reflection analyzer.

Any reflection-related problem can be fixed manually with <u>declarative</u> <u>obfuscation using custom attributes</u>.



Troubleshooting Features

Troubleshooting features is a special set of features that helps to solve the most common problems which can appear after obfuscation. This set of features can be accessed by expanding Eazfuscator.NET Assistant window as shown below:



Expanded Eazfuscator.NET Assistant window looks as shown below:



Stack Trace Decoding

Stack trace decoding is a feature which is designed to be used in conjunction with <u>symbol names encryption</u>. Stack trace decoding can be used to decode error stack traces and log files which contain symbol names. To be able to use stack trace decoding, <u>symbol names encryption</u> should be setup for your product first.

Stack trace decoding user interface consists of two main parts: stack trace decoding zone and decoding window. Let's overview them both below.

Stack Trace Decoding Zone

Stack trace decoding zone has the following look:



This zone supports text and text file drag and drop. Also it's possible to doubleclick the zone.

Stack Trace Decoding Window

Stack trace decoding window appears whenever corresponding zone gets drag and dropped with text or text file or double-clicked. This window has the following look:

Stack Trace Decoder – Eazfuscator.NET Assistant		
Enter obfuscated stack trace and password in the fields Please note, to be able to use this feature your assemb	; below and press 'Decode' butt ly must be obfuscated with enc	on. rypted symbol names
Password:		
	Show characters	Remember password
<enter here="" obfuscated="" stack="" text="" trace=""></enter>		*
		-
		*
Help	C	lose Decode

As you can see it's possible to enter password and obfuscated text in the window above. When you enter text with encoded symbol names and corresponding password then you can easily decode it by pressing Decode button.

Please note that you can use drag and drop operations to deliver encoded text from different sources such as text files, fields and editors.



Inspection-Friendly Obfuscation

Inspection-friendly obfuscation is a special mode of obfuscation when you can review the resulting assembly by an unaided eye. This mode can be achieved by temporarily applying one or more inspection-friendly settings to your assembly.

Caution



Please take care when you apply inspection-friendly settings to production assemblies; otherwise original symbol names may leak to the outside world.

Preserving the Original Names

This is the most powerful inspection-friendly setting. It allows to keep the original names for all classes and members while preserving other obfuscation features on.

Instructions on preserving the original names

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "Apply to type *:
renaming", ApplyToMembers = true, Exclude = true)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="Apply to type *: renaming",
ApplyToMembers:=True, Exclude:=True)>
```

Disabling ILDASM Suppression

ILDASM is a special .NET tool which allows to translate the binary assembly files into text files filled with readable IL code.

By default, Eazfuscator.NET automatically adds SuppressIldasmAttribute to the output assembly whenever possible in order to block the possibility of running ILDASM on your obfuscated assembly.

You may prefer to override that behavior and make output assembly friendly to ILDASM.

Instructions on disabling ILDASM suppression

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "ildasm suppression",
Exclude = true)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="ildasm suppression",</pre>

Exclude:=True)>



About InternalsVisibleToAttribute

.NET Framework defines

System.Runtime.CompilerServices.InternalsVisibleToAttribute class which allows to specify that types that are ordinarily visible only within the input assembly are visible to another assembly.

Although this can be useful in some scenarios, it's strongly not recommended to use that feature of .NET Framework in Release builds because it makes the obfuscation theoretically and practically useless.

Eazfuscator.NET gives a warning message when InternalsVisibleToAttribute attribute is defined in input assembly and shuts down all obfuscation features except <u>string encryption</u> to save assembly functionality.

So, how to resolve this? This is not hard, really. The possible solutions are described at the sections below.

Solution #1. Do not use InternalsVisibleToAttribute at all

The recommended way is not to use InternalsVisibleToAttribute at all in Release builds. At the same time it may be profitable to use the attribute in Debug builds: for example, unit test projects rely on InternalsVisibleToAttribute to test the internal parts of the assemblies. This can be achieved by using the following code pattern, effectively applying the attribute in Debug configuration only:

```
#if DEBUG
[assembly: InternalsVisibleTo(<attribute parameters according to
your existing code>)]
#endif
```

Solution #2. Swap with EditorBrowsable attribute

A less known but decent alternative is to use

System.ComponentModel.EditorBrowsableAttribute to mark the classes and members that you want to hide from end-users. Detailed information and sample code are available in corresponding <u>MSDN article</u>.

Solution #3. Hide the warning

It may be profitable to just hide the warning without affecting the behavior of Eazfuscator.NET.

Instructions on hiding the warning about InternalsVisibleToAttribute

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "disable warning EF-4001")]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="disable warning EF-4001")>
```

Solution #4. Ignore the attribute

If you think that previous solutions are not feasible then you can make Eazfuscator.NET to completely ignore InternalsVisibleToAttribute by following the instructions below.

