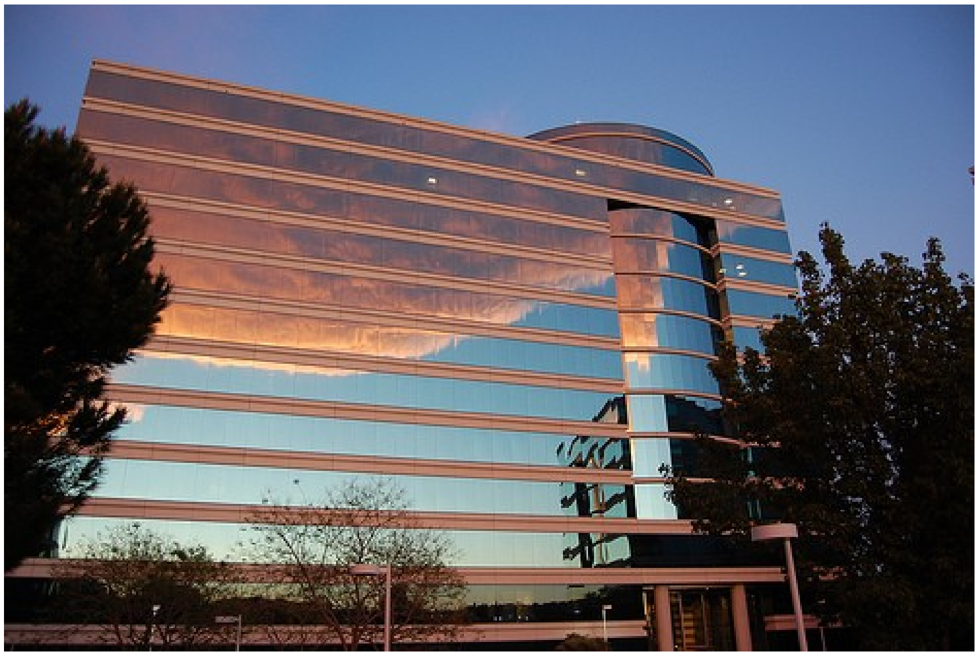
# REFLECTION'S HIDDEN POWER



# "MODIFYING PROGRAMS AT RUN-TIME"

By J~~~~~~~ M~~~~~~
5/27/02

## Contents

Contents ii
Abstract iv
Glossary v
Introduction vi
Implementation of Reflection Manipulation 1
Real Life Usage: Reflection 101 2
Implementing Reflection 4
- Load an Assembly 4
- Getting the Types From an Assembly 4
- Getting and Invoking Constructor of an Object Type 4
- Traversing Instantiated Objects 6
- Invoking Functionality on an Object 6
- Change the Values of an Object 7

The DotNet World: From System Process to Class Level 8
High Level View: What Can Reflections Do and What Is It? 9
How to Navigate to a Specific Object 10
How to Access the Form Object 11
- New Vectors: Access by Reflection 12
- Limitations and Brick walls 12

Demo Attacks 13
- The SQL Injection 13
- The Social Engineering Vector 14

Conclusion 15

## Illustrations

Figure 1 . Code - Common Flags....2

Figure 2 . Code - Build Instance Flag ....2

Figure 3 . Code - Build Static Flag ....2

Figure 4 . Code - Load Assembly ....4

Figure 5 . Code - Gain Access to Types in Assembly....4

Figure 6 . Code - Get and Load Constructor of an Object....5

Figure 7 . Code - Traversing Objects....6

Figure 8 . Code - Invoking Functionality on Objects....6

Figure 9 . Code - Change Values on Objects....7

Figure 10 . Image - System Process Overview....8

Figure 11 . Image - System Process Overview Alternate Implementations ....8

Figure 12 . Code - DotNet Call to Get Forms....11

Figure 13 . Code - System Call to Get Forms....11

Figure 14 . Code - Demo SQL Injection....13

Figure 15 . Code - Demo Social Engineering....14

## Abstract

This paper will demonstrate using Reflection to take control over a DotNet (.Net) compiled code. The focus of this paper will be on how to use Reflection to navigate and gain access to values and functionality that would normally be off limits. This paper will be geared for any DotNet programmer (focus will be in C#). No special knowledge of Reflection is necessary. The basic concept of Reflection and DotNet will be given, along with some light training on using reflection. This paper is written for the DotNet v2.0 and v3.5 versions of DotNet. Examples will be given on attacks, like forcing a program to change values and execute functionality.

