User's Guide

· Community Edition ·

Version 3.0

Manual Version 3.0-072005

www.preemptive.com

TRADEMARKS

Dotfuscator, DashO, Overload-Induction, the PreEmptive Solutions logo, the Dotfuscator logo, and the DashO logo are trademarks of PreEmptive Solutions, LLC

.NET™ , MSIL™, and Visual Studio .NET™ are trademarks of Microsoft, Inc.

Java™ is a trademark of Sun Microsystems, Inc.

All other trademarks are property of their respective owners.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD CONTAIN TYPOGRAPHIC ERRORS AND/OR TECHNICAL INACCURACIES. UPDATES AND MODIFICATIONS MAY BE MADE TO THIS DOCUMENT AND/OR SUPPORTING SOFTWARE AT ANY TIME.

PreEmptive Solutions, LLC has intellectual property rights relating to technology embodied in this product. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and/or other countries.

This product is distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product may be reproduced in any form by any means without prior written authorization of PreEmptive Solutions, LLC.

Introduction 8
About the Documentation 9
Why You Need Obfuscation 9
Goal of Obfuscation 9
Benefits of Dotfuscator 10
Dotfuscator significantly enhances code security. 10
Dotfuscator significantly decreases the size of .NET programs. 10
Dotfuscator improves run-time performance. 10
The Dotfuscator Solution 11
Obfuscation 15
Renaming 15
String Encryption 15
Control Flow Obfuscation 15
Enhanced Overload Induction 16
Pruning 16
Assembly Linking 17
Watermarking 17
Incremental Obfuscation 17
Debugging Obfuscated Code 17
Dotfuscator FAQ 18
Conclusion 18
New in Dotfuscator 3.0 19
Overview of Changes 19
Declarative Obfuscation 19
Finding Referenced Assemblies 19
Assembly Signing 20
Build Events 20
Assembly Linking 20
PreMark Watermarking 20
Support for .Net Framework 2.0 20
Enabling and Disabling Features 20
Getting Started With Dotfuscator 22
Standalone GUI Quick Start 22

Step 1 – Launching Dotfuscator GUI 22
Step 2 – Creating a Default Project 22
Step 3 – Building the Project 23
Step 4 – Configuring the Project 23
Step 5 – Rebuilding the Project 25
Step 6 – Browsing the Output 25
Next Steps 26
Command Line Quick Start 26
Using Existing Configurations 26
Using Command Line Switches Only 26
Using Advanced Command Line Switches 27
Observing and Understanding Obfuscated Output 27
Step 1 – Using a Disassembler 27
Step 2 –Decompiling 30
Configuration File Reference 32
Version 32
Property List and Properties 33
Global Section 34
Library Global Option 35
Verbose, Quiet, and Investigate Global Options 35
Input Assembly List 35
Library Mode By Assembly 36
Declarative Obfuscation By Assembly 37
Enabling or Disabling Declarative Obfuscation 38
Stripping Declarative Obfuscation Attributes 38
Output Directory 38
Temp Directory 39
Obfuscation Attribute Feature Map 39
Renaming Section 40
Renaming Options 40
Renaming Exclusion List 41
Output Mapping File 41
A Note about XML Configuration Files 42
Summary 43

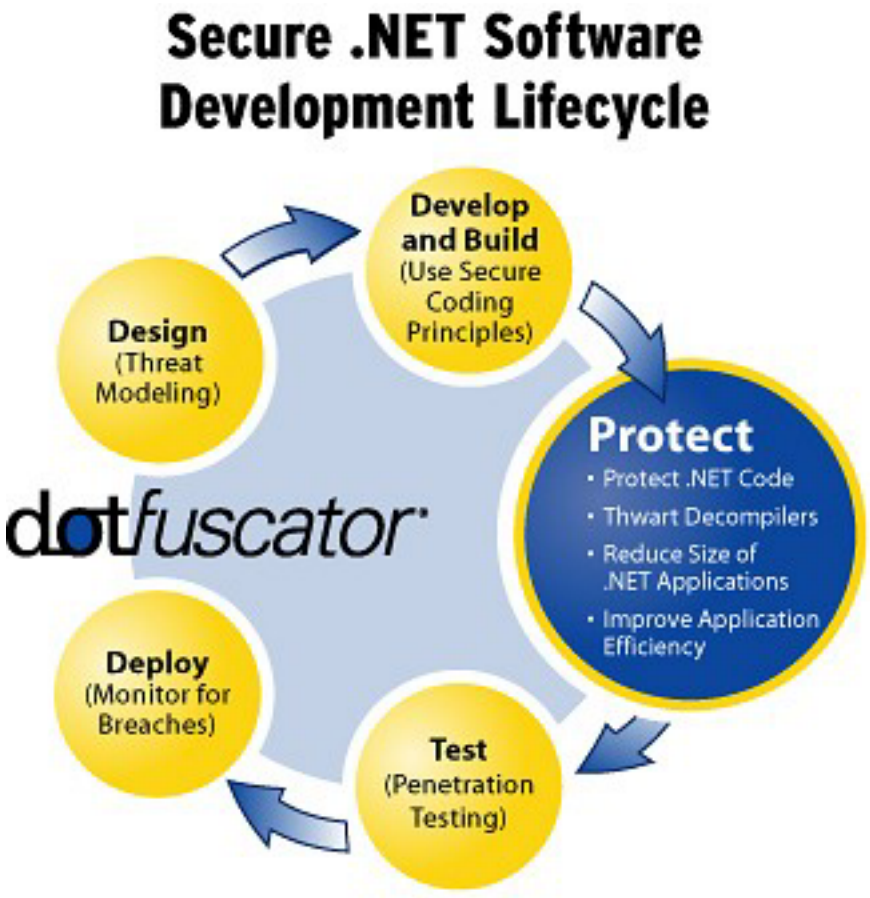
Identifier Renaming 44
Class Renaming Options 44
Keepnamespace Renaming Option 45
Keephierarchy Renaming Option 45
Full Class Renaming (default) 45
Overload-Induction Method Renaming 46
Renaming Exclusion Rules 47
Excluding Namespaces 48
Excluding Types 48
Excluding Methods 50
Excluding Fields 51
Excluding By Custom Attribute 52
Excluding Assemblies 53
Excluding Modules 53
The Map File 54
Control Flow Obfuscation 55
User String Encryption 56
Pruning 57
Linking 58
Watermarking 59
Advanced Topics 60
P/Invoke Methods 60
Dotfuscating Assemblies with Managed Resources 60
Dotfuscating Multi-module Assemblies 60
Dotfuscating Strong Named Assemblies 61
Manually Resigning after Obfuscation 61
Dotfuscating 64 Bit Assemblies 62
Reflection and Dynamic Class Loading 62
Declarative Obfuscation Using Custom Attributes 63
System.Reflection.ObfuscateAssemblyAttribute 63
System.Reflection.ObfuscationAttribute 64
Enabling or Disabling Declarative Obfuscation 65
Stripping Declarative Obfuscation Attributes 65
Using Feature Map Strings 66

Friend Assemblies 66
Finding External Tools....66
Using the Command Line Interface 68
Command Line Option Summary 68
Traditional Options 68
Extended Options....70
Supplementing or Overriding a Configuration File from the Command Line 71
Saving a Configuration File from the Command Line....73
Launching the GUI from the Command Line 74
GUI Reference 75
Using the Visual Studio Integrated UI....75
Using the Standalone GUI 75
GUI Orientation....76
Working with Projects 78
Working with Input Assemblies 80
The Setup Tab 82
The Options Tab....84
Configuring 85
Building the Project 85
Viewing Project Files and Reports 87
The Help Menu....88
GUI Configuration Editors....89
The Renaming Editor 89
The Control Flow Editor 92
The String Encryption Editor 92
The Removal Editor 92
The Linking Editor 92
The PreMark Editor....92
Using the Graphical Rules Editing Interface 93
Selecting Individual Elements 93
Creating Custom Rules....97
Using Declarative Obfuscation with Rules....103
Previewing Rules....104
Reference 106

dotfuscator_v2.1.dtd 106
dotfuscatorMap_v1.1.dtd 106
Registering Dotfuscator 106
Samples 108

Introduction

Dotfuscator is an obfuscator, pruner, linker, and watermarker for .NET applications. It is a key component of a Secure .NET Software Development Life Cycle Process. It adds a new level of protection and application efficiency to any .NET application.



This section is an overview of the benefits of using Dotfuscator.

About the Documentation

This documentation is explicitly for Dotfuscator Community Edition users. In places it mentions features available in Dotfuscator Professional and Standard Editions, but not to the same level of detail. For a full version of the Dotfuscator Professional Edition documentation, see the online user's guide.

Why You Need Obfuscation

Programs written for .NET are easy to reverse engineer. This is not in any way a fault in the design of .NET; it is simply a reality of modern, intermediate-compiled languages. .NET uses expressive file syntax for delivery of executable code, or MSIL (Microsoft Intermediate Language). Being much higher-level than binary machine code, the intermediate files are laden with identifiers and algorithms that are immediately observable and ultimately understandable. After all, it is obviously difficult to make something easy to understand, flexible, and extendable while simultaneously hiding its crucial details.

So anyone with a freely available .NET decompiler can easily reverse engineer code. Suddenly, software licensing code, copy protection mechanisms, and proprietary business logic are much more available for all to see - whether it's legal or not. Anyone can peruse the details of software for whatever reason they like. They can search for security flaws to exploit, steal unique ideas, crack programs, etc.

This need not be a risk or a showstopper. Organizations concerned with their intellectual property on the .NET platform need to understand that there is a solution to help thwart reverse engineering. Obfuscation is a technique that provides for seamless renaming of symbols in assemblies as well as other tricks to foil decompilers. Properly applied obfuscation can increase the protection against decompilation by many orders of magnitude, while leaving the application intact.

Goal of Obfuscation

The goal of obfuscation is to create confusion. As confusion builds, the ability of the human mind to comprehend multi-faceted intellectual concepts deteriorates. Note that this precept says nothing about altering the forward (executable) logic – only representing it incomprehensibly. When a well-written obfuscator tool goes to work on readable program instructions, a likely side effect is that the output will not only confuse a human interpreter, it will break a decompiler. While the forward (executable) logic has been preserved, the reverse semantics have been rendered non-deterministic. As a result, any attempt to reverse-engineer the instructions to a "programming dialect" like C# or VB will likely fail because the translation is ambiguous. Deep obfuscation creates a myriad of decompilation possibilities, some of which might

produce incorrect logic if recompiled. The decompiler, as a computing machine, has no way of knowing which of the possibilities could be recompiled with valid semantics. Humans write and employ decompilers to automate decompilation algorithms that are too challenging for the mind to follow. It is safe to say that any obfuscator that confuses a decompiler will pose even more of a deterrent to a less-capable human attempting the same undertaking.

Primitive obfuscators essentially rename identifiers found in the code to something that is unreadable. They may use hashing techniques or arithmetically offset the character set to unreadable or unprintable characters. While superficially effective, it is obvious that these are reversible techniques, and as such, are hardly protective. PreEmptive's obfuscation tools go far beyond this primitive renaming approach with additional ingenious ways of "creating confusion" that make it nearly impossible (and certainly not worth the effort) to reverse-engineer someone else's intellectual property.

Benefits of Dotfuscator

Dotfuscator is a post-development recompilation system for .NET applications. It analyzes applications and makes them smaller, faster, and harder to reverse-engineer. In short, it makes them ***better***.

Dotfuscator significantly enhances code security.

Dotfuscator includes state-of-the-art technology to protect .NET applications - securing the important intellectual property contained within.

Dotfuscator significantly decreases the size of .NET programs.

Dotfuscator Professional Edition analyzes your application and figures out exactly which parts of your program you're really using. From there it can strip out those pieces, leaving you with the smallest executable possible.

Dotfuscator improves run-time performance.

By removing unneeded program elements and renaming identifiers to small names, Dotfuscator can actually speed up programs.

In addition, Dotfuscator provides many other benefits, such as linking many assemblies into one and watermarking your application with hidden information.

The Dotfuscator Solution

Today, most commercial obfuscators employ a renaming technique that applies trivial identifiers. Typically, these can be as short as a single character. As the obfuscator processes the code, it selects the next available trivial identifier for substitution. This seemingly simple renaming scheme has an advantage over hashing or character-set offset: it cannot be reversed. While the program logic is preserved, the names become nonsense. At this point, we have frustrated human understanding to a large degree. Faced with identifiers like **`a`**, **`t.bb()`**, **`ct`**, and **`2s(e4)`**, it is a stretch to translate the semantic purpose to be concepts like **`invoiceID`**, **`address.print()`**, **`userName`**, and **`deposit(amount)`**. Nevertheless, the program logic can be reverse engineered.

A deeper form of obfuscation uses Overload Induction, a patented algorithm devised by PreEmptive Solutions. Trivial renaming is still used; however, a crafty twist is added. Method identifiers are maximally overloaded after an exhaustive scope analysis. Instead of substituting one new name for each old name, Overload Induction will rename as many methods as possible to the same name. After this deep obfuscation, the logic, while not destroyed, is beyond comprehension. The following simple example illustrates the power of the Overload Induction technique:

Original Source Code Before Obfuscation

```
private void CalcPayroll(SpecialList employeeGroup) {
   while (employeeGroup.HasMore()) {
        employee = employeeGroup.GetNext(true);
        employee.UpdateSalary();
        DistributeCheck(employee);
    }
}
```

Reverse-Engineered Source Code After Overload Induction Dotfuscation

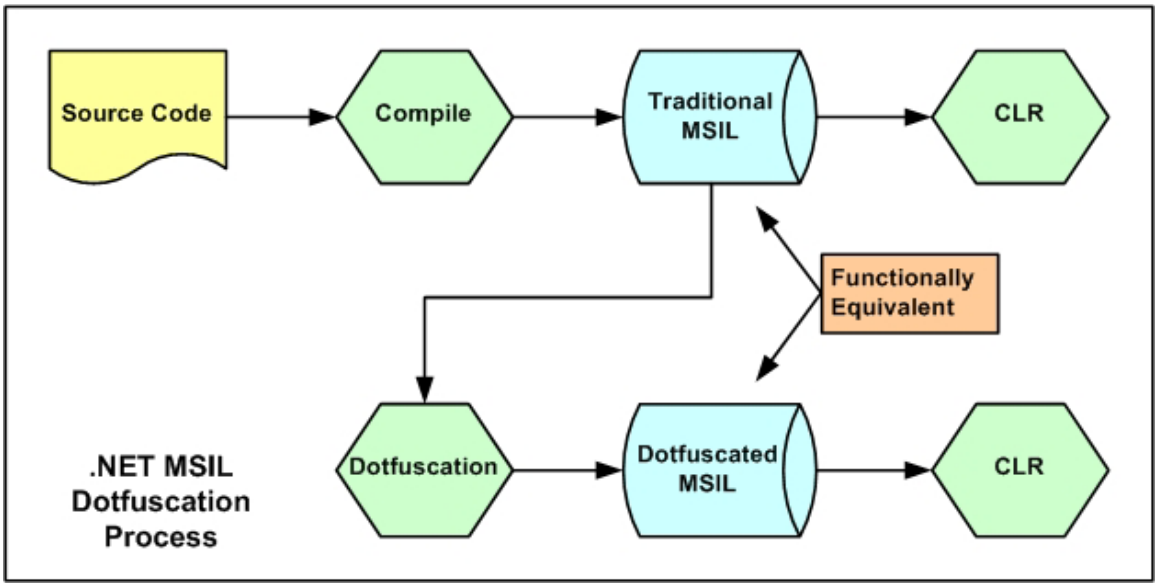
```
private void a(a b) {
    while (b.a()) {
        a = b.a(true);
        a.a();
        a(a);
    }
}
```

The example shows that the obfuscated code is more compact. A positive side effect of renaming is size reduction. For example, if a name is 20 characters long, renaming it to **`a()`** saves a lot of space (specifically 19 characters). Renaming also saves space by conserving string heap entries. Renaming everything to "**`a`**" means that "**`a`**" is stored only once, and each method or field renamed to "**`a`**" can point to it. Overload Induction

enhances this effect because the shortest identifiers are continually reused. Typically, an Overload Induced project will have up to 70% of the methods renamed to `a()`.

Dotfuscator removes debug information and non-essential metadata from a MSIL file as it processes it. Aside from enhancing protection and security, this also contributes to the size-reduction of MSIL files.

It is important to understand that obfuscation is a process that is applied to compiled MSIL code, not source code. The development environment and tools will not change to accommodate renaming. Source code is never altered, or even read, in any way. Obfuscated MSIL code is functionally equivalent to traditional MSIL code and will execute on the Common Language Runtime (CLR) with identical results. (The reverse, however, is not true. Even if it were possible to decompile strongly obfuscated MSIL, it would have significant semantic disparities when compared to the original source code.) The following illustration shows the flow of the Dotfuscation process.



PreEmptive Solutions has been protecting and improving intermediate-compiled software since 1996, beginning with its DashO tools for Java. Its products have enjoyed market-dominance due to their power, versatility, and unique patented features.

Dotfuscator is offered as a family of tools that allow you to take advantage of the powerful .NET platform without worrying about protecting your intellectual property. Dotfuscator has three editions:

 Dotfuscator Community Edition is a free version that offers basic obfuscation. Its main purpose is to rename identifiers, raising the bar against reverse engineering. Dotfuscator Community Edition incorporates advanced technologies to facilitate this protection. In addition, some size reduction is achieved (as a result of renaming to trivial identifiers).

Dotfuscator Community Edition 3.0 does not:

- Operate separately from Visual Studio, meaning it cannot be used in a commercial build environment.
- Rename .NET 2.0 generic types and methods.
- Dotfuscate managed code written for Microsoft Office integration.
- Dotfuscate managed code meant to run inside Microsoft SQL Server 2005.
- Support Library mode other than as a global option (Library mode is either on or off for all input assemblies).
- Support Declarative Obfuscation settings other than as global options (Declarative Obfuscation settings are either on or off for all input assemblies).
- Support Managed C++ applications.

If you need to go beyond these limitations, contact PreEmptive Solutions for more information about Dotfuscator Professional Edition 3.0.

Features available to Dotfuscator Community Edition licensees are flagged with a  icon.

 Dotfuscator Standard Edition is an entry level commercial obfuscator that has all the advanced renaming capabilities of Dotfuscator Community Edition. On top of that, it adds capabilities such as Managed C++ support, support for generating debug symbol files for obfuscated assemblies, the ability to decode obfuscated stack traces, and better reporting. Features available to Dotfuscator Standard Edition licensees are flagged with a  icon.

 Dotfuscator Professional Edition includes the features of Dotfuscator Standard Edition and much more. It is the industry leading obfuscator, and is designed for organizations that produce commercial and enterprise applications. Dotfuscator Professional Edition provides superior protection to foil decompilation, advanced size reduction to conserve memory and improve load times, deep Visual Studio.NET integration for seamless configuration, incremental obfuscation to release patches, watermarking to uniquely tag assemblies, and phone and email technical support. Features available to Dotfuscator Professional Edition licensees are flagged with a  icon.

Here is a side by side comparison of Dotfuscator product features:

Feature	Professional Edition	Standard Edition	Community Edition
Deep Visual Studio Integration	✔		
Compacting/Pruning	✔		
Comprehensive support for the .NET Compact Framework	✔		
Control Flow Obfuscation	✔		
Enhanced Overload Induction	✔		
Incremental Obfuscation	✔		
Seamless obfuscation of satellite DLLs	✔		
String Encryption	✔		
Assembly Linking	✔		
Support for Pre and Post Build Events	✔		
Automatic Strong Named Assembly Re-signing	✔		
Integration with MSBuild	✔		
Watermarks Software	✔		
Can Run Independent of Visual Studio.NET	✔	✔	
Various Renaming Schemes	✔	✔	
PDB Debugging Support	✔	✔	
Rename Prefix	✔	✔	
Supports Managed C++ assemblies	✔	✔	
Automated Stack Trace Translation	✔	✔	
XML/HTML Report Files	✔	✔	
Support for Generic Types and Methods	✔	✔	
Integration with Build Scripts	✔	✔	
Cross Assembly Obfuscation	✔	✔	✔
Removal of Unused Metadata	✔	✔	✔
Renaming	✔	✔	✔
Support for Declarative Obfuscation	✔	✔	✔

Obfuscation

CE STD PRO Dotfuscator provides all "normal" obfuscation methodologies in addition to many unique ones. No obfuscation technology is 100 percent secure. As with other obfuscators, Dotfuscator makes life more difficult for decompilers and disassemblers; it does not claim 100 percent protection.

