Reverse code engineering of .NET applications

Shukhrat Nekbaev

Supervised by
Dr. Simo Juvaste

Master's thesis.
August 12, 2013
School of Computing
University of Eastern Finland, Joensuu, Finland.

Acknowledgements

First of all I would like to thank my family for all their support and understanding. Special thanks to Jussi Parkkinen and everyone related to IMPIT program for the opportunity to study at the UEF.

In particular, I would like to thank my thesis supervisor Dr. Simo Juvaste for the fantastic “Parallel programming” course whose mind-intensive homeworks caused sleepless nights for many students and, without a doubt, it was worth it.

Special thanks to Daniel Pistelli[1], an author of several articles and tools used in this research, who kindly agreed to proofread it.

Thanks to all my friends for their valuable comments and for just being around when needed. Last but not least, huge thanks to George Carlin[2], a brilliant stand-up comedian who has influenced my view of life.

[1] http://www.ntcore.com

[2] http://en.wikipedia.org/wiki/George_Carlin

Abstract

This work is a research into the reverse code engineering of .NET applications in which the operational principles of the .NET Framework are analyzed. It is based on studying the changes introduced to the structure of the Windows Portable Executable file format (PE) in order to accommodate .NET specific requirements. Moreover, the thesis presents information about the .NET application execution process and its runtime handling, and points out the differences in the protection schemes between unmanaged and managed applications.

The study provides insight into some of the most popular .NET application reverse engineering methods and common protection schemes applicable to the .NET. A concrete commercial application was selected for the conclusive part of the practical chapter. This is due it being protected by an interesting, yet powerful and heavily obfuscated virtual machine based code protector. What's more, no information on its analysis was available on the Internet at the time of writing.

Furthermore, an example of an attack during runtime is shown and the potential benefits of such an attack are evaluated. Lastly, the advantages of attacking the .NET Framework itself are considered.

Keywords: reverse code engineering, .NET Framework, protection analysis, disassembly, decompilation, static analysis, debugging, code protection, intellectual property

DISCLAIMER

THE INFORMATION PROVIDED IN THIS WORK IS FOR EDUCATIONAL PURPOSES ONLY. THE USE OF THIS MATERIAL FOR ANY AND ALL ILLEGAL PURPOSES IS STRICTLY PROHIBITED AND THE MATERIAL ITSELF IS NOT BE TREATED AS A SOURCE OF INSPIRATION FOR SUCH ACTIVITIES. IT CANNOT BE USED TO INITIATE LEGAL PROCEEDINGS AGAINST THE AUTHOR HIMSELF AND/OR AUTHORS REFERENCED THROUGHOUT THE MATERIAL. THE READER HEREBY AGREES TO HAVE A NEUTRAL OPINION ON THE SUBJECT.

Contents

List of Figures

List of abbreviations

.NET	.NET Framework
BCL	Base Class Library
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
COM	Component Object Model
CTS	Common Type System
DLL	Dynamic link library
FCL	Framework Class Library
IAT	Import Address Table
IIS	Internet Information Services
INT	Import Name Table
IP	Intellectual property
JVM	Java Virtual Machine
OS	Operating system
PE	Portable executable
RCE	Reverse code engineering

RVA	Relative Virtual Address
VA	Virtual Address
VES	Virtual Execution System

Introduction

It has been over a decade since the introduction and the standardization of the .NET Framework. Over the years, the framework has dramatically evolved, becoming rich in features as well as a popular development platform among developers. Many companies have developed numerous applications, components, integrated development environments, compilers and even open source implementations. Nevertheless, since the platform introduction, a major concern of resilience to reverse code engineering (RCE) remained. Quite a few solutions followed up. For several years these solutions seemed satisfactory. However, with the growing popularity of the platform, RCE community finally turned its attention to it, and anticipated countermeasures appeared.

Following the quote of Stanley Lieber[1]:

"With great power there must also come - great responsibility!"

a similar quote can be formulated as applied to software:

"With great popularity of an application there must also come - great concern regarding protection of intellectual property".

However, one might wonder: "Why just a concern, but not bullet-proof protection?" The answer can be found within the chapters of this thesis.

1.1 The topic

Reverse code engineering (RCE) can be simply considered a process of obtaining the knowledge about the internal working principles of the software without having access to its source code. This subject is often neglected in academic institutions or mentioned shallowly [Pot05]. Debates on the topic raise ethical concerns, though its importance and existence cannot be underestimated or ignored. The RCE process

[1] better known as Stan Lee. A famous American comic book writer

provides numerous opportunities, such as but not limited to: bridge interoperability gaps between systems, performing security analysis [Mor08] and development of countermeasures, detection of software bugs, analysis of software behavior, and development of alternative software using Cleanroom [LT13] software development process. On the other hand, this offers the opportunity to circumvent copy protection (prevention) schema in the original software, create malicious software, disclose valuable algorithms, and create an almost identical replica or utilize vulnerabilities for potentially illegal activity. RCE use cases are not limited to the aforementioned options, but can be further used in conjunction with other methods. Despite the fact that the purpose of a concrete RCE process is solely related to its goals, an important matter of intellectual property (IP) protection remains.

RCE, apart from being an art, is a chess-like game between the person who has written the code and the person who wants to understand how it works from its binary representation. The internal structure of the .NET application, the execution process and the RCE specific to the .NET are explained in the following chapters.

1.2 Motivation behind the topic and thesis problem statement

I can honestly say that this thesis was motivated by curiosity. I have developed numerous web applications and free software utilities throughout my .NET career. IP protection was not really a concern because the former were deployed to internal corporate servers, thus protected by an enterprise firewall[2] and the latter were open source. However, sooner or later, all developers face the question of code protection regardless of the motivation behind it. Preliminary research on the subject yields not only some general recommendations, but also commercial and free software protection solutions.

From the developer's perspective I want to understand the principles of .NET application protection and methods to work around them. From my point of view such a dualistic approach provides valuable insight to the .NET internals, promotes better understanding and improves programming expertise regarding code protection. It is obvious that this knowledge is crucial for anyone doing binary analysis, e.g. experts working in antivirus or security companies. Malicious software written in

[2] this is however not an excuse not to consider the protection

managed code is a reality too.

Generally, I consider this work applicable not only to the .NET Framework, but also to any other intermediate language based frameworks, including Java software platform, because their internal implementation and operational principles are almost identical.

1.3 Scope of the thesis

This thesis focuses on RCE analysis of existing code protection solutions and copy prevention mechanisms for .NET rich client applications [Mic13o]. This application type represents a stand-alone client application which is usually installed on the end users' computers, but can also be executed over the network share. Nevertheless, the presented RCE methods can also be applied to other .NET application types with regard to specifics of a concrete application type and its execution environment, e.g. a web application compiled as a dynamic link library (DLL) and hosted by Internet Information Services (IIS) application.
The following areas of interest are researched:

- resilience of .NET applications to RCE
- .NET application internal structure and its difference from unmanaged application
- execution process of a .NET application and .NET runtime
- existing code protection solutions for .NET applications, their strengths and weaknesses
- hands-on RCE examples with sample applications
- RCE of a protected commercial application
- recommendations for developers for improving the resilience of the software

The Windows operating system has been used as a host platform for .NET Framework. In order to support .NET Framework executables, modifications were introduced to the Windows portable executable (PE) file structure. These changes are explained later.

It should be explicitly mentioned that existing RCE methods are not limited to the ones discussed in this thesis, because each concrete task requires an individual approach. Based on personal experience and the tools available, a reverse engineer selects suitable methods and adapts them accordingly in order to solve a particular case. Furthermore, background information is only presented in a scope relevant to the topic of the thesis and to sample applications used for analysis. The reader is recommended to visit the referenced sources in order to better understand the material.

1.4 Structure of the thesis

Chapter 2 provides background information about the origin of the .NET Framework, the typical .NET application structure in addition to its compilation and execution process details.

Chapter 3 is divided into two parts. The first part describes the most notable RCE methods applicable to .NET with practical examples, whereas the second part is dedicated to the analysis of a commercial protector. Therefore, a practical example of a software application protected by it is presented.

Chapter 4 is related to attacking .NET applications at runtime, benefits of modification of .NET Framework core libraries and power of the .NET reflection mechanism.

The closing Chapter 5 summarizes the results of the performed research, outlines the most interesting aspects and proposes ideas relevant for future work.

Chapter 2

Background

The .NET Framework is a Microsoft-developed software framework supporting development and execution of applications that leverage code access security, language interoperability, automatic memory management, runtime security, simplified deployments and version conflict resolution, massive class library, etc. [Mic13j]. At the moment of writing the thesis, Microsoft officially supported and developed the .NET Framework only for its proprietary operating systems[1] despite the fact that the framework was designed to be platform independent. Additionally, other implementations of the underlying standard exist. For example, the Mono project is a well-known open source cross platform implementation of the .NET Framework [Xam13].

2.1 Java exists already, why use .NET?

The .NET Framework shares a lot of similarities [Far13] with the Java software platform that has been developed by Sun Microsystems[2]. To understand the reason we have to look back at the second half of the 1990s when Java was introduced in 1996 and rapidly became one of the most popular programming languages. Its platform featured automatic memory management, rich class library, portability, security, and enabled the possibility to easily develop functional desktop applications and more interactive web sites. Microsoft joined the trend in 1996 and licensed Java, but soon was sued by Sun Microsystems for licensing agreement violations [CNE13]. Besides millions of dollars lost, Microsoft found itself in an unfavorable position, and decided to abandon Java under the circumstances. Nevertheless, there was a desperate need

[1] mainly Windows family, but their Xbox console runs its own OS[sha13]

[2] was acquired by Oracle Corporation in 2010

for Microsoft to come up with a solution that could compete with the Java platform. The .NET Framework was introduced several years later, yet it took an additional couple of years until its official release in early 2002.

At the moment of introduction there was a confusion regarding what the .NET Framework actually was and how it compared to the Java platform. One of the early pre-release attempts was presented by Jim Farley [Far13]. He compared the underlying concept of the .NET Framework with the J2EE side by side and confirmed that both platforms were very much alike. It is certain that for this thesis the most important .NET design feature is the code compilation and execution model. Just as Java code is compiled into bytecode which is then executed by the Java Virtual Machine (JVM), most CLI compilers[3] generate a platform and source language independent .NET output called the Common Intermediate Language (CIL) executed by the Common Language Runtime (CLR). Because of technical accuracy it is worth mentioning that the CLI compiler outputs the CIL and relevant metadata in a single unit called a *module* which in turn is enclosed into another logical container called an *assembly* because the CLR can only operate on *assemblies*. This is discussed in more detail later on in the thesis.

2.2 .NET Framework structure

The .NET Framework consists of two major components:

- Common Language Runtime (CLR)
- Framework Class Library (FCL)[4]

According to the Microsoft .NET Framework Conceptual overview [Mic13h] the CLR is a foundation of the .NET Framework that handles code execution, provides supportive core functionality, and enforces both type safety and code accuracy promoting robustness and security. It is designed to provide a simplified, fast and feature rich programming infrastructure that automatically handles most of the lower level APIs, e.g. communication, memory management, I/O operations etc. [Fra03]. On the other hand the FCL is a comprehensive collection of reusable object-oriented types that the developer can utilize for a variety of .NET application types [Mic13h].

[3] some CLI compilers are capable of outputting the CIL and native instructions

[4] the FCL includes the Base Class Library (BCL) that contains fundamental types and core functionality

The CLR is Microsoft's implementation of the Common Language Infrastructure (CLI). Microsoft developed the CLI specification with its partners and it was standardized by ISO and ECMA. ECMA-335 describes the standard as follows:

"This Standard defines the Common Language Infrastructure (CLI) in which applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments" [Mic13n]

Partition I "Concepts and Architecture" of the standard describes the overall architecture of the CLI, Common Type System (CTS), Virtual Execution System (VES), Common Language Specification (CLS) and metadata.

The description of the VES in Partition I and the metadata in Partition II represent particular interest for RCE purposes along with Partition III of the standard that provides details on the CIL instruction set. For that reason ECMA-335 is frequently referenced throughout the thesis. Also, aside from being a valuable source of knowledge for RCE, it is highly recommended reading material for all .NET/CLI developers.

In order to understand the .NET Framework, H. Gilbert [Gil13] suggested to compare it to the OS. The OS takes advantage of x86 hardware level security capabilities to isolate kernel space from user space (privilege rings, segmentation). Processes are separated into their own address spaces while the OS manages the memory for multiple processes running in it, allocates and cleans up requested resources, and handles and recovers from errors. Similarly, the .NET Framework manages the execution of its applications by adding an additional level of security and integrity on top of the one provided by the OS.

Figure 2.1 illustrates a high-level overview of .NET Framework. The left hemisphere illustrates the relationship between the .NET application and the OS. The .NET application can use the FCL (at least its BCL subset because it encapsulates core types and core functionality) and third-party libraries (.NET or unmanaged). Execution of the .NET application is handled by the CLR and eventually only the runtime communicates with the OS directly. The same concept is applicable to .NET web applications: they are also executed by the CLR but inside an unmanaged pipeline via the CLR hosting extensibility. Unmanaged applications[5] communicate with the OS directly via APIs.

[5] a.k.a. native code applications not supervised by the CLR

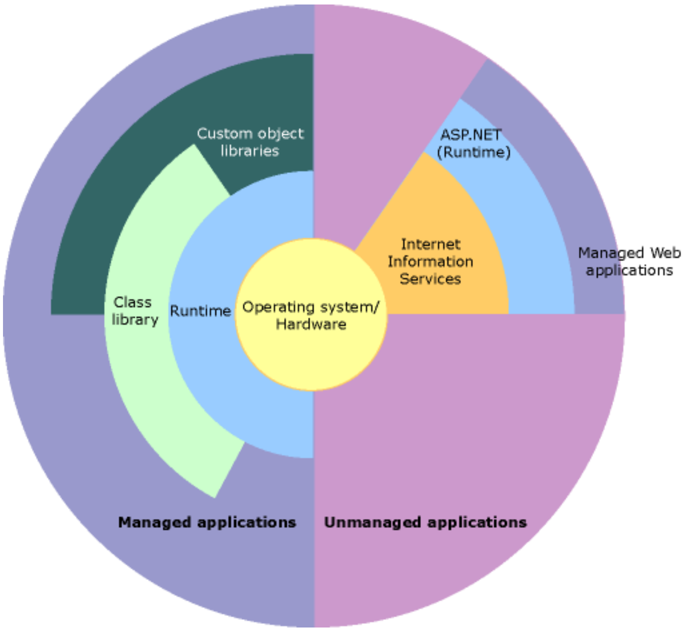


Figure 2.1: .NET Framework overview [Mic13h]

2.3 From source code to executable binary

It is established that any application is developed using a specific programming language and must afterwards be converted into either a representation natively understood by the target execution platform, or into an intermediate language that will be interpreted and executed by a native host application running on the target execution platform. This is called compilation. If an application must be executed natively by the target platform, then a conversion to an executable is performed in two main steps. First, the compilation process produces object files, which are then processed by the linker application. Afterwards, the linker processes the object files, analyses their structure and groups entities into sections by logical similarity, performs additional target platform specific operations, and outputs a single executable file. A prime example of such an executable is the Microsoft's proprietary PE file format described by Microsoft [Mic13f]. Despite Microsoft providing an official specification document [Mic13f], a note on the first page states that it is provided to aid the development of tools and applications for Windows products, and is not guaranteed to be a complete specification. It can change without notice. PE is often referred as PE32 for 32-bit systems and PE32+ for 64-bit systems. Figure 2.2 illustrates the PE file structure.

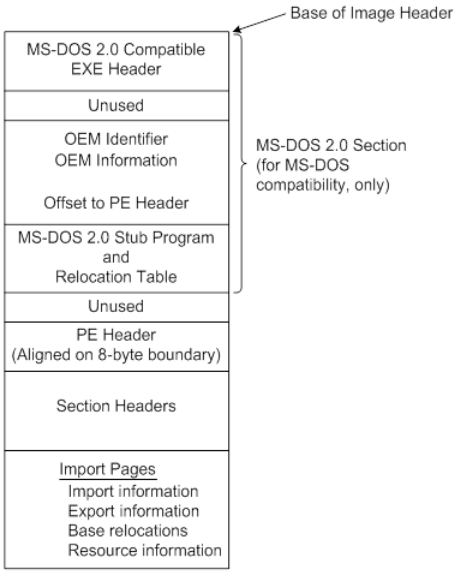


Figure 2.2: The high-level overview of a PE [Mic13f]

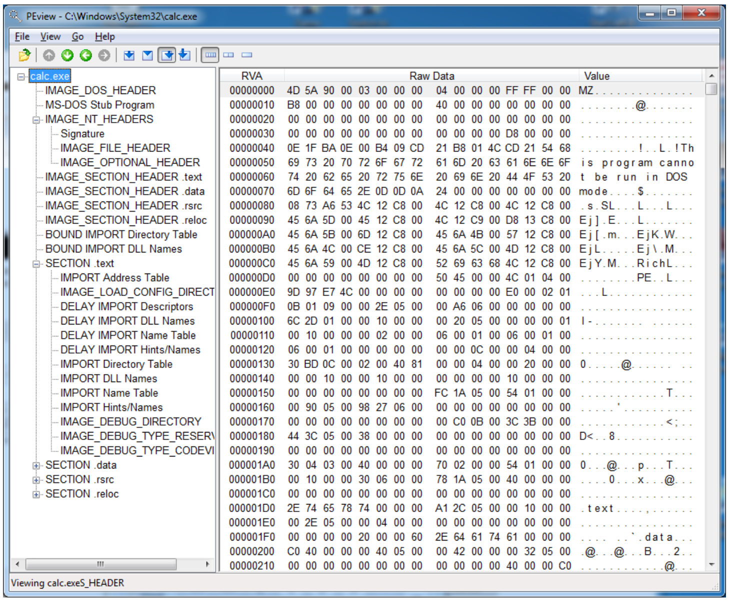
Figure 2.3 illustrates a concrete example of the PE file structural layout:

2.3.1 .NET Assembly

When the .NET Framework application is compiled in Visual Studio, a PE file (EXE or DLL) is produced. It is called an *assembly*[6] in the terms of .NET. Note that the terms *assembly*, *.NET file* and *managed code applications* are used interchangeably throughout the thesis. Some compilers such as Microsoft's C++/CLI are capable of mixed assembly compilation. The mixed assembly contains both CIL and native code [Mic13g]

Assembly is a self-describing logical container and a fundamental unit of physical code grouping [Hew10] [Gun13]. The PE file format was extended to accommodate the new structure. Partition II of ECMA-335 [Mic13n] provides a detailed reference of the CLI file format.

[6] not to be confused with Assembly language

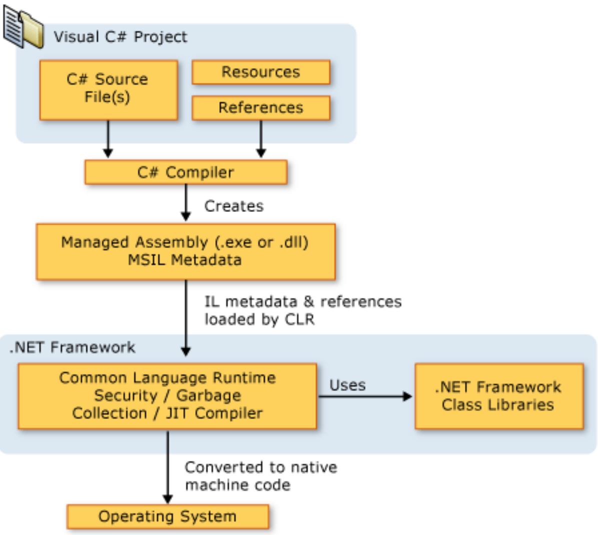


```
calc.exe
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
    Signature
    IMAGE_FILE_HEADER
    IMAGE_OPTIONAL_HEADER
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .reloc
  BOUND IMPORT Directory Table
  BOUND IMPORT DLL Names
  SECTION .text
    IMPORT Address Table
    IMAGE_LOAD_CONFIG_DIRECT
    DELAY IMPORT Descriptors
    DELAY IMPORT DLL Names
    DELAY IMPORT Name Table
    DELAY IMPORT Hints/Names
    IMPORT Directory Table
    IMPORT DLL Names
    IMPORT Name Table
    IMPORT Hints/Names
    IMAGE_DEBUG_DIRECTORY
    IMAGE_DEBUG_TYPE_RESERV
    IMAGE_DEBUG_TYPE_CODEVI
  SECTION .data
  SECTION .rsrc
  SECTION .reloc
```

```
RVA       Raw Data                                          Value
00000000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ..............
00000010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00  00 00 00 00 D8 00 00 00  ................
00000040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ........!..L.!Th
00000050  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode....$.......
00000080  08 73 A6 53 4C 12 C8 00  4C 12 C8 00 4C 12 C8 00  .s.SL...L...L...
00000090  45 6A 5D 00 45 12 C8 00  4C 12 C9 00 D8 13 C8 00  Ej].E...L.......
000000A0  45 6A 5B 00 6D 12 C8 00  45 6A 4B 00 57 12 C8 00  Ej[.m...EjK.W...
000000B0  45 6A 4C 00 CE 12 C8 00  45 6A 5C 00 4D 12 C8 00  EjL.....Ej\.M...
000000C0  45 6A 59 00 4D 12 C8 00  52 69 63 68 4C 12 C8 00  EjY.M...RichL...
000000D0  00 00 00 00 00 00 00 00  50 45 00 00 4C 01 04 00  ........PE..L...
000000E0  9D 97 E7 4C 00 00 00 00  00 00 00 00 E0 00 02 01  ...L............
000000F0  0B 01 09 00 00 2E 05 00  00 A6 06 00 00 00 00 00  ................
00000100  6C 2D 01 00 00 10 00 00  00 20 05 00 00 00 00 01  l-..............
00000110  00 10 00 00 00 02 00 00  06 00 01 00 06 00 01 00  ................
00000120  06 00 01 00 00 00 00 00  00 00 0C 00 00 04 00 00  ................
00000130  30 BD 0C 00 02 00 40 81  00 00 04 00 00 20 00 00  0.....@.........
00000140  00 00 10 00 00 10 00 00  00 00 00 00 10 00 00 00  ................
00000150  00 00 00 00 00 00 00 00  FC 1A 05 00 54 01 00 00  ............T...
00000160  00 90 05 00 98 27 06 00  00 00 00 00 00 00 00 00  .....'..........
00000170  00 00 00 00 00 00 00 00  00 C0 0B 00 3C 3B 00 00  ............<;..
00000180  44 3C 05 00 38 00 00 00  00 00 00 00 00 00 00 00  D<..8...........
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001A0  30 04 03 00 40 00 00 00  70 02 00 00 54 01 00 00  0...@...p...T...
000001B0  00 10 00 00 30 06 00 00  78 1A 05 00 40 00 00 00  ....0...x...@...
000001C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000001D0  2E 74 65 78 74 00 00 00  A1 2C 05 00 00 10 00 00  .text.....,.....
000001E0  00 2E 05 00 00 04 00 00  00 00 00 00 00 00 00 00  ................
000001F0  00 00 00 00 20 00 00 60  2E 64 61 74 61 00 00 00  .......`.data...
00000200  C0 40 00 00 00 40 05 00  00 42 00 00 00 32 05 00  .@...@...B...2..
00000210  00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 C0  ............@...
```

Viewing calc.exeS_HEADER

Figure 2.3: calc.exe opened in PEView. PE loader reads this structure, processes and loads it into memory for execution.



Visual C# Project
C# Source File(s)
Resources
References
C# Compiler
Creates
Managed Assembly (.exe or .dll) MSIL Metadata
IL metadata & references loaded by CLR
.NET Framework
Common Language Runtime Security / Garbage Collection / JIT Compiler
Uses
.NET Framework Class Libraries
Converted to native machine code
Operating System

Figure 2.4: The compilation and execution process of the assembly [Mic13e]

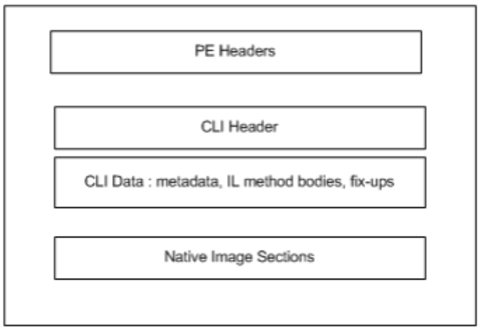


Figure 2.5: A high-level overview of the extended PE [Mic13n].

2.3.2 Physical layout of internal structures

Figure 2.5 represents a typical .NET file structure. The purpose of native sections is explained in section 2.4.