Instructions on making Eazfuscator.NET to ignore InternalsVisibleToAttribute

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for

C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb

3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "ignore
InternalsVisibleToAttribute", Exclude = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
```

```
<<u>Assembly</u>: Obfuscation(Feature:="ignore
InternalsVisibleToAttribute", Exclude:=False)>
```



"Option Strict Off" Compatibility for VB.NET

Introduction

By default, the Visual Basic .NET compiler does not enforce strict data typing and uses late binding to access methods, properties and fields of the classes. This is somehow simplifies the coding for certain kind of people but it has several implications:

- The resulting application has a greater chance of errors during runtime
- Late binding is considerably slower than a direct strongly-typed access
- Late binding may break after obfuscation

That's why a very good advice for Visual Basic .NET programmers is to use Option Strict On for their programs. Unfortunately, it is not always possible due to legacy code or personal long-term preferences that are hard to change.

Compatibility Mode

Eazfuscator.NET provides a special compatibility mode that allows to workaround the issues with late binding. It comes at the expense of a lower obfuscation coverage but your code remains functional and runs perfectly after obfuscation.

Instructions

Instructions on activating Option Strict Off compatibility mode for VB.NET

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.vb. You may prefer to use another name instead of ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
```

```
<Assembly: Obfuscation(Feature:="vb option strict off
compatibility", Exclude:=False)>
```



Nonintrusive Debugging

Introduction

When <u>debugging</u> is on, not only the corresponding .pdb file gets processed during obfuscation but some of code optimizations are turned off to improve the interactive debugging experience for the resulting assembly. This leads to a bit different timing and size characteristics of the resulting code. The micro changes in characteristics may mask or unmask the defects in your code, especially those tied to unpredictable factors such as time. The multithreaded deadlock is a canonical example of such defect.

Sample Scenario

Let's take a look on concrete example. Suppose your obfuscated application suffers from intermittent multithreaded deadlock. You want to fix that. Everything you currenty have is an obfuscated .exe file without debugging information.

The next logical step is to find the source file names and line numbers where deadlock occurs. Being a quick and somewhat lazy person, you temporarily disable obfuscation for your assembly. Then you build it just to find out that deadlock does not occur anymore.

You think: "Hm... probably the issue is tied to that exact obfuscated file somehow". You enable obfuscation again for that assembly. Then you build it. The deadlock shows itself again.

You think: "Ok, let's try <u>debug</u> directive and then attach debugger to the process". You write:

```
[assembly: Obfuscation(Feature = "debug", Exclude = false)]
```

Then you build and run your project again only to find out that deadlock mysteriously does not occur anymore. How is that possible that a debug directive affects the runtime behavior?
The answer is debug directive does not affect the runtime behavior. It just induces slight changes in code speed and size. It turns out that those slight changes are enough to mask the multithreading defect in the code.

No problem, just use the nonintrusive flag:

```
[assembly: Obfuscation(Feature = "debug [nonintrusive]", Exclude
= false)]
```

It minimizes the amount of changes applied by Eazfuscator.NET to the assembly that are required to provide the debugging functionality. In this way, the assembly characteristics stay the same even when debugging is on. Now you get a reproducible defect together with debug information. So you build your project again and run it. The deadlock is reproduced.

What you do next is attach debugger to your running deadlocked process. Launch Visual Studio and use Debug \rightarrow Attach to Process... (**Ctrl+Alt+P**) menu item. Then, select your process from the list.

The next step is to freeze all running threads with Debug \rightarrow Break All (**Ctrl+Alt+Break**) menu item. Then take a look at Debug \rightarrow Windows \rightarrow Threads (**Ctrl+D,T**) window and go through threads one by one while examing their call stacks. Once you find the suspected call stacks please write down the file names and line numbers of possible deadlock locations.

You now have the information to proceed with a fix to your source code.

Note



Stepping through the code (e.g. interactive debugging) is hugely limited when nonintrusive flag is specified. Essentially you can only attach debugger to a process and freeze the threads as a last resort solution.



