最新の逆コンパイラによる脅威とは

“難読化”なくしてソフトウェアは守れない!

～知的財産保護や情報セキュリティのために～

内容

ソースコードがないのに、ソースコードが公開されてしまう!?.............................. 2
難読化なくしてソースコードは守れない 3
　テストプログラムの作成 3
　ILDASM による逆アセンブル... 4
　逆コンパイル... 5
　Dotfuscator CE による難読化... 8
　製品版の Dotfuscator による難読化 10
　文字列の暗号化... 12
Silverlight アプリケーションなど最新技術への対応 14
難読化なくしてソフトウェアは守れない 17
開発プロジェクトへの責任 18

AG-TECH CORP.

ソースコードがないのに、ソースコードが公開されてしまう!?

“難読”という言葉は「読み方が難しい」という意味で辞書にも載っていますが、“難読化”はあまり聞き慣れた言葉ではありません。英語では obfuscate と言い、「意図的に分かりにくくする」という意味を持つ古くからある単語です[1]。これを一言で表す言葉が辞書にないことが、日本人が難読化を直観的に理解しにくいことをあらわしているかもしれません。

かつて、コンピューターウィルスが世間を騒がせ、大量の迷惑メールが送られてくるという事態がパソコンを使う人たちの大きな障害となっていました。しかし、今日、ウィルスや迷惑メールがニュースとして取り上げられることはほとんどありません。必ずしもウィルスや迷惑メールの脅威がなくなったわけではなく、むしろ、コンピューターウィルスは悪質化しており、迷惑メールの手口は巧妙化しています。これらが大きな話題にならなくなったのは、脅威がなくなったのではなく、脅威を取り除く技術の進歩と普及によるところが大きいと言えます。

もし、開発プロジェクトに対し、「実行ファイルには常にソースコードを添付すること」という条件が課せられたとしたら、「なんて非常識な！」と、とまどうプロジェクトリーダーや関係者は多いでしょう。オープンソース・プロジェクトならばともかく、実行ファイルの中にはユーザーの個人情報を処理するコード、ID やパスワードというセンシティブな情報を扱う部分、時間をかけて開発した高度な分析ロジックなどがあり、ソースコードが開示されれば保護すべきものが保護できなくなってしまうおそれがあります。企業のコンプライアンスや情報セキュリティが重視される昨今では、ソースコードを厳重に管理し、勝手に持ち出したり、逆に個人コンピューターの持ち込みを厳しく制限している企業も少なくありません。

しかし、.NET Framework で開発されたアプリケーションは、ソースコードがなくても、アプリケーション（実行ファイル）から元のソースコードへ、容易に逆コンパイルできてしまいます。.NET では、“アセンブリ”と呼ばれる構成単位が、相互作用できる情報（メタデータ）を保持しているため、これを使って内部を解析しやすくなっているのです[2]。

この文書では、バイナリレベルの実行ファイルからソースコードを生成する逆コンパイル技術や、Dotfuscator による難読化がどのように解析を防止するかについて説明しています。

---

[1] Dictionar.com による説明（http://dictionary.reference.com/browse/obfuscate）。
[2] Java でも同様です。逆に、C/C++言語用の逆コンパイルツールは存在しますが、それほど高精度ではありません。

難読化なくしてソースコードは守れない

実行ファイルがどのように解析されるかを示すため、実際にプログラムを作成します。

テストプログラムの作成

まず、簡単なプログラムを作成します。リスト 1 は、テスト用の簡単なコンソールアプリケーション（ScoreRank）で、順位付けする関数（CalcRank）と、それを使う Main 本体を定義しています。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ScoreRank
{
    class Program
    {
        const int MAXNUM = 30;                          // 最大人数
        const int MAXSCORE = 100;                       // 最大得点（0～MAXSCORE点）

        static int[] score = new int[MAXNUM];           // MAXNUM人数分の点数データ
        static int[] rank = new int[MAXNUM];            // MAXNUM人数分の順位データ
        static int[] count = new int[MAXSCORE + 1];     // n点の人の数
        static int[] ptsrank = new int[MAXSCORE + 1];// n点の人の順位

        static void CalcRank()
        {
            Array.Clear(count, 0, MAXNUM);          // 得点別配列をクリア
            for (int i = 0; i < MAXNUM; i++)        // 全員に対し得点別の配列をカウントアップ
                count[score[i]]++;
            int j = 0;                              // 最高点の人の順位（0基準）
            for (int i = MAXSCORE; i >= 0; i--)     // 最高点から順に順位づけ
            {
                ptsrank[i] = j;
                j += count[i];
            }
            for (int i = 0; i < MAXNUM; i++)
                rank[i] = ptsrank[score[i]];     // 得点別順位を全員に割り当てる
        }

