Instrumentation and Obfuscation Quick Start Guide

Contents

Instrumentation 2
Quick Start 2
Best Practices for Specific Platforms 4
Instrumentation Customization 4
Debugging 6
Obfuscation 7
Renaming 7
Library Mode 10
Control Flow 11
String Encryption 12
Removal 14
Watermarking (PreMark) 14
Linking 15
XAML Considerations 15
Smart Obfuscation Rules & Warnings 16
Property and Event Metadata 16
Map Files & Stack Trace Decoding 16
Exclusions/Inclusions in Code 16
Troubleshooting Build Errors 17
Troubleshooting Runtime Errors 17

 ©2012 PreEmptive Solutions, LLC

Instrumentation

Quick Start

Required Attributes

- ***BusinessAttribute***
  - Ensures that your data is correctly partitioned such that only users from your company can view your data.
    - PreEmptive will supply you with a ***CompanyKey***
  - Must be added to a root-level assembly.

Attribute Properties	
CompanyKey	7d2b02e0-064d-49a0-bc1b-4be4381c62d3
CompanyName	PreEmptive Solutions

- ***ApplicationAttribute***
  - Enables your data to be reported on per application.
    - There is a GUID generator button to create new applications.
  - Must be added to a root-level assembly.

Attribute Properties	
ApplicationType	Windows Phone 7
Guid	3e22c5ac-ea79-4f85-9f58-534a19c4a6c9
Name	Dotfuscator Sample - Windows Phone XNA Game
Version	1.0.0.0

- ***SetupAttribute***
  - Sets up to send messages to the correct endpoint. Use the *Commercial* endpoint for the Runtime Intelligence Service, and a *Custom* endpoint for PA for TFS.
  - Must be added to a method; usually added to all the entry points of an application.
  - Sends ***Application.Start*** & ***Session.Start*** messages.
    - *These are not surfaced inside PreEmptive Analytics for TFS Community Edition BUT Dotfuscator Community Edition can instrument these and send them to a Custom endpoint*.
- ***TeardownAttribute***
  - Sends ***Session.Stop*** & ***Application.Stop*** messages.
  - Must be added to a method; usually added to all the exit points of an application.
  - Cleans up reporting state and turns off ability to send messages.

Exception Reporting Attributes

- ***ExceptionTrackAttribute***
  - Three types:
    - ***Unhandled***
    - ***Caught*** (*NOT supported in PA for TFS Community Edition)*
    - ***Thrown*** (*NOT supported in PA for TFS Community Edition)*
  - Can be added at the assembly level or method level.

- Supported by all editions of PA for TFS:
    - Assembly-level Unhandled: Hooks into the AppDomain's UnhandledException event.
    - Method-level Unhandled: Wraps the method in a try/catch (with rethrow).
- **ONLY supported in PA for TFS Professional:**
    - Assembly-level Caught: Equivalent to putting a Method-level Caught on every method.
    - Method-level Caught: At the beginning of each existing catch block in the method, report the exception that was caught.
    - Assembly-level Thrown: Equivalent to putting a Method-level Thrown on every method.
    - Method-level Thrown: At every instance of the "throw" instruction, report the exception object that is about to be thrown.

Feature Reporting Attributes

- ***FeatureAttribute***
    - Placed on methods that should be considered the entry point to a particular application feature.
        - *These are not surfaced inside PreEmptive Analytics for TFS Community Edition BUT Dotfuscator Community Edition can instrument these and send them to a custom endpoint.*
    - There are two types:
        - ***Tick*** – Just a simple counter that indicates this feature has been used.
        - ***Duration*** – Requires two feature attributes with the same name. Specify a ***FeatureEventType*** of ***Start*** on one and ***Stop*** on the other. The report will include the amount of time spent in that feature.
            - They can be added to the same method, and the Start will be injected at the top of the method, the Stop at the bottom.

Attribute Properties	
ExtendedKeyMethodArgume	
ExtendedKeySourceElemen	None
ExtendedKeySourceName	
ExtendedKeySourceOwner	
FeatureEventType	Tick
Name	**Game Over By Points**

- ***PerformanceProbeAttribute***
    - Collects and sends the amount of memory currently used by the application.
- ***SystemProfileAttribute***
    - Collects and sends the DeviceName, DeviceManufacturer, and DeviceTotalMemory.
- ***ExceptionTrackAttribute***
    - See the section devoted to this above.

 ©2012 PreEmptive Solutions, LLC

Best Practices for Specific Platforms

Windows Phone platform

It is not possible to send messages on teardown on the phone platform. During application shutdown on the phone, the mechanism used for checking whether a network connection reports that there exists a network connection, but it cannot actually be used for message sending at that point.

