Java Security

Mike Reiter

Based on Securing Java by G. McGraw and E. Felten.

Mobile Code and Java

- Mobile code
  - Programs that travel across network and run in target computer

- Advantages
  - Support users with weak communication devices
  - Activate interactions in the WWW
  - Enable component-based software development

- Java: a programming language for mobile code
  - Cross-platform: byte codes interpreted by virtual machine
  - “write once, run anywhere” ---SUN
  - Java applet: embedded program in web pages

- Other mobile code systems:
  - ActiveX, TCL, JavaScript, Word Macros, Postscript, etc.
Mobile Code Security

- Security risks of mobile code
  - modification
  - compromise of privacy
  - DoS
  - antagonism

- Two major security problems
  - Protect computers from mobile code
  - Protect mobile code from computers

Securing Java

- Mainstream browsers have JVMs and can download and execute applets automatically
- Fortunately, Java designers take security seriously
  - Java has built-in access control: public, private, protected and default
  - Java has cryptographic APIs and security models
- Java security models
  - Original model: Make untrusted code run inside a “box” and limit its ability to do risky things
  - Code signing and access control
    - Mobile code can be digitally signed
    - Access granted to computing resources according to trustworthiness of the signer
Potential threats

- Applets can mount four types of attacks on clients
  - **System modification**: Applets illegally alter the client’s computing system. For example, delete files, change memory, etc.
  - **Invasion of privacy**: Applets steal client’s confidential data. For example, password, credit card numbers, etc.
  - **Denial of service**: Applets exhaust client’s computing resources. For example, filling a file system, allocating all of a system’s memory, etc.
  - **Antagonism**: Applets annoy the user. For example, make noise, open several windows, etc.

- Current Java security models are robust (relatively) to the first two attacks while vulnerable to the other two

---

Type Safety

- **Java programs are composed of classes**
  - A *class* defines a collection of data fields and functions (methods) that operate on those fields
  - Instances of classes are *objects*

- **Type safety**
  - Only objects for which the program has a reference can be accessed
    - References cannot be forged
    - Memory cannot be accessed directly
  - A program can perform an operation on an object only if that operation is valid for that object

- **Type safety is the cornerstone of Java security**
Why Type Safety Matters

Consider the following example
- "Alarm" class defines a method "turnOn" that sets a field to "true"
- "Applet" class has fields indicating permissions

If turnOn could be invoked on Applet, then it might set fileAccessAllowed to true (!)

Architecture

The basic security model is composed by three components
- bytecode verifier: helps ensure type safety
- class loader: loads and unloads classes dynamically from the Java runtime environment
- security manager: guards potentially dangerous functionality

Security is dependent on all three components working correctly
Bytecode Verifier

- Objective: Check if the code is valid.
- What will be verified
  - Correct format, type safety (through built-in theorem prover) and compatibility with dynamically loaded classes

- Verification strategy: Do the verification during class loading (loading byte code into the VM) and before execution
  - This improves the efficiency of Java program execution
- Two major verification steps
  - Internal checks and runtime checks

---

Bytecode Verifier

- Static checks
  - Java labels every object by putting a class tag next to the object
  - The verifier statically checks the operations on the tagged objects
  - Other various checks (e.g., proper format, final classes not subclassed)

- Plus runtime checks
  - VM loads in the definition of any not-yet-loaded classes
  - VM replaces reference-by-name with reference to specific class object
  - If object does not exist or reference is illegal, VM throws an exception
Class Loader

- Loads byte codes into VM and construct corresponding classes
- More specifically
  - Load byte code once the VM looks for a specific class
  - Manage namespaces
    - Applets have separated namespace for their classes
- Potential security breach: another example for the tradeoff between security and functionality
  - Java’s class loader architecture was originally meant to be extensible
  - Users can add their own class loaders into a running environment
  - But, attacker may insert a class loader into your VM which will load a security manager for you. This is known as class spoofing.
  - As a result, current Java implementations prohibit untrusted code from making class loaders.

Class Loader (cont.)

- Two types of class loaders
  - Primordial class loader: bootstrap the Java environment, e.g., loading API classes.
  - Class loader objects (written in Java, can be extended): load other classes
- Each class loader defines a name space
  - An applet gets its own class loader, can only “see” its own classes
  - Different applets can define classes with the same name
Loading Procedure

1. Determine whether the class has been loaded before. If so, return the previously loaded class.
2. Consult the Primordial Class Loader to attempt to load the class from the CLASSPATH.
3. See whether the Class Loader is allowed to create the class being loaded. The Security Manager makes this decision. If not, throw a security exception.
4. Read the class file into an array of bytes. The way this happens differs according to particular class loaders. Some class loaders may load classes from a local database. Others may load classes across the network.
5. Construct a Class object and its methods from the class file.
6. Resolve classes immediately referenced by the class before it is used. These classes include classes used by static initializers of the class and any classes that the class extends.
7. Check the class file with the Verifier.