        static void Main(string[] args)
        {
            Random r = new Random();
            for (int i = 0; i < MAXNUM; i++)
                score[i] = r.Next(MAXSCORE + 1);
            CalcRank();
            for (int i = 0; i < MAXNUM; i++)
                Console.WriteLine("#{0}: Score {1} ... Rank {2}", i, score[i], rank[i]);
        }
    }
}
```

リスト 1　テスト用のソースコード

このプログラムは、MAXNUM 人分の得点データ（score 配列）を元に、rank 配列に順位（0 基準）をつけるためのものです。並び替えを使わずに順位を求めるため、人数に比例した時間で順位を計算できる特徴があります。この、プログラムをコンパイルすると 5K バイト程度の実行ファイル（ScoreRank.exe）が作成されます。

ILDASM による逆アセンブル

Visual Studio には、ILDASM[3]というユーティリティが付属しており、.NET でコンパイルされたアセンブリ[4]を解析できます。この実行ファイルを解析すると、Main 関数は次のように表示されます。

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // コード サイズ       93 (0x5d)
  .maxstack  5
  .locals init ([0] class [mscorlib]System.Random r,
           [1] int32 i,
           [2] int32 V_2)
  IL_0000:  newobj     instance void [mscorlib]System.Random::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldc.i4.0
  IL_0007:  stloc.1
  IL_0008:  br.s       IL_001d
  IL_000a:  ldsfld     int32[] ScoreRank.Program::score
  IL_000f:  ldloc.1
  IL_0010:  ldloc.0
  IL_0011:  ldc.i4.s   101
  IL_0013:  callvirt   instance int32 [mscorlib]System.Random::Next(int32)
  IL_0018:  stelem.i4
  IL_0019:  ldloc.1
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.1
  IL_001d:  ldloc.1
  IL_001e:  ldc.i4.s   30
  IL_0020:  blt.s      IL_000a
  IL_0022:  call       void ScoreRank.Program::CalcRank()
  IL_0027:  ldc.i4.0
  IL_0028:  stloc.2
  IL_0029:  br.s       IL_0057
  IL_002b:  ldstr      "#{0}: Score {1} ... Rank {2}"
  IL_0030:  ldloc.2
  IL_0031:  box        [mscorlib]System.Int32
  IL_0036:  ldsfld     int32[] ScoreRank.Program::score
  IL_003b:  ldloc.2
  IL_003c:  ldelem.i4
  IL_003d:  box        [mscorlib]System.Int32
  IL_0042:  ldsfld     int32[] ScoreRank.Program::rank
  IL_0047:  ldloc.2
  IL_0048:  ldelem.i4
  IL_0049:  box        [mscorlib]System.Int32
  IL_004e:  call       void [mscorlib]System.Console::WriteLine(string,
```

[3] ILDASM＝Intermediate Language DisAssembler、中間言語用逆アセンブラ。
[4] 実行ファイルや DLL のこと。

```
                                                              object,
                                                              object,
                                                              object)
  IL_0053:  ldloc.2
  IL_0054:  ldc.i4.1
  IL_0055:  add
  IL_0056:  stloc.2
  IL_0057:  ldloc.2
  IL_0058:  ldc.i4.s   30
  IL_005a:  blt.s      IL_002b
  IL_005c:  ret
} // end of method Program::Main
```

リスト２　ScoreRank.exe の逆アセンブルリスト（Main 関数のみ）

どのように難読化しても、IL（中間言語）レベルの逆アセンブルを防御することはできません。逆アセンブラは、機械語としてのバイナリ[5]をアセンブリ言語に変換する単純作業を行うものであり、プログラムを実行する条件を満たしていればよいためです。逆アセンブラによる出力は読みやすいものではありませんが、難読化されていない場合は、どのような変数名を使って、どのような関数を呼び出しているかを追跡することができます。

逆コンパイル

次に、.NET Reflector という.NET 用の逆コンパイラツールを使います[6]。図１は、リスト１で生成された実行ファイルを逆コンパイルしている様子です。

