Understanding .NET code obfuscation

Using Red Gate’s SmartAssembly for obfuscation
Introducing obfuscation

The .NET framework helps developers by managing memory usage, threads and exceptions for them, as well as providing other features that can help them write stable code faster. That’s great, but from an intellectual property point-of-view, it comes at a price.

Traditionally, code written by developers was compiled to a binary format that computers could execute. That binary format had to target a specific CPU architecture and bitness, which could complicate releases, and make it difficult for end-users to install your program. Crucially, however, once in a binary form, it was very difficult to reverse-engineer it to obtain the original source code.

In .NET, code is instead compiled to an Intermediate Language (IL) for distribution. When the executable or DLL containing IL is run by an end-user, the .NET Just-In-Time compiler (JIT) is activated, and compiles the IL to binary format when it is needed. The advantage is that you only have to distribute one version of the software; JIT will ensure that the resulting binary targets the architecture of the user’s computer. The disadvantage is that decompiling the distributed IL to obtain the source code is trivial.

Decompiling .NET code

To demonstrate how easy it is to decompile IL, let’s take a look at a real example. This example uses an application called QueryBee (a Red Gate tutorial application) but the principle is the same for any unobfuscated managed code.

1. Open `QueryBee.exe` with Red Gate’s .NET Reflector or with `ildasm.exe` to see the IL it contains:

   ```
   .method private hidebysig instance void m_ComboDatabase_TextChanged(object sender, class [mscorlib]System.EventArgs e) cil managed
   {
      .maxstack 8
      L_0000: nop
      L_0001: ldarg.0
      L_0007: ldarg.0
      L_0012: ldstr ""
      L_0017: call bool [mscorlib]System.String::op_Inequality(string, string)
      L_0021: nop
      L_0022: ret
   }
   ``

2. Use .NET Reflector to decompile the IL to C#, Visual Basic .NET, or any other managed language:

   ```
   private void m_ComboDatabase_TextChanged(object sender, EventArgs e)
   {
      this.m_connectButton.Enabled = this.m_ComboDatabase.Text != "";
   }
   ```

Note that the source code shown by .NET Reflector isn’t exactly the same as the original, but it is sufficiently similar that any developer can see how the code works.

The conclusion should be clear: if your program has any intellectual property that your company values (for example, algorithms that distinguish your product’s accuracy from its competitors), this ability is undesirable.

Put simply, if your company invests in .NET code, you need an obfuscator to protect it.

If you invest in .NET code, you need an obfuscator
How an obfuscator helps

Obfuscation is a process that makes your code harder to understand when the IL is decompiled. Obfuscation is not one technology; rather it is a series of steps which work together to make your code progressively harder to read.

No obfuscator can completely protect your code. After all, the IL still has to be ‘understood’ by the JIT, or else the program could not run. Obfuscation instead aims to make it more difficult for a human to understand how your software works.

How much obfuscation to employ depends on how valuable you consider your code. The more obfuscation you can add, the better protected your product. Some types of obfuscation may impact performance, however, and if this is becomes a major issue, you may wish to reduce the amount of obfuscation applied.

You may be asking whether obfuscation can stop your application from working. The answer (for any obfuscator) is yes. To try to avoid this, SmartAssembly includes a lot of complex logic to work out where it is safe to obfuscate code, and only applies obfuscation where appropriate. Because we can’t know exactly how your software is designed, however, there are inevitably cases in complex software where SmartAssembly makes the wrong decision.

For this reason, we recommend that you use SmartAssembly on every build as part of your Continuous Integration process, rather than just applying it immediately before release. This gives you time to investigate and solve any problems created by the obfuscation process.

Types of obfuscation available in SmartAssembly

Obfuscation is not a single task. There are many types, and you should apply as many as possible to your code, being aware of the performance trade-off presented by some options, and the risk of stopping your application from working.

The types of obfuscation available in Red Gate’s SmartAssembly are shown on the following page.

Setting up Control Flow Obfuscation in SmartAssembly. Most settings can be applied with just a couple of clicks.
<table>
<thead>
<tr>
<th>Obfuscation type</th>
<th>What it does</th>
<th>Works with</th>
<th>Can be applied to</th>
<th>Performance impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pruning</td>
<td>Removes non-essential metadata, such as the names of events, properties and method parameters.</td>
<td>All managed code</td>
<td>The main assembly, and any merged dependencies.</td>
<td>None</td>
</tr>
<tr>
<td>Type / Methods Name Mangling</td>
<td>Renames classes and methods either to meaningless ASCII strings, or Unicode unprintable characters.</td>
<td>All managed code</td>
<td>The main assembly, and any merged dependencies.</td>
<td>None</td>
</tr>
<tr>
<td>Fields Name Mangling</td>
<td>Renames fields either to meaningless ASCII strings, or Unicode unprintable characters. Optionally, fields can have the same name as another field in the same class.</td>
<td>All managed code</td>
<td>The main assembly, and any merged dependencies.</td>
<td>None</td>
</tr>
<tr>
<td>Control Flow Obfuscation</td>
<td>Makes the structure of your code more complex.</td>
<td>All managed code, except Compact Framework 2 or 3.</td>
<td>The main assembly, and any merged dependencies.</td>
<td>Some, depending on the level chosen.</td>
</tr>
<tr>
<td>References Dynamic Proxy</td>
<td>Adds a proxy at runtime, through which calls to external methods are made.</td>
<td>Applications built against .NET 2 or later, and PC applications created with XNA.</td>
<td>The main assembly, and any merged dependencies.</td>
<td>Mild</td>
</tr>
<tr>
<td>Resources Compression and Encryption</td>
<td>Compresses and encrypts resources.</td>
<td>Applications built against .NET 2 or later, and PC applications created with XNA.</td>
<td>The main assembly, and any merged dependencies.</td>
<td>Some when the application first runs (while the resources are decompressed), then none.</td>
</tr>
<tr>
<td>Strings Encoding</td>
<td>Encodes passwords, SQL queries, serial numbers and other strings.</td>
<td>Basic level: All managed code With improved protection: Applications built against .NET 2 or later, and PC applications created with XNA.</td>
<td>The main assembly, and any merged dependencies.</td>
<td>Strings are decoded when needed. If your application uses strings intensively, there may be a performance impact.</td>
</tr>
<tr>
<td>Prevent ildasm.exe from opening the assembly</td>
<td>Adds an attribute to the assembly to prevent the Microsoft IL Disassembler (ildasm.exe) opening it. Note that this option does not provide protection against other disassemblers.</td>
<td>Applications built against .NET 2 or later, and PC applications created with XNA.</td>
<td>The main assembly</td>
<td>None</td>
</tr>
</tbody>
</table>
The result of obfuscation

It is not always possible to demonstrate obfuscation in a short code sample. For example, the References Dynamic Proxy only exists at runtime, and so cannot easily be captured. Similarly, you would need to read the IL for a substantial part of the assembly to see the effect of Control Flow Obfuscation.

We can, however, use .NET Reflector to show you the result of applying some types of obfuscation to a given method:

```csharp
public void CreatePrimes(int maxPrime, CancellationToken ct)
{
    if (m_primes.Count > 0)
        throw new InvalidOperationException("CreatePrimes has already been called");
    Thread.CurrentThread.Name = "PrimeGeneration";
    IEnumerable<int> candidates = Enumerable.Range(2, maxPrime - 2).ToArray();
    int i = 0;

    // This is an implementation of the Euler sieve
    while (i < candidates.Count())
    {
        // The current element is always a prime
        int multiplier = ((int[])candidates)[i];
        ulong ulMultiplier = (ulong) multiplier;
    }
}
```

```csharp
public void #D75b(int maxPrime, CancellationToken ct)
{
    #a2 #a = new #A2
    #75b = maxPrime
    if (this.m75b.Count > 0)
        throw new InvalidOperationException("CreatePrimes has already been called");
    Thread.CurrentThread.Name = "PrimeGeneration";
    IEnumerable<int> source = Enumerable.Range(2, #a.#75b - 2).ToArray<>()
    int index = 0;
    while (index < source.Count<int>())
    {
        #22 #a = new #22
        #19c = #a,
    }
}
```

```csharp
public void int maxPrime, CancellationToken ct)
{
    PrimeTable. = new PrimeTable.
        = maxPrime
    if (this..Count > 0)
        throw new InvalidOperationException("0x124");
    Thread.CurrentThread.Name = "0x16:d"
    IEnumerable<int> source = Enumerable.Range(2, . - 2).ToArray<>()
    int index = 0;
    while (index < source.Count<int>())
    {
        PrimeTable. = new PrimeTable.
        = ,
    }
}
```

