Decompilation Injection

June 29

2009

This paper presents a novel way to protect .NET assemblies against reverse-engineering and recompilation. By injecting them with commands that are activated only at the recompilation stage, the application retroactively detects the reverse-engineering process and acts upon it.

CHECKMARX
Research Labs

Maty SIMAN, CISSP
maty@checkmarx.com

CHECKMARX
SOURCE CODE ANALYSIS TECHNOLOGIES

Contents

Abstract

This paper presents a novel way to protect .NET assemblies against reverse-engineering and decompilation by injecting them with commands that are activated only at the recompilation stage, the application retroactively detects the reverse-engineering process and acts upon it.

This technique goes beyond standard obfuscation processes and not only makes it difficult to reverse-engineer the code, but rather allows the application to actively respond to such attempts.

Many thanks to Alex Roichman and Adar Weidman (alexr,adarw @ checkmarx.com) from the Checkmarx Team for their invaluable comments.

Prologue

.NET Assemblies Overview

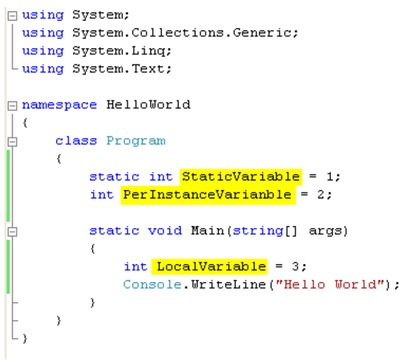
An *assembly,* in the Microsoft .NET framework, is a semi-compiled code library, which contains CIL commands (*Common Intermediate Language* command – previously known as *MSIL*). The assembly is usually created using a .NET compiler of a language such as C#, VB.Net, etc.
The assembly in-turn is compiled into machine-language at runtime using the *CLR.*

.NET Assemblies are comprised of several parts, one of which is the *Metadata* carrying the information about implementations, declarations and references specific to that component. As .NET components are designed to be *self-describing* (unlike COM and CORBA, for example), they contain all the necessary information to operate correctly.

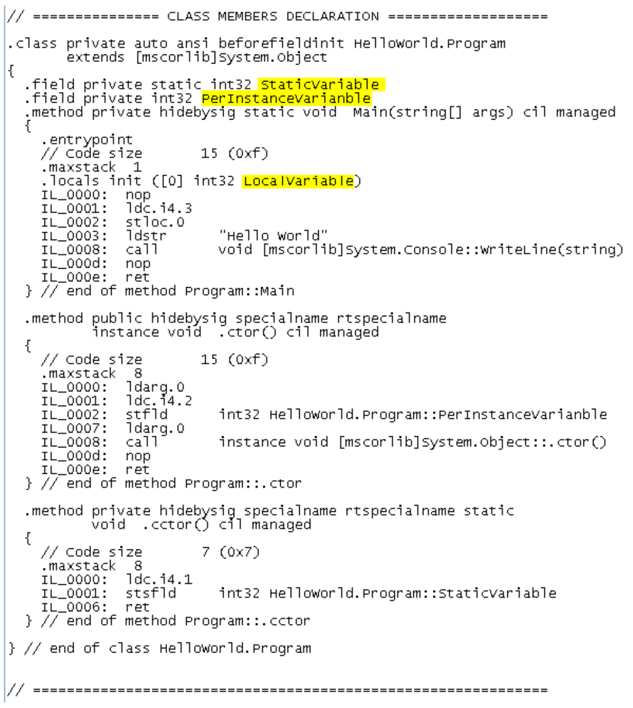
The *CLI* (Common Language Infrastructure) is defined using standard ECMA-335. All references in this document are obtained from the 4th edition of the standard which can be found in:
http://www.ecma-international.org/publications/standards/Ecma-335.htm

Field Declaration and Definition

Fields are typed memory places which hold data for the application. It is common to distinguish between two types of fields – static and per-instance. Static fields are shared across all instances of the containing object, whereas per-instance, as their name implies, are unique for each instance of the object.
A further distinction is done between Fields and *(Local) Variables*. The difference between the two is their scope – while fields are valid through all the methods of a class (and its descendants – depending on their visibility), a variable is valid only within the containing method.