PE Headers are the same for unmanaged applications and most information is irrelevant for the CLR. An entry in Optional Header's Data Directories list (ordinal number 15 or 14 if zero-based) contains a relative virtual address (RVA) to the CLI header and its size. This entry is defined in WinNT.h (Microsoft SDK) as

```
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

and is used for .NET purposes despite its name. The Microsoft PE Specification describes it as the CLR Runtime Header [Mic13f]. The actual CLI header is defined in WinNT.h[7][8]:

```
// CLR 2.0 header structure.
typedef struct IMAGE_COR20_HEADER
{
    // Header versioning
    DWORD                   cb;
    WORD                    MajorRuntimeVersion;
    WORD                    MinorRuntimeVersion;

    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY    MetaData;
    DWORD                   Flags;

    // If COMIMAGE_FLAGS_NATIVE_ENTRYPOINT is not set, EntryPointToken
```

[7] also in CorHdr.h. It still exists in SDK 7.0A and probably either one of them will be removed in the future.

[8] side note: CorHdr.h contains a link to Wikipedia :)

```
    // represents a managed entrypoint.
    // If COMIMAGE_FLAGS_NATIVE_ENTRYPOINT is set, EntryPointRVA
    // represents an RVA to a native entrypoint.
    union {
        DWORD               EntryPointToken;
        DWORD               EntryPointRVA;
    } DUMMYUNIONNAME;

    // Binding information
    IMAGE_DATA_DIRECTORY    Resources;
    IMAGE_DATA_DIRECTORY    StrongNameSignature;

    // Regular fixup and binding information
    IMAGE_DATA_DIRECTORY    CodeManagerTable;
    IMAGE_DATA_DIRECTORY    VTableFixups;
    IMAGE_DATA_DIRECTORY    ExportAddressTableJumps;

    // Precompiled image info (internal use only - set to zero)
    IMAGE_DATA_DIRECTORY    ManagedNativeHeader;

} IMAGE_COR20_HEADER, *PIMAGE_COR20_HEADER;
```

The CLI Header contains information about the required version of the CLR runtime, RVAs to the metadata structure, resources, a strong name, and an entry point. To obtain additional information about executing an application, the CLR parses the `Flags` entry which is represented by the `ReplacesCorHdrNumericDefines` structure defined in CorHdr.h:

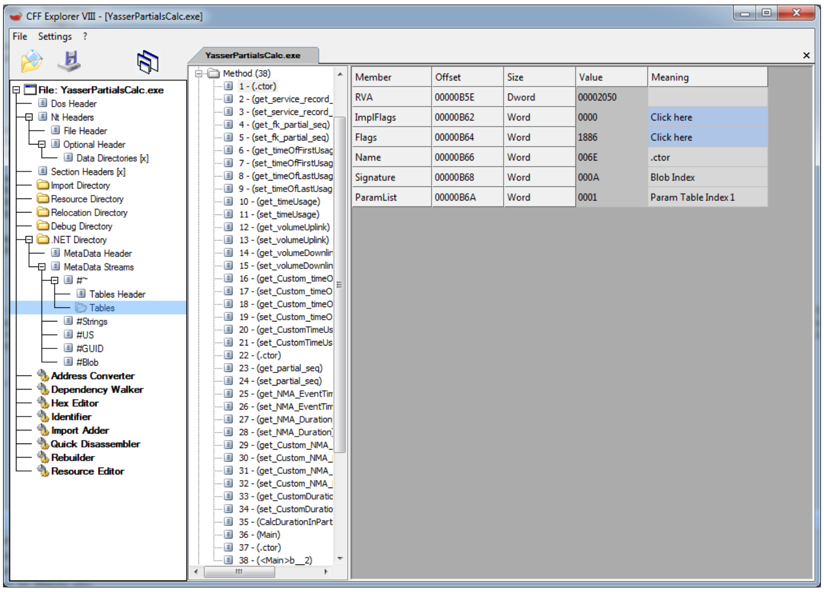
```
typedef enum ReplacesCorHdrNumericDefines
{
// COM+ Header entry point flags.
    COMIMAGE_FLAGS_ILONLY              =0x00000001,
    COMIMAGE_FLAGS_32BITREQUIRED       =0x00000002,
    COMIMAGE_FLAGS_IL_LIBRARY          =0x00000004,
    COMIMAGE_FLAGS_STRONGNAMESIGNED    =0x00000008,
    COMIMAGE_FLAGS_NATIVE_ENTRYPOINT   =0x00000010,
    COMIMAGE_FLAGS_TRACKDEBUGDATA      =0x00010000,
... other entries omitted
```

Important flags:

- `COMIMAGE_FLAGS_ILONLY` - contains CIL only. No native code
- `COMIMAGE_FLAGS_32BITREQUIRED` requires 32-bit process to execute
- `COMIMAGE_FLAGS_STRONGNAMESIGNED` - if set then assembly is signed with a strong name signature
- `COMIMAGE_FLAGS_NATIVE_ENTRYPOINT` - if set then entry point in `IMAGE_COR20_HEADER` is represented by `EntryPointRVA`, thus RVA to native code method, otherwise

by `EntryPointToken`, a token that points to a method in managed module via Method Definition table

In order to remain focused on the topic I have to take a shortcut, and fortunately there happens to be an excellent article written from the developer's perspective about the internal .NET file format. It is based on ECMA-335, PE specification and RCE experience, which eventually became a PE editor side project called CFF Explorer. The reader is strongly encouraged to download the CFF Explorer [Pis13a], load any unprotected .NET application into it, and read the article in question [Pis13e] simultaneously. The knowledge in the article in question is essential in order to understand the execution process described later on.



Member	Offset	Size	Value	Meaning
RVA	00000B5E	Dword	00002050	
ImplFlags	00000B62	Word	0000	Click here
Flags	00000B64	Word	1886	Click here
Name	00000B66	Word	006E	.ctor
Signature	00000B68	Word	000A	Blob Index
ParamList	00000B6A	Word	0001	Param Table Index 1

Figure 2.6: A .NET file loaded in CFF Explorer [Mic13n]

2.3.3 CLR's perspective

Formerly, the assembly was defined as a self-describing logical container. It contains a *manifest*, a *module*[9] and/or resources (e.g. image files, icons, etc.).

"An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope

[9] multiple modules per assembly are also supported (via assembly linker AL.exe) but rarely used

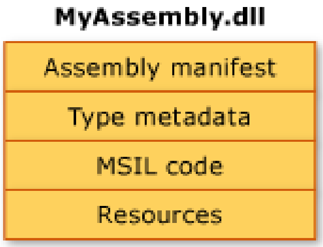


Figure 2.7: Single module assembly content [Mic13a]

of the assembly and resolve references to resources and classes" [Mic13b].

IL DASM tool[10] can display manifest content.

ECMA-335 defines the module as "a single file containing executable content". A module consists of type metadata and the CIL. However, the CIL content is not mandatory. In practice, a module always contains both type metadata and the CIL. An example of a module is presented in Figure 2.7 as *Type metadata* and *MSIL code*[11].

A module, similarly to the assembly, has a self-descriptive nature. Its type metadata is used to "describe runtime types (classes, interfaces, and value types), fields, and methods, as well as internal implementation and layout information that is used by the common language runtime (CLR)" [Mic13i]. Despite the CIL being contained in modules, the CLR itself operates on assemblies and relies on metadata.

Partition II of ECMA-335 [Mic13n] along with two Microsoft articles [Mic13i], [Mic13b] provide further details.

The CIL is the instruction set common for all CLI applications and its specification is described in Partition III of ECMA-335 [Mic13n]. It is generated by the CLI compiler and is platform and language independent. The same application written and compiled in two different CLI languages has a high probability of generating the same CIL output. The CIL exposes complete functional capabilities of the CLI, but higher-level CLI languages utilize only a subset of this instruction set. ECMA-335 and the CLR ensure the correctness and verifiability. More specifically, the CLR management is exposed as an execution time type control, exception handling (not by OS) and automatic disposal of unused objects [Lid06]. Compared to x86 assembly, it is a high-level object oriented programming language that uses an *evaluation stack*. The evaluation stack is a storage for values, and the CIL instructions must put

[10] part of Windows SDK

[11] synonym of CIL

values to and take values from it to operate. Unlike the stack of an x86 unmanaged application, the evaluation stack is not addressable, and additionally the CIL has no registers. Also its size, types in slots, and local variables are predetermined. Such strict enforcement rules are required for the CLR to be in charge, and in my opinion, assist a reverse engineer to some extent because the .NET protectors also have to follow the same rules, thus limiting their CIL manipulation options. Consider a simple example: it is possible to jump into the middle of an instruction during execution in x86 assembly code. As the result, an updated EIP register may point to a valid beginning of the new instruction from which the CPU will transparently continue the execution. Meanwhile, the static analysis in a disassembler would be presenting disassembly that might not make much, if any, sense unless manually corrected. Obviously, this *feature* is used as an anti-RCE measure by native code protectors. Finally, the CIL opcodes for .NET applications, just as Java bytecode, are simple to decompile and obtain high-level language representation due to the design nature of runtime-based platforms. Free third-party tools are available[12]. Contrary to managed code applications, unmanaged applications are disassembled. Machine instructions are mapped to the corresponding assembly language instructions. Moreover, no well-organized type metadata is available. Decompilation of such a disassembly is known to be a very difficult task. A decent set of tools such as the Hex-Rays IDA and the Hex-Rays Decompiler may cost thousands of dollars. However, the gained benefits are usually tremendous.

2.4 .NET application execution

The execution process can be divided into two major parts. The first part would be analyzing the binary data of a .NET executable on a disk until the CLR takes control. The CLR being in control can be seen as the second part. Such an approach is required to reflect the .NET file structure presented earlier in a concrete example. For that purpose, a simple .NET application was compiled in Visual Studio 2010 using the release configuration:

```
using System;
namespace SampleApp
{
    class Program
    {
```

[12] similar tools are included as part of Windows SDK

```
        static void Main(string[] args)
        {
            Console.WriteLine("I'm not hiding anything");
        }
    }
}
```

2.4.1 Execution: step one

The compiled assembly was processed by the DumpBin utility which is a part of the Visual C++ build tools. This tool analyses the PE file and outputs a detailed, well-formatted, human-readable description of the file structure[13]:

```
dumpbin /all SampleApp.exe > SampleApp.txt
```

A quick analysis yielded the following results:

- 32-bit executable
- 3 sections: *.text, .rsrc and .reloc*
- 0x400000 image base with entry point at RVA 0x276E (0x0040276E)
- non-empty COM Descriptor Directory
- single import entry to `_CorExeMain` exported by `mscoree.dll`
- NX compatible and no SEH
- CLR Header: `IL Only` and `32-Bit Required` flags set
- CLR Header: no strong name signature
- CLR Header: managed code entry point token 6000001

The entry point (0040276E) points to the stub code in *.text* section to start the CLR. The import directory starts at RVA 0x2720 and contains a single entry for `mscoree.dll`. This entry's Import Address Table (IAT) is located at RVA 0x2000 for the single imported function - `_CorExeMain` (or `_CorDllMain` in case of DLL). It is known that on disk the IAT and the Import Name Table (INT) both point to the same location, in this case the RVA 0x2750. During execution the PE loader will fill the IAT with correct addresses of the imported functions. Windows does not support position independent code and if the PE file is not loaded at the desired image base then the PE loader uses *.reloc* section to update addresses throughout the loaded PE sections. In the current example a single entry in *.reloc* points to the RVA 0x2770 (entry point at RVA 0x276E + 2 bytes = RVA 0x2770). The RVA 0x2770 or the VA

[13] content (except raw data to preserve space) is presented in A

0x402770 points to a location that contains the target address for the *jmp* instruction as explained further on. The CFF Explorer has a built-in simple disassembler. The entry point address reveals the following opcodes:

```
Address      Opcode          Instruction
0040276E:    FF25 00204000   jmp [0x402000]
```

A two bytes opcode `FF25` is a far indirect jump. Here it jumps to the address at memory location 0x402000 (i.e.: pointer dereference). During execution the IAT will contain the correct address of the imported `_CorExeMain` function. As the result this imported function call bootstraps the CLR for OSes not aware of .NET. For OSes aware of .NET[14], such as Windows XP and newer, the stub code is simply skipped [Hew10]. Their PE loader is capable of recognizing the .NET file and bootstrapping the CLR if required.

Third *.rsrc* PE section contains two entries: `VERSIONINFO` structure and WinSxS application manifest file. This manifest file is not the same as the assembly manifest described earlier and has only a conceptual relation to .NET. It is used by Windows *side by side* technology to address the DLL versioning problem for unmanaged applications.

Finally, an NX-compatible flag being set and the SEH flag being absent is also common for .NET files. As has been noted, a .NET file is a self-describing managed code container which is handled by the CLR, thus there is no need for Windows's SEH for executing application because the CLR provides its own exception handling mechanism. In conclusion, the last three entries in the analysis list describe the .NET specific characteristics of the file: only managed code that CLR must compile as 32-bit native instructions, absence of strong name signature and first entry in Method Definition metadata table are a managed entry point. As described by M. Hewardt [Hew10] and D. Pistelli [Pis13e], the managed entry point is represented by a 4-byte token. The high-order byte (0x06) is a table number and the three low-order bytes (0x000001) are an index into that table.

2.4.2 Execution: step two

A .NET aware OS PE loader, depending on the bitness of executing application, loads a corresponding 32-bit or 64-bit version of the aforementioned `mscoree.dll`[15]. This

[14] modified PE loader

[15] acronym for `Microsoft Common Object Runtime Execution Engine`

DLL is a *shim* DLL which is used to validate the loading PE file via _CorValidateImage function and bootstrap the CLR. Therefore, if validation succeeds, the PE loader calls _CorExeMain (or _CorDllMain for DLL) directly regardless of the entry point value specified in the Optional Header of the executing PE file [Mic13c]. This call instantiates the required version of the CLR and delegates the execution to the entry point defined in the CLI Header. The CLR itself is implemented as a Component Object Model (COM) server in Clr.dll for the framework version 4.0 and in MSCorWks.dll for earlier versions [Ric10]. COM implementation is an extensibility feature and allows any application to host the CLR. When the COM server is started, or in other words when the CLR is initialized, it creates three application domains (AppDomains): *system*, *shared* and *default* [Hew10]. Certainly, an application can explicitly create additional AppDomains. The executing assembly is loaded into the *default* AppDomain. The *system* AppDomain tracks and manages the loaded AppDomains. The *shared* AppDomain is used as a storage for assemblies whose types can safely be shared between domains such as MSCorLib.dll. The AppDomain is a .NET specific concept which serves as a secure and isolated code execution environment, thus a single application can have multiple AppDomains loaded, and code from one AppDomain cannot access code in another AppDomain directly but only through well-defined mechanisms [Ric10]. The OS has no knowledge of the AppDomains.

Microsoft's CLR is a proprietary CLI implementation, and obviously there is no source code available. In order to continue the recital I have to temporarily stray from the CLR context. Fortunately, Microsoft Research has developed a shared source CLI implementation called SSCLI also known as the *Rotor*. The SSCLI resembles a modified subset of the CLR. Core implementation aspects are applicable to the CLR, especially to versions before 4.0, because the SSCLI hasn't been developed since its version 2.0.

According to J. Pobar et al. [PNSS08], every type instance created via newobj or newarr CIL instructions has a reference to the Method Definition metadata table[16]. Consequently, the ClassLoader uses MethodTable and EEClass structures to build the in-memory pointer-based layout representation essential to the VES. These two structures represent the same concept but are required to separate type fields by usage frequency: frequently used ones are stored in the MethodTable and the rest are stored in the EEClass structure[Hew10]. J. Pobar et al. [PNSS08] define it as "hot" and "cold" data separation, and it is an optimization feature. Another known and notable

[16] named as MethodTable on Figure 2.8

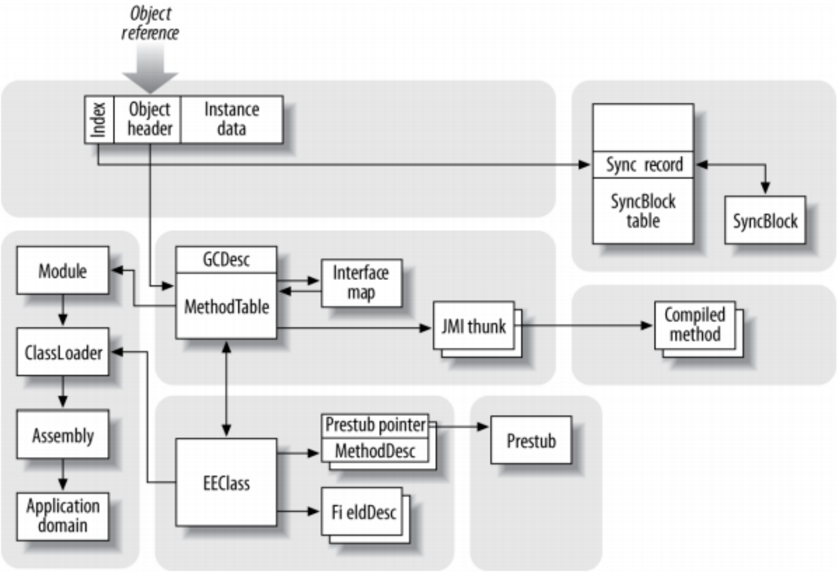


Figure 2.8: Layout of an object with its underlying structures [PNSS08]

performance optimization is the deferral compilation: if a class has two methods, e.g. A and B, and an arbitrary code is calling only A, then only A will be compiled, unless A calls B internally, which will also trigger the compilation for B. Deferral compilation is achieved with the help of special code stubs. When a `MethodTable` table is laid out, each entry is mapped to a temporary entry point function called *precode helper*, which invokes a method specific code called *prestub* [PNSS08]. In this case the precode stub simply passes the method's `MethodDesc` pointer to the prestub. The prestub uses the passed argument which points to method specific information and performs the actual compilation. Finally, it modifies the entry in the `MethodTable` by overwriting the initial pointer to the precode stub with a pointer to a memory region that contains compiled native code for the method in question. The latter modification ensures that subsequent method calls will occur without a performance hit. This happens because the compilation is not required and the previously compiled native code can be reused.

The CLR delegates the CIL compilation to its crucial component - the Just-In-Time (JIT) compiler. The CLR JIT compiler is implemented as a DLL[17] which exports the `getJit` function. In SSCLI the `fjitcompiler.cpp` contains a `getJit` function that returns a pointer to the `ICorJitCompiler` interface. The `FJitCompiler`

[17] `ClrJit.dll` for the framework version 4.0 and in `MSCorJit.dll` for earlier versions

class is a concrete implementation of the `ICorJitCompiler` abstract class as defined in `fjit.h`:

```
class FJitCompiler : public ICorJitCompiler
{
public:
    /* the jitting function */
    CorJitResult __stdcall compileMethod (
            ICorJitInfo*           comp,               /* IN */
            CORINFO_METHOD_INFO*   info,               /* IN */
            unsigned               flags,              /* IN */
            BYTE **                nativeEntry,        /* OUT */
            ULONG  *               nativeSizeOfCode    /* OUT */
            );

    /* notification from VM to clear caches */
    void __stdcall clearCache();
    BOOL __stdcall isCacheCleanupRequired();

    static BOOL Init();
    static void Terminate();

private:
    /* grab and remember the jitInterface helper addresses that we need at runtime */
    BOOL GetJitHelpers(ICorJitInfo* jitInfo);
};
```

Two static functions with self-descriptive names `Init` and `Terminate` are both called from DLL's `DllMain` function: when the DLL is loaded into and unloaded from the process memory. The most important function is the `compileMethod` function as it is responsible for CIL to native code conversion. The opportunity to intercept and/or call this function directly appears to be promising, and as a result, certain .NET protectors and RCE utilities take that advantage.

The CLR JIT determines the bitness for native instructions based on the value of `Magic` field in PE's Optional Header and `Flags` field in the CLI Header (`IMAGE_COR20_HEADER`):

- `10B` and `COMIMAGE_FLAGS_32BITREQUIRED` set - 32-bit only native instructions
- `10B` and `COMIMAGE_FLAGS_32BITREQUIRED` not set - 32-bit or 64-bit native instructions depending on OS bitness
- `20B` and `COMIMAGE_FLAGS_32BITREQUIRED` not set - 64-bit only native instructions

A brief summary:

- .NET aware OSes skip the native code CLR bootstrapper
- `mscoree.dll` starts the CLR and initiates execution

- AppDomain is a .NET specific concept and handled by the CLR
- managed code is isolated into its own AppDomain
- an application can create additional AppDomains inside the same process while code integrity and isolation between AppDomains is maintained
- an application can host the CLR via functionality exposed by `mscoree.dll`
- deferral compilation via helper code stubs
- .NET Framework `RuntimeHelpers.PrepareMethod` method can be used to force the JIT compilation for a given method
- `compileMethod` function is responsible for CIL to native code conversion
- instance methods receive `this` pointer as the first argument
- for performance reasons, the JIT compiler uses Microsoft's own `__fastcall` calling convention in native code generation

Further information on the SSCLI internals is provided by J. Pobar et al. [PNSS08]. Practical approach to metadata tables in-memory using debugger is described by M. Hewardt [Hew10]. An in-depth description of application development for the .NET Framework with respect to the core facilities and advanced topics, such as CLR Hosting, garbage collection etc., is presented by J. Richter [Ric10]. An overview of the PE file format in the context of .NET and a complete description of the CIL is provided by S. Lidin [Lid06]. Finally, the ECMA-335 [Mic13n] standard serves as an additional reference for the aforementioned sources.

2.5 Execution security

As mentioned previously, the .NET Framework was designed to be a secure managed code execution environment. For that purpose, several security mechanisms are built into the CLR. When a managed application is first loaded its PE structure and metadata integrity are verified. Next, the assembly's evidence is evaluated to grant the executing application's requested permissions and the CLR decides if and which permissions are granted in the context of the security policy. Specifically, these permissions are granted and enforced by the code access security (CAS) whereas the security policy is a set of configurable rules. The CAS also ensures that no code in a call chain exceeds the granted permissions. To accomplish this task, the runtime initiates a stack walk which checks that all assemblies participating in a call chain have also been granted the same permissions as the assembly which initiated the call. In Framework 4.0 the CAS has been vastly redesigned and implemented as the new Security Transparency model [Dai13] [Mic13k]. Finally, the JIT compiler performs simultaneous code verification and compilation. The JIT verification ensures that the code is safe, meaning that it does not circumvent the imposed security or lead to unexpected behavior. Obviously, most of the security checks are skipped for `FullTrust` assemblies. An extensive security model review of the .NET Framework prior to version 4.0 is presented by N. J. Murison [Mur05].

The assembly evidence evaluation step allows the runtime to authenticate the assembly and authorize the requested permissions. The .NET Framework was built with proper application versioning and publisher verifiability in mind. This was achieved in the form of digital signatures and strong naming that rely on public key cryptography [Lip13]. The former is required to establish the identity of the publisher, whereas the latter is used to uniquely identify the assembly itself. As a side effect, strong naming provides protection from tampering, but it is not a security feature. It does provide protection from tampering or spoofing only when the strong name signature is present. When an assembly is deployed into the GAC its strong name is verified. However, when a `FullTrust` .NET application is referencing another assembly in the GAC, the target's assembly strong name signature is not verified for performance reasons. This is very dangerous, because all .NET applications reference one or more system libraries and if a library is tampered **`all`** applications are affected. Malicious software would take advantage of that by targeting the .NET Framework directly.

To summarize, the strong name signature is intended for versioning only. Moreover, it can be easily disabled by, e.g. byte patching, as presented by A. Bertolotto [Ber13]. As such, it cannot be considered a serious security enabler that many .NET developers think it is. Rather, it is a valuable addition to other code protection mechanisms.

Chapter 3

Practical part

.NET applications can be considered open source by default, because, as described earlier, every application is compiled into CIL which is a high-level object oriented language that operates on an evaluation stack, and the runtime heavily relies on metadata. Thus, CIL can easily be decompiled to virtually any CLI language. Code obfuscation is very popular in the .NET environment. In order to protect IP, commercial versions of Microsoft Visual Studio are shipped with the Dotfuscator application, which performs code obfuscation similar to several other .NET protectors available on the market. From the runtime's perspective everything is valid, since renaming fields, types or methods does not alter the application behavior. However, from the reverse engineer's point of view, obfuscation is yet another barrier to overcome, as it makes it difficult to understand the code. To make things even more complicated, the obfuscation process can insert garbage CIL instructions, unused types, dummy fields, alter control flow, utilize various tricks to shutdown the disassembler and/or decompiler, and encrypt string constants throughout the application. An analysis of .NET application obfuscation is presented by L. M. Frydenberg [Fry06]. The main idea is that some modifications can be partially or completely reversed. Each obfuscator has its own *signature*, in other words, the type and characteristics of modifications it performs. By distinguishing these signatures one can precisely target specific obfuscator's modifications and reverse them. Apparently, this process can be automated to some extent, and the popular de4dot utility is a perfect example of a feature-rich .NET deobfuscator which supports a wide range of various obfuscators and packers. Besides obfuscation, some part of the .NET application functionality can be implemented as unmanaged, i.e. native code DLL and called via P/Invoke (Platform Invoke) features of the CLR. In addition, such DLL can be protected with native code protectors. The

RCE of native code (especially with decent protection) is of an absolutely different level of complexity compared to the RCE of .NET applications. Native code protectors operate at the lowest level, hence opportunities are quite extensive. For example, there are two types of breakpoints: software and hardware. In the x86 architecture, a software breakpoint is `int 3` - an instruction represented by `CC` opcode. On the other hand, hardware breakpoints are, in essence, CPU debug registers. It is certain that breakpoints exist for and are used by developers, however, a native code protector can utilize the CPU's debug registers to store values used for data decryption. As a result, when a hardware breakpoint is set by a reverse engineer, it overwrites the previous value set by the protector and the entire decryption routine fails. In addition, the protector can insert code that will detect software breakpoints or tampering by calculating the checksum of the given code block in-memory. Even more interesting anti-debugging tricks include intentional hardware faults that are intercepted by the protector's designated handlers. A fault must occur for successful execution and if there is a *person in the middle* (a debugger) it intercepts the fault condition first, and if not properly resolved and passed through, the respective routine fails. Stealing the OEP code bytes, resolving the IAT at runtime, timed code execution and a lot of other options are available in the native code.

The developer is responsible for the evaluation of all pros and cons whether to implement certain parts of the application as native code DLL or not. Imposed limits, complexity and feasibility must be taken into account. Some protectors, however, are capable of extracting the source CIL, and converting it into native code, which is then placed into a separate DLL, and inserting trampoline calls into that unmanaged DLL in place of the original CIL code. The Salamander .NET protector completely converts CIL into native code with the .NET metadata preserved. Nevertheless, it is not a clean approach and can lead to problems. This protector is reviewed, and the benefits and drawbacks of native compilation are evaluated by D. Pistelli [Pis13c] [Pis13d].

Another popular protection involves a native code wrapper. Upon execution, it decrypts the .NET assembly, starts up the .NET runtime and hands a decrypted byte array to it. The runtime does the rest. While on disk, this schema provides decent protection. However, the runtime expects the byte array to be a valid .NET assembly so it is prone to a memory dump. The protector may alter the assembly after it has been loaded, thus a plain dump will not be valid. In most cases, the correction of the PE headers and the .NET metadata entries is sufficient. For example, an application protected by the Themida Winlicence can be dumped with the SND DotNet Dumper 1.0 and then *corrected* with the SND Universal Fixer 1.0, and finally processed by the aforementioned de4dot. The correction step is mandatory, because the dumped .NET assembly has a malformed structure and will fail to execute. Once corrected, it executes normally. However, the file size is still huge, but processing it with the de4dot dramatically reduces its size.

Furthermore, a reasonable question arises on how to render memory dumping useless. One option is to prevent the original CIL from being loaded and kept in memory in the first place. Basically, protected routines can contain dummy code that will trigger the execution of the original CIL. To achieve its goal, a protector must hook into the CLR internals. The previous chapter described how a JIT-compiler compiles CIL to native code via the `compileMethod` function. If the application controls the invocation of this function, it can perform numerous things such as call tracing, code injection, etc. When `compileMethod` function is hooked, the protector intercepts every JIT compilation request. Based on provided data, the intercepting code decides what to compile. Firstly, the custom routine will analyze what is requested, and then, for example, decrypt the referenced method body and substitute stub CIL with the decrypted original CIL. Then, the protector may choose to either call the original `compileMethod` or execute a custom code that resembles the internal implementation of the `compileMethod`. This protection schema description is applicable to the DNGuard and similar protectors. Further information on SSCLI/.NET code injection is presented in an excellent article by D. Pistelli [Pis13b].

This chapter is aimed at the CIL level analysis and manipulation of .NET applications. It is divided into two parts. In Part 1 each subchapter describes a specific area of interest from the RCE perspective of .NET. There is no single correct solution that always works. Therefore, my intention was to describe some of the approaches, highlights, opportunities and analysis flow in a broader scope rather than plainly reviewing each protector. Every subchapter contains background information, a practical part and discussion. I decided not to compare the .NET protectors with each other. After all, tutorials on how to crack the most popular .NET protectors are freely available on the Internet. Nevertheless, I could not find any information on the InishTech SLP code protector and licensing manager. In Part 2, I analyze the protector and finally apply that knowledge to a real commercial software protected by the InishTech SLP.

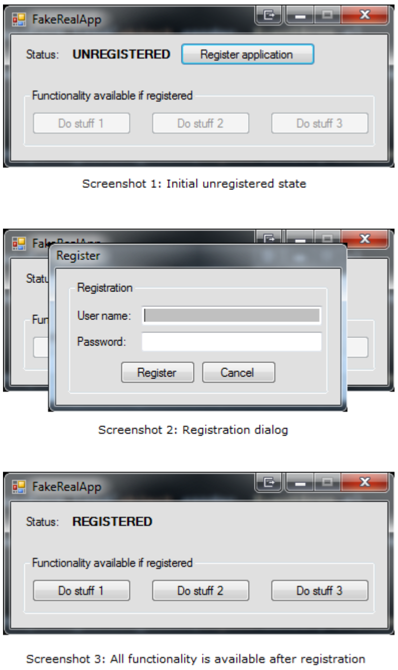
All tools used are available on the web. Another important step is to obtain the Windows symbol information from the Microsoft Symbol Server. Debug symbol information dramatically increases the code readability. Download instructions are provided on the Microsoft website [Mic13m]. First, I downloaded the debug symbol to my local cache folder `c:\symbols`. Addresses of loaded DLLs displayed in the debugger may be different on other PCs. Next, the JIT compiles CIL into native code at runtime, memory is allocated dynamically, and as a result the addresses of these *code chunks* vary for every compilation.

3.1 Part 1: Common RCE techniques applicable to .NET

I have developed a simple test .NET application[1] called FakeRealApp that represents an example of a popular Pay Licensed Closed-Source software license category, i.e. pay to use and no sources. For the rest of the chapter, it is assumed that I have downloaded this application from the Internet and obtained the registration information. Why do I need to have a working key? The answer is to simplify the RCE process. Of course, it may not always be required, but one way or another the registration data can be utilized in some decryption routine which is not susceptible to bruteforcing or cracking. In addition, a valid key allows shifting the attack vector. Take for example a random application that uses asymmetric cryptography to check the registration data with "am I registered?" state checks placed throughout the application. After a successful decryption step, the application verifies with the home web service if the current registration information matches the hash which was calculated during initial registration. The hash can be calculated from the computer's main HDD serial number or CPUID data. This avoids the usage of the same key on multiple computers. Thus, it is more convenient to crack the key uniqueness check. Now, a single key becomes usable on an unlimited number of computers. A more sophisticated approach would be to replace the original cryptography keys with your own keys and develop a custom web service to check if the key is unique. From that point on, it is possible to generate and distribute unique software keys just as its real authors do, but for the patched version of the software.
Application analysis summary:

- 32-bit .NET 4.0 WinForms application. No protector used
- cannot perform any useful operations unless it is registered. UI button controls

[1] compiled as 32-bit application for research convenience



Screenshot 1: Initial unregistered state

Screenshot 2: Registration dialog

Screenshot 3: All functionality is available after registration

Figure 3.1: An example of a pseudo-real commercial software

are disabled

- registration requires a valid username and password combination
- registration dialog notifies the user in both cases: whether registration was successful or not. Notification is displayed with a regular message box
- if the registration was successful the UI is unlocked and the application becomes fully functional
- once registered and restarted, the application starts in its registered state meaning it has saved the registration information somewhere

3.1.1 Native code debugger

This subchapter is focused on the applicability of a native code debugger targeting a .NET application.
Tools used:

- User mode Olly debugger (OllyDbg) is one of the most popular native code debuggers. For its version 1.x users have developed a huge amount of various plugins and scripts in response to vast RCE needs. Any decent native code protector contains special tricks to detect and even crash it. If certain issues are mitigated by plugins other issues are resolved by patching the debugger binary. Bundles that contain patched and configured Olly with plugins are common. The most recent version is 2.01. Unfortunately, old plugins are not compatible and 64-bit PE executables are not yet supported. This version is capable of loading .NET executables. However it does not have the desired level of .NET awareness. Software and documentation are presented by O. Yuschuk [Yus13].

Starting from the initial configuration, the folder was opened with the Olly debugger. Steps were the following: right click on ollydbg.exe, then select “Properties”. In “Compatibility” tab the checkbox “Run this program as an administrator” must be checked. Next, Olly was configured to use downloaded symbol files. In the program options the “Debugging data” entry was selected and the path to local symbols folder was specified. The “Allow access to Microsoft Symbol Server” check was set, followed by clicking “OK” and restarting Olly.

In running Olly, RegMe.exe was selected via “File” - “Open” dialog. For new executables a recurring analysis step takes longer to complete. However, its results are cached for a faster start in the future. Olly stopped at a .NET entry point inside `mscoree.dll`. The application execution was resumed by pressing F9. An exception was encountered in `kernelbase.dll` and it was passed to the program by pressing Shift + F9. For the second exception in the same `kernelbase.dll` the Shift + F9 key combination was pressed again, and the target application was loaded.

From the preliminary analysis we know that the application notifies the user in both cases whether registration information is correct or not. The notification is displayed via what appears to be a standard Windows message box defined in `user32.dll`. This assumption was verified by going back to the Olly menu bar “View” - “Executable modules”. In a newly opened window corresponding library should be found, in my case like so:

```
Executable modules, item 33
  Base = 75790000
  Size = 00100000 (1048576.)
  Entry = 757AB6ED
  Name = USER32
  Type =
  File version = 6.1.7601.17514 (win7sp1_rtm.101119-1850)
  Static links = ADVAPI32, GDI32, KERNEL32, ntdll
  Path = C:\Windows\syswow64\USER32.dll
```

After right clicking on the item and selecting “Show names”, the text “messagebox” without quotes was typed. This window has a quick search option which selects the first matching item. After scrolling down a little

```
Names in USER32, item 2828
  Address = 757FFD3F
  Section = .text
  Type = Export
  Name = MessageBoxW
  Comments = Ordinal = #2046
```

was encountered. In general, it is useful to set breakpoints to all exported unicode[2] versions of message box functions to determine which one is used. The Unicode version of any WinAPI function ends with a “W” character such as `MessageBoxW`. For this example it was sufficient to set a breakpoint on only that function by selecting it and pressing F2. The address column became red. In the target application the “Register application” button was clicked and in the registration dialog a text was input: “Test” as the user name and “Password” as the password (both without quotes). After clicking on the “Register button” Olly stopped inside `user32.dll`:

```
Address   Hex dump          Command                     Comments
757FFD3F  /$  8BFF          MOV EDI,EDI                 ; UNICODE "Invalid regist...
757FFD41  |.  55            PUSH EBP
757FFD42  |.  8BEC          MOV EBP,ESP
757FFD44  |.  6A 00         PUSH 0                      ; /LanguageID=LANG_NEUTRAL
757FFD46  |.  FF75 14       PUSH DWORD PTR SS:[ARG.4]   ; |Type => [ARG.4]
757FFD49  |.  FF75 10       PUSH DWORD PTR SS:[ARG.3]   ; |Caption => [ARG.3]
757FFD4C  |.  FF75 0C       PUSH DWORD PTR SS:[ARG.2]   ; |Text => [ARG.2]
757FFD4F  |.  FF75 08       PUSH DWORD PTR SS:[ARG.1]   ; |hOwner => [ARG.1]
757FFD52  |.  E8 A3FFFFFF   CALL MessageBoxExW          ; \USER32.MessageBoxExW
757FFD57  |.  5D            POP EBP
757FFD58  \.  C2 1000       RETN 10
```

Olly tried to be helpful and displayed useful information in the comments column. The `EDI` register contains a pointer to a Unicode string. A pointer to the same string was pushed onto the stack at 0x757FFD4C as a second argument to the `MessageBoxExW` function call at 0x757FFD52. The column row at 0x757FFD3F was red because there was a breakpoint. Clicking on that row and pressing F2 removed the breakpoint. Now it was important to return to the code fragment of the test application that called the message box routine. Ctrl + F9 was pressed which executed the code until the function return, and when Olly stopped at the `ret` instruction, F8 was pressed. It was

[2] .NET conforms to the Unicode standard

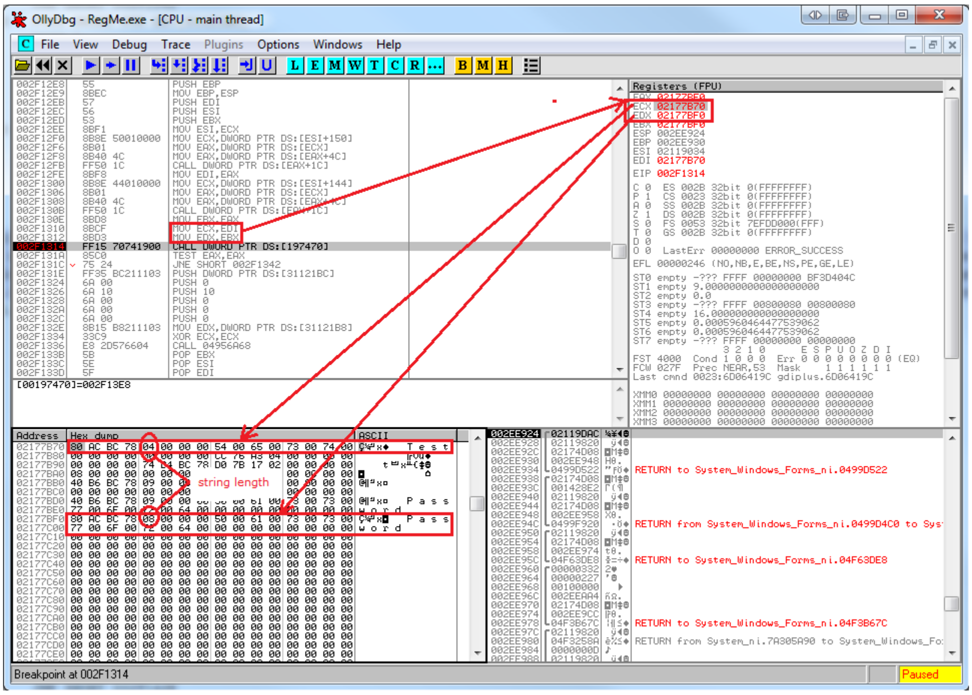
repeated until Olly's title bar contained the [CPU - main thread] text. After the first Ctrl + F9 a message box was displayed and it was necessary to close it by clicking the "OK" button.

```
Address   Hex dump          Command
002F1310   8BCF             MOV ECX,EDI
002F1312   8BD3             MOV EDX,EBX
002F1314   FF15 70741900    CALL DWORD PTR DS:[197470]
002F131A   85C0             TEST EAX,EAX
002F131C   75 24            JNE SHORT 002F1342
002F131E   FF35 BC211103    PUSH DWORD PTR DS:[31121BC]
002F1324   6A 00            PUSH 0
002F1326   6A 10            PUSH 10
002F1328   6A 00            PUSH 0
002F132A   6A 00            PUSH 0
002F132C   6A 00            PUSH 0
002F132E   8B15 B8211103    MOV EDX,DWORD PTR DS:[31121B8]
002F1334   33C9             XOR ECX,ECX
002F1336   E8 2D576604      CALL 04956A68
002F133B   5B               POP EBX
```

The `EIP` is at 0x002F133B. Previous instruction at 0x002F1336 is a call that triggered the message box. A conditional jump at 0x002F131C was not taken because `EAX` register was zero for some reason. The reason could be because the call at 0x002F1314 performs a username and password validation. In order to be sure, the 0x002F1314 row was selected, and the breakpoint was set by pressing F2 and execution was continued with F9. Back in the registration dialog, the "Register" button was clicked again and Olly stopped at 0x002F1314. Note the two instructions above the call. As described in the previous chapter, this is Microsoft's own `__fastcall` calling convention, i.e. the first two arguments that can be accommodated into a register are passed through `ECX` and `EDX` registers. Right clicking on the `ECX` register in Registers window and selecting "Follow in Dump" allowed seeing the first string object bytes in memory.

It was now evident that the call at 0x002F1314 received two arguments which were strings with user input. Furthermore, there was no *this* pointer as the result of a static function call. Pressing F7 allowed stepping into the call. I added comments between code lines:

```
Address   Hex dump          Command

Standard prologue, setup new stack frame
002F13E8   55               PUSH EBP
002F13E9   8BEC             MOV EBP,ESP
```



```
OllyDbg - RegMe.exe - [CPU - main thread]

002F12E8    55               PUSH EBP
002F12E9    8BEC             MOV EBP,ESP
002F12EB    57               PUSH EDI
002F12EC    56               PUSH ESI
002F12ED    53               PUSH EBX
002F12EE    8BF1             MOV ESI,ECX
002F12F0    8B8E 50010000    MOV ECX,DWORD PTR DS:[ESI+150]
002F12F6    8B01             MOV EAX,DWORD PTR DS:[ECX]
002F12F8    8B40 4C          MOV EAX,DWORD PTR DS:[EAX+4C]
002F12FB    FF50 1C          CALL DWORD PTR DS:[EAX+1C]
002F12FE    8BF8             MOV EDI,EAX
002F1300    8B8E 44010000    MOV ECX,DWORD PTR DS:[ESI+144]
002F1306    8B01             MOV EAX,DWORD PTR DS:[ECX]
002F1308    8B40 4C          MOV EAX,DWORD PTR DS:[EAX+4C]
002F130B    FF50 1C          CALL DWORD PTR DS:[EAX+1C]
002F130E    8BD8             MOV EBX,EAX
002F1310    8BCF             MOV ECX,EDI
002F1312    8BD3             MOV EDX,EBX
002F1314    FF15 70741900    CALL DWORD PTR DS:[197470]
002F131A    85C0             TEST EAX,EAX
002F131C  v 75 24            JNE SHORT 002F1342
002F131E    FF35 BC211103    PUSH DWORD PTR DS:[31121BC]
002F1324    6A 00            PUSH 0
002F1326    6A 10            PUSH 10
002F1328    6A 00            PUSH 0
002F132A    6A 00            PUSH 0
002F132C    6A 00            PUSH 0
002F132E    8B15 B8211103    MOV EDX,DWORD PTR DS:[31121B8]
002F1334    33C9             XOR ECX,ECX
002F1336    E8 2D576604      CALL 04956A68
002F133B    5B               POP EBX
002F133C    5E               POP ESI
002F133D    5F               POP EDI
[00197470]=002F13E8

Registers (FPU)
EAX 02177BF0
ECX 02177B70
EDX 02177BF0
EBX 02177BF0
ESP 002EE924
EBP 002EE930
ESI 02119034
EDI 02177B70

EIP 002F1314

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -??? FFFF 00000000 BF3D404C
ST1 empty 9.0000000000000000000
ST2 empty 0.0
ST3 empty -??? FFFF 00800080 00800080
ST4 empty 16.000000000000000000
ST5 empty 0.00059604644775390625
ST6 empty 0.00059604644775390625
ST7 empty -??? FFFF 00000000 00000000
               3 2 1 0      E S P U O Z D I
FST 4000  Cond 1 0 0 0  Err 0 0 0 0 0 0 0 0 (EQ)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
Last cmnd 0023:6D06419C gdiplus.6D06419C

XMM0 00000000 00000000 00000000 00000000
XMM1 00000000 00000000 00000000 00000000
XMM2 00000000 00000000 00000000 00000000
XMM3 00000000 00000000 00000000 00000000

Address  Hex dump                                         ASCII
02177B70 80 AC BC 78 04 00 00 00 54 00 65 00 73 00 74 00  Ç¼╝x♦   T e s t
02177B80 00 00 00 00 00 00 00 00 CC 76 A3 04 00 00 00 00          ╠vú♦
02177B90 00 00 00 00 74 C4 BC 78 D0 7B 17 02 00 00 00 00      t─╝x╨{↨☻
02177BA0 08 00 00 00 00 00 00 [illegible]    00 00 00 00  ◘
02177BB0 40 B6 BC 78 09 00 00 [illegible]    00 00 00 00  @╢╝x○
02177BC0 00 00 00 00 00 00 00 [illegible]    00 00 00 00
02177BD0 40 B6 BC 78 09 00 00 00 50 00 61 00 73 00 73 00  @╢╝x○   P a s s
02177BE0 77 00 6F 00 72 00 64 00 00 00 00 00 00 00 00 00  w o r d
02177BF0 80 AC BC 78 08 00 00 00 50 00 61 00 73 00 73 00  Ç¼╝x◘   P a s s
02177C00 77 00 6F 00 72 00 64 00 00 00 00 00 00 00 00 00  w o r d
02177C10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177C90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177CA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177CB0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177CC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177CD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02177CE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

002EE924  02119DAC
002EE928  02119820
002EE92C  02174D08
002EE930  002EE948
002EE934  0499D522  RETURN to System_Windows_Forms_ni.0499D522
002EE938  02174D08
002EE93C  001428E2
002EE940  02119820
002EE944  02174D08
002EE948  002EE958
002EE94C  0499F920  RETURN from System_Windows_Forms_ni.0499D4C0 to Sys
002EE950  02119820
002EE954  02174D08
002EE958  002EE974
002EE95C  04F63DE8  RETURN to System_Windows_Forms_ni.04F63DE8
002EE960  00000332
002EE964  00000227
002EE968  00100000
002EE96C  002EEAA4
002EE970  02174D08
002EE974  002EE9CC
002EE978  04F3B67C  RETURN to System_Windows_Forms_ni.04F3B67C
002EE97C  02119820
002EE980  04F3258A  RETURN from System_ni.7A305A90 to System_Windows_Fo
002EE984  0000000D
002EE988  02119820

Breakpoint at 002F1314                                   Paused
```

Figure 3.2: ECX and EDX contain pointers to string objects with user data

```
Save caller registers
002F13EB    57              PUSH EDI
002F13EC    56              PUSH ESI
002F13ED    53              PUSH EBX
002F13EE    50              PUSH EAX

Zero EAX and set local variable value SS:[EBP-10] to zero
002F13EF    33C0            XOR EAX,EAX
002F13F1    8945 F0         MOV DWORD PTR SS:[EBP-10],EAX

Copy pointer to user password string to EDI register
002F13F4    8BFA            MOV EDI,EDX

Check if username is not null, will jump otherwise
002F13F6    85C9            TEST ECX,ECX
002F13F8    74 0C           JE SHORT 002F1406

Compare username string length to zero, store result in EAX
002F13FA    8379 04 00      CMP DWORD PTR DS:[ECX+4],0
002F13FE    0F94C0          SETE AL
002F1401    0FB6C0          MOVZX EAX,AL
002F1404    EB 05           JMP SHORT 002F140B
002F1406    B8 01000000     MOV EAX,1
```

```
Check username comparison result. If zero - good, string has
characters, so no jump
002F140B    85C0            TEST EAX,EAX
002F140D    75 1F           JNE SHORT 002F142E

The following block is identical to the previous blocks. Same checks
for the password string: if it is null or empty
002F140F    85FF            TEST EDI,EDI
002F1411    74 0C           JE SHORT 002F141F
002F1413    837F 04 00      CMP DWORD PTR DS:[EDI+4],0
002F1417    0F94C0          SETE AL
002F141A    0FB6C0          MOVZX EAX,AL
002F141D    EB 05           JMP SHORT 002F1424
002F141F    B8 01000000     MOV EAX,1
002F1424    85C0            TEST EAX,EAX
002F1426    75 06           JNE SHORT 002F142E

Next two instructions ensure that the username string must be
at least six characters long
002F1428    8379 04 06      CMP DWORD PTR DS:[ECX+4],6
002F142C    7D 08           JGE SHORT 002F1436

Zero EAX. Upon return the value of EAX is checked and if it is zero, the
application displays a message box saying "Invalid registration information"
002F142E    33C0            XOR EAX,EAX

Restore caller registers
002F1430    59              POP ECX
002F1431    5B              POP EBX
002F1432    5E              POP ESI
002F1433    5F              POP EDI

Function epilogue
002F1434    5D              POP EBP
002F1435    C3              RETN
```

A string in .NET is a special kind of citizen unlike a simple null terminated character array. Every string is an object that encapsulates character array handling and additional information about itself, such as length. Hence a string object does not necessarily need to contain any characters. Such a string is considered to be an empty string, but not null because the object exists. Figure 3.2 illustrates the string object in memory. I highlighted the starting offset of the string length information bytes. Previous assembly code listing checked that the username and password strings are not null or empty. To succeed, the last check at 0x002F1428 requires the username to contain at least six characters.

The jump at 0x002F142C was not taken because the current username string contained the text “Test” which is less than six characters. I had the previous breakpoint removed and a new one set at 0x002F1436. F9 was pressed to continue the execution. The application showed a message box about a failed registration, “OK” was clicked, and the username was updated to “Test12” in order to match the six characters requirement. Then, the “Register” button was clicked again and the breakpoint was encountered:

```
Address   Hex dump          Command

Zero ESI. It is used as loop counter
002F1436   33F6             XOR ESI,ESI

Save username string length to EDX and another compiler sanity
check is username string is empty
002F1438   8B51 04          MOV EDX,DWORD PTR DS:[ECX+4]
002F143B   3BF2             CMP ESI,EDX
002F143D   73 44            JAE SHORT 002F1483

[ECX+8] - username + 8 bytes offset = actual string character array. Copy WORD
length content (two bytes as it Unicode) from (loop counter * 2 + character array)
This is an index for string character array. Basically following line copies
character at given index to EBX. Loop counter is multiplied by 2 because
it is Unicode and each character is represented with two bytes.
002F143F   0FB75C71 08      MOVZX EBX,WORD PTR DS:[ESI*2+ECX+8]

Copy current loop counter value into EAX.
002F1444   8BC6             MOV EAX,ESI

Increase EAX by 1.
002F1446   40               INC EAX

Compare increased loop counter by 1 with the length of the username string.
If less do not jump
002F1447   3BC2             CMP EAX,EDX
002F1449   73 38            JAE SHORT 002F1483

Next lines explain the need for increased loop counter. EAX contains
next character. Then ESI which is current loop counter increased by 2.
Pseudo code for previous MOVZX and following is:
EBX = username[i]
EAX = username[i + 1]
002F144B   0FB74441 08      MOVZX EAX,WORD PTR DS:[EAX*2+ECX+8]
002F1450   46               INC ESI
002F1451   46               INC ESI

Jump back to 0x002F143B if ESI is less than 6. So when loop counter
is 6 or above loop terminates.
002F1452   83FE 06          CMP ESI,6
002F1455 ^ 7C E4            JL SHORT 002F143B
```

Loop has ended. Multiply current value in EBX by 0x3E8 and save result to EDX

```
002F1457    69D3 E8030000   IMUL EDX,EBX,3E8
```

Take current value in EAX and add it to EDX.

```
002F145D    03D0            ADD EDX,EAX
```

Save to local variable.

```
002F145F    8955 F0         MOV DWORD PTR SS:[EBP-10],EDX
```

At this point ECX contains pointer to username string. EDX has value of 0xBF9A. Next call is unknown.

```
002F1462    E8 492A7D78     CALL 78AC3EB0
```

Result is pushed to stack. ECX contains local variable value of 0xBF9A, EDX is 0 and another unknown call.

```
002F1467    50              PUSH EAX
002F1468    8B4D F0         MOV ECX,DWORD PTR SS:[EBP-10]
002F146B    33D2            XOR EDX,EDX
002F146D    E8 76E5180F     CALL 0F47F9E8
```

At this point I speculated that the previous two calls were required to create a new string and to set its value to 49050, a decimal of 0xBF9A.
Also, when the EIP was at 0x002F146D and memory was viewed at the EAX, it was revealed that this new string object had a length of five and value of 49050.

Three parameters were passed to a static function call at 0x002F1478. The first two parameters were newly created strings with values of 49050 and a user input password string with the value of 'Password'. The third parameter was 5. The entire routine was almost finished. The following call is the last one that can affect the EAX value. I assumed that the following call was some sort of a string comparison. Otherwise it is not clear why it would need a calculated value of 49050 and the user input password as parameters.