Warnings and Errors

Eazfuscator.NET can produce warning and error messages. Generally, every warning and error message can be identified by a special identifier which has the form EF-XXXX. Example identifier: EF-4001.

Disabling Warnings

Please follow the instructions below to disable a specific warning.

Instructions on disabling a specific warning

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "disable warning EF-XXXX")]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
```

```
<Assembly: Obfuscation(Feature:="disable warning EF-XXXX")>
```



Note

EF-XXXX must be changed with an identifier of a warning you want to disable.

Treat Warnings as Errors

Please follow the instructions below to treat all warnings as errors.

Instructions on making Eazfuscator.NET to treat all warning as errors

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "treat all warnings as
errors")]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="treat all warnings as
errors")>
```

Tip

There is a <u>command line</u> option --warnings-as-errors all which can be specified to achieve this functionality.

Sometimes it can be useful to treat just a specific warning as an error. Please follow the instructions below to achieve this.

Instructions on making Eazfuscator.NET to treat a specific warning as an error

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer

to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb

3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "treat warning EF-XXXX as
error")]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="treat warning EF-XXXX as
error")>
```



Tip

There is a <u>command line</u> option --warnings-as-errors EFxxxx which can be specified to achieve this functionality.



Note

EF-XXXX must be changed with an identifier of a warning you want to treat as an error.



Long-Term Compatibility

Eazfuscator.NET constantly evolves from one version to another. That's why some provisions should be made to ensure the successful integration of Eazfuscator.NET with your project over the time. This chapter describes all aspects related to compatibility in long-term perspective.

Compatibility Version

Compatibility version option instructs Eazfuscator.NET to mimic its corresponding version from the past. Why it should be used? The answer is very straighforward: usually Eazfuscator.NET is integrated with a project just once; after that the user of Eazfuscator.NET expects that integration will continue to flawlessly work whatever future version of Eazfuscator.NET is installed.

<u>Eazfuscator.NET Assistant</u> automatically adds a compatibility version option -v to obfuscation command line in post-build event of a project:

>ject-configuration "\$(ConfigurationName)" -nnewline-flush					
		-			

The value of compatibility version should be equal to the version of Eazfuscator.NET that was used during the project integration stage. This guarantees that the future versions of Eazfuscator.NET will mimic the integrated version, thus delivering a solid upgrade path.

Important



If you manually invoke Eazfuscator.NET from <u>command line</u> or from custom script then please ensure that compatibility version is supplied with -v command line option.

Demanding the Specific Version of Eazfuscator.NET

Sometimes it may be useful to restrict the version of Eazfuscator.NET to work with. For example, some previous version of Eazfuscator.NET contained a bug which was later fixed, and some of your colleagues may still have that old version. It is not always possible to explicitly force the team members to upgrade Eazfuscator.NET to a newer version, that's why an ability to restrict the version of Eazfuscator.NET would be a good way to achieve this.

Please follow the instructions below to instruct Eazfuscator.NET to fail when its version is lower than required.

Instructions on forcing Eazfuscator.NET to fail when its version is lower than a given value

- 1. Open obfuscatable project inside the IDE
- 2. Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "require eazfuscator.net
version >= X.Y")]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="require eazfuscator.net
version >= X.Y")>
```

Note



Change X.Y with Eazfuscator.NET version number.

Important



Support of this syntax appeared since Eazfuscator.NET 3.2. The syntax is ignored by previous versions of Eazfuscator.NET. If you have an absolute necessity to cover the previous versions too then please use the batch script approach shown in the section below.

Demanding the specific version from batch script

Batch scripts usually reside in .bat or .cmd files, but can also be coded in postbuild event of a project.

The following batch script can be used to demand a specific version of Eazfuscator.NET which is greater or equal to a given value X.Y:

```
if /I "$(ConfigurationName)" NEQ "Release" goto SkipObfuscation
Eazfuscator.NET.exe --check-version GEQ X.Y >NUL 2>NUL
if %ErrorLevel% NEQ 0 (
    echo Eazfuscator.NET X.Y or higher is not installed on this
machine. Obfuscation failed.
    REM The line below resets error level to 0. Uncomment it if
you want to force script to continue execution when no required
version of Eazfuscator.NET is present
    REM verify >NUL 2>NUL
) else (
    Eazfuscator.NET.exe "$(TargetPath)" --msbuild-project-path
"$(ProjectPath)" --msbuild-project-configuration
"$(ConfigurationName)" --msbuild-project-platform
"$(PlatformName)" --msbuild-solution-path "$(SolutionPath)" -n -
-newline-flush -v <compatibility_version>
)
:SkipObfuscation
```

Note



Change X.Y with Eazfuscator.NET version number.