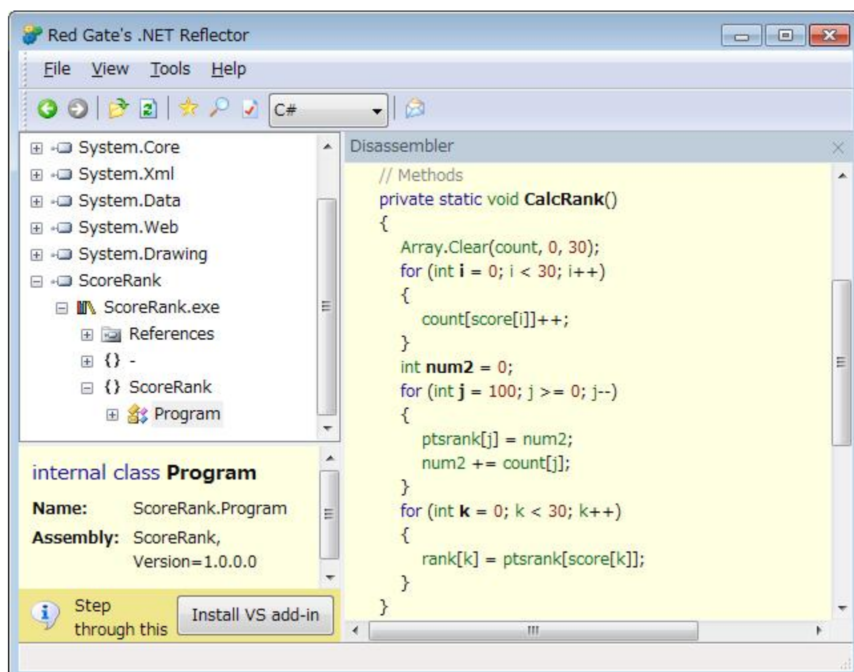

図 1　.NET Reflector による解析画面

---

5 .NET Framework の場合は、IL（Intermediate Language、中間言語）。
6 .NET Reflector（http://www.red-gate.com/products/reflector/）

逆コンパイラで生成された結果は、リスト 3 のようになります。このように、元のプログラムに近い結果が得られることがわかります。

```
internal class Program
{
    // Fields
    private static int[] count = new int[0x65];
    private const int MAXNUM = 30;
    private const int MAXSCORE = 100;
    private static int[] ptsrank = new int[0x65];
    private static int[] rank = new int[30];
    private static int[] score = new int[30];

    // Methods
    private static void CalcRank()
    {
        Array.Clear(count, 0, 30);
        for (int i = 0; i < 30; i++)
        {
            count[score[i]]++;
        }
        int num2 = 0;
        for (int j = 100; j >= 0; j--)
        {
            ptsrank[j] = num2;
            num2 += count[j];
        }
        for (int k = 0; k < 30; k++)
        {
            rank[k] = ptsrank[score[k]];
        }
    }

    private static void Main(string[] args)
    {
        Random random = new Random();
        for (int i = 0; i < 30; i++)
        {
            score[i] = random.Next(0x65);
        }
        CalcRank();
        for (int j = 0; j < 30; j++)
        {
            Console.WriteLine("#{0}: Score {1} ... Rank {2}", j, score[j], rank[j]);
        }
    }
}
```

リスト 3 .NET Reflector による逆コンパイルリスト

これは、デバッグ情報を持たない Release ビルド版ですが、デバッグ情報が残っていれば定数シンボルやローカル変数の名前を含めて、さらに正確な逆コンパイルリストが得られます。しかも、逆コンパイル時には言語を「C#」や「Visual Basic」など好きな言語を選択できます。リスト 9 は、言語として Visual Basic を指定し、Debug ビルド版を逆コンパイルした例です。C#から Visual Basic への言語コンバーターのように見えるかもしれませんが、この結果を生成するために元のソースコードは必要ないのです。

```vb
Friend Class Program
    ' Methods
    Private Shared Sub CalcRank()
        Dim i As Integer
        Array.Clear(Program.count, 0, 30)
        i
        For i = 0 To 30 - 1
            Program.count(Program.score(i)) += 1
        Next i
        Dim j As Integer = 0
        i = 100
        Do While (i >= 0)
            Program.ptsrank(i) = j
            j = (j + Program.count(i))
            i -= 1
        Loop
        i
        For i = 0 To 30 - 1
            Program.rank(i) = Program.ptsrank(Program.score(i))
        Next i
    End Sub

    Private Shared Sub Main(ByVal args As String())
        Dim i As Integer
        Dim r As New Random
        i
        For i = 0 To 30 - 1
            Program.score(i) = r.Next(&H65)
        Next i
        Program.CalcRank
        i
        For i = 0 To 30 - 1
            Console.WriteLine("#{0}: Score {1} ... Rank {2}", i, Program.score(i), Program.rank(i))
        Next i
    End Sub


    ' Fields
    Private Shared count As Integer() = New Integer(&H65  - 1) {}
    Private Const MAXNUM As Integer = 30
    Private Const MAXSCORE As Integer = 100
    Private Shared ptsrank As Integer() = New Integer(&H65  - 1) {}
    Private Shared rank As Integer() = New Integer(30  - 1) {}
    Private Shared score As Integer() = New Integer(30  - 1) {}
End Class
```

リスト 4　Debug 版の逆コンパイルリスト（Visual Basic を指定）

コンパイラは、選んだ言語の文法に沿って書かれたプログラムコードから、.NET 用のコード（IL、中間言語）を生成します。どの文法が、どのようなコードに置き換えられるかを分析し、生成されたコードから元のプログラムを推測するのが逆コンパイラの仕組みです。コード生成のパターン分析は着実に進歩しており、LINQ やラムダ式を使う最新版の.NET Framework にも対応しています。
また、.NET Framework が、プログラミングの柔軟性を高めるために、アセンブリ中にさまざまなメタデータを埋め込んでいることも精度の高い解析に役立っています。