## Glossary

|  |  |
|---|---|
| *AppDomain* - | The DotNet AppDomain is equivalent to a process. It provides separation and protection between AppDomains. This is typically a separation between code that has independent execution. Ex. Say 'void Main()' has crashed, the program can still report the problem and try to recover with a secondary AppDomain. |
| *Assembly* - | The DotNet Assemblies contain the definition of types, a manifest, and other meta-data. Standard DotNet Assemblies may or may not be executable; they might exist as the .EXE(Executable) or .DLL(Dynamic-link library). |
| .*GetType*() – | Is a function that is inherited from *Object* in DotNet. It returns the *System.Type* for the object it is called on. |
| UML - | Is an acronym for Unified Modeling Language, it is a graphical language for visualizing, specifying, constructing, and documenting the artifacts in software. |

# Introduction

## Subject

Reflection grants access to a meta level of programs. This paper will look at how to use Reflection to gain access and control over an outside .EXE or .DLL. Reflection makes it possible to gain access to private and protected areas of a program, as well as directly modifying most any variable or trigger functionality.

## Purpose

This paper is a resource for programmers researching Reflection. This report will give the reader a basic concept of Reflection and some example code to play with. It will give an overview of Reflection and in-depth usages in "real world settings."

## Scope

This paper will cover Reflection on managed DotNet specifically at Run-Time.

## Over View

This paper will start off by covering how to use Reflection to manipulate a compiled program. It will cover some basic parts of reflection; some example code will be given. Some supporting background information on DotNet and Reflection will be provided. Afterwards we get to direct attacks opened up by Reflection

## Implementation of Reflection Manipulation

The first step in a Reflection attack is loading the outside codebase. This is done by loading the Assembly from an .EXE or .DLL into an accessible AppDomain, which will grant easy access.

The second step will be finding the object types in the program. This will grant access to launch constructors, access static objects, and invoke static functions.

The third step, depending on the targeted outcome, is to run the program on its normal path. To do this get the common entry point of "void Main()" and invoke it.

The fourth step is gaining access to the part of the program to be controlled. Most of the time this will be an instance object that will need to be found by traversing between instance objects to get a reference. This can be extraordinarily difficult.

The fifth step is impacting changes. This is normally accomplished by setting a value or invoking some specific functionality on an object.

Some optional parts to this sequence are:

- Re-code "void Main()" to take complete control over the program's entry point.
- Load the target compiled code base into a different AppDomain.
- Access the Form object(s) and take over the GUI.

As with any endeavor, knowing the lay of the land is invaluable. After reading over the code base and/or looking at the UML, an experienced programmer should be well equipped to move around inside and control the target program. Also Sequence diagrams can also come in handy as they show a specific execution path.

## Real Life Usage: Reflection 101

One of the basic tasks for Reflection is changing a value on an object. To do this simply do a *GetType* on the target instance object-- access its fields with *GetFields* and do a *SetValue* on the target field.

By doing a *GetType* on an object it will return a *System.Type*. This can then be used to gain access to fields, methods, attributes, and more. The main trick to using these functions and most of Reflection is setting the flags on the requests.

Some flags commonly used are:

```
System.Reflection.BindingFlags.Instance            = retrieve from the instance part of an object
System.Reflection.BindingFlags.Static              = retrieve from the static area of object type
 System.Reflection.BindingFlags.NonPublic          = retrieve non-public items
System.Reflection.BindingFlags.Public              = retrieve public items
System.Reflection.BindingFlags.FlattenHierarchy    = retrieve from derived classes
```

**Figure 1 . Code - Common Flags**

The flags are constructed by an **OR** operation as they are an enumeration type.

This is a demo flag built to access material on an instance object that is public and non-public as well as material from its derived classes. Example below:

```
System.Reflection.BindingFlags flag = System.Reflection.BindingFlags.Instance |
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Public |
System.Reflection.BindingFlags.FlattenHierarchy;
```

**Figure 2 . Code - Build Instance Flag**

This is a demo flag built to access material on a static object that is public and non-public as well as material from its derived classes. Example below:

```
System.Reflection.BindingFlags flag = System.Reflection.BindingFlags.Static |
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Public |
System.Reflection.BindingFlags.FlattenHierarchy;
```

**Figure 3 . Code - Build Static Flag**

To get the Fields for an object it would be:

```
objectIn.GetType().GetFields(flag);
```

To get the Methods for an object it would be:

```
objectIn.GetType().GetMethods(flag);
```

Flags do not cancel each other out, so it is ok to do a request for instance and static or public and non-public on the same flag.