Error Codes Knowledge Base

EF-1099: Unable to load input assembly, reflection load failed

Error EF-1099^[1] occurs when input assembly or one of its dependencies can not be loaded. Possible solutions for this problem:

- Specify a probing path
- Put a missing assembly near the input file

EF-3035: Assembly or part of it is already obfuscated

Error EF-3035 occurs when input assembly is already obfuscated. Eazfuscator.NET stops processing and exits with error code 1, thus indicating the error condition.

You may prefer to just skip the processing of an assembly without raising the error. In order to do that, please follow the instructions below.

Instructions on making Eazfuscator.NET to ignore EF-3035 error

- 1. Open obfuscatable project inside the IDE
- Add new source file to the project and call it ObfuscationSettings.cs (for C#) or ObfuscationSettings.vb (for Visual Basic .NET). You may prefer to use another name instead of ObfuscationSettings.cs or ObfuscationSettings.vb
- 3. Fill ObfuscationSettings.cs with the following content (C#):

```
using System;
using System.Reflection;
[assembly: Obfuscation(Feature = "ignore error EF-3035",
StripAfterObfuscation = false)]
```

For Visual Basic .NET, fill ObfuscationSettings.vb with the following content:

```
Imports System
Imports System.Reflection
<Assembly: Obfuscation(Feature:="ignore error EF-3035",
StripAfterObfuscation:=False)>
```

^[1] Eazfuscator.NET before version 3.2 produced error code EF-E-1099 instead of EF-1099.



Chapter 8. Best Practices

Table of Contents

Introduction General Best Practices Keeping the Balance Human Factors Keeping It Simple The Paralysis of Simplicity

Introduction

The chapter about best practices shares the most essential knowledge for getting the best results with Eazfuscator.NET.

Let's start with general best practices.



General Best Practices

Please carefully read the list below. Not only this will bring you better obfuscation results but also this will help you to avoid possible pitfalls.

- Use minimal but feasible visibility of the classes and their members
- Avoid using System.Runtime.CompilerServices.InternalsVisibleToAttribute attribute for production assemblies
- Use CLSCompliantAttribute(true) to mark CLS-compliant assemblies. If you produce library or component which can be used by your customers then you should mark the assembly as CLS-compliant to ensure successful interoperability. Eazfuscator.NET internally uses the value of CLSCompliantAttribute to make a decision on the level of obfuscation to apply. Eazfuscator.NET turns off all CLS-incompatible obfuscation features when the input assembly is marked as CLS-compliant
- Try to keep reasonable small number of the assemblies in your product. It leads to a greater code integration inside one assembly, thus it is harder to deduct the logic that is contained inside the assembly. Assembly number reduction can be achieved at the application design phase. Alternatively, <u>assemblies merging feature</u> can be used for that purpose
- Use <u>symbol names encryption</u> in production releases of your product. It allows you to decode error stack traces and log files
- It is highly recommended to <u>sign your assemblies with a strong name</u>. The signed assemblies are better protected against integrity violations
- It is highly recommended to use <u>resource encryption</u> to hide assembly resources from prying eyes
- Please consider to <u>virtualize</u> the methods that you want to hide by any means possible
- Test your product after obfuscation

Additional for Visual Basic .NET

• It is highly recommended to use Option Strict On to achieve good obfuscation results and avoid common problems. Alternatively, you may use a <u>compatibility mode</u> to workaround possible issues



Keeping the Balance

Security always works against speed and usability. That's why it's important to keep the right balance between them.



There is no such thing as absolute security. *There is always a tradeoff between security, usability and speed.* So, the best solution is to provide the point of balance between those ends. Eazfuscator.NET provides the balanced solution: your code will be in safety and it will work <u>fast</u>.

At the same time, obfuscation does not stop well-planned attacks on a particular method or a class. For example, it is not a so big deal to try to crack the licensing portion of your code. Going beyond the obfuscation, <u>code virtualization</u> <u>technology</u> covers this scenario making it extra hard and costly for an intruder.

You can just look inside C:\Program Files

(x86)\Eazfuscator.NET\Eazfuscator.NET.exe file with a decompiler to get an example of Eazfuscator.NET vision. Yes, you see some calls or instructions but you are unable to comprehend inner constructs, crack or resell the product.