Renaming

CE STD PRO Dotfuscator includes a patented Overload-Induction™ renaming system. Dotfuscator renames all classes, methods, and fields to compact (usually one character) names. Dotfuscator's renaming system also provides inherent obfuscation properties far greater than typical renaming systems. Many "obfuscating" renamers rely on renaming to unprintable characters or confusing sequences in attempts to foil decompiled output. Unlike these attempts, Dotfuscator's renaming system is irreversible, and results in more compact applications.

String Encryption

PRO Crackers will frequently search for specific strings in an application to locate strategic logic. For example, someone looking to bypass a registration and verification process can search for the string displayed when the program asks the user for a serial number. When the attacker finds the string, he can look for instructions near it and alter the logic. String Encryption makes this much more difficult to do, because the attacker's search will come up empty. The original string is nowhere to be found in the code. Only its encrypted version is present.

Control Flow Obfuscation

PRO This process synthesizes branching, conditional, and iterative constructs that produce valid forward (executable) logic, but yield non-deterministic semantic results when decompilation is attempted. Control Flow obfuscation produces spaghetti logic that can be very difficult for a cracker to analyze. Consider the following example processed by Dotfuscator Professional Edition:

Original Source Code Before Obfuscation
© 2001, Microsoft Corporation
(Snippet from WordCount.cs C# example code)

```
public int CompareTo(Object o) {
    int n = occurrences - ((WordOccurrence)o).occurrences;
    if (n == 0) {
```



```
        n = String.Compare(word, ((WordOccurrence)o).word);
    }
    return(n);
}
```

Reverse-Engineered Source Code
After Control Flow Obfuscation
By Dotfuscator Professional Edition

```
public virtual int _a(Object A_0) {
    int local0;
    int local1;
    local0 = this.a - (c) A_0.a;
    if (local0 != 0) goto i0;
    goto i1;
    while (true) {
       return local1;
       i0: local1 = local0;
    }
    i1: local0 = System.String.Compare(this.b, (c) A_0.b);
    goto i0;
}
```

Enhanced Overload Induction

PRO Dotfuscator Professional Edition extends Overload-Induction™ by allowing a method's return type or a field's type to be used as a criterion in determining method or field uniqueness. This feature can allow up to 15 percent more redundancy in method and field renames. In addition, since overloading on method return type or field type is typically not allowed in source languages (including C# and VB), this further hinders decompilers.

Pruning

PRO Smaller applications install faster, load faster and may even run faster. Dotfuscator Professional Edition's pruning feature removes unused code. This multi-pass iterative process detects and removes unused types, methods, and fields. Obviously, if an application is using well designed, general purpose libraries, it will be ripe for such pruning. Amazing space reduction can be achieved, conserving computing resources and reducing instantiation times.

Assembly Linking

Assembly linking, sometimes called merging, is the ability to merge multiple assemblies into one or more output assemblies. This can make an application even smaller and simplify deployment scenarios. When combined with obfuscation and pruning, assembly linking provides a powerful packaging solution for .NET applications.

Watermarking

Watermarking is one method that can be used to track unauthorized copies of your software back to the source. With PreMark™, Dotfuscator Professional Edition version 3.0 adds support for watermarking .NET assemblies. Watermarking can be used to unobtrusively embed data such as copyright information or unique identification numbers into a.NET application without impacting its runtime behavior.

Incremental Obfuscation

Incremental obfuscation is an advanced feature of particular interest to enterprise development teams maintaining an integrated application environment. By generating name mapping records during an obfuscation run, obfuscated API names can be reapplied and preserved in successive runs. A partial build can be done with full expectation that its access points will be renamed the same as a prior build. As a result, the distributed patch files integrate into the previously deployed system without a hitch.

Debugging Obfuscated Code

One major drawback of obfuscation is that the task of maintaining and troubleshooting an obfuscated application becomes more difficult. In a well obfuscated application, the names of all types, methods, and fields are changed from their carefully thought out, informative names into meaningless, semi-random names.

This impacts the usefulness of bug reports sent in from the field in the form of stack traces. A stack trace is essentially a listing of where in the program a problem occurred. The list includes the names embedded in the program (e.g. method and type names). A stack trace from an obfuscated program will contain the obfuscated names, and will thus be very difficult to read by support personnel and developers.

Dotfuscator Standard and Professional Editions address this challenge by providing a tool that automatically decodes obfuscated stack traces using the renaming map file. Given an obfuscated stack trace, the tool replaces the obfuscated names with the original names and displays the results. The tool is built into Dotfuscator Standard and

Professional Editions. PreEmptive also offers this same function in a standalone tool called Lucidator, which can be licensed by group.

Even in-house debugging of obfuscated applications need not be as complex as it once was. Dotfuscator Standard and Professional Editions have the ability to output debugging symbol files for obfuscated applications (in Microsoft's PDB format) that correspond as closely as possible to the original symbol files output by the compiler. Using these files, developers can use a debugger to step through an obfuscated assembly and see the original source code.

Dotfuscator FAQ

Frequently Asked Questions regarding Dotfuscator may be viewed online at www.preemptive.com/products/dotfuscator/FAQ.html

Conclusion

Dotfuscator .NET Obfuscator is completely non-intrusive and requires no changes to source code, making it the most natural way to protect intellectual property on the .NET platform. In addition, many of the elements within the Dotfuscator family of .NET packaging tools make unprecedented steps towards smaller and faster .NET applications. Dotfuscator employs a mix of established and novel algorithms to allow customers to produce the best .NET applications possible.

New in Dotfuscator 3.0

This section is an overview of the major changes and new features in Dotfuscator Version 3.0. It is organized by new feature or change and contains links to related documentation elsewhere in the user guide.

Overview of Changes

Declarative Obfuscation

The .NET Framework version 2.0 provides two new custom attributes designed to make it easier to automatically obfuscate assemblies without having to set up configuration files.

For a description, see Declarative Obfuscation Using Custom Attributes.

For Standalone UI support, see The Setup Tab.

For XML Configuration File, see Declarative Obfuscation By Assembly, Obfuscation Attribute Feature Map.

Finding Referenced Assemblies

Dotfuscator version 3.0 has a modified algorithm for finding assemblies referenced by your input assemblies. Now, in addition to using rules similar to those of the CLR, it also looks in the same locations that Visual Studio looks for references.

Assembly Signing

PRO Dotfuscator can now re-sign the output assemblies after processing is complete. It does this by using the sn tool that ships with the .NET Framework SDK. This eliminates the need for a separate "post obfuscation" task in your build scripts.

Build Events

PRO Dotfuscator now allows you to specify a program to run before and after obfuscation.

Assembly Linking

PRO You can use Dotfuscator Professional Edition to link or merge multiple input assemblies into one or more output assemblies.

PreMark Watermarking

PRO Dotfuscator Professional Edition adds support for watermarking .NET assemblies.

Support for .Net Framework 2.0

CE STD PRO Dotfuscator 3.0 fully supports applications compiled on the 2.0 version of the .NET Framework. Including:

- Generic Types and Methods
- Friend Assemblies

CE Note that while Dotfuscator Community Edition can process assemblies containing generic types and methods, it does not support renaming generic types and methods.

Enabling and Disabling Features

CE STD PRO Previous versions of the Dotfuscator standalone GUI allowed you to check a box on each tab (e.g. renaming, string encryption, removal, etc.) that allowed you to disable or enable the feature during a build. All of these enable/disable checkboxes have been moved to the Options tab and are now properties in the "Feature" category.

Getting Started With Dotfuscator

This section shows you how to obfuscate code using Dotfuscator.


Some of the features discussed in this section are available only in Dotfuscator Professional Edition; those sections may be skipped if they do not apply to your version of Dotfuscator.

The types of obfuscation that you will be exposed to include: renaming; pruning; control flow and string encryption.

Standalone GUI Quick Start

This section shows you how to use Dotfuscator's standalone GUI. For a complete guide to Dotfuscator's User Interface, see GUI Reference.

Step 1 – Launching Dotfuscator GUI

- Click Start | Programs | PreEmptive Solutions | Dotfuscator Professional Edition 3.0 | Dotfuscator.
- The About Dotfuscator Professional dialog appears. Click or press any key to proceed or wait 5 seconds.

Step 2 – Creating a Default Project

- Select "Create New Project" and click OK. The Dotfuscator main project window appears with the Input tab selected.
- Here you will select the assembly you would like to obfuscate.
- Click Browse.
- Browse to:

```
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition
3.0\samples\cs\GettingStarted\bin\Debug
```

and select GettingStarted.exe.

- Click Open. The path to the executable is now shown in the list box under Input Files.
- Select File | Save Project to save the project.
- In the Save Project dialog, navigate to:

```
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition
3.0\samples\cs\GettingStarted\
```

- In the file name field, enter "Dotfuscator.xml" and click Save.

Step 3 – Building the Project

- Click on the Build tab. The Destination Directory is populated by default as: ${configdir}\Dotfuscated. Note: ${configdir} is a variable that holds the path to your Dotfuscator configuration file.
- Now the project is ready to be obfuscated. Click Build. Wait a few moments as Dotfuscator builds an obfuscated version of the HelloWorld application. The obfuscated assembly is now stored in:

```
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition
3.0\samples\cs\GettingStarted\Dotfuscated
```

You can now go to the output tab and browse the obfuscated symbols, or look at the renaming map file (map.xml) that Dotfuscator created in the output directory, or even run the obfuscated application if you wish. Out of the box, with little configuration, your application has been successfully obfuscated.

Next, with a little more configuration, we can use some of Dotfuscator's more powerful features.

Step 4 – Configuring the Project

- Click the Options tab.
- Set the "Build Progress" property to "Verbose". This will cause Dotfuscator to provide additional information about its progress during execution at the bottom of the build tab.
- Set the "Emit Debugging Symbols" to "Yes". Setting this property tells Dotfuscator to create a symbol file in PDB format for each output assembly. Debuggers can then use these files to provide useful information in a debugging session. Typically, they contain information like line numbers, source file names, and local variable

names. The PDB files are placed in the output directory along with the output assemblies.

- Set the values for "Disable Renaming", "Disable Control Flow", "Disable String Encryption", and "Disable Removal" to "No". You have fine grained control over which transforms Dotfuscator will apply to your assemblies; these are the features we will configure and use in the next steps.
- Click the Rename tab. Renaming will obscure code by renaming methods and fields to names that are not understandable. It is turned on by default and items must be selected to be excluded from the renaming process. For this application, you do not need to exclude anything.
- Click the Renaming Options sub tab. Check "Use Enhanced Overload Induction". This feature can allow up to 15 percent more redundancy in method and field renames. Since overloading on method return type or field type is typically not allowed in source languages (including C# and VB), this further hinders decompilers.
- Note that the "Map Output File" text box has defaulted to: ${configdir}\Dotfuscated\Map.xml
- Check "Output as HTML" to get a useful report containing renaming information and statistics on your application. This report will output into the same directory as the map file. The default location is ${configdir}\Dotfuscated\Map.html.
- Click the Control Flow tab. Control Flow obfuscation synthesizes branching, conditional, and iterative constructs (such as if, for, and while) that produce valid forward (executable) logic, but yield non-deterministic semantic results when decompilation is attempted. In other words, the code runs as before, but decompilers cannot reproduce the original code.
- Click the String Encryption tab. String Encryption will scramble the strings in your application. For example, someone looking to bypass your registration and verification process can search for the string where your program asks the user for a serial number. When they find the string, they can look for instructions near it and alter the logic. String Encryption makes this much more difficult to do, because their search will come up empty.
- String encryption is inclusion based, so you must mark the assembly's checkbox at the root of the tree shown in the left pane to include all methods in the input assembly.
- Click the Removal tab. Removal instructs Dotfuscator to detect and remove unused types, methods, and fields. This can potentially save significant space in the final application.
- For removal to work, Dotfuscator needs to know what the entry points to the application are. In this case, the entry point is the standard "Main" method, and Dotfuscator is able to determine this without extra configuration.

Step 5 – Rebuilding the Project

- Click Build; the project is now ready to be re-obfuscated. As before, the obfuscated assembly is stored in:

```
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition 3.0\samples\cs\GettingStarted\dotfuscated
```

Step 6 – Browsing the Output

- Click the Output tab. Now, you can navigate a tree that shows how Dotfuscator obfuscated your code.
- Expand the root tree and all sub-trees. Notice the blue diamond shaped icons. These are the renamed methods and fields. The parents of each of these icons display their original names. Dotfuscator has renamed each method and field to make it almost impossible to decipher the purpose of each method. This can severely impact the process of reverse engineering the code.

- Notice the currently highlighted SaySomething and set_Name methods, as well as the Name property. Dotfuscator has detected that these items are not being used in this application. As a result, Dotfuscator's Pruning feature is able to remove them, resulting in a more compact application.

Next Steps

Now that you have successfully obfuscated using the GUI, you can see how to use the command line interface to do the same things. Or you can examine the obfuscated output assembly in detail and see how effective the obfuscation was.

Command Line Quick Start

This section demonstrates how to use the command line interface to obfuscate using the same settings as in the Standalone GUI Quick Start.

You can start Dotfuscator from the command line using the following syntax:

```
dotfuscator [options] [configfile]
```

The command line options are documented in the Command Line Options Summary.

The configuration file is an XML document that specifies various options for Dotfuscator. When you ran the standalone GUI and filled in the various dialogs, you were populating a configuration file. All elements of the configuration file are documented in the Configuration File Reference.

Using Existing Configurations

You can feed previously created configuration files into the command line tool. For example, using the configuration file that you created in the last section, you can obfuscate from the command line using this command:

```
dotfuscator Dotfuscator.xml
```

Using Command Line Switches Only

Alternatively, you can Dotfuscate on the command line without a configuration file because most of the configuration options are available as command line switches. To get powerful obfuscation for our example assembly, all you need to do is specify your input assembly.

```
dotfuscator /in:GettingStarted.exe
```

- The "in" switch lets you specify a list of input assemblies separated by commas.
- Because the input assembly is an EXE, the application type is automatically set for maximal obfuscation. DLLs default to library mode.
- By default, the output assembly is placed in a subdirectory of the working directory called "Dotfuscated". You can override this with the "out" command line switch.
- By default, renaming is enabled and the renaming map file is called "map.xml". It is also placed in the "Dotfuscated" subdirectory. You can override this with the "mapout" switch.
- By default, string encryption, control flow, and removal are turned on.

Using Advanced Command Line Switches

If you want to run the obfuscator from the command line with the same options that you set in the standalone GUI in the previous section, you need a command like this:

```
dotfuscator /in:GettingStarted.exe /debug:on /v /enha:on /
```

- The "in" option is as before.
- The "v" option runs Dotfuscator in verbose mode.
- The "debug" option tells Dotfuscator to generate the debugging symbols for the obfuscated output assemblies.
- The "enha" option turns on Enhanced Overload Induction.

Observing and Understanding Obfuscated Output

Step 1 – Using a Disassembler

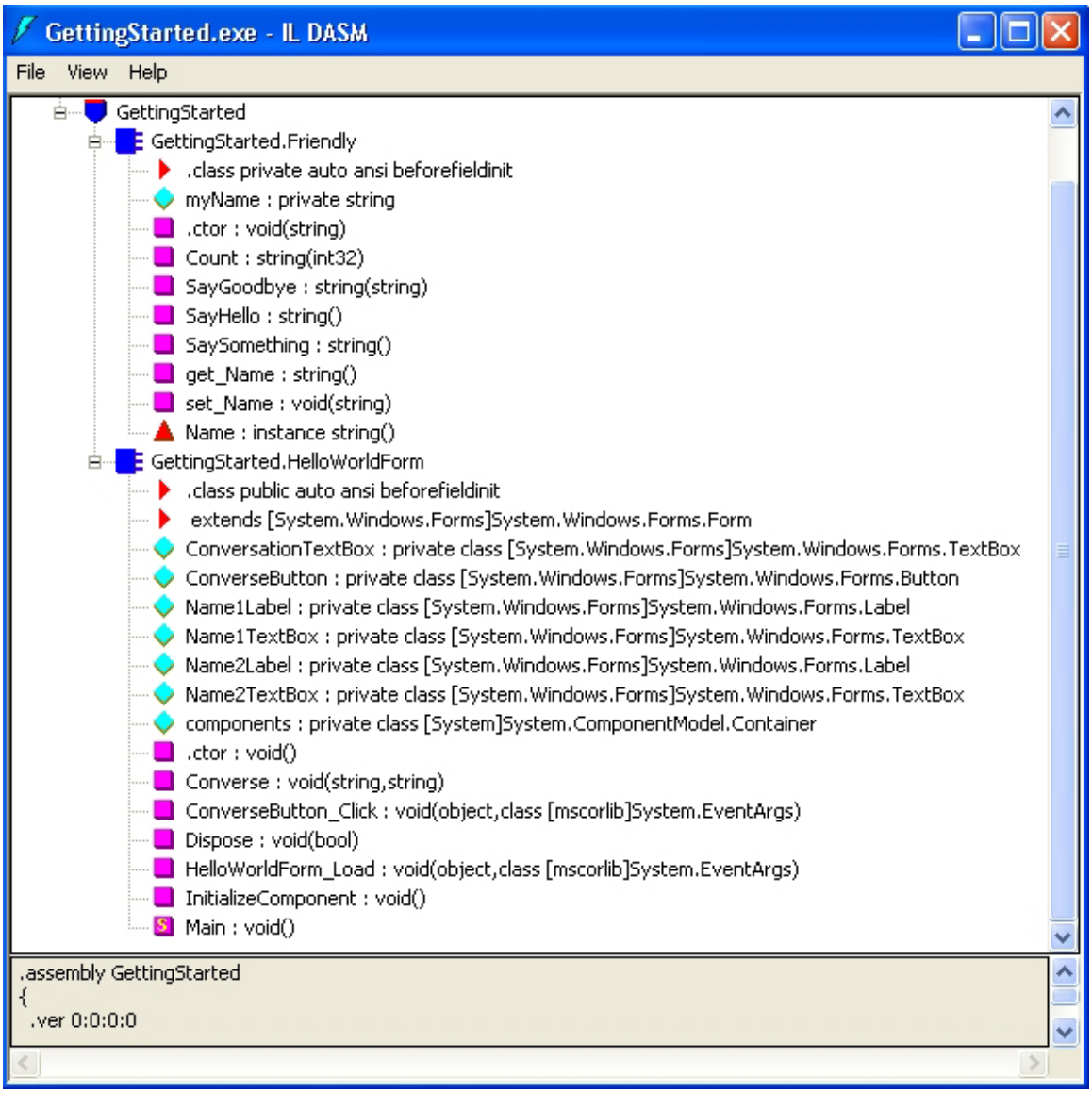
The .NET Framework SDK ships with a disassembler utility called ildasm that allows you to decompile .NET assemblies into IL Assembly Language statements. To start ildasm , make sure that the .NET Framework SDK is installed and in your path and type ildasm on the command line.

Note: if this does not work and you have installed Visual Studio.NET, ildasm is probably not in your path. To open a Visual Studio command prompt, from the start menu, select Visual Studio.NET [version] | Visual Studio.NET Tools | Visual Studio.NET [version] Command Prompt. Type ildasm.

