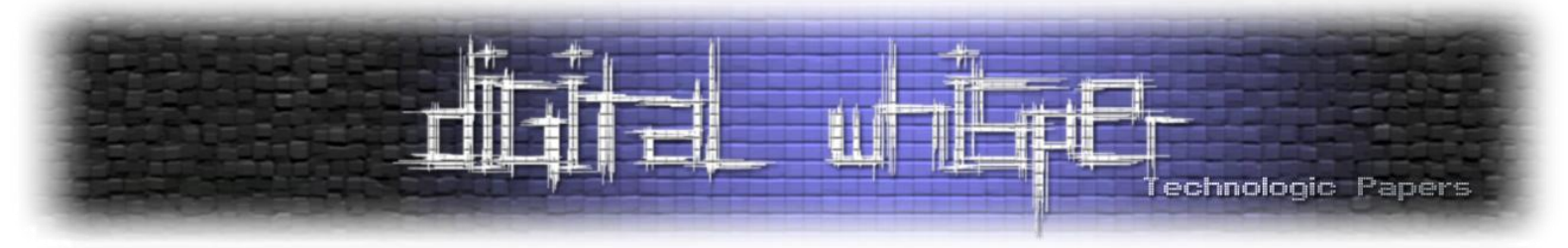


.NET Reverse Engineering

מאת: בר

הקדמה

ב-2002 מיקרוסופט יצאה בהכרזה על פלטפורמה ששמה NET. הפלטפורמה מספקת ממשק פיתוח אחיד, הן לתוכנות שולחניות, הן לפיתוח אתרי Web, הן לתוכנות לסמארטפונים המריצים Windows Phone ואף מאפשרת לקשור ביחד התקני מחשוב שונים ולשתף ביניהם יישומים ומידע. היא פותחה כמענה תחרותי לארכיטקטורת J2EE מבית Sum Microsystems. בתחום התוכנות השולחניות, מספקת NET. ספרייה אחידה לפיתוח על כל משפחת מערכת ההפעלה חלונות, החל מגרסת Windows 98, במקום ממשקי תכנות היישומים של מערכות ההפעלה עצמן, או ספריית MFC.

במאמר זה נביא לכם טעימה מניתוח תוכנות NET. ונציג מעט איך מגנים עליהן. מאמרים על הנושא בעברית כמעט ולא קיימים, אך יחד עם זאת בארץ יש דווקא קהילה גדולה של מפתחים ב-NET. במאמר נשלב דוגמאות קוד גם ב-C# וגם Cil ונסביר על נושא ערפול קוד (Obfuscated code) ב-NET.

שפות התכנות בסביבת NET. מהודרות לשפת ביניים הנקראת Common Intermediate Language. את שפת ביניים זו אפשר בעזרת כלים פשוטים להמיר לכל שפה ב-NET. עובדה זו גרמה לכך שההגנה על אלגוריתם והזכויות של מפתחיו הפכו להיות יותר קשים לאכיפה מאשר בעבר, כאשר היה מדובר בקוד המקומפל של כל שפה.

Code Refactoring

ראשית נציג כמה תוכנות שקימות ברשת לשחזור קוד לשפה גבוהה (מ-Cil ל-C# או אל כל שפת NET. אחרת). מכיוון שמטרת הערפול היא למנוע מתוכנות אלו לתרגם את הקוד נעבור על התוכנות העיקריות אבל חשוב לדעת שיש עוד תוכנות שניתן למצוא בחיפושים ברשת.

ILdasm:

Ildasm אולי לא מסוגלת להפוך את הקוד בחזרה לשפה גבוהה אבל יש לה מקום של כבוד כאחת התוכנות הראשונות שאפשרו להראות את ה-Cil ומידע נוסף על קבצי NET. היא מציגה מגוון אופציות,

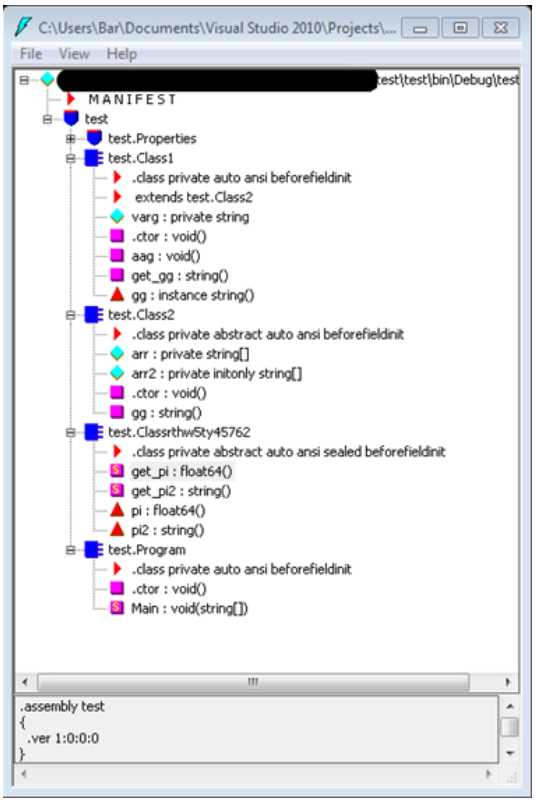
החל מצפייה ב-Headers, ועד סטטיסטיקות על הקובץ וכמובן אפשרות לראות את ה-Cil של קובץ כולל ה-Byte Code.

היתרונות העיקריים:

- מגיעה עם Visual Studio.
- מאפשרת צפייה ב-Headers.
- מאפשרת צפייה ב-Byte code לצד ה-Cil.

החסרונות העיקריים:

- לא מאפשרת המרה לשפה גבוה.
- קל מאוד למנוע ממנה להראות את הקוד.
- לא תומכת בתוספים.
- עולה כסף (עלות של Visual Studio).



.NET Reflector

.NET Reflector היא אחת התוכנות הנפוצות והמוכרות ביותר בתחום. יחסית ל-Ildasm היא נראית הרבה פחות מאיימת מבחינת הממשק.