So, as you can see, the obfuscation is the art of balance.

Human Factors

While Eazfuscator.NET does its best to automatically provide the balanced

solution, there are situations when it may get distorted by human factors.

Let's overview some human factors that may come into play.

"I Need to Virtualize Everything"

The factor comes into play when a person decides that <u>code virtualization</u> is the only reliable protection technique. While virtualization is indeed a decent protection scheme, it has its drawbacks. The assembly gets larger and virtualized code becomes 50x slower.

In general, virtualization should not be applied to every class and every method. Instead, it should be carefully used to protect the most important parts of your application. For example, licensing algorithms are perfect target for virtualization.

"I Need to Rename Everything"

The factor applies when a person wants to improve obfuscation coverage. This is a noble aim and it is perfectly achievable by following <u>general best practices</u>.

But what the person should beware of and avoid is trying to shoot himself in the foot by renaming the unrenamable.

For example, the person can decide that he needs to get serializable classes renamed while they are used by binary serializer to work with interchangeable file formats. This is a conflict situation: either class names are kept intact or the file serialization gets broken.

"I Need to Put Everything into Single File"

The factor comes into play when a person decides that his whole application should be deployed as a single file. Single file deployment may be a requirement for portable applications despite the fact of known performance implications.

If the person is not bound to portable application requirements then he should thoroughly ask himself whether single file deployment is really needed. More often than not, single file deployment is not a necessity. Once person drops "The Single File" mantra, he becomes more open to careful and responsible usage of assembly <u>embedding</u> and <u>merging</u> features. As a result, the larger applications will likely have better runtime performance and memory footprint together with more granular and simpler build process during development.

The rule of a thumb goes as follows:

- Keep your assembly sizes below 10 MB, preferably below 5 MB
- Neither embed nor merge third-party libraries (A good exception from this rule is licensing libraries)

Of course, it all depends on your particular scenario and you should not take these suggestions as a dogma.



Keeping It Simple



Eazfuscator.NET provides a simple integration path with Visual Studio projects as shown in <u>Quick Start</u> guide.

The provided integration is light yet powerful. What it essentially does is register Eazfuscator.NET in post-build event of the project:

Application Build	Configuration: N/A v	Platfor <u>m</u> : N/A	Ŷ
Build Events	Pre-build event command line:		
Debug		~	
Resources			
Services	<	>	
Settings		Edit Dre-build	
Reference Paths	Post-build event command line:	Luit ric build in	
Signing	if /l "\$(ConfigurationName)" == "Release" Eazfus	cator.NET.exe "\$(TargetP 🔺	
Security			
Publish		Υ	
Code Analysis	<	>	
		Edit Post-b <u>u</u> ild	
	Run the post-build event:		

Once registered, Eazfuscator.NET comes into action and obfuscates the assembly every time the project is built in Release configuration.

Eazfuscator.NET messages can be found at Output Window of Visual Studio with View \rightarrow Output (**Ctrl+W,O**):

Output	
Show output from: Build 🔹 🚽 🖆 🖆 🔁	
<pre>1> Rebuild All started: Project: IncrediFlow, Configuration: Release Any CPU 1> IncrediFlow -> C:\Users\gl\documents\visual studio 2013\Projects\IncrediFlow\IncrediFlow\bin\Release\IncrediFlow 1> Obfuscating assembly 'IncrediFlow.exe' 1> Done ============= Rebuild All: 1 succeeded, 0 failed, 0 skipped =============</pre>	ow.exe

That's it. The rest of the things just work.

The Paralysis of Simplicity

The paralysis of simplicity is a problem that may apply to a person who experience Eazfuscator.NET for the first time:

- "How does Eazfuscator.NET know which key to use for the assembly signing?"
- "How is it possible it just works everytime?"
- "What should I do to make Eazfuscator.NET work with MSBuild?"
- and so on.

The general answer to these quesions: *it just works* and there is no need to worry about it; just give it a try.



Chapter 9. Deployment

Table of Contents

About Eazfuscator.NET Deployment Microsoft Installer (MSI) NuGet Package Manager Command Line Interface

About Eazfuscator.NET Deployment

Eazfuscator.NET can be deployed in various ways. This chapter describes all available deployment methods.



Microsoft Installer (MSI)

MSI acronym stands for Microsoft Installer. This is the default and recommended deployment method for Eazfuscator.NET.

Eazfuscator.NET website provides a <u>download page</u> where MSI setup file can be retrieved. Once you have the MSI file you can install it on your machine.