```
002F1472    8BD0            MOV EDX,EAX
002F1474    6A 05           PUSH 5
002F1476    8BCF            MOV ECX,EDI
002F1478    E8 83FA7C78     CALL 78AC0F00
```

After stepping over the call at 0x002F1478 the EAX became zero. Clearly this call performed some logic based on the calculated value of 49050 and the user input of 'Password'. At this point it was obvious that the next logical step was to replace the old 'Password' text in the registration dialog with the value 49050. By doing so this call compared 49050 to 49050 and set the EAX to be not zero.

Restore caller registers

```
002F147D    59              POP ECX
002F147E    5B              POP EBX
002F147F    5E              POP ESI
002F1480    5F              POP EDI
```

```
Function epilogue
002F1481    5D             POP EBP
002F1482    C3             RETN
```

The presented assembly code listing can be converted into the following C pseudocode:

```
int loopCounter;
int locVar1;
int locVar2;
int finalValue;

for(loopCounter = 0; loopCounter < 6; loopCounter += 2)
{
  locVar1 = username[loopCounter];
  locVar2 = username[loopCounter + 1];
}

finalValue = locVar2 + locVar1 * 0x3E8;
```

Apparently the use of the loop is useless because it overwrites the previous values. It loops three times and only the last results are used. It can be safely optimized into:

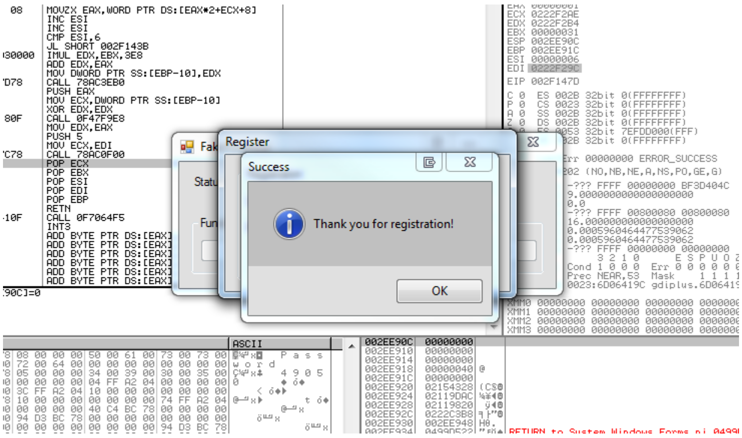
```
locVar1 = username[4]; // get 5th character
locVar2 = username[5]; // get 6th character
```

Therefore the final valid key is calculated as:

```
validKey = (int)username[5] + (int)username[4] * 1000; // 0x3E8 is 1000
```

The key generation and validation algorithms were now known, which means that a key generator was now easily implementable. When the `EIP` was at 0x002F145F, the `EDX` register had a value of a calculated valid key. The remaining code from 0x002F145F until return simply converted this value into the string and performed a string comparison. If the strings match then the `EAX` is nonzero. With all the breakpoints removed, the application continued its execution by pressing F9. Back in the registration dialog, “49050” (without quotes) was input as the password. Success!

Though it was not the most convenient .NET application analysis (at the lowest level), the goal was achieved. The assembly code of the validation routine was generated and allocated on the heap at runtime. The crucial requirement was to get it as close as possible. I used the `MessageBoxW` WinAPI function as I observed its usage during the initial analysis. Protection could have been silent, i.e. with no message



```
08      MOVZX EAX,WORD PTR DS:[EAX*2+ECX+8]
        INC ESI
        INC ESI
        CMP ESI,6
        JL SHORT 002F143B
30000   IMUL EDX,EBX,3E8
        ADD EDX,EAX
        MOV DWORD PTR SS:[EBP-10],EDX
D78     CALL 78AC3EB0
        PUSH EAX
        MOV ECX,DWORD PTR SS:[EBP-10]
        XOR EDX,EDX
80F     CALL 0F47F9E8
        MOV EDX,EAX
        PUSH 5
        MOV ECX,EDI
C78     CALL 78AC0F00
        POP ECX
        POP EBX
        POP ESI
        POP EDI
        POP EBP
        RETN
10F     CALL 0F7064F5
        INT3
        ADD BYTE PTR DS:[EAX]
        ADD BYTE PTR DS:[EAX]
        ADD BYTE PTR DS:[EAX]
        ADD BYTE PTR DS:[EAX]
        ADD BYTE PTR DS:[EAX]
        ADD BYTE PTR DS:[EAX]
90C]=0
```

```
ECX 0222F2AE
EDX 0222F2B4
EBX 00000031
ESP 002EE90C
EBP 002EE91C
ESI 00000006
EDI 0222F29C

EIP 002F147D

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
     FS 0053 32bit 7EFDD000(FFF)
       02B 32bit 0(FFFFFFFF)

Err 00000000 ERROR_SUCCESS
202 (NO,NB,NE,A,NS,PO,GE,G)

-??? FFFF 00000000 BF3D404C
9.0000000000000000000
0.0
-??? FFFF 00800080 00800080
16.000000000000000000
0.00059604644775390625
0.00059604644775390625
-??? FFFF 00000000 00000000
     3 2 1 0      E S P U O Z
Cond 1 0 0 0  Err 0 0 0 0 0 0
Prec NEAR,53  Mask    1 1 1 1
0023:6D06419C gdiplus.6D06419

XMM0 00000000 00000000 00000000 0000000
XMM1 00000000 00000000 00000000 0000000
XMM2 00000000 00000000 00000000 0000000
XMM3 00000000 00000000 00000000 0000000
```

```
                                                 ASCII
8 08 00 00 00 50 00 61 00 73 00 73 00            P a s s
0 72 00 64 00 00 00 00 00 00 00 00 00        w o r d
8 05 00 00 00 34 00 39 00 30 00 35 00            4 9 0 5
0 00 00 00 00 04 FF A2 04 00 00 00 00        0
0 3C FF A2 04 10 00 00 00 00 00 00 00          <
8 10 00 00 00 00 00 00 00 74 FF A2 04               t
0 00 00 00 00 40 C4 BC 78 00 00 00 00
0 94 D3 BC 78 00 00 00 00 00 00 00 00
0 00 00 00 00 00 00 00 00 94 D3 BC 78
```

```
002EE90C  00000000
002EE910  00000000
002EE914  00000000
002EE918  00000040  @
002EE91C  00000000
002EE920  02154328  (C$
002EE924  02119DAC
002EE928  02119820
002EE92C  0222C3B8
002EE930  002EE948  H0.
002EE934  0499D522        RETURN to System.Windows.Forms.ni.0499
```

Thank you for registration!

Figure 3.3: Successful registration. Username is “Test12” and corresponding password is “49050”

box at all. In that case, it made sense to set a breakpoint on another WinAPI function that retrieves user input from textbox control, such as `GetWindowTextW`, because this is a Windows Forms application and proceed from that point. However, it is not always that simple. Every time the test application started, it silently checked the “is registered” condition and adjusted UI accordingly. Also, the registration process could have been different and always required an application restart for any user input without instant validation. Surely, that input is saved somewhere, and the registry and the file WinAPI functions could be good places to look from. The Sysinternals Process Monitor can trace and log the file I/O and registry access activity, but even so the entire RCE process becomes very complicated and time consuming. For achieving the best results an analysis at the native code level must be addressed to dissection of the native code portions utilized by certain protectors. For .NET application analysis a debugger must have certain .NET-awareness and be capable of handling the .NET application at the CIL + .NET metadata level.

3.1.2 Native code debugger with .NET-awareness

At this point our test application was registered and the registration information was stored in the `user.config` file. In order to reset the registration this file was deleted from one of the subdirectories in the `c:\Users\<your profile name>\AppData\Local\RegMe` directory. A simple file search was sufficient.

The .NET application is all about the CIL and the metadata. Both of them are

handled by the CLR. This subchapter presents another native code debugger but with .NET-aware plugins. The goal was to find the registration check performed at the application start.
Tools used:

- Microsoft WinDbg. Very powerful debugger for Windows capable of user and kernel mode debugging in addition to debugging drivers and crash dumps. Extendable by plugins. Distributed as a part of the Debugging Tools for Windows [Mic13d]
- Microsoft SOS.dll (Son Of Strike Debugging Extension). Installed with the .NET Framework to assist the debugging of managed applications. The details and a commands list are presented by Microsoft [Mic13l]
- SOSEX.dll is a third-party plugin developed to fulfill shortcomings of SOS. Software and commands list are made available by S. Johnson [Joh13]

Our target application was 32-bit, so the 32-bit versions of WinDbg and SOSEX were used. SOSEX DLL was copied to the WinDbg folder.

Initial configuration. The folder with 32-bit WinDbg[3] was opened. Right clicking the windbg.exe was followed by selecting “Properties”. In the “Compatibility” tab the checkbox “Run this program as an administrator” was checked. Next, the debug symbols were configured. I started WinDbg, then clicked on the “File” menu and selected the “Symbol File Path”’. Input:

```
SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
```

checked “Reload”, clicked “OK” and then restarted WinDbg. In case the symbols are not properly loaded refer to [Mic13m]. The next steps were “File” - “Open Executable” and selecting RegMe.exe. The target was loaded and breakpoint triggered in the `ntdll!LdrpDoDebuggerBreak` function.

```
CommandLine: C:\_dotNETPlayground\RegMe.exe
WARNING: Whitespace at end of path element
Symbol search path is: SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
(475c.5fa4): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=6c490000 edx=0012df08 esi=fffffffe edi=00000000
eip=778e0fab esp=003cf698 ebp=003cf6c4 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
778e0fab cc              int     3
```

[3] on my PC it is c:\Program Files (x86)\Debugging Tools for Windows (x86)

At this point the CLR was not loaded hence the helper extensions could not be loaded. A one-time breakpoint on the CLR JIT load event was set and execution was resumed:

```
sxe ld:clrjit
g
```

The breakpoint was hit:

```
(475c.5fa4): Unknown exception - code 04242420 (first chance)
ModLoad: 56400000 5646e000   C:\Windows\Microsoft.NET\Framework\v4.0.30319\clrjit.dll
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=7efdd000 edi=003ce6b0
eip=7785fc42 esp=003ce584 ebp=003ce5d8 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!NtMapViewOfSection+0x12:
7785fc42 83c404          add     esp,4
```

Now it was possible to load SOS and SOSEX extensions:

```
.loadby sos clr
.load sosex
```

WinDbg loaded a correct 32-bit version of SOS from the .NET Framework installation directory. The SOSEX was loaded from the debugger's local folder. Generally, no output is displayed if extensions are loaded successfully. Next, the information about the current location was viewed from the following stack trace:

```
0:000> !clrstack
OS Thread Id: 0x5fa4 (0)
Child SP       IP Call Site
003cf048 7785fc42 [PrestubMethodFrame: 003cf048] RegMe.Program.Main()
003cf1f4 7785fc42 [GCFrame: 003cf1f4]
```

Here is the application entry point function - `RegMe.Program.Main`, which is not yet compiled. Currently this is a stub that triggers the compilation as discussed in the previous chapter. However, the method name was available and was used to find method metadata information:

```
0:000> !name2ee *!RegMe.Program.Main
Module:      787d1000
Assembly:    mscorlib.dll
--------------------------------------
Module:      00182e94
Assembly:    RegMe.exe
Token:       0600000e
MethodDesc:  001837d8
Name:        RegMe.Program.Main()
Not JITTED yet. Use !bpmd -md 001837d8 to break on run.
```

As expected, this method is not JIT-compiled, yet fortunately it was possible to set a deferred breakpoint. The execution was resumed with the `g` command and the breakpoint was hit:

```
0:000> !bpmd -md 001837d8
MethodDesc = 001837d8
Adding pending breakpoints...

0:000> g
(475c.5fa4): CLR notification exception - code e0444143 (first chance)
JITTED RegMe!RegMe.Program.Main()
Setting breakpoint: bp 00270054 [RegMe.Program.Main()]
Breakpoint: JIT notification received for method RegMe.Program.Main() in AppDomain 00497a20.
Breakpoint 0 hit

EDIT: irrelevant lines of output were removed!
```

Display the disassembly of generated native code:

```
0:000> !u 00270054
Normal JIT generated code
RegMe.Program.Main()
Begin 00270050, size 2d
*** WARNING: Unable to verify checksum for RegMe.exe
*** ERROR: Module load completed but symbols could not be loaded for RegMe.exe
00270050 55              push    ebp
00270051 8bec            mov     ebp,esp
00270053 56              push    esi
>>> 00270054 e87366d004      call    System_Windows_Forms_ni+0x1966cc (04f766cc)
(System.Windows.Forms.Application.EnableVisualStyles(),
mdToken: 06000610)
00270059 33c9            xor     ecx,ecx
0027005b e82467d004      call    System_Windows_Forms_ni+0x196784 (04f76784)
(System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(Boolean),
mdToken: 0600061c)
00270060 b92c641800      mov     ecx,18642Ch (MT: RegMe.MainForm)
00270065 e81cdb210f      call    clr!JIT_NewCrossContext (0f48db86)
0027006a 8bf0            mov     esi,eax
0027006c 8bce            mov     ecx,esi
0027006e e8f5bff1ff      call    0018c068 (RegMe.MainForm..ctor(), mdToken: 06000001)
00270073 8bce            mov     ecx,esi
00270075 e81224d104      call    System_Windows_Forms_ni+0x1a248c (04f8248c)
(System.Windows.Forms.Application.Run(System.Windows.Forms.Form),
mdToken: 0600061a)
0027007a 5e              pop     esi
0027007b 5d              pop     ebp
0027007c c3              ret
```

This assembly is easier to read since the all calls are selfexplanatory, but what about the CIL? `!muf` command outputs the CIL. By default it is in mixed mode with assembly code, but the `-il` switch limits the output only to the CIL. The same method in CIL:

```
0:000> !muf -il 00270054
RegMe.Program.Main(): void
>>>>IL_0000: call System.Windows.Forms.Application::EnableVisualStyles
    IL_0005: ldc.i4.0
    IL_0006: call System.Windows.Forms.Application::SetCompatibleTextRenderingDefault
    IL_000b: newobj RegMe.MainForm::.ctor()
    IL_0010: call System.Windows.Forms.Application::Run
    IL_0015: ret
```

Clean and compact code. Apparently this was a default WinForms startup code and nothing related to the registration check was present. The code creates an instance of `RegMe.MainForm` type which represents a Form. Consequently, it was the next place to search. First of all, the `RegMe.MainForm` type constructor must be found and its CIL displayed:

```
0:000> !name2ee RegMe RegMe.MainForm..ctor
Module:      00182e94
Assembly:    RegMe.exe
Token:       06000001
MethodDesc:  00186358
Name:        RegMe.MainForm..ctor()
Not JITTED yet. Use !bpmd -md 00186358 to break on run.

0:000> !muf 00186358
No native code.  The method has not been JIT compiled.
RegMe.MainForm..ctor(): void
Source information not available.
IL_0000: ldarg.0
IL_0001: call System.Windows.Forms.Form::.ctor
IL_0006: ldarg.0
IL_0007: call RegMe.MainForm::InitializeComponent()
IL_000c: ldarg.0
IL_000d: newobj RegMe.Register::.ctor()
IL_0012: call RegMe.MainForm::set_RegForm(System.Windows.Forms.Form)
IL_0017: ldarg.0
IL_0018: call RegMe.MainForm::get_RegForm()
IL_001d: ldarg.0
IL_001e: callvirt System.Windows.Forms.Form::set_Owner
IL_0023: ret
```

Here, the `RegMe.MainForm` constructor calls the base `System.Windows.Forms.Form` constructor. Then the `InitializeComponent` method is called. The .NET WinForms developers know that this method is automatically maintained by Visual Studio's Windows Forms Designer and responsible for creating child controls and assigning properties. Next the `RegMe.Register` type instance is created and assigned to the `RegMe.MainForm.RegForm` instance property. Finally, the `RegMe.MainForm` object instance is assigned to the `Owner` property of the `RegMe.Register` instance. In other words, the `RegMe.MainForm` type constructor creates another Form instance of the type `RegMe.Register` and assigns itself as the owner, i.e. parent - child form relationship. Probably, the child form is the registration dialog we had already seen. There were no traces of the registration checks so far. The MSDN documentation for the `System.Windows.Forms.Form` class has a list of events. The most interesting is the `Load` event that occurs before the form is displayed for the first time. If there is a callback it must be attached in the `InitializeComponent` method. The following CIL of the `InitializeComponent` method was to be analyzed:

```
0:000> !name2ee RegMe RegMe.MainForm.InitializeComponent
Module:      00182e94
Assembly:    RegMe.exe
Token:       0600000d
MethodDesc:  001863e4
Name:        RegMe.MainForm.InitializeComponent()
Not JITTED yet. Use !bpmd -md 001863e4 to break on run.

0:000> !muf -il 001863e4
RegMe.MainForm.InitializeComponent(): void
EDIT: not relevant lines of output were removed!
IL_0488: ldarg.0
IL_0489: ldarg.0
IL_048a: ldftn RegMe.MainForm::MainForm_Load(object, System.EventArgs)
IL_0490: newobj System.EventHandler::.ctor
IL_0495: call System.Windows.Forms.Form::add_Load
EDIT: not relevant lines of output were removed!
```

`RegMe.MainForm::MainForm_Load` method was subscribed to that event and its content was viewed:

```
0:000> !name2ee RegMe RegMe.MainForm.MainForm_Load
Module:      00182e94
Assembly:    RegMe.exe
Token:       06000003
MethodDesc:  00186370
Name:        RegMe.MainForm.MainForm_Load(System.Object, System.EventArgs)
Not JITTED yet. Use !bpmd -md 00186370 to break on run.

0:000> !muf -il 00186370
RegMe.MainForm.MainForm_Load(object, System.EventArgs): void
IL_0000: ldarg.0  (sender)
IL_0001: call RegMe.Security::LoadAndCheckRegistrationInformation()
IL_0006: call RegMe.MainForm::set_IsRegistered(bool)
IL_000b: ldarg.0  (sender)
IL_000c: call RegMe.MainForm::SetupUI()
```

```
IL_0011: ret
```

Finally, the registration check - `RegMe.Security::LoadAndCheckRegistrationInformation` method call was found. The result was saved into a boolean property named `IsRegistered`. Then the `RegMe.MainForm::SetupUI` method was called. I speculated that this method was responsible for the registered/unregistered UI state adjustment based on the `IsRegistered` property value. This assumption was confirmed by viewing the method's CIL:

```
0:000> !name2ee RegMe RegMe.MainForm.SetupUI
Module:      00182e94
Assembly:    RegMe.exe
Token:       06000002
MethodDesc:  00186364
Name:        RegMe.MainForm.SetupUI()
Not JITTED yet. Use !bpmd -md 00186364 to break on run.

0:000> !muf -il 00186364
RegMe.MainForm.SetupUI(): void
    loc 0:bool
    loc 1:bool

EDIT: content truncated!

IL_0033: ldarg.0
IL_0034: call RegMe.MainForm::get_IsRegistered()
IL_0039: brtrue.s IL_0042
IL_003b: ldstr "UNREGISTERED"
IL_0040: br.s IL_0047
IL_0042: ldstr "REGISTERED"
IL_0047: callvirt System.Windows.Forms.Control::set_Text

EDIT: content truncated!
```

Setting a breakpoint on the `RegMe.MainForm.MainForm_Load` method after it was compiled and viewing assembly:

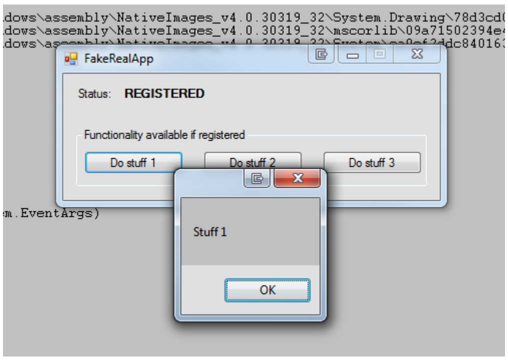
```
0:000> !bpmd RegMe RegMe.MainForm.MainForm_Load
Found 1 methods in module 00182e94...
MethodDesc = 00186370
Adding pending breakpoints...

0:000> g
(475c.5fa4): CLR notification exception - code e0444143 (first chance)
JITTED RegMe!RegMe.MainForm.MainForm_Load(System.Object, System.EventArgs)
Setting breakpoint: bp 00271016 [RegMe.MainForm.MainForm_Load(System.Object,
                                                              System.EventArgs)]
Breakpoint: JIT notification received for method RegMe.MainForm.MainForm_Load(
                            System.Object, System.EventArgs) in AppDomain 00497a20.
Breakpoint 1 hit

0:000> !u 00271016
Normal JIT generated code
RegMe.MainForm.MainForm_Load(System.Object, System.EventArgs)
Begin 00271010, size 21
00271010 55              push    ebp
00271011 8bec            mov     ebp,esp
00271013 56              push    esi
00271014 8bf1            mov     esi,ecx
>>> 00271016 ff1558741800    call    dword ptr ds:[187458h] (RegMe.Security
                          .LoadAndCheckRegistrationInformation(), mdToken: 06000014)
0027101c 0fb6d0          movzx   edx,al
0027101f 889660010000    mov     byte ptr [esi+160h],dl
00271025 8bce            mov     ecx,esi
00271027 e834b1f1ff      call    0018c160 (RegMe.MainForm.SetupUI(), mdToken: 06000002)
0027102c 5e              pop     esi
0027102d 5d              pop     ebp
0027102e c20400          ret     4
```

The return value of the static method call was copied from the `al` register to the `edx` register and then stored in the `esi+0x160`, which is the aforementioned property. Thus the `esi` register contained *this* pointer of the type `RegMe.MainForm`. The object information at a specified address by the `esi` register was viewed via the `!do esi` command and the statement was confirmed. My next step was to alter the method's behavior. The plan was to skip the registration check call completely and set the `al` register to 1 as this call would normally do if the registration was valid. The current call instruction is six bytes in length and modifications must remain within this boundary:

```
0:000> a 00271016
mov al, 1
nop
nop
nop
nop
EDIT: press enter to exit edit mode

0:000> !u 00271016
Normal JIT generated code
RegMe.MainForm.MainForm_Load(System.Object, System.EventArgs)
Begin 00271010, size 21
00271010 55              push    ebp
00271011 8bec            mov     ebp,esp
00271013 56              push    esi
00271014 8bf1            mov     esi,ecx
```

```
>>> 00271016 b001            mov     al,1
00271018 90             nop
00271019 90             nop
0027101a 90             nop
0027101b 90             nop
0027101c 0fb6d0         movzx   edx,al
0027101f 889660010000   mov     byte ptr [esi+160h],dl
00271025 8bce           mov     ecx,esi
00271027 e834b1f1ff     call    0018c160 (RegMe.MainForm.SetupUI(), mdToken: 06000002)
0027102c 5e             pop     esi
0027102d 5d             pop     ebp
0027102e c20400         ret     4
```

If everything was correct and there were no additional checks, the application should load as if it was registered. Execution was resumed by typing in the character `g` and pressing the Enter key (or just F5). Success!



```
.dows\assembly\NativeImages_v4.0.30319_32\System.Drawing\78d3cd0
.dows\assembly\NativeImages_v4.0.30319_32\mscorlib\09a71502394e
.dows\as                                              ddc840163
m.EventArgs)
```

FakeRealApp

Status: REGISTERED

Functionality available if registered

Stuff 1

Figure 3.4: Patching at runtime. Application was tricked into believing it is registered

The entire analysis was rather easy as the SOS and SOSEX extensions were very helpful. Moreover, I only used less than ten percent of all available commands. Besides the commands used earlier, the following commands are especially valuable:

- `!DumpDomain` - displays details of all application AppDomains
- `!DumpModule <module address>` - prints information about .NET module. `-mt` command option lists all declared and referenced types by module
- `!DumpMT <method table address>` - prints method table details for specified address. `-md` command option lists object methods
- `!DumpMD <method descriptor address>` - displays method descriptor information
- `!DumpStackObjects` - displays information about all managed objects on the current stack

- `!IP2MD <code address>` - very useful. Finds method descriptor by address
- `!DumpVC <method table address> <address>` - displays information about value type. Reference type counterpart is `!do <object address>` command
- `!DumpIL <method descriptor address or dynamic method address>` - similar to `!muf -il <address>` and outputs method's CIL and CIL address

Refer to [Mic13l] for further details on commands. Debugging is essential when a static analysis does not fulfill the requirements or is simply too complicated. Debugging allows seeing the object types and their values at runtime, analyzing the control flow, performing modifications if needed, and more. Certain protectors create types and operate with values at runtime. In that case a sole static analysis is not efficient, if feasible at all. A combination of a static and dynamic analysis yields the best results.

The conclusive step was to see the CIL of the method that performs the actual username and password validation. However, I decided to leave it for the next subchapter dedicated to static analysis.
There are several other debuggers worth mentioning:

- Microsoft MDbg is a .NET command-line debugger. Old source code is available on Microsoft website
- Visual Studio debugger. It is possible to debug .NET applications without source code with SOS plugin
- SmidgeonSoft PEBrowser debugger. Yet another powerful .NET-aware debugger. Software and documentation are available on the Internet [Smi13]
- Dotnet IL Editor (DILE). Fantastic open source .NET debugger! In spite of minor downsides ability to view the objects in memory as real .NET entities is simply mind-blowing. I use it to analyze the InishTech SLP protector later in this chapter. Details can be found on the web [Zso13]

3.1.3 Static analysis and application modification

This subchapter describes the CIL disassemblers and decompilers along with the CIL assembler and how to use them to modify an existing .NET executable. The previous subchapter identified how direct assembly code manipulation at runtime can trick an application to assume it is registered. Unfortunately, all changes are lost when the application is restarted, and the entire procedure must be repeated. This is not a viable long term solution, as the performed modifications should persist.

Tools used:

- ILDASM - CIL disassembler is part of Windows SDK
- ILASM - CIL assembler is installed with .NET Framework
- .NET Reflector - very popular CIL disassembler and decompiler. Various third-party extensions are available
- Reflexil - .NET assembly editor plugin for Reflector and Telerik's JustDecompile
- CFF Explorer - powerful PE editor with full .NET file format support

3.1.4 CIL round-tripping

The CIL round-tripping [Sta13] is a procedure where the .NET binary is first disassembled to the CIL, after which the CIL is modified and finally compiled back into the binary.

To begin with, the ILDASM was started, and RegMe.exe was selected via the "File" - "Open" dialog. The analysis was no doubt facilitated by the GUI: all the declared types were well-organized in a tree structure which was easy to navigate. The startup registration check, as it was discovered, is performed in the `RegMe.MainForm.MainForm_Load` method. After locating it in the tree structure, I double clicked it:

```
.method private hidebysig instance void
MainForm_Load(object sender, class [mscorlib]System.EventArgs e) cil managed
{
  // Code size       18 (0x12)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       bool RegMe.Security::LoadAndCheckRegistrationInformation()
  IL_0006:  call       instance void RegMe.MainForm::set_IsRegistered(bool)
  IL_000b:  ldarg.0
  IL_000c:  call       instance void RegMe.MainForm::SetupUI()
  IL_0011:  ret
} // end of method MainForm::MainForm_Load
```

This CIL snippet has already been encountered in the previous subchapter. There definitely was a temptation to view the CIL of the `RegMe.Security::LoadAndCheckRegistrationInformation` method and find the actual validation routine given the simplicity of the GUI navigation. It was not, however, the priority at that point. The next step was to disassemble the entire application. I used the "Developer Command Prompt for VS2012"[4] which is available if a commercial version of Visual Studio is installed

[4] this is regular command prompt with configured environmental settings for various SDK and VS tools

from the path “Start menu” - “Microsoft Visual Studio 2012” - “Visual Studio Tools”. It is a matter of convenience rather than a requirement.