## Implementing Reflection

Some key parts to Reflection are: load an Assembly, getting the types from an Assembly, getting and invoking constructors of an object type, traversing instantiated objects, invoking functionality on an object, and changing the values of an object.

### *Load an Assembly*

```
public static System.Reflection.Assembly LoadAssembly(string filePath)
{
        System.Reflection.Assembly AM = System.Reflection.Assembly.LoadFile(filePath);

        return AM;
}
```

**Figure 4 . Code - Load Assembly**

### *Getting the Types From an Assembly*

```
public static System.Type[] LoadAssembly(System.Reflection.Assembly assemblyIn)
{
        myAssembly = assemblyIn;

        System.Type[]typesInAssembly = null;

        typesInAssembly = GetTypesFromAssembly(myAssembly);
        return typesInAssembly;
}
```

**Figure 5 . Code - Gain Access to Types in Assembly**

### *Getting and Invoking Constructor of an Object Type*

```
public static object LoadObject(System.Type theType)
{
    System.Reflection.ConstructorInfo[] ConstructorList = GetConstructors(theType);

    //pick the default constructor from the list, some times it will be 0
    System.Reflection.ConstructorInfo defaultConstructor = ConstructorList[0];

    return LoadConstructor(defaultConstructor, new object[]{});
}

public static System.Reflection.ConstructorInfo[] GetConstructors(System.Type theType)
{
    return theType.GetConstructors();
}

public static object LoadConstructor(System.Reflection.ConstructorInfo theConstructor, object[] param)
{
    return theConstructor.Invoke(param);
}
```

**Figure 6 . Code - Get and Load Constructor of an Object**

### *Traversing Instantiated Objects*

```
public static object GetSubObject(object objectIN)
{
   System.Reflection.FieldInfo[] fields = ReflectionPower.GetFields(objectIN, true);

   // select fields[0], most of the time you will not pick [0]
   System.Reflection.FieldInfo field = fields[0];

   // return the value object for the field
   return field.GetValue(objectIN);
}

public static System.Reflection.FieldInfo[] GetFields(object objectIn, bool ShowPrivate)
{
      System.Reflection.BindingFlags flag = System.Reflection.BindingFlags.Instance |
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Public;

   if (ShowPrivate)
      return objectIn.GetType().GetFields(flag);
   else
      return objectIn.GetType().GetFields();
}
```

**Figure 7 . Code - Traversing Objects**

### *Invoking Functionality on an Object*

```
public static object CallFunctionalityOnObject(object objectIN)
{
   System.Reflection. MethodInfo [] methods = ReflectionPower. GetMethods (objectIN, true);

   // select methods[0], most of the time you will not pick [0]
   System.Reflection. MethodInfo method = methods [0];

   // This the a list of parameters to pass into the function
   object[] params = new object[]{};

   // pick the method to Invoke, pass the object to Invoke it on, pass the parameters
   return LoadMethodStatic (method , objectIN, params);
}

public static System.Reflection.MethodInfo[] GetMethods(object objectIn)
{
    return objectIn.GetType().GetMethods();
}

public static object LoadMethodStatic(System.Reflection.MethodInfo methodIN, object objectIn, object[] param)
{
    return methodIN.Invoke(objectIn, param);
}
```

**Figure 8 . Code - Invoking Functionality on Objects**

***Change the Values of an Object***

```
public static void ChangeSomeValue(object objectIN, object valueIN)
{
   System.Reflection.FieldInfo[] fields = GetFields(objectIN, true);

   // pick the field you wish to change
   System.Reflection.FieldInfo aField = fields[0];

   aField.SetValue(objectIN, valueIN);
}

public static System.Reflection.MemberInfo[] GetFields(object objectIn, bool ShowPrivate)
{
      System.Reflection.BindingFlags flag = System.Reflection.BindingFlags.Instance |
System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Public;

      if (ShowPrivate)
          return objectIn.GetType().GetFields(flag);
       else
          return objectIn.GetType().GetFields();
}
```

**Figure 9 .   Code  - Change Values on Objects**

## The DotNet World: From System Process to Class Level

The AppDomain is the main boundary in DotNet. A normal DotNet process contains an AppDomain. Inside of an AppDomain lives Assemblies-- an Assembly is a complete code base and resource structure. Inside of an Assembly is where Classes and NameSpaces exist along with most other features that make up a program.

