Review SmartAssembly von Red Gate

Motivation

In den letzten Ausgaben der arcAKTUELL drehten sich die Themen dieser Rubrik um Fragestellungen zu ArcObjects, Softwareentwicklung und Best Practices. Dieses Mal verlassen wir dieses Feld ein wenig und stellen uns der Herausforderung, wie man geistiges Eigentum schützen kann.

Gerade in der heutigen Zeit erlauben Disassembler – auch Decompiler genannt – einen schnellen Zugriff auf die Interna einer Anwendung. Beispiele dieser Softwaregattung auf der .NET-Plattform sind der weithin bekannte Reflector (seit einigen Monaten leider kommerziell) und seine kostenfreien Alternativen dotPeek von JetBrains und ILSpy aus der Open Source Community.

Teilweise kann mittels weniger Klicks aus einem Assembly der komplette Sourcecode gewonnen werden – in verschiedenen .NET-Sprachen (C#, VB.NET, F#, Boo …) mit zugehöriger Projektdatei. Aus Sicht des Autors und Rechteinhabers ist dies meist unerwünscht oder gar katastrophal – letzteres insbesondere dann, wenn komplexe Algorithmen gestohlen werden. Zukünftig wird noch eine ganz andere Problematik an Gewicht gewinnen: Auditings und Codereviews durch den Auftraggeber. Durch eine (vom Auftragnehmer ungewollte) Disassemblierung können die Interna der Zulieferung begutachtet werden – eine Aufgabe die auch wir schon wahrgenommen haben. Und welche Überraschungen wir da erleben konnten, kann sich jeder, der das Umfeld der Softwareentwicklung kennt, gut vorstellen.

Kurzum: Was auch immer der Grund einer nicht gewünschten Disassemblierung ist, wie lässt sie sich verhindern?

Dazu gibt es die Softwaregattung der Obfuscators (Verschleierer). Sie verändern den .NET-IL- (oder Java-) Zwischencode in der Art, dass ein Disassemblieren nicht möglich ist oder zumindest das Analysieren von disassemblierten Codes mit massivem Aufwand verbunden ist.

Review

SmartAssembly aus dem Hause Red Gate ist ein derartiger Obfuscator. Diese Software trat in der Vergangenheit immer mal wieder auch in anderen Kontexten ins Rampenlicht – beispielsweise für automatisches Error-Reporting.
Im Zusammenhang mit dem GWB-Influencer-Programm ergab sich für mich die Möglichkeit, diese Software umfassend zu testen.

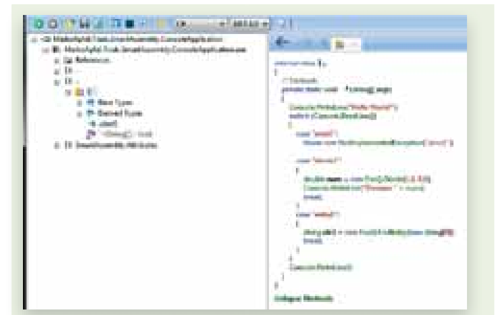
Erste Schritte

Nach der Installation und dem Start von SmartAssembly wird man von einer dieser neumodischen aufgabenorientierten Oberflächen empfangen und quasi dazu gezwungen, ein erstes Projekt zu starten. Dazu wählt man ganz einfach das Originalkompilat, welches transformiert werden soll, und im Anschluss, welche Transformation appliziert werden soll.
Die Vielfalt der möglichen Transformationen war für mich sehr überraschend. Ich hatte wahrlich nicht damit gerechnet, dass ein Werkzeug, welches als Obfuscator weitläufig bekannt ist, noch so viele weitere sinnvolle Funktionen anbietet.

Obfuscation

Die Codeverschleierung war der initiale Beweggrund, um SmartAssembly zu evaluieren. Zur Auswahl stehen zwei verschiedene Aufgaben:

Obfuscation



- Durch diese Maßname wird es dem Hacker nach einer Disassemblierung äußerst schwer gemacht, den Code einzusehen und zu verstehen. Dazu werden u. a. die Namen diverser Bezeichner durch Namen mit speziellen Sonderzeichen ersetzt, deren Darstellung für viele Editoren nicht möglich ist und die für den Anwender keinen Bezug zur umgesetzten Logik ergeben. Insbesondere die Standardeinstellung „I want to obfuscate using only Unicode unprintable characters" wird ihrem Namen gerecht.
- Nach der Verschleierung meines Democodes sind sowohl die Namen von Namespaces als auch von Methoden der Klassen völlig unverständlich und kryptisch oder werden als UTF-Codes dargestellt.