- Select the File | Open menu and browse to:

```
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition
3.0\samples\GettingStarted\bin\Debug
```

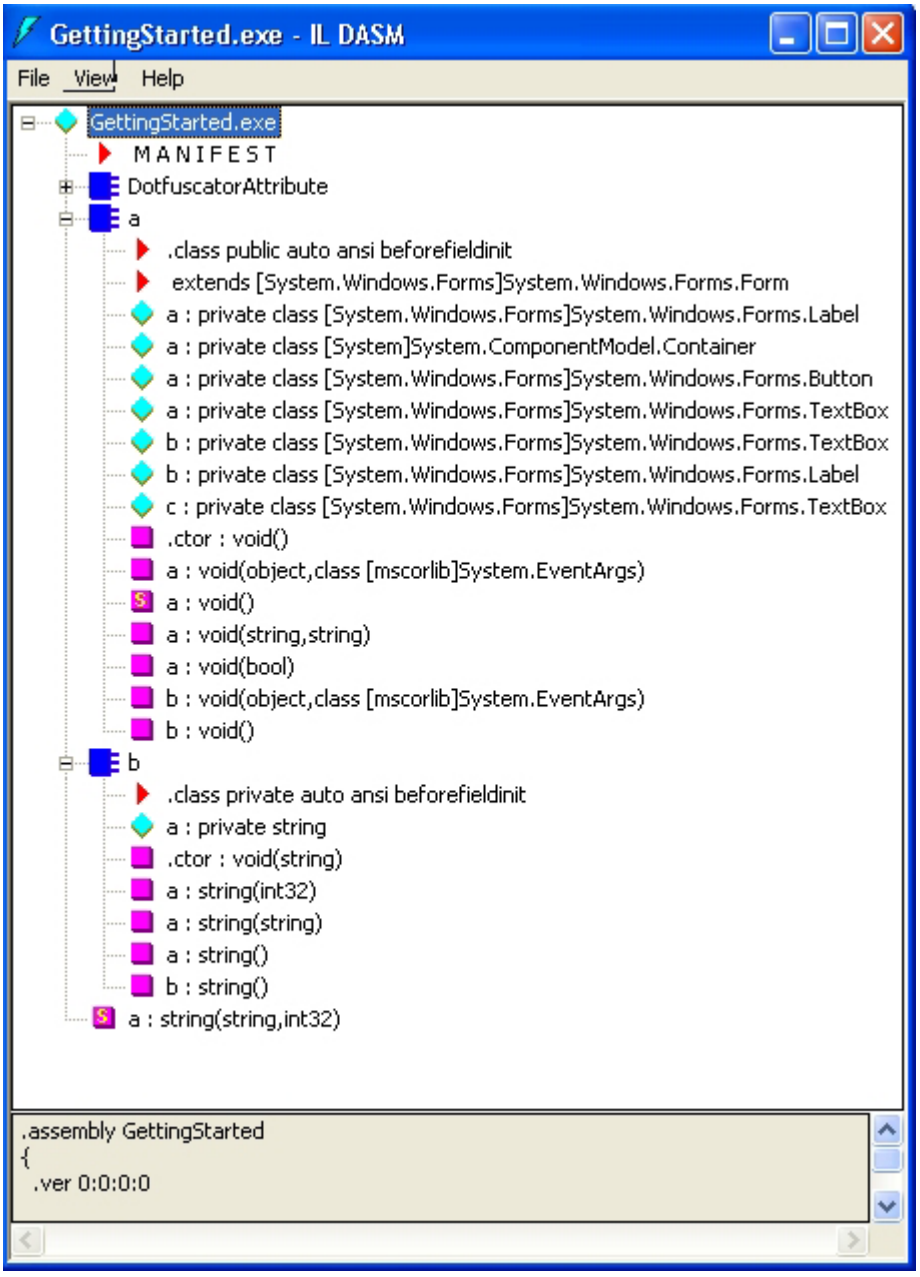

and select GettingStarted.exe. Click Open. A view of the disassembled assembly appears:



- To compare the currently shown, un-obfuscated HelloWorld application to the obfuscated version, start another copy of ildasm. This time browse to

```
C:\Program Files\PreEmptive Solutions\Dotfuscator Professional Edition
3.0\samples\GettingStarted\Dotfuscated
```

and select GettingStarted.exe. Click Open.



Place each ildasm window side-by-side. Compare and contrast both figures.

Notice the unobfuscated disassembly contains names of methods that are fairly understandable. For example, it is safe to assume that the ConverseButton_Click: void (object, class [mscorlib]System.EventArgs) method is called when the Converse button is clicked. Now look at the obfuscated version. Which method is called when the converse button is clicked? It is hard to tell. Also notice the missing SaySomething method. It was removed because the method wasn't being used anywhere in the code.

Double-click the methods SayHello:string() from the original assembly and a:string() from the obfuscated assembly. These two methods are in fact the same; however, when examining the disassembled IL code further, you will notice that the strings have

been encrypted in the obfuscated version to make the code harder to read. For example, locate the following line in the un-obfuscated version:

```
IL_0000:  ldstr      "Hello, my name is "
```

Now view the obfuscated version, and try to find the above string. If you're having trouble finding it, it's because it's encrypted and looks like the following:

```
IL_0000: ldstr bytearray (09 42 26 44 29 46 2B 48 26 4A 67 4C 6D 4E 22 50
                          28 52 73 54 3B 56 36 58 34 5A 3E 5C 7D 5E 36 60
                          12 62 43 64 )
```

You can imagine how confusing this can be for people who are trying to reverse-engineer the code, especially with more complex applications.

Step 2 –Decompiling

If you're now thinking your source code will be accessible only to a small circle of technical folks who actually know IL Assembly Language, think again. You can take this a step further and actually recreate the source code from our application by using a decompiler such as Reflector or Anakrino. These utilities can decompile a .NET assembly directly back to a high level language like C#, VB .NET, or Managed C++.

In this section we use two freely available decompilers:

1. Reflector for .NET, http://www.aisto.com/roeder/dotnet/
2. Anakrino (GUI version) / Exemplar (Command Line version), http://www.saurik.com/net/exemplar/

Running Anakrino/Exemplar against the Dotfuscated GettingStarted.exe file will produce the following error:

```
Exemplar.exe has encountered a problem and needs to close.  We are sorry
for the inconvenience.
```

Running .NET Reflector against the Dotfuscated GettingStarted.exe file and trying to examine a method such as a() will throw the following exception:

```
This item appears to be obfuscated and can not be translated.
```

```
System.NotSupportedException: Cannot resolve local variable 'Label_0047'.
   at Reflector.CodeModel.Memory.GotoStatement.Resolve()
   at _12.VisitBlockStatement(IBlockStatement statement)
   at _111.VisitStatement(IStatement value)
   at _119.VisitMethodDeclaration(IMethodDeclaration value)
   at _125.VisitMethodDeclaration(IMethodDeclaration value)
   at _126.VisitMethodDeclaration(IMethodDeclaration value)
   at _123.VisitMethodDeclaration(IMethodDeclaration value)
   at _146._1(Boolean )
```

Thus, Dotfuscator Professional was able to successfully prevent two major decompilers from reverse engineering your Dotfuscated code.

Configuration File Reference

Dotfuscator configuration files may have any name or extension, but usually have a `.xml` extension. Configuration files contain information about how a given application is to be "Dotfuscated". The configuration file is an XML document conforming to `dotfuscator_v2.1.dtd` (or one of its predecessors), referenced in the appendix.

This section documents Dotfuscator's XML configuration file. It contains detailed descriptions of each configuration option, making it useful as a reference, even if you are using Visual Studio, the standalone GUI, or the command line interface to generate a configuration file for you.

Version

The .xml file version attribute must be present and must be applicable to your version of Dotfuscator. It should match the version number of the DTD to which it conforms. "Point" releases of Dotfuscator are designed to be able to use unmodified configuration files from earlier versions. For example, you should be able to run Dotfuscator 1.1 using a version 1.0 configuration file without having to edit the configuration file.

Note: The configuration file version is not always the same as the version of Dotfuscator. Every version of Dotfuscator expects to see a specific, but not necessarily identical, version of the configuration file.

```
<dotfuscator version="2.1">
```

Property List and Properties

The optional property list section allows for the definition and assignment of variables (called "properties") that may be used later in the configuration file. Property definitions defined in this section are referred to as "internal" properties.

```
<!-- define expandable properties -->
<!-- optional -->
<propertylist>
 <property name="projectname" value="myproject"/>
 <property name="projectdir" value="c:\myprojects"/>
</propertylist>
```

Variables ("property references") may also be used in the configuration file without being defined in this section. For example they may be defined on the command line or come from the environment.

Properties work via string substitution, using the following algorithm to find a value associated with the property:

3. Check the external property list for a value.
4. If not found, check for an environment variable with the same name as the property,
5. If not found, check for an internal definition in the propertylist section of the configuration file,
6. If still not found, use the empty string as the value.

External properties are passed in on the command line using the `-p` option. There are three built-in external properties:

- "applicationdir", which reflects Dotfuscator's installation directory.
- "appdatadir", which reflects Dotfuscator's local data directory.
- "configdir", which reflects the directory in which the configuration file resides.

Properties are useful for creating configuration files that act as templates for multiple projects, or for different versions of the same project, or for simple portability across different build environments.

A property is referenced with the following syntax:

```
${property_name}
```

Property references are case sensitive, so `${MyProjectDir}` references a different property than does `${myprojectdir}`.

Property references may not be used just anywhere in the configuration file. Currently, property references may only be used as values in the "dir" or "name" attributes of the <file> element. Here is a list of sections that use the <file> element:

inputassembly	mapinput	mapoutput
output	tempdir	assembly
removalreport	transform	key
loadpaths	program	filelist

A property reference will be interpreted literally in any other place in the configuration file.

Property references may not be nested. Nesting them will result in an error.

Here is an example of a property reference in use:

```
<output>
 <file dir="${testdir}\output"/>
</output>
```

Global Section

The optional global section is for defining configuration options that apply across the entire run. The following sections describe each global option in detail. Global options are not case sensitive.

Library Global Option

This option has been deprecated in Dotfuscator 3.0. It has been replaced with a more granular library option that can be applied to individual input assemblies. When reading older configuration files, Dotfuscator will read this option and honor it, but the user interface will save the configuration file using the new options. See Library Mode By Assembly.

```
<global>
 <!—set library option -->
 <option>library</option>
</global>
```

Verbose, Quiet, and Investigate Global Options

These options are the same as the corresponding command line options. Setting the command line option sets the global option at run time. Alternatively, if the global option is set and the command line option is not set, then the global option takes precedence. In other words, there is no way to "unset" a set global option from the command line.

```
<global>
 <!-- run in verbose mode -->
 <option>verbose</option>
 <!-- run in quiet mode -->
 <option>quiet</option>
 <!-- investigate only and generate a map -->
 <option>investigate</option>
</global>
```

Input Assembly List

CE STD PRO The input assembly list contains the file names and directories of the assemblies you want to Dotfuscate. It also contains configuration options that are set at the assembly level.

If you have a multi-module assembly, you only need to list the module containing the manifest.

```
<input>
 <asmlist>
   <inputassembly>
    ...
    <file dir="c:\temp" name="myproj.dll"/>
   </inputassembly>
  ...
 </asmlist>
</input>
```

The <inputassembly> element can contain configuration options that apply only to the input assembly. See:

Library Mode By Assembly

Declarative Obfuscation By Assembly

Library Mode By Assembly

STD PRO This setting tells Dotfuscator that a particular input assembly constitutes a library. (For Dotfuscation purposes, a library is defined as an assembly that is going to be referenced from other components not specified as one of the inputs in this run) This has implications for renaming and pruning, regardless of any custom excludes you may have set.

CE In Dotfuscator Community Edition, the library option applies to all input assemblies. If any one input assembly is marked as a library, then all input assemblies are considered libraries.

Here are the rules when using the library option:

1. Names of public classes and nested public classes are not renamed. Members (fields and methods) of these classes are also not renamed if they have public, family, or famorassem access.
2. In addition, no virtual methods are renamed, regardless of access specifier. This allows clients of your library to override private virtual methods if need be (this is allowed behavior in the .NET architecture).
3. Any user-specified custom renaming exclusions are applied in addition to the exclusions implied by the above rules.
4. Property and Event metadata are always preserved.

If you do not have the library option set for an assembly, then you are telling Dotfuscator that your input assembly is a standalone application, or that it will only be referenced by other input assemblies. In this case obfuscation is much more aggressive:

1. Everything is renamed except methods that override classes external to the application (i.e. classes in assemblies that are not included in the run.)
2. Property and Event metadata is removed, since this metadata is not required to run the application (it is meant for "consumers" of library code).
3. As usual, user specified custom renaming exclusions are also applied.

To specify library mode for an input assembly, add an <option> element to its <inputassembly> element.

```
<inputassembly>
  <option>library</option>
  <file dir="c:\temp" name="myproj.dll"/>
</inputassembly>
```

Declarative Obfuscation By Assembly

For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes.

Enabling or Disabling Declarative Obfuscation


Dotfuscator allows you to switch Declarative Obfuscation on or off for specific input assemblies. If not enabled, Dotfuscator will ignore obfuscation related custom attributes.


In Dotfuscator Community Edition, the "honorOAs" option applies to all input assemblies. If any one input assembly is marked with "honorOAs", then the option is set for all input assemblies.

To enable declarative obfuscation via the configuration file, you add a "honorOAs" option to each <inputassembly> element.

```
<inputassembly>
  <option>honoroas</option>
  ...
</inputassembly>
```

Stripping Declarative Obfuscation Attributes


To tell Dotfuscator to strip obfuscation attributes via the configuration file, you add a "stripOAs" option to each <inputassembly> element.


In Dotfuscator Community Edition, the "stripOAs" option applies to all input assemblies. If any one input assembly is marked with "stripOAs", then the option is set for all input assemblies.

```
<inputassembly>
  <option>stripoas</option>
  ...
</inputassembly>
```

Output Directory


This is the directory where output assemblies will be written. The application will always overwrite files in this directory without prompting the user.

```
<!-- destination directory is required -->
<output>
 <file dir="c:\work"/>
</output>
```

Temp Directory

This section is optional and specifies Dotfuscator's working directory. If not specified, the working directory defaults to the system's temporary directory. The application uses the working directory to run ildasm and ilasm on the input assemblies. The disassembled output is stored in this directory along with any resources (managed or unmanaged) embedded in the input assemblies. These files are automatically deleted after processing.

```
<!-- scratch directory is optional -->
<!-- If absent, defaults to system's temp dir -->
<tempdir>
 <file dir="c:\temp"/>
</tempdir>
```

Obfuscation Attribute Feature Map

The feature map is for Declarative Obfuscation. For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes. That section describes the Feature Map and lists the native feature strings that Dotfuscator understands.

In the configuration file, the <obfuscationattributemap> element is where you can map strings obtained from an obfuscation attribute's Feature property to one or more feature strings that Dotfuscator understands.

Here is what such a mapping looks like in the XML configuration file.

```
  <obfuscationattributemap>
    <feature name="testmode">renaming, controlflow</feature>
  </obfuscationattributemap>
```


Renaming Section

The renaming section allows you to specify options that are specific to renaming, input and output mapping file locations, and fine-grained rules for excluding items from renaming.

The renaming section is optional. If not present, the following defaults apply:

- Default renaming (namespaces are removed).
- New names are chosen using the "loweralpha" renaming scheme.
- No mapping file read or written.
- No exclusions beyond those dictated by your application type.

The section on identifier renaming describes the renaming options, the mapping file, and custom exclusions in great depth. The following sections present an overview of each.

Renaming Options

Dotfuscator allows several options that govern how namespaces are treated by the renaming algorithm. These are: "keepnamespace" and "keephierarchy" and are explained in detail in the section on identifier renaming.

The "disable" option is primarily for convenience and troubleshooting purposes. When set, Dotfuscator skips renaming altogether, regardless of what's in the rest of the renaming section.

```
<renaming>
 <!--  Keep namespaces as they are, but rename types. -->
 <option>keepnamespace</option>

 <!-- Preserves namespace hierarchy but rename -->
 <!-- namespace names -->
 <option>keephierarchy</option>
```

```
<!-- Skip renaming, ignoring rest of section -->
<option>disable</option>
...
</renaming>
```

Renaming options are not case sensitive.

Renaming Exclusion List

This section provides a dynamic way to fine tune the renaming of the input assemblies. It can contain a list of "exclusion rules" that are applied at runtime. If a rule selects a given class, method, or field, then that item is not renamed.

These rules are applied *in addition to* rules implied by global options such as the library option.

The rules are logically ORed together, so any item that is selected by at least one rule is not renamed.

The exclusion list has support for excluding names by type, method, field, assembly, module, or namespace.

Each type of rule is explained in detail in the section on identifier renaming.

Output Mapping File

This feature of Dotfuscator produces a log of all the renaming mappings used by Dotfuscator during a specific run. It also provides a statistics section.

Specifying this option instructs Dotfuscator's renamer to keep track of how things were renamed for both your immediate review and to possibly be used as input in a future Dotfuscator (Professional Edition) run. A file is created from this option that is then used in the incremental input file option.

Accidental loss of this file can destroy your chances of incrementally updating your application in the future. Therefore, proper backup of this file is crucial. For this reason, Dotfuscator will automatically rename an existing map file with the same name before overwriting it with a new version of the map file.

If you do not want Dotfuscator to rename existing map files before overwriting, set the attribute `overwrite="true"`.

The format of the mapping file is discussed in the section on identifier renaming.

```
<renaming>
 ...
 <mapping>
  <mapoutput overwrite="true">
   <file dir="c:\work" name="testout.xml"/>
  </mapoutput>
 </mapping>
</renaming>
```

A Note about XML Configuration Files

Dotfuscator uses XML formatted documents for the configuration and mapping files. When loaded, both of these documents are validated according to the Document Type Definitions (DTDs) specified in the "doctype". In order to perform the validation, Dotfuscator must be able to access the relevant DTD.

Dotfuscator takes the following steps to locate DTDs:

1. If the DTD URI specifies a local file, Dotfuscator searches for it in the indicated place. If it is not found there, an error occurs.
2. If the DTD URI specifies a web resource, Dotfuscator first searches its cache for a file with the same name as that specified in the URI. Dotfuscator keeps its cache in the [CommonApplicationData]\PreEmptive Solutions\Common directory, where [CommonApplicationData] refers to the OS specific directory that serves as a common repository for application-specific data that is used by all users.
3. If not found, Dotfuscator accesses the URI to obtain the DTD. If found, Dotfuscator caches the DTD so subsequent requests will not need to access the network. If the

DTD is not found, or if Dotfuscator is unable to retrieve it from the network, an error occurs.

See the appendices for links to the DTDs.

Summary

Dotfuscator is highly configurable and correct configuration is essential for correct output. It is possible to create and maintain complex configuration files using just a text editor and knowledge of the configuration options defined in this section. That is not the optimal way to make the most of Dotfuscator though; for everyday use, you will want to use Dotfuscator's GUI or command line interface to set options.

Identifier Renaming

Dotfuscator is capable of renaming all classes, methods, and fields to short (space-saving) names. In addition to making decompiled output much more difficult to understand, it also makes the resulting executable smaller in size.

Class Renaming Options

Several options exist for class renaming. You may:

- Specify classes to be renamed while keeping their namespace membership (`keepnamespace`)
- Rename namespace names while preserving namespace hierarchy (`keephierarchy`)
- Rename completely, removing the notion of namespace (default).

```
<renaming>
<!-- NOTE: these two options are mutually exclusive -->

 <!--  Keep namespaces as they are, but rename types. -->
 <option>keepnamespace</option>

 <!-- Preserves namespace hierarchy but rename -->
 <!-- namespace names -->
 <option>keephierarchy</option>
 ...
```

```
</renaming>
```

Keepnamespace Renaming Option

This methodology is excellent as a way to hide the names of your classes while maintaining namespace hierarchy. You give up some size reduction and obfuscation potential, but preserve your namespace names. This is useful for libraries that may be linked to obfuscated code, or for applications that use already obfuscated code. An example of this type of renaming is:

Original Name	New Name
Preemptive.Application.Main	Preemptive.Application.a
Preemptive.Application.LoadData	Preemptive.Application.b
Preemptive.Tools.BinaryTree	Preemptive.Tools.a
Preemptive.Tools.LinkedList	Preemptive.Tools.b

Keepnamespace and Keephierarchy are mutually exclusive options.

Keephierarchy Renaming Option

This option tells Dotfuscator to preserve the namespace hierarchy, while renaming the namespace and class names. For example:

Original Name	New Name
Preemptive.Application.Main	a.a.a
Preemptive.Application.LoadData	a.a.b
Preemptive.Tools.BinaryTree	a.b.a
Preemptive.Tools.LinkedList	a.b.b

KeepNamespace and KeepHierarchy are mutually exclusive options.

Full Class Renaming (default)

CE STD PRO The default methodology renames the class and namespace name to a new, smaller name. The idea behind this is quite simple. Our example becomes:

Original Name	New Name
Preemptive.Application.Main	a
Preemptive.Application.LoadData	b
Preemptive.Tools.BinaryTree	c
Preemptive.Tools.LinkedList	d

Note that all classes are now in the "global" namespace.

Overload-Induction Method Renaming

CE STD PRO Dotfuscator implements patented technology for method renaming called Overload-Induction™. Whereas most renaming systems simply assign one new name per old-name (i.e. "`getX()`" will become "`a()`", "`getY()`" will become "`b()`"), Overload-Induction induces method overloading maximally. The underlying idea being that the algorithm attempts to rename as many methods as possible to exactly the same name. Many customers report a full thirty-three percent of all methods being renamed to "`a()`". In these cases, that is a full fifty-percent of renameable methods! We say renameable because many methods inherently cannot be renamed, these include constructors, "class constructors", and methods intended to be called by the runtime.

