kurz & bündig	
Inhalt	Leistungsfähigkeit von Decompilern und Obfuskatoren und der Schutz geistigen Eigentums
Zusammenfassung	Wer seine Software vor neugierigen Blicken schützen möchte, nimmt an einem ausweglosen Wettlauf teil. Lesen Sie, mit welch einfachen Mitteln die Geschäftslogik Ihrer .NET-Applikationen offen gelegt werden kann.
Quellcode mit angeliefert	nein

Veröffentlichung: 04/2006.

Decompiler und Obfuscator

.NET-Assemblies sind für jedermann offen lesbar. Selbst Obfuskatoren können eine Software nicht wirksam verschlüsseln. Wie kann geistiges Eigentum dann vor dem Zugriff Dritter geschützt werden? Ist ein solcher Schutz überhaupt sinnvoll?

Manu Carus

Wer Software erstellt und die Kosten für das Design, die Entwicklung und die Qualität eines Produkts tragen muss, ist zu Recht an dem Schutz seiner Investionen interessiert. Jedoch: Wie können Ideen, Algorithmen und Executables vor neugierigen Blicken geschützt werden? Wie sicher ist ein solcher Schutz? Wer crackt den Schutz? Und mit welchen Motiven? Am Ende der Diskussion muss durchaus die Frage nach dem Sinn gestellt werden...

Der nachfolgende Artikel zeigt, wie eine im Release-Modus erstellte Assembly disassembliert und dekompiliert werden kann. Es wird dargestellt, wie leicht es ist, den ursprünglichen Source-Code zu rekonstruieren. Anschließend wird diskutiert, auf welche Weise Software-Hersteller sich vor der Offenlegung ihrer Produkte schützen, und dass es ein Leichtes ist, die angewandten Schutzmechanismen wieder zu brechen.

Als Fallbeispiel wird die Methode *Validate()* einer Klasse *IbanValidator* betrachtet, ein Algorithmus, der die Prüfziffer einer “International Bank Account Number” (IBAN) berechnet und verifiziert, ob es sich um eine grundsätzlich gültige Kontonummer handelt (Kasten 1).

Kasten 1: International Bank Account Number (IBAN)

Eine IBAN repräsentiert die internationale Darstellung einer Kontonummer und Bankleitzahl. Wie im elektronischen Zahlungsverkehr seit langem üblich, wird die IBAN um eine Prüfziffer ergänzt, die sich nach einem länderunabhängigen Algorithmus aus der Bankleitzahl und der Kontonummer berechnet. Durch eine Verifizierung dieser Prüfziffer können in den meisten Fällen Zahlendreher, Vertipper und Tastaturpreller im Zahlungsverkehr erkannt und verhindert werden.

Aufbau einer IBAN

Länge und Aufbau einer IBAN hängen von dem Herkunftsland der kontoführenden Bank ab. Deutsche IBANs beginnen mit “DE” und sind 22 Zeichen lang. Es folgen zwei Ziffern, die gemeinhin als Prüfziffern bezeichnet werden. Anschließend wird die Bankleitzahl und die Kontonummer angehangen.

Land	Länge	Struktur																				
		Land (ISO)		Prüf-ziffer		Nummer																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Deutsch-land	**22**	**D**	**E**	**p**	**p**	**BLZ**							**Kontonummer**										
Beispiel		D	E	6	0	7	0	0	5	1	7	5	5	0	0	0	0	0	0	7	2	2	9

Die IBAN “DE60700517550000007229” setzt sich beispielsweise aus den folgenden Informationen zusammen:

- Länderkennzeichnung “DE” nach ISO
- Prüfziffer “60”
- Bankleitzahl “70051755”
- Kontonummer “7229”

Kontonummern werden linksbündig durch führende Nullen auf 10 Ziffern aufgefüllt.

Validierung der Prüfziffer einer IBAN

Die IBAN wird inklusive der enthaltenen Prüfziffer einem vorgegebenen Prüfziffernberechnungsverfahren unterworfen. Dieses Verfahren ersetzt die alphanumerischen Zeichen des Ländercodes durch Ziffern und erreicht durch Verschiebung von Ziffernblöcken und Durchführung von Divisionen die Berechnung eines Restwertes. Entspricht dieser Rest dem Wert 1, ist die in der IBAN aufgeführte Prüfziffer korrekt und die IBAN somit gültig. Für eine weiterführende Darstellung sei der interessierte Leser auf [1] verwiesen.

ILDASM

Der erste Weg, Informationen über den Inhalt einer Assembly zu erhalten, führt über das Tool “IL Disassembler” (ILDASM), das gemeinsam mit der .NET-Framework ausgeliefert wird.

Eine .NET-Assembly ist eine Datei, die Programmcode und Metadaten in einer Dynamic Link Library (DLL) oder in einem Executable (EXE) zusammenfasst. Der kompilierte Code ist auf dem Zielrechner nicht direkt ausführbar, sondern besteht aus IL-Anweisungen, der sog. Intermediate Language (IL). Nachdem die Assembly zur Laufzeit in den Hauptspeicher geladen wurde, erfolgt eine Just-in-Time-Übersetzung der IL in ausführbaren, nativen Maschinencode.

Eine Assembly kann mit Hilfe des IL-Disassemblers “ILDASM” disassembliert werden. Dabei werden via Reflection die in der Assembly enthaltenen Metadaten ausgewertet und in einer Baumstruktur angezeigt (Abbildung 1). Listing 1 zeigt beispielhaft den disassemblierten IL-Code der Methode *IbanValidator.Validate()*.