```
ildasm RegMe.exe /out=RegMe.il
```

Several output files were created. After opening RegMe.il in a text editor, a `MainForm_Load` method inside the `RegMe.MainForm` class was found. Registration check call was substituted by pushing the integer 1, which is treated as the boolean “true” by the CIL, on the stack. The labels were also updated to the correct values based on instruction length. Here is the new method body:

```
IL_0000:  ldarg.0
IL_0001:  ldc.i4.1
IL_0002:  call       instance void RegMe.MainForm::set_IsRegistered(bool)
IL_0007:  ldarg.0
IL_0008:  call       instance void RegMe.MainForm::SetupUI()
IL_000d:  ret
```

After saving the file, the application was assembled back:

```
ilasm /res=RegMe.res RegMe.il
```

Running the newly generated RegMe.exe was successful. The application started in its registered state.

3.1.5 Byte-patching

Byte-patching is also applicable to the .NET executables. However, it is not a convenient method to modify the .NET binary. It is best suited for a very small changes. If the patching offset is not known, the sequence of bytes that represent the CIL opcodes must be searched. In the previous subchapter `MainForm_Load`’s CIL was presented using the `!muf -il` command. This time the `!DumpIL <method descriptor>` command was used:

```
ilAddr = 012720e5
IL_0000: ldarg.0
IL_0001: call RegMe.Security::LoadAndCheckRegistrationInformation
IL_0006: call RegMe.MainForm::set_IsRegistered
IL_000b: ldarg.0
IL_000c: call RegMe.MainForm::SetupUI
IL_0011: ret
```

The output was familiar, but it included an important difference: the CIL address. The total length of the instruction stream was IL_0011 + one byte `ret` opcode = 0x12 (decimal 18) bytes. Outputting the CIL opcodes resulted in:

```
0:000> db 012720e5
012720e5  4a 02 28 14 00 00 06 28-09 00 00 06 02 28 02 00  J.(....(.....(..
012720f5  00 06 2a 3a 02 28 0a 00-00 06 02 6f 16 00 00 0a  ..*:.(.....o....
```

The `db` command is used to display byte values at a specified memory location. The x86 is a little-endian format architecture i.e.: the bytes that are stored on the disk as 0x12345678 are loaded to memory with the last byte being the most significant: 0x78563412. In contrast to the other `d*` commands that group bytes, per-byte output of the `db` command does not require a manual right to left byte swap. 0x012720e5 + 18 length are method bytes with CIL body opcodes starting at 0x012720e6:

```
4a 02 28 14 00 00 06 28 09 00 00 06 02 28 02 00 00 06 2a
```

RegMe.exe was opened in a hex editor and the desired byte sequence was searched. It was encountered once, and then the actual registration function call represented by 5 bytes 2814000006 were replaced with the opcode byte 17, which is the CIL ldc.i4.1 instruction, and 4 additional zero bytes:

```
4A 02 17 00 00 00 00 28 09 00 00 06 02 28 02 00 00 06 2A
```

The CIL opcode 00 is *nop* (no operation) instruction[5]. The changes were saved and the application started in its registered state.

Some CIL disassemblers have an option to display raw bytes next to the CIL disassembly. The "View" - "Show bytes" check enables this option in ILDASM:

```
// Method begins at RVA 0x20e5
// Code size       18 (0x12)
.maxstack  8
IL_0000:  /* 02   |             */ ldarg.0
IL_0001:  /* 28   | (06)000014  */ call    bool RegMe.Security::LoadAnd
CheckRegistrationInformation()
IL_0006:  /* 28   | (06)000009  */ call    instance void RegMe.MainForm
::set_IsRegistered(bool)
IL_000b:  /* 02   |             */ ldarg.0
IL_000c:  /* 28   | (06)000002  */ call    instance void RegMe.MainForm
::SetupUI()
IL_0011:  /* 2A   |             */ ret
```

Note the byte order after the "|" symbol: all bytes after the vertical bar must be reversed in order for them to be searchable on the disk. ILDASM also provides a valuable detail: a method start RVA where the raw CIL bytes are located. Another notable detail is the correlation between the CIL location in memory and the method's RVA:

[5] similar to opcode 90 - a *nop* for x86 used earlier

```
ilAddr = 0x012720e5
RVA    = 0x000020e5
```

To find the method CIL bytes the RVA must be translated to the file offset. The RVA belongs to the range of a concrete PE section which must be found first. RegMe is a regular .NET application. According to the .NET file structure described in the previous chapter, the managed code is contained inside the *.text* PE section. The `dumpbin` utility or any other PE viewer can be used to view section details. I used the CFF Explorer to view the *.text* section properties:

- Virtual size - 0x00002D04
- Virtual address - 0x00002000
- Raw size - 0x00002E00
- Raw address - 0x00000200

Any PE section has these characteristics. *Virtual ...* are used by the OS PE loader when a section is mapped to memory, whereas the *Raw ...* are applicable to the PE file on a disk. In order to convert the RVA to a file offset:

```
Method RVA - Virtual address + Raw address
0x000020e5 - 0x00002000 + 0x00000200 = 0x2E5
```

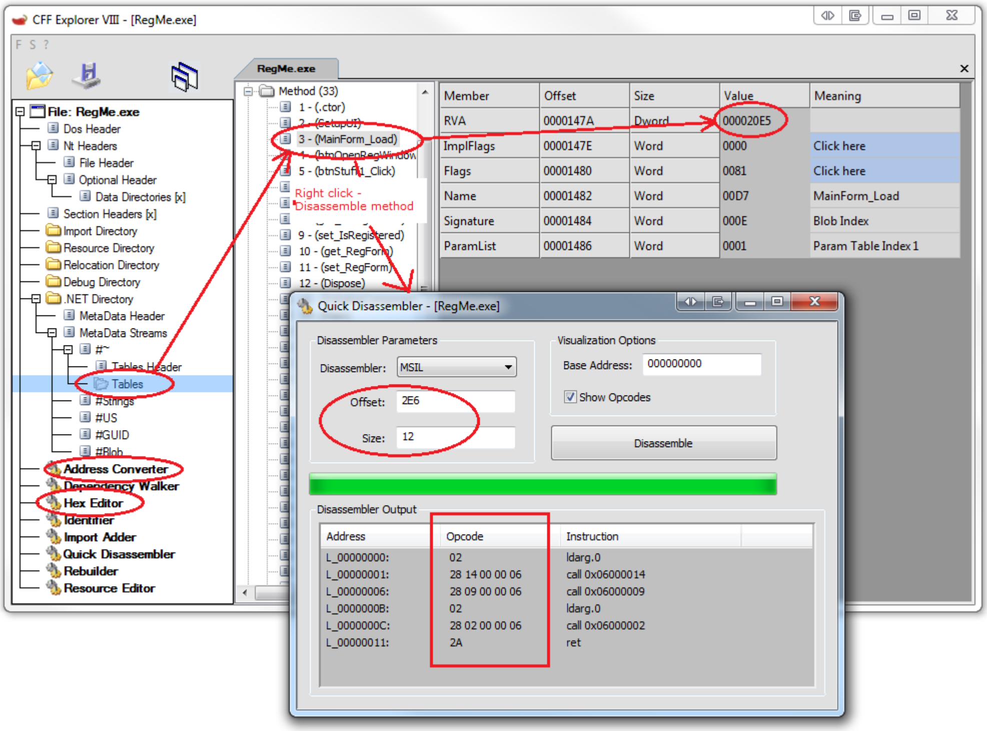
The bytes at the file offset 0x2E6[6] match their counterpart displayed by ILDASM. The CFF Explorer has a built-in “Address Converter” that eliminates the necessity for manual address calculations. In addition, it has a hex editor and a built-in disassembler. Finally, the `MainForm_Load` method can be easily found in the “.NET Directory”, as illustrated by Figure 3.5.

3.1.6 Patching with decompiler and Mono.Cecil

Due to its convenience, this is the most anticipated .NET assembly modification approach amongst everything presented so far. There were several decompilers available:

- .NET Reflector - the most popular one and I suspect it is the first of its kind. It was freeware even when it was acquired by RedGate but later new owner ceased the free version what in turn has motivated the community to develop freeware alternatives. It is capable to export .NET assembly as a VS source code project. Its main outstanding feature compared to the competitors is

[6] 0x2E5 + 1 byte to the actual CIL body



Right click -
Disassemble method

Member	Offset	Size	Value	Meaning
RVA	0000147A	Dword	000020E5	
ImplFlags	0000147E	Word	0000	Click here
Flags	00001480	Word	0081	Click here
Name	00001482	Word	00D7	MainForm_Load
Signature	00001484	Word	000E	Blob Index
ParamList	00001486	Word	0001	Param Table Index 1

Disassembler: MSIL

Offset: 2E6

Size: 12

Base Address: 000000000

☑ Show Opcodes

```
Address        Opcode            Instruction
L_00000000:    02                ldarg.0
L_00000001:    28 14 00 00 06    call 0x06000014
L_00000006:    28 09 00 00 06    call 0x06000009
L_0000000B:    02                ldarg.0
L_0000000C:    28 02 00 00 06    call 0x06000002
L_00000011:    2A                ret
```

Figure 3.5: CFFExplorer navigation and modification capabilities

the ability to *step into* any third-party .NET assembly during debugging in VS. Target .NET assembly is automatically decompiled thus no source code is required

- ILSpy - popular, free and open source
- Telerik JustDecompile - Telerik is mainly famous for its .NET component libraries, however, recently they have developed their own freeware decompiler. Its features are similar to the .NET Reflector
- JetBrains dotPeek - freeware
- DevExtras .NET CodeReflect - freeware
- Spices .Net Decompiler - commercial
- Assembly Analyzer - freeware and open source. Almost a .NET Reflector clone

The .NET Reflector was selected. The Reflexil plugin for the .NET Reflector was a decent candidate for the .NET assembly editor's role. In general, most of the editors utilize Mono.Cecil for performing the modifications. Mono.Cecil is a library developed by Jb Evain for generating and analyzing .NET binaries. It is used in many tools such as the aforementioned Reflexil plugin, de4dot[7], Mono Debugger etc.

In the beginning of this chapter I mentioned that .NET applications were considered to be open source and easily decompilable, but everything presented up to this point contradicts that statement. It is time to "unleash" the decompiler![8]. First, the .NET Reflector was started, then, via "File" - "Open Assembly", the RegMe.exe was selected. I selected C# in the language toolbar dropdown, navigated to the RegMe namespace, then the MainForm class, and selected the MainForm_Load method. The following C# code was displayed:

```
private void MainForm_Load(object sender, EventArgs e)
{
    this.IsRegistered = Security.LoadAndCheckRegistrationInformation();
    this.SetupUI();
}
```

Clicking on LoadAndCheckRegistrationInformation revealed the method's source code:

```
public static bool LoadAndCheckRegistrationInformation()
{
    try
    {
        string userName = Encoding.UTF8.GetString(Convert.FromBase64String(
                                                Settings.Default.UserName));
        string password = Encoding.UTF8.GetString(Convert.FromBase64String(
                                                Settings.Default.Password));
        return CheckRegInformation(userName, password);
    }
    catch
    {
        return false;
    }
}
```

In this case, both the username and password were stored in the application settings, in the XML configuration file. Moreover, strings are Base64 encoded. Finally, a click on the CheckRegInformation revealed the long-awaited registration validation method:

```
public static bool CheckRegInformation(string userName, string password)
{
    if ((string.IsNullOrEmpty(userName) || string.IsNullOrEmpty(password)) ||
                                                        (userName.Length < 6))
    {
```

[7] up to version 2.0.0

[8] greetings from "Pirates of the Caribbean" movie :)

```
        return false;
    }
    int num = 0;
    int num2 = 0;
    for (int i = 0; i < 6; i += 2)
    {
        num = userName[i];
        num2 = userName[i + 1];
    }
    int num4 = (num * 0x3e8) + num2;
    return string.Equals(password, num4.ToString(), StringComparison.OrdinalIgnoreCase);
}
```

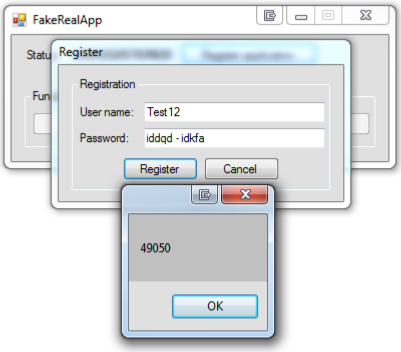
Compare this output to C pseudo-code discovered in the subchapter 3.1.1. The goal is the same, but was achieved by two distinctive ways: several clicks and perfect readability versus time consuming debugging and complex analysis of assembly code chunks. Certainly manual decompilation of the CIL would have been easier, however, my intention was to present the contrast between the RCE benefits of easily decompilable managed applications and the absence of such benefits for their unmanaged counterparts.

Next, a stand-alone application can be developed for key generation purposes or an existing application can serve that purpose. I decided to proceed with the latter. I loaded the editor plugin via "Tools" - "Reflexil v1.6". The plugin UI displayed CIL instructions on a per row basis. The context menu revealed two editing options: editing per CIL instruction or entirely replacing the current type's CIL with C# or VisualBasic code. After editing the CIL per instruction, the final result was:

```
71 nop
72 ldloca.s -> (3)  (System.Int32)
74 call System.String System.Int32::ToString()
79 call System.Windows.Forms.DialogResult System.Windows.Forms.MessageBox::Show(
                                                                     System.String)
84 pop
85 ldc.i4.0
86 ret
```

The following steps were performed: the .NET assembly root tree item ("RegMe (1.0.0.0)" - "Reflexil v1.6" - "Save as") was right clicked and the patched binary was saved. The RegMe.Patched.exe was ran, and the "Register application" button was clicked. In the registration dialog, "Test12" was entered as the username and the password was left to be whatever. The introduced modifications had converted this application into a key generator. When the "Register" button was clicked, the correct password was calculated and displayed in a message box as illustrated by

Figure 3.6. Furthermore, the method always returned the boolean “false”, meaning that the application can never be registered.



User name: Test12

Password: iddqd - idkfa

49050

Figure 3.6: The original application converted into a key generator

With some additional minor changes the application can be converted into a better looking key generator, but this task was left to the reader, as, after all, the primary goal was achieved.

A reasonable question: what was the necessity to describe several complicated solutions when the decompiler can instantly provide easily readable C`#` source code without any troubles? Because it cannot be expected to always do so. The beginning of this chapter introduced native code wrapper protection, CIL code injection based protection and obfuscators. If some counter-measures diminish the value of the decompiler, others can shut it down completely. In addition, de4dot may not have the support for the encountered obfuscator or ILDASM may reject to open the binary because the `SuppressIldasmAttribute` attribute is set[9]. In fact, the .NET Reflector ignores that attribute yet it may not be able to disassemble due to *some* error and only ILSpy may be capable to display more or less accurate CIL, whereas the decompilation will fail. If the CIL round-tripping is not possible or reasonable, then the Mono.Cecil library is a good candidate. However, despite Mono.Cecil’s powerful features it did not satisfy the needs of the de4dot utility and was substituted with a homebrew dnlib library which better suited the task of processing heavily obfuscated .NET assemblies. The ability to “handle” .NET binaries **only** when the decompiler is capable to output high-level language source code is not an RCE skill. To summa-

[9] easy to overcome

rize, every task has its own set of applicable tools and the desired work principle is to achieve better results with less effort.

3.2 Part 2: Analysis of the InishTech SLP software protection

This part is dedicated to the InishTech SLP[10] analysis. At first, the intention was to write a detailed analysis. After all, the thesis is about the .NET RCE, but an instant second thought prevented this from happening. In order to not endanger thousands of protected applications by providing sensitive details, the analysis was performed in a scope sufficient for removing the protector in the simplest case. This limitation will definitely not affect moderately skilled reverse engineers, but will raise the entry level for their beginner-level counterparts. Finally, the plan was to not modify the protector itself to skip certain validations, but rather attempt to recover the CIL and untie the application from the protector.
Tools used:

- WinDbg with SOS and SOSEX
- .NET Reflector with Reflexil plugin
- Dotnet IL Editor (DILE) debugger
- ANTS Performance Profiler - .NET profiler

3.2.1 Initial analysis

The vendor's product overview brochure is an excellent source of information about the protector:

"Software Potential allows the ISV to create and package editions on demand, according to the requirements and the opportunities they see in the marketplace. Editions can have distinct feature sets and can be subject to particular time and usage-based limitations.

Each of the three traditional methods for protecting code - obfuscation, encryption, and code splitting - has strengths and weaknesses, and the technique used by the Software Potential service takes the best from each method to provide a more effective, convenient software-only solution. The fundamental strategy used by Software Poten-

[10] previously named Microsoft SLPS which was acquired from Microsoft in 2009

tial is to take selected code, pass it through a one-way transformation that leaves the original logic effectively encrypted and obfuscated, and then package it direct execution on the CLR.

Key to the Software Potential code protection strategy is code transformation. Using the Software Potential Code Protector application compiled MSIL is transformed into Secure Execution Engine Language (SEEL) which can no longer be run directly by the CLR or reverse engineered by decompilation tools, and then encrypted so that the resulting SEEL "byte code" cannot be directly inspected. Decompilation of Microsoft .NET code is possible because MSIL is a fixed and widely known specification. The SEEL command set, on the other hand, is unique - changing with each unique Software Potential Permutation - and requires a unique Agent to execute it. A combination of the SEEL and the Agent comprises a vendor Permutation. If SEEL was simply another "byte code" made up of instructions to be executed, a very skilled hacker could, theoretically, spend the time to reverse engineer the entire command set and use this knowledge to build a decompiler; however, because the SEEL is actually encrypted, there is a huge barrier to even that already complex task. Further, because each Agent is different, any decompiler would only be effective against a specific Agent associated with that specific permutation. Different applications from the same vendor could use different permutations to ensure that any compromise of security is limited to a single application. This layered approach makes Software Potential's code transformation a premier code protection technology. This ability to create an encrypted, unique, and transformed version of your code is a powerful protection against reverse engineering, and simple to achieve. After building your .NET application, use Code Protector to identify the methods you want to protect. Because there is a performance cost to decrypting and executing protected code, it is best to transform only those methods that contain sensitive intellectual property, information about your enterprise infrastructure, passwords, etc. After protection is complete, the protected code, when viewed in a .NET decompiler, shows only a call into the SVM with a string of random characters. When your application is executed, the CLR handles the original unprotected MSIL, but the SVM executes the protected SEEL. The Agent is a series of DLLs that link directly into your .NET application and works as an integral part of your application logic, thus making it more difficult to bypass licensing routines." [Ini13]

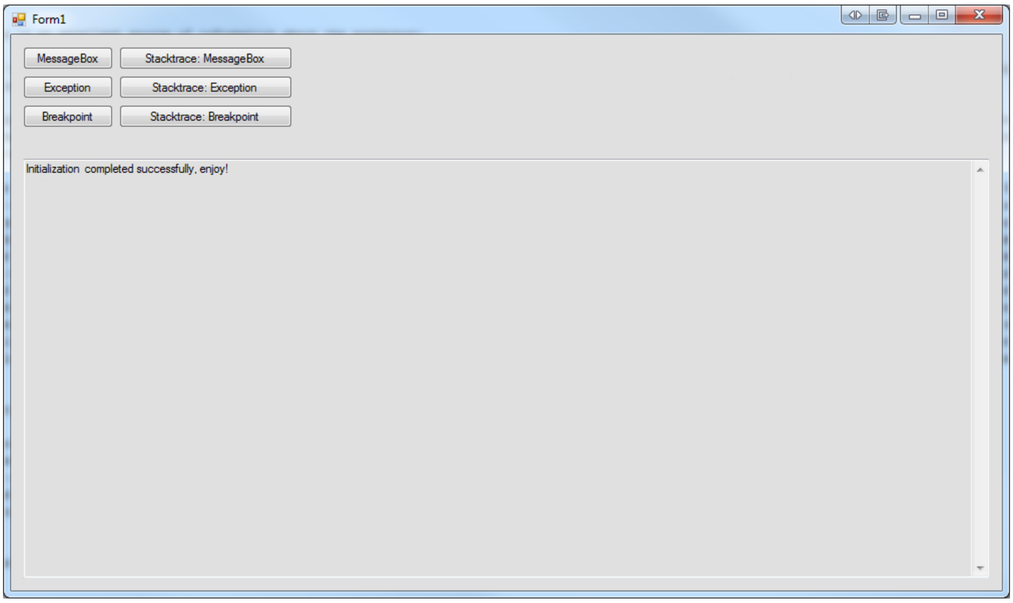
And a short summary by the very same brochure [Ini13]:

- the logic of protected methods is removed and replaced with a call into the

SVM, so traditional .NET decompilers cannot reverse engineer protected code

- the SEEL is encrypted to make reverse engineering the new “byte code” even more difficult
- because each Permutation is unique, any reverse engineering that was accomplished would be limited to that single Permutation.
- the original code never exists fully in memory
- because licensing is based on the process of executing protected functions, the SVM is an integral part of enforcing your licenses

Another application[11] was developed for the initial analysis illustrated by Figure 3.7



```
Initialization completed successfully, enjoy!
```

Figure 3.7: Helper application

The application’s output is displayed in the large text box control in the middle. The six buttons on the left represent the protected functionality. The buttons in the first column are selfexplanatory:

```
private void btnMessageBox_Click(object sender, EventArgs e)
{
  MessageBox.Show("Hello world!");
}
```

[11] also compiled as .NET 4.0 32-bit application

```
private void btnException_Click(object sender, EventArgs e)
{
  throw new ApplicationException("ApplicationException");
}

private void btnBreakpoint_Click(object sender, EventArgs e)
{
  Debugger.Break();
}
```

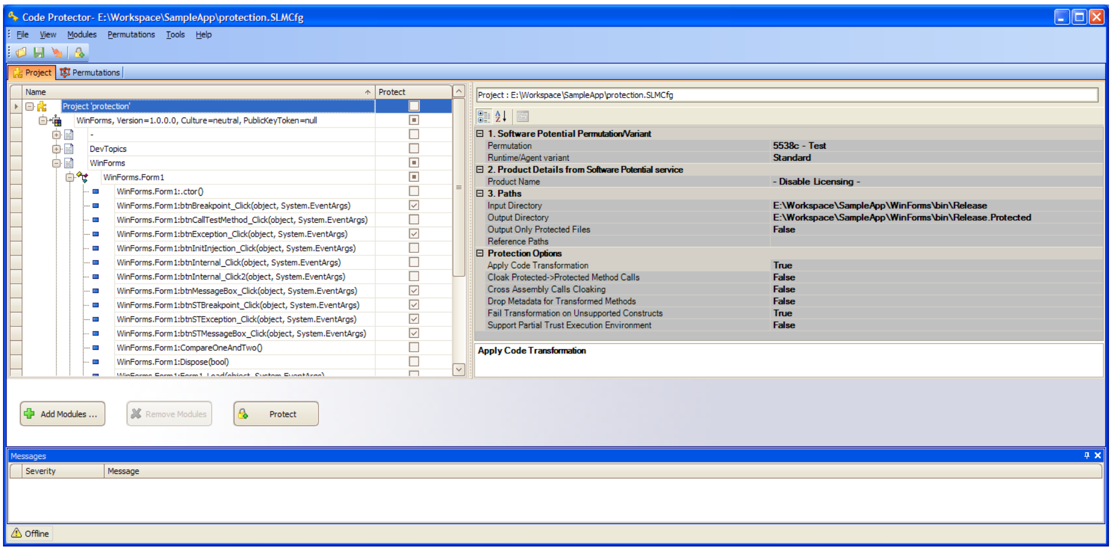
The second column counterparts were otherwise identical, except for the added stack tracing for providing method execution details:

```
private void btnSTMessageBox_Click(object sender, EventArgs e)
{
  StackTrace stackTrace = new StackTrace();
  StackFrame[] frames = stackTrace.GetFrames();

  for (int i = 0; i < stackTrace.FrameCount; i++)
  {
    Log(string.Format("\t--- StackFrame #: {0} - {1} ---", i, DateTime.Now.ToString()));
    StackFrame sf = stackTrace.GetFrame(i);