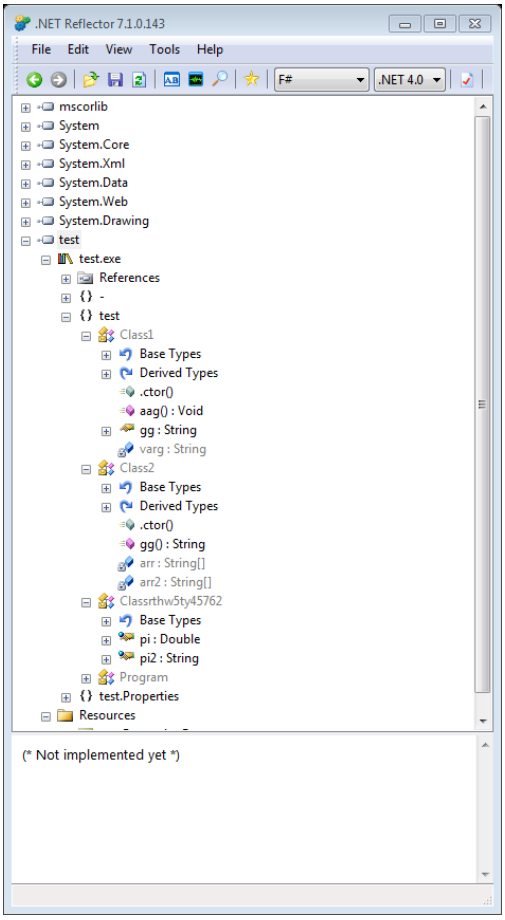
אחת מהתכונות הבולטות שלה היא האפשרות להחזיר את הקוד למגוון שפות, ובנוסף לגרפיקה החלקה של התוכנה, יש לה גם התאמות מובנת לתוך Windows ול-Visual Studio.

היתרונות עיקרים:

- תמיכה בהמרה ל-6 שפות עליות (C# ,Visual basic ,Delphi ,MC++,Oxygene,F#).
- תמיכה בהרחבות.
- סימניות.

חסרונות עיקריים:

- עולה כסף.
- לא יודעת להתמודד עם ערבול קוד בסיסי.



ILSpy

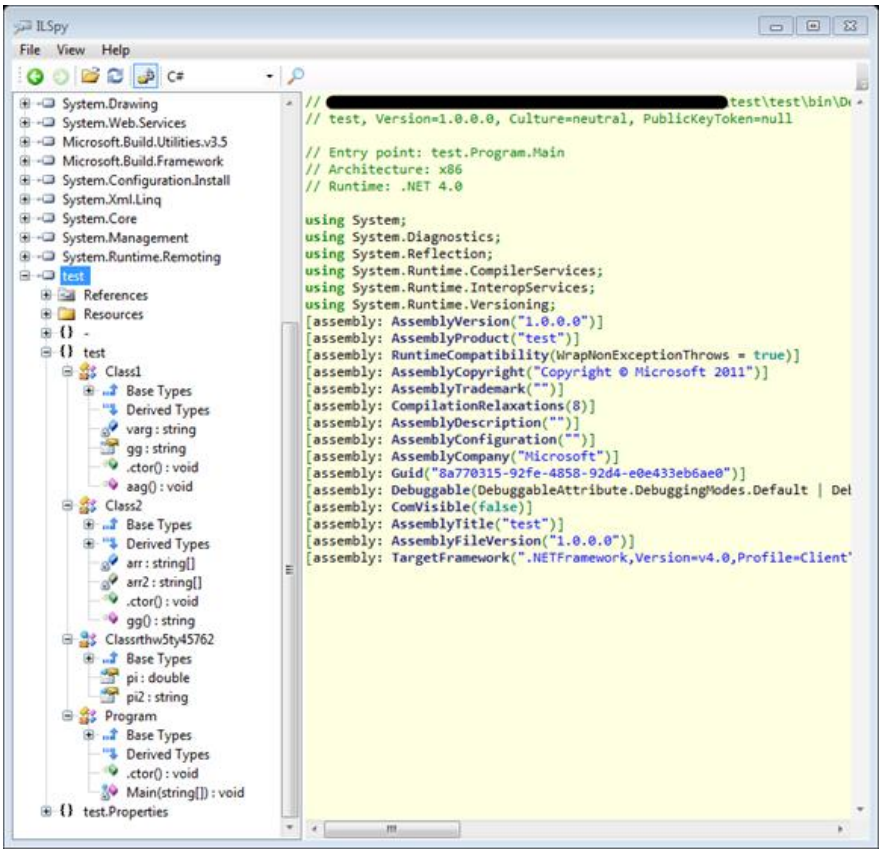
ILSpy היא האחת התוכנות הטובות בתחום. הפיתוח של ILSpy החל כאשר Red Gate, המפתחת של NET Reflector הצהירה שהיא מפסיקה את הגרסה החינמית של ה-NET Reflector. כיום התוכנה הזאת עולה על מרבית התוכנות הקיימות כיום בשוק. היא מפותחת על ידי הצוות של SharpDevelop שמפיצים גם קומפיילר ל-NET., ולפי ה-Road Map באתר הפיתוח בגרסאות הבאות היא תכלול גם Debugger ותכונות נוספות.

יתרונות עיקריים:

- קוד פתוח.
- תמיכה בהרחבות.
- עמידות גבוהה לקוד ספגטי ולערפול בכלל.

חסרונות עיקריים:

- לא מכילה פונקציות מתקדמות.
- תמיכה בהמרה לשפה גבוהה אחת - שפת C#.



תוכנות נוספות:

תוכנות נוספות עליהן לא נרחיב אבל שוות בדיקה לקוראים המתעניינים: dotPeek, JustDecompile ו-CodeReflect. בהמשך המאמר נתייחס למושג "תוכנת שחזור" כאל אחת התוכנות שהצגנו. במאמר, אנו נשתמש ב-ILSpy. בגלל העמידות הגבוהה שלה בפני קוד ספגטי וערפול בכללי, וגם ב-NET Reflector. משום שרוב התוכנות שמבצעות ערפול מדגימות עליה את יכולתן.