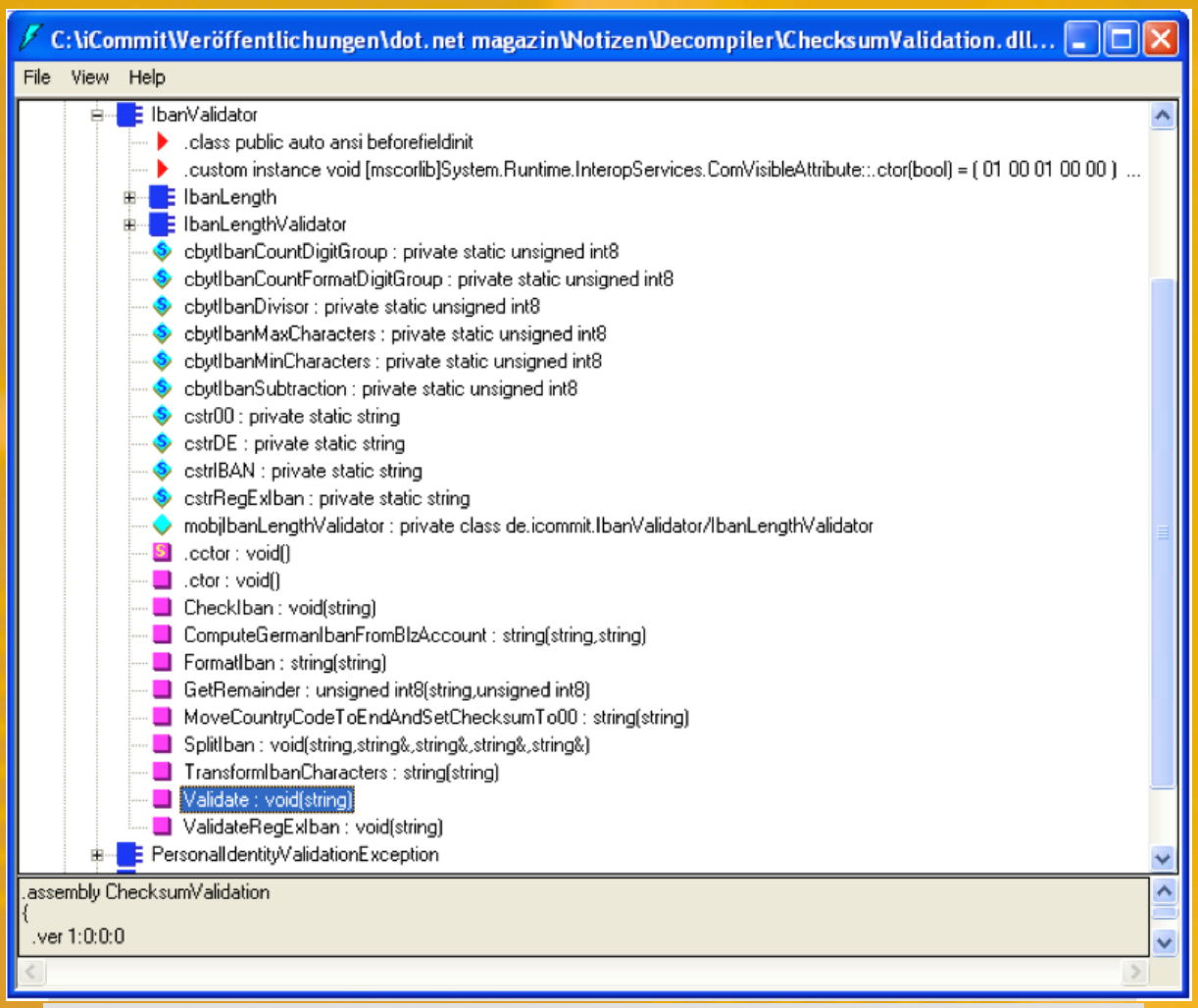

Abb. 1: Disassemblierung einer Assembly via ILDASM

Listing 1: Disassemblierter IL-Code

```
.method /*060000A5*/ public hidebysig instance void
        Validate(string strIban) cil managed
// SIG: 20 01 01 0E
{
  // Method begins at RVA 0x8884
  // Code size       192 (0xc0)
  .maxstack  6
  .locals /*11000027*/ init (string V_0,
           string V_1,
           string V_2,
           string V_3,
           string V_4,
           unsigned int8 V_5,
           unsigned int8 V_6,
           string V_7,
           string[] V_8)
  IL_0000:  /* 02   |                  */ ldarg.0
  IL_0001:  /* 03   |                  */ ldarg.1
  IL_0002:  /* 28   | (06)0000A8       */ call       instance void
                                                     de.icommit.IbanValidator
                                                     /* 02000006 */
                                                     ::CheckIban(string)
                                                     /* 060000A8 */

  IL_0007:  /* 02   |                  */ ldarg.0
  IL_0008:  /* 03   |                  */ ldarg.1
  IL_0009:  /* 28   | (06)0000A9       */ call       instance void
                                                     de.icommit.IbanValidator
```

```
                                                  /* 02000006 */
                                                  ::ValidateRegExIban(string)
                                                  /* 060000A9 */
IL_000e:  /* 02   |                  */ ldarg.0
IL_000f:  /* 03   |                  */ ldarg.1
IL_0010:  /* 12   | 00               */ ldloca.s   V_0
IL_0012:  /* 12   | 01               */ ldloca.s   V_1
IL_0014:  /* 12   | 02               */ ldloca.s   V_2
IL_0016:  /* 12   | 03               */ ldloca.s   V_3
IL_0018:  /* 28   | (06)0000AA       */ call       instance void
                                                  de.icommit.IbanValidator
                                                  /* 02000006 */
                                                  ::SplitIban(string,
                                                              string&,
                                                              string&,
                                                              string&,
                                                              string&)
                                                  /* 060000AA */
IL_001d:  /* 02   |                  */ ldarg.0
IL_001e:  /* 03   |                  */ ldarg.1
IL_001f:  /* 28   | (06)0000AB       */ call       instance string
                                                  de.icommit.IbanValidator
                                                  /* 02000006 */
                                                  ::MoveCountryCodeToEndAndSet
                                                    ChecksumTo00(string)
                                                  /* 060000AB */
IL_0024:  /* 13   | 04               */ stloc.s    V_4
IL_0026:  /* 02   |                  */ ldarg.0
IL_0027:  /* 11   | 04               */ ldloc.s    V_4
IL_0029:  /* 28   | (06)0000AC       */ call       instance string
                                                  de.icommit.IbanValidator
                                                  /* 02000006 */
                                                  ::TransformIbanCharacters
                                                    (string)
                                                  /* 060000AC */
IL_002e:  /* 13   | 04               */ stloc.s    V_4
IL_0030:  /* 02   |                  */ ldarg.0
IL_0031:  /* 11   | 04               */ ldloc.s    V_4
IL_0033:  /* 7E   | (04)000040       */ ldsfld     unsigned int8
                                                  de.icommit.IbanValidator
                                                  /* 02000006 */
                                                  ::cbytIbanDivisor
                                                  /* 04000040 */
IL_0038:  /* 28   | (06)0000AD       */ call       instance unsigned int8
                                                  de.icommit.IbanValidator
                                                  /* 02000006 */
                                                  ::GetRemainder(string,
                                                                 unsigned int8)
                                                  /* 060000AD */
IL_003d:  /* 13   | 05               */ stloc.s    V_5
IL_003f:  /* 7E   | (04)000041       */ ldsfld     unsigned int8
                                                  de.icommit.IbanValidator
                                                  /* 02000006 */
                                                  ::cbytIbanSubtraction
                                                  /* 04000041 */
IL_0044:  /* 11   | 05               */ ldloc.s    V_5
IL_0046:  /* 59   |                  */ sub
IL_0047:  /* D2   |                  */ conv.u1
IL_0048:  /* 13   | 06               */ stloc.s    V_6
IL_004a:  /* 11   | 06               */ ldloc.s    V_6
IL_004c:  /* 1F   | 0A               */ ldc.i4.s   10
IL_004e:  /* 2F   | 15               */ bge.s      IL_0065
IL_0050:  /* 72   | (70)001D0C       */ ldstr      "0" /* 70001D0C */
IL_0055:  /* 12   | 06               */ ldloca.s   V_6
IL_0057:  /* 28   | (0A)000015       */ call       instance string
                                                  [mscorlib/* 23000001 */]
                                                  System.Byte/* 01000014 */
                                                  ::ToString() /* 0A000015 */
IL_005c:  /* 28   | (0A)000031       */ call       string [mscorlib/* 23000001 */]
                                                  System.String/* 01000010 */
                                                  ::Concat(string,
                                                           string) /* 0A000031 */
IL_0061:  /* 13   | 07               */ stloc.s    V_7
IL_0063:  /* 2B   | 09               */ br.s       IL_006e
IL_0065:  /* 12   | 06               */ ldloca.s   V_6
IL_0067:  /* 28   | (0A)000015       */ call       instance string
```

```
                                                  [mscorlib/* 23000001 */]
                                                  System.Byte/* 01000014 */
                                                  ::ToString() /* 0A000015 */
  IL_006c:  /* 13   | 07              */ stloc.s   V_7
  IL_006e:  /* 11   | 07              */ ldloc.s   V_7
  IL_0070:  /* 07   |                 */ ldloc.1
  IL_0071:  /* 6F   | (0A)000022      */ callvirt  instance int32
                                                  [mscorlib/* 23000001 */]
                                                  System.String/* 01000010 */
                                                  ::CompareTo(string)
                                                  /* 0A000022 */
  IL_0076:  /* 2D   | 02              */ brtrue.s  IL_007a
  IL_0078:  /* 2B   | 00              */ br.s      IL_007a
  IL_007a:  /* 11   | 07              */ ldloc.s   V_7
  IL_007c:  /* 07   |                 */ ldloc.1
  IL_007d:  /* 6F   | (0A)000022      */ callvirt  instance int32
                                                  [mscorlib/* 23000001 */]
                                                  System.String/* 01000010 */
                                                  ::CompareTo(string)
                                                  /* 0A000022 */
  IL_0082:  /* 2C   | 3B              */ brfalse.s IL_00bf
  IL_0084:  /* 1B   |                 */ ldc.i4.5
  IL_0085:  /* 8D   | (01)000010      */ newarr    [mscorlib/* 23000001 */]
                                                  System.String/* 01000010 */
  IL_008a:  /* 13   | 08              */ stloc.s   V_8
  IL_008c:  /* 11   | 08              */ ldloc.s   V_8
  IL_008e:  /* 16   |                 */ ldc.i4.0
  IL_008f:  /* 72   | (70)000726      */ ldstr     "parity failure -> computed
                                                   checksum = '" /* 70000726 */
  IL_0094:  /* A2   |                 */ stelem.ref
  IL_0095:  /* 11   | 08              */ ldloc.s   V_8
  IL_0097:  /* 17   |                 */ ldc.i4.1
  IL_0098:  /* 11   | 07              */ ldloc.s   V_7
  IL_009a:  /* A2   |                 */ stelem.ref
  IL_009b:  /* 11   | 08              */ ldloc.s   V_8
  IL_009d:  /* 18   |                 */ ldc.i4.2
  IL_009e:  /* 72   | (70)0019F2      */ ldstr     "' <> '" /* 700019F2 */
  IL_00a3:  /* A2   |                 */ stelem.ref
  IL_00a4:  /* 11   | 08              */ ldloc.s   V_8
  IL_00a6:  /* 19   |                 */ ldc.i4.3
  IL_00a7:  /* 07   |                 */ ldloc.1
  IL_00a8:  /* A2   |                 */ stelem.ref
  IL_00a9:  /* 11   | 08              */ ldloc.s   V_8
  IL_00ab:  /* 1A   |                 */ ldc.i4.4
  IL_00ac:  /* 72   | (70)001A78      */ ldstr     "' = given checksum" /* 70001A78 */
  IL_00b1:  /* A2   |                 */ stelem.ref
  IL_00b2:  /* 11   | 08              */ ldloc.s   V_8
  IL_00b4:  /* 28   | (0A)000010      */ call      string [mscorlib/* 23000001 */]
                                                  System.String/* 01000010 */
                                                  ::Concat(string[])
                                                  /* 0A000010 */
  IL_00b9:  /* 73   | (06)0000A3      */ newobj    instance void de.icommit.
                                                  IbanValidationException
                                                  /* 02000005 */::.ctor(string)
                                                  /* 060000A3 */
  IL_00be:  /* 7A   |                 */ throw
  IL_00bf:  /* 2A   |                 */ ret
} // end of method IbanValidator::Validate
```