**Diagram of a System Process with an AppDomain, Assembly, and Class:**

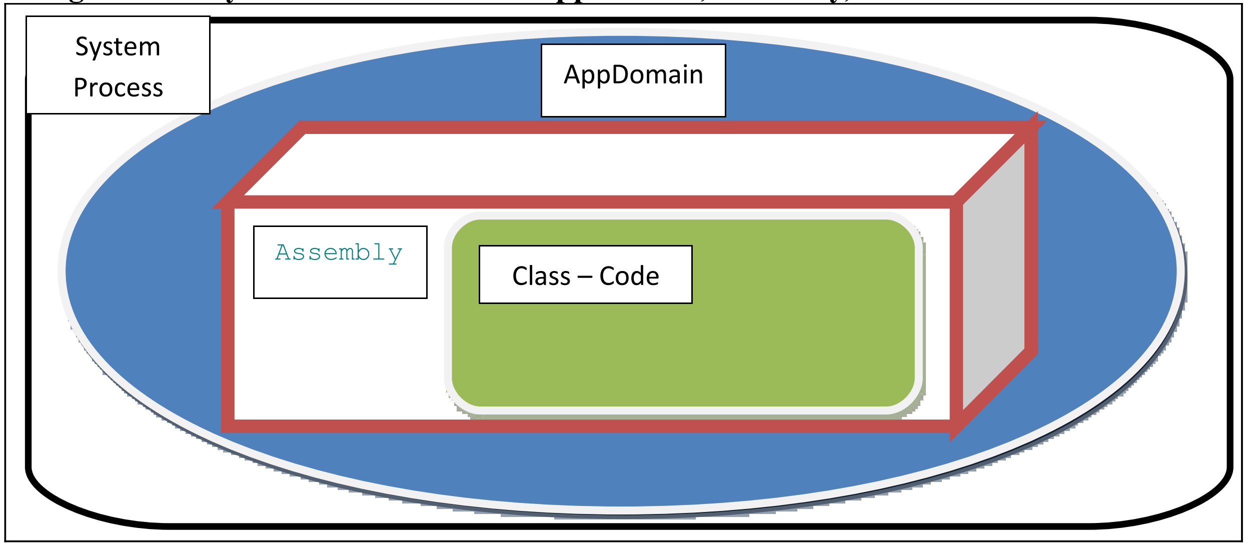


Figure 10 . Image - System Process Overview

AppDomains are self contained. They can crash and not take down the process they live in or a neighboring AppDomain. AppDomains are the main place DotNet segments memory. The way this was implemented is similar to how operating systems segment memory for processes.

More than one AppDomain can be loaded into a process and more than one Assembly can be loaded into an AppDomain. Once an Assembly is loaded it cannot be unloaded except by unloading the AppDomain it is in. Cross Appdomain memory access is limited by DotNet.