After this deep obfuscation, the logic, while not destroyed, is beyond comprehension. The following simple example illustrates the power of the Overload Induction technique:

Original Source Code Before Obfuscation

```
private void CalcPayroll(SpecialList employeeGroup) {
   while (employeeGroup.HasMore()) {
        employee = employeeGroup.GetNext(true);
        employee.UpdateSalary();
        DistributeCheck(employee);
    }
}
```

Reverse-Engineered Source Code After Overload Induction Dotfuscation

```
private void a(a b) {
    while (b.a()) {
        a = b.a(true);
        a.a();
        a(a);
    }
}
```

The example shows that the obfuscated code is more compact. A positive side effect of renaming is size reduction. For example, if a name is 20 characters long, renaming it to **a()** saves a lot of space (specifically 19 characters). Renaming also saves space by conserving string heap entries. Renaming everything to "**a**" means that "**a**" is stored only once, and each method or field renamed to "**a**" can point to it. Overload Induction enhances this effect because the shortest identifiers are continually reused.

There are several distinct advantages to this methodology:

1. Renaming has long been a way to make decompiled output harder to understand. Renaming to unprintable characters (or names illegal in the target source language) is futile since decompilers are easily equipped to re-rename such identifiers. Considering that Overload-Induction could make one of three method names "`a()`", understanding decompiled output is more difficult to say the least.

2. Overload-Induction has no limitations that do not exist in all renaming systems (such limitations are discussed later).

3. Since overload-induction tends to use the same letter more often, it reaches into longer length names more slowly (e.g. `aa`, `aaa`, etc.). This also saves space.

Overload-Induction's patented algorithm determines all possible renaming collisions and only induces method overloading when it is safe to do so. The procedure is provably irreversible. In other words, it is impossible (even via running Overload-Induction again) to reconstruct the original method name relationships.

Renaming Exclusion Rules

The "exclude list" section provides a dynamic way to fine tune the renaming of the input assemblies. The user specifies a list of "rules" that are applied at runtime. If a rule selects a given class, method, or field, then that item is not renamed.

These rules are applied *in addition to* rules implied by global options such as library.

Rules are logically ORed together.

Regular Expressions (REs) may be used to select namespaces, types, methods or fields. The optional "regex" attribute is used for this purpose. The default value of "regex" is false. If "regex" is true then the value of the name attribute is interpreted as a regular expression; if it is false, the name is interpreted literally. This is important since REs assign special meaning to certain characters (e.g. the period).

Here are some examples of simple regular expressions:

```
.*                  Matches anything
MyLibrar.           Matches MyLibrary, MyLibrari, etc.
My[\.]Test[\.]I.*   Matches My.Test.Int1,My.Test.Internal, etc.
Get.*               Matches GetInt, GetValue, etc.
Get*                Matches Ge,Get,Gett,Gettt, etc.
```

Please refer to the .NET Framework documentation for a full description of the regular expression syntax.

Excluding Namespaces

This option excludes all types and their members in a given namespace. You can use an RE to specify the namespace.

```
<namespace name="My.Excluded.Namespace"/>
```

Excluding Types

This option excludes a type by name or by attribute specifier. You can use an RE to specify the type name.

Type names should be fully qualified names.

Inner (nested) classes are specified by using the '/' as a delimiter between outer and inner class. For example:

```
<type name="Library.Class1/NestedClass"/>
```

Attribute specifiers are selected or deselected with the "speclist" attribute. The "speclist" attribute is a comma-separated list of legal attribute specifiers for types. The legal values are:

```
abstract
interface
nestedassembly
nestedfamily
nestedfamorassem
nestedprivate
nestedpublic
notpublic
public
sealed
serializable
enum
```

A '-' preceding an attribute specifier negates the rule (i.e. it excludes all classes that do not have the specified attribute). A '+' may be specified but is not required. The rules implied in this list are logically ANDed together (that is, the set of excluded types is the intersection of all types that match each rule.). For instance, the following rule excludes any type that is public AND sealed.

```
<type name=".*" speclist="+public,+sealed" regex="true"/>
```

The <type> element may also be used just to select a type in order to specify rules for field and method exclusion within it. This allows type names to be renamed while preserving member names. The optional "excludetype" attribute is used for this purpose. If not specified, the default value is "true", meaning that the type name will be excluded.

```
<type name="MyCo.Test.MyOtherTest" excludetype="false">
```



```
 <!-- methods and fields excluded here -->
 ...
</type>
```

If a <type> element contains no nested field or method elements, then no methods and fields are selected for exclusion. This allows type names to be preserved while allowing members to be renamed.

Property and event names in excluded types are also preserved. (Remember that if a type is not excluded and the library option is not set, Dotfuscator removes property and event names.)

Excluding Methods

Methods may be excluded by first selecting the type using the <type> element, then providing a rule for selecting methods to exclude. Methods may be excluded by name and attribute specifier (as explained in the type section above), as well as by signature.

Allowed attribute specifiers are:

```
abstract
assembly
family
familyorassembly
final
private
public
static
virtual
```

If the attribute specifier is not set explicitly, then the "speclist" attribute will not be used at all as a matching criterion.

The following example selects all public instance methods beginning with "Set":

```
<method regex="true" name="Set.*" speclist="+public,-static"/>
```

Method signatures are specified using the "signature" attribute. A signature is a (possibly empty) list of types that match the types in the method's parameter list:

```
signature=""  <!-- empty parameter list -->
```

```
signature="int,MyClass,MyClass[]"
```

If the signature is not set explicitly, then the method signature will not be used at all as a matching criterion.

The following example selects a method by signature:

```
<method name="DoIt" signature="int, System.Console,
System.Collection.ICollection, float[]"/>
```

Global methods may be specified by using a special type selector with the name "Module:*mod_name*" where *mod_name* is the name of the module containing the global method.

Excluding Fields

Fields may be excluded by first selecting the type using the <type> element, then providing a rule for selecting fields to exclude. Fields may be excluded by name and attribute specifier (as explained in the type section above).

Allowed attribute specifiers are:

```
public
private
static
assembly
family
familyandassembly
familyorassembly
notserialized
```


If the attribute specifier is not set explicitly, then field attribute will not be used at all as a matching criterion.

The following example selects all static fields starting with "ENUM_":

```
<field regex="true" name="ENUM_.*" speclist="+static"/>
```

Global fields may be specified by using a special type selector with the name "Module:*mod_name*" where *mod_name* is the name of the module containing the global field.

Excluding By Custom Attribute

Types, methods, and fields may be selectively excluded by custom attribute. A custom attribute rule selects an item (type, method, or field) based on matching against the names of custom attributes that annotate the item. One or more custom attribute rules may be nested inside any rule that selects types, methods, or fields.

A type, method, or field rule may have multiple custom attribute rules associated with it. In this case, an item is selected if at least one of the custom attribute rules selects it.

The following example selects all types that are annotated with either MyCustomAttribute or MyOtherCustomAttribute:

```
<type name=".*" excludetype="false" regex="true>
   <customattribute name="MyCustomAttribute"/>
...<customattribute name="MyOtherCustomAttribute"/>
</type>
```

Custom attribute rules can also be written using regular expressions to match custom attribute names. The following example is another way to select all types annotated with either MyCustomAttribute or MyOtherCustomAttribute:

```
<type name=".*" excludetype="false" regex="true>
```

```
   <customattribute name="My.*CustomAttribute" regex="true"/>
</type>
```

The next example shows how to exclude all methods annotated with a custom attribute named MyCustomAttribute:

```
<type name=".*" excludetype="false" regex="true">
   <method name=".*" regex="true">
      <customattribute name="MyCustomAttribute"/>
   </method>
</type>
```

Excluding Assemblies

Assemblies may be excluded by name. When an assembly is excluded, all types and members within any of the assembly's modules are excluded. It makes sense to exclude an assembly when you have a scenario like the following:

- Assembly A should be Dotfuscated.
- Assembly B should not be Dotfuscated.
- Assembly B depends on assembly A

In other words, A provides services to B and no one else. You want the references to A embedded inside B to be Dotfuscated, so you include B in the same run as A, but you exclude B from renaming.

```
<assembly>
 <file dir="c:\projects\project1\" name="ExcludedLib.dll"/>
</assembly>
```

Excluding Modules

Modules may be excluded by name. Use the assembly attribute to qualify the module to a particular assembly. When specified, the assembly

name should be the logical assembly name rather than its physical file name. When a module is excluded, all its defined types and members are excluded.

Obviously, if a given module is shared among multiple assemblies, then the module will be excluded from all the assemblies.

```
<module name="MyLibResource.dll" assemblyname="MyLib"/>
```

The Map File

Dotfuscator generates a mapping file that associates old with new names. The new names of the classes, methods, and fields are shown. Bug tracking becomes difficult after renaming, especially with a high incidence of method overloading, making the map file essential.

The map file can be used to decode obfuscated stack traces as well as for incremental obfuscation. The map file also provides statistics regarding the overall effectiveness of renaming.

The elements of the mapping file are all very similar. A few things are noteworthy:

- If a <newname> element is absent, then the item was not renamed.
- In type names, nested class names are separated from the parent using the "/" character.
- Constructors are named ".ctor", while static constructors (aka static initializers, class constructors, etc) are named ".cctor". These are never renamed.

For additional reference, see the governing DTD for the map file.

Control Flow Obfuscation

Dotfuscator Professional Edition employs advanced control flow obfuscation. Traditional control flow obfuscation introduces false conditional statements and other misleading constructs in order to confuse and hopefully break decompilers. Instead of adding code constructs, Dotfuscator works by destroying the code patterns that decompilers use to recreate source code. The end result is code that is semantically equivalent to the original but contains no clues as to how the code was originally written. Even if highly advanced decompilers come to pass, their output will at best be guesswork.

For details, see the online user's guide.

User String Encryption

Dotfuscator Professional Edition allows you to hide user strings that are present in your assembly. A common cracker attack is to find critical code sections by looking for string references inside the binary. For example, if your application is time locked, it may display a message when the timeout expires. Crackers will do a simple text search for this message inside the disassembled or decompiled output and chances are, when they find it, they will be very close to your sensitive time lock algorithm.

Dotfuscator addresses this problem by allowing you to encrypt strings in these sensitive parts of your application, providing an effective barrier against this type of attack.

Since string encryption incurs a slight runtime penalty (for on-the-fly decryption when the string is used), the configuration rules are inclusion rules. That is, by default, no strings are encrypted unless you specifically include a method that uses the string. The intention is that you will only want to encrypt strings in the sensitive parts of your application.

For details, see the online user's guide.

Pruning

Dotfuscator Professional Edition has the ability to statically analyze your application and determine which pieces are not actually being used. This includes searching for unused types, unused methods, and unused fields. This is of great benefit if application size is a concern, particularly if you are building your application from reusable components. Typically, not every type, method, and field is used in such applications. Dotfuscator can extract exactly the pieces you need for any given application, making it the absolute smallest it can be.

The static analysis works by traversing your code, starting at a set of methods called "triggers". These are your application's entry points. In general, any method that you expect external applications to call must be defined as a trigger. For example, in a simple standalone application, the "Main" method would be defined as a trigger. An assembly can have more than one trigger defined for it.

As Dotfuscator traverses each trigger method's code, it notes which fields, methods, and types are being used. It then analyses all the called methods in a similar manner. The process continues until all called methods have been analyzed. Upon completion, Dotfuscator is able to determine a minimum set of types and their members necessary for the application to run. These and only these types are included in the output assembly.

For details, see the online user's guide.

Linking

You can use Dotfuscator to link multiple input assemblies into one or more output assemblies. For example, if you have input assemblies A, B, C, D, and E, you can link assemblies A,B, and C and name the result F. At the same time, you can also link D and E and name the result G. The only rule is that you can't link the same input assembly into multiple output assemblies.

The linking feature is fully integrated with the rest of Dotfuscator, so in one step, you can obfuscate, remove unused types, methods, and fields, and link the result into one assembly.

If you have other input assemblies that reference the assemblies you are linking together, Dotfuscator will transparently update all the assembly and type references so the output assemblies will correctly work together.

Linking is not supported for Managed C++ assemblies.

For details, see the online user's guide.

Watermarking

Dotfuscator Professional Edition supports watermarking .NET assemblies. Watermarking can be used to unobtrusively embed data such as copyright information or unique identification numbers into your .NET application without impacting its runtime behavior. This is one method that can be used to track unauthorized copies of your software back to the source. Dotfuscator's watermarking algorithm does not increase the size of your application, nor does it introduce extra metadata that could break your application.

For details, see the online user's guide.

Advanced Topics

This section describes different scenarios and issues encountered when Dotfuscating .NET applications and libraries.

P/Invoke Methods

P/Invoke methods (i.e. native platform methods) are automatically not renamed if Dotfuscator detects that the name is used to find the corresponding native function. P/Invoke methods that are mapped to native functions by an alias or by ordinal can be renamed.

Dotfuscating Assemblies with Managed Resources

Managed resources may be embedded inside a module (internal) or may be in files external to the module. Often, part (or all) of the name of the managed resource is a type name (see the .NET Framework documentation for more information about the "hub and spoke model" for lookup of managed resources.).

When the type name is renamed, Dotfuscator attempts to locate and rename the corresponding managed resource. If the resource is internal to the assembly, this is automatic. If the resource is embedded inside an external file, then the file must be in the same directory as the referencing module. If the resource is embedded inside another assembly, then that assembly must be one of the input assemblies.

Dotfuscating Multi-module Assemblies

A .NET assembly may be made up of multiple modules (i.e. files on disk). Usually an assembly is made up of one module, and this is the scenario that most tools such as Visual Studio support. Occasionally it is desirable to create assemblies made up of more than one module. Dotfuscator supports this scenario. Note that Dotfuscating a multi-module assembly is not the same as Dotfuscating multiple input assemblies (a separate scenario, also supported).

To Dotfuscate a multi-module assembly, only the prime module needs to be listed as an input assembly. The non-prime modules will be searched for in the same directory as the prime module.

In the prime module's assembly manifest, Dotfuscator will automatically update the hash values of the other modules.

Dotfuscating Strong Named Assemblies

Strong named assemblies are digitally signed. This allows the runtime to determine if an assembly has been altered after signing. The signature is an SHA1 hash signed with the private key of an RSA public/private key pair. Both the signature and the public key are embedded in the assembly's metadata.

Since Dotfuscator modifies the assembly, it is essential that signing occur *after* running the assembly through Dotfuscator.

Dotfuscator Professional Edition can handle this step as part of the obfuscation process. If you are using the Standard or Community Edition, you will need to complete signing in a separate build step after obfuscation is complete.

Manually Resigning after Obfuscation

You should *delay sign* the assembly during development and before Dotfuscation, then complete the signing process afterward. Please refer to the .NET Framework documentation if you require detailed information about delay signing assemblies.

To successfully obfuscate a strongly named assembly, follow these steps:

- Delay sign the assembly during development. This is done by embedding two custom attributes into your assembly. For C#, you would include the following lines into AssemblyInfo.cs:

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```


Where keyfile.snk is the name of the file containing your public key.

- Use the strong name tool (sn.exe) that comes with the .NET Framework to turn off the strong name verification while you are testing your assembly:

```
sn -Vr TestAsm.exe
```

- Obfuscate the delay signed assembly using Dotfuscator.
- After running through Dotfuscator turn on the verification for the obfuscated assembly using sn.exe. This unregisters the Dotfuscated assembly for verification skipping:

```
sn -Vu TestAsm.exe
```

- Now complete the signing process of the Dotfuscated assembly, where keyfile.snk is the name of the file containing your private key:

```
sn -R TestAsm.exe keyfile.snk
```

Remember to turn off strong name validation while testing your delay signed assemblies!

Dotfuscating 64 Bit Assemblies

Dotfuscator 3.0 can transparently obfuscate managed assemblies written explicitly for specific CPU architectures, including 64 bit architectures.

Dotfuscator itself is a managed application and can run on 32 bit and 64 bit versions of Windows.

Reflection and Dynamic Class Loading

Reflection and dynamic class loading are extremely powerful tools in the .NET architecture. This level of runtime program customization prevents Dotfuscator from infallibly determining whether it is safe to rename all types loaded into a given program.

Consider the following (C#) code fragment:

```
public object GetNewType() {
   Type type = Type.GetType( GetUserInputString(), true );
   object newInstance = Activator.CreateInstance( type );
```

```
    return newInstance;
}
```

This code loads a type by name and dynamically instantiates it. In addition, the name is coming from a string input by the user! (Or a runtime database, or another program, etc.)

There is obviously no way for Dotfuscator to predict which type names the user will enter. The solution is to exclude the names of all potentially loadable types (note that method and field renaming can still be performed). This is where manual user configuration (and some knowledge of the application being Dotfuscated) plays an important role.

Often the situation is less serious. Consider a slight variation:

```
public MyInterface GetNewType() {
    Type type = Type.GetType( GetUserInputString(), true );
    object newInstance = Activator.CreateInstance( type );
    return newInstance as MyInterface;
}
```

Now it is immediately obvious that only a subset of types need to be excluded: those implementing MyInterface.

Declarative Obfuscation Using Custom Attributes

The .NET Framework version 2.0 provides two new custom attributes designed to make it easy to automatically obfuscate assemblies without having to set up configuration files. This section outlines how you can use these attributes with Dotfuscator. It is assumed that you are familiar with custom attributes and how to apply them in your development language.

System.Reflection.ObfuscateAssemblyAttribute

This attribute is used at the assembly level to tell Dotfuscator how to obfuscate the assembly as a whole. Setting the AssemblyIsPrivate property to false tells Dotfuscator to run the assembly in library mode. If you set it to true, Dotfuscator will not run the

assembly in library mode and will rename as much as possible, including public types and members.

System.Reflection.ObfuscationAttribute

This attribute is used on types and their members and tells Dotfuscator how to obfuscate the item.

Feature Property

This string property has a default value of "all". This property is provided so that you can configure multiple obfuscation transforms independently by tagging an item with multiple ObfuscationAttributes, each with a different feature string.

Dotfuscator maps the "default" and "all" feature strings to "renaming".

Here is a list of other feature strings that Dotfuscator understands.

Feature String	Action
renaming	attribute configures renaming
controlflow	attribute configures control flow obfuscation.
stringencryption	attribute configures string encryption
trigger	attribute configures pruning by marking the annotated item as an entry point
conditionalinclude	attribute configures pruning by conditionally including the annotated item

If necessary, you can map other feature strings to "renaming" using the "Feature Map Strings" property sheet on the Setup Tab.

Dotfuscator ignores attributes with feature strings that it does not understand.

Exclude Property

This Boolean property has a default value of True. When True, it indicates that the tagged item should be excluded from the transforms implied by the Feature property. When False, it indicates that the tagged item should be included.

The current version of Dotfuscator supports one value of the Exclude property for any given transform. Dotfuscator will ignore rules that have unsupported Exclude values. The following list summarizes.

Feature String	Supported Exclude Value
renaming	True
controlflow	True
stringencryption	False
trigger	False
conditionalinclude	False

ApplyToMembers Property

This Boolean property has a default value of True. When the attribute is applied to an assembly or a type, a True value indicates that the operation should be applied to all the members (including nested types) of selected types as well. If false, the operation is applied to types only and not their members (or nested types).

Enabling or Disabling Declarative Obfuscation

Dotfuscator allows you to switch Declarative Obfuscation on or off for all input assemblies. If not enabled, Dotfuscator will ignore obfuscation related custom attributes. You can also switch it off for specific assemblies.

Stripping Declarative Obfuscation Attributes

Dotfuscator can strip out the obfuscation attributes when processing is complete, so your output assemblies will not contain clues about how it was obfuscated. Both of the declarative obfuscation attributes include a Boolean "StripAfterObfuscation" property whose default value is true

Dotfuscator also has configuration settings that interact with the value of the StripAfterObfuscation property at obfuscation time.

The settings that effect declarative obfuscation attribute stripping and how they interact are summarized in the below table.

Dotfuscator is Honoring Attributes	Dotfuscator is Stripping Attributes	Attribute's StripAfterObfuscation Property	Result

Yes	Yes	True or False	Strip Attribute
Yes	No	True	Strip Attribute
Yes	No	False	Keep Attribute
No	Yes	True or False	Strip Attribute
No	No	True or False	Keep Attribute

Using Feature Map Strings

Dotfuscator allows you to map values contained in an Obfuscation Attribute's Feature property to feature strings that Dotfuscator understands.