Cil

כאשר יש לנו את הכלים, אפשר לעבור על שפת Cil ועל פקודות הבסיסיות בה.

שפת Cil כמו רוב שפות הסף עובדת על מחסנית. למידע נוסף:

http://he.wikipedia.org/wiki/מחסנית_(מבנה_נתונים)

הפונקציה הבסיסית/ראשית ב-#C היא ה-Main, ממנה בעצם מתחילה עליית התוכנית שלנו. הפונקציה כמו שהיא מופיעה ב-Visual Studio:

```
private static void Main(string[] args)
{ }
```

אחרי שמקמפלים את התוכנית, נוצר קובץ הרצה, נפתח את הקובץ בעזרת אחת מתוכנות ה-Refactoring:

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2050
    // Code size 2 (0x2)
    .maxstack 8
    .entrypoint

    IL_0000: nop
    IL_0001: ret
} // end of method Program::Main
```

הסבר על מה שאנחנו רואים:

.method	סימון שזו פונקציה.
private	סימון רמת הגישה לפונקציה.
static	מקטע סטטי, לא שייך לעצם ספציפי במחלקה אלא לכולם.
Void	החזר הפונקציה.
Main	שם הפונקציה.
string[] args	פרמטרים.
cil managed	הצהרה שהקוד מנוהל.
RVA 0x2050	מיקום הפונקציה.
.maxstack 8	גודל המחסנית.
.entrypoint	נקודת כניסה ,הפונקציה ראשונה שתופעל עם הפעלת הקובץ.
IL_0000: nop	פקודת No Operation (לא עושה כלום).
IL_0001: ret	חזרה מהפונקציה (אם החזר הפונקציה שונה מ-void אז הערך שיוחזר הוא הראשון מחסנית)

כמעט כל הפקודות ב-Cil הם מאוד פשוטות וקלות להבנה לדוגמה, את כל הפקודות תוכלו למצוא ב:

http://en.wikipedia.org/wiki/List_of_CIL_instructions

לדוגמא לטעינה של מספרים למחסנית בעזרת ldc. בעת הטעינה חשוב לציין את סוגו של המספר:

```
i4  - int32
i8  - int 64
r4  - flot32
```

ולאחר סוגו את המספר עצמו: אם המספר הוא בין 0 ל-9 כותבים נקודה ואת המספר, אחרת כותבים רווח ואז ישר את המספר. לדוגמה:

```
ldc.i4.1
```

יטען את המספר 1 מסוג int32 למחסנית.

```
ldc.i8 9654
```

יטען את המספר 9654 מסוג int64.

מחרוזות ניתן לטעון בעזרת הפוקדה "Ldstr". לדוגמא, הפקודה הבאה:

```
ldstr "DigitalWhisper"
```

דוחפת למחסנית את המחרוזת DigitalWhisper

עוד פעולות חשובות שטוב לדעת:

- Call - קריאה לפונקציה אחרת בקובץ.
- Ceq - השווה של 2 אובייקטים (מחזירה True או False).

קפיצות והפניות:

קיימות מספר רב של דרכים לבצע קפיצות והפניות, נציג 2 מהן:

הראשונה היא br.s: שהיא ה-GoTo של ה-Cil, משתמשים בה הרבה בלולאות ועוד יותר בערפול קוד כדי להקשות על מי שינסה לקרוא את הקוד. ה-s. שבה אחרי br מציין short הגרסה הקלה של הפקודה שאחריה באה כתובת בגודל Byte אחד בלי ה-s. הכתובת היא בגודל 4 בייטים.

דוגמה:

```
br.s IL_0030
```

הפקודה השנייה היא brtrue.s שקופצת לכתובת שכתובה אם הערך האחרון במחסנית הוא true או שונה מאפס. בדוגמה, אם הערך במחסנית הוא true אז נקפוץ לשורה IL_0011 אחרת הוא ימשיך לשורה IL_000d:

```
IL_000b: brtrue.s IL_0011
IL_000d: nop
```

לאחר שראינו מספר פעולות בסיסיות ננתח את הקוד הבא (שכתוב ב-Cil):

```
.method public hidebysig static
    int32 sod () cil managed
{
    // Method begins at RVA 0x2068
    // Code size 27 (0x1b)
    .maxstack 2
    .locals init (
        [0] int32 num,
        [1] bool mybool,
        [2] int32 CS$1$0000,
        [3] bool CS$4$0001
    )

    IL_0000: nop
```

```
    IL_0001: ldc.i4.0
    IL_0002: stloc.0
    IL_0003: ldc.i4.1
    IL_0004: stloc.1
    IL_0005: ldloc.1
    IL_0006: ldc.i4.0
    IL_0007: ceq
    IL_0009: stloc.3
    IL_000a: ldloc.3
    IL_000b: brtrue.s IL_0015

    IL_000d: nop
    IL_000e: ldc.i4 1234
    IL_0013: stloc.0
    IL_0014: nop

    IL_0015: ldloc.0
    IL_0016: stloc.2
    IL_0017: br.s IL_001b
   IL_0019: ldc.i4.0
    IL_001a: stloc.2