Anstatt des hexadezimalen Bytecodes bereitet ILDASM den IL-Code in einem lesbaren Format auf. Ohne die Einzelheiten der Intermediate Language genau kennen zu müssen, kann die grundsätzliche Arbeitsweise der Funktion an dem Listing abgelesen werden, da die Verarbeitung aus dem Aufruf weiterer Unterfunktionen besteht, die ihrerseits disassembliert werden können:

- IL_0002: *IbanValidator.CheckIban()*
  ⇨ überprüft, ob eine nicht-leere IBAN übergeben wurde
- IL_0009: *IbanValidator.ValidateRegExIban()*

⇨ überprüft, ob die übergebene IBAN den regulären Ausdruck `^[A-Z]{2}[0-9]{2}[0-9A-Z]{1,30}$` erfüllt

- IL_0018: *IbanValidator.SplitIban()*
  ⇨ unterteilt die übergebene IBAN in vier Substrings (Ländercode, Prüfziffer, BLZ, Kontonummer)
- IL_001f: *IbanValidator.MoveCountryCodeToEndAndSetChecksumTo00()*
  ⇨ verschiebt die Positionen der Zeichen innerhalb der übergebenen IBAN
- IL_0029: *IbanValidator.TransformIbanCharacters()*
  ⇨ ersetzt den Ländercode zeichenweise gemäß einer vorgegebenen Umsetzungstabelle
- IL_0033 - IL_0082: Berechnung der Prüfziffer und Vergleich mit der in der übergebenen IBAN enthaltenen Prüfziffer
- IL_0084 - IL_00bf: Fehlerbehandlung bei Nicht-Übereinstimmung der Prüfziffer

Um das eigentliche Prüfziffernberechnungsverfahren nachvollziehen zu können, ist sicherlich ein tieferer Einstieg in die IL-Syntax erforderlich. Jedoch kann die grundsätzliche Arbeitsweise der Methode mit Hilfe von ILDASM bereits "gelesen" werden.

Decompiler

IL ist eine idealisierte Programmiersprache, die auf eine relativ einfach strukturierte virtuelle Maschine zugeschnitten ist. Als solche ähnelt IL einem Assembler-Code, wie man ihn von Intel-x386-Architekturen her kennt, ist zugleich aber wesentlich näher an den Klassen und Methoden der .NET-Framework angelehnt und kennt die in C# und VB.NET verfügbaren objektorientierten Sprachkonstrukte (Vererbung, Polymorphie). IL hat einen wesentlich höheren Abstraktionsgrad als Maschinencode und verzichtet auf bloße Registeroperationen. Aus diesem Grund ist es viel einfacher, IL-Code in C# oder VB.NET zurück zu übersetzen. Genau diese Aufgabe erfüllen sog. Decompiler.

Einen exzellenten Decompiler bietet die Firma Remotesoft an, den sog. "Salamander .NET Decompiler" ([2]). Auf seiner Homepage stellt der Anbieter eine limitierte Demo-Version zur Verfügung, die das Dekompilieren einer Assembly auf max. 10 Methoden beschränkt ([3]). Listing 2 zeigt das Ergebnis einer Dekompilierung der IL-Methode *IbanValidator.Validate()* aus Listing 1.

Listing 2: Dekompilierter IL-Code der Methode *IbanValidator.Validate()*

```
public void Validate(string strIban)
{
  string str1;
  string str2;
  string str3;
  string str4;
  string str6;

  CheckIban(strIban);
  ValidateRegExIban(strIban);
  SplitIban(strIban, out str1, out str2, out str3, out str4);

  string str5 = MoveCountryCodeToEndAndSetChecksumTo00(strIban);
  str5 = TransformIbanCharacters(str5);
  byte b1 = GetRemainder(str5, cbytIbanDivisor);
  byte b2 = (byte)(cbytIbanSubtraction - b1);

  if (b2 < 10)
  {
    str6 = String.Concat("0", b2.ToString());
  }
  else
  {
    str6 = b2.ToString();
  }

  if (str6.CompareTo(str2) != 0)
  {
    throw new IbanValidationException(String.Concat(new string[]{"parity failure
              -> computed checksum = \'", str6, "\' <> \'", str2, "\' = given
                 checksum"}));
  }
  else
  {
    return;
  }
}
```

Selbstverständlich wird die gesamte Assembly dekompiliert, so dass auch der Source-Code der verwendeten Methoden *CheckIban()*, *ValidateRegExIban()*, *SplitIban()*, *MoveCountryCodeToEndAndSetChecksumTo00()*, *TransformIbanCharacters()* und *GetRemainder()* rekonstruiert wird. Wer es nicht bei einem Schreibtisch-Studium des Codes belassen will, kann nun ein Visual Studio-Projekt anlegen, die dekompilierten Sourcen importieren und schrittweise durch den Source-Code debuggen...

Ein Decompiler ist also in der Lage, das Ergebnis einer wochenlangen Projektarbeit in kürzester Zeit zu entzaubern!

Dabei ist es allerdings nicht möglich, die ursprünglichen Sourcen exakt zu rekonstruieren, “lediglich” die in der kompilierten Assembly enthaltenen Informationen können zurück übersetzt werden. In Listing 2 drückt sich dies durch die Verwendung generischer Bezeichner für lokale Variablen aus (*str1*, ..., *str6*, *b1*, *b2*). Weitere Abweichungen zwischen Dekompilat und Original-Source-Code sind (im Allgemeinen):

- Die Deklarationen von Methoden, Eigenschaften und Membervariablen werden in ihrer Reihenfolge vertauscht. Das ist einleuchtend, da der Decompiler den Reflection-Mechanismus auf die Assembly anwendet und abhängig ist von der Reihenfolge, in der Enumerationen wie *GetMethods()*, *GetProperties()* und *GetFields()* die Klassendefinitionen auflisten. Zudem entscheidet der Programmierer des Decompilers, in welcher Reihenfolge die benötigten Reflection-Methoden aufgerufen werden.