Security Manager

- Security manager essentially is a reference monitor
  - Decides if an applet’s requested operation should be allowed
- When an applet makes a potentially dangerous request
  - Java API code invokes the Security Manager
  - Security Manager throws a Security Exception if operation is denied
  - If operation is permitted, the Security Manager returns without exception
    - API code performs requested operation
Security Manager Trust Model

- Trust model has become increasingly refined over time

- **JDK 1.0.2**
  - Untrusted = any applet
  - Trusted = application

- **JDK 1.1**
  - Untrusted = unsigned applet
  - Trusted = signed applet or application

- **JDK 1.2 and beyond**
  - Many “shades of gray” in between

---

Security Manager Trust Model (cont.)

- **What untrusted code can do**
  - Access CPU and memory to build its objects and execute
  - Connect to the web server from which the applet was downloaded

- **What untrusted code can’t do**
  - Operations on the client’s file system such as read, write, delete or rename a file
  - Connect to destinations other than its origin
  - Making critical system calls, such as system.exit()
  - Creating a classloader or security manager
  - Other dangerous actions, e.g., manipulate other threads

---
Security Manager Duties

1. Prevent installation of new class loaders.
2. Protect threads and thread groups from each other.
3. Control the execution of other application programs.
4. Control the ability to shut down the VM.
5. Control access to other application processes.
6. Control access to system resources such as print queues, clipboards, event queues, system properties, and windows.
7. Control file system operations such as read, write, and delete. Access to local files is strictly controlled.
8. Control network socket operations such as connect and accept.

Code Signing and Access Control

- **Extension of basic security model: code signing**
  - JDK 1.0.2 security model trusts code only according to “applet” or “application”
  - JDK 1.1 introduced code signing: Allows VM to trust code according to the trustworthiness of the signer

- **The main objectives:**
  - Gain a fine-grained control over mobile code
    - in JDK 1.1 a signed applet also can be trusted
  - Principle of least privilege: each applet has the minimal privilege it needs to complete its task
Stack Inspection

- When security manager evaluates a request, what code does it hold responsible for the request?

- Consider a request to open file
  - There are possibly many layers of “library” code between the untrusted applet and the point at which the Security Manager is invoked
  - Which of these calls should be granted?

- Consider, for example, an application like a browser
  - “Parts” of the browser have to read and write cache files
  - Other “parts” (e.g., untrusted applets) should not
  - Both probably open files via intermediate Java code

Simplified Stack Inspection

- Assume two principals (“system” & “untrusted”) and one privilege
- Each Java thread has a runtime stack that tracks method calls.
- Every stack frame is labeled with principal and a privilege tag.
  - A system class can set the tag while untrusted code cannot.
  - Whenever a frame completes its work, the tag disappears.
- Per access request
  - Algorithm searches frames from newest to oldest
  - If a frame with “full” tag is first found, then access is permitted
  - If an untrusted frame is encountered, access is denied
Stack Inspection Operations

- **enablePrivilege()**
  - When code wants to use a resource \( R \), it must first call `enablePrivilege(R)`
  - If permitted, the current stack frame is annotated with an enabled privilege mark.

- **revertPrivilege()**
  - Removes annotation from stack frame

- **disablePrivilege()**
  - Creates a stack annotation to hide the earlier enabled privilege

- **checkPrivilege()**
  - Searches stack from the newest to the oldest to check if a frame has proper privilege to make a specific system call

Stack Inspection Algorithm

```java
checkPrivilege (target) {
   // loop, newest to oldest stack frame
   foreach stackFrame {
      if (local policy forbids access to target by class executing in stackFrame)
         throw ForbiddenException;
      if (stackFrame has enabled privilege for target)
         return; // allow access
      if (stackFrame has disabled privilege for target)
         throw ForbiddenException;
   }
   // if we reached here, we fell off the end of the stack
   if (Netscape 4.0)
      throw ForbiddenException;
   if (Microsoft IE 4.0 || Sun JDK 1.2)
      return; // allow access
}
```
Java 2 Security Policy

- **Identity**
  - In Java 2, there are two identity-defining characteristics: Origin (where the code comes from) and Signature (who vouches for it).
  - Origin and Signature are represented with `java.security.CodeSource`