C#や Visual Basic のような言語レベルに復元されてしまうと、プログラムの理解は容易になります。とくに“コメントがなくてもわかりやすいプログラム”を書いている場合は、ソースコードが公開されているのと似たような状態だと言えます。つまり、開発プロジェクトの中で、ソースコードの持ち出しを厳しく制限していても、知的財産の保護や情報セキュリティの面でリスクにさらされてしまうのです。.NET Reflector の単体版は無料です[7]。開発中のプロジェクトがあれば、手元の実行ファイルで逆コンパイルを試してみてもよいでしょう。おそらく想像以上に正確なソースコードが得られるはずです。

ここで挙げた例は単純なコンソールアプリケーションですが、Windows フォームのようなビジュアルな要素を持つ場合でも逆コンパイルされます。技術的には、Windows フォームはクラスであり、フォームを構成する部分は、クラスの一部として別ファイルになっているだけです。逆コンパイルした結果は、分割されず一つのクラスになりますが、InitializeComponent や Dispose など一部のメソッドとメンバー定義を抜き出すだけで、フォーム形式に戻すことができます。

Dotfuscator CE による難読化

.NET Framework が利便性のために実装している仕組みによって、逆コンパイルされやすくなること、そのことが知的財産やセキュリティ上の問題になることは当初から認識されていました。このため、.NET Framework が初めてリリースされた 2002 年の翌年から、Visual Studio のすべての製品版には Dotfuscator という難読化ツールが提供されてきたのです[8]。

解析の手掛かりになるのは、まずシンボル名です。リスト 3 では、Program というクラス名や CalcRank という関数名がそのまま表示されています。クラスやメソッドに意味のある名前を付けてプログラミングしている限り、それらがプログラムを理解する大きな手掛かりとなります。Visual Studio に付属の Dotfuscator はこうしたシンボルを意味のない短い名前に置き換えます。