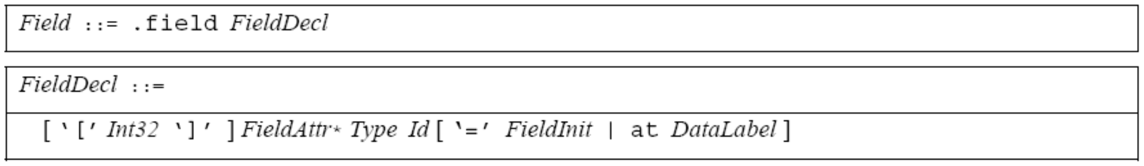
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static int StaticVariable = 1;
        int PerInstanceVarianble = 2;

        static void Main(string[] args)
        {
            int LocalVariable = 3;
            Console.WriteLine("Hello World");
        }
    }
}
```



```
// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloWorld.Program
       extends [mscorlib]System.Object
{
  .field private static int32 StaticVariable
  .field private int32 PerInstanceVarianble
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       15 (0xf)
    .maxstack  1
    .locals init ([0] int32 LocalVariable)
    IL_0000:  nop
    IL_0001:  ldc.i4.3
    IL_0002:  stloc.0
    IL_0003:  ldstr      "Hello World"
    IL_0008:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000d:  nop
    IL_000e:  ret
  } // end of method Program::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       15 (0xf)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  ldc.i4.2
    IL_0002:  stfld      int32 HelloWorld.Program::PerInstanceVarianble
    IL_0007:  ldarg.0
    IL_0008:  call       instance void [mscorlib]System.Object::.ctor()
    IL_000d:  nop
    IL_000e:  ret
  } // end of method Program::.ctor

  .method private hidebysig specialname rtspecialname static
          void  .cctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldc.i4.1
    IL_0001:  stsfld     int32 HelloWorld.Program::StaticVariable
    IL_0006:  ret
  } // end of method Program::.cctor

} // end of class HelloWorld.Program


// =============================================================
```

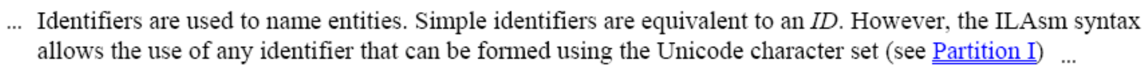
The ECMA standard permits every character that can be represented as a UNICODE character to appear in a variable name (Section 16 Partition II, Section 5.3 Partition II – ECMA-335):

"



```
Field ::= .field FieldDecl

FieldDecl ::=
  [ '[' Int32 ']' ] FieldAttr* Type Id [ '=' FieldInit | at DataLabel ]
```

…



... Identifiers are used to name entities. Simple identifiers are equivalent to an *ID*. However, the ILAsm syntax allows the use of any identifier that can be formed using the Unicode character set (see Partition I) ...

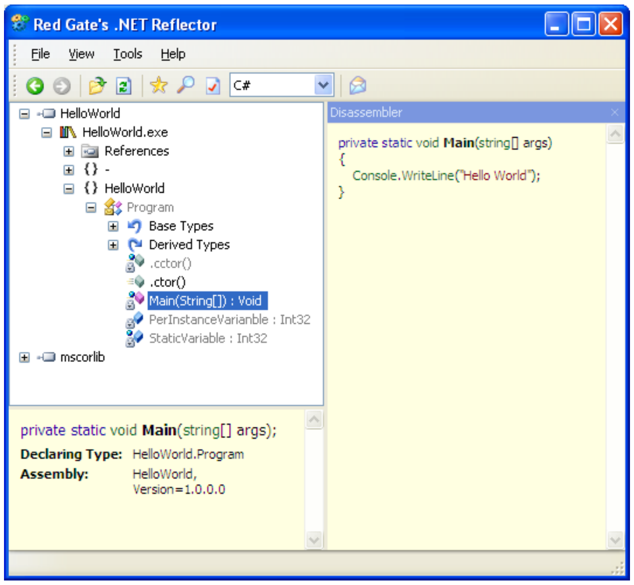
"

.NET Reverse-Engineering

As described above – in the .NET environment (and Java framework as well), source code is transformed into IL at the compilation stage, which in turn is transformed into machine code only at execution time. This is unlike other development languages (such as C and C++) in which the compilation transforms the code directly into machine code representation. Since machine code is platform dependent, it allows .NET assemblies to run on every platform, by only replacing the CLR. To achieve this platform independency the language in which .NET assemblies are presented (The IL), must be of a higher level of abstraction compared to machine code. Consequently it makes IL human readable to a certain extent, enabling rather easy translation from assembly back to the source code. This process is called reverse-engineering, and might reveal proprietary intellectual-property. It may also expose the code to changes and manipulation by rewriting the code and recompiling it. This allows, for example, circumventing access control, authentication or a licensing mechanism.

There are many open-source and commercial tools that aid in the reverse engineering process. Two well-known tools are Remotesoft's Salamander *.NET* Decompiler and Red-Gate's Reflector.

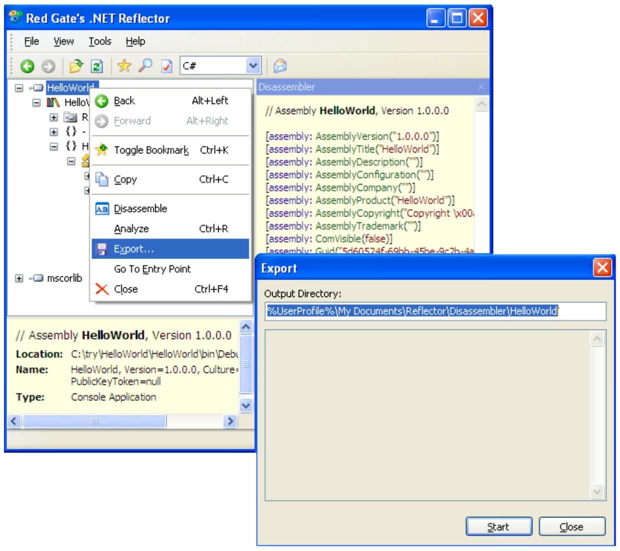
The following example shows the output of Reflector on HelloWorld.exe, the compiled code of the HelloWorld application seen above.