For example, you can annotate your application with obfuscation attributes that reference a feature called "testmode". Dotfuscator by default will not understand this feature string; therefore it will ignore the attributes. If later on you want Dotfuscator to use these attributes to configure renaming and controlflow obfuscation, you can map the feature string "testmode" to Dotfuscator's built in "renaming" and "controlflow" strings.

Friend Assemblies

.NET Framework v2.0 introduces the concept of "friend assemblies", where an assembly may declare that its internal type definitions are visible to specified other assemblies. This is done using the System.Runtime.CompilerServices.InternalsVisibleToAttribute.

Dotfuscator Community Edition assumes that all friend assemblies to an input assembly are also input assemblies; if that is not the case, manual configuration is required to ensure that referenced internal names are not mangled.

Finding External Tools

Dotfuscator uses ildasm and ilasm to process the input assemblies. Ildasm is the MSIL disassembler that ships with the .NET Framework SDK. Ilasm is the MSIL assembler that ships with the .NET Framework Redistributable.

Dotfuscator attempts to match each input assembly with the toolset that ships with the version of the .NET Framework that it was compiled with. So Dotfuscator will use the 1.1 versions of ildasm and ilasm on an assembly compiled on the 1.1 version of the Framework; likewise, it will use the 2.0 tools on an assembly compiled on the 2.0 version of the framework.

If Dotfuscator cannot find the version appropriate toolset for an input assembly, it will use a later version if present. It will never use an older version.

By default, Dotfuscator searches for these external tools using the following algorithm:

- Determine the version of the .NET Framework that the input assembly was compiled on.
- Search the .NET Framework and .NET Framework SDK directories corresponding to the .NET Framework version determined in step 1.
- Search the .NET Framework and .NET Framework SDK directories corresponding to later versions of the .NET Framework determined in the first step.

If Dotfuscator cannot find one or both of these programs, it will issue an error.

Using the Command Line Interface

The command line interface is designed to allow you to:

- Obfuscate from the command line without requiring you to create a configuration file.
- Override or supplement options in an existing configuration file using command line options.
- Create a configuration file from the command line.
- Launch the standalone GUI with options and/or a configuration file specified on the command line.

Command Line Option Summary

Command line options may begin with the '/' or the '-' characters.

```
Usage: dotfuscator [options] [config_file]
```

Traditional Options

The following is a summary of the traditional command line options.

Traditional Options	Description
/g	Launch the standalone GUI

/i	Investigate only
/p=<property list>	Specifies values for user defined properties in the configuration file. Comma separated list of name-value pairs (e.g. /p=projectdir=c:\\temp,projectname=My App.exe)
/q	Quiet output
/v	Verbose output
/?	Print help
[config_file]	configuration file containing runtime options.

The `-v` option induces Dotfuscator to provide information about its progress during execution. The level of detail here will likely change between releases.

The `-i` option tells Dotfuscator to not create any output assemblies files. If the configuration file specifies a map file, the results of the run will be found there (this option is close to worthless without generating a map).

The `-q` option tells Dotfuscator to run completely without printed output. This is suitable for inclusion into application build sequences. This option overrides verbose mode.

The `-p` option tells Dotfuscator to set external properties at the command line. Setting these properties here will override those specified in the `<properties>` section of the configuration file.

The *<proplist>* is a list of comma-separated name-value pairs. For example, property declaration and assignment in conjunction with the `-p` option, might look like:

```
/p=projectdir=c:\temp,projectname=MyApp
```

Properties may be quoted if they contain spaces as illustrated below:

```
/p=MyProperty="value has spaces"
```

Property names are case sensitive.

The `-g` option tells Dotfuscator to start up the standalone GUI. You can start the GUI with external properties and a specific configuration file using this option:

```
Dotfuscator /g /p=projectdir=c:\temp project_template.xml
```

The GUI will also start up if Dotfuscator is run with no command line arguments.

The *`configfile`* is a configuration file that is required for every run of Dotfuscator. Notice you do not enter configuration information or target assemblies on the command line. This information must be found in the configuration file.

Extended Options

Extended options are designed to allow for basic obfuscation from the command line, without requiring you to first create a configuration file. If you use a configuration file with an extended command line option, the command line option will supplement or override the commands in the configuration file. See Supplementing or Overriding a Configuration File from the Command Line for more information.

Extended options are recognized by the first four characters.

The following is a summary of the extended command line options. An asterisk denotes the default setting if an option is missing and no configuration file is specified.

	Extended Options	Description
	/in <file>[,<file>]	Specify input assemblies. Default is governed by assembly file extension (EXEs are private; .DLLs are run in library mode).
	/out:<directory>	Specify output directory. Default is ".\Dotfuscated".
	/makeconfig:<file>	Save all runtime options (from command line and configuration file if present) to <file>.
	/disable	Disable all transforms regardless of other options
	/rename:[on|off*]	Enable/disable renaming.
	/mapout:<file>	Specify output mapping file.

		Default is ".\Dotfuscated\map.xml".
	/clobbermap:[on|off*]	Specify map file overwrite mode.
	/keep:[namespace|hierarchy|none*]	Specify type renaming scheme.

Examples:

```
dotfuscator -in:my.dll
```

Obfuscates my.dll as a library (visible symbols preserved and unpruned) with renaming, control flow, pruning, and string encryption turned on. The output assembly is written to a directory called .\Dotfuscated, and the map file is written to .\Dotfuscated\map.xml since no output directories were specified.

```
dotfuscator -in:myapp.exe,private.dll
```

Obfuscates myapp.exe and private.dll together as a standalone application. Even visible symbols inside the DLL are obfuscated. Pruning is enabled based on the entry point method contained in myapp.exe.

```
dotfuscator -in:myapp.exe -mapo:MyNames.xml
```

This command obfuscates myapp.exe as a standalone application. An output renaming map is specified.

Supplementing or Overriding a Configuration File from the Command Line

Dotfuscator has the unique ability to accept a complete or partial configuration file, yet allow you to supplement or override its options from the command line. This allows you to quickly adjust and tweak settings using a standard configuration file as a template.

Command Line Option	Configuration File Option	Notes
/in <file>[,<file>]	"input" section	adds
/out: <directory>	"output" section	overrides

/disable	Sets "disable" option in "renaming", "controlflow", "stringencrypt", and "removal" sections	overrides
/rename:[on:off]	Sets (or unsets) "disable" option in "renaming" section.	overrides
/mapout:<file>	"mapoutput" section	overrides
/clobbermap:[on|off]	"overwrite" attribute in "mapoutput" section	overrides
/keep:[namespace|hierarchy|none]	Sets (or unsets) renaming options: "keepnamespace", "keephierarchy"	overrides

Examples:

The following examples use this configuration file that enables renaming with an output mapping file. It is referenced as "myconfig.xml" in the examples.

```
<?xml version="1.0"?>
<!DOCTYPE dotfuscator SYSTEM
"http://www.preemptive.com/dotfuscator/dtd/dotfuscator_v2.1.dtd">
<dotfuscator version="2.1">
   <renaming>
      <mapping>
         <mapoutput overwrite="true">
            <file dir="${configdir}\reports" name="MyMap.xml"/>
         </mapoutput>
      </mapping>
   </renaming>
</dotfuscator>
```

```
dotfuscator -in:my.dll myconfig.xml
```

This command specifies my.dll as an input assembly in library mode (because of the DLL extension), and applies the renaming options in the configuration file. In this case, control flow, string encryption, and pruning are disabled because they are implicitly disabled in the configuration file.

The output DLL will go in a directory called ".\Dotfuscated", since an output is not specified in the configuration file or on the command line.

```
dotfuscator -in:my.dll -keep:namespace myconfig.xml
```

This command also specifies my.dll as an input assembly. In addition, it tells the renamer to keep namespaces.

Saving a Configuration File from the Command Line

Once you have the command line settings that you want for your application, you can save a configuration file containing those settings by using the /makeconfig option. This will take all your command line options, merge them with your configuration template if you have one, and save a custom configuration file that you can use by itself for future runs.

Example:

```
dotfuscator -in:my.dll -keep:namespace -make:new.xml myconfig.xml
```

The resulting configuration file (new.xml) shows the command line options merged with the options from the original configuration (myconfig.xml):

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE dotfuscator SYSTEM
"http://www.preemptive.com/dotfuscator/dtd/dotfuscator_v2.1.dtd">
<dotfuscator version="2.1">
  <input>
    <asmlist>
      <inputassembly>
        <option>library</option>
        <file dir="." name="my.dll" />
      </inputassembly>
    </asmlist>
  </input>
  <output>
    <file dir="C:\MSProjects\dotfuscatortest\Dotfuscated" />
  </output>
  <renaming>
    <option>keepnamespace</option>
```



```
    <mapping>
      <mapoutput overwrite="true">
        <file dir="${configdir}\reports" name="MyMap.xml" />
      </mapoutput>
    </mapping>
  </renaming>
</dotfuscator>
```

Launching the GUI from the Command Line

If you invoke Dotfuscator with no command line options, the standalone GUI will start up with an empty project.

Alternatively, you can specify any combination of legal command line options along with the /g option to launch the GUI with those options in effect.

Example:

```
dotfuscator -g -in:my.dll myconfig.xml
```

This command launches the standalone GUI. The settings from myconfig.xml will be loaded, and my.dll will be set as an input assembly.

GUI Reference

There are two user interfaces to Dotfuscator.

- PRO Dotfuscator Professional Edition provides deep Visual Studio integration, so you can create and edit Dotfuscator projects within Visual Studio and run Dotfuscator as part of your builds.
- All versions of the product come with a standalone GUI. That allows you to configure and run Dotfuscator on your applications.

The first two topics in this section describe the UI types, while the remaining sections describe elements and features that are common to the two UIs.

Using the Visual Studio Integrated UI

PRO From inside Visual Studio, you can create and edit Dotfuscator projects and add them to your existing solutions. During a build, a Dotfuscator project can consume binary outputs (i.e. your compiled .NET assemblies) from the development projects in your solution, and in turn, expose the obfuscated outputs to deployment projects. Using the Visual Studio Integrated UI is the easiest way to configure and seamlessly use Dotfuscator. Note: The Visual Studio Integrated UI is not part of the Community Edition. If you are using the Community Edition, we recommend that you download the free trial version of the Professional Edition to experience this seamless configuration first hand.

For details, see the online user's guide.

Using the Standalone GUI

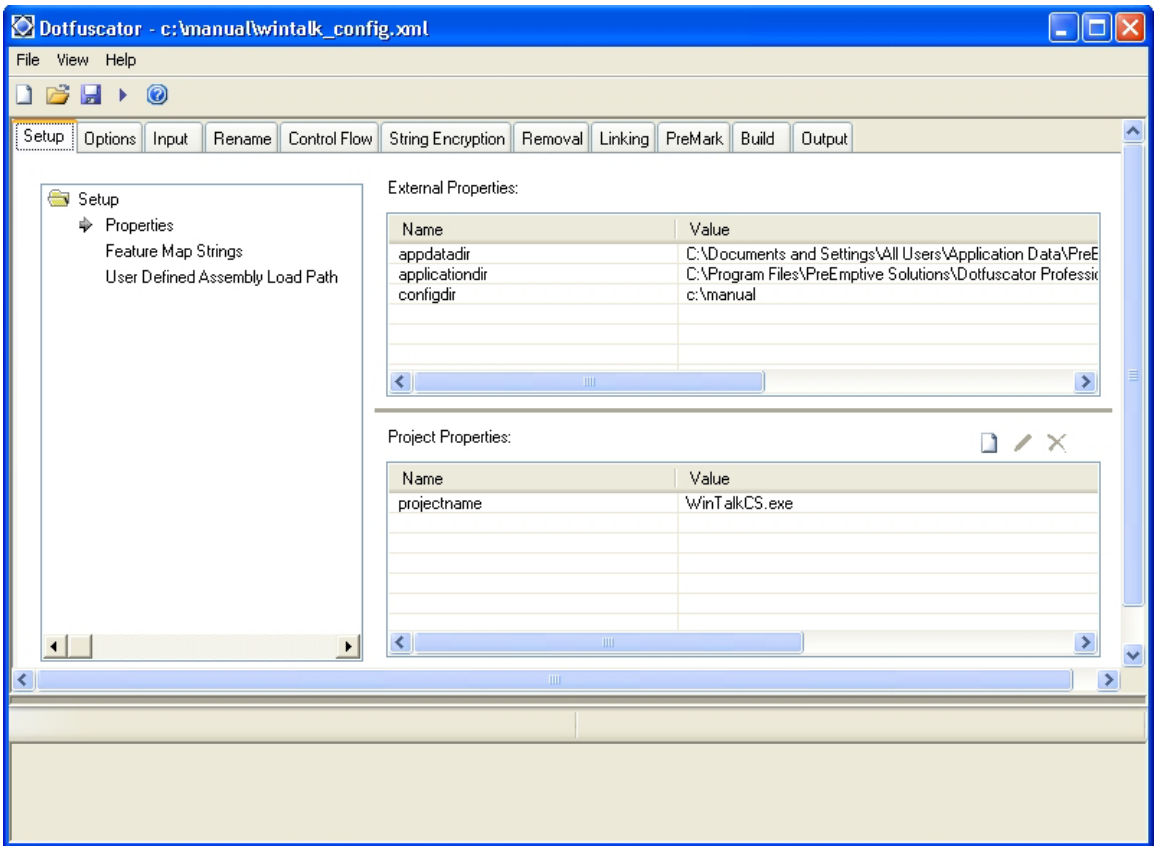
In this section, we describe how to use Dotfuscator's standalone GUI. You can use the standalone GUI to create new projects or edit existing ones. The resulting project can be saved and used later by the command line interface or you can obfuscate right in the GUI and view the results.

GUI Orientation

The Main Window

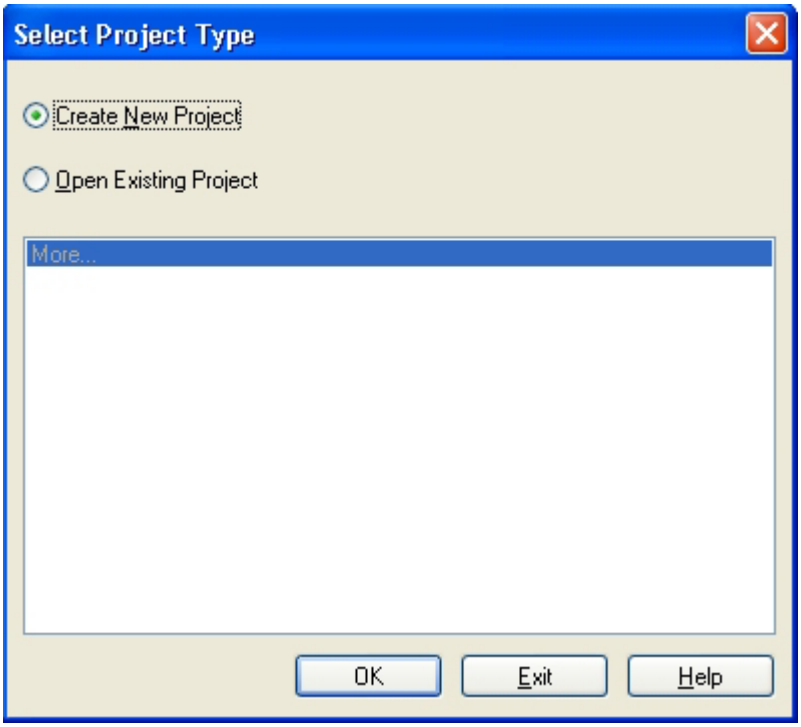
The main Dotfuscator window consists of four vertically arranged activity zones:

- At the top, the familiar Windows Menu Bar
- Below that menu, a toolbar appears for frequently accessed actions
- Below the toolbar, a tabbed section appears that organizes the specification and command activities for the Project
- At the bottom, a scrollable console pane is provided for viewing output.



The Select Project Window

Start the GUI, without a project loaded, by invoking `dotfuscator.exe` (no arguments) from the installation directory, or by clicking on the icon created by the installer. After the splash screen vanishes, the Select Project Type window will appear. From here, you can either create a new project, select an existing project from the most recently used list, or by selecting "More...", browse the file system for a project.



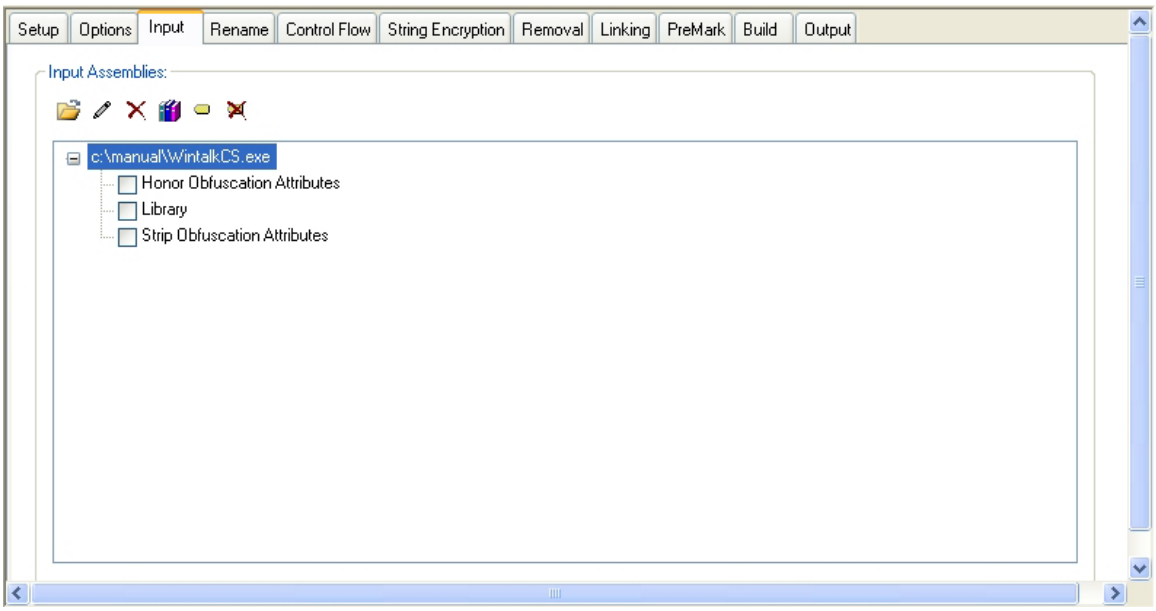
Working with Projects

To create a new project, you need to do three things:

1. Select Input Assemblies
2. Specify the Destination Directory
3. Save the Project Configuration File

Select Input Assemblies

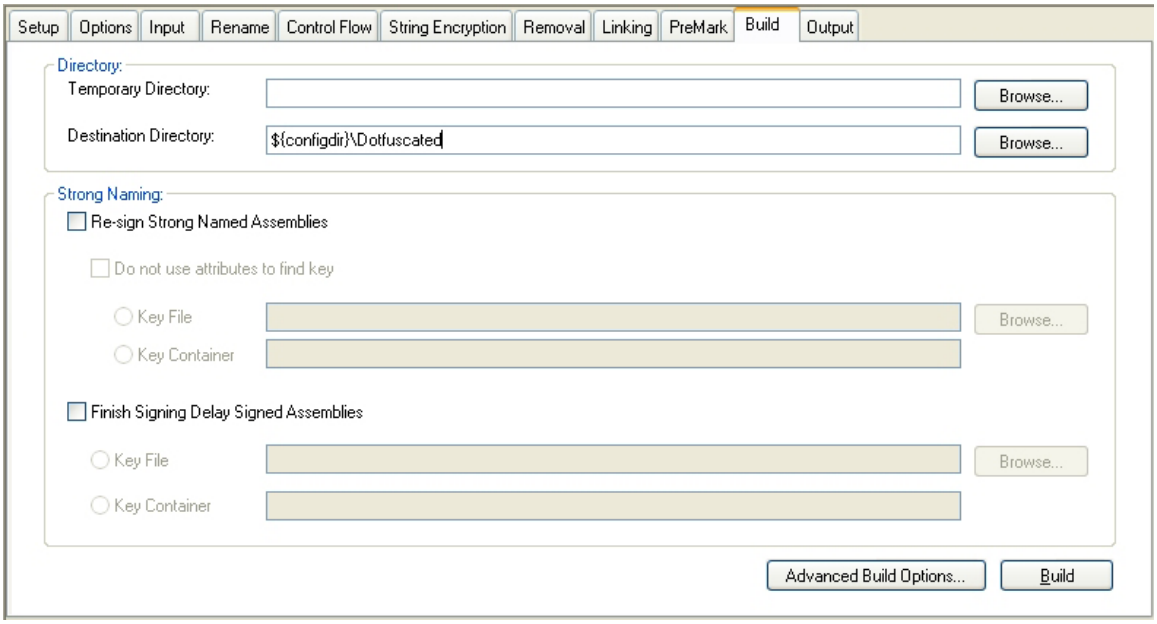
From the Input Tab, you can use the toolbar to add an input assembly. From the "Add Input Assembly" dialog, you can type in the assembly's directory and file name, or browse for it in the file system.



Specify the Destination Directory

When you create a new project, the output directory is set by default to "${configdir}\Dotfuscated". The ${configdir} is a built in property that is expanded to the folder that your project file is saved to.

If you want to choose a different destination directory, you can type it in directly on the Build tab.



Alternatively, you can browse your file system for the intended destination directory. The Browse button on the right of the Destination Directory field brings up the Select Destination Directory dialog that provides a directory navigation tree.

Save the Project Configuration File

You can save your project by either selecting File/Save Project or File/Save Project As from the menu or by clicking on the Save Project button on the toolbar. Navigate to your project directory, fill in your project configuration file name, click the Save button, and the project will be saved.

Working with Input Assemblies

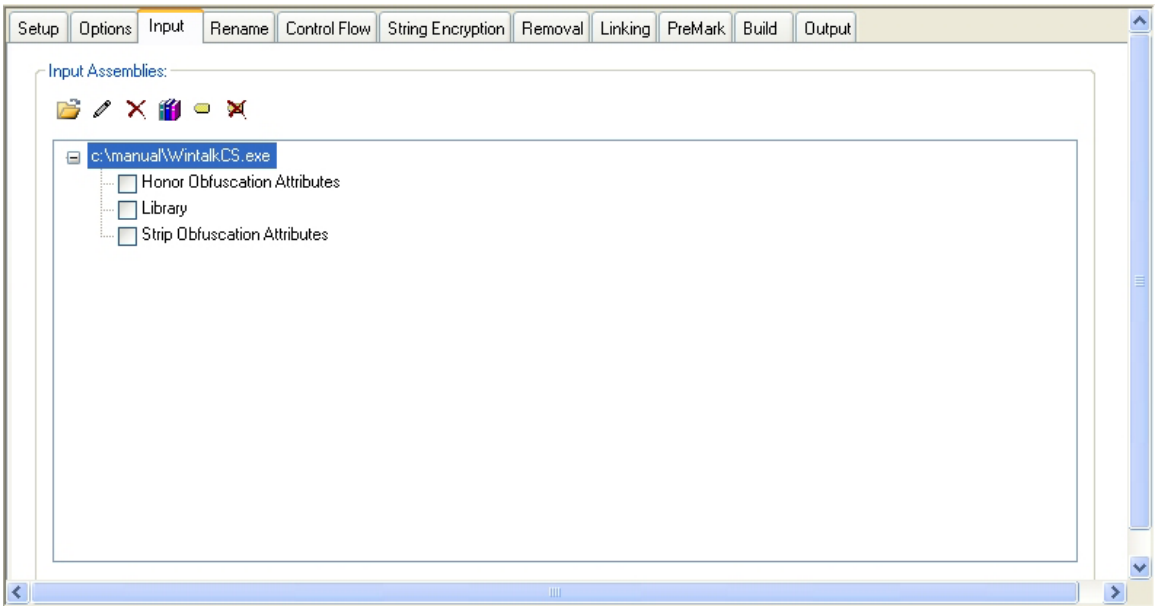
Adding Assemblies

The input tab is used to specify input assemblies for the project. With the "Add New Assembly" button on the toolbar, you can add a new assembly to your project. Clicking on the button brings up the "Add Input Assembly" dialog, where you can enter the assembly's directory and file name, or browse for it in the file system.

Using the browse window, you can add multiple assemblies by multi-selecting the ones you wish to add.

Editing and Removing Assemblies

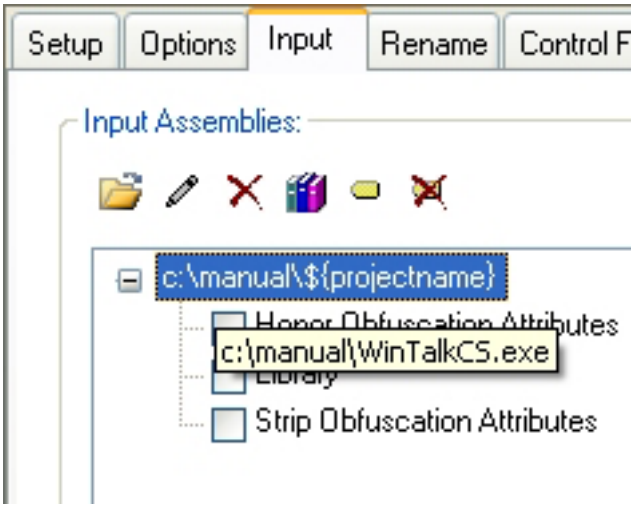
The Edit and Remove buttons are used to change or remove assemblies from the project. To use, highlight an input assembly on the list and click on the appropriate toolbar button. You can also delete an input assembly by highlighting it and pressing the Delete key.



When editing an input assembly path, the text you type can have a Project Property embedded in it. For example:

```
c:\${inputdir}\myapplication.exe
```

Property substitution takes place based on the precedence rules specified in the section on Property List and Properties. You can view the actual, fully resolved value by placing the cursor on the item that has a Project Property embedded in it.



Library Mode

Library mode can be toggled for all assemblies using the library button on the toolbar.

Declarative Obfuscation

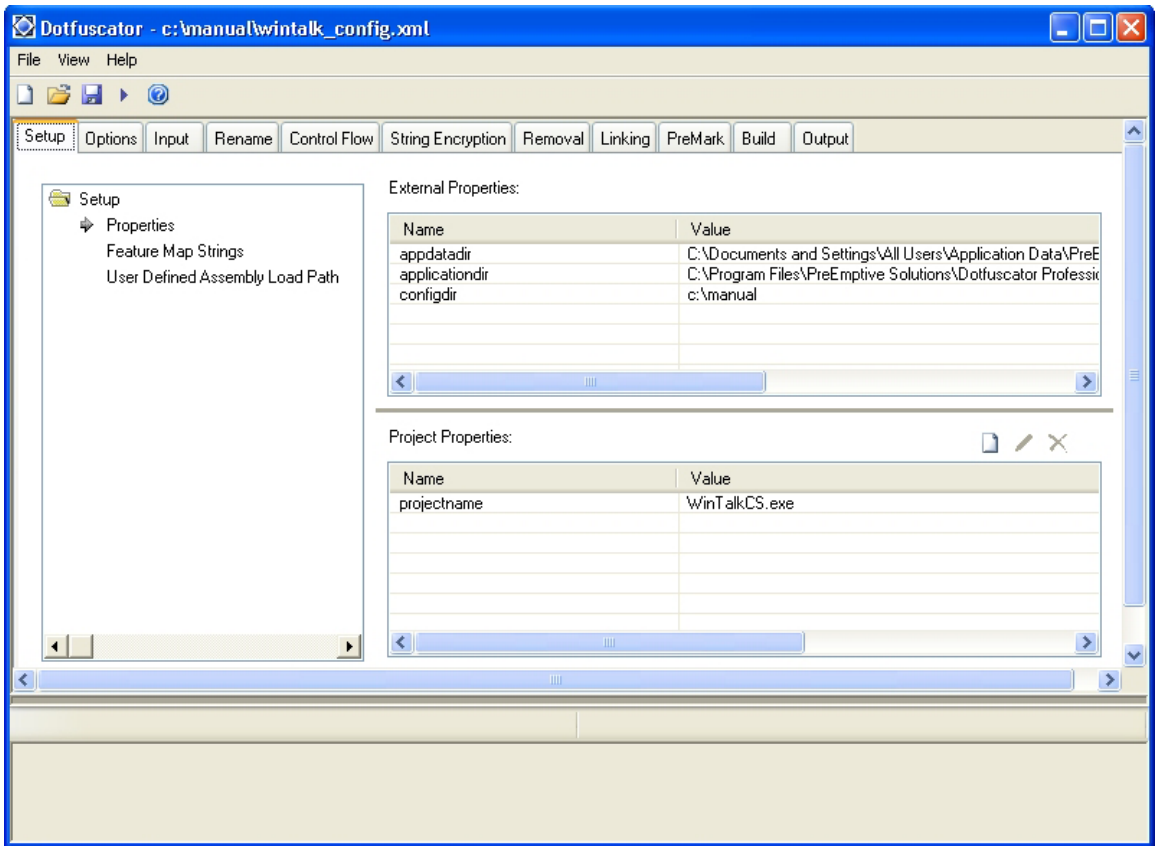
The Honor Obfuscation Attributes and Strip Obfuscation Attributes settings can be toggled for all input assemblies using the respective button on the toolbar.

The Setup Tab

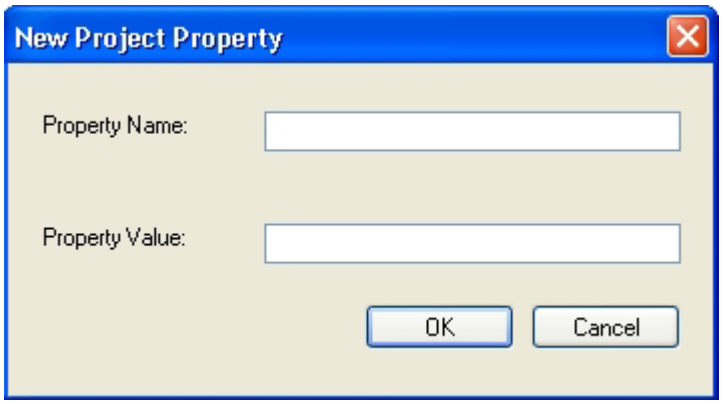
The setup tab allows you to configure Project Properties, Feature Map Strings, and your Assembly Load Path. You can choose the feature you wish to edit by selecting the appropriate node in the navigation pane on the left side of the tab.

Properties

The properties editor allows you to view and add user-defined name-value pairs as Project Properties and to view External Properties that have been defined from the command line. See Property List and Properties for a full explanation. To add a Project Property, click the New button on the project properties toolbar.



This brings up the New Project Property dialog. Type in a name and a value in the fields provided. Click on the OK button, and the property will be set.



You can use the Edit and Delete toolbar buttons in a similar manner to modify or remove Project Properties. You can also delete a property by selecting it and pressing the Delete key.

Feature Map Strings

The feature map editor is for Declarative Obfuscation. For a complete description of Dotfuscator's support for Declarative Obfuscation, see Declarative Obfuscation via Custom Attributes. That section describes the Feature Map and lists the native feature strings that Dotfuscator understands.

From the toolbar, you can add, edit, and remove feature map strings. The Add and Edit buttons bring up a dialog that allows you to map feature strings to supported Dotfuscator features. You can also delete a feature map string by selecting it and pressing the Delete key.

User Defined Assembly Load Path

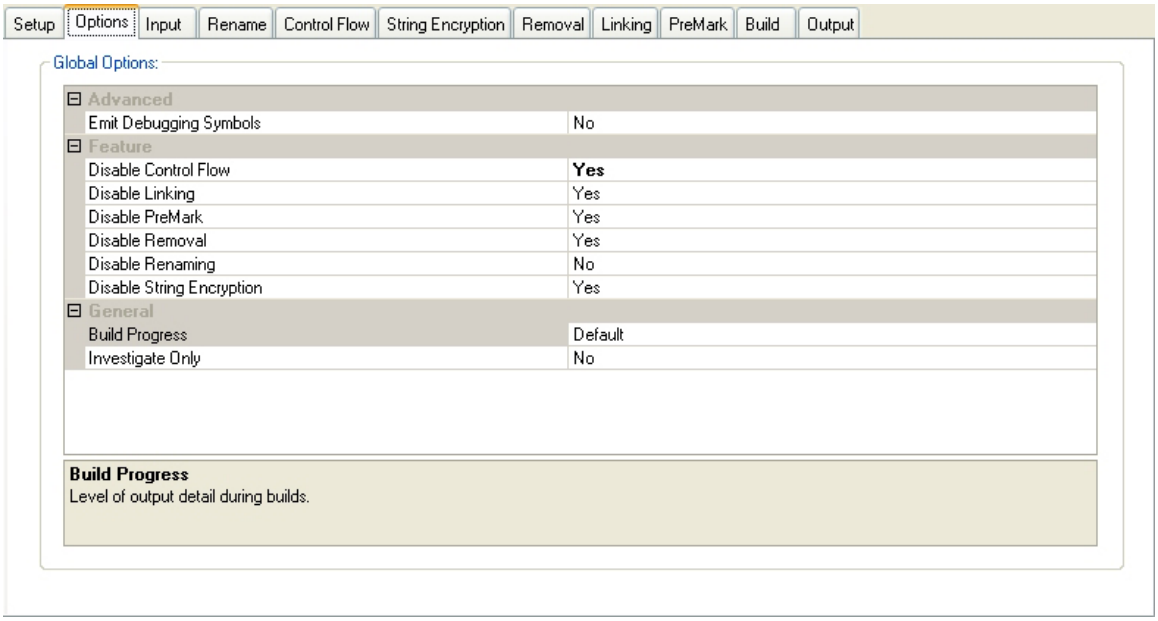
Using this editor, you can edit your project's user defined assembly load path.

For details, see the online user's guide.

The Options Tab

This tab allows you to set the global options for the Project. You can also selectively enable or disable Dotfuscator's features, such as renaming, from this tab. Here is a summary of the options you can set:

- Disable [feature]. Dotfuscator allows you to enable or disable each of its transforms.
- Build Progress. This controls the verbosity of Dotfuscator's output during a build.
- Investigate Only. This tells Dotfuscator to generate reports but no output assemblies.



Configuring

The standalone GUI allows you to configure settings independently for each feature. Each feature has an editor that has its own tab on the standalone GUI. The editors are fully described in the following topics:

- Renaming Editor
- Control Flow Editor
- String Encryption Editor
- Removal Editor
- Linking Editor
- PreMark Editor

Building the Project

There are three ways to Dotfuscate in the standalone GUI:

- You can click on the Build button found on the Build Tab.
- You can click on the Build Project button on the Toolbar.
- You can click from the Menu: File/Build.

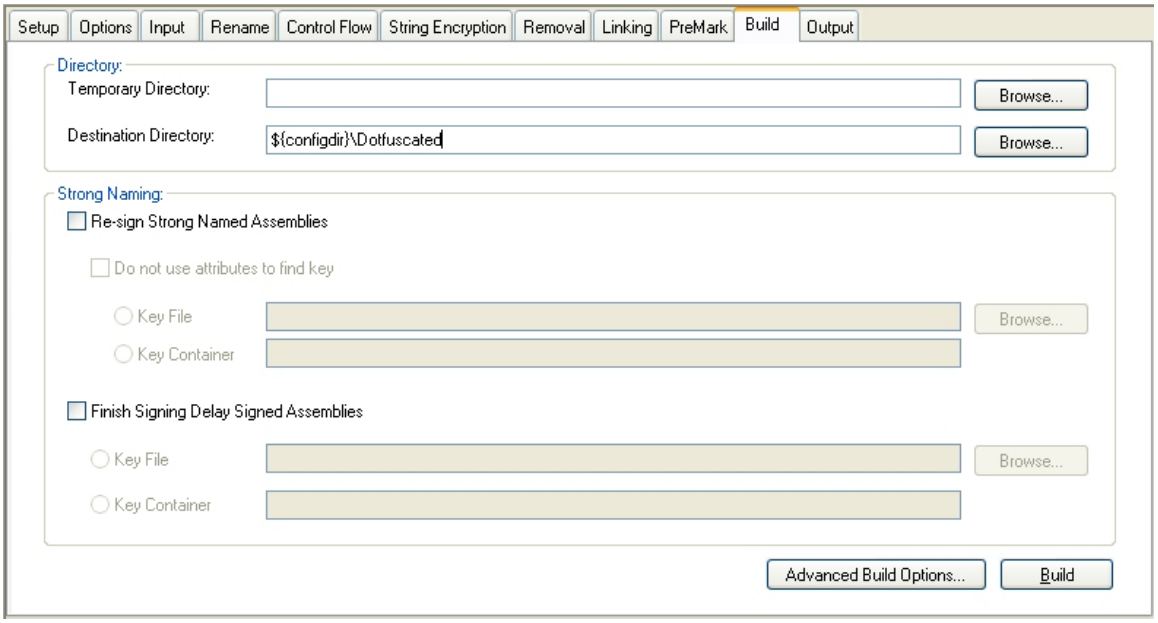
During and after the build, you can see Dotfuscator's output in the console area.

The Build Tab

We previously discussed using the Build tab to specify the destination directory. This tab has several other purposes, described below.

Build Directories

In addition to the destination directory where output assemblies are written, you can specify a Temporary directory for Dotfuscator to use for scratch files during processing. This is optional. If not specified, the Temp directory specified by the Windows environment will be used. If you know the specific Temporary Directory you want to use for Dotfuscator scratch files, you can type it in directly on the Build tab or use the Browse button to select one.



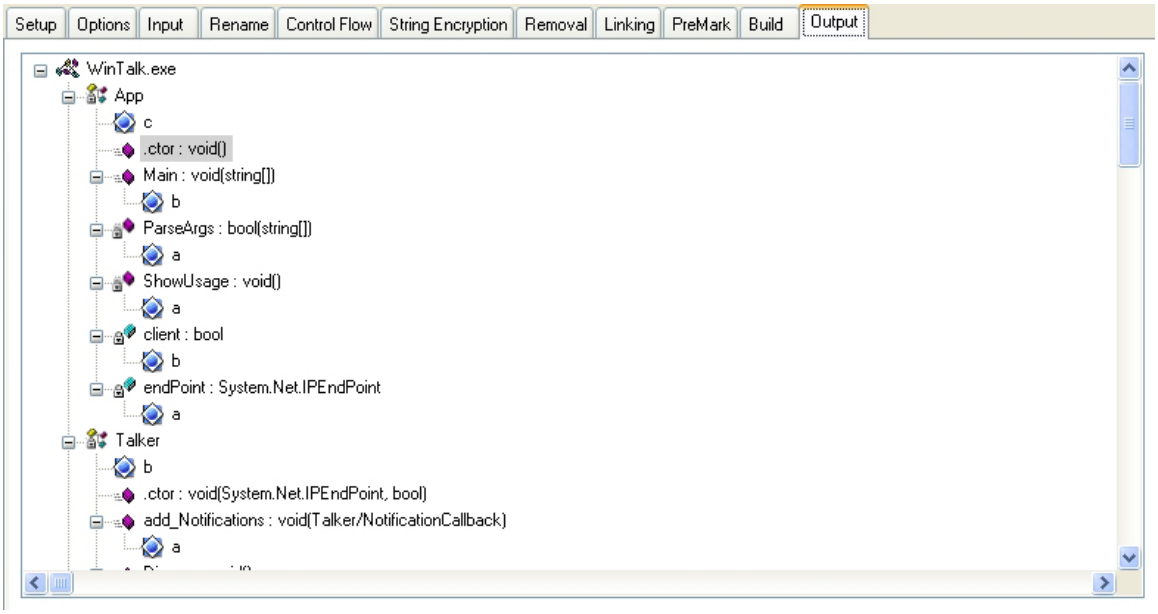
Advanced Build Options

PRO The Advanced Build Options dialog allows you to specify build events for your Dotfuscator project. For each event, you can specify an external program ("Program Path") that will run when the event occurs. You can also specify a working directory and command line options for the program.

For details, see the online user's guide.

The Output Tab

Once your project is Dotfuscated, you can inspect the results from the Output Tab.



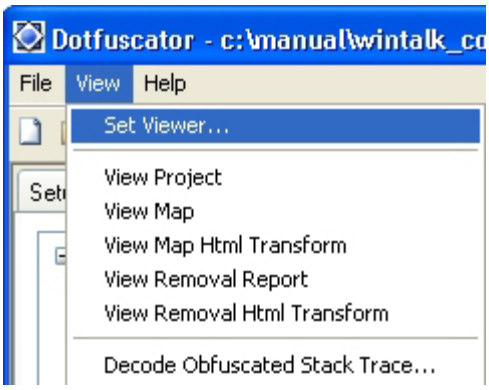
Here you can browse the tree view and see how Dotfuscator renamed your types, methods, and fields. The new names appear as child nodes under the original nodes in the tree.

Viewing Project Files and Reports

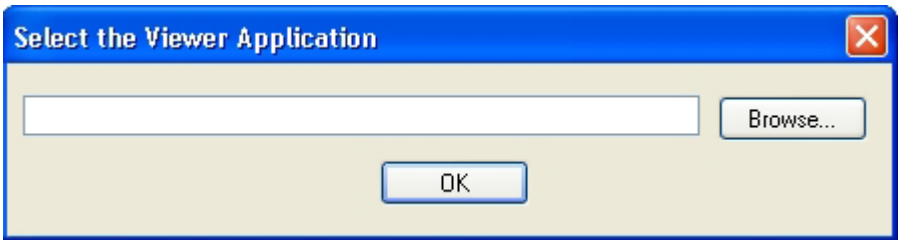
The standalone GUI's View menu allows you to specify an external program to view the XML and HTML documents that Dotfuscator produces.

Setting the Viewer Utility

Dotfuscator allows you to directly view the XML Configuration, mapping, and report files for your project by loading them in an XML viewer utility of your choice. You specify the utility you want to use by choosing View/Set Viewer from the menu. The viewer utility can be any application that displays text files.



The Select the Viewer Application window appears. You can directly type in the Viewer Application filename and its path. Click on the OK button to confirm.



Alternatively, you can browse your file system for the Viewer Application. To activate the Select Viewer dialog box, click on the Browse button.

Using an External Viewer

You are able to use your previously specified external XML Viewer Utility to inspect the Project Configuration File or the Map file you specified to be created when Dotfuscator runs.

- To load the Project Configuration file in the External Viewer, select View Project from the standalone GUI's View menu.
- To load the Map file in the Viewer, select View Map from the standalone GUI's View menu. This menu option will display the XML version of the report.

The Help Menu

The standalone GUI's help menu contains the following items:

- Help. This item brings up Dotfuscator's online help (this document).
- Register Product. This item is only enabled if your copy has not yet been registered. When selected, it will bring up Dotfuscator's registration dialog. See Registering Dotfuscator for more information.
- What's New. When selected, this will launch a browser that will take you to Dotfuscator's home page.
- Check For Updates. This item allows you to toggle Dotfuscator's automatic update checking.
- Check For Updates Now. When selected, Dotfuscator will immediately check the web for updates.
- About Dotfuscator. When selected, this will bring up Dotfuscator's About Box which displays user and version information.

GUI Configuration Editors

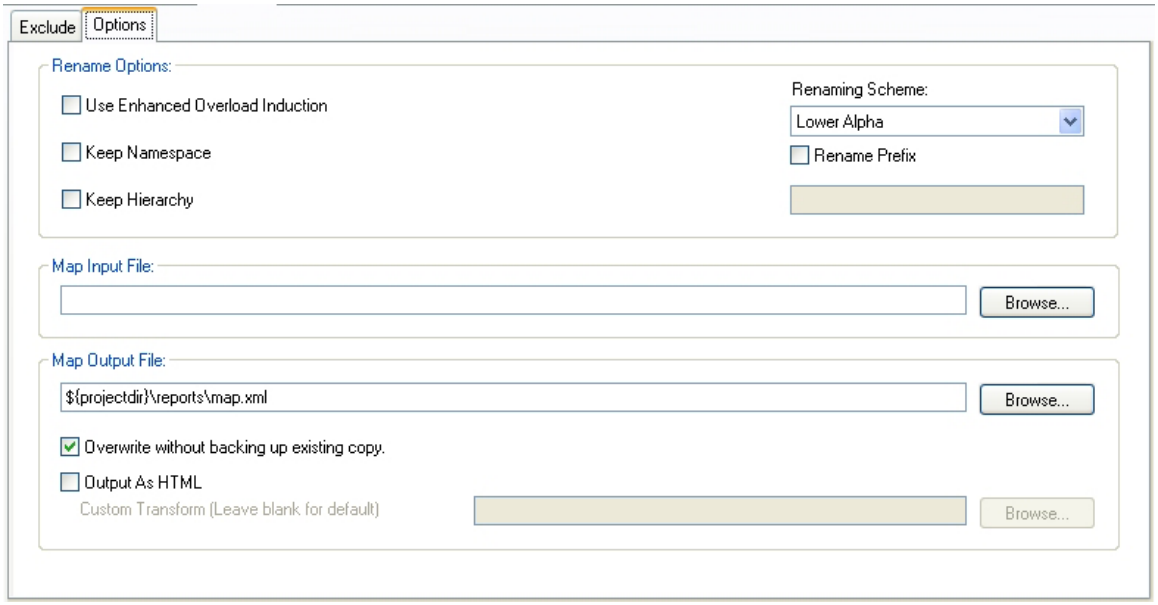
This section describes the editors used to configure Dotfuscator's main features. If you are using the standalone GUI, every editor is available as a tab on the main UI.

The Renaming Editor

The Renaming editor displays two configuration tabs: the Exclude tab, which is used to graphically set custom exclusion rules, and the Options tab, which is used to configure other options related to renaming.

The Rename Options Tab

The Rename Options tab is used to set renaming options and identify map files to be used for incremental obfuscation.



The Renaming Option section contains three checkboxes used to select or deselect the Enhanced Overload Induction, Keep Namespace, and Keep Hierarchy options. These are fully described in Class Renaming Options

STD PRO With Professional and Standard Editions, you can select your preferred renaming scheme from the dropdown list.

STD PRO You can also select a rename prefix by checking the Rename Prefix checkbox.

PRO The Input Map File section allows you to specify a map file from a previous run so that the naming scheme is retained across successive Dotfuscator runs.

CE STD PRO The Output Map File section allows you to specify a map file to be built for the next run in preparation for retention of the naming scheme across successive Dotfuscator runs and to support browsing of the renamed results. Specific details of this feature are presented in the Output Mapping File section. Like the Mapping Input File, two approaches are available. If you know the name and path of the mapping file you want to use, you can type it directly into the edit box. Alternatively, you can browse your file system for the intended file location.

The Browse button on the right of the edit box brings up the Select Map Output File window that provides a familiar navigational dialog. This section has an Overwrite Map File check box that allows you to specify whether you want an existing map file overwritten during a run without backing it up.

STD PRO The output mapping file can also be written out as a human readable, HTML formatted document (in addition to the XML formatted version). This report provides a quick cross reference that allows you to quickly navigate through the renamed types, fields, and methods.

The Rename Exclude Tab

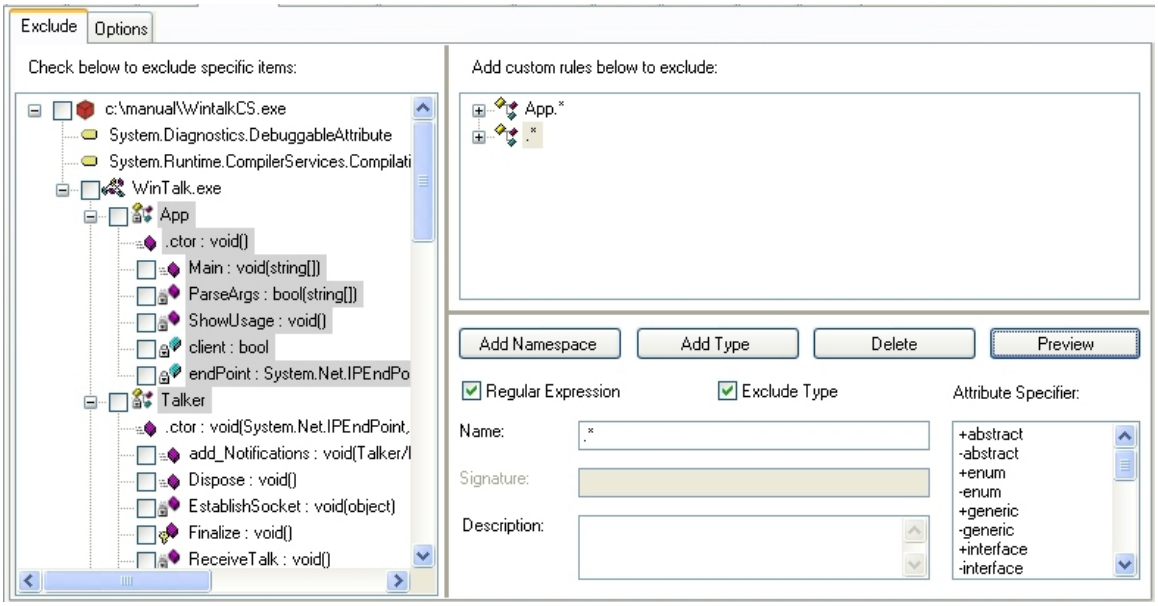