    IL_001b: ldloc.2
    IL_001c: ret
} // end of method Program::sod
```

מה אנחנו רואים?

מהשורה הראשונה ניתן להביא כי מדובר בפונקציה סטטית בשם "sod" שמחזירה int ולא מקבלת פרמטרים. ה-RVA וגודל הקוד והמחסנית לא מעניינים אותנו בשלב זה.

לאחר מכן מגיעה שורה בה עוד לא נתקלנו. זוהי השורה שבה מוגדרים המשתנים (המשתנים תמיד מוגדרים בתחילת הפונקציה).(לפעמים הקומפיילר מוסיף משתני עזר משלו, בדרך כלל יהיו להם שמות כגון: CS$1$0000.) ניתן לראות שמוגדרים פה 4 משתנים: 2 מסוג int ו-2 מסוג bool. המשתנים עצמם ממוספרים. לכל משתנה יש מספר שבעזרתו מתייחסים אליו.

לימוד מהיר:

- Stloc - פקודת אחסון ערך מהמחסנית למשתנה.
- Ldloc - פקודת טעינת ערך ממשתנה למחסנית.

עד שורה IL_0003 אנחנו מאתחלים את משתנה מספר 0 (num) בערך 0.

עד שורה IL_000b אנחנו מאתחלים את משתנה מספר 1 (mybool) בערך 1 ומשווים (ceq) אותו ל-0 ואת התוצאה מאחסנים מהמחסנית למשתנה מספר 3 (CS$4$0001) ולאחר מכן מציבים את הערך של

משתנה 3 במחסנית. כאן מתבצעת בדיקה אם הערך שבמחסנית הוא true אזי נקפוץ לשורה IL_0011 אחרת נעבור לשורה IL_000d.

מכיוון ש-1 שונה מ-0 אז תוצאת ההשוואה היא false ולכן נעבור לשורה IL_000d שלא עושה כלום.

עד שורה IL_0011 אנחנו מציבים 1234 במשתנה מספר 0 (num) ועד שורה IL_0017 אנחנו מעבירים את הערך של משתנה 0 למשתנה 2.

לבסוף, אנו מגיעים לפקודת קפיצה הישירה (br או br.s) שמורה לנו לקפוץ בקוד ל-IL_001b, שטוענת למחסנית את הערך שבמשתנה 2 **ומחזירה** את הערך שבמשתנה 2 (שהוא: 1234).

הקוד כפי שהוא נכתב במקור ב-#C:

```
public static int sod()
{
      int num = 0;
      bool mybool = true;
      if (mybool)
      {
         num = 1234;
      }

      return num;
}
```

[הערה: הקוד ב-cil נערך קצת למטרות ההדגמה (שורה IL_0019- IL_001a נוספו באופן ידני) אבל אתם יכול לקמפל את הקוד ה-#C בעצמכם וליראות איך הקוד נראה.]

טכניקות Obfuscation:

לאחר הדוגמאות נעבור לקטע המעניין (הקטע של איך כל זה מתקשר לאבטחת מידע). מכוון שאפשר בקלות להפוך את הקוד לשפה עילית ("Reflecting") בעזרת מגוון התוכנות השונות קיימות בשוק מספר תוכנות שמטרתם להקשות על ביצוע ההמרה לשפה עילית וגם מקשות את קריאה בשפת הביניים (cil).

קיימות מספר דרכים שבהן אותן תוכנות מבצעות שימוש בכדי לבצע Reflection:

הדרך הקלאסית:

הרעיון של השיטה הוא להשתמש בשיטות שמבצעים על קבצי exe רגילים כגון Packing , קריאה לפונקציות מה-api של windows כגון isdebuggerpresent ו-isdebuggerattached. ניתן לראות שחלק מתוכנות ערפול מצפינות גם את ה-Byte code של הפונקציות שכתובות ב-Cil ובזמן ריצת התוכנה מבצאות Hooking ל-JIT בכדי לתרגם את הקוד בחזרה.

strong name:

Strong Name הוא שם הקובץ ללא הסיומת + הגרסה, ה-Culture, ארכיטקטורת המעבד וה-PublicKeyToken, לדוגמה:

test, Version=1.0.0.0, Culture=neutral, PublicKeyToken=80c0e92d0d9731ca

הדבר נועד למנוע את בעיית ה-"DLL Hell" (http://en.wikipedia.org/wiki/DLL_hell). אחת הדרכים שבהן הוא מבצע שימוש בכדי למנוע זאת היא על ידי בנייה של מפתח ציבורי בגודל 1024 ביט, ועליו הוא מבצע SHA-1 ולוקח את 8 הביטים האחרונים ושם אותם כ-PublicKeyToekn. כאשר התוכנה מתקמפלת הקומפילר חותם את קובץ של התוכנה בעזרת מפתח פרטי שתואם למפתח הציבורי (שממנו נוצר ה-PublicKeyToken) ובזמן הריצה המערכת בודקת האם הקובץ חתום. תנאי זה מאפשר את ריצת התוכנה. במידה ותהיה אי-התאמה בין המפתח לקובץ- הקובץ לא ירוץ.

ע"י כך ניתן לראות האם הקובץ לא ניזוק או השתנה -וברגע שהקובץ ניזוק או השתנה אז הקובץ לא יופעל, וכך גם מונעים מגורמים לא רצויים לשנות את הקובץ. יש אנשים שחשבו צעד אחד קדימה והוסיפו גם בדיקות בקוד להימצאות החתימה. קוד לדוגמה (בהנחה שהקובץ נחתם):