```
private static void Main(string[] args)
{
    Console.WriteLine("Hello World");
}
```

```
private static void Main(string[] args);
Declaring Type:  HelloWorld.Program
Assembly:        HelloWorld,
                 Version=1.0.0.0
```

As can be seen, the entire structure of the application, as well as its full source code, is visible to hackers' eyes. Furthermore, Reflector's "Export" feature creates a full .NET project with all the sources mimicking the project which the original developer had on his computer.



```
// Assembly HelloWorld, Version 1.0.0.0

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyTitle("HelloWorld")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("HelloWorld")]
[assembly: AssemblyCopyright("Copyright \x00
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("5d60524f-69bb-45be-9c2b-4a
```

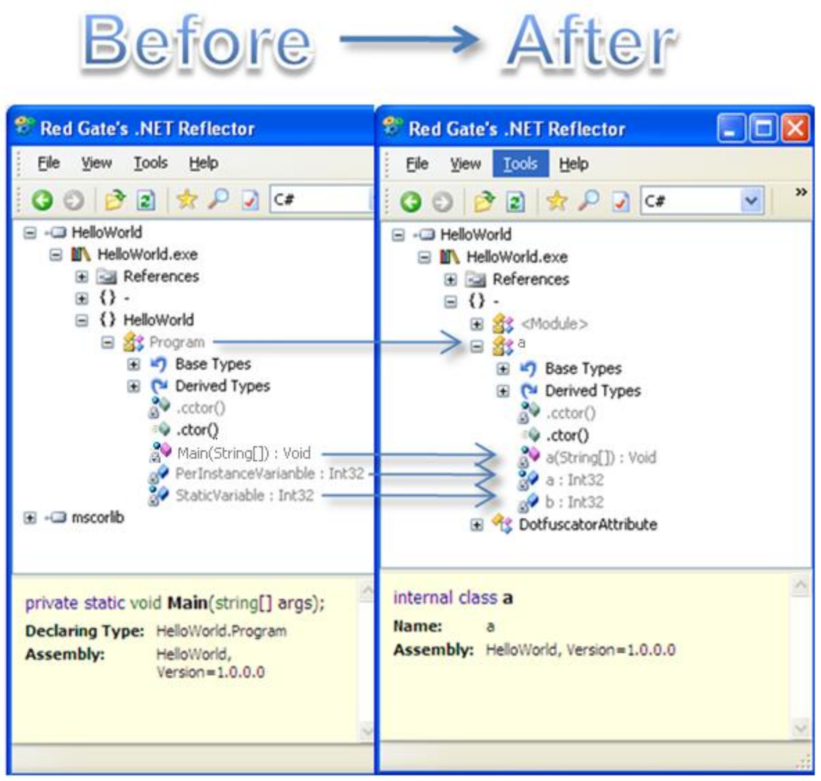
```
// Assembly HelloWorld, Version 1.0.0.0
Location:  C:\try\HelloWorld\HelloWorld\bin\Debu
Name:      HelloWorld, Version=1.0.0.0, Culture=
           PublicKeyToken=null
Type:      Console Application
```

Output Directory: %UserProfile%\My Documents\Reflector\Disassembler\HelloWorld

It is possible to digitally sign an assembly so changes to the binary will be detected by the application, but this does not protect against reverse-engineering the application back into its source form and manipulating it afterwards.

Assembly Obfuscation

In general, the most common way to protect an assembly while maintaining compatibility with the .NET Framework is by implementing a technique called *Obfuscation*. Code obfuscation is the processes of changing an assembly in such a way that it will maintain its functionality while making its content unreadable for the hacker – hence protecting proprietary IP from praying eyes. Code obfuscation is achieved by various techniques – one of which is renaming objects with difficult to read names. For example, see below the result of obfuscating done by PreEmptive Dotfuscator Community Edition.