- Sprachspezifische Operatoren wie “+ ” (C#) oder “& ” (VB.NET) für eine String-Konkatenation werden bei der Kompilierung in die IL durch ihre Äquivalente ersetzt. Ein Dekompilat enthält in diesem Fall *String.Concat(str1,str2)* anstatt *str1 + str2* bzw. *str1 & str2*.
- Nicht alle Schlüsselwörter einer .NET-Programmiersprache haben ihre IL-Entsprechung. Beispielsweise wird das C# Schlüsselwort *foreach* bei der Kompilierung in einen Enumerator umgesetzt: Anstatt

```
foreach(Object objKey in mobjCreditCards.Keys)
{
  CreditCardInstitute objCreditCardInstitute = (CreditCardInstitute)objKey;
  ...
}
```

erstellt der Microsoft C# Compiler *csc.exe* den folgenden Code:

```
IEnumerator iEnumerator = mobjCreditCards.Keys.GetEnumerator();
try
{
  while (iEnumerator.MoveNext())
  {
      CreditCardInstitute creditCardInstitute =
                        (CreditCardInstitute)iEnumerator.Current;
      ...
  }
}
finally
{
  IDisposable iDisposable = iEnumerator as IDisposable;
  if (iDisposable != null)
  {
    iDisposable.Dispose();
  }
}
```

- Leerzeilen und Kommentare gehen bei der Kompilierung unwiderruflich verloren.
- Debug-Anweisungen wie *Trace.Assert()* werden in einem Release-Kompilat nicht berücksichtigt.

Listing 3 zeigt zum Vergleich mit dem dekompilierten Code den Original-Source-Code der Methode *IbanValidator.Validate()*.

Listing 3: Original-Source-Code der Methode *IbanValidator.Validate()*

```
public void Validate(String strIban)
{
  // validation
  CheckIban(strIban);
  ValidateRegExIban(strIban);

  // splitting
  String strCountryIsoCode2;
  String strGivenChecksum;
  String strBlz;
  String strAccount;

  SplitIban(strIban,
            out strCountryIsoCode2,out strGivenChecksum,out strBlz,out strAccount);

  // algorithm
  String strModifiedIban = MoveCountryCodeToEndAndSetChecksumTo00(strIban);
  strModifiedIban = TransformIbanCharacters(strModifiedIban);
  byte bytRemainder = GetRemainder(strModifiedIban,cbytIbanDivisor);

  // compute checksum
  byte bytComputedChecksum = (byte)(cbytIbanSubtraction - bytRemainder);

  String strComputedChecksum;

  if (bytComputedChecksum < 10) strComputedChecksum = "0" +
                                                       bytComputedChecksum.ToString();
  else  strComputedChecksum = bytComputedChecksum.ToString();

  // compare checksums
  if (strComputedChecksum.CompareTo(strGivenChecksum) == 0)
    Trace.WriteLine(" --> OK");
  else
    Trace.WriteLine(" --> ERROR");

  if (strComputedChecksum.CompareTo(strGivenChecksum) != 0)
    throw new IbanValidationException("parity failure -> computed checksum = '" +
          strComputedChecksum + "' <> '" + strGivenChecksum + "' = given checksum");
}
```

Ein toolbasierter Textvergleich, beispielsweise mit Hilfe des Tools *WinDiff*, ist zwischen den beiden Source-Dateien aus den oben genannten Gründen nicht möglich. Bei dem Vergleich der beiden Listings ist aber deutlich zu erkennen, dass das Dekompilat in Listing 2 dem ursprünglichen Source-Code gefährlich nahe kommt! In den Händen der Konkurrenz oder eines Crackers (Hacker mit kriminellen Absichten) kann ein Decompiler zu einem gefährlichen Werkzeug werden, da es den Source-Code einer Software entschlüsselt und alle darin enthaltenen "Geheimnisse" offen legt. Seien es

- spezielle Algorithmen, die als geistiges Eigentum begriffen werden, oder
- fest verdrahtete Benutzernamen für Datenbank-Verbindungen, oder einfach nur
- Schwachstellen in der Programmierung, die in Form eines Exploits genutzt werden können, um unberechtigten Dritten einen privilegierten Zugriff auf den Server, die Datenbank oder das Netzwerk zu verschaffen.

Der Hersteller "Remotesoft" möchte nach eigener Aussage mit dem Werkzeug "Salamander .NET Decompiler" demonstrieren, wie einfach es ist, den Source-Code einer kompilierten Assembly wiederherzustellen. Die eigentliche Produktpalette des Herstellers soll dem gegenteiligen Schutz von Software dienen:

- Der “Salamander Obfuscator” erschwert das Dekompilieren von .NET-Assemblies, indem String-Literale, Klassen-, Methoden- und Variablennamen ins nahezu Unverständliche verschlüsselt werden (Abbildung 2). Selbst der Kontrollfluss der Applikation wird mutiert, um Decompilern die Rekonstruktion lesbarer Schleifenkonstrukte zu erschweren. Unter [4] stellt der Hersteller eine Online-Demo des Werkzeugs bereit. Assemblies, die mit Hilfe dieser Online-Demo obfuskiert werden, dürfen nicht zur weiteren Nutzung verteilt werden. Eine Vollversion für bis zu fünf Entwickler-Lizenzen wird für $799 angeboten.

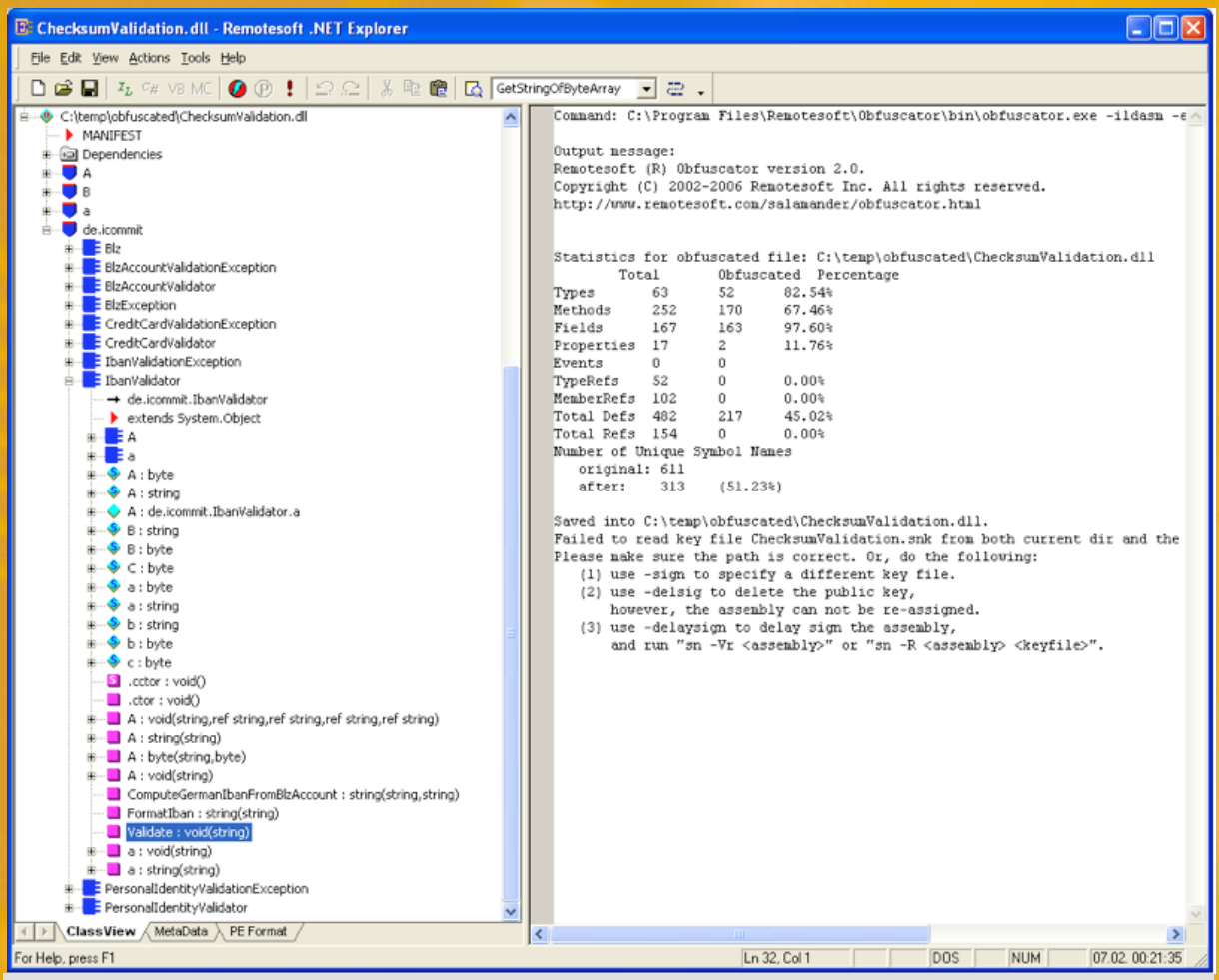

Abb. 2: Remotesoft .NET Obfuscator

- Der “Salamander Protector” ([5]) konvertiert den IL-Code einer Assembly in ein natives Format und erhält dabei alle .NET-Metadaten innerhalb der Assembly. Klassen-, Methoden- und andere Symbolnamen bleiben also erhalten. Dabei ergibt sich ein vergleichbarer Schutz wie bei nativ kompilierten C/C++ Applikationen. Es können Strings, Ressourcen und Code-Abschnitte verschlüsselt werden. Das Werkzeug setzt das Dekompilieren einer Assembly damit grundsätzlich außer Kraft. Die Vollversion kostet $1.899 für fünf Entwicklerlizenzen. Leider steht zum aktuellen Zeitpunkt keine Demoversion zur Verfügung.
  Ein besonders interessantes Feature des Protectors ist die Option, vor dem Verpacken der Assembly ein externes Verschlüsselungspasswort zu vergeben. Der Benutzer der Applikation muss dieses Passwort