    CallingMethod tmp = new CallingMethod(sf);
    ShowCallingMethod(tmp);
  }

  MessageBox.Show("Hello world!");
}

EDIT: other methods follow here
```

The trial version the InishTech SLP was downloaded, the Code Protector was started, the trial permutation file was loaded, and the compiled WinForms.exe binary was opened. After navigating through the types tree, selecting the six aforementioned event handlers, and clicking on the “Protect” button, the Protector output several files into a new folder `Release.Protected`:

- WinForms.exe
- Microsoft.Licensing.Permutation_5538c_2.0.dll - 909,5KB
- Microsoft.Licensing.Runtime2.0.dll - 88KB
- Microsoft.Licensing.Utils2.0.dll - 120KB

Opening the WinForms.exe in the .NET Reflector and viewing protected methods showed that they all had the same body, except for the GUID value parameter:



Name	Protect
Project 'protection'	□
WinForms, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null	▣
-	□
DevTopics	□
WinForms	▣
WinForms.Form1	▣
WinForms.Form1:.ctor()	□
WinForms.Form1:btnBreakpoint_Click(object, System.EventArgs)	☑
WinForms.Form1:btnCallTestMethod_Click(object, System.EventArgs)	□
WinForms.Form1:btnException_Click(object, System.EventArgs)	☑
WinForms.Form1:btnInitInjection_Click(object, System.EventArgs)	□
WinForms.Form1:btnInternal_Click(object, System.EventArgs)	□
WinForms.Form1:btnInternal_Click2(object, System.EventArgs)	□
WinForms.Form1:btnMessageBox_Click(object, System.EventArgs)	☑
WinForms.Form1:btnSTBreakpoint_Click(object, System.EventArgs)	☑
WinForms.Form1:btnSTException_Click(object, System.EventArgs)	☑
WinForms.Form1:btnSTMessageBox_Click(object, System.EventArgs)	☑
WinForms.Form1:CompareOneAndTwo()	□
WinForms.Form1:Dispose(bool)	□

Project : E:\Workspace\SampleApp\protection.SLMCfg

1. Software Potential Permutation/Variant	
Permutation	5538c - Test
Runtime/Agent variant	Standard
2. Product Details from Software Potential service	
Product Name	- Disable Licensing -
3. Paths	
Input Directory	E:\Workspace\SampleApp\WinForms\bin\Release
Output Directory	E:\Workspace\SampleApp\WinForms\bin\Release.Protected
Output Only Protected Files	False
Reference Paths	
Protection Options	
Apply Code Transformation	True
Cloak Protected->Protected Method Calls	False
Cross Assembly Calls Cloaking	False
Drop Metadata for Transformed Methods	False
Fail Transformation on Unsupported Constructs	True
Support Partial Trust Execution Environment	False

Apply Code Transformation

Severity	Message

Figure 3.8: Code Protector application ready to process WinForms.exe

```
[MethodImpl(MethodImplOptions.NoInlining)]
private void btnSTMessageBox_Click(object sender, EventArgs e)
{
  object[] args = new object[] { sender, e };
  SLMRuntime.ExecuteMethod(this, "D5A898FC66D2409CADD0D2C5896F2C8B", "", args, null, null);
}
```

The next step was to click on the `ExecuteMethod` method, and then on the `InternalExecuteMethod` method:

```
public static object InternalExecuteMethod(Assembly declaringAssembly, string SVMMethodId,
                                                      IExecutionEngineParams paramsReader)
{
  if (declaringAssembly == null)
  {
    throw new ArgumentNullException("declaringAssembly");
  }
  using (ISLMRuntime runtime = SLMRuntimeFactory.CachedForProtectedAssembly(
                                                                  declaringAssembly))
  {
    return runtime.ExecutionEngine.ExecuteMethod(declaringAssembly,
                                                           SVMMethodId, paramsReader);
  }
}
```

The .NET Reflector automatically navigated to the .NET assembly that contained the declaring type, the Microsoft.Licensing.Runtime2.0. A quick overview revealed the `DotfuscatorAttribute` class, an indicator of the Dotfuscator obfuscator. Many type names began with Microsoft1LicensingRuntime210 and were indeed obfuscated. Type names containing words “engine”, “runtime”, “invocation context”, “activation” and “status” hinted that this software could be a complicated execution engine based pipeline. Clicking on the `ISLMRuntime` allowed to view the interface type:

```
[Guid("FBAEF771-0D4E-42d4-9D63-627D7CCD7F02"), ComVisible(true)]
public interface ISLMRuntime : IDisposable
{
  ISLMLicenseStores LicenseStores { get; }
  ISLMLicenses Licenses { get; }
  ISLMStatus Status { get; }
  ISLMActivation Activation { get; }
  [ComVisible(false)]
  IExecutionEngine ExecutionEngine { get; }
  [ComVisible(false)]
  ISLMCustomizations Customizations { get; }
  string VendorName { get; }
  ILicenseSession Session { get; }
  bool IsSessionOpen { get; }
  void OpenSession(string product, string version);
  [Obsolete("as of 3.1.1919, invalidateHandler will no longer be called")]
  void OpenSession(string product, string version, LicenseSessionEvent invalidateHandler);
  void CloseSession();
}
```

The previously encountered `SLMRuntime` class implemented the interface. The class constructor contained obfuscated code and some initialization:

```
this.B = A_2 ? A(A_1, A_0) : new Microsoft1Licensing1Runtime210G(A_1, A_0);
```

The `Microsoft1Licensing1Runtime210G` type constructor created another type, `Microsoft1-Licensing1Runtime210R`, which called the `Microsoft1Licensing1Runtime210H.A` method that read “_SLM_HD.DAT” embedded resource bytes and passed it to another type’s constructor etc. The `IExecutionEngine` was another interesting interface:

```
[ComVisible(false)]
public interface IExecutionEngine
{
  // Methods
  object ExecuteMethod(Assembly methodAssembly, string svmMethodId,
                                                IExecutionEngineParams parameters);
}
```

The .NET Reflector has a great feature to discover type dependencies and their usage. The next steps were to right click the `IExecutionEngine` in the tree and to select “Analyse”. After that, “Used By” was expanded and the following was clicked:

```
Microsoft.Licensing.SLMRuntime.get_ExecutionEngine() : IExecutionEngine
```

A menu was opened with a right click and “Go To Member” was selected:

```
public IExecutionEngine get_ExecutionEngine()
{
  return this.B().ExecutionEngine;
}
```

The method `B` was clicked:

```
private ISLMRuntimeImpl B()
{
  return this.B.A();
}
```

Next, the `ISLMRuntimeImpl` was clicked. Some interface property fields were identical to the `ISLMRuntime` interface property fields. Supposedly, *Impl* meant *implementation*. Several property fields and methods returned this type, and a particularly interesting method was contained in the `Microsoft1LicensingRuntime210R` type:

```
private static ISLMRuntimeImpl B(Assembly A_0)
{
  Type type;
  ISLMRuntimeImpl impl;
  try
  {
      type = A(A_0);
  }
  catch (Exception exception)
  {
      Log.Write(exception);
      throw new Microsoft1Licensing1Runtime210U("Failed to find implementation type",
                                                exception);
  }
  try
  {
      impl = (ISLMRuntimeImpl) Activator.CreateInstance(type);
  }
  catch (Exception exception2)
  {
      Log.Write(exception2);
      throw new Microsoft1Licensing1Runtime210U("Failed to create implementation class",
                                                exception2);
  }
  return impl;
}
```

In the context of exception handling texts and `Activator.CreateInstance` I suspected that the `A(A_0)` method searched and returned a type that implemented the `ISLMRuntimeImpl` interface. The method examination confirmed this assumption. In addition, the B method itself was called by the same type's C method which attempted to find a permutation assembly, then called B and finally performed some initialization.

The next logical step was to open the Microsoft.Licensing.Permutation_5538c_2.0 permutation assembly in the .NET Reflector and the find type that implemented the ISLMRuntimeImpl. A jungle of over 350 obfuscated and tightly coupled types was found. After proceeding further with the search of SLMRuntimeRedirect, the type was found. The type's static constructor created a couple of type instances and initialized the ISLMRuntimeImpl via the newly assigned A property field of the type Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft. What's more, several methods were responsible for protected method execution via the ((IExecutionEngine) A).ExecuteMethod. Thus, the property A's type should be reviewed. Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft type:

```
internal class Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft :
                                                        ISLMRuntimeImpl,
                                                        IExecutionEngine,
                                                        DistributorHooks
```

From what has been observed, there was no doubt that this was a predefined execution pipeline with substitutable components. A protected application required three additional .NET assemblies to execute. I suspected that two of these were common for all protected applications. Generic inheritance based runtime resolved the actual implementation from a permutation .NET assembly. Permutation was advertised to be unique, thus generic runtime simply executed a unique implementation of its core functionality including the execution engine. Consider a real world analogue: a car whose engine is easily replaceable. Internal engine implementation is unique for each instance, but the car itself remains the same. But here is the important question: what makes each engine unique? The research and implementation of a fully featured engine is known to be expensive and the engine must meet the car's requirements. For that reason is it possible that the engine itself is not so unique but rather only the fuel supply system is different? In the context of this protector, it was possible that the runtime that included execution was the same for all protections, but the data it operated on was unique for a concrete execution engine. As a result, the entire protection schema could be driven by unique data processed by a generic pipeline. This, however, was just an assumption.

Performing a quick analysis was required not only for the initial overview but also to ensure that the protector operated as advertised. Some of the protectors available on the market are known to never fulfill the promised level of protection. Their product marketing PDF had more lines of text than the actual protection code.

Protected code is concealed and handled by protector's runtime, but eventually it must be executable. There must be one or more endpoints for that purpose that delegate further execution to the CLR. Most probably, the transition at the endpoint involves a .NET reflection mechanism. The next step was to analyze the stack trace from the protected application. By clicking the "Stacktrace: MessageBox" button, a message box was displayed along with 37 traced methods. The detailed trace can be found in Appendix B. The essential part was:

```
--- StackFrame #: 6 - 11/7/2013 4:40:45 PM ---
System.Activator.CreateInstance
--- StackFrame #: 7 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
                Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS.A
--- StackFrame #: 8 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
                Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BZ.C
--- StackFrame #: 9 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi.I
--- StackFrame #: 10 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2F.A
--- StackFrame #: 11 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v.E
--- StackFrame #: 12 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
--- StackFrame #: 13 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
--- StackFrame #: 14 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
--- StackFrame #: 15 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
--- StackFrame #: 16 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A
--- StackFrame #: 17 - 11/7/2013 4:40:45 PM ---
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A
--- StackFrame #: 18 - 11/7/2013 4:40:45 PM ---
Microsoft.Licensing.SLMRuntime.InternalExecuteMethod
--- StackFrame #: 19 - 11/7/2013 4:40:45 PM ---
Microsoft.Licensing.SLMRuntime.ExecuteMethod
--- StackFrame #: 20 - 11/7/2013 4:40:45 PM ---
WinForms.Form1.btnSTMessageBox_Click
```

The trace must be analyzed from the bottom up. The aforementioned `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft` type was indeed a crucial entity. The button click event handler transitioned to SLMRuntime assembly, which transitioned to a permutation assembly. The endpoint was:

```
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
                    Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS.A
```

Clicking on the “Stacktrace: Exception” button also produced 37 traced methods and the same endpoint. With the “Stacktrace: Breakpoint” results were eventually the same. The suspected endpoint method was viewed:

```
protected override Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rN A(
                       MethodBase A_0,
                       Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi
                         .Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BD[] A_1)
{
  EDIT: not relevant lines were removed!

  object[] args = Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE.
  Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bc.A(A_1);

  return new Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rN(
    Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rw.B(Activator.CreateInstance(
    reflectedType, bindingAttr, new Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE.
    Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bc.
    Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BV(A_0), args, null)),
    Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rN.A.B);
}
```

The method’s body did not resemble an expected endpoint code. According to the MSDN, the .NET Framework `Activator.CreateInstance` method “creates an instance of the specified type using the constructor that best matches the specified parameters”. However, if the type instance was created here, then where was it used and where were the parameters passed? In Part 1 a breakpoint on the `MessageBoxW` WinAPI function was used as a trampoline to the caller code. A similar approach, but with a breakpoint set on a **managed** message box method appeared to be sufficient. After starting WinDbg, loading WinForms.exe and setting up the environment as described in WinDbg subchapter, the breakpoint was set:

```
!bpmd System.Windows.Forms.dll System.Windows.Forms.MessageBox.Show
```

The “MessageBox” button was clicked and the breakpoint was hit. The call stack was the following:

```
0:000> !clrstack
OS Thread Id: 0x8274 (0)
Child SP       IP Call Site
0037e298 0509f649 System.Windows.Forms.MessageBox.Show(System.String)
0037e5b4 0f472672 [DebuggerU2MCatchHandlerFrame: 0037e5b4]
0037e37c 0f472672 [HelperMethodFrame_PROTECTOBJ: 0037e37c] System.RuntimeMethodHandle.
```

```
          InvokeMethod(System.Object, System.Object[], System.Signature, Boolean)
0037e650 78aa376d System.Reflection.RuntimeMethodInfo.UnsafeInvokeInternal(
                                  System.Object, System.Object[], System.Object[])
0037e674 78ad8fcd System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
                                                      System.Reflection.BindingFlags,
                                                      System.Reflection.Binder,
                                                      System.Object[],
                                                      System.Globalization.CultureInfo)
0037e6a8 78a9b080 System.Reflection.MethodBase.Invoke(System.Object, System.Object[])
0037e6b4 01334c8f Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
  Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bn.A(System.Object,
                                                           System.Object[])

EDIT: remaining important, but currently not relevant stack trace was removed!
```

As anticipated, a .NET reflection was used. MSDN's description of the `Invoke` method said it "invokes the method or constructor represented by the current instance, using the specified parameters". From this stack trace it was evident that the `MessageBox.Show` was called via the `Invoke`. And the call originated from:

```
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
                      Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bn.A
```

This was a decent endpoint candidate. The method was further found in the .NET Reflector:

```
protected object A(object A_0, object[] A_1)
{
  Action<object> action2 = null;
  if (base.B.D() == null)
  {
      throw new Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rB(
                            "Null method {40E6ABCC-566F-4b9b-A23E-23DF00B28272}");
  }
  if (!Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BR.A(base.B.D(), A_0))
  {
      return base.B.D().Invoke(A_0, A_1);
  }
  if (action2 == null)
  {
      action2 = new Action<object>(this.A);
  }
  Action<object> action = action2;
  return Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BR.A(base.B.D(), action, A_1);
}
```

Important observation: `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bn` type, similarly to `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS`, was a nested type of the `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE` type. Thus, the latter presumably had a "create and execute" role. Given that this code was reusable, the type and method to call must be substitutable:

```
base.B.D().Invoke(A_0, A_1);
```

`base.B` was of the type `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bs`. The method `D` internally returned `this.B` class field value of the expected `MethodBase` type. The latter `B` was assigned in the constructor. The type `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bs` was just a wrapper. The next step was to find the instantiations of that wrapper. The type `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Ba` had the following method:

```
private static Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bs A(
                        IExecutionEngineGenericArguments A_0,
                        Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BT A_1)
{
  Assembly assembly = A_1.D(A_0).Assembly;
  bool flag = A_1.E();
  if (A_1.C())
  {
      return new Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bs(assembly,
                                                                      A_1.D().B(),
                                                                      flag,
                                                                      A_1.D().A());
  }
  return new Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bs(A_1.A(A_0),
                                                                      assembly,
                                                                      A_1.D().B(),
                                                                      flag,
                                                                      A_1.D().A());
}
```

The last code line was relevant where the first constructor parameter was of type `MethodBase`. This was the assignment of the `B` class field inside the constructor.

At this point it was feasible to test the assignment and invocation using the debugger. WinDbg was a great tool, however this task strongly required a quick browse and a detailed view of in-memory objects, and the DILE debugger was great in that respect. DILE was launched, WinForms.exe was loaded, and the debugging was started. The DILE's request to analyze all referenced assemblies was confirmed. In the tree structure the type `Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Ba` was navigated to and the earlier presented method `A` was found by signature. A breakpoint at the call at IL_0056 was set:

```
IL_0056:  callvirt instance string Microsoft_Licensing_Permutation_5538c_2_0_3_2_
                                                                1941_2U::A()
IL_005b:  newobj instance void Microsoft_Licensing_Permutation_5538c_2_0_3_2_
                    1941_2Bs::.ctor(class [mscorlib]System.Reflection.MethodBase,
                                    class [mscorlib]System.Reflection.Assembly,
                                    valuetype [mscorlib]System.Guid, bool, string)
IL_0060: ret
```

A click on the "MessageBox" button was followed by DILE breaking at IL_004f. After opening the "Local Variables Panel", the V_2 variable's type was `RuntimeMethodInfo`. Right clicking on it and selecting "Display Object in Viewer" led to DILE displaying a modal dialog where detailed .NET specific object information was presented. The dialog was closed, "Watch Panel" was opened and a new expression was added for evaluation pusposes: `V_2.FullName`. Then the watch window was displaying the full name of the method. Pressing F5 continued the execution. The breakpoint was hit again and the watched expression changed. This "run-stop" happened tens of times, and it was important to pay attention to the watched expression, because interesting details about the protector runtime internals were exposed.
At some point, the following value appeared in the watched expression:

```
System.Windows.Forms.MessageBox.Show(System.String)
```

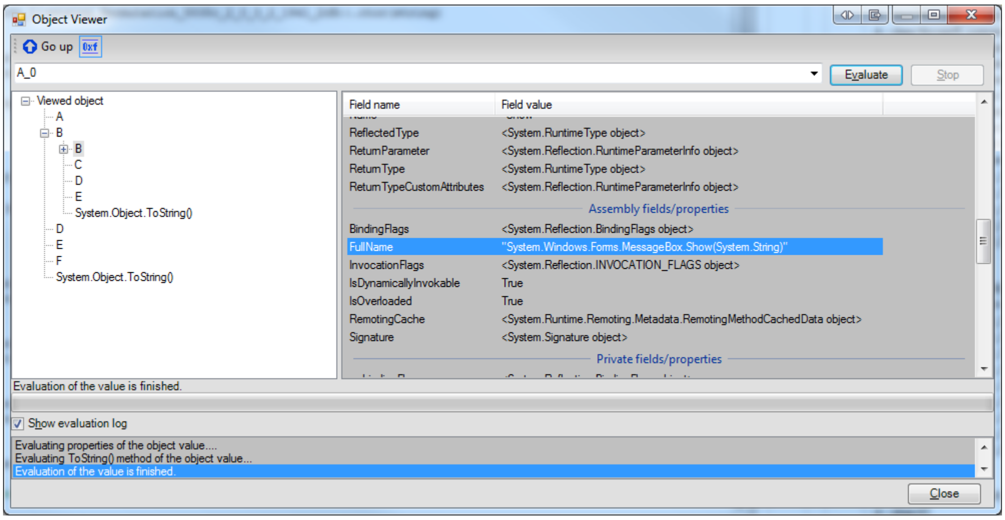
A_1 in "Arguments Panel" contained details. Basically, it was information about the type itself, assembly it belonged to, method name, binding flags, return type, parameters etc.
Next the following type was navigated to:

```
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
                        Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bn
```

it was expanded and a breakpoint was set at IL_005d:

```
IL_005d:    ldarg.1
IL_005e:    ldarg.2
IL_005f:    callvirt instance object [mscorlib]System.Reflection.MethodBase::Invoke(
                                                              object, object[])
```

Pressing F5, the second breakpoint was reached at IL_0052. There were three arguments in the "Arguments Panel" tab. The first one encapsulated invocation details including the assigned runtime method in the previous breakpoint. These details could be retrieved if the object was analyzed in the "Object Browser" as illustrated by Figure 3.9. The second argument was the instance object, and the



A_0

```
Viewed object
  A
  B
    B
    C
    D
    E
    System.Object.ToString()
  D
  E
  F
  System.Object.ToString()
```

Field name	Field value
ReflectedType	<System.RuntimeType object>
ReturnParameter	<System.Reflection.RuntimeParameterInfo object>
ReturnType	<System.RuntimeType object>
ReturnTypeCustomAttributes	<System.Reflection.RuntimeParameterInfo object>
Assembly fields/properties	
BindingFlags	<System.Reflection.BindingFlags object>
FullName	"System.Windows.Forms.MessageBox.Show(System.String)"
InvocationFlags	<System.Reflection.INVOCATION_FLAGS object>
IsDynamicallyInvokable	True
IsOverloaded	True
RemotingCache	<System.Runtime.Remoting.Metadata.RemotingMethodCachedData object>
Signature	<System.Signature object>
Private fields/properties	

Evaluation of the value is finished.

☑ Show evaluation log

```
Evaluating properties of the object value....
Evaluating ToString() method of the object value...
Evaluation of the value is finished.
```

Figure 3.9: The first argument analysis

third contained an array of arguments to be passed. Obviously, for static method calls the second argument was null.

Viewing the third argument in "Object Browser" revealed runtime method arguments: the array had a single item of string type with the value "Hello world!". Thus

```
base.B.D().Invoke(A_0, A_1);
```

was about to call the static method:

```
System.Windows.Forms.MessageBox.Show("Hello world!")
```

which of course resembled the unprotected `btnMessageBox_Click` method body. As a result, if local variables and arguments were analyzed for both breakpoints it was possible to retrieve details about instances at runtime, and to find instances that were part of the invocation (if static then null) and their arguments.

On the other hand, if `btnSTMessageBox_Click` was reviewed some valuable information would be missing. Unprotected `btnSTMessageBox_Click` had a loop which iterated over stack frames. The current interception approach provided details about types including type instantiation[12], properties accessed or assigned, methods called, and arguments. However, there was no information about the aforementioned loop. Obviously, when the second breakpoint was consequently reached numerous times and

[12] this is known because object constructor is a method thus visible here

executing logic repeated itself it was an indication of a potential loop. Yet it was not a reliable indication, and what's more any flow control information was absent. Hence, there must have been an *orchestra conductor*, an entity or entities that described the semantics of the original method, because the protected code must have functioned exactly as its unprotected counterpart.

This has been a simple 10K feet view of the protector. Nevertheless, points of interest were easily identified without the in-depth study. At this point the description has to be concluded in order to adhere to the initial personal intention: to reduce the impact on existing applications. However, the presented shallow study was sufficient to remove the protection for the unfortunate application reviewed in the next subchapter.

3.2.2 Testing on a commercial application

The following subchapter determines the applicability of the knowledge obtained to a commercial application. The target was a .NET WPF application. The software was installed and activated with the obtained key. To conceal the application identity it is further referred to as the APP and sensitive information is censored.

The previously used application was designed to assist the endpoints search, but how to begin with the APP? The first approach was to perform static analysis with the .NET Reflector. Even if the execution pipeline structure was different, the core interfaces must have existed, unless the entire pipeline implementation was unique too. I have already used the analogue of a car and its engine as an example of the feasibility of the pipeline being unique for every application. If the pipeline structure remained intact then a simple pattern search would have revealed the endpoints. Nevertheless, I decided to use a different approach: application profiling. An earlier review of endpoints at runtime determined that their invocation frequency was very high, hence the profiler marked such methods as "hot".

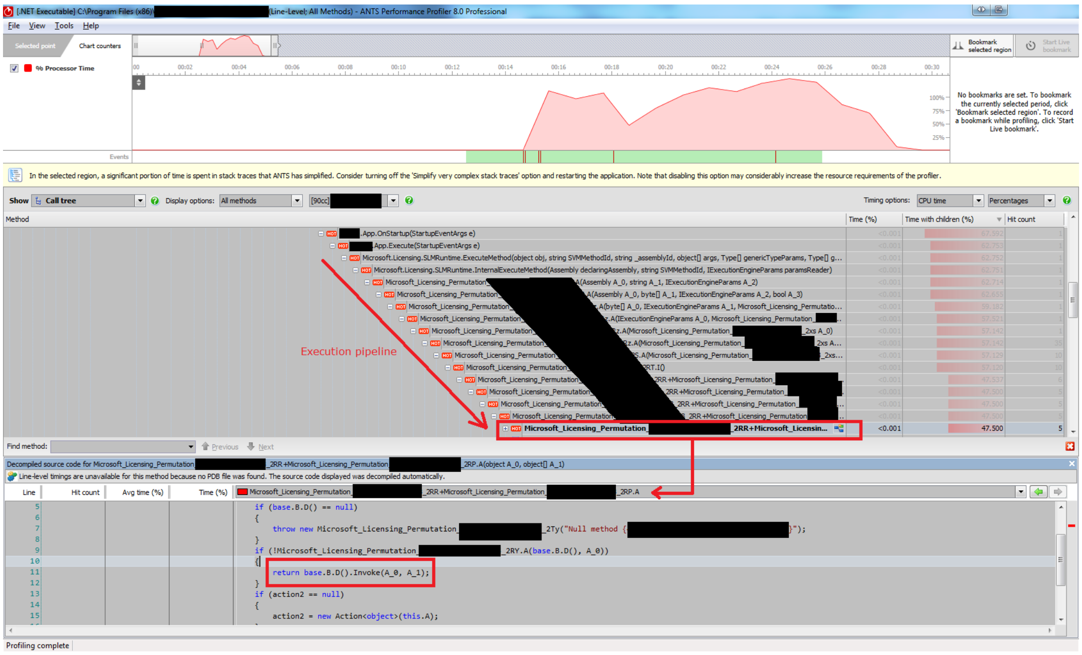
First the APP's protected methods had to be identified in the decompiler. Only one protected method was found which was executed during application startup `OnStartup` callback in `App` class derived from `Application` class. It was probably responsible for the application initialization. The following three `App` class methods were particularly interesting:

```
protected override void OnStartup(StartupEventArgs e)
{
  FrameworkElement.LanguageProperty.OverrideMetadata(typeof(FrameworkElement),
                        new FrameworkPropertyMetadata(XmlLanguage.GetLanguage(
                                   CultureInfo.CurrentCulture.IetfLanguageTag)));
  if (VerifyInstalled())
  {
    this.Execute(e);
  }
  else
  {
    base.Shutdown(1);
  }
}

[MethodImpl(MethodImplOptions.NoInlining)]
private void Execute(StartupEventArgs e)
{
  object[] args = new object[] { e };
  SLMRuntime.ExecuteMethod(this, "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX", "",
                           args, null, null);
}