```
private static void Main(string[] args);
Declaring Type:  HelloWorld.Program
Assembly:        HelloWorld,
                 Version=1.0.0.0
```

```
internal class a
Name:       a
Assembly:   HelloWorld, Version=1.0.0.0
```

This kind of protection is somewhat passive, as it is still possible to export the resulting assembly into its source code, and it will be fully executable. The technique below adds an active protection by enabling the system to identify that it had been exported and the source was revealed.

Decompilation Injection

The motivation behind Decompilation Injection is to allow a reverse-engineered application to identify that indeed it was reverse-engineered and act upon that knowledge.

As described above, the ECMA standard allows *every* character to appear in a field's name - even reserved words and characters can be used. This makes perfect sense, as reserved words are specific to the language used to write the code, but the IL itself should remain agnostic to the language used (For example, in VB.net, the keyword "me" is equivalent to "this" in C#, hence we can create a field called "me" in our C# application. Obviously the .NET framework should support that). The most interesting part is that even reserved characters can be used in the name, such the equal "=" sign and semi-colon ";".

Option 1

The easiest option making use of the ability to change the name of fields with any Unicode character is by invalidating the code. Unlike standard obfuscation, which makes the reading of the source by a human being more difficult by renaming objects to arbitrary names - it is possible to rename objects to *reserved words* ("for", "if", and even "//" for remarks…), which is perfectly valid at the IL level, but completely ruins the source code after decompilation. This method not only makes it difficult for a human hacker to understand the code, but also prevents standard compilers from analyzing the source.

Example 1

By opening the executable file in its binary form, and changing the variable 'abcde' into 'int//', 'b' into '5' and 'ccc' into 'for', we can change the following code:

```
int abcde = 1;
int b = 2;

static void Main(string[] args)
{

      int ccc = a + b;
      if (ccc > 3)
      {
            Console.WriteLine("Hello World");
      }
}
```

Into this one (changes are highlighted) – which is completely useless for standard compilers:

```
int int// = 1;
int 5 = 2;

static void Main(string[] args)
{

      int for = int// + 5;
      if (for > 3)
      {
            Console.WriteLine("Hello World");
      }
}
```

Option 2

The second option for using the Injection technique to protect an assembly from prying eyes not only makes the code useless, but goes the extra step of making it completely different than the actual source code. This means that a hacker will not be able to understand the IP behind the code.

Example 2

Let's assume we have a very sophisticated algorithm which sums all the numbers between 1 and 5.
We expect the result to be 15.
See code below:

```
int b = 5;
void func1()
{
    int d = 30;
    int sum = 0;
    for (int i = 1; i <= b; ++i)
    {
        sum += i;
    }
    Console.WriteLine("{0}", sum);
}
```

Now, after compiling the code, we open the executable in its binary format and rename the variable "d" into "b". This will result with the following code (changes are highlighted):

```
int b = 5;
void func1()
{
    int b = 30;
    int sum = 0;
    for (int i = 1; i <= b; ++i)
    {
        sum += i;
    }
    Console.WriteLine("{0}", sum);
}
```

Since the "new b" is within a more specific scope, it will hide the broader one – and if decompiled and read by a hacker, the application will look like it should return the sum of the numbers 1..30 = 465.
We have successfully managed to change the algorithm into a different one. Even without ever executing it, a hacker will not be able to tell what the correct form of the code is, and will fail to understand the code and the proprietary IP.

Option 3

This option is somewhat similar to the second option, as it changes the code during decompilation, but the motivation here is completely different. In the second option, we changed the code to prevent a hacker from understanding the code. The technique described below enables the system to automatically identify the decompilation attempt during runtime and act upon it.

Example 3

The following code has a very simple functionality – every time button 1 is clicked, a message box with the text "Everything is fine" appears. (FirstVariable is initialized to 1 and never changes afterwards.)