To get around this, all ***TeardownAttribute*** messages are automatically persisted to Isolated Storage rather than sent immediately. Unfortunately, no other attributes are currently capable of this behavior, so if a ***FeatureAttribute*** is hit during application shutdown (say, on the same method where the ***TeardownAttribute*** lives), then that data will be lost.

Silverlight

Two possible approaches:

- Approach 1: Count every app entrance (regardless of whether it's a fresh app run, rehydration, etc.) as a new run, gathering data for each portion independently.
    - Put setup on both Application_Launching() and Application_Activated()
    - Put teardown on both Application_Closing() and Application_Deactivated()
    - Pros – More correct count of "incomplete sessions" since every entrance/exit is accounted for.
    - Cons – Depending on your definition, incorrect application runs count, since a run that gets temporarily interrupted and resumed will count as two runs. This will also impact statistics on features per run, etc.
- Approach 2: Only count every fresh app run.
    - Only put setup on Application_Launching(), and only put teardown on Application_Closing()
    - Pros – Depending on your definition, correct count of application runs, features used per run, etc.
    - Cons – More incorrect count of "incomplete sessions."

XNA

We have found that this works best if the ***SetupAttribute*** is placed on the Initialize() method (the one that overrides void Microsoft.XNA.Framework.Game.Initialize()), and the ***TeardownAttribute*** placed on any method tied to the Game.Exiting event.

Instrumentation Customization

User Opt-In

The ***SetupAttribute*** provides the ability to specify an ***OptInSource***, which is a boolean (that can be read at runtime from a method's return value, property, field, etc.) that tells PreEmptive Analytics (PA) whether or not it should send analytics messages for the current session. If the ***OptInSource*** is not specified, it defaults to sending messages.

Offline Caching

(**not available with Dotfuscator Community Edition**)

Instrumented applications have the ability to store usage data in situations when network access is unavailable and then transmit the data when connectivity is restored. Usage data is stored in Isolated Storage. This behavior is enabled by default and default connectivity detection code is injected into instrumented applications.

Developers can override the default behavior by changing the ***OfflineStateSourceElement***. If the ***OfflineStateSourceElement*** value is changed to ***None*** then usage data will not be stored when the application is unable to connect to the network and that usage data will be dropped.

Custom Data

(**not available with Dotfuscator Community Edition**)

Most message types allow user defined data (in the form of key-value pairs) to be gathered and sent along with the message. To send extended key information, specify an ***ExtendedKeySource*** on the attribute corresponding to the message you wish to send.

Dotfuscator uses the ***ExtendedKeySource*** to generate code that gathers the key-value pairs at runtime. The ***ExtendedKeySource*** is an IDictionary or IDictionary<string,string> valued property, method, field, or method argument; it is the developer's responsibility to ensure that a correct value is available in the ***ExtendedKeySource*** at the time the attributed method is executed.

Custom Instance ID

The SetupAttribute provides the ability to specify an ***InstanceIdSource***, which is a string (that can be read at runtime a variety of ways) that is used to uniquely identify the current user of the application. This is typically populated with some sort of serial number value.

There are specific reports on the portal that show usage trends per ***InstanceId***.

If the ***InstanceId*** is not specified, it defaults to a GUID we store in isolated storage. Also:

- The username defaults to the user's anonymous ID (aka "ANID") if we can get it; if we can't get it, the username also falls back to the GUID we store in isolated storage.
- The ***MachineName*** defaults to the ***DeviceUniqueId*** if we can get it.

Choice of Endpoint

- Custom Endpoint
  - Use this for PA for TFS.
  - Users can also set up their own endpoint to catch, aggregate, and report data.
- PreEmptive's Commercial Runtime Intelligence Service
  - Requires contract with PreEmptive.
  - Better data retention period, storage allocation, etc.

Debugging

- Was the application properly instrumented?
  - The Dotfuscator build output should call out every attribute that it is injecting individually. If no such list appears, then it was not properly configured.
- Is any data leaving the app?
  - Check with Fiddler, WireShark, or similar tools (be sure to turn off SSL first)
  - The method with the ***SetupAttribute*** must be called before any ***FeatureAttributes*** (or others).
  - Are the messages being persisted to Isolated Storage (due to lack of network or app teardown considerations)?

 ©2012 PreEmptive Solutions, LLC

Obfuscation

Dotfuscator Community Edition only includes renaming and does not include a command line interface.