[MethodImpl(MethodImplOptions.NoInlining)]
public void _XXXXXXXXXXX_System.Windows.Application, PresentationFramework,
    Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35_OnStartup(
                                                     StartupEventArgs A_0)
{
  base.OnStartup(A_0);
}
```

The first method was a startup event callback. A quick analysis of `VerifyInstalled` revealed that it checked if the application was properly installed, and if the protector's license storage was accessible. Then the protected `Execute` method was called. The third method's signature hinted that this method was inserted by the protector and contained a part of the protected code that was not possible or convenient to execute otherwise. As a result, that code was wrapped into a local method which was invoked by the protector's runtime. This assumption was trustworthy because the enclosing type was `System.Windows.Application`, whose `OnStartup` method was overridden, but base implementation which was required for the successful execution was never called. Hence, the original unprotected `Execute` method version called `base.OnStartup(e)`[13].

The profiling was started after loading the APP in ANTS Performance Profiler. The details presented by Figure 3.10 were available in several seconds. These results were compared to the profiling results of the WinForms.exe used in Part 1 and confirmed that the pipelines were identical with minor difference of an occasional `SLMRuntimeRedirect` call and obviously different type names due to obfuscation.

[13] argument type is `StartupEventArgs`



[.NET Executable] C:\Program Files (x86)\[redacted](Line-Level; All Methods) - ANTS Performance Profiler 8.0 Professional

No bookmarks are set. To bookmark the currently selected period, click 'Bookmark selected region'. To record a bookmark while profiling, click 'Start Live bookmark'.

In the selected region, a significant portion of time is spent in stack traces that ANTS has simplified. Consider turning off the 'Simplify very complex stack traces' option and restarting the application. Note that disabling this option may considerably increase the resource requirements of the profiler.

Show: Call tree; Display options: All methods; [90cc][redacted]; Timing options: CPU time; Percentages

Method	Time (%)	Time with children (%)	Hit count
[redacted].App.OnStartup(StartupEventArgs e)	<0.001	67.592	1
[redacted].App.Execute(StartupEventArgs e)	<0.001	62.753	1
Microsoft.Licensing.SLMRuntime.ExecuteMethod(object obj, string SVMMethodId, string _assemblyId, object[] args, Type[] genericTypeParams, Type[] g...	<0.001	62.752	1
Microsoft.Licensing.SLMRuntime.InternalExecuteMethod(Assembly declaringAssembly, string SVMMethodId, IExecutionEngineParams paramsReader)	<0.001	62.751	1
Microsoft_Licensing_Permutation_[redacted]A(Assembly A_0, string A_1, IExecutionEngineParams A_2)	<0.001	62.714	1
Microsoft_Licensing_Permutation_[redacted]A(Assembly A_0, byte[] A_1, IExecutionEngineParams A_2, bool A_3)	<0.001	62.655	1
Microsoft_Licensing_Permutation_[redacted].A(byte[] A_0, IExecutionEngineParams A_1, Microsoft_Licensing_Permutatio...	<0.001	59.182	1
Microsoft_Licensing_Permutation_[redacted]z.A(IExecutionEngineParams A_0, Microsoft_Licensing_Permutation_[redacted]..	<0.001	57.521	1
Microsoft_Licensing_Permutation_[redacted]z.A(Microsoft_Licensing_Permutation_[redacted]_2xs A_0)	<0.001	57.142	1
Microsoft_Licensing_Permutation_[redacted]Rz.A(Microsoft_Licensing_Permutation_[redacted]_2xs A...	<0.001	57.142	35
Microsoft_Licensing_Permutation_[redacted]S.A(Microsoft_Licensing_Permutation_[redacted]_2xs...	<0.001	57.129	10
Microsoft_Licensing_Permutation_[redacted]RT.I()	<0.001	57.120	10
Microsoft_Licensing_Permutation_[redacted]2RR+Microsoft_Licensing_Permutation_[redacted].	<0.001	47.537	6
Microsoft_Licensing_Permutation_[redacted]2RR+Microsoft_Licensing_Permutation_[redacted].	<0.001	47.500	5
Microsoft_Licensing_Permutation_[redacted]2RR+Microsoft_Licensing_Permutation_[redacted]...	<0.001	47.500	5
Microsoft_Licensing_Permutation_[redacted]2RR+Microsoft_Licensing_Permutation_[redacted]...	<0.001	47.500	5
Microsoft_Licensing_Permutation_[redacted]2RR+Microsoft_Licensin...	<0.001	47.500	5

Decompiled source code for Microsoft_Licensing_Permutation_[redacted]_2RR+Microsoft_Licensing_Permutation_[redacted]_2RP.A(object A_0, object[] A_1)

Line-level timings are unavailable for this method because no PDB file was found. The source code displayed was decompiled automatically.

Line	Hit count	Avg time (%)	Time (%)

Microsoft_Licensing_Permutation_[redacted]_2RR+Microsoft_Licensing_Permutation_[redacted]_2RP.A

```
if (base.B.D() == null)
{
    throw new Microsoft_Licensing_Permutation_[redacted]_2Ty("Null method {[redacted]}");
}
if (!Microsoft_Licensing_Permutation_[redacted]_2RY.A(base.B.D(), A_0))
{
    return base.B.D().Invoke(A_0, A_1);
}
if (action2 == null)
{
    action2 = new Action<object>(this.A);
```

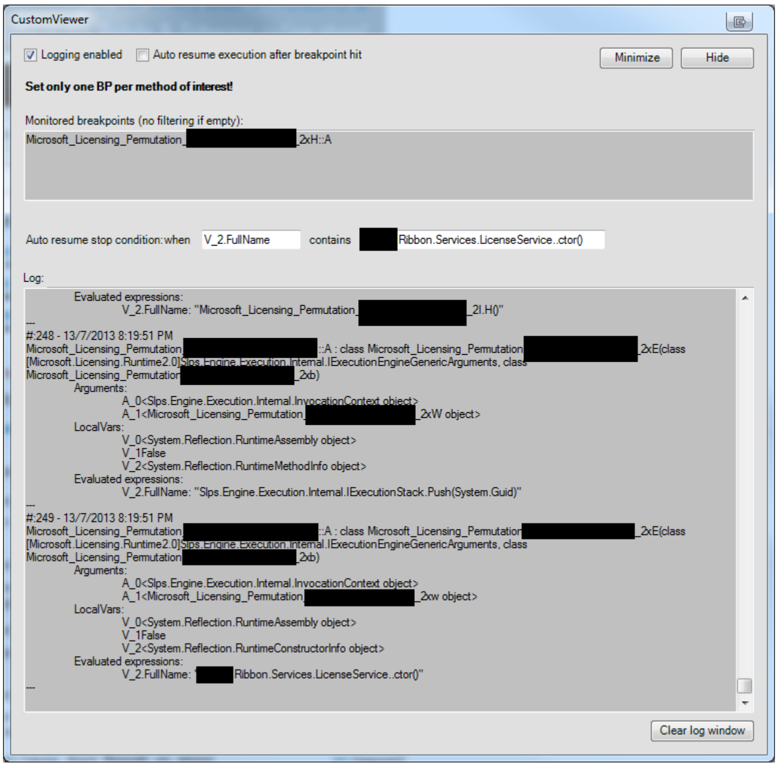
Profiling complete

Figure 3.10: Revealed execution pipeline and endpoint.

According to previous experience, the next step was to load the APP in DILE debugger, set two breakpoints to already known locations, and monitor the execution. The entire procedure has one major drawback: every time a breakpoint was reached a manual action was required. To automate the process DILE sources were downloaded, an application form called CustomViewer was developed and integrated it into the debugger.

The form's code was mainly based on DILE's own code with additional interfaces to control the debugging process. The main features, as illustrated by Figure 3.11, included detailed logging of the current method information including its arguments and local variables. It was possible to filter which breakpoints were monitored. Furthermore, the watch expression could be evaluated and the most important feature was the conditional automatic resume.

Profiling revealed the method invocation endpoint, but the assignment endpoint was also required which was easily found using the knowledge from the previous subchapter. Setting a similar breakpoint in DILE and then enabling automatic resume in CustomViewer was followed by starting the debugging. A charming logging process began. After some time, the logging stopped while the application continued to run.



☑ Logging enabled □ Auto resume execution after breakpoint hit

Set only one BP per method of interest!

Monitored breakpoints (no filtering if empty):

Microsoft_Licensing_Permutation_[redacted]_2xH::A

Auto resume stop condition:when V_2.FullName contains [redacted]Ribbon.Services.LicenseService..ctor()

Log:

```
          Evaluated expressions:
                    V_2.FullName: "Microsoft_Licensing_Permutation_[redacted]_2I.H()"
---
#:248 - 13/7/2013 8:19:51 PM
Microsoft_Licensing_Permutation[redacted]::A : class Microsoft_Licensing_Permutation[redacted]_2xE(class
[Microsoft.Licensing.Runtime2.0]Slps.Engine.Execution.Internal.IExecutionEngineGenericArguments, class
Microsoft_Licensing_Permutation[redacted]_2xb)
          Arguments:
                    A_0<Slps.Engine.Execution.Internal.InvocationContext object>
                    A_1<Microsoft_Licensing_Permutation_[redacted]_2xW object>
          LocalVars:
                    V_0<System.Reflection.RuntimeAssembly object>
                    V_1False
                    V_2<System.Reflection.RuntimeMethodInfo object>
          Evaluated expressions:
                    V_2.FullName: "Slps.Engine.Execution.Internal.IExecutionStack.Push(System.Guid)"
---
#:249 - 13/7/2013 8:19:51 PM
Microsoft_Licensing_Permutation[redacted]::A : class Microsoft_Licensing_Permutation[redacted]_2xE(class
[Microsoft.Licensing.Runtime2.0]Slps.Engine.Execution.Internal.IExecutionEngineGenericArguments, class
Microsoft_Licensing_Permutation[redacted]_2xb)
          Arguments:
                    A_0<Slps.Engine.Execution.Internal.InvocationContext object>
                    A_1<Microsoft_Licensing_Permutation_[redacted]_2xw object>
          LocalVars:
                    V_0<System.Reflection.RuntimeAssembly object>
                    V_1False
                    V_2<System.Reflection.RuntimeConstructorInfo object>
          Evaluated expressions:
                    V_2.FullName: "[redacted]Ribbon.Services.LicenseService..ctor()"
---
```

Figure 3.11: Custom debugger assistant.

There were over two hundred log entries. The last log entries provided details about the logic of protected code that started from the `XXXXX.Ribbon.Services.LicenseService..ctor()` entry. The debugging was stopped. The requirement was to stop debugger when a method with that name was assigned. This condition was set in CustomViewer as presented by 3.11. The debugging was started again and eventually CustomViewer stopped the debugger when the conditions were satisfied. Next, the breakpoint was set inside the invocation endpoint. Now, by analyzing the output in CustomViewer and arguments inside the invocation method for each breakpoint hit, it was possible to retrieve the execution details of the protected code. With minor additional research I was able to convert my findings into a new version of `OnStartup` method:

```
protected override void OnStartup(StartupEventArgs e)
{
  FrameworkElement.LanguageProperty.OverrideMetadata(typeof(FrameworkElement),
                      new FrameworkPropertyMetadata(XmlLanguage.GetLanguage(
                        CultureInfo.CurrentCulture.IetfLanguageTag)));
  LicenseService licenseService = new LicenseService();
  SplashScreenViewModel splashScreenViewModel = new SplashScreenViewModel(
                                                    licenseService.IsTrial());
  SplashScreenView splashScreenView = new SplashScreenView
  {
    DataContext = splashScreenViewModel
  };

  splashScreenView.Show();

  base.OnStartup(e);

  new XXXXXBootstrapper(e.Args,
                        splashScreenView,
                        splashScreenViewModel,
                        licenseService)
                        .Run();
}
```

The new `OnStartup` method body could be easily inserted[14] as C# code opposed to per CIL instruction manipulation using the same Reflexil plugin. I also modified the `licenseService.IsTrial()` to always return "false". The patched binary had to be tested, but first of all, the existing registration had to be removed. It was stored in the registry key[15]:

`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\SLP Services`

For that purpose, the binary value that started with the `ComputerIdData_` was renamed, then the APP was started and the anticipated activation dialog appeared. The next steps were to quit the application, overwrite existing binary with the patched version and, finally, start it again. Success!

The original application and the patched binary are provided **only** for the thesis supervisor as a proof of concept.

[14] "Replace all with code" feature. Existing class methods must be reinserted too

[15] activation data is stored in a single binary entry and during debugging it is possible to retrieve activation details for all protected applications

Chapter 4

Additional areas of interest

Obviously there are more software utilities available than presented in this work. They range from tracers of the JIT events to automatic CIL patching at runtime and beyond. In addition, numerous tutorials, research papers, useful blog posts, training courses such as OpenSecurityTraining and so on are spread across the Internet, and a simple, but patient search reveals the information. Furthermore, http://www.crackmes.de is one of the most valuable sources for RCE.

The previous chapters presented background information about the .NET and described static and dynamic code analysis emphasizing the benefits of their combination. However, another interesting research area was left out. Particularly the .NET reflection mechanism in conjunction with runtime code injection and an attack on the .NET Framework itself. The .NET reflection is a very powerful framework feature. It provides an opportunity to obtain detailed information about any type, retrieve current values from a type instance and what's more, assign new values despite the declared accessibility. Its capabilities are definitely not limited to the aforementioned. Basically, there are two approaches to utilize it inside the target application: static reference which requires binary modification or runtime which is a .NET specific form of DLL injection. For that purpose it is convenient to first develop a separate .NET assembly and execute it in the target application using either one of the approaches. Attacking at runtime has certain benefits especially when static integration is not feasible. The assumption is that the target is protected by a native code wrapper or contains integrity validation of its binary file. However, when an application is already in memory and running, these procedures have already completed, thus such a type of an attack becomes more relevant and easier. Nevertheless, there is an obstacle to overcome which is to retrieve the pointer to the required instance object.

There are several notable researchers in the area of attacking the .NET applications at runtime and the .NET Framework. One of them is Jon McCoy[1] who is famous for his tools and various conference presentations for developers and security experts. All of the tools, sample applications, research papers and links to video presentations are available on his website. I contacted Jon and he kindly agreed to share his views and ideas on the subject. Moreover, he provided me with details on new upcoming tools.

In my opinion, the GrayDragon is one of the most powerful tools he has developed. It allows injection of custom .NET assemblies to the target .NET application at runtime. The most interesting part is how the actual .NET specific injection is performed and how the CLR hosting feature[2] is utilized to facilitate the task. The GrayDragon contains several ready-made payloads that assist the target application analysis. Basically, the payload is a .NET assembly that will be injected.

Here is a quick proof of concept for the FakeRealApp program. I developed a simple .NET application that consists of only one button. The button's click event handler has the following code:

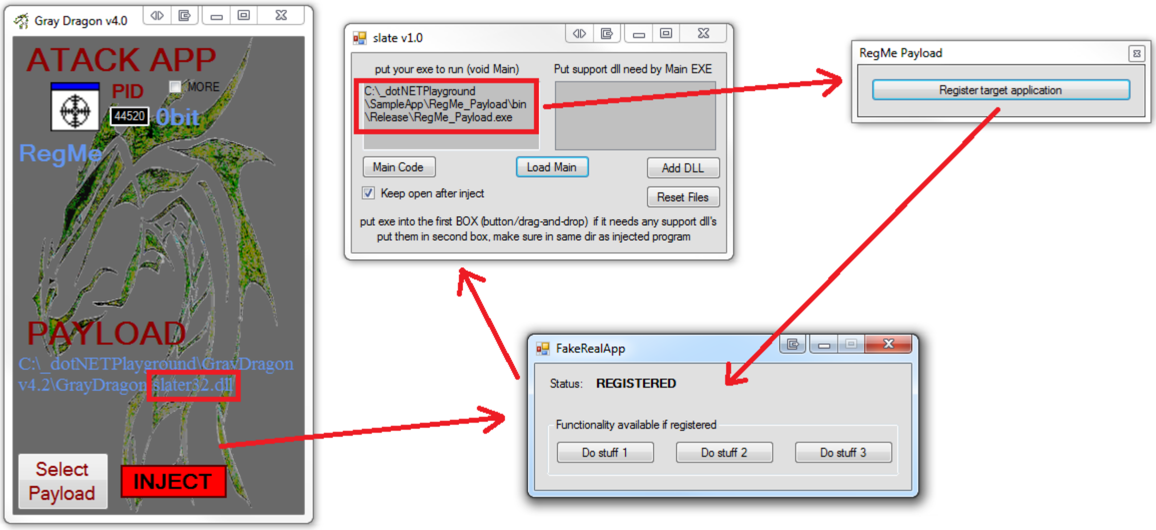
```
private void btnRegIt_Click(object sender, EventArgs e)
{
    object inst = null;
    foreach (Form frm in Application.OpenForms)
    {
        if (frm.Name == "MainForm")
        {
            inst = frm;
            break;
        }
    }

    if (inst == null)
    {
        MessageBox.Show("Target form is not found!");
        return;
    }

    var type = inst.GetType();
    var flags = BindingFlags.Instance | BindingFlags.NonPublic;

    var prop = type.GetProperty("IsRegistered", flags);
    prop.SetValue(inst, true, null);

    var method = type.GetMethod("SetupUI", flags);
```

[1] http://www.digitalbodyguard.com

[2] was described in 2.4.2 subchapter

```
    method.Invoke(inst, null);
}
```

It enumerates through opened application forms, finds the required one and then uses the .NET reflection to set the form's `IsRegistered` property to "true" and call the form's `SetupUI` method. It is important to mention that this property and method are not declared as public, thus they cannot be accessed directly. However the boundary is eliminated by a `BindingFlags.NonPublic` flag. Finally, the payload is loaded into target FakeRealApp through the GrayDragon as illustrated by Figure 4.1 and the application state is changed to registered without the actual registration procedure.



C:_dotNETPlayground\GrayDragon v4.2\GrayDragon slater32.dll

put your exe to run (void Main)

C:_dotNETPlayground \SampleApp\RegMe_Payload\bin \Release\RegMe_Payload.exe

Put support dll need by Main EXE

☑ Keep open after inject

put exe into the first BOX (button/drag-and-drop) if it needs any support dll's put them in second box, make sure in same dir as injected program

Status: **REGISTERED**

Functionality available if registered

Figure 4.1: .NET reflection at its glory

A very simple .NET code has altered the third-party application at runtime. The use cases for the .NET reflection are tremendous. I think the most valuable for RCE is the manipulation of existing application methods i.e. altering a method at runtime. The analysis of the InishTech protector required a debugger, but what if those endpoint methods were modified in a certain way and could report their activity? The injected payload via reflection can access methods defined in another assembly. Moreover, it is possible to retrieve the pointer to a native code of the method, once it is compiled. In addition, methods can be entirely substituted at runtime. Several of such approaches are discussed in [Wan13], [Cou13] and [Elm13].

Another powerful approach is to modify a compiled method's native code directly.

For testing purposes I have developed a simple application that has only one button and a textbox. When the button is clicked a message box is displayed. The main form also has a static method called Log that accepts arguments of the type `object`. This method calls the `ToString` on that argument and the result is shown in the textbox. I started the application in WinDbg, found the compiled native code for the button click event handler, and manipulated the assembly code to redirect it to the Log method. After the method was called I redirected the execution back. From now on, when the button is clicked, a textbox displays the value. Then, the original message box is displayed. To be honest, it was slightly more complicated, because such edits require sufficient space to exist, so I had to add additional code[3] whose native code instructions I can safely overwrite in-memory. Moreover, the Log method was internally calling the LogActual (the actual logging method) for the same purpose, because I needed to return the execution back to the initiator. But, despite this minor drawback the approach has proven itself to be feasible. I think it is applicable to the aforementioned InishTech protector too, as endpoints can now be monitored without a debugger. What's more, a special program that is able to inject into specified methods and log all activity including values of local variables and arguments can be developed. Such integration with logging capabilities reminds the aspect-oriented programming paradigm applied at runtime but without a preliminary setup.

Finally, the last thing I want to mention is the modification of the .NET Framework itself. If malicious software attacks it, the consequences can be devastating. Erez Metula discusses managed code rootkits in his "Managed Code Rootkits: Hooking into Runtime Environments" book. Its scope is not limited to just .NET or Java runtimes. In addition, the aforementioned Jon McCoy has similar sample applications that target the framework available on his website. From the RCE's perspective, becoming *friends* with framework is extremely beneficial. The very same analysis of the second invocation endpoint in the InishTech protector which calls the `MethodBase.Invoke` method is a good example. The `MethodBase` type is defined in framework's library and if it is modified in certain way it can perform the logging automatically for `all` .NET applications. And if similar modifications are extended to all framework's methods related to, e.g. reflection then, well, I think any additional comments are redundant at this point.

[3] a dummy for loop

Chapter 5

Conclusion and future work

The RCE complexity of .NET applications depends on a concrete protector. Basically, it is either a protection implemented using native code (native wrappers, CIL injectors, calling unmanaged DLL code etc.) or a managed counterpart (CIL obfuscation). Furthermore, a combination of these two is possible. I have demonstrated tools and presented analysis applicable for both cases with the experiments on sample applications. Nevertheless, as seen, it is easier to analyze managed protection due to available .NET metadata and the high-level object-oriented nature of the CIL. Enough background information has been provided to understand the main principles of the .NET specific RCE. The benefits of application analysis using a debugger have been outlined regarding the protector which is strongly resilient to static analysis. In addition, the power of attacking the application at runtime and the .NET reflection have been described. What's more, the benefit of directly integrating custom logic into the .NET Framework libraries has been mentioned.

RCE is a very large topic. When I initially started my research, I studied a variety of sources. Everything felt relevant. However, I soon realized that the material I had gathered was enough for a whole book, hence I had to narrow everything to a scope of the thesis requirements. As a result, I decided to follow a different approach and create a document that contains, in my opinion, the most fundamental information on the subject that can serve as a decent supplementary to any of the freely available .NET RCE sources. In addition, I now have a clear vision of RCE tools and important features I would like to have. Some of these ideas were shared with Jon McCoy and, hopefully, this will have an effect on his upcoming tools. Next, the presented knowledge allows me, a regular .NET developer, to be conscious of existing code protection solutions for .NET and their major pitfalls.

Finally, a general recommendation for developers is to create decent software with a reasonable price tag. Custom validation is always welcome as an addition to external code protector. The question is not *if* the software will be cracked, but *when*. I consider that the end users' appreciation of the software is what actually brings the revenue. Those who want to use it for free will use it for free anyway.

An old member of the RCE scene once said:
"If it runs, it can be defeated!"

Bibliography

[Ber13] Andrea Bertolotto. Removing Strong-Signing from Assemblies at File Level (byte patching). http://www.codeproject.com/Articles/15374/Removing-Strong-Signing-from-As, June 2013.

[CNE13] CNET News. Sun, Microsoft settle Java suit. http://news.cnet.com/2100-1001-251401.html, March 2013.

[Cou13] Julien Couvreur. Modifying IL at runtime. http://blog.monstuff.com/archives/000058.html, June 2013.

[Dai13] Andrew Dai. Exploring the .NET Framework 4 Security Model. http://msdn.microsoft.com/en-us/magazine/ee677170.aspx, June 2013.

[Elm13] Ziad Elmalki. CLR Injection: Runtime Method Replacer. http://www.codeproject.com/Articles/37549/CLR-Injection-Runtime-Method-Replacer, June 2013.

[Far13] Jim Farley. Microsoft .NET vs. J2EE: How Do They Stack Up? http://www.oreillynet.com/pub/a/oreilly/java/news/farley_0800.html, March 2013.

[Fra03] S.R.G. Fraser. *Managed C++ and .NET Development*. Apress, 2003.

[Fry06] Linn Marie Frydenberg. Analysis of Obfuscated CIL code. Master's thesis, August 2006.

[Gil13] Howard Gilbert. .NET Framework. http://pclt.cis.yale.edu/tp/framework.htm, March 2013.

[Gun13] Mike Gunderloy. Understanding and Using Assemblies and Namespaces in .NET. http://msdn.microsoft.com/en-us/library/ms973231.aspx, March 2013.

[Hew10] Mario Hewardt. *Advanced .NET Debugging.* Addison-Wesley, 2010.

[Ini13] InishTech. Software Potential. Software Licensing from the Cloud. http://support.inishtech.com/getdoc/00359bb0-84de-4a79-910f-a331f8c5b9ef/Software-Potential-Service-Overview.aspx, June 2013.

[Joh13] Steve Johnson. Sosex home page. http://www.stevestechspot.com, June 2013.

[Lid06] S. Lidin. *Expert .NET 2.0 Assembler.* Apress, 2006.

[Lip13] Eric Lippert. What's the Difference, Part Five: certificate signing vs strong naming. http://blogs.msdn.com/b/ericlippert/archive/2009/09/03/what-s-the-difference-part-five-certificate-signing-vs-strong-naming.aspx, June 2013.

[LT13] Richard Linger and Carmen Trammell. Cleanroom Software Engineering Reference. http://www.sei.cmu.edu/library/abstracts/reports/96tr022.cfm, January 2013.

[Mic13a] Microsoft. Assembly Contents. http://msdn.microsoft.com/en-us/library/zst29sk2.aspx, March 2013.

[Mic13b] Microsoft. Assembly Manifest. http://msdn.microsoft.com/en-us/library/1w45z383.aspx, March 2013.

[Mic13c] Microsoft. _CorValidateImage Function. http://msdn.microsoft.com/en-us/library/4ce9k7xb(v=vs.110).aspx, April 2013.

[Mic13d] Microsoft. Download and Install Debugging Tools for Windows. http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx, June 2013.

[Mic13e] Microsoft. Introduction to the C# Language and the .NET Framework. http://msdn.microsoft.com/en-us/library/vstudio/z1zx9t92.aspx, March 2013.

[Mic13f] Microsoft. Microsoft Portable Executable and Common Object File Format Specification. http://msdn.microsoft.com/library/windows/hardware/gg463125, March 2013.

[Mic13g] Microsoft. Mixed (Native and Managed) Assemblies. http://msdn.microsoft.com/en-us/library/x0w2664k.aspx, March 2013.

[Mic13h] Microsoft. .NET Framework Conceptual Overview. http://msdn.microsoft.com/en-us/library/vstudio/zw4w595w(v=vs.100).aspx, March 2013.

[Mic13i] Microsoft. Overview of Metadata. http://msdn.microsoft.com/en-us/library/ms404430.aspx, March 2013.

[Mic13j] Microsoft. Overview of the .NET Framework. http://msdn.microsoft.com/en-us/library/zw4w595w.aspx, February 2013.

[Mic13k] Microsoft. Security Policy Management. http://msdn.microsoft.com/en-us/library/vstudio/c1k0eed6(v=vs.100).aspx, June 2013.

[Mic13l] Microsoft. SOS.dll (SOS Debugging Extension). http://msdn.microsoft.com/en-us/library/bb190764.aspx, June 2013.

[Mic13m] Microsoft. Use the Microsoft Symbol Server to obtain debug symbol files. http://support.microsoft.com/kb/311503, June 2013.

[Mic13n] Microsoft and partners. Standard ECMA-335 Common Language Infrastructure (CLI) 6th edition (June 2012). http://www.ecma-international.org/publications/standards/Ecma-335.htm, March 2013.

[Mic13o] Microsoft Patterns & Practices Team. Microsoft Application Architecture Guide, 2nd Edition. http://msdn.microsoft.com/en-us/library/ee658104.aspx, January 2013.

[Mor08] J.A. Morales. Threat of renovated. net viruses to mobile devices. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, pages 367–372. ACM, 2008.

[Mur05] Nicholas John Murison. .NET Framework Security. Master's thesis, January 2005.

[Pis13a] Daniel Pistelli. Cffexplorer. http://www.ntcore.com/exsuite.php, March 2013.

[Pis13b] Daniel Pistelli. .NET Internals and Code Injection. http://ntcore.com/files/netint_injection.htm, March 2013.

[Pis13c] Daniel Pistelli. .NET Internals and Native Compiling. http://www.ntcore.com/files/netint_native.htm, March 2013.

[Pis13d] Daniel Pistelli. Remotesoft's Salamander 1.1.6.0 (Native Compiling). http://www.ntcore.com/files/salamander.htm, March 2013.

[Pis13e] Daniel Pistelli. The .NET File Format. http://www.ntcore.com/files/dotnetformat.htm, March 2013.

[PNSS08] J. Pobar, T. Neward, D. Stutz, and G. Shilling. *Shared Source CLI 2.0 Internals*. 2008.

[Pot05] V.E. Poteat. Classroom ethics: hacking and cracking. *Journal of Computing Sciences in Colleges*, 20(3):225–231, 2005.

[Ric10] Jeffrey Richter. *CLR via C#*. Microsoft Press, 2010.

[sha13] shaheeng. The XBox Operating System. http://blogs.msdn.com/b/xboxteam/archive/2006/02/17/534421.aspx, February 2013.

[Smi13] SmidgeonSoft. Windows Debugger, Disassembler, Code Analyzers. http://www.smidgeonsoft.prohosting.com/software.html, June 2013.

[Sta13] Mike Stall. Debugabbility with Roundtripping Assemblies. http://blogs.msdn.com/b/jmstall/archive/2006/01/13/debug-roundtripping.aspx, June 2013.

[Wan13] Jerry Wang. .NET CLR Injection: Modify IL Code during Run-time. http://www.codeproject.com/Articles/463508/NET-CLR-Injection-Modify-IL-Code-during-Run-time, June 2013.

[Xam13] Xamarin and Mono community. Mono project. http://www.mono-project.com, February 2013.

[Yus13] Oleh Yuschuk. Ollydbg. http://www.ollydbg.de, June 2013.

[Zso13] Petrny Zsolt. Dotnet IL Editor. http://sourceforge.net/projects/dile, June 2013.

Appendix A

Dumpbin utility output for SampleApp.exe

RAW DATA content was skipped