Another popular scenario is deploying MSI through the global policy objects (GPO) in <u>Active Directory (AD)</u>. AD MSI deployment tends to be a preffered way of installing software in middle and large software houses. More information on AD software distribution is available in corresponding <u>knowledge</u> <u>base article</u> from Microsoft.

MSI technology is proven and reliable. If you have any doubts about what deployment method to choose for Eazfuscator.NET then please strongly consider MSI.



NuGet Package Manager

<u>NuGet</u> is the package manager for the Microsoft development platform. It allows to quickly add a library or a tool to your project.

Eazfuscator.NET is not exception and can be added to your solution via NuGet too.

Why ever bother to use NuGet when we have MSI? A good question. Actually, there are no many reasons to do so. However some usage scenarios *can not* work with MSI. Let's take a look at the list:

- Hosted TFS Build Agents may disallow software installs. This depends on TFS infrastructure administration policies. They may prohibit the installation of third-party software on host machines
- <u>Visual Studio Online</u> does not allow software installs

This is the point when NuGet becomes useful for Eazfuscator.NET deployment. So, let's add Eazfuscator.NET NuGet package to your solution:

1. Ensure gapotchenko.com NuGet repository is configured

To do that, please go to Tools \rightarrow NuGet Package Manager menu and click Package Manager Settings item:

TO	DLS TEST ANALYZE WINDOW	HELP		
ø®	Attach to Process	Ctrl+Alt+P	1	
*#	Connect to Database			
*#	Connect to Server			
₿ [₩]	Add SharePoint Connection			
-	Connect to Windows Azure			
	SQL Server	•		
÷	Code Snippets Manager	Ctrl+K, Ctrl+B		
	Choose Toolbox Items			
	Add-in Manager			
	NuGet Package Manager	•	C:\	Package Manager Console
¢	Extensions and Updates		Ħ	Manage NuGet Packages for Solution
	Create GUID		Ф	Package Manager Settings
	PreEmptive Dotfuscator and Analytics			Ŵ
	Spy++ (хб4)			
	Eazfuscator.NET Assistant			
œ	WCF Service Configuration Editor			
	External Tools			
	Import and Export Settings			
	Customize			
ø	Options			

Options window will show:

			Options	?	×
Search Options (Ctrl+E)	ρ	Availabl	le <u>p</u> ackage sources:	₽ × +	+
 Source Control Text Editor Debugging Performance Tools Database Tools F# Tools HTML Designer 	^	♥ nug htt ♥ gap htt	get.org ps://www.nuget.org/api/v2/ potchenko.com p://www.gapotchenko.com/nuget		
 Office Tools Package Manager General Package Sources SQL Server Tools Text Templating 	1	Machine ✓ Mie htt	e-wide package sources: crosoft and .NET ps://www.nuget.org/api/v2/curated-feeds/m	icrosoftdotnet/	
 Web Performance Test Tools Windows Forms Designer Workflow Designer xUnit.net 	~	<u>N</u> ame: <u>S</u> ource:	gapotchenko.com http://www.gapotchenko.com/nuget	<u>U</u> pda	ite
				OK Canc	el

Please ensure there is a registered package source with gapotchenko.com name and http://www.gapotchenko.com/nuget URL as shown above.

2. Install Eazfuscator.NET NuGet package for your solution

Open context menu for solution and click on Manage NuGet Packages for Solution... item:



NuGet packages management window will open:

	Incred	liFlow.sln - Manage NuC	Get Packages	? ×
Installed packages	Stable Only	Sort by: Most Dov	vnloads •	Search Online (Ctrl+E)
 Online All nuget.org gapotchenko.com Updates 	Eazfusca Official p the obfu:	tor.NET (Official) ackage for Eazfuscator.NET, scator for .NET platform.	Install	Created by: Gapotchenko Id: Gapotchenko.Eazfuscator.NET Version: 4.3 Last Published: 18.03.2014 Downloads: 0 License View License Project Information Description: Official package for Eazfuscator.NET, the obfuscator for .NET platform. The main purpose of obfuscator is to protect intellectual property of the software. Tags: obfuscator Eazfuscator.NET Dependencies: No Dependencies
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.		1		
Settings				Close

Please ensure you are going to install Eazfuscator.NET (Official) package from gapotchenko.com source and press Install button.

What happens to my projects after Eazfuscator.NET NuGet package has been installed in solution?