**Some Alternate Diagrams of Loading AppDomains and Assemblies:**

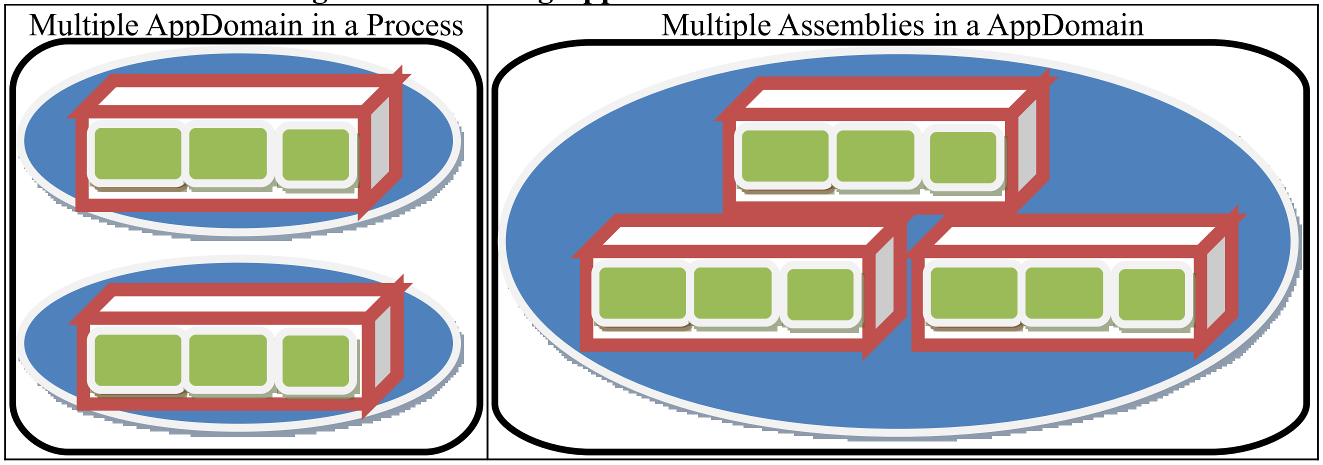

Multiple AppDomain in a Process
Multiple Assemblies in a AppDomain

Figure 11 . Image - System Process Overview Alternate Implementations

## High Level View: What Can Reflections Do and What Is It?

Reflection can impact code by opening an object or code base and giving access to its values and functionality. This can allow a programmer to interact with compiled programs, in order to cause the target program to act in different ways, such as sending commands to the Database that should not be sent or adding an interface to help blind people use the program. The power of reflection can force a program to interface, to give up or change its information, or to activate its functionality.

Reflection can impact the target program solely in memory. This allows for control over the target program with a minimal footprint on the target program. Also in the DotNet framework Reflection is directly under the *System* NameSpace, so it should be in every project by default.

## How to Navigate to a Specific Object

The object-web, formed by a program at run time can make even the craziest UML look tame. With Reflection we navigate the tangled-web of objects and gain the ability to make change. Having access to the decompiled code base is not necessary, but a map always helps. The decompiled code base can help in developing a path out to the target object or in finding chinks to help gain references deeper into the target program.

After a program is loaded and Reflection has access, the best place to start is by getting a form object and working back from that. Another possibility is working back from a static object. Events and Delegates can also be valuable in this endeavor. Events and Delegates can be modified to lay traps that can gain a reference to an object as it fires an Event or Delegate. Also it is possible to look at what is hooked to an Event or targeted by Delegates to gain information from that.

If the program is nice enough to grant one, a normal API can also be a place to hitch into the program. This will help to quickly get deep into the programs instance object structure.

Every program is different so no one approach is best, some programs will be easy to infiltrate and others difficult. Regardless of how it is done, with some skill or luck, once the target object is found it should be easy to impact the desired changes or access needed information.

## How to Access the Form Object

Two easy ways to get form objects is with a DotNet call or a system call. The DotNet OpenForms call returns a formCollection. With the windows system call it returns window handles. Note that the window handles can reference forms that cannot be accessed.

### DotNet Call to System.Windows.Forms.Application.OpenForms:

```
Public System.Windows.Forms.Control[] GetWindowList()
{
   System.Collections.Generic.List<System.Windows.Forms.Form> formList = new List<System.Windows.Forms.Form>();

   foreach (System.Windows.Forms.Form f in System.Windows.Forms.Application.OpenForms)
   {
      formList.Add(f);
   }

   return formList.ToArray();
}
```

Figure 12 . Code - DotNet Call to Get Forms

### Windows System Call to “user32.dll”->EnumWindows:

```
[System.Runtime.InteropServices.DllImport ("user32.dll")]
private static extern int EnumWindows(EnumWindowsProc ewp, int lParam);

[System.Runtime.InteropServices.DllImport ("user32.dll")]
private static extern bool IsWindowVisible(int hWnd);

//delegate used for EnumWindows() callback function
delegate bool EnumWindowsProc(int hWnd, int lParam);

public static System.Windows.Forms.Control[] myWindows()
{
    System.Collections.Generic.List<System.Windows.Forms.Control> WList;

    WList = new System.Collections.Generic.List<System.Windows.Forms.Control>();

    // Declare a callback delegate for EnumWindows() API call
    EnumWindowsProc ewp = new EnumWindowsProc(delegate(int hWnd, int lParam)
    {
        System.Windows.Forms.Control aForm;

        aForm = System.Windows.Forms.Form.FromChildHandle((IntPtr)hWnd) as System.Windows.Forms.Control;

        // Check if form object is not null
        if (f != null)
            WList.Add(f);

        return (true);
    });

    // Call DllImport("user32.dll") to Enumerate all Windows
    EnumWindows(ewp, 0);

    // Send Forms back
    return WList.ToArray();
}
```