unmittelbar nach dem Aufruf der Applikation eingeben, bevor das eigentliche Programm ausgeführt werden kann. Neben der Verschlüsselung des Codes kann dadurch ein interessanter Nebeneffekt erreicht werden: Wird für jeden Kunden ein individuelles Passwort eingesetzt, und sind die Kunden allesamt namentlich bekannt, kann jede Kopie der Applikation auf den ursprünglichen Erwerber zurückverfolgt werden. Im worst case lassen sich also auch Raubkopien zurückverfolgen (auch wenn dem ursprünglichen Erwerber keine Absicht zu unterstellen ist).

- Der “Remotesoft .NET Explorer” ([6]) integriert die Tool-Suite des Herstellers in einer ILDASM-ähnlichen Benutzeroberfläche. Ein Objekt-Browser ermöglicht das Navigieren durch die Inhalte einer Assembly. Kontextsensitive Menus unterstützen das Disassemblieren, Dekompilieren, Obfuskieren, Protektieren und Ausführen der betrachteten Assembly. (Abbildung 2).

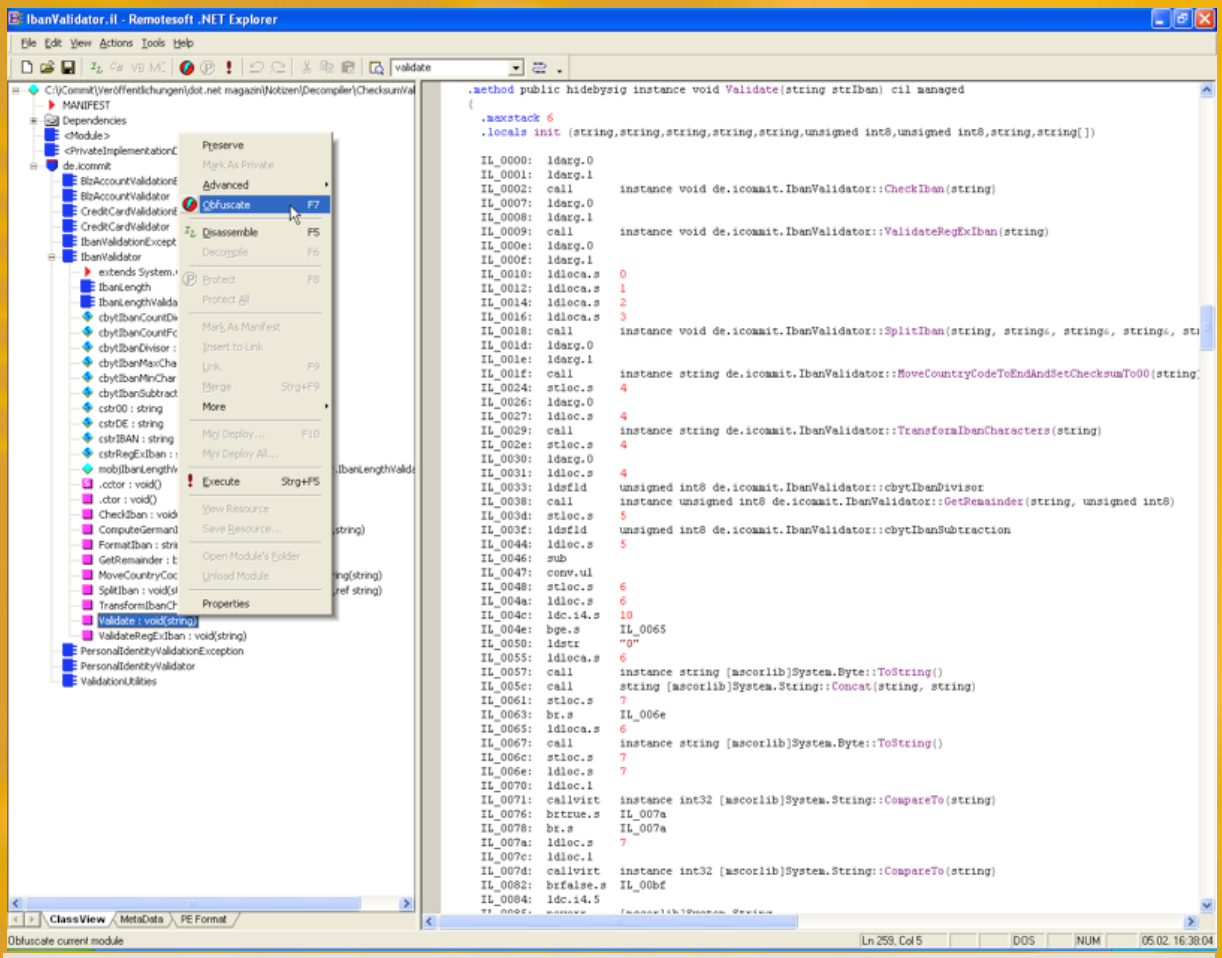

Abb. 3: Remotesoft .NET Explorer

Der Funktionsumfang des Decompilers verdeutlicht, wie umfassend der Source-Code einer Assembly rekonstruiert werden kann und wie ausgereift das Werkzeug wirklich ist:

- Erkennung aller .NET-sprachspezifischen Konstrukte, wie beispielsweise Attribute, Eigenschaften, Ereignisse, Felder, Methoden und verschachtelte Typen
- automatische Erkennung des verwendeten Compilers und Generierung von Source-Code in der entsprechenden Sprache, wie beispielsweise C#, Managed C++, VB.NET

- Unterstützung von Generics (.NET 2.0), unsafe code und Zeiger-Arithmetik
- Erstellung von Visual Studio.NET-Projektdateien zwecks einfacher Rekompilierung

Lt. Hersteller wurden über 10.000 Klassen, 800 Assemblies und 150.000 Dateien getestet.

Der “Salamander .NET Decompiler” ist damit quasi das Verkaufsargument des Herstellers für die eigentlich angebotenen Schutzwerkzeuge. Das Tool selbst schlägt mit $1.099 je Lizenz zu Buche.

Dies erstaunt zunächst.

Dem Autor ist es nicht gelungen, einen kommerziellen Markt für Decompiler zu identifizieren. Entwickler werden sich ihren Source-Code kaum versehentlich löschen. Die geschätzte Leserschaft sei aufgerufen, zur Aufklärung beizutragen!

Es bleibt festzustellen, dass ein Decompiler in den falschen Händen beträchtlichen Schaden anrichten kann. Software-Hersteller müssen keinen Decompiler erwerben, um zu dieser Erkenntnis zu gelangen. Einige Aufrufe der Online-Demo reichen aus, um zu sehen, wie verblüffend exakt der Source-Code wiederhergestellt werden kann. Vielleicht ist es beruhigend zu wissen, dass der Einstieg in ein solches Werkzeug über den hohen Preis gewisse Barrieren schafft. Dennoch gibt es immer Mittel und Wege...

Es geht auch gratis...

Einen brillianten Decompiler bietet Lutz Roeder auf seiner Site zum freien Download an ([7]). Der “Reflector for .NET” ist ein Klassenbrowser, der die Inhalte einer Assembly analysiert und die Abhängigkeiten zwischen Assemblies, Klassen, Methoden und Membervariablen aufzeigt (Abbildung 4).