- Control Flow Obfuscation
  Ein derartig verschleierter Code soll den IL-Code derart verwürfeln und zu Spaghetticode machen, dass Decompiler ihren Dienst quittieren. Und tatsächlich gibt der Red-Gate-eigene Reflector nur noch folgende Anmerkung aus:
  // This item is obfuscated and cannot be translated.
  So weit, so gut. Aber was sagt die Decompiler-Konkurrenz? Die (bisher) freie Alternative dotPeek von JetBrains dekompiliert den Code sehr wohl und zwar so, wie er im Original war – zumindest in der Standardeinstellung „Fastest". In der Einstellung „Strongest" wird zumindest Code erzeugt, dem man durch viele Label- und sonstige Sprünge schwer folgen kann. Aber dekompilierbar war er dennoch.

Verschlüsselung

Eine weitere interessante Aufgabe ist die Verschlüsselung, um verschiedenartige Ressourcen effektiv zu schützen:

- Dependencies Embedding
  Mit dem Dependencies Embedding können referenzierte Assemblys als Ganzes verschlüsselt und als Ressource eingebettet werden. Ein spezieller statischer Assembly-Resolver-Mechanismus sorgt während der Laufzeit für ein Entschlüsseln der eingebetteten Assemblys und deren Rückgabe an die CLR. Um dieses Management zu bewerkstelligen, wird eine komplexe zusätzliche Logik in den IL-Code eingewebt.

Weiter auf der nächsten Seite

- Resources Compression and Encryption
  Damit werden Ressourcen verschlüsselt und komprimiert. Während der Laufzeit stellt die zusätzliche Logik sicher, dass die Entschlüsselung und Dekomprimierung nur einmal beim ersten Aufruf erfolgt und fortan gecacht ist.
- Strings Encoding
  Statt die Ressourceninformationen als Ganzes zu verschlüsseln, werden mit dieser Option nur die Strings verschlüsselt.

Durch die Verschlüsselung der String-Ressourcen mittels einer dieser Wege bricht ein guter Ansatzpunkt zum Reengineering der Programmlogik weg. Man hat keine Beziehung zwischen UI-Feedback und Code. Darüber hinaus werden auch sensible Informationen wie SQL-Querys, Lizenznummern, generische Passwörter etc. durch diese Verschlüsselungen geschützt.

Continuous Integration

Ein sehr wichtiger Aspekt in der Erzeugung einer auszuliefernden Software ist die Automatisierung der einzelnen Build-Schritte. Dazu zählen neben dem eigentlichen Kompilieren diverse qualitätssichernde Maßnahmen wie statische Codeanalysen und Unittests, die Erstellung des Setups und vieles mehr. Natürlich muss auch die Verschleierung als ein derartiger Schritt automatisiert umsetzbar sein – idealerweise durch Aufruf einer Konsolenanwendung ohne weiter benötigten Input eines Anwenders. Dazu gibt es in SmartAssembly ein eigenes Programm, das als Parameter nur die entsprechende Projektdatei benötigt. Ein kleiner Test mit Einbindung der Demoanwendung in den CI-Server TeamCity lief ohne Probleme sofort durch.

Build Steps

There are 2 build steps defined.

Build Step	Description
Build Project SmartAssembly	MSBuild Build file: .\SmartAssembly\src\MarkoApfel.Trials.SmartAssembly.sln Targets: default
Obfuscate	Command Line Command: .\SmartAssembly\tools\SmartAssembly\SmartAssembly.com .\SmartAssembly\src\MarkoApfel.Trials.SmartAssembly.saproj

Ausblick

Der ausführliche Testbericht findet sich in meinem Blog. Vor allem die zusätzlichen Aufgaben Optimierung, Dependencies Merging und Error bzw. Feature Usage Reporting sind einen Blick wert.

Insbesondere auch durch die CI-Integrierbarkeit deckt SmartAssembly das Aufgabenfeld der Code-Verschleierung ideal ab und bietet durch eine Vielzahl weiterer Programmoptionen einen echten Mehrwert. Obfuscation bildet eine gute Basis für den Schutz des geistigen Eigentums und sollte in jedem Softwareentwicklungsteam auf dem Radar sein. ++

Marko Apfel
Esri Deutschland GmbH
Kranzberg
m.apfel@esri.de
http://geekswithblogs.net/mapfel/archive/2011/11/01/147515.aspx