```
bool isSignAsmm = Assembly.GetExecutingAssembly().GetName().GetPublicKey().Length == 0;
Console.WriteLine("IsAssemblySignedDelete=" + isSignAsmm);
```

בדיקה קצת יותר נוקשה:

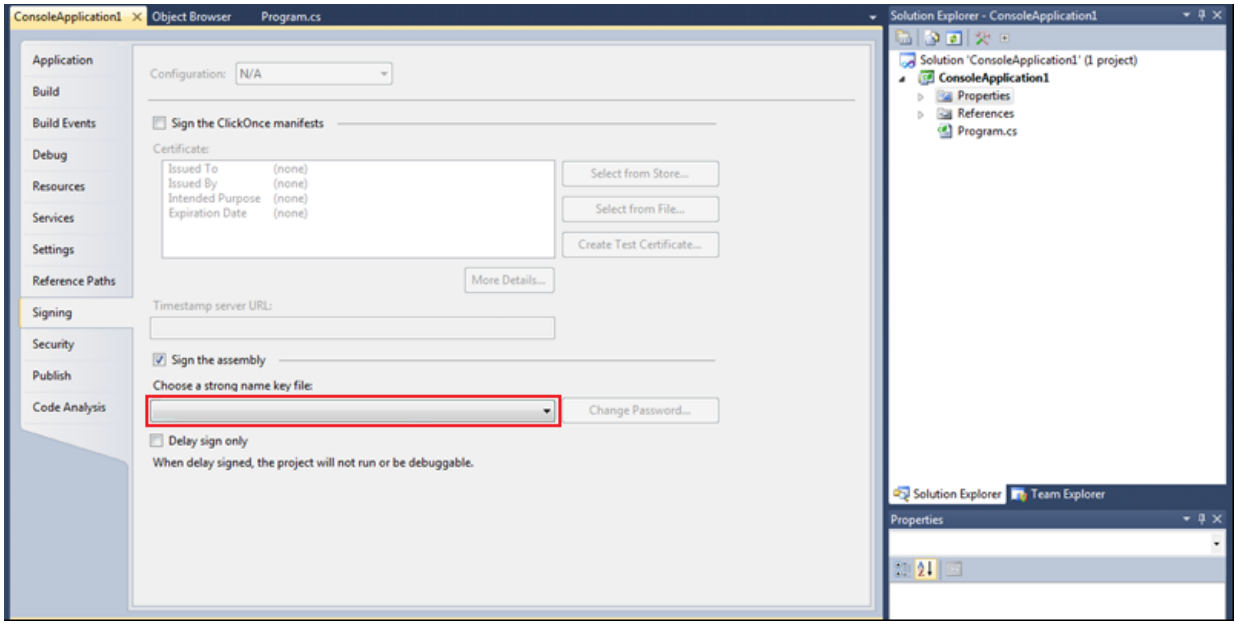
```
[DllImport("mscorwks.dll", SetLastError = true)]
[return: MarshalAs(UnmanagedType.U1)]
static extern bool StrongNameSignatureVerificationEx(
        [MarshalAs(UnmanagedType.LPWStr)] string wszFilePath,
        [MarshalAs(UnmanagedType.U1)] bool fForceVerification,
        [MarshalAs(UnmanagedType.U1)] ref bool pfWasVerified
        );
         /*
         *wszFilePath= הפונקציה מקבלת את מקום הקובץ
          * fForceVerification=האם לבדוק בניגוד להגדרות
          * pfWasVerified=האם הבדיקה נעשה
          * בסדר הקובץ אם true ומחזירה
          */

private static void Main(string[] args)
{
        bool WasVerified = false;
        Assembly executingAssembly = Assembly.GetExecutingAssembly();
        var VerificationOk=StrongNameSignatureVerificationEx(executingAssembly.Location,
        true, ref WasVerified);
        if (!WasVerified || !VerificationOk)
        {
             throw new SecurityException("Assembly has been tampered");
             /*אם הבדיקה לא נעשתה או שהבדיקה נעשה שלילית*/
        }
}
```

ועוד בדיקה אחרונה ודי פשוטה:

```
if (!executingAssembly.FullName.EndsWith("80c0e92d0d9731ca"))
//פה נכתוב את ה-PublicKeyToken של הקובץ
{
      throw new SecurityException("Assembly has been tampered");
}
```

איך יוצרים חתימה?

בשביל לבצע חתימה לקובץ לא צריך להשתמש בכלים מיוחדים. נסביר את דרך הפעולה: כל מה שצריך זה להיכנס למאפייני הפרויקט ללחוץ על הטאב "signing", לסמן ב-v את "sign the assembly" ולאחר מכן ללחוץ על התיבה הריקה:



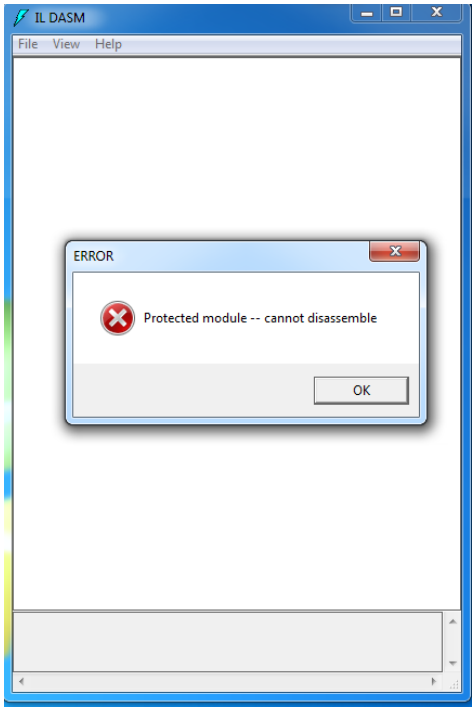
כעת יש לבחור ב-"<new>" ושם לבחור את שם המפתח ואת הסיסמה והקובץ שלנו חתום.

Suppressing ILDASM:

מיקרוסופט הוסיפה אפשרות למנוע מ-ildasm לנתח תוכנות ולהציג שגיאה במקום. בכדי לגרום ל-ildasm להציג שגיאה כל מה שצריך זה להוסיף את ה-Attribute הבא לתוכנה שלכם:

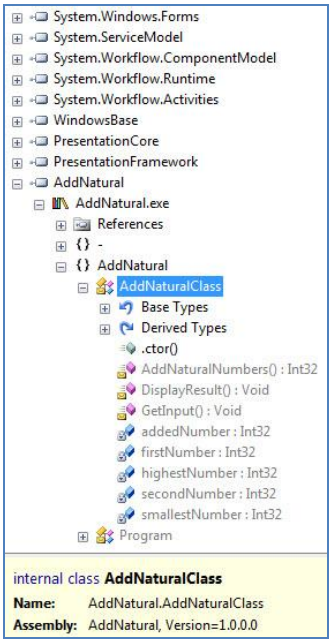
```
[assembly: SuppressIldasmAttribute()]
```

והתוצאה ב-ildasm תהיה:

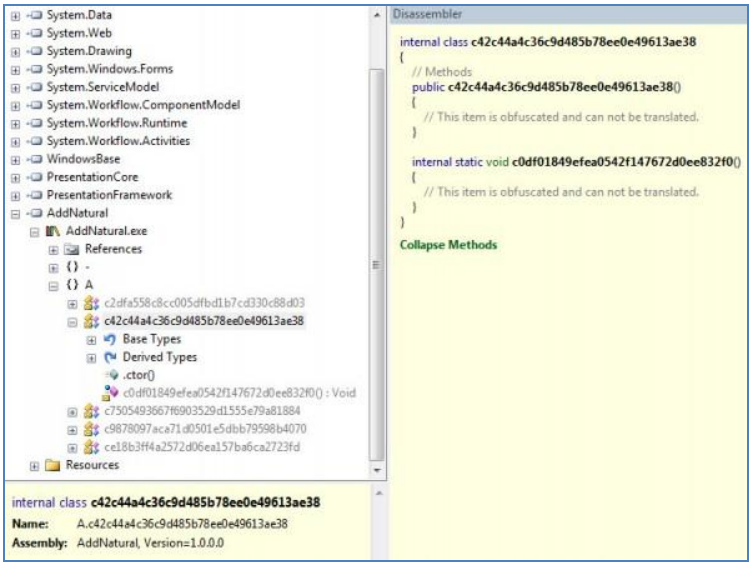


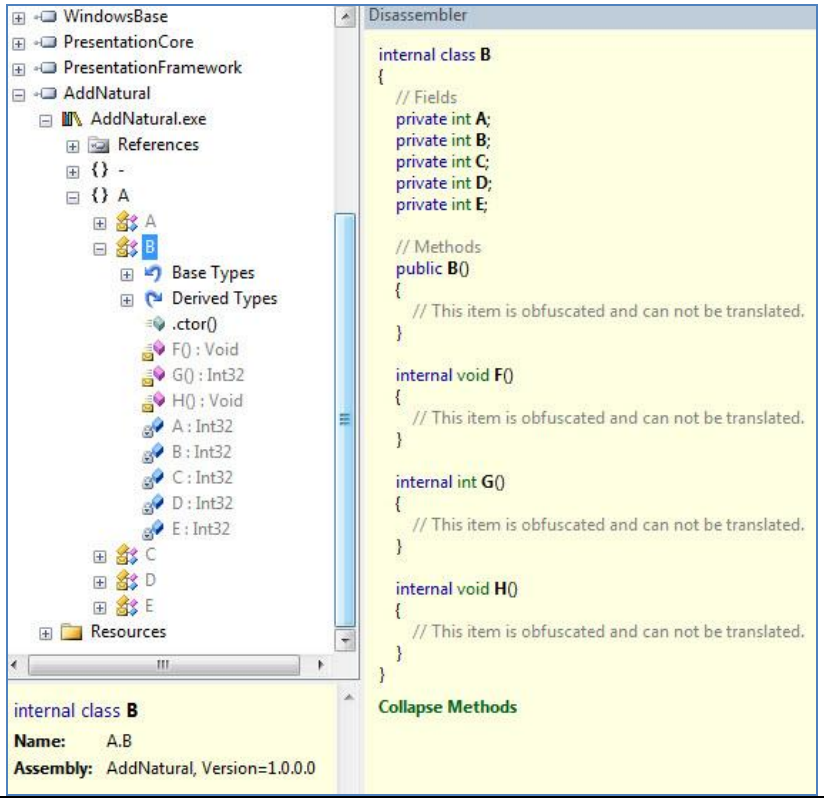
שינוי שמות:

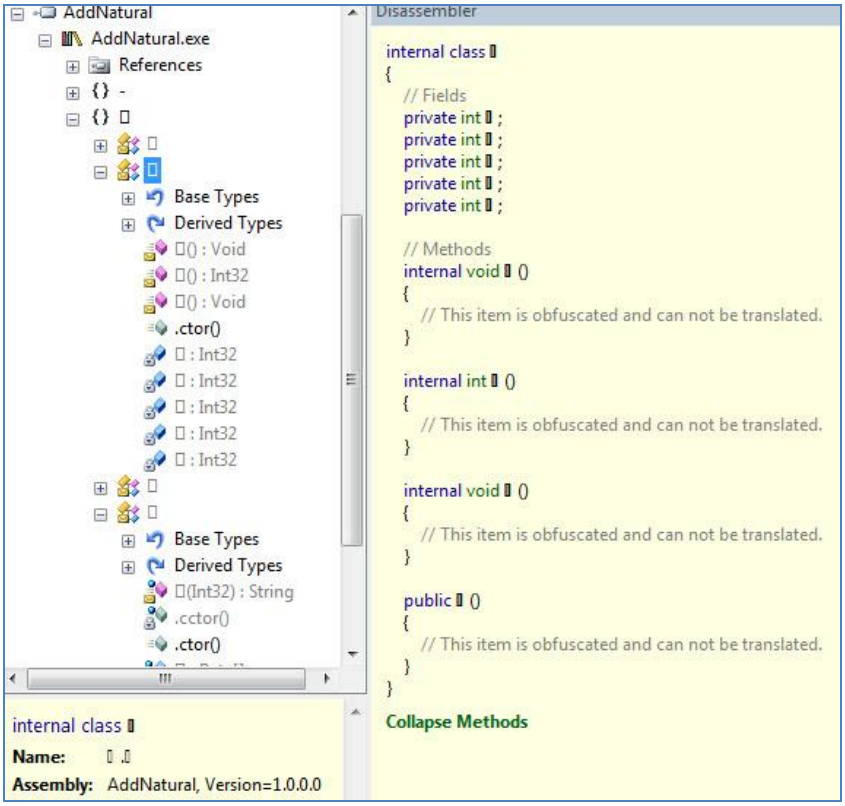
שינוי שמות של פונקציות, פרמטרים, מחלקות וכו', היא שיטה מאוד נפוצה אשר יש לה מספר וריאציות נציג את הווריאציות העיקריות ואת היתרונותיהם: הקוד ללא שינוי כפי שהוא נראה ב-Reflector:



1. **הצפנת השם:** שם הפונקציה, המחלקה או שם המשתנה מוצפנים כך שבהמרה של קוד הביניים חזרה לשפת המקור מקבלים שמות מוצפנים. בניגוד לשאר הווריאציות, שיטה זאת מאפשרת לשחזר את שמות הפונקציות למצבם הקודם. דבר זה יכול לעזור במקרה של שגיאה, אפשר לקחת את ה-stacktrace של התוכנה, להוריד את ההצפנה ולראות את השמות האמיתיים של האובייקטים.







2. **כתיבת אות במקום שם:** בוריאציה זו מחליפים את שם הפונקציה, המחלקה או המשתנה באות בודדת, שיטה זו בעיקר מעצבנת ומבלבלת וברב המקרים לא מאפשרת הבנה של ה-stacktrace.

3. **תווים מיוחדים**: בוריאציה זו משנים את שם הפונקציה לתו לא מוגדר (הערכים של תווים אלו הם בין 0x20 ל- 0xD7FF) או לתו מיוחד אחר (כמו סמיילי ☺ שיש לו את הערך 0x263a או כל תו מוזר אחר). דוגמה לתווים לא מוגדרים:

```
Disassembler

private void OpenFile()
{
   OpenFileDialog dialog = new OpenFileDialog {
      Filter = □.□(0xa11),
      FilterIndex = 1,
      RestoreDirectory = true
   };
```

4. **הצפנה:** בשיטה זו מצפינים את המחרוזות כך שלא היה אפשר לזהות אותם בעזרת ה-reflctor אלגוריתם ההצפנה וגם המפתח אם קיים צריכים להיות בתוך התוכנה כדי שבזמן ריצה התוכנה

תוכל להפוך את המחרוזת המוצפנת למחרוזת לא מוצפנת בנוסף למחרוזות בדרך כלל גם מצפינים את המשאבים (Resource) של התוכנה.

5. **invalid bytecode:** בשיטה זו מכניסים לפונקציה ביטים שאין להם משמעות (הם לא פקודה מוכרת) ובכך מבלבלים את ה-reflctor ומונעים ממנו לתרגם את הקוד.

6. **ערבול קוד, קוד ספגטי וקוד שלא עושה כלום:** כל אחת מהדרכים שכתובות למעלה עושות את הקוד קשה יותר לקריאה ב-cil וקשה יותר לתרגום לשפה עילית. דוגמה לקוד שעבר ערבול (הקוד נכתב ב-#C למען הנוחות) קפיצות בקוד:

```
            int t = 4;
            goto jump;
        jump3:
            t = t + 6;
            goto end;
        jump2:
            t = 4;
            goto jump3;
        jump:
            t = t + 3;
            goto jump2;
        end:
Console.WriteLine(t);
```

הקוד אחרי הורדת הקפיצות:

```
            int t = 4;
            t = t + 3;
            t = 4;
            t = t + 6;
Console.WriteLine(t);
```

הקוד המקורי:

```
            int t = 4;
            t = t + 6;
Console.WriteLine(t);
```

עוד דוגמה רק הפעם ערבול קוד Cil עם הערות ושורות ממוספרות לפי סדר ביצוע ההוראות:

```
.method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2058
    // Code size 19 (0x13)
    .maxstack 2
    .entrypoint
    .locals init (
```

```
        [0] int32 num
    )

[1] IL_0000: ldc.i4.4
[2] IL_0001: br.s IL_000f
    // loop start (head: IL_000f)
[3]     IL_0003: ldloc.0
[4]      IL_0004: ldc.i4.6
[5]      IL_0005: add
[6]      IL_0006: stloc.0
[7]      IL_0007: ldloc.0
[8]      IL_0008: call void [mscorlib]System.Console::WriteLine(int32)
[9]      IL_000d: br.s IL_0014

[10]     IL_000f: stloc.0
[11]     IL_0010: br.s IL_0003
    // end loop
         IL_0012:  ldc.i4.4
         IL_0013: stloc.0
[12]IL_0014: ret
[13]} // end of method Program::Main
```

מקרא:
קוד שאף פעם לא מתבצע
קפיצות מיותרות

קריאה לפונקציה חיצונית (WriteLine) ומקבלת פרמטר אחד מסוג int32 היא נמצאת בקובץ mscorlib תחת המחלקה Console.

1	הגדרת משתנים
2	דוחפים למחסנית 4
3	קפיצה ל-IL_000f
4	טעינת משתנה 0 אל המחסנית
5	קפיצה ל-IL_0003
6	דוחפים למחסנית 6
7	חיבור 2 איברים אחרונים במחסנית
8	טעינת התוצאה מהמחסנית אל משתנה מספר 0
9	דוחפים למחסנית 6
10	טעינת משתנה 0 אל המחסנית
11	קפיצה ל-IL_0014
12	הוצאת איבר מהמחסנית למשתנה מספר 0 (num)
13	קפיצה ל-IL_0003

בכדי למחוק את ערבול הקוד צריך:

- לזהות מי מה-br לא שייך ללולאה (במקרה שלנו כולם...).
- לראות איזה קוד אף פעם לא מתבצע (בדרך כלל תהיה קפיצה לפני הקוד וגם אף קפיצה לא תפנה לאותו קוד)
- לבסוף, לבנות את הקוד מחדש ולסדר את ה-Offsets.

התוצאה תראה כך:

```
.method private hidebysig static .method private hidebysig static
    void Main (
        string[] args
    ) cil managed
{
    // Method begins at RVA 0x2058
    // Code size 19 (0x13)
    .maxstack 2
    .entrypoint
    .locals init (
        [0] int32 num
    )
    IL_0000: ldc.i4.4
    IL_0001: stloc.0
    IL_0002: ldloc.0
    IL_0003: ldc.i4.6
    IL_0004: add
     IL_0005: stloc.0
     IL_0006: ldloc.0
     IL_0007: call void [mscorlib]System.Console::WriteLine(int32)
    IL_00c: ret
} // end of method Program::Main
```

אתגר פשוט לתרגול עבור הקוראים: מה ידפיס קטע הקוד הבא:

```
.method private hidebysig static     void Main ( string[]
args     ) cil managed
{
    // Method begins at RVA 0x2058
    // Code size 69 (0x45)
    .maxstack 2
    .entrypoint
    .locals init ( [0] int32 a, [1] int32 b, [2] bool  c)

    IL_0000: ldc.i4.6
    IL_0001: br.s IL_0041
    // loop start (head: IL_0041)
        IL_0003: ldloc.0
        IL_0004: ldc.i4.8
        IL_0005: add
        IL_0006: stloc.0
        IL_0007: br.s IL_0009
        IL_0009: ldc.i4.0
        IL_000a: br.s IL_0036
```

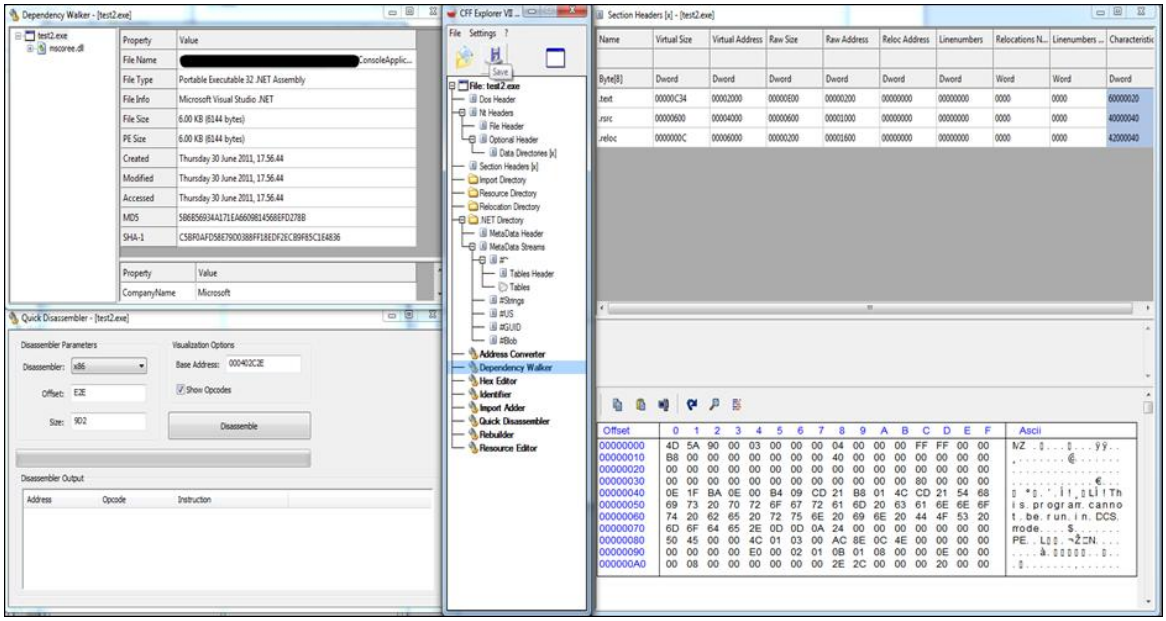
```
        // loop start (head: IL_0036)
            IL_000c: ldc.i4.1
            IL_000d: stloc.2
            IL_000e: br.s IL_0010

            IL_0010: ldloc.0
            IL_0011: ldc.i4.s 25
            IL_0013: add
            IL_0014: br.s IL_002c

            IL_0016: ldc.i4.0
            IL_0017: stloc.2
            IL_0018: ldc.i4.0
            IL_0019: stloc.1
            IL_001a: br.s IL_0025

            IL_001c: ldloc.0
            IL_001d: ldc.i4.s 22
            IL_001f: sub
            IL_0020: stloc.0
            IL_0021: ldloc.1
            IL_0022: ldc.i4.1
            IL_0023: add
            IL_0024: stloc.1

            IL_0025: ldloc.1
            IL_0026: ldc.i4.0
            IL_0027: ceq
            IL_0029: stloc.2
            IL_002a: br.s IL_002f

            IL_002c: stloc.0
            IL_002d: br.s IL_0016
            IL_002f: ldloc.2
            IL_0030: brtrue.s IL_0034

            IL_0032: br.s IL_0039

            IL_0034: br.s IL_001c

            IL_0036: stloc.2
            IL_0037: br.s IL_000c
        // end loop

        IL_0039: ldloc.0
        IL_003a: call void [mscorlib]System.Console::WriteLine(int32)
        IL_003f: br.s IL_0044

        IL_0041: stloc.0
        IL_0042: br.s IL_0003
    // end loop

    IL_0044: ret
} // end of method Class0::Main
```

כמה דברים לפני סיום

הייתי רוצה להציג מספר דברים לפני סיום:

- כלי שימושי למחקר קבצי net. ובכלל קבצים ברי הרצה, שמו CFF Explore ויש לו מבחר פונקציות. עוד מידע תוכלו למצוא באתר:

http://www.ntcore.com/exsuite.php



- ספרייה בשם mono cecil היא ספרייה בקוד פתוח ובעזרתה אפשר לנתח לשנות ולערוך קבצי net. היא מתעדכנת באופן תדיר ובנוסף היא נמצאת בשימוש בilspy שדיברנו עליו בתחילת הכתבה. עוד מידע בכתובת http://www.mono-project.com/Cecil

סיכום

כאן הגענו לסיומו של המאמר, מדובר במעין חלק ראשון לקראת מאמר שככל הנראה יפורסם בגליון הבא בנוגע למחקר שביצעתי בנושא. במאמר הבא אציג כלי שכתבתי שבעזרתו ניתן לתקן ולהוריד חלק מהמנגנונים שהוצגו במאמר זה בנוסף אנסה לעבור על כל השיטות שבמאמר זה ואנסה להסביר איך אפשר להוריד אותם ידנית.