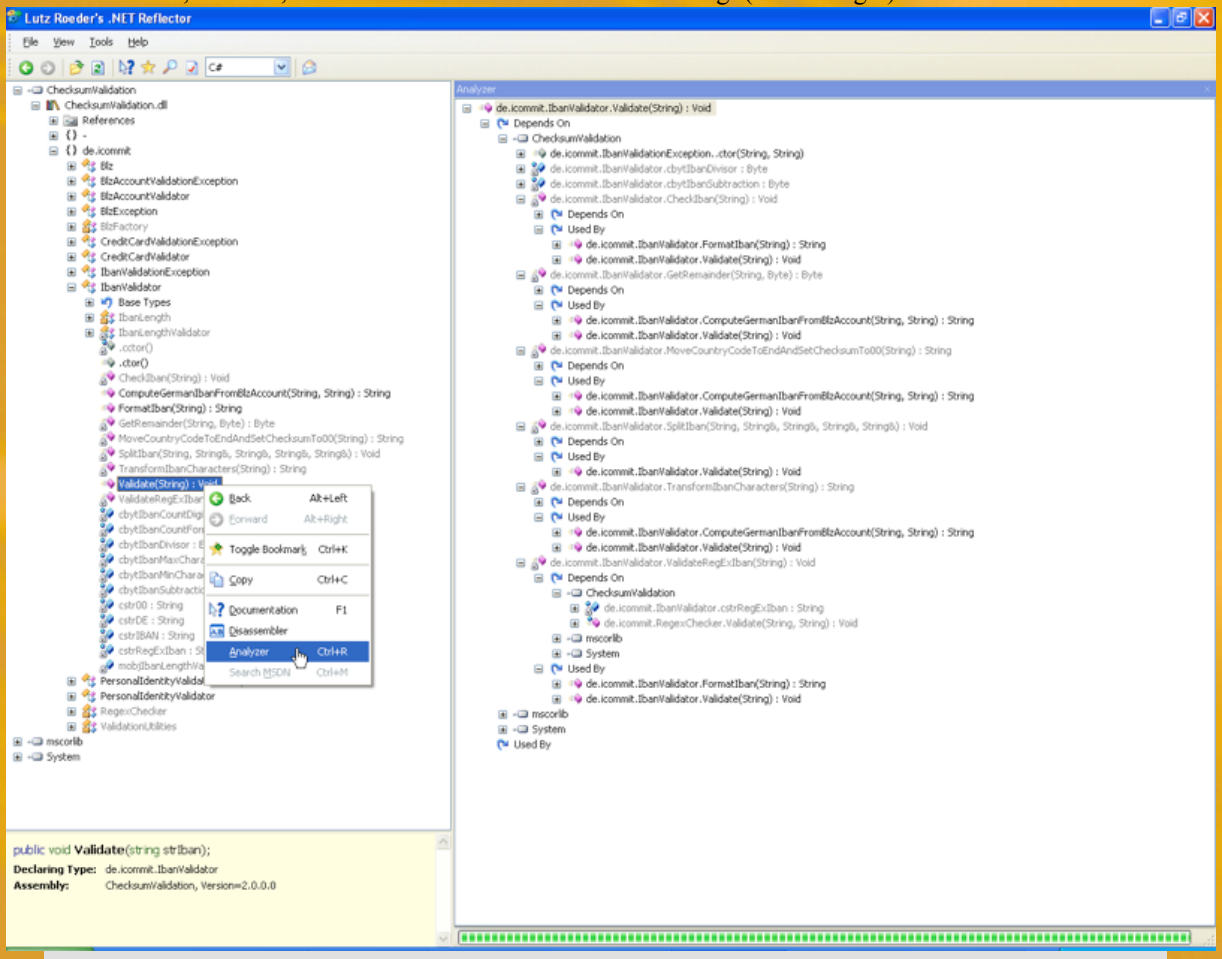

Abb. 4: Reflector for .NET

Dabei wird für jede Methode ein Graph der aufgerufenen (“call graph”) sowie der aufrufenden Methoden (“callee graph”) erzeugt. Der Benutzer kann auf diese Weise in dem TreeView erkennen, an welchen Stellen im Code die betrachtete Methode überall aufgerufen wird und innerhalb des Klassenbrowsers dorthin navigieren. Die analysierten Abhängigkeiten beschränken sich dabei nicht auf eine einzelne Assembly, vielmehr ist es möglich, sämtliche Assemblies einer Applikation inklusive aller Assemblies der .NET-Framework einzubeziehen!

Über den Eintrag “Disassembler” kann eine einzelne Klasse oder Methode disassembliert oder dekompiliert werden (Abbildung 5).

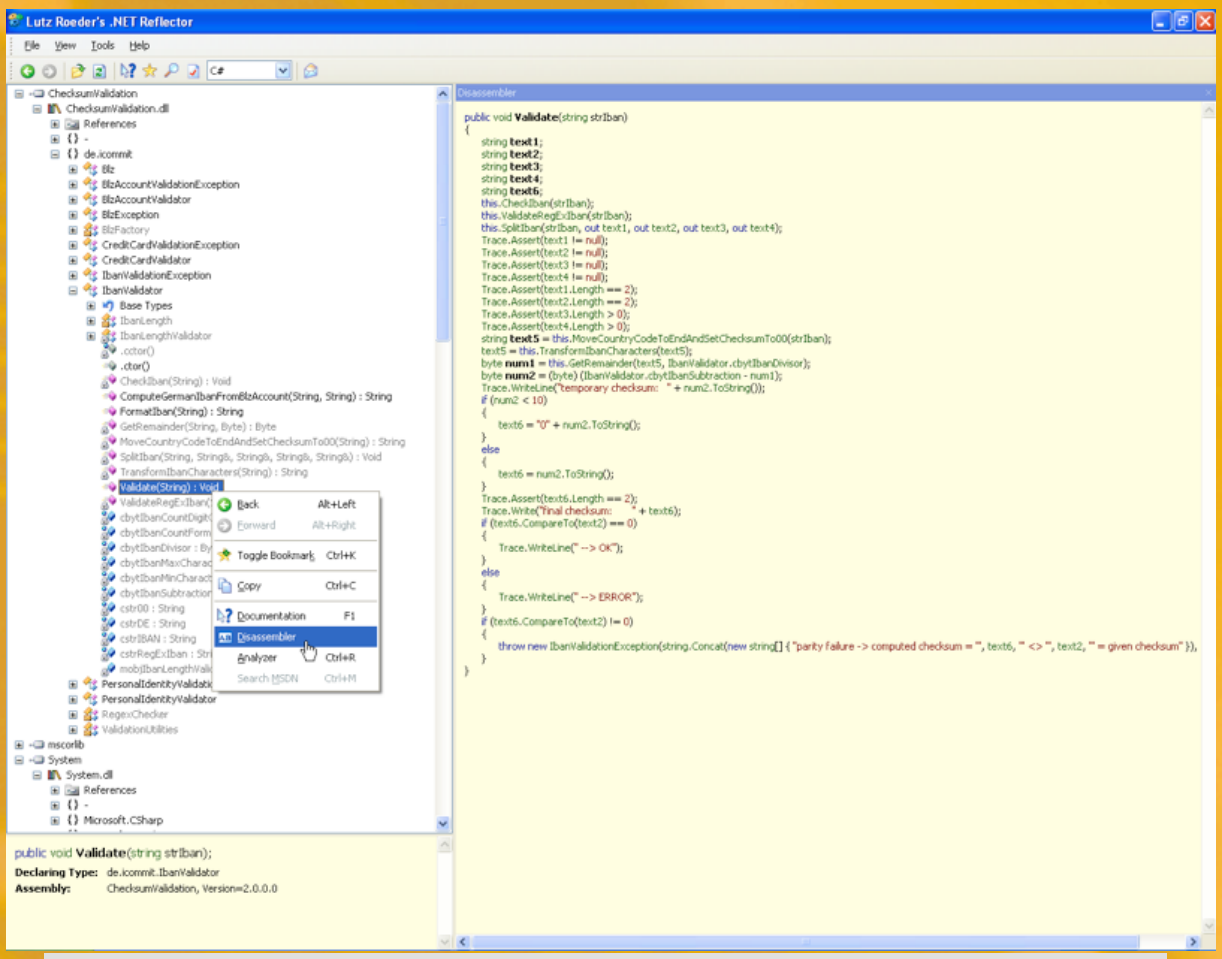

Abb. 5: Dekompilieren mit dem “Reflector for .NET”

Die Zielsprache wird über eine ComboBox in der Toolbar des Reflectors eingestellt. Durch Änderung dieses Eintrags kann der Code “on-the-fly” zwischen IL, C#, VB.NET und Delphi übersetzt werden. Per Klick auf eines der im Source-Code verwendeten Symbole navigiert der Klassenbrowser zu der entsprechenden Methode bzw. Typendefinition und zeigt deren Source-Code in der eingestellten Sprache an. Auf diese Weise wird ein bequemes Lesen und Verstehen der untersuchten Applikation ermöglicht.