The projects remain intact, e.g. they don't become protected or unprotected. They behave the same way they did before.

So, the basic workflow of project integration remains <u>the same</u>:

- Project can be protected by dropping to the green zone of Eazfuscator.NET Assistant
- Project can be unprotected by dropping to the red zone of Eazfuscator.NET Assistant

You can launch Eazfuscator.NET Assistant from Desktop, or you can invoke it from Package Manager Console, it does not matter. It just works no matter what deployment method is in use currently.

OK. If Eazfuscator.NET NuGet package does not affect projects' behavior then what it does?

Well, it does obfuscation for projects that are considered as protected, e.g. were dropped to the green zone *any* time before or after. This is the very same behavior of <u>MSI-deployed</u> Eazfucator.NET.

The big difference is this: Eazfuscator.NET is now able to travel to the cloud or a hosted TFS build agent and work there. Just like any source part of your project. Thanks to NuGet, Eazfuscator.NET does not need to be installed on machine.

Important



Eazfuscator.NET does not support NuGet <u>package restore</u>. Instead, Eazfuscator.NET package should be stored together with your sources.


Command Line Interface

Eazfuscator.NET can be run from command line. Although this is uncommon usage pattern it can be useful for some projects.

The command-line interface of Eazfuscator.NET is accessed by invoking Eazfuscator.NET.exe executable. Installer adds Eazfuscator's install directory to PATH system variable, so Eazfuscator.NET.exe can be invoked from any location within the file system and you don't have to worry how to find the file.

Here is the full list of available options (it can be retrieved by running **Eazfuscator.NET.exe --help** at the command line):

```
Usage: Eazfuscator.NET.exe [options] <input file 1> [input file
21 ...
Generic options:
  -? [ --help ] Produce detailed help message for
available options.
  --version
                       Print version string.
  -n [ --nologo ]
                       Suppress logo message.
Configuration options:
  -o [ --output ] arg Put obfuscated assembly to the
specified output file. If this option is not specified then
output assembly
                            overwrites the input file. The
option cannot be specified when multiple input files are given.
  -k [ --key-file ] arg If this option is specified then
obfuscated assembly will be signed with a key from specified
file.
                            PLEASE NOTE: obfuscated assembly
that had a strong name before obfuscation MUST BE resigned to
work properly;
                            otherwise it will not be able to
load. Also note that only assemblies with strong name can be
resigned -
                            assemblies without strong name are
not affected.
  -c [ --key-container ] arg If this option is specified then
obfuscated assembly will be signed with a key from specified
container.
```

This option cannot be used with 'key-file' option. -q [--quiet] Do not print any information and diagnostic messages. Advanced features: --decode-stack-trace-with-password arg Decodes encrypted stack trace. Password for decryption is given with this option. Encrypted stack trace must be fed to standard input stream of the application. Decrypted stack trace will be fed to standard output stream. --error-sandbox arg Runs application given as argument in exception sandbox. Every unhandled exception is caught by sandbox environment. This feature is useful when obfuscated application cannot be started and bails out with default unexpected error window. --ensure-obfuscated Checks the input file and ensures it is obfuscated. Integration options: --msbuild-project-path arg MSBuild project path. --msbuild-project-configuration arg MSBuild project configuration name. --msbuild-project-platform arg MSBuild project platform name. --msbuild-solution-path arg MSBuild solution path. Protect project. Project --protect-project is obfuscated by Eazfuscator.NET on every build when protection is active. This option should be used with 'msbuild-project-path' option. --unprotect-project Remove project protection. This option should be used with 'msbuild-project-path' option. Compatibility options: -v [--compatibility-version] arg A version of Eazfuscator.NET to be compatible with. --check-version Instructs to check the installed version of Eazfuscator.NET and return the result as exit code. This option cannot be combined with other options. (To get more help, please try to use it)

Advanced configuration options: --probing-paths arg Probing paths separated by semicolon. --warnings-as-errors arg A list of warnings to treat as errors separated by comma. Example: --warnings-as-errors EF-4001, EF-4002 To treat all warning as errors please put an argument 'all' to this option: --warnings-as-errors all --configuration-file arg Configuration file in C# or VB.NET format with a list of assembly attributes for obfuscation. Please refer to documentation for configuration syntax. -s [--statistics] Produce obfuscation statistics report. --newline-flush Flushes output messages with new line (CR/LF) symbols. This feature is useful when integrating with third-party IDEs.