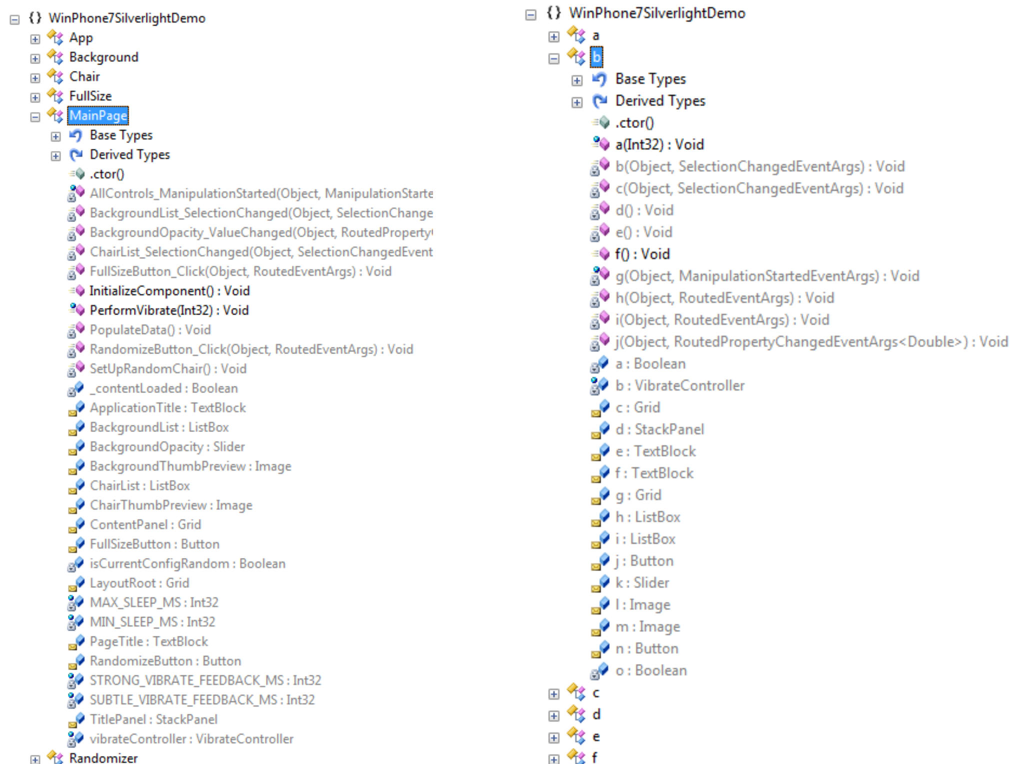
Renaming

Overview

Dotfuscator is capable of renaming all classes, methods, fields, properties and events 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.

Example: Before & After on the sample included in the install at

```
PreEmptive Solutions\Dotfuscator Windows Phone Edition
4.9\samples\cs\WinPhone7SilverlightDemo
```



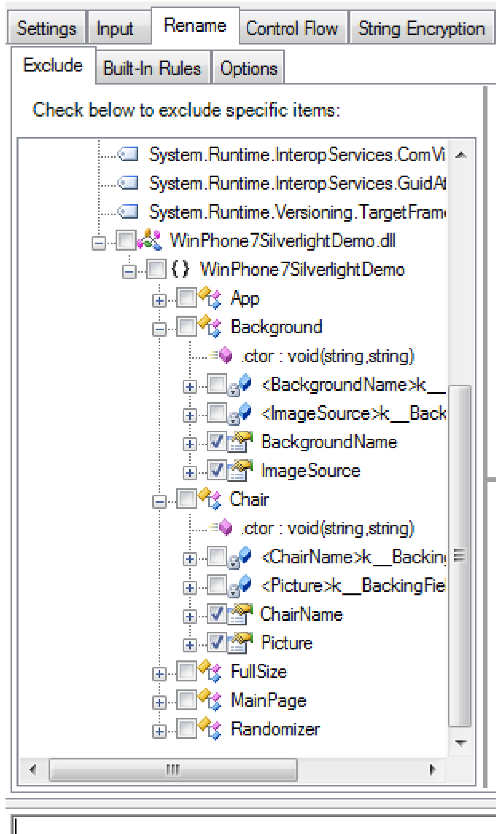
Limitations

Problems can occur with Renaming when any type of reflection is used (including XAML), configuration files that specify entry points, libraries that are called by other applications, etc. It is important to fully test applications after Renaming to ensure no such problems exist.