```csharp
public void int maxPrime, CancellationToken ct)
{
    // This item is obfuscated and cannot be translated.
}
```
Working with dependencies

We've now explored obfuscation with a simple stand-alone application. Most applications do not exist as entirely independent assemblies, however. They make calls to dependencies, such as DLLs. Naturally, if you obfuscate parts of the main assembly, calls to the dependencies may no longer work.

To avoid this potential problem, SmartAssembly provides three solutions:

- **Dependency Merging**
  
  SmartAssembly can merge dependencies into your main assembly, meaning that the code in the dependency becomes part of the main assembly. This is the preferred option, because you can apply obfuscation to the dependency code.

- **Dependency Embedding**
  
  Dependency Merging doesn't work with dependencies that employ code integrity protection. In this case, you can embed the dependency into the main assembly, meaning that the dependency is packed into the assembly, and is unpacked at runtime.

  Dependency Embedding simplifies deployment, but because the dependency code isn't changed, it isn't obfuscated. Calls to the dependency from the main assembly also aren't name mangled.

  You can use the References Dynamic Proxy to hide calls to embedded dependencies.

  Note that you cannot embed dependencies with Compact Framework 2 or 3, Windows Phone 7.x native applications, or Windows Phone 7.x and Xbox XNA applications.

- **Other Dependencies**
  
  If you have dependencies that you cannot embed or merge, the dependency is not obfuscated. Calls to the dependency from the main assembly also aren't obfuscated.

  You distribute the dependency with your software in the normal way.

---

Dependencies Merging

Merging integrates dependency code into the main assembly code. You can then obfuscate and prune the merged assembly code.

Some dependencies may encounter issues when merged, for example, a library that checks for its binary integrity refuses to load. In this case, exclude these dependencies from the merge, and embed them into the assembly instead. You should also embed third-party libraries, rather than merge them. See ‘Dependencies Embedding’.

Select the dependencies you want to merge into the resulting assembly:

- [ ] RedGate.Frontier.UserEvents
- [ ] Rescan dependencies

Merging dependencies in SmartAssembly
Why obfuscation can go wrong

SmartAssembly analyzes your code to ascertain whether or not it is safe to obfuscate code before obfuscation is applied.

We can use Strings Encoding as a simple example of this. Most text that is displayed in the user interface, such as static labels, is safe to encode because the strings are decoded at runtime, but what about text that is displayed dynamically?

Naturally, you would not want garbled strings to be displayed in the user interface. SmartAssembly avoids this problem by assuming that enum values that have ToString() called on them may also be displayed, and so these are never obfuscated. The result is that SmartAssembly will err on the side of caution. Note that in this case some enum values might not be obfuscated when they actually could be.

Now consider the following code, from a calculator application:

A simple method called AddNumb(), which adds two integers, is built in a DLL:

```csharp
public class MyClass
{
    public static int AddNumb(int i, int j)
    {
        int result = i+j;
        return result;
    }
}
```

In an executable, the AddNumb() method is called by reflection:

```csharp
public static int m_reflectionTest()
{
    // Create MyClass object
    MyClass myClassObj = new MyClass();
    // Get the Type information.
    Type myTypeObj = myClassObj.GetType();
    // Get Method Information.
    MethodInfo myMethodInfo = myTypeObj.GetMethod("AddNumb");
    object[] mParam = new object[] { 5, 10 };
    // Get and display the Invoke method.
    myMethodInfo.Invoke(null, mParam);
    return 0;
}
```

If you merge the dependency, apply method name obfuscation, the name of the method AddNumb() is changed. There is no way that SmartAssembly can ascertain that the string “AddNumb” in m_reflectionTest() is a call to the AddNumb() method in the DLL, and so the string is not changed. The result is that the code no longer works.

Reflection (and techniques that use reflection, such as serialization), is the main source of problems when obfuscating code. SmartAssembly allows you to exclude methods from obfuscation (either within the SmartAssembly user interface, or by adding an attribute to your code), which lets you work around problems of the type described above. You should be aware, however, that it may require some effort to identify where problems occur, especially if your application has a particularly complex architecture.

Using Automated Error Reporting to identify problems

SmartAssembly helps you resolve problems caused by obfuscation through its exception reporting mechanism. Enabling Automated Error Reporting in your assembly allows you to test your obfuscation settings.

If your program throws an unhandled exception, you receive a report within minutes. The report describes where the problem occurred, and you can jump straight to the relevant line in the Visual Studio source code. If you have Name Mangling enabled, the names are de-obfuscated automatically.

To demonstrate this, a call to m_reflectionTest() is added to QueryBee. The assembly is built in Visual Studio to test that it works correctly prior to adding obfuscation. It does.

Now, in SmartAssembly, merge the DLL and apply Name Mangling. When the assembly is rebuilt and run, it throws an exception:

You should now return to your application and enable Automated Error Reporting:

- I want errors reported in my application.
  - Select the template you want to use to report the errors:
    - Standard template provided by SmartAssembly:

When you run your program again, you can send yourself an error report:
When you view the report in SmartAssembly, you can see the full, de-obfuscated, stack trace:

![SmartAssembly Stack Trace Image]

You can jump straight to the relevant line in Visual Studio from the stack trace.

![Visual Studio Stack Trace Image]

This procedure greatly simplifies finding errors caused by adding obfuscation.

If you leave error reporting enabled after release, you receive reports about unhandled exceptions from all of your customers, too, helping you improve your software.
Conclusion
In this paper, we have demonstrated why you need an obfuscator and the different types of obfuscation provided by SmartAssembly.

We have also described some of the issues you should be aware of when applying obfuscation, including impacting your application’s performance and the risk of stopping your assembly from working. We have explained that you can use Automated Error Reporting in SmartAssembly to identify quickly the causes of such problems.

To download a free, 14-day trial of SmartAssembly, see:
http://www.red-gate.com/products/dotnet-development/smartassembly/