The Rename Exclude Tab gives you complete granular control over all parts of your program that you may wish to exclude from the renaming process.

You may exclude specific items from renaming by browsing the tree view of your application and checking the items you want to exclude. In addition, you may visually create your own custom rules for selecting multiple items for exclusion.

To help you fine-tune your exclusion rules, you can preview their effects at any time. The application tree view will shade all items selected for exclusion. You can preview the cumulative effects of all rules, or preview a specific rule that you select.

See the section on Using the Graphical Rules Editing Interface for detailed information about working with Inclusion and Exclusion Rules.

See the section on Renaming Exclusion Rules for a detailed discussion of renaming exclusion rules.



The Control Flow Editor


The Control Flow editor displays two configuration tabs: the Exclude tab, which is used to graphically set custom exclusion rules, and the Options tab, which is used to configure other options related to control flow obfuscation.

For details, see the online user's guide.

The String Encryption Editor


The String Encryption editor displays only one configuration tab: the Include tab, which is used to graphically set custom inclusion rules for string encryption.

For details, see the online user's guide.

The Removal Editor


The Removal editor displays three configuration tabs: the Include Triggers tab, which is used to graphically set custom inclusion rules for trigger methods; the Conditional Includes tab, which is used to graphically specify types for conditional inclusion; and the removal options tab, used to configure the removal report.

For details, see the online user's guide.

The Linking Editor


The Linking editor provides easy drag and drop functionality that allows you to quickly map your input assemblies to one or more output assemblies. You can then configure the linker for each output assembly.

For details, see the online user's guide.

The PreMark Editor


The PreMark editor allows you to select assemblies for watermarking and to set up the watermark that will be applied to the selected assemblies.

For details, see the online user's guide.

Using the Graphical Rules Editing Interface

The Dotfuscator GUI uses a common interface to graphically specify rules for including and excluding elements in your application. The rules editing interface is used to set rules for the following operations:

Renaming exclusion rules.

Control Flow Obfuscation exclusion rules.

String Encryption inclusion rules.

Removal Trigger Method selection rules.

Removal Conditional Includes selection rules.

This section explains how to get the most out of the rules editing interface.

There are two methods of creating rules. The first is by checking individual elements in the application tree view. By doing this, you generate a simple rule that selects that particular element. The second method is by adding "nodes" to the rule editing view. This type of rule is more powerful and customizable. You can use regular expressions and other selection criteria, based on the type of rule. This method also provides ways to preview the items selected by each rule.

The next sections discuss each method in detail. You may use either method or both methods in conjunction for maximum flexibility.

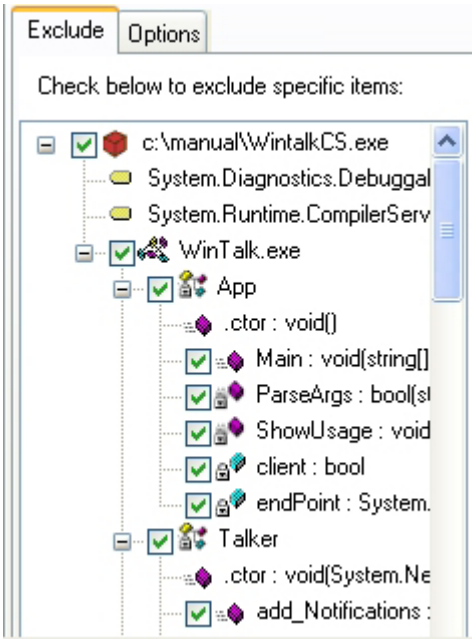
Selecting Individual Elements

You can create selection rules for individual elements by simply checking the box next to the item in the application tree view. You can check assemblies, modules, types, methods, and fields in this manner.

Selecting an Assembly

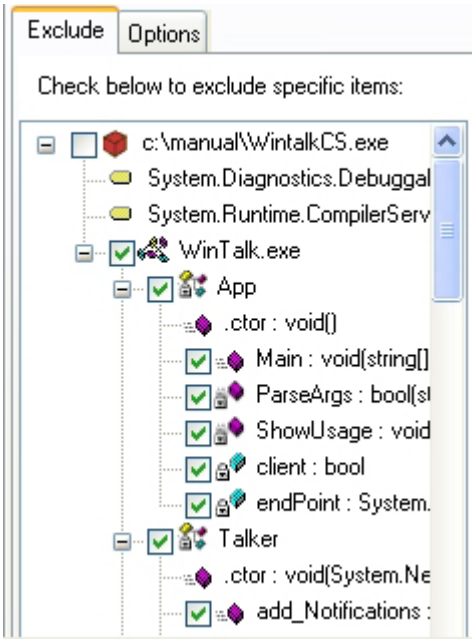
The top-most nodes in the application tree view represent the assemblies. If you check an assembly node, all child nodes will also become checked. This reflects the fact that

selecting an assembly means that you are in fact selecting all items contained in that assembly: modules, types, and their members.



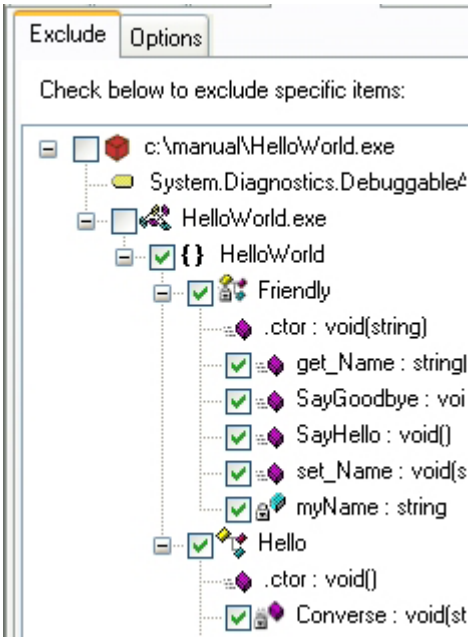
Selecting a Module

The nodes immediately below an assembly node in the application tree view represent the modules that make up the assembly (in most cases there is one module per assembly). If you check a module node, all child nodes will also become checked. This reflects the fact that selecting a module means that you are in fact selecting all items contained in that module: "global" methods and fields, types, and their members.



Selecting a Namespace

Namespace nodes are child nodes of module nodes in the application tree view. If you check a namespace node, all child nodes will also become checked. This reflects the fact that selecting a namespace means that you are in fact selecting all items contained in that namespace: types and their members.

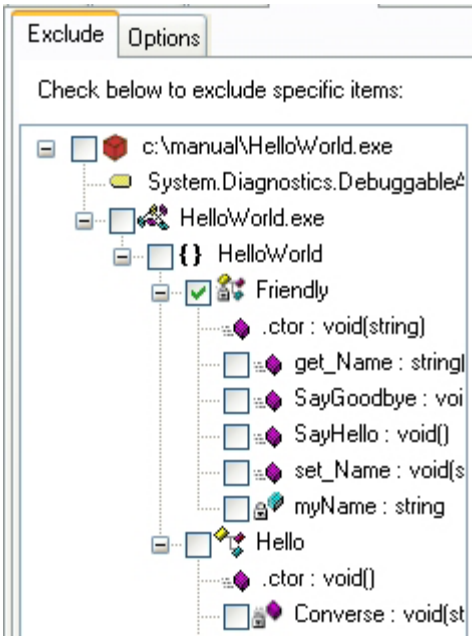


Selecting a Type

Type nodes appear under module or namespace nodes. Nested types are represented at this same level, with a name prefixed with the parent type name(s) delimited with the '/' character. If you check a type node, one of two things happens, depending on what type of rule you are creating.

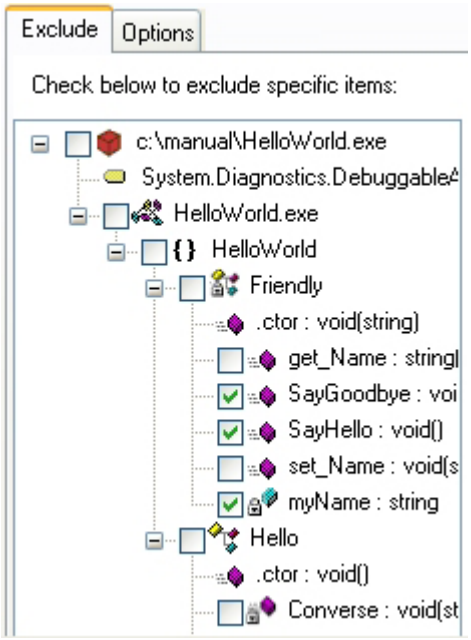
If you are creating a renaming exclusion rule, child nodes remain unchecked. This reflects the fact that types are selected independently of their members for renaming exclusion rules. Checking a type node will generate a rule that excludes *just the type name* from renaming.

If you are specifying any other kind of rule, all child nodes will also become checked. This reflects the fact that in these cases, selecting a type means that you are in fact selecting all members defined by that type.



Selecting a Member

Members can be methods or fields. Member nodes can appear under module nodes in the case of "global" members; more commonly, they appear under type nodes. Checking a member node will generate a rule that selects that member.



Creating Custom Rules

You can create custom rules by adding nodes to the rule editing view. Depending on the type of rule, you can attach regular expressions and/or other selection criteria to the rule. Once the rule is configured, you can preview its effects by right clicking on the node and selecting "preview" from the menu. Items selected by the rule will be shaded in the application tree view. Each type of rule is covered in the following sections.

Selecting By Namespace

A namespace rule will select all types and their members in matching namespaces.

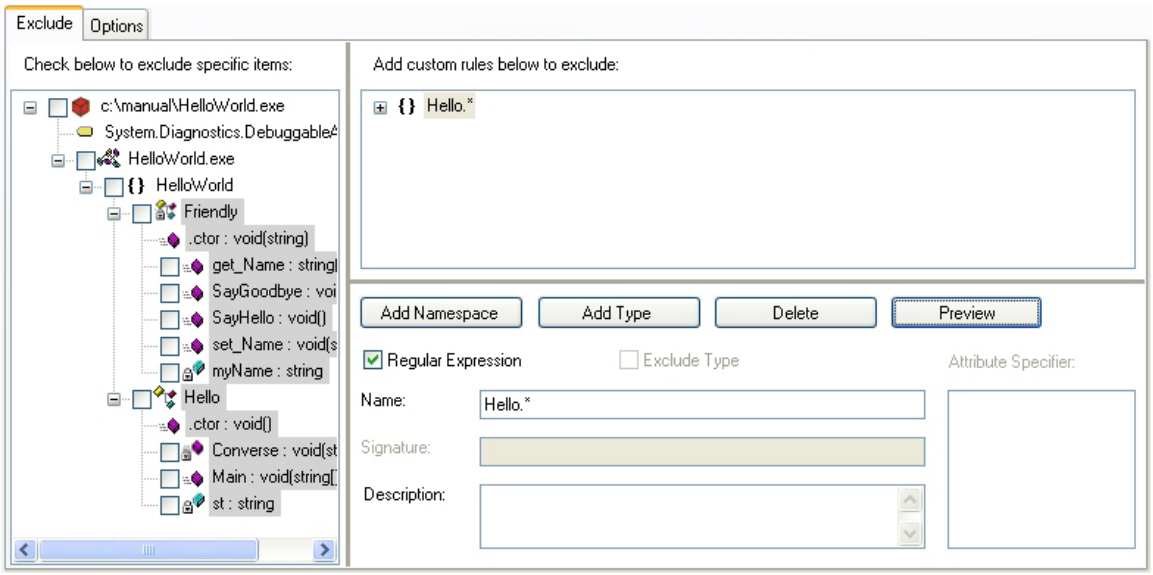
Namespace Name

You create a namespace rule by clicking the Add Namespace button, then typing a name in the Name field. The name will be interpreted as a regular expression if the "Regular Expression" checkbox is checked; otherwise the name will be interpreted literally (and thus will match at most one namespace).

Namespace Rule Node

The corresponding node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression. You can preview the items selected

by the rule by right clicking on the node and selecting the preview option from the menu.



Selecting By Type

A type rule will select differently depending on what type of rule you are creating.

If you are creating a renaming exclusion rule, the rule will select just the type name for exclusion (provided the ExcludeType checkbox is checked), leaving members alone.

If you are specifying any other kind of rule, the rule will select zero or more types and all their members. This reflects the fact that in these cases, selecting a type means that you are in fact selecting all members defined by that type.

Type Name

You create a type rule by clicking the Add Type button, then typing a name in the Name field. The name will be interpreted as a regular expression if the "Regular Expression" checkbox is checked; otherwise the name will be interpreted literally. The name must be a fully qualified type name that includes the namespace and parent class information if it is a nested type.

Type Attribute Specifier

In addition to type name, you can also select based on type attribute specifiers, using the values provided in the "Spec" list box. A '-' preceding an attribute specifier negates the attribute (i.e. it selects all types that do not have the specified attribute). You can

select multiple attributes from the list; the criteria implied by multiple selections are logically ANDed together (that is, the set of selected types is the intersection of all types that match each attribute specifier.). For example, you can select types that are both public and abstract by selecting "+public" and "+abstract" from the list.

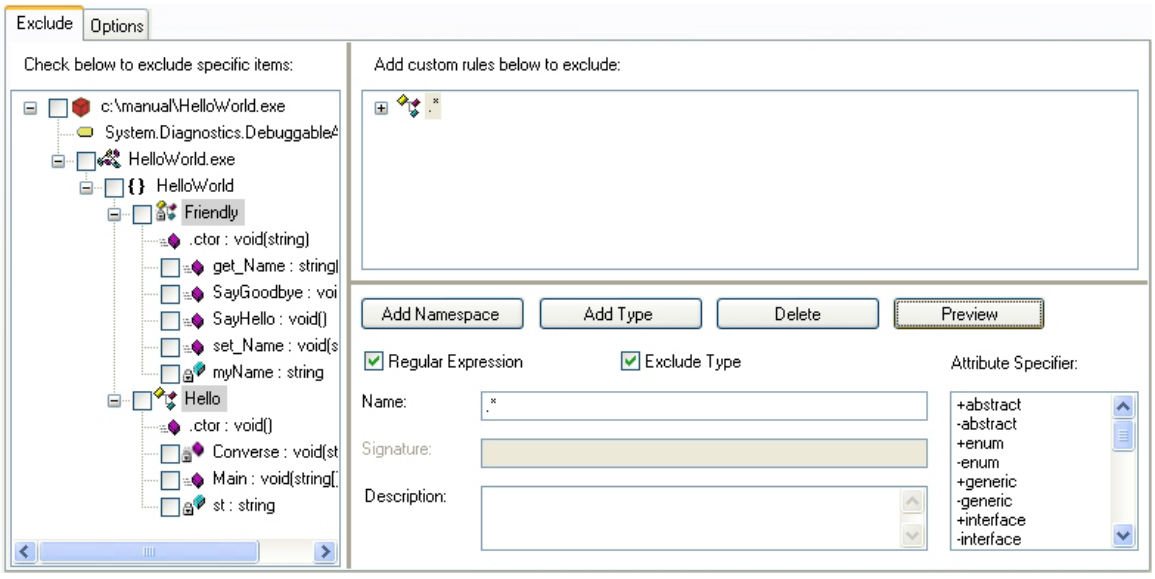
The attribute specifications are logically ANDed with the type name, so if you want to select all types with a specific set of attributes, you need to provide a regular expression for the type name that selects all types (i.e. ".*").

Exclude Type Checkbox

The Exclude Type checkbox is only active if you are working with renaming exclusion rules. If checked, the rule will exclude the names of matching types from renaming and allow you to provide additional rules for selecting members of matching types. If left unchecked, the rule will still select matching types for the purposes of applying rules to members of the types, but it will not select the type name. In this manner, you can write renaming exclusion rules that exclude methods and fields, but allow type names to be obfuscated.