Specific Exclusions

Presuming library mode is turned off (explained later), Dotfuscator will try to rename everything it possibly can. Certain uses of reflection can cause Dotfuscator to not update all references to a particular

entity upon renaming. For instance, using the included sample Silverlight application, we've made 4 specific exclusions:

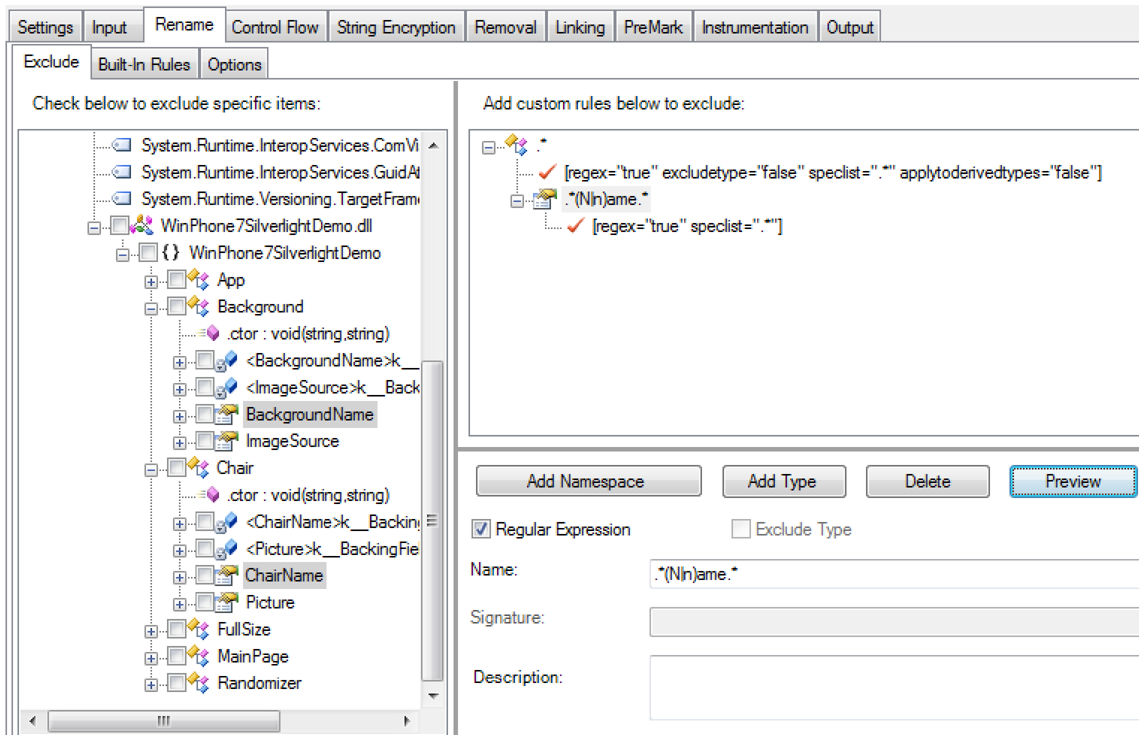


Those four properties are referenced from XAML in a way that Dotfuscator does not currently statically analyze, so if these properties are renamed, then the references from XAML will no longer work. Hence, we exclude them from renaming via the checkboxes.

Custom Exclusion Rules

There may be times when you may need to make many specific exclusions for a particular coding convention. And even then, when a developer adds more code that follows that convention, they would have to remember to make yet another specific exclusion for their new code.

The solution to this is the custom exclusion rules. Using the previous example, let's say we figured out that all properties with "Name" or "name" in its name will need to be excluded from renaming for some reason. Rather than creating a bunch of specific exclusions, we can make one custom rule:



The root node on the right is a Type node with the name ".*" with regex="true" to indicate that the ".*" should be treated as a regular expression. This will match all types, but we have excludetype="false", which means it is not our goal here to be excluding the type, just whatever comes beneath it (that we have not specified yet).