Figure 13 . Code - System Call to Get Forms

## The New Rules Under Reflection

### *New Vectors: Access by Reflection*

The attack vector opened by Reflection is at Run-Time. With Reflection it is possible to delve into a code base and Run "void Main()" or drop down into its class structure and create a single object to wield as *you* wish.

Since Reflection is not decompiling, it can have a more automated and faster integration time with the target code base along with less of a foot print.

Reflection can easily add functionality to a preexisting code base. No longer do programs have to be written with extensibility in mind or accessible technology to integrate.

Because Reflection does not impact the code base it can get past CRC checks and code signing.

### *Limitations and Brick walls*

Some road blocks to using Reflection are:  It is necessary for the target to be a DotNet application. Reflection also is limited by memory access rights imposed by the operating system. Objects need to have a proper reference to be accessed. Programs can also be constructed with countermeasures that could be triggered if they detect an intrusion.

Once access to the code base is gained with Reflection getting to the target object maybe harder than one might think; because normally we are the programs designer easily keeping references to important objects, but as we are coming into another programmer's world with Reflection manipulation we have to find each object by hand.

## Demo Attacks

Reflection can augment some of the old attacks with new powers. I will demonstrate the attacks of a SQL Injection and Social Engineering.

### *The SQL Injection*

SQL Injection, is sending commands to a DB that should not be sent. This SQL Injection will be on a client side app, the app would normally sanitize the commands before they are sent. Normally with Reflection we would not need to do SQL Injection at all, as we could send any SQL we wish; but for the sake of this demo we will disable the SQL Injection cleaning mechanism, and make it vulnerable to SQL Injection.

**Demo - SQL Injection**

| |
|---|
| Load target code |
| Find SQL Object |
| Find SQL cleaning thing |
| Disable SQL cleaning |

**Figure 14 . Code - Demo SQL Injection**

### *The Social Engineering Vector*

Users currently do not expect a client side app to lie to them and trick them into divulging critical information. After taking over a program this attack will pop-up a fake window and lock the program until the user enters critical data. Preferably this would be best done at a logical choke point in the program such as on file access or DB connection.

## Demo - Social Engineering

“Load target code”

```
System.Reflection.Assembly AM = System.Reflection.Assembly.LoadFile(fileOn);
AM.ModuleResolve += new System.Reflection.ModuleResolveEventHandler(AM_ModuleResolve);
System.AppDomain.CurrentDomain.AssemblyResolve += new ResolveEventHandler(CurrentDomain_AssemblyResolve);
```

Find an object in a critical area

“Find an event to add password request”

```
System.Reflection.FieldInfo FI = xObject.GetFields(objIN, true);

string targetName = "saveFileEvent";
System.Reflection.FieldInfo targetField;
foreach (System.Reflection.FieldInfo FI in aReflectorPower.GetFields(objIN, true))
{
   if (targetField.Name == targetName)
   {
      targetField = FI;
      break;
   }
}
```

Make a copy of the event targets

Put call to fake password Form in event

Put the copied event targets back in the event

**Figure 15 . Code - Demo Social Engineering**

## Conclusion

With Reflection we have the potential to take control of a program and enact changes that are outside of the original creator's scope. Reflection can be used for good or evil by granting flexibility and adaptability, however it is used it opens previously closed doors. No longer are programmers subservient to programs if they can reach in and manipulate objects using Reflection.

Reflection is simple with few requirements and has a small footprint. This makes it a good choice for small changes to a compiled program.

Suggested Additional readings:

White Paper: Advanced Programming Language Features for Executable Design Patterns "Better Patterns Through Reflection"

ftp://publications.ai.mit.edu/ai-publications/2002/AIM-2002-005.pdf

White Paper: An Introduction to Reflection-Oriented Programming

http://www.cs.indiana.edu/~jsobel/rop.html