```
Microsoft (R) COFF/PE Dumper Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file SampleApp.exe
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
             14C machine (x86)
               3 number of sections
        519A91C8 time date stamp Tue May 21 00:12:40 2013
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             102 characteristics
                   Executable
                   32 bit word machine
OPTIONAL HEADER VALUES
             10B magic # (PE32)
           11.00 linker version
             800 size of code
             800 size of initialized data
               0 size of uninitialized data
            276E entry point (0040276E)
            2000 base of code
            4000 base of data
          400000 image base (00400000 to 00407FFF)
            2000 section alignment
             200 file alignment
            4.00 operating system version
            0.00 image version
            4.00 subsystem version
               0 Win32 version
            8000 size of image
             200 size of headers
               0 checksum
               3 subsystem (Windows CUI)
            8540 DLL characteristics
                   Dynamic base
                   NX compatible
                   No structured exception handler
                   Terminal Server Aware
          100000 size of stack reserve
            1000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
               0 [       0] RVA [size] of Export Directory
            2720 [      4B] RVA [size] of Import Directory
            4000 [     540] RVA [size] of Resource Directory
               0 [       0] RVA [size] of Exception Directory
               0 [       0] RVA [size] of Certificates Directory
            6000 [       C] RVA [size] of Base Relocation Directory
            26AC [      1C] RVA [size] of Debug Directory
               0 [       0] RVA [size] of Architecture Directory
               0 [       0] RVA [size] of Global Pointer Directory
               0 [       0] RVA [size] of Thread Storage Directory
               0 [       0] RVA [size] of Load Configuration Directory
               0 [       0] RVA [size] of Bound Import Directory
            2000 [       8] RVA [size] of Import Address Table Directory
               0 [       0] RVA [size] of Delay Import Directory
            2008 [      48] RVA [size] of COM Descriptor Directory
               0 [       0] RVA [size] of Reserved Directory

SECTION HEADER #1
   .text name
     774 virtual size
    2000 virtual address (00402000 to 00402773)
     800 size of raw data
     200 file pointer to raw data (00000200 to 000009FF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read
RAW DATA #1 - SKIPPED
  Debug Directories
```

```
        Time Type       Size      RVA  Pointer
    -------- ------ -------- -------- --------
    519A91C8 cv           57 000026C8      8C8
Format: RSDS, {0B35ACCC-A7E2-459F-92F5-1A98510CE2E8}, 15,
c:\Workspace\SampleApp\SampleApp\obj\x86\Release\SampleApp.pdb

  clr Header:
              48 cb
            2.05 runtime version
            2064 [     648] RVA [size] of MetaData Directory
               3 flags
                   IL Only
                   32-Bit Required
         6000001 entry point token
               0 [       0] RVA [size] of Resources Directory
               0 [       0] RVA [size] of StrongNameSignature Directory
               0 [       0] RVA [size] of CodeManagerTable Directory
               0 [       0] RVA [size] of VTableFixups Directory
               0 [       0] RVA [size] of ExportAddressTableJumps Directory
               0 [       0] RVA [size] of ManagedNativeHeader Directory

  Section contains the following imports:
    mscoree.dll
                402000 Import Address Table
                402748 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference

                    0 _CorExeMain

SECTION HEADER #2
   .rsrc name
     540 virtual size
    4000 virtual address (00404000 to 0040453F)
     600 size of raw data
     A00 file pointer to raw data (00000A00 to 00000FFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only
RAW DATA #2  -SKIPPED

SECTION HEADER #3
  .reloc name
       C virtual size
    6000 virtual address (00406000 to 0040600B)
     200 size of raw data
    1000 file pointer to raw data (00001000 to 000011FF)
```

```
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only
RAW DATA #3 - SKIPPED
BASE RELOCATIONS #3
    2000 RVA,        C SizeOfBlock
     770  HIGHLOW            00402000
       0  ABS
  Summary
        2000 .reloc
        2000 .rsrc
        2000 .text
```

Appendix B

Stack trace of the protected method call

```
--- StackFrame #: 0 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstance
MethodNameFull=System.RuntimeTypeHandle.CreateInstance
MethodSignature=CreateInstance(System.RuntimeType, Boolean, Boolean, Boolean ByRef,
            System.RuntimeMethodHandleInternal ByRef, Boolean ByRef)
MethodSignatureFull=System.Object System.RuntimeTypeHandle.CreateInstance(
            System.RuntimeType, Boolean, Boolean, Boolean ByRef,
            System.RuntimeMethodHandleInternal ByRef, Boolean ByRef)
Namespace=System
ReturnName=System.Object
Text=System.Object System.RuntimeTypeHandle.CreateInstance(System.RuntimeType, Boolean,
            Boolean, Boolean ByRef, System.RuntimeMethodHandleInternal ByRef,
            Boolean ByRef)[]
TypeName=RuntimeTypeHandle
TypeNameFull=System.RuntimeTypeHandle

--- StackFrame #: 1 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstanceSlow
MethodNameFull=System.RuntimeType.CreateInstanceSlow
MethodSignature=CreateInstanceSlow(Boolean, Boolean, Boolean, System.Threading
            .StackCrawlMark ByRef)
MethodSignatureFull=System.Object System.RuntimeType.CreateInstanceSlow(Boolean,
            Boolean, Boolean, System.Threading.StackCrawlMark ByRef)
Namespace=System
ReturnName=System.Object
Text=System.Object System.RuntimeType.CreateInstanceSlow(Boolean, Boolean, Boolean,
            System.Threading.StackCrawlMark ByRef)[]
```

```
TypeName=RuntimeType
TypeNameFull=System.RuntimeType

--- StackFrame #: 2 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstanceDefaultCtor
MethodNameFull=System.RuntimeType.CreateInstanceDefaultCtor
MethodSignature=CreateInstanceDefaultCtor(Boolean, Boolean, Boolean, System
            .Threading.StackCrawlMark ByRef)
MethodSignatureFull=System.Object System.RuntimeType.CreateInstanceDefaultCtor(
            Boolean, Boolean, Boolean, System.Threading.StackCrawlMark ByRef)
Namespace=System
ReturnName=System.Object
Text=System.Object System.RuntimeType.CreateInstanceDefaultCtor(Boolean, Boolean, Boolean,
            System.Threading.StackCrawlMark ByRef)[]
TypeName=RuntimeType
TypeNameFull=System.RuntimeType

--- StackFrame #: 3 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstance
MethodNameFull=System.Activator.CreateInstance
MethodSignature=CreateInstance(System.Type, Boolean)
MethodSignatureFull=System.Object System.Activator.CreateInstance(System.Type, Boolean)
Namespace=System
ReturnName=System.Object
Text=System.Object System.Activator.CreateInstance(System.Type, Boolean)[]
TypeName=Activator
TypeNameFull=System.Activator

--- StackFrame #: 4 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstanceImpl
MethodNameFull=System.RuntimeType.CreateInstanceImpl
MethodSignature=CreateInstanceImpl(System.Reflection.BindingFlags, System.Reflection
            .Binder, System.Object[], System.Globalization.CultureInfo,
            System.Object[], System.Threading.StackCrawlMark ByRef)
MethodSignatureFull=System.Object System.RuntimeType.CreateInstanceImpl(System.Reflection
            .BindingFlags, System.Reflection.Binder, System.Object[],
            System.Globalization.CultureInfo, System.Object[], System.Threading
            .StackCrawlMark ByRef)
Namespace=System
ReturnName=System.Object
Text=System.Object System.RuntimeType.CreateInstanceImpl(System.Reflection.BindingFlags,
            System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo,
            System.Object[], System.Threading.StackCrawlMark ByRef)[]
TypeName=RuntimeType
TypeNameFull=System.RuntimeType

--- StackFrame #: 5 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstance
MethodNameFull=System.Activator.CreateInstance
MethodSignature=CreateInstance(System.Type, System.Reflection.BindingFlags, System
            .Reflection.Binder, System.Object[], System.Globalization.CultureInfo,
            System.Object[])
MethodSignatureFull=System.Object System.Activator.CreateInstance(System.Type,
            System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[],
            System.Globalization.CultureInfo, System.Object[])
Namespace=System
ReturnName=System.Object
Text=System.Object System.Activator.CreateInstance(System.Type, System.Reflection
            .BindingFlags, System.Reflection.Binder, System.Object[], System
.Globalization.CultureInfo, System.Object[])[]
TypeName=Activator
TypeNameFull=System.Activator

--- StackFrame #: 6 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=CreateInstance
MethodNameFull=System.Activator.CreateInstance
MethodSignature=CreateInstance(System.Type, System.Reflection.BindingFlags, System
            .Reflection.Binder, System.Object[], System.Globalization.CultureInfo)
MethodSignatureFull=System.Object System.Activator.CreateInstance(System.Type,
            System.Reflection.BindingFlags, System.Reflection.Binder,
            System.Object[], System.Globalization.CultureInfo)
Namespace=System
ReturnName=System.Object
Text=System.Object System.Activator.CreateInstance(System.Type, System.Reflection
            .BindingFlags, System.Reflection.Binder, System.Object[],
            System.Globalization.CultureInfo)[]
TypeName=Activator
TypeNameFull=System.Activator

--- StackFrame #: 7 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS.A
MethodSignature=A(System.Reflection.MethodBase,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BD[])
MethodSignatureFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rN
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS.A(
            System.Reflection.MethodBase,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BD[])
Namespace=
ReturnName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rN
Text=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2rN
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS.A(
            System.Reflection.MethodBase,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BD[])[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BS

--- StackFrame #: 8 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=C
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BZ.C
MethodSignature=C()
MethodSignatureFull=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BZ.C()
Namespace=
ReturnName=Void
Text=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BZ.C()[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BZ
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BE+
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BZ

--- StackFrame #: 9 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=I
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi.I
MethodSignature=I()
MethodSignatureFull=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi.I()
Namespace=
ReturnName=Void
Text=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi.I()[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bi

--- StackFrame #: 10 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2F.A
MethodSignature=A(Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)
MethodSignatureFull=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2F.A(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)
Namespace=
ReturnName=Void
Text=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2F.A(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2F
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2F

--- StackFrame #: 11 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=E
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v.E
MethodSignature=E(Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)
MethodSignatureFull=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v.E(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)
Namespace=
ReturnName=Void
Text=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v.E(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v

--- StackFrame #: 12 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
MethodSignature=A(Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v)
MethodSignatureFull=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v)
Namespace=
ReturnName=Void
Text=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2v)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr

--- StackFrame #: 13 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
MethodSignature=A(Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)
MethodSignatureFull=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)
Namespace=
ReturnName=Void
Text=Void Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BB)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr

--- StackFrame #: 14 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
MethodSignature=A(Slps.Engine.Execution.Internal.IExecutionEngineParams,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BL)
MethodSignatureFull=
            System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Slps.Engine.Execution.Internal.IExecutionEngineParams,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BL)
Namespace=
ReturnName=System.Object
Text=System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Slps.Engine.Execution.Internal.IExecutionEngineParams,
Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2BL)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr

--- StackFrame #: 15 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A
MethodSignature=A(Byte[], Slps.Engine.Execution.Internal.IExecutionEngineParams,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bo, Boolean)
MethodSignatureFull=
            System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Byte[], Slps.Engine.Execution.Internal.IExecutionEngineParams,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bo, Boolean)
Namespace=
ReturnName=System.Object
Text=System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr.A(
            Byte[], Slps.Engine.Execution.Internal.IExecutionEngineParams,
            Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2Bo, Boolean)[]
```

```
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2wr

--- StackFrame #: 16 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A
MethodSignature=A(System.Reflection.Assembly, Byte[],
            Slps.Engine.Execution.Internal.IExecutionEngineParams, Boolean)
MethodSignatureFull=
            System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A(
            System.Reflection.Assembly, Byte[],
            Slps.Engine.Execution.Internal.IExecutionEngineParams, Boolean)
Namespace=
ReturnName=System.Object
Text=System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A(
            System.Reflection.Assembly, Byte[],
            Slps.Engine.Execution.Internal.IExecutionEngineParams, Boolean)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft

--- StackFrame #: 17 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=A
MethodNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A
MethodSignature=A(System.Reflection.Assembly, System.String,
            Slps.Engine.Execution.Internal.IExecutionEngineParams)
MethodSignatureFull=
            System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A(
            System.Reflection.Assembly, System.String,
            Slps.Engine.Execution.Internal.IExecutionEngineParams)
Namespace=
ReturnName=System.Object
Text=System.Object Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft.A(
            System.Reflection.Assembly, System.String,
            Slps.Engine.Execution.Internal.IExecutionEngineParams)[]
TypeName=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft
TypeNameFull=Microsoft_Licensing_Permutation_5538c_2_0_3_2_1941_2ft

--- StackFrame #: 18 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=InternalExecuteMethod
MethodNameFull=Microsoft.Licensing.SLMRuntime.InternalExecuteMethod
MethodSignature=InternalExecuteMethod(System.Reflection.Assembly, System.String,
            Slps.Engine.Execution.Internal.IExecutionEngineParams)
```

```
MethodSignatureFull=System.Object Microsoft.Licensing.SLMRuntime.InternalExecuteMethod(
            System.Reflection.Assembly, System.String,
Slps.Engine.Execution.Internal.IExecutionEngineParams)
Namespace=Microsoft.Licensing
ReturnName=System.Object
Text=System.Object Microsoft.Licensing.SLMRuntime.InternalExecuteMethod(
            System.Reflection.Assembly, System.String,
            Slps.Engine.Execution.Internal.IExecutionEngineParams)[]
TypeName=SLMRuntime
TypeNameFull=Microsoft.Licensing.SLMRuntime

--- StackFrame #: 19 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=ExecuteMethod
MethodNameFull=Microsoft.Licensing.SLMRuntime.ExecuteMethod
MethodSignature=ExecuteMethod(System.Object, System.String, System.String,
            System.Object[], System.Type[], System.Type[])
MethodSignatureFull=
            System.Object Microsoft.Licensing.SLMRuntime.ExecuteMethod(
            System.Object, System.String, System.String, System.Object[],
            System.Type[], System.Type[])
Namespace=Microsoft.Licensing
ReturnName=System.Object
Text=System.Object Microsoft.Licensing.SLMRuntime.ExecuteMethod(System.Object,
            System.String, System.String, System.Object[], System.Type[],
            System.Type[])[]
TypeName=SLMRuntime
TypeNameFull=Microsoft.Licensing.SLMRuntime

--- StackFrame #: 20 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=btnSTMessageBox_Click
MethodNameFull=WinForms.Form1.btnSTMessageBox_Click
MethodSignature=btnSTMessageBox_Click(System.Object, System.EventArgs)
MethodSignatureFull=
            Void WinForms.Form1.btnSTMessageBox_Click(System.Object, System.EventArgs)
Namespace=WinForms
ReturnName=Void
Text=Void WinForms.Form1.btnSTMessageBox_Click(System.Object, System.EventArgs)[]
TypeName=Form1
TypeNameFull=WinForms.Form1

--- StackFrame #: 21 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=OnClick
```

```
MethodNameFull=System.Windows.Forms.Control.OnClick
MethodSignature=OnClick(System.EventArgs)
MethodSignatureFull=Void System.Windows.Forms.Control.OnClick(System.EventArgs)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Control.OnClick(System.EventArgs)[]
TypeName=Control
TypeNameFull=System.Windows.Forms.Control

--- StackFrame #: 22 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=OnClick
MethodNameFull=System.Windows.Forms.Button.OnClick
MethodSignature=OnClick(System.EventArgs)
MethodSignatureFull=Void System.Windows.Forms.Button.OnClick(System.EventArgs)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Button.OnClick(System.EventArgs)[]
TypeName=Button
TypeNameFull=System.Windows.Forms.Button

--- StackFrame #: 23 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=OnMouseUp
MethodNameFull=System.Windows.Forms.Button.OnMouseUp
MethodSignature=OnMouseUp(System.Windows.Forms.MouseEventArgs)
MethodSignatureFull=Void System.Windows.Forms.Button.OnMouseUp(
            System.Windows.Forms.MouseEventArgs)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Button.OnMouseUp(System.Windows.Forms.MouseEventArgs)[]
TypeName=Button
TypeNameFull=System.Windows.Forms.Button

--- StackFrame #: 24 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=WmMouseUp
MethodNameFull=System.Windows.Forms.Control.WmMouseUp
MethodSignature=WmMouseUp(System.Windows.Forms.Message ByRef,
            System.Windows.Forms.MouseButtons, Int32)
MethodSignatureFull=Void System.Windows.Forms.Control.WmMouseUp(
            System.Windows.Forms.Message ByRef,
            System.Windows.Forms.MouseButtons, Int32)
Namespace=System.Windows.Forms
ReturnName=Void
```

```
Text=Void System.Windows.Forms.Control.WmMouseUp(
            System.Windows.Forms.Message ByRef,
            System.Windows.Forms.MouseButtons, Int32)[]
TypeName=Control
TypeNameFull=System.Windows.Forms.Control

--- StackFrame #: 25 - 11/7/2013 4:40:45 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=WndProc
MethodNameFull=System.Windows.Forms.Control.WndProc
MethodSignature=WndProc(System.Windows.Forms.Message ByRef)
MethodSignatureFull=Void System.Windows.Forms.Control.WndProc(
            System.Windows.Forms.Message ByRef)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Control.WndProc(
            System.Windows.Forms.Message ByRef)[]
TypeName=Control
TypeNameFull=System.Windows.Forms.Control

--- StackFrame #: 26 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=WndProc
MethodNameFull=System.Windows.Forms.ButtonBase.WndProc
MethodSignature=WndProc(System.Windows.Forms.Message ByRef)
MethodSignatureFull=Void System.Windows.Forms.ButtonBase.WndProc(
            System.Windows.Forms.Message ByRef)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.ButtonBase.WndProc(
            System.Windows.Forms.Message ByRef)[]
TypeName=ButtonBase
TypeNameFull=System.Windows.Forms.ButtonBase

--- StackFrame #: 27 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=WndProc
MethodNameFull=System.Windows.Forms.Button.WndProc
MethodSignature=WndProc(System.Windows.Forms.Message ByRef)
MethodSignatureFull=Void System.Windows.Forms.Button.WndProc(
            System.Windows.Forms.Message ByRef)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Button.WndProc(
            System.Windows.Forms.Message ByRef)[]
```

```
TypeName=Button
TypeNameFull=System.Windows.Forms.Button

--- StackFrame #: 28 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=OnMessage
MethodNameFull=System.Windows.Forms.Control+ControlNativeWindow.OnMessage
MethodSignature=OnMessage(System.Windows.Forms.Message ByRef)
MethodSignatureFull=Void System.Windows.Forms.Control+
            ControlNativeWindow.OnMessage(
            System.Windows.Forms.Message ByRef)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Control+ControlNativeWindow.OnMessage(
            System.Windows.Forms.Message ByRef)[]
TypeName=ControlNativeWindow
TypeNameFull=System.Windows.Forms.Control+ControlNativeWindow

--- StackFrame #: 29 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=WndProc
MethodNameFull=System.Windows.Forms.Control+ControlNativeWindow.WndProc
MethodSignature=WndProc(System.Windows.Forms.Message ByRef)
MethodSignatureFull=Void System.Windows.Forms.Control+
            ControlNativeWindow.WndProc(
            System.Windows.Forms.Message ByRef)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Control+ControlNativeWindow.WndProc(
            System.Windows.Forms.Message ByRef)[]
TypeName=ControlNativeWindow
TypeNameFull=System.Windows.Forms.Control+ControlNativeWindow

--- StackFrame #: 30 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=Callback
MethodNameFull=System.Windows.Forms.NativeWindow.Callback
MethodSignature=Callback(IntPtr, Int32, IntPtr, IntPtr)
MethodSignatureFull=IntPtr System.Windows.Forms.NativeWindow.Callback(
            IntPtr, Int32, IntPtr, IntPtr)
Namespace=System.Windows.Forms
ReturnName=IntPtr
Text=IntPtr System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32,
            IntPtr, IntPtr)[]
TypeName=NativeWindow
```

```
TypeNameFull=System.Windows.Forms.NativeWindow

--- StackFrame #: 31 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=DispatchMessageW
MethodNameFull=System.Windows.Forms.UnsafeNativeMethods.DispatchMessageW
MethodSignature=DispatchMessageW(MSG ByRef)
MethodSignatureFull=
            IntPtr System.Windows.Forms.UnsafeNativeMethods.DispatchMessageW(MSG ByRef)
Namespace=System.Windows.Forms
ReturnName=IntPtr
Text=IntPtr System.Windows.Forms.UnsafeNativeMethods.DispatchMessageW(MSG ByRef)[]
TypeName=UnsafeNativeMethods
TypeNameFull=System.Windows.Forms.UnsafeNativeMethods

--- StackFrame #: 32 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=System.Windows.Forms.UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop
MethodNameFull=System.Windows.Forms.Application+ComponentManager.System.Windows.Forms
            .UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop
MethodSignature=System.Windows.Forms.UnsafeNativeMethods.IMsoComponentManager
            .FPushMessageLoop(IntPtr, Int32, Int32)
MethodSignatureFull=Boolean System.Windows.Forms.Application+ComponentManager.System
            .Windows.Forms.UnsafeNativeMethods.IMsoComponentManager
            .FPushMessageLoop(IntPtr, Int32, Int32)
Namespace=System.Windows.Forms
ReturnName=Boolean
Text=Boolean System.Windows.Forms.Application+ComponentManager.System.Windows.Forms
            .UnsafeNativeMethods.IMsoComponentManager.FPushMessageLoop(IntPtr, Int32,
            Int32)[]
TypeName=ComponentManager
TypeNameFull=System.Windows.Forms.Application+ComponentManager

--- StackFrame #: 33 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=RunMessageLoopInner
MethodNameFull=System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner
MethodSignature=RunMessageLoopInner(Int32, System.Windows.Forms.ApplicationContext)
MethodSignatureFull=Void System.Windows.Forms.Application+ThreadContext
            .RunMessageLoopInner(Int32, System.Windows.Forms.ApplicationContext)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Application+ThreadContext.RunMessageLoopInner(
            Int32, System.Windows.Forms.ApplicationContext)[]
TypeName=ThreadContext
```

```
TypeNameFull=System.Windows.Forms.Application+ThreadContext

--- StackFrame #: 34 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=RunMessageLoop
MethodNameFull=System.Windows.Forms.Application+ThreadContext.RunMessageLoop
MethodSignature=RunMessageLoop(Int32, System.Windows.Forms.ApplicationContext)
MethodSignatureFull=Void System.Windows.Forms.Application+ThreadContext
            .RunMessageLoop(Int32, System.Windows.Forms.ApplicationContext)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32,
            System.Windows.Forms.ApplicationContext)[]
TypeName=ThreadContext
TypeNameFull=System.Windows.Forms.Application+ThreadContext

--- StackFrame #: 35 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=Run
MethodNameFull=System.Windows.Forms.Application.Run
MethodSignature=Run(System.Windows.Forms.Form)
MethodSignatureFull=Void System.Windows.Forms.Application.Run(System.Windows.Forms.Form)
Namespace=System.Windows.Forms
ReturnName=Void
Text=Void System.Windows.Forms.Application.Run(System.Windows.Forms.Form)[]
TypeName=Application
TypeNameFull=System.Windows.Forms.Application

--- StackFrame #: 36 - 11/7/2013 4:40:46 PM ---
nFileName=
FilePath=
LineNumber=0
MethodName=Main
MethodNameFull=WinForms.Program.Main
MethodSignature=Main()
MethodSignatureFull=Void WinForms.Program.Main()
Namespace=WinForms
ReturnName=Void
Text=Void WinForms.Program.Main()[]
TypeName=Program
TypeNameFull=WinForms.Program
```