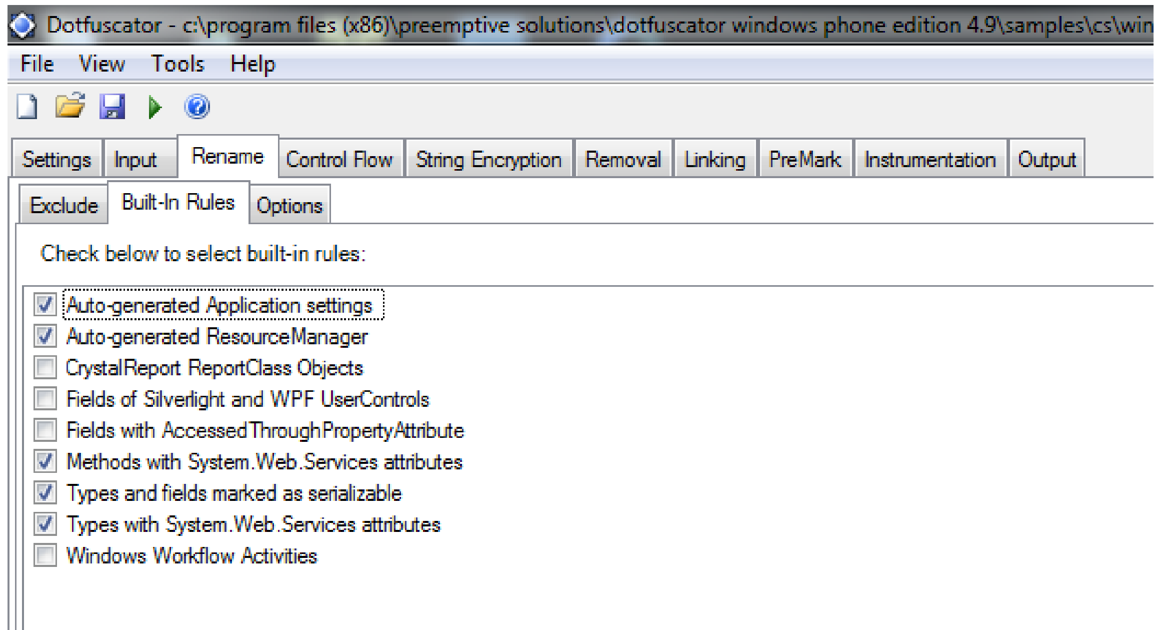
The child node is a Property node with the name ".*(N|n)ame.*", and again we've indicated that this should be treated as a regular expression. Once this is properly configured, you can click the "Preview" button and Dotfuscator will highlight the entries that match your custom rule in the treeview on the left. Here you see we have matched "BackgroundName" and "ChairName", which will both be excluded from renaming without needing to click their individual checkboxes.

The resulting configuration of this custom rule is stored as XML in the configuration file. This example would produce the following config xml:

```
<type name=".*" regex="true" excludetype="false">
     <propertymember name=".*(N|n)ame.*" regex="true" />
</type>
```

Built-in Rules

Built-in rules are custom exclude rules that are fairly universally useful, so we include them by default so users do not have to re-implement them.



The exact definitions of each of these can be found at

`%ProgramData%\PreEmptive Solutions\Common\dotfuscatorReferenceRule_v1.3.xml`

For example, “Fields of Silverlight and WPF UserControls” translates to the custom rule

```
<type name=".*" regex="true" excludetype="false">
     <field name=".*" regex="true" />
     <supertype name="System.Windows.Controls.UserControl" />
</type>
```

Options

There are many renaming options available on the Rename | Options tab, including what renaming scheme should be used, whether to introduce explicit overrides, and many others. They are well documented in the user manual.

Note: Overload Induction is typically not performed on assemblies containing any XAML code because of the way the runtime determines how XAML & Code are linked up. As a result, the “Use Enhanced Overload Induction” option will not change anything for such assemblies.

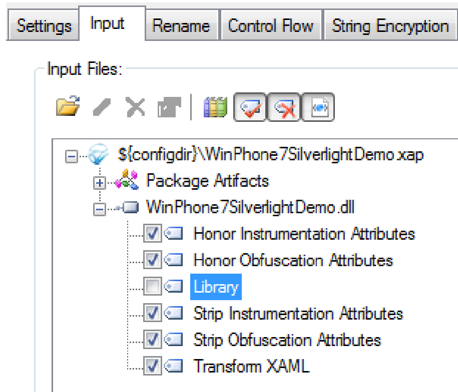
Library Mode

Library Mode is a per-assembly option that tells Dotfuscator not to rename or remove anything that could be accessed from outside that assembly.

- This includes things like public methods within private types. Since the type is private, the “public” could not be accessed from outside that assembly.

Library mode is *enabled by default for all input assemblies*. As a result, applications are likely to work after renaming. However, users may be concerned that so little of their application has been renamed. Turning Library Mode off will allow Dotfuscator to rename much more.

An exact specification of Library Mode rules can be found in the user guide.



Control Flow

Overview

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.

The goal of this feature is to prevent automatic decompilation of MSIL within methods back to high-level source. End users can still view the MSIL itself, but it is harder to fully comprehend and make large changes in functionality.