Dotfuscator CE の使い方は簡単です。アセンブリ名のところに難読化したいアセンブリ（実行ファイルや DLL）を登録するだけです。

[7] Visual Studio に統合できる.NET Reflector Pro があります。
[8] 無料版など一部のエディションを除く。

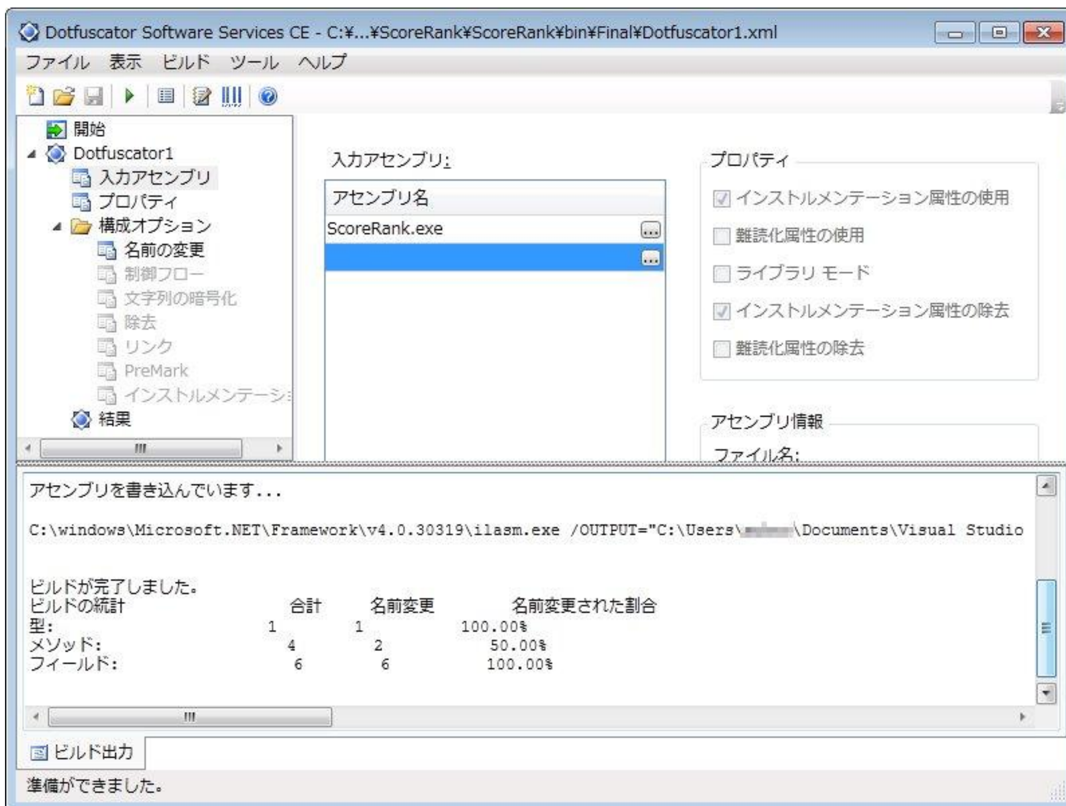

図 2　Dotfuscator CE を使っている様子

ビルドボタンを押すと、指定されたアセンブリが難読化されます。難読化されたアセンブリは、通常 Dotfuscated というフォルダに保存されます。こうして生成された実行ファイルを、ふたたび.NET Reflector で逆コンパイルすると、リスト 4 のような結果になります。

```
internal class a
{
    // Fields
    private const int a = 30;
    private const int b = 100;
    private static int[] c = new int[30];
    private static int[] d = new int[30];
    private static int[] e = new int[0x65];
    private static int[] f = new int[0x65];

    // Methods
    private static void a()
    {
        Array.Clear(e, 0, 30);
        for (int i = 0; i < 30; i++)
        {
            e[c[i]]++;
        }
        int num2 = 0;
        for (int j = 100; j >= 0; j--)
        {
            f[j] = num2;
            num2 += e[j];
```

```
        }
        for (int k = 0; k < 30; k++)
        {
            d[k] = f[c[k]];
        }
    }

    private static void a(string[] A_0)
    {
        Random random = new Random();
        for (int i = 0; i < 30; i++)
        {
            c[i] = random.Next(0x65);
        }
        a();
        for (int j = 0; j < 30; j++)
        {
            Console.WriteLine("#{0}: Score {1} ... Rank {2}", j, c[j], d[j]);
        }
    }
}
```

リスト 4　Dotfuscator CE で難読化された ScoreRank.exe の逆コンパイルリスト

ここに示す通り、外部シンボルは a、b のような短い名前に変更されるため、それぞれの関数の目的はわかりにくくなります。

製品版の Dotfuscator による難読化

Dotfuscator CE による難読化は、外部シンボルを分かりにくい短い名前に変更しますが、プログラムのロジックは隠せません。逆コンパイルでソースコードが得られれば、そこで.NET Framework の機能が呼び出されたり、「Login」や「パスワード」といった重要な文字列が見つかることで、センシティブな情報を扱う部分であることが突き止められてしまうおそれがあります。

製品版の Dotfuscator は、より実用的な難読化処理を施します。Dotfuscator CE に比べて大幅に機能は増えていますが、使い方はあまり変わりません。入力として難読化したいアセンブリを追加し、ビルドボタンを押します。図 3 は、前述のアプリケーション（ScoreRank.exe）を追加した様子です。