Type Rule Node

The corresponding node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers associated with it. You can preview the types selected by the rule by right clicking on the node and selecting the preview option from the menu.



In the screen shot, a type rule is defined that selects the names of all concrete (not abstract) types for exclusion from renaming.

Selecting By Method

Method rules are qualified by type rules, so they appear in the rules view as children of type nodes. A method rule will select all methods (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include method name, method attributes, and signature.

Method Name

You create a method rule by right clicking on the parent type rule's node and selecting "Add Method", then typing a name in the Name field. The name will be interpreted as a regular expression if the "Regular Expression" checkbox is checked; otherwise the name will be interpreted literally.

Method Attribute Specifier

In addition to method name, you can also select based on method attribute specifiers, using the values provided in the "Attribute Specifier" list box. A '-' preceding an attribute specifier negates the attribute (i.e. it selects all methods that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically ANDed together (that is, the set of selected methods is the intersection of all methods that match each attribute specifier.). For example, you can select methods that are both public and virtual by selecting "+public" and "+virtual" from the list.

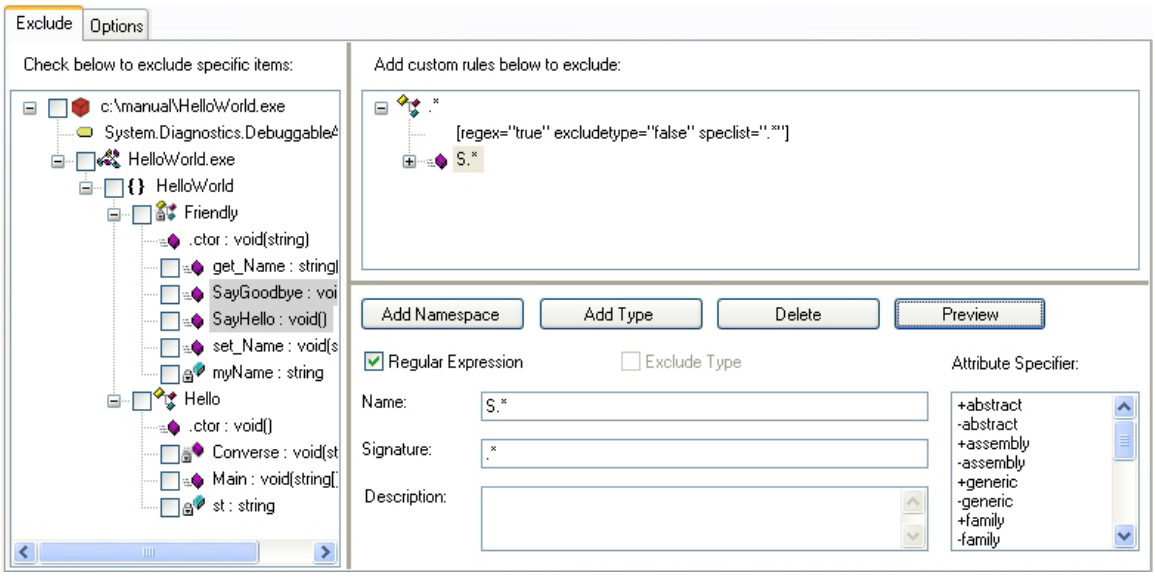
The attribute specifications are logically ANDed with the method name and signature list, so if you want to select all methods with a specific set of attributes, you need to provide a regular expression for the method name that selects all methods (i.e. ".*").

Method Signature

You can also select methods by signature. A signature is a list of types that match the types in the method's parameter list. The method signature is logically ANDed with the method name and attribute specifications, so if you want to create a rule that selects methods regardless of signature, you need to provide a regular expression for the signature that selects all signatures (i.e. ".*"). This is the default value. *An empty signature list will select methods with no parameters.*

Method Rule Node

The corresponding method node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers, and/or a signature associated with it. You can preview the items selected by the rule by right clicking on the node and selecting the preview option from the menu.



In the screen shot, a method rule is defined that selects the names of all public methods (in all types) whose names start with "S".

Selecting By Field

Field rules are qualified by type rules, so they appear in the rules view as children of type nodes. A field rule will select all fields (in all types matched by the parent type rule) that match your criteria. Supported matching criteria include field name and field attributes.

Field Name

You create a field rule by right clicking on the parent type rule's node and selecting "Add Field", then typing a name in the Name field. The name will be interpreted as a regular expression if the "Regular Expression" checkbox is checked; otherwise the name will be interpreted literally.

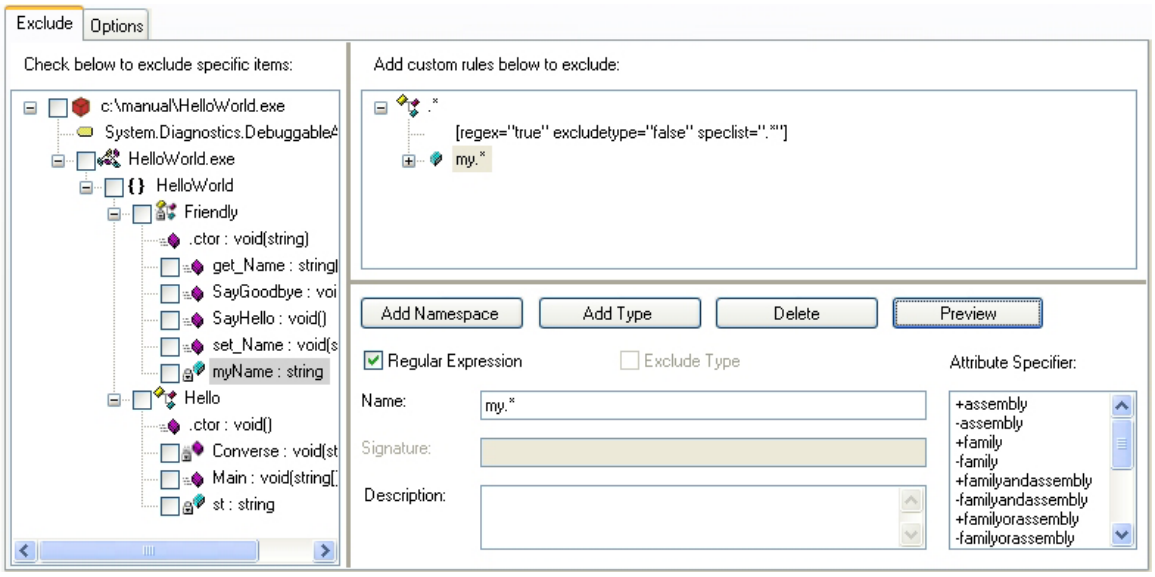
Field Attribute Specifier

In addition to field name, you can also select based on field attribute specifiers, using the values provided in the "Attribute Specifier" list box. A '-' preceding an attribute specifier negates the attribute (i.e. it selects all fields that do not have the specified attribute). You can select multiple attributes from the list; the criteria implied by multiple selections are logically ANDed together (that is, the set of selected fields is the intersection of all fields that match each attribute specifier.). For example, you can select fields that are both public and static by selecting "+public" and "+static" from the list.

The attribute specifications are logically ANDed with the field name, so if you want to select all fields with a specific set of attributes, you need to provide a regular expression for the field name that selects all fields (i.e. ".*").

Field Rule Node

The corresponding field node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression and whether the rule has attribute specifiers. You can preview the fields selected by the rule by right clicking on the node and selecting the preview option from the menu.



In the screen shot, a field rule is defined that selects the names of all fields (in all types) whose names start with "my".

Selecting by Custom Attribute

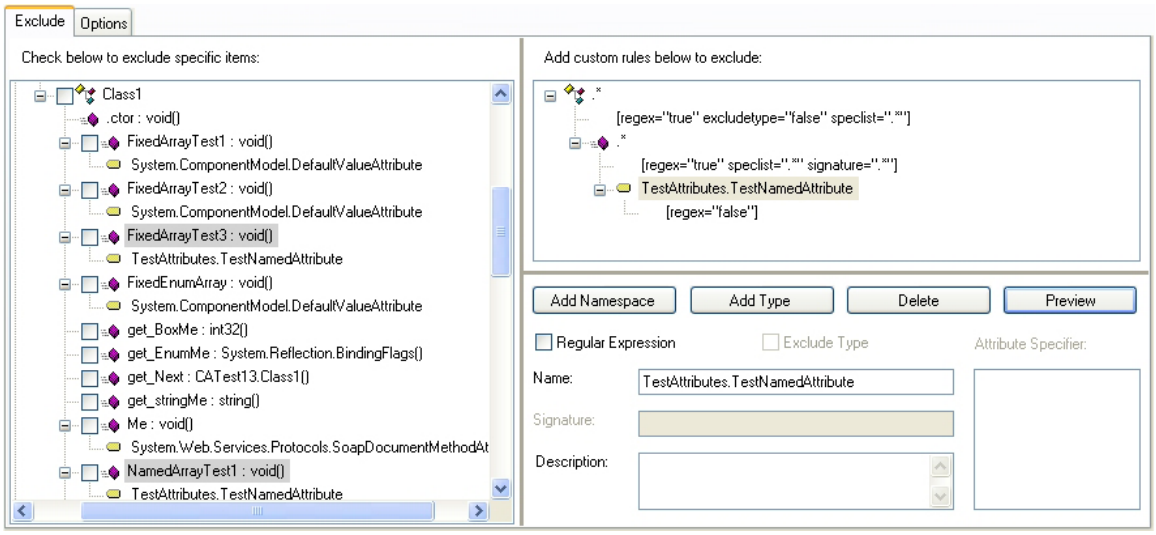
Custom attribute rules are qualified by type, method, or field rules, so they appear in the rules view as children of type, method, or field nodes. A custom attribute rule will select all items selected by the parent node that are also annotated with a matching custom attribute.

Custom Attribute Name

You create a custom attribute rule by right clicking on the parent type, method, or field rule's node and selecting "Add Custom Attribute", then typing a name in the Name field. The name will be interpreted as a regular expression if the "Regular Expression" checkbox is checked; otherwise the name will be interpreted literally.

Custom Attribute Rule Node

The corresponding custom attribute node displayed in the rule editing view has a child element that indicates whether the rule is a regular expression. You can preview the types, methods, or fields selected by the rule by right clicking on the node and selecting the preview option from the menu.



In the screen shot, a custom attribute rule is defined that selects all methods that are annotated with a custom attribute named "TestAttributes.TestNamedAttribute".

Editing and Deleting Rules

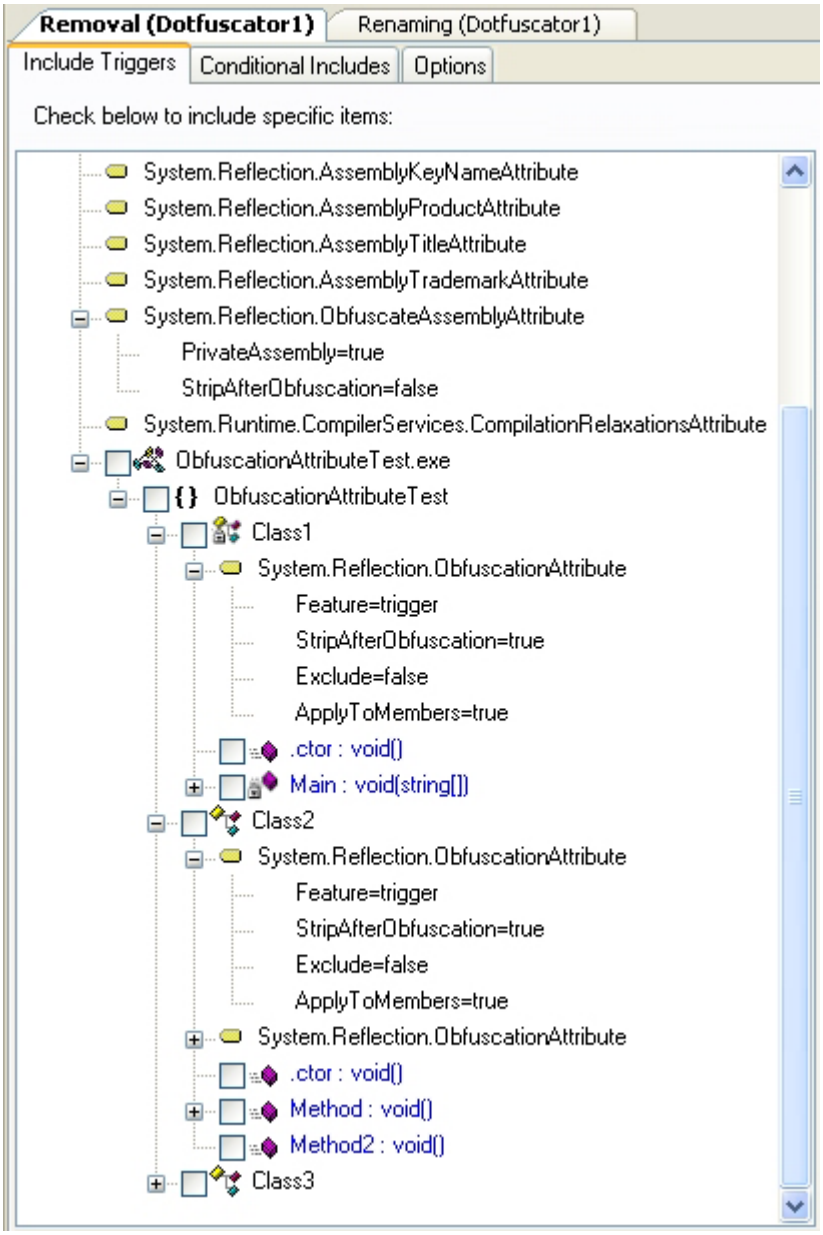
To edit an existing rule, simply click on the rule in the rule editing view. You can then use controls below the view to edit the values associated with the node (e.g. name, attribute specifier list, etc.).

To delete a rule, click on the rule in the rule editing view, and press the Delete button.

Using Declarative Obfuscation with Rules

The rules editor provides support for Declarative Obfuscation by displaying the arguments of all obfuscation attributes (i.e. System.Reflection.ObfuscateAssemblyAttribute and System.Reflection.ObfuscationAttribute) in the application tree view. Items in the application tree view (types, methods, fields) that are selected by an obfuscation attribute appear in blue.

In the screenshot below, the methods of Class1 and Class2 are marked as removal triggers using ObfuscationAttributes. Each attribute's properties and values are expanded in the view.



Previewing Rules

To preview the effects of a single rule in the rule editing view, right click on the rule's node and select "Preview" from the menu. Items selected by the rule will appear shaded in the application tree view.

To preview the combined effects of all the rules defined in the rule editing view, click on the Preview button. All items selected by applying all the rules will appear shaded in the application tree view.

Reference

This section contains pointers to Dotfuscator's configuration and mapping file DTDs, as well as information about Dotfuscator's product registration.

Additional resources such as articles, whitepapers, and tutorials can be found online at www.preemptive.com/downloads/Documentation.html.

dotfuscator_v2.1.dtd

The dotfuscator_v2.1.dtd DTD describes the format of the configuration file produced by Dotfuscator version 3.0. A copy is available at www.preemptive.com/dotfuscator/dtd/dotfuscator_v2.1.dtd.

dotfuscatorMap_v1.1.dtd

The dotfuscatorMap_v1.1.dtd DTD describes the format of the renaming map file produced and consumed by Dotfuscator version 3.0. A copy is available at www.preemptive.com/dotfuscator/dtd/dotfuscatorMap_v1.1.dtd.

Registering Dotfuscator

For Dotfuscator Community Edition, registration is optional. When you register your copy, you receive free access to support resources located on the web at www.preemptive.com/support/DotfuscatorSupport.html.

You may register by either pressing the Register Now button (located on the command line pop up and on the GUI splash screen), or by selecting the "Register Product" menu option on the GUI.

Registration is a two-step process. In the first step, you fill out the registration form; Dotfuscator will then send this information to the registration server via email or via the web, depending on your choices. Upon successful receipt, the server will generate a serial number and confirmation code. This information will be emailed back to you, using the address that you provided.

After this step is completed, the next time you run Dotfuscator (or try to register again), you will be prompted for your serial number and confirmation code. Once you enter this information and it is validated, registration is complete.

Samples

If you are looking for a simple example that will let you quickly get familiar with Dotfuscator, see Getting Started with Dotfuscator.

The getting started and other samples are available on the PreEmptive Solutions web site at www.preemptive.com/support/dotfuscator/samples.