Example: Before & After on the sample included in the install at

`PreEmptive Solutions\Dotfuscator Windows Phone Edition 4.9\samples\cs\WinPhone7XNADemo`

```
private void HandleWallInteractions()
{
   bool hittingWallX = false;
   bool hittingWallY = false;
   if (this.floatingIconPosition.X < (this.floatingIcon.Width / 2))
   {
      this.floatingIconPosition.X = this.floatingIcon.Width / 2;
      this.yVelocity = (-1.0 * this.yVelocity) * 0.8;
      this.WallHitEffects("X");
      hittingWallX = true;
   }
   if (this.floatingIconPosition.Y < (this.floatingIcon.Height / 2))
   {
      this.floatingIconPosition.Y = this.floatingIcon.Height / 2;
      this.xVelocity = (-1.0 * this.xVelocity) * 0.8;
      this.WallHitEffects("Y");
      hittingWallY = true;
   }
   if (this.floatingIconPosition.X > (this.world.Height - (this.floatingIcon.Width / 2)))
   {
      this.floatingIconPosition.X = this.world.Height - (this.floatingIcon.Width / 2);
      this.yVelocity = (-1.0 * this.yVelocity) * 0.8;
      this.WallHitEffects("X");
      hittingWallX = true;
   }
   if (this.floatingIconPosition.Y > (this.world.Width - (this.floatingIcon.Height / 2)))
   {
      this.floatingIconPosition.Y = this.world.Width - (this.floatingIcon.Height / 2);
      this.xVelocity = (-1.0 * this.xVelocity) * 0.8;
      this.WallHitEffects("Y");
      hittingWallY = true;
   }
   UpdateWallHitTimers(hittingWallX, hittingWallY);
}
```

```
private void HandleWallInteractions()
{
   // This item is obfuscated and can not be translated.
}
```

 ©2012 PreEmptive Solutions, LLC

Limitations

Problems with correctness very rarely occur with Control Flow. It is important to test application *performance* after Control Flow obfuscation, particularly any code that is executed many times (such as a game loop, or computationally/algorithmically intensive method).

Control Flow obfuscation is only able to effectively defeat decompilers when the method contains sufficiently many basic blocks, among other requirements.

It is also not fully deterministic based on input IL, so two input methods with the same IL contents may result in different (but functionally equivalent (ignoring performance)) IL listings after obfuscation. As a result, if there was little code in those matching methods, one of them may result in IL that is decompilable, while the other may result in the "//This item is obfuscated and can not be translated." message you see above.

Exclusions

Control Flow obfuscation excludes work in the same way as Renaming Exclusions, with both specific exclusions and custom rules, but for Control Flow they are only applicable to methods. If users see performance degradation during testing, then these should be used to exclude any methods or classes where computationally intensive work is done (basically any situation in which the CPU is the limiting factor on performance).

Options

"High" is the only level designed to defeat automatic decompilers. "Medium" or "Low" may be used if "High" causes performance degradation that cannot be resolved using exclusions or for debugging purposes if Control Flow is believed to be the cause of a runtime error.

String Encryption

Overview

A common attacker method is to locate 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. Attackers 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.