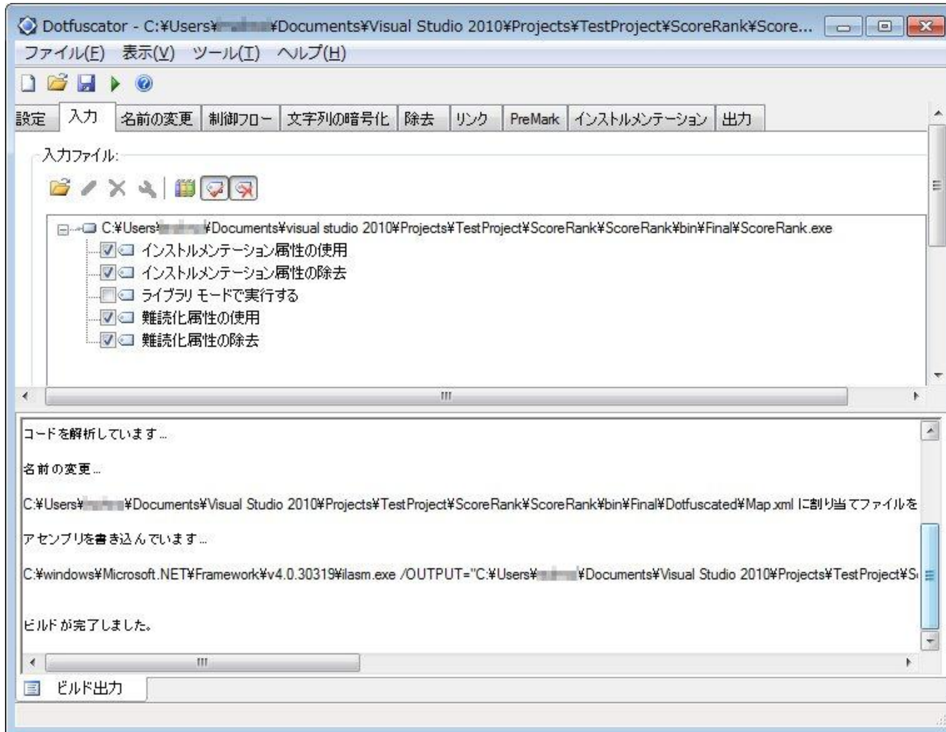

図 3　製品版 Dotfuscator を使っている様子

得られた実行ファイルを再び.NET Reflector で逆コンパイルすると、リスト 5 のようになります。

```
internal class a
{
    // Fields
    private static int[] c;
    private static int[] d;
    private static int[] e;
    private const int a = 30;
    private const int b = 100;
    private static int[] f;

    // Methods
    static a()
    {
        // This item is obfuscated and can not be translated.
    }

    private static void a(string[] A_0)
    {
        // This item is obfuscated and can not be translated.
    }

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

リスト 5　製品版の Dotfuscator で難読化された ScoreRank.exe の逆コンパイルリスト

ここでは、関数内部が逆コンパイルできず、「難読化されたため変換できない」という意味のコメントだけが表示されています。製品版の Dotfuscator は、生成された実行コードを論理的に同等な異なる制御フローに置き換えます。通常、コンパイラが生成しないコードになるため、このように逆コンパイルができなくなるか、できたとしても元の制御フローよりもわかりにくいものになります。これにより、ソースコードレベルでの理解は難しくなります。

文字列の暗号化

前述のとおり、逆コンパイルできない場合でも IL レベルの逆アセンブルを抑止することはできません。バイナリレベルで「Login」や「パスワード」といった文字列を検索される恐れがあります。
こうした解析を抑止するため、製品版 Dotfuscator には文字列を暗号化する仕組みがあります。この機能はデフォルトで無効にされているため[9]、「設定」画面で「文字列の暗号化を無効にする」を「いいえ」に変更し、「文字列の暗号化」画面で、対象となるモジュールを選択する必要があります。
このように設定して難読化した実行ファイルを前述の ILDASM で逆アセンブルすると、リスト 6 のようになります。

```
.method private hidebysig static void  a(string[] A_0) cil managed
{
  .entrypoint
  // コード サイズ       288 (0x120)
  .maxstack  6
  .locals init (class [mscorlib]System.Random V_0,
           int32 V_1,
           int32 V_2,
           int32 V_3,
           int32 V_4)
  IL_0000:  ldc.i4     0x7
  IL_0005:  stloc      V_4
  IL_0009:  br.s       IL_0034
  IL_000b:  ldloc      V_3
  IL_000f:  switch     (
                        IL_00e1,
                        IL_0107,
                        IL_011a,
                        IL_00f7,
                        IL_00ae,
                        IL_00d1,
                        IL_0097,
                        IL_0047)
  IL_0034:  newobj     instance void [mscorlib]System.Random::.ctor()
  IL_0039:  stloc.0
  IL_003a:  ldc.i4.0
  IL_003b:  stloc.1
  IL_003c:  ldc.i4     0x7
  IL_0041:  stloc      V_3
  IL_0045:  br.s       IL_000b
  IL_0047:  br         IL_00f9
  IL_004c:  ldstr      bytearray (4C 6E 4E 34 50 61 52 2E 54 6F 56 77 58 0A 5A 38   // LnN4PaR.ToVwX.Z8
```

[9] 文字列が暗号化されたアセンブリは、実行時に文字列復号のためのオーバーヘッドが発生します。暗号化が必要なモジュールだけを選択するようにしてください。

```
                                 5C 32 5E 2D 60 04 62 43 64 1E 66 56 68 14 6A 4B   // ¥2^-`.bCd.fVh.jK
                                 6C 43 6E 41 70 5F 72 53 74 27 76 16 78 17 7A 10   // lCnAp_rSt'v.x.z.
                                 7C 5D 7E 04 80 B3 82 FE )                         // |]~.....
  IL_0051:  ldloc      V_4
  IL_0055:  call       string a$PST06000001(string,
                                            int32)
  ...
} // end of method a::a
```

リスト６　文字列の暗号化を指定して難読化された ScoreRank.exe の逆アセンブルリスト

ロジックが入れ替わっているため分かりにくくなっていますが、リスト２の「IL_002b」で示された場所にある「ldstr　　"#{0}: Score {1} ... Rank {2}"」という文が、リスト 6 の「IL_004c: ldstr bytearray (4C 6E 4E ...)」から「IL_0055」までに相当します。ここでは、元の文字列（"#{0}: Score {1} ... Rank {2}"）をそのまま使うのではなく、暗号化したバイナリデータを置いておき、呼び出しの際に復号して元の文字列を返しているのです。

このような文字列の暗号化とロジックの組み替えにより、アプリケーションの解析は極めて困難なものとなります。

最新技術への対応

Dotfuscator 製品版では、アセンブリの再署名対応、改ざんの検出など、.NET Framework の最新技術に対応した難読化を提供しています。ここでは、注目度の高い Silverlight アプリケーションと ClickOnce についてご紹介します。

Silverlight アプリケーションの難読化

Dotfuscator 製品版では、Silverlight アプリケーションも難読化できます。リスト 7 に Silverlight アプリケーションのための簡単な XAML 定義の例を示します。

```
<UserControl x:Class="SimpleLogin.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    Width="300" Height="120">