Interessant ist die Arbeitsweise des Reflectors: Das Werkzeug setzt anstatt der .NET Reflection API ein eigenes Modell zum Auslesen der Metadaten, IL-Instruktionen, Resourcen und XML-Dokumentationen einer

Assembly ein. Daher können Assemblies, die mit der .NET-Framework 2.0 erstellt wurden, analysiert werden, ohne dass die .NET-Framework 2.0 selbst auf dem Rechner installiert sein muss.

Zusammenfassend lässt sich feststellen, dass der "Reflector for .NET" für eine schrittweise, interaktive Dekompilierung von Sourcen besonders geeignet ist, während der "Salamander .NET Decompiler" seine Stärke als Kommandozeilenwerkzeug zeigt und das Dekompilat einer gesamten Assembly in einem einzigen Schritt erstellt. Beide Decompiler erzeugen ähnlich gute Ergebnisse.

Obfuscation

Viele Anbieter beantworten das Problem der Dekompilation mit dem Begriff "Obfuscation" (Verschleierung). Dabei geht es darum, den kompilierten IL-Code nachträglich derart zu verändern, dass es sehr schwierig wird, die modifizierte Assembly zu lesen und zu verstehen. Das Ziel besteht nicht darin, den Prozess des "Reverse Engineering" grundsätzlich zu verhindern. Das ist auch gar nicht möglich. Vielmehr soll das Ergebnis dieses Prozesses weitgehend unbrauchbar gemacht werden.

Der erste Anhaltspunkt zum Verständnis einer Applikation liegt in dem Studium der darin verwendeten Zeichenketten. Beispiel:

```
.method private hidebysig specialname instance string get_ConnectionString()
        cil managed
{
  .maxstack 1
  .locals init (string)

  IL_0000:  ldarg.0
  IL_0001:  ldstr      "Data Source=MySqlServer;Initial Catalog=pubs;
                        User Id=sa;Password=abc123;"
  IL_0006:  stloc.0
  IL_0007:  br.s       IL_0009
  IL_0009:  ldloc.0
  IL_000a:  ret
}
```

Entsprechend fokussieren sich Obfuskatoren in erster Linie auf die Verschlüsselung aller Strings, die in der Assembly aufgeführt sind. Bei dem Vorgang der "String Encryption" werden alle String-Literale nach einem vorgegebenen Algorithmus verschlüsselt und erst zur Laufzeit wieder entschlüsselt. Aus dem String `"Data Source=MySqlServer;Initial Catalog=pubs;User Id=sa;Password=abc123;"` wird ein Byte-Array, beispielsweise `"C3 08 51 4F 5C E2 B8 CA 11 E7 EE D1 6D EA 36 F2 B3 BC 1E 87 D8 59 7B 8B 9A 81 C6 6A 26 D8 8D 53 0E 09 B2 46 DE 94 07 6E".` Ohne Kenntnisse des Verschlüsselungsalgorithmus ist es sicherlich nicht einfach, solche String zu entschlüsseln:

```
.method private hidebysig specialname instance void string get_ConnectionString()
        cil managed
{
      .maxstack 1
      .locals init (string)

      IL_0000:  ldarg.0
      IL_0001:  ldstr bytearray(C3 08 51 4F 5C E2 B8 CA 11 E7 EE D1 6D EA 36 F2 B3
                                BC 1E 87 D8 59 7B 8B 9A 81 C6 6A 26 D8 8D 53 0E 09
                                B2 46 DE 94 07 6E)
      IL_0006:  stloc.0
      IL_0007:  br.s       IL_0009
      IL_0009:  ldloc.0
      IL_000a:  ret
}
```

Das Problem dieser Art der Obfuskation ist allerdings inhärent! Zur Laufzeit muss die Applikation das Byte-Array wieder in den ursprünglichen String zurück übersetzen. Obfuskatoren injizieren zu diesem Zweck eine entsprechende Entschlüsselungsmethode in die Ziel-Assembly, beispielsweise

```
.method public hidebysig static string Decrypt(string inputStr) cil managed
{
  ...
}
```

Die Methode *get_ConnectionString()* besteht demnach in Wirklichkeit aus dem folgenden IL-Code:

```
.method private hidebysig specialname instance void string get_ConnectionString()
        cil managed
{
      .maxstack 1
      .locals init (string)

      IL_0000:  ldarg.0
      IL_0001:  ldstr bytearray(C3 08 51 4F 5C E2 B8 CA 11 E7 EE D1 6D EA 36 F2 B3
                                BC 1E 87 D8 59 7B 8B 9A 81 C6 6A 26 D8 8D 53 0E 09
                                B2 46 DE 94 07 6E)
      IL_0006:  call string Decrypt(string)
      IL_000b:  stloc.0
      IL_000c:  br.s       IL_000e
      IL_000e:  ldloc.0
      IL_0010:  ret
}
```

Der Vorgang der Obfuskation ist damit ad absurdum geführt! Der verschlüsselte String ist in der IL klar als Byte-Array lesbar. Um den gewünschten String zu erhalten, muss nun lediglich die IL-Methode *Decrypt()* aufgerufen werden. Bei der Eingabe des Byte-Arrays erhält man als Ausgabe dann die ursprüngliche Zeichenkette.

Eine detaillierte Abhandlung zu diesem Thema kann dem Artikel “The truth about string protection by obfuscators” entnommen werden ([8]).

Können Obfuskatoren die Software schützen?

String Encryption wird im allgemeinen überschätzt. Es wird nur ein geringer Schutz erreicht, und dies zu Lasten einer schlechteren Performance. Daher sollten Obfuskatoren höchstens nach dem Motto “besser ein geringer Schutz als gar keiner” eingesetzt werden. Der “Salamander .NET Decompiler” ist sogar in der Lage, String Encryption in einer Assembly zu erkennen, die injizierte *Decrypt()*-Methode zu finden, aufzurufen und an deren Stelle den entschlüsselten String in den Source-Code einzufügen!

Darüber hinaus nehmen Obfuskatoren Umbenennungen aller Symbolnamen vor. Allerdings können Verwirrungen bei der Benennung von Klassen, Methoden und Variablen mit ein wenig Zeit und Geduld durch Debugging des rekonstruierten Source-Codes aufgelöst werden.

Das Problem der derzeit am Markt angebotenen Obfuskatoren besteht darin, dass der Verschlüsselungsalgorithmus einschließlich des verwendeten Schlüssels in der “geschützten” Assembly ausgeliefert wird und damit jedem Hacking ausgeliefert ist. Ein verschlüsselter Inhalt kann nach dem heutigem Sicherheitsverständnis jedoch nur dann wirkungsvoll vor unberechtigten Dritten bewahrt werden, wenn der Schlüssel geheim bleibt, nur dem Empfänger des Inhalts bekannt ist und losgelöst von der eigentlichen Nachricht verwaltet wird.

Ein kurzer Ausflug in die PKI

Die beschriebene Verschüsselungsproblematik wird mit der Public/Private-Key-Infrastruktur (PKI) gelöst. Der Absender einer Nachricht verschlüsselt den Inhalt der Nachricht mit dem öffentlichen Schlüssel des Empfängers. Dieser “public key” ist, wie der Name beschreibt, der Öffentlichkeit frei zugänglich. Der Empfänger wiederum besitzt einen geheimen Schlüssel, den sog. “private key”, den nur er kennt. Beide Schlüssel, “public key” und “private key”, bilden ein Schlüsselpaar und sind so konstruiert, dass mit dem

öffentlichen Schlüssel verschlüsselte Nachrichten ausschließlich von dem zugehörigen geheimen Schlüssel entschlüsselt werden können. Will ein unberechtigter Dritter die verschlüsselte Nachricht “hacken”, muss er den verwendeten Schlüssel brechen. Aufgrund der mathematischen Natur der konstruierten Schlüsselpaare, der Verwendung von sehr langen Primzahlen, benötigt ein Hochleistungsrechner viele Jahre, um eine verschlüsselte Nachricht ohne Kenntnis des privaten Schlüssels zu entschlüsseln.