Example: Before & After on the sample included in the install at

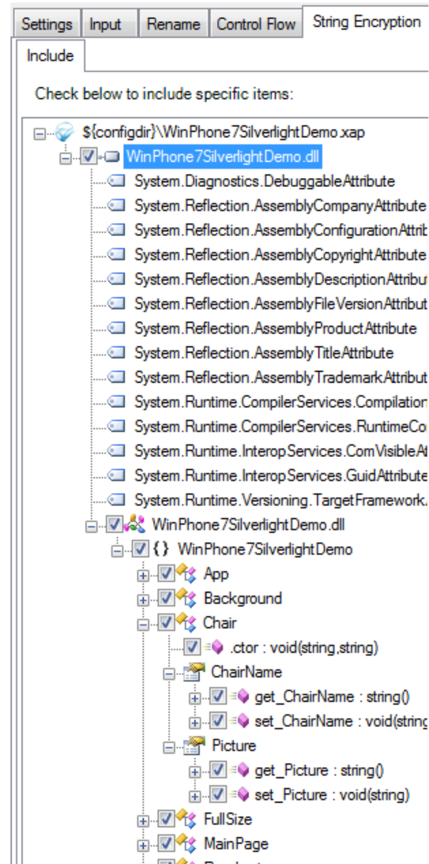
```
PreEmptive Solutions\Dotfuscator Windows Phone Edition
4.9\samples\cs\WinPhone7SilverlightDemo
```

```
private void PopulateData()
{
    List<Chair> list = new List<Chair>();
    list.Add(new Chair("Red Chair", "Images/Chairs/redchair.png"));
    list.Add(new Chair("Brown Chair", "Images/Chairs/brownchair.png"));
    list.Add(new Chair("Black Chair", "Images/Chairs/blackchair.png"));
    this.ChairList.set_ItemsSource(list);
    List<Background> list2 = new List<Background>();
    list2.Add(new Background("Blue", "Images/Backgrounds/blue.png"));
    list2.Add(new Background("Green", "Images/Backgrounds/green.png"));
    list2.Add(new Background("Orange", "Images/Backgrounds/orange.png"));
    list2.Add(new Background("Purple", "Images/Backgrounds/purple.png"));
    this.BackgroundList.set_ItemsSource(list2);
}
```

```
private void PopulateData()
{
    int num = 11;
    List<Chair> list = new List<Chair>();
    list.Add(new Chair(App.b("龔胤莚翫□蔯軜飦鵕", num), App.b("□箨蟟軡勫黖邰旼鯑矞匰诸褅剆筳撴扇眅湪栈戊缌൬愐紒熂", num)));
    list.Add(new Chair(App.b("▒韄竂棠蓛剆 畣騆鋘鳴蔬", num), App.b("□箨蟟軡勫黖邰旼鯑矞匰诸褅剆鷒觘嫸熇椆檤抴獝明筀洊擱磈縘", num)));
    list.Add(new Chair(App.b("▒鵸蟟譌胉剆 畣騆鋘鳴蔬", num), App.b("□箨蟟軡勫黖邰旼鯑矞匰诸褅剆鷒泋慞眅譛檤抴獝明筀洊擱磈縘", num)));
    this.ChairList.set_ItemsSource(list);
    List<Background> list2 = new List<Background>();
    list2.Add(new Background(App.b("▒鵸鍋寚", num), App.b("□箨蟟軡勫黖邰旹銔雭崺驉鬈鋼諾褧扇瘄：樠朊砌樀颵捌笔滜", num)));
    list2.Add(new Background(App.b("□韄苦寚蓛", num), App.b("□箨蟟軡勫黖邰旹銔雭崺驉鬈鋼諾褧扇瘄：濟磐枏樀線洊擱磈縘", num)));
    list2.Add(new Background(App.b("弔韄蟟醫犄袯", num), App.b("□箨蟟軡勫黖邰旹銔雭崺驉鬈鋼諾褧扇瘄：旽磐獝慏痰瘺瞜腂音篛", num)));
    list2.Add(new Background(App.b("뫄郤贉駒螗袯", num), App.b("□箨蟟軡勫黖邰旹銔雭崺驉鬈鋼諾褧扇瘄：礶鎰缌緞紐瘺瞜腂音篛", num)));
    this.BackgroundList.set_ItemsSource(list2);
}
```

Limitations

Typically the only problems with String Encryption are properly configuring the feature. See the "includes" section below.



Const strings are not encrypted, but all the places that use them are updated with encrypted versions of the string. To get around this, Enable Removal, and set Removal | Options | Removal Kind to "Remove only literals (const definitions)".

If some computationally intensive (highly repeated code) is generating strings (maybe for logging purposes), then there may be performance degradation as extra method calls must be performed to decrypt those strings.

Includes

Unlike Renaming and Control Flow which transform as much code as they can unless you specifically exclude items, *String Encryption by default encrypts no strings whatsoever*. To enable encryption of all strings in an assembly, simply go to the String Encryption | Include tab, and click the checkbox on the assembly node (which will recursively check all subitems).

©2012 PreEmptive Solutions, LLC

Removal

Overview

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, methods, and fields. This is of great benefit if application size is a concern, particularly if you are building your application from reusable components.

Limitations

The problems that can occur with Removal are very similar to those that may occur with Renaming. If Dotfuscator is unable to tell that certain methods are being called (due to reflection / XAML / etc.), then it may try to remove things that are required at runtime.

Includes

There are two types of inclusions possible, and both can be controlled by specific inclusions & custom rules. Consider an application in which method A() calls method B():

- Include Triggers: If you select a method as an Include Trigger, Dotfuscator will make sure to keep that method, as well as any descendants of that method in the call graph Dotfuscator sees (again skipping things like reflection). If an Include Trigger is set on method A(), then both A() and B() will be kept.
- Conditional Includes: Any method set as a Conditional Include is kept, but its call tree is not traversed for additional methods to keep. If a Conditional Include is set on method A(), then A() will be kept, but B() will be removed (as long as it is also not called by any other methods that Dotfuscator knows about).

Options

There are two options for “Removal Kind”

- “Remove unused metadata and code” – This is used when you want Dotfuscator to actively search out unused types/methods and remove them.
- “Remove only literals (const definitions)” – This is used to strengthen the power of String Encryption, so that unencrypted string consts get removed. If String Encryption is enabled, users almost always want Removal turned on with this option selected.

Watermarking (PreMark)

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.

Typically the only problems with PreMark are properly configuring the feature and using the command-line tool for extracting watermarks.

Linking

Dotfuscator supports linking multiple input assemblies into one or more output assemblies. Assembly linking can be used to make your application even smaller, especially when used with renaming and pruning, and to simplify deployment scenarios.

Limitations

We do not update the assembly name in Pack URIs in XAML. As such, linking of applications with any XAML is generally advised against.