    <Grid x:Name="LoginGrid" Background="White">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <sdk:Label Margin="4">Login...</sdk:Label>
        <sdk:Label Margin="4" Grid.Row="1" HorizontalAlignment="Right">ID</sdk:Label>
        <TextBox x:Name="tbID" Grid.Row="1" Grid.Column="1"></TextBox>
        <sdk:Label Margin="4" Grid.Row="2" HorizontalAlignment="Right">Password</sdk:Label>
        <PasswordBox x:Name="tbPassword" Grid.Row="2" Grid.Column="1"></PasswordBox>
        <Button Grid.Row="3" Grid.ColumnSpan="2" VerticalAlignment="Center"
                HorizontalAlignment="Center" Padding="20,4,20,4"
                Click="Button_Click">OK</Button>
    </Grid>
</UserControl>
```

リスト 7　Silverlight アプリケーションのための XAML 定義の例

Silverlight アプリケーション（.xap）の実体は、.NET ベースのアセンブリとマニフェストが ZIP 形式で圧縮されたファイルです。拡張子を.xap から.zip に変更することで、アセンブリ（.dll）やマニフェストファイル（AppManifest.xaml）などが展開でき、dll にはプログラムロジック（IL コード）に加えて、リスト 7 のような XAML 定義がリソースとして取り込まれています。

Dotfuscator は、XAP ファイルを直接入力ソースとして設定可能です。XAP ファイルを入力ソースにした場合、XAP ファイル内のアセンブリは自動的に認識され、名称の変更、制御フローの難読化、文字列の暗号化といった難読化処理を行う事ができます。

また、XAML 側の定義も自動的に難読化されたアセンブリと一致した名称に加工されます。リスト８は、上記の XAML を Dotfuscator で難読化した結果です。 コントロールの名称や、呼び出しメソッド名が、難読化されたアセンブリに対応するように自動的に変更されている事が確認できます。