Dieser Sicherheitsstandard wird im heutigen Geschäftsverkehr beispielsweise zur Verschlüsselung der Kommunikation zwischen Clients und Servern eingesetzt (Secure Socket Layer, SSL): Auf dem Server wird ein Server-Zertifikat installiert. Die Clients haben freien Zugriff auf den darin enthaltenen öffentlichen Schlüssel des Servers und verschlüsseln damit ihre Nachricht (zum Beispiel einen geheimen Session-Key für die weitere Kommunikation zwischen den beiden Partnern). Aufgrund der Natur des Schlüsselpaars kann die verschlüsselte Nachricht ausschließlich über den geheimen Schlüssel des Servers entschlüsselt werden. Der Server entschlüsselt den vom Client übertragenen Session-Key und verschlüsselt fortan seine Nachrichten für diesen Client mit diesem Session-Key.

Schutz versus Aufwand

Ein wirklich standhafter Schutz von Software durch Obfuskation kann nur in Verbindung mit der Installation von Client-Zertifikaten erreicht werden. Hierzu muss der Software-Hersteller einen externen Schlüssel für den Prozess der Obfuskation verwenden: den öffentlichen Schlüssel des Clients. Die verschlüsselten Inhalte der ausgelieferten Assembly können dann ausschließlich über den geheimen Schlüssel des Kunden entschlüsselt werden.

Leider mangelt es bei der derzeitigen IT-Infrastruktur an der Verwendung von Client-Zertifikaten. In dem beschriebenen Obfuskationsszenario müßte entweder der Kunde vor dem Kauf der Software ein Zertifikat beschaffen und bereitstellen. Beides ist für den Kunden mit Kosten verbunden. Alternativ müßte der Software-Hersteller ein Client-Zertifikat erstellen, im Nachgang das Problem lösen, wie dieses Zertifikat sicher, und vor den Augen Dritter geschützt, den richtigen Adressaten erreicht. In jedem Fall kann die Software nicht einfach von einem Vervielfältigungsmedium (CD, Internet) installiert werden, da der Hersteller die Software vor ihrer Auslieferung kundenspezifisch obfuskieren muss.

Aufgrund des beschriebenen Aufwands ist verständlich, warum zu schützendes Know-how auf diese Weise fast ausschließlich in Hochsicherheitsszenarien bewahrt wird, der betriebene Aufwand also nicht massentauglich ist.

Diskussion

Letztlich bleibt die Frage, aus welchen Gründen die Hersteller ihre Produkte schützen und welchen Effekt sie dabei erzielen. Die nachfolgenden Argumente sind sicherlich subjektiv zu bewerten, daher sei der Leser herzlich zu einer kontroversen Diskussion eingeladen.

Welcher Obfuscator schützt am besten? Wie lange wird es dauern, bis ein Tool verfügbar ist, den obfuskierten Code wieder zu entschlüsseln? Welche Kosten sind mit der Beteiligung an einem solchen Wettlauf verbunden?

Am Ende ist der Kunde der Benachteiligte. Er allein trägt die Kosten und muss gleichzeitig die einhergehenden Nachteile in Kauf nehmen.

Vergleichbare Szenarien lassen sich in der Unterhaltungsbranche ausmachen. CDs und DVDs werden mit immer ausgeklügelteren Kopierschutzverfahren ausgeliefert. Dabei wird die Oberfläche einer Disc gezielt

beschädigt, um Kopierern die Arbeit zu erschweren. Das Ergebnis jedoch ist, dass es in der Regel nicht lange dauert, bis ein geeignetes Kopierwerkzeug in der Community auftaucht. Und der Käufer der CD ärgert sich darüber, dass er seine Lieblingsmusik nur zu Hause am PC und nicht im Auto hören kann, was ihn wiederum dazu ermutigen könnte, die nächste CD nicht käuflich zu erwerben...

Zurück zur Software-Branche: Warum wird Software geschützt? Warum wird Source-Code nicht einfach offengelegt?

Vertreter der Open-Source-Community argumentieren, dass Software gekauft wird, um Zeit und Kosten zu sparen, nicht um geistiges Eigentum zu rauben. Firmen kaufen Software und versichern, die erforderlichen Lizenzen zu erwerben. Durch die Offenlegung des Source-Code würde ersichtlich, was die Software tut. Bisher undurchsichtige APIs würden verständlich. Treten Fehler auf, sind Kunden in der Lage, die Fehlerursache zu untersuchen und zu fixen, bevor der Hersteller einen Patch oder ein Service Pack bereitstellt. Der Hersteller profitiert gleichzeitig von dem Umstand, dass viele Augen den Source-Code untersuchen und insgesamt zu einem besseren, stabileren Produkt beitragen.

Fazit

Möchte sich ein Software-Hersteller vor dem Zugriff Anderer auf seinen Source-Code schützen, muss er einen erheblichen Aufwand mit fragwürdigem Ergebnis betreiben. Es gibt grundsätzlich nichts, was in der .NET-Welt nicht dekompiliert werden kann (gleiches gilt übrigens auch für die Java-Welt). Wer die Lösung in nativem Maschinencode sieht, kann auf das Tool “Ngen.exe” der .NET-Framework zurückgreifen und sicher sein, dass ein Dekompilat kaum möglich ist, muss jedoch zugleich alle Nachteile in Kauf nehmen, die mit der Auslieferung von nativem Maschinencode einhergehen: Für jede potenzielle Zielplattform muss ein pre-JIT-Kompilat erstellt werden, während die mit .NET einhergehende Hardware-Unabhängigkeit verloren geht.

Obfuskatoren verschlüsseln den IL-Code einer Assembly bis zur Unlesbarkeit und erzeugen dabei einen nicht zu vernachlässigenden Performance-Impact, ohne die Software wirksam schützen zu können. Deobfuskatoren und Decompiler können den ursprünglichen Source-Code dennoch weitgehend rekonstruieren. Verwirrungen bei der Benennung von Klassen, Methoden und Variablen lassen sich durch Debugging des rekonstruierten Source-Codes auflösen.

Der Wettlauf zwischen Obfuskation (Verschlüsselung) und Deobfuskation (Entschlüsselung) kostet Zeit und Geld. Folglich gibt es nur wenige Anwendungsbereiche, in denen dieser Aufwand betrieben wird und in denen er durchaus gerechtfertigt ist. Nicht ohne Grund implementieren die betroffenen Hersteller ihre Software meist in C/C++.

Microsoft hat sich aus gutem Grund dazu entschieden, die ausgelieferten Assemblies der .NET-Framework nicht zu obfuskieren. Der Konzern Microsoft, dem der “Closed-Source-Gedanken” häufig nachgesagt wird, hat dazu beigetragen, dass Software-Hersteller die Funktionsweise der Framework und ihrer internen Struktur besser verstehen können. Nicht zuletzt ist daraus Mono entstanden, eine (Teil-)Portierung der .NET-Framework auf eine andere Plattform...

Manu Carus ist Certified Ethical Hacker (CEH) und auf die Sicherheit von Software, Hardware und Netzwerken spezialisiert. Sie erreichen ihn unter manu.carus@ethical-hacking.de.

Links & Literatur

[1] Prüfziffernberechnung für IBANs: http://www.pruefziffernberechnung.de/I/IBAN.shtml
[2] Remotesoft: http://www.remotesoft.com/
[3] Salamander .NET Decompiler: http://www.remotesoft.com/salamander/index.html
[4] Salamander .NET Obfuscator: http://www.remotesoft.com/salamander/obfuscator.html
[5] Salamander .NET Protector: http://www.remotesoft.com/salamander/protector.html
[6] Remotesoft .NET Explorer: http://www.remotesoft.com/dotexplorer/index.html
[7] Lutz Roeder's PROGRAMMING.NET: http://www.aisto.com/roeder/dotnet/
[8] String Encryption: http://www.remotesoft.com/salamander/stringencrypt/index.html