```
public class Form1 : Form
 {
    private Button button1;
    private IContainer components = null;
    private static int FirstVariable = 1;
    private int SecondVariableSomeText;
    public Form1()
    {
       this.InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
       this.SecondVariableSomeText = 2;
       if (FirstVariable == 1)
       {
          MessageBox.Show("Everything is fine");
       }
       else
       {
          MessageBox.Show("Reverse engineering detected");
       }
    }


    ... <snip> ...

}
```

Now, after compiling the code, we can change the name of the variables as we like. Specifically, we can change the name of “SecondVariableSomeText” into “SecondVa=FirstVariable”, which is perfectly acceptable by the IL. Running the executable after renaming keeps the same functionality. However, when reverse-engineering the executable using Reflector, we get the following code:

```
 public class Form1 : Form
  {
     private Button button1;
     private IContainer components = null;
     private static int FirstVariable = 1;
     private int SecondVa=FirstVariable;
     public Form1()
     {
        this.InitializeComponent();
     }

     private void button1_Click(object sender, EventArgs e)
     {
        this.SecondVa=FirstVariable= 2;
        if (FirstVariable == 1)
        {
           MessageBox.Show("Everything is fine");
        }
        else
        {
           MessageBox.Show("Reverse engineering detected");
        }
     }


     ... <snip> ...

 }
```

This brings up the message box “Reverse engineering detected” when running the application, since FirstVariable is assigned a value of 2 (which overwrites the value 1 set during initialization), so the condition returns *false*).

This means that the application has successfully identified being executed after decompilation

.

The executable which can be found here: http://checkmarx.com/Resources/DecompilationInjection.exe is a compiled version of the code above. Running it and clicking on the button should display “Everything is fine”. Decompiling it with Reflector and running through the Visual Studio will change the action of the button to display “Reverse engineering detected”.

Open Issues

The examples discussed in this paper demonstrate attacks to the .NET Reflector. We are confident that the same technique is applicable to Java reverse engineering tools and probably to the .NET ILDASM as well (although it seems to correctly escape Unicode characters, making such an attack more difficult).

We have seen one specific attack based on the ability to insert meta-characters into object names within IL. The ability to conduct different types of attacks should be further researched.

Epilogue

We have demonstrated a technique that takes advantage of the difference between the ECMA standard, which requires encoding Unicode characters within object names, and .NET Reflector which doesn't conform to this standard, thus allowing manipulation of the code retrieved by decompilation.

About

Maty Siman is the Founder and CTO of Checkmarx. He has been active in the IT industry for the past 12 years and has experience in software development, IT security and source-code analysis. Prior to founding Checkmarx, Mr. Siman worked for two years at the Israeli Prime Minister's Office as a senior IT security expert and project manager. Prior to that he spent six years with the Israel Defense Forces (IDF), where he was elected for the STAR excellence program and taught several consecutive sessions of their prestigious application development course (Mamram). He established and led a development team in the Information Security Center (InfoSec) and completed military academy as an IT Security R & D Officer. He regularly speaks at IT security conferences and holds the highly regarded CISSP certification since 2003.

Checkmarx's vision is to provide a comprehensive solution for automated security code review. The company pioneered the concept of a query-language-based solution for identifying technical and logical code vulnerabilities with virtually zero false-positives. The leading provider of the new generation of security software source code analysis solutions, Checkmarx technology is used by developers and R&D departments in large and medium size companies worldwide. The Checkmarx CxSuite facilitates the enforcement of regulatory compliance requirements, including internal company policies throughout the Software Development Life Cycle. Checkmarx has developed a growing network of strategic partnerships with worldwide industry leaders understanding the value of providing integrated, comprehensive solutions and expert support to protect companies from the threats posed by security flaws in software applications. Checkmarx products are deployed with Fortune 1000 and mid-size companies in the Defense, Financial, Telecommunication, Government, and Military industries, as well as to some of the largest Independent Software Vendors. Read more about Checkmarx at www.checkmarx.com.