```
<UserControl x:Class="SimpleLogin.MainPage" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" Width="300" Height="120"><Grid x:Name="a" Background="White"><Grid.ColumnDefinitions><ColumnDefinition Width="100" /><ColumnDefinition Width="*" /></Grid.ColumnDefinitions><Grid.RowDefinitions><RowDefinition Height="Auto" /><RowDefinition Height="Auto" /><RowDefinition Height="Auto" /><RowDefinition Height="*" /></Grid.RowDefinitions><sdk:Label Margin="4">Login...</sdk:Label><sdk:Label Margin="4" Grid.Row="1" HorizontalAlignment="Right">ID</sdk:Label><TextBox x:Name="b" Grid.Row="1" Grid.Column="1" /><sdk:Label Margin="4" Grid.Row="2" HorizontalAlignment="Right">Password</sdk:Label><PasswordBox x:Name="c" Grid.Row="2" Grid.Column="1" /><Button Grid.Row="3" Grid.ColumnSpan="2" VerticalAlignment="Center" HorizontalAlignment="Center" Padding="20,4,20,4" Click="a">OK</Button></Grid></UserControl>on></Grid></UserControl>
```

リスト８　Dotfuscator で難読化された XAML 定義

ClickOnce の難読化

ClickOnce は、クライアントアプリケーションの配布を容易にする.NET Framework の技術です。Web サイトなどで配布される ClickOnce アプリケーションは、新しいバージョンがリリースされると自動的に更新される仕組みを持っており、改ざんされたアプリケーションに更新されないよう署名しておく必要があります。

製品版の Dotfuscator は、入力として配置マニフェストファイル（.application）を指定するだけで、自動的に必要なファイルを解析します。ClickOnce の難読化に必要なことは、元のアプリケーションを作成する際に使われた証明書ファイルとパスワードをパッケージオプションとして指定することだけです。

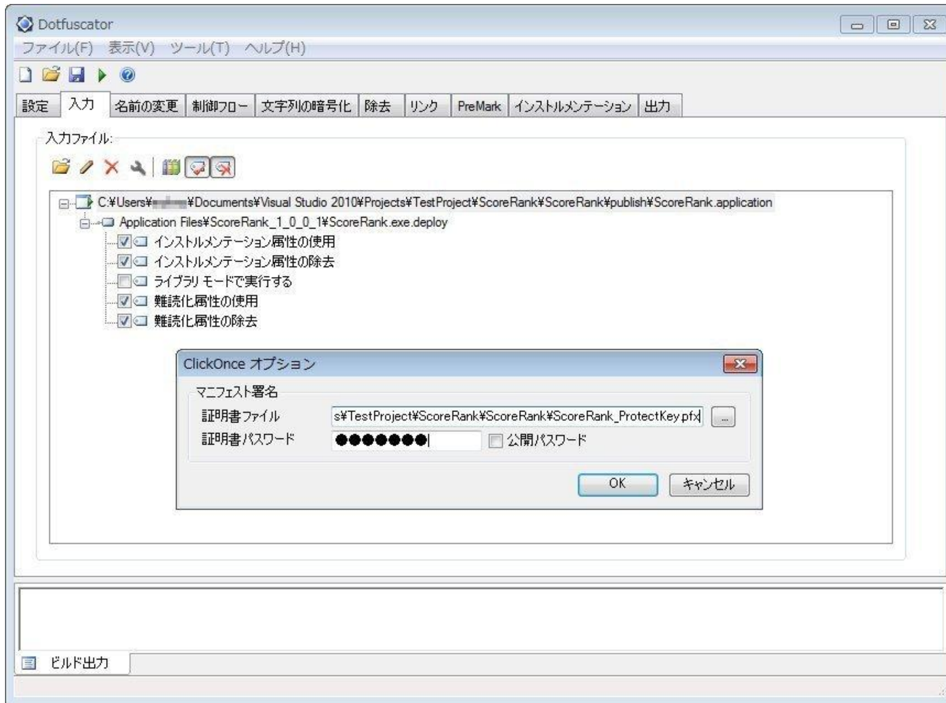

図 4　ClickOnce アプリケーションのためのオプションを設定

これにより、配布用のアセンブリ（.deploy）が自動的に難読化され、再度 ClickOnce で配布できるように署名されます。

難読化なくしてソフトウェアは守れない

ソースコードレベルで解析されるということは、ソースコードだけでなくソフトウェアそのものがリスクにさらされるということです。一般的なソフトウェアのプロテクト方式には、「ドングル方式」と呼ばれるハードウェアを使ったプロテクト、「アクティベーション方式」と呼ばれるネットワークを使った認証によるプロテクトが知られています。どちらも、それらが機能しているうちはソフトウェアを保護できますが、どんなに強力なプロテクトも、その処理を迂回されてしまえば効果がありません。

また、リバース・エンジニアリングと呼ばれるソフトウェアの解析については、法的に明確な保護があるとは言えない状況です。市販のソフトウェアの多くは、「使用許諾契約」によって、ソフトウェアの解析や逆コンパイルを禁じていますが、現在のところ、日本でリバース・エンジニアリングの違法性を認めた裁判例はありません。

かつて世間を騒がせたウィルスや迷惑メールは、Windows のセキュリティ対策やアンチウィルスソフトの浸透によって、さほど日常業務に影響しなくなりました。難読化によってソフトウェアの解析を完全に防止することはできませんが、解析に対するハードルを大きく引き上げることで実用上は十分な防御を施すことができます。ソフトウェアが攻撃されるのは愉快犯ではなく、金銭的利益を意図したものが増えています[10]。悪意の解析に対して手間のかかるように防御しておくことは、ソフトウェアの知的財産やセキュリティ保護のための有効な手段です。

もちろん、プロジェクトを開発する過程で作られる、あらゆるソフトウェアを難読化する必要はありません。個人の開発者レベルで作るツールや、画面設定を切り替えるといったユーティリティなどは、まったく難読化が必要ないか、Dotfuscator CE で処理できる難読化で十分対応できるものも多いでしょう。しかし、中規模以上のプロジェクト、あるいは開発に手間をかけたアプリケーションについては、製品版の Dotfuscator で難読化しておくことを強く推奨します。

Dotfuscator では「ビルドマシン」単位でのライセンスを採用しています。つまり、大規模なプロジェクトにおいても、すべての開発者が Dotfuscator を購入・使用する必要はなく、ソフトウェアの最終版をビルドする環境においてのみ Dotfuscator を導入すればよいのです[11]。

---

10 「シマンテック、金銭的利益を求めてホームユーザーを狙う攻撃が増加、と報告」（http://www.symantec.com/ja/jp/about/news/release/article.jsp?prid=20060927_01）など

11 システム全体で複数のビルドマシンを使う場合には、それに対応するライセンスが必要です。

開発プロジェクトへの責任

逆コンパイルなどの解析技術は日々進歩しています。.NET Framework が開発者に利便性の高い機能を提供することが、結果として解析を容易にさせる可能性もあります。ソースコードを厳重に管理する必要を感じているなら、バイナリからの解析に無防備であってはなりません。

「自分のソフトウェアが狙われることはない」という保証はどこにもありません。車を運転するときに「絶対に事故を起こさない自信がある」という人でも自動車保険に入らない人はまずいません。ほとんどの人は自動車保険の恩恵を受けることはありませんが、だからといって自動車保険に入らなくてもよいわけではありません。

いったん問題が発生したら、後からこれを回復することは困難であり、結果として企業の信頼を揺るがすことにもなります。重要な開発プロジェクトに対しては、製品版の Dotfuscator で難読化することをご検討ください。そうすることで、お客様から開発プロジェクトのセキュリティ対策を尋ねられた時に、自信を持って万全の態勢をとっているとお答えになれるでしょう。アプリケーションに対して十分な難読化を施すことによって、はじめて知的財産の保護や情報セキュリティに対する責任を果たせるのです。