XAML Considerations

XAML Rewriting

Dotfuscator parses & updates:

- Binding Markup Extensions, if the base property in the expression belongs to a class in the xml hierarchy above that node.
  - Ex: <Textbox Text="{Binding Path='Dealership.SalesTeam'}">
- Element names if we determine it is backed by a class we can find in the input (based on the namespace).
  - If we can find a match, then we also rewrite attributes/children nodes appropriately

XAML Rewriting Limitations

Dotfuscator does not handle:

- Actual XML namespaces
- We try to update things in templates/styles, but we may do so incorrectly. It is typically better to just exclude properties & types referenced from templates/styles.
- Attached properties/events
  - We do not update the backing field, methods, and registered name for the XAML references

Best Practices

Manually exclude:

- all attached properties & events.
- all backing fields & registered methods of attached properties & events.
  - Look for fields of type System.Windows.DependencyProperty
  - Look for methods that have any parameter of type System.Windows.DependencyObject
    - Unfortunately, a custom rule cannot be made to support this in the current version, so all instances must be found manually.
- all properties referenced from templates/styles (and potentially all other properties in the input with the same literal name).

XAML Build Warnings

In most situations when Dotfuscator analyzes a piece of XAML and cannot match it up with a CLR object, it will issue a warning to notify users they will need to make a manual change. It is important that warnings be checked when trying to determine the cause of runtime failures.

Smart Obfuscation Rules & Warnings

Smart Obfuscation is an ongoing effort to identify and apply obfuscation rules automatically for known API usage patterns and application types.

Smart obfuscation rules are a lot like Built-In renaming exclusion rules, but for situations that cannot be described by simple regular expressions. The simplest example of this is the exclusion of Enum members when Dotfuscator sees ToString() being called on one of them. If the user is requiring the Enum member to be ToString()'d, then they probably want it to have the original name. There is no possible way to describe this situation using the Custom Rule Exclusion mechanism.

Smart obfuscation produces two reports:

- A listing of the items excluded from renaming/removal due to Smart Obfuscation.
- A listing of items that Dotfuscator found but could not resolve itself that may indicate the need for manual exclusions.
  - There are many, many possible warnings. Here is just one example of such a warning:
    - FrameworkElementRule flagged something in Method Namespace.ContextMenu::void OnApplyTemplate() for the following reasons: Examine possible name arguments to System.Windows.FrameworkElement::FindName and manually exclude the referenced elements.
    - The user should simply follow the instructions - Find the ContextMenu class and look at the OnApplyTemplate() method. Find all places in that method that call FindName(string), and figure out if that string represents a Property from the input that needs to be manually excluded.

Property and Event Metadata

Properties (and events) consist of metadata defining them, as well as their actual backing methods. Dotfuscator is able to remove property metadata, backing methods, or both.

The Output tab will display what was done after each build. If the property node itself is grayed out, then the metadata was removed. If the method nodes underneath are grayed out then those backing methods were removed.

Map Files & Stack Trace Decoding

Dotfuscator provides the ability to decode obfuscated stack traces that result from errors in renamed applications. This process is explained in detail in the manual.

Exclusions/Inclusions in Code

The ***System.Reflection.ObfuscationAttribute*** class (Or the ***PreEmptive.ObfuscationAttributes.dll***) allows users to set exclusions in code rather than by using the GUI.

Troubleshooting Build Errors

- Are all 3rd party assemblies marked as artifacts?
- Are they using the latest version? Using the WP7 SKU?
- Can't find a reference assembly?
    - Run from command line with /v /e /bindlog=true for additional info about where Dotfuscator is looking. Add appropriate paths to the User Defined Assembly Load Path.
- In case of Dotfuscator errors, get a Dotfuscator stack trace by run from command line with options: /v /e

Troubleshooting Runtime Errors

- Are all 3rd party assemblies marked as artifacts?
- Are they using the latest version? Using the WP7 SKU?
- Is the application signed / resigned properly?
- Can you determine which transform is causing the problem?
    - Does it work when all obfuscation transforms are turned off?
    - Does it work with Library Mode enabled on all input assemblies?
    - Does it work with Transform XAML turned off on all input assemblies?
    - Runtime errors are almost always related to Renaming…
        - Dependency Properties – is the property and all backing methods excluded from renaming?
        - Does it work with the Property Exclusion Catch-All custom rule?

```
<type name=".*" regex="true">
        <propertymember name=".*" regex="true" />
</type>
```

- Are all of the warnings on the Warnings tab accounted for?
- Catch runtime exceptions and display them in a messagebox to see what is actually going wrong.

©2012 PreEmptive Solutions, LLC