Glossary

C

Common Intermediate Language (CIL)

During compilation of .NET programming languages, the source code is translated into CIL code rather than platform or processor-specific object code. CIL is a CPU- and platform-independent instruction set that can be executed in any environment supporting the .NET framework. CIL code is verified for safety during runtime, providing better security and reliability than natively compiled binaries. [WikiCIL]

E

Encryption

In cryptography, encryption is the process of transforming information (referred to as plaintext) using an algorithm (called cipher) to make it unreadable to anyone except those possessing special knowledge, usually referred to as a key. The result of the process is encrypted information (in cryptography, referred to as ciphertext). In many contexts, the word encryption also implicitly refers to the reverse process, decryption (e.g. "software for encryption" can typically also perform decryption), to make the encrypted information readable again (i.e. to make it unencrypted). [WikiENC]

I

Intellectual Property (IP)

In law, intellectual property (IP) is an umbrella term for various legal entitlements which attach to certain names, written and recorded media, and inventions. The holders of these legal entitlements may exercise various exclusive rights in relation to the subject matter of the IP. The adjective "intellectual" reflects the fact that this term concerns a process of the mind. The noun "property" implies that ideation is analogous to the construction of tangible objects. Consequently, this term is controversial. [WikiIP]

Р

Peephole optimization

In compiler theory, peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognising sets of instructions that don't actually do anything, or that can be replaced by a leaner set of instructions. [WikiPeepholeOptimization]

R

Reflection

In computer science, reflection is the process by which a computer program of the appropriate type can be modified in the process of being executed, in a manner that depends on abstract features of its code and its runtime behavior. Figuratively speaking, it is then said that the program has the ability to "observe" and possibly to modify its own structure and behavior. [WikiReflectionCS]

Remoting (.NET Remoting)

.NET Remoting is a Microsoft application programming interface (API) for interprocess communication released in 2002 with the 1.0 version of .NET Framework. .NET Remoting allows an application to make an object (termed remotable object) available across remoting boundaries, which includes different appdomains, processes or even different computers connected by a network. [WikiRemoting]

Reverse engineering

Reverse engineering is the process of discovering the technological principles of a device or object or system through analysis of its structure, function and operation. It often involves taking something (e.g. a mechanical device, an electronic component, a software program) apart and analyzing its workings in detail, usually to try to make a new device or program that does the same thing without copying anything from the original. [WikiRE]



Bibliography

[WikiCIL] Wikipedia contributors. *Common Intermediate Language*. Wikipedia, The Free Encyclopedia. 11 December 2007 07:20 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?</u> <u>title=Common_Intermediate_Language&oldid=177166979</u>. Accessed December 22, 2007.

[WikiENC] Wikipedia contributors. *Encryption*. Wikipedia, The Free Encyclopedia. November 29, 2007, 12:02 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?title=Encryption&oldid=174579616</u>. Accessed December 12, 2007.

[WikiIP] Wikipedia contributors. *Intellectual property*. Wikipedia, The Free Encyclopedia. December 10, 2007, 21:30 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?</u> title=Intellectual_property&oldid=177069633 . Accessed December 12, 2007.

[WikiObCode] Wikipedia contributors. *Obfuscated code*. Wikipedia, The Free Encyclopedia. December 10, 2007, 15:46 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?title=Obfuscated_code&oldid=177002435</u>. Accessed December 12, 2007.

[WikiReflectionCS] Wikipedia contributors. *Reflection (computer science)*. Wikipedia, The Free Encyclopedia. 19 December 2007 00:14 UTC. *Available at:* http://en.wikipedia.org/w/index.php? title=Reflection_%28computer_science%29&oldid=178837188. Accessed December 22, 2007.

[WikiRE] Wikipedia contributors. *Reverse engineering*. Wikipedia, The Free Encyclopedia. November 30, 2007, 14:10 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?</u> title=Reverse_engineering&oldid=174829884 . Accessed December 12, 2007.

[WikiRemoting] Wikipedia contributors. *.NET Remoting*. Wikipedia, The Free Encyclopedia. May 19, 2009, 13:20 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?title=.NET_Remoting&oldid=290938735</u>. Accessed June 8, 2009.

[WikiPeepholeOptimization] Wikipedia contributors. *Peephole optimization*. Wikipedia, The Free Encyclopedia. 28 December 2010 19:38 UTC. *Available at:* <u>http://en.wikipedia.org/w/index.php?</u> <u>title=Peephole_optimization&oldid=404686923</u>. Accessed March 5, 2012.