- **The abstract class `java.security.Permission` types and parameterizes access permissions granted to classes**
  - Permissions can be subclassed from this class (and its subclasses)
  - Permissions include:
    - `java.io.FilePermission` for file system access
    - `java.net.SocketPermission` for network access
    - `java.lang.PropertyPermission` for Java properties
    - `java.lang.RuntimePermission` for access to runtime system
    - `java.security.NetPermission` for authentication
    - `java.awt.AWTPermission` for access to graphical resources
  - An example of a file permission:
    ```java
    p = new FilePermission("/applets/tmp/scratch","read");
    ```

- **Policy: A mapping from identity to permissions**
  - The policy object is a runtime representation of policy usually set up by the VM at startup time (much like the Security Manager). E.g.,
    ```java
    grant CodeBase "https://www.rstcorp.com/users/gem",
      SignedBy "*" {
        permission java.io.FilePermission "read,write",
          "/applets/tmp/*/";
        permission java.net.SocketPermission "connect",
          "*.rstcorp.com";
    }
    ```
  - Sun Java 2 policy can be set by users (bad idea) or system administrators.

- **Permissions are additive**
Malicious Applets

- Java provides impressive access control facilities, but many types of attack are still possible
  - Harassment
  - Stealing CPU cycles

- Java's security model cannot completely eliminate such attacks.

- In addition, bugs in implementations of Java security have permitted other attacks in the past (now fixed)
  - Invading your privacy (e.g., monitoring the web pages you visit)
  - Forge mail from your computer to masquerade as you

DoS Applets

- Example of a malicious (DoS) applet:
  - Create an applet that starts a thread with its priority set to MAX_PRIORITY
  - Redefine the stop() method.
  - Do something innocent (draw a picture), then sleep for a while.
  - Once the thread wakes up, do some CPU-intensive activity in an infinite loop.
    - This will bring down the browser.

- DoS applet may do some other annoying things with the same techniques
  - For example, pop up windows indefinitely
Other Examples of DoS Applets

- **Business assassin applet**
  - An implementation bug through JDK 1.1.5 allows applets to terminate each other
  - Several have implemented “business assassin” applets
    - The applet activates a thread watching silently for other applets being downloaded.
    - If an applet from a specific website is identified, the thread kills it. The attack blocks you from downloading any code from the site.

Type Confusion

- In type-confusion attack, an applet utilizes a bug in the JVM to create two references with different types to same object
  - E.g., suppose type $T$ has a private variable “privilege” set by JVM, while type $U$ has a public variable called “privilege”. Thus, applet can freely set its own privilege.

- It has been demonstrated that any type confusion can be leveraged to disarm Java security.
Type Confusion and Class Loading

Recall that a class loader translates a class name into its identity and fetches its byte code (maybe across the network).

1. Java calls Class Loader's loadClass method, passing it the name to look up.
2. loadClass consults internal dictionary. If one exists, that class is returned.
3. If no class with the requested name, it tries to find one.
4. After getting the byte code, the Class Loader calls defineClass() to turn the byte code into a usable class.
5. When defineClass is finished, returns the resulting class to Java.

If an applet circumvents the security manager to insert an evil class loader, the class loader can map different types to the same object in memory.

Type Confusion and Class Loaders

Java security prohibits applets from creating class loaders

- To make a class loader, the Classloader class must be extended.
- Constructor of Classloader calls security manager. Thus, any efforts to illegally create a class loader will be intercepted.

However, there have been problems ...

- Researchers discovered a way to generate a Classloader object without calling its superclass’ constructor

Sun and Netscape had two choices to fix the problem

- fixing the verifier to forbid an applet to extend Classloader
- forcing the Classloader object to call its superclass’ constructor

They chose the latter

- make defineClass() private and permit it to be executed only if a field is set indicating that the superclass constructor was run
Type Confusion and Interface Casting

- Interface management is also a source of type confusion

```java
interface Inter { void f(); }
class Secure implements Inter { private void f(); }
class Dummy extends Secure implements Inter {
    public void f();
    Dummy() {
        Secure s = new Secure();
        Inter i = (Inter) s;
        i.f(); // should be illegal }
}
```

- This led to a type confusion
  - The private function `f()` can be called by other classes.

Type Confusion and Interface Casting

```java
interface Inter { void f(); }
class Secure implements Inter { private void f(); }
class Dummy implements Inter {
    public void f();
    static void attack() {
        Inter inter[2] = { new Dummy(), new Secure() };
        for(int j=0; j<2; ++j) inter[j].f();
    }
}
```

- To improve performance, Java only checked the first component of the array.
- Problems like this led to Netscape overhauling its implementation.
Problems with Code Signing

- In earlier versions, an attack applet was able to commandeer the privileges of signed applets:
  - A method `Class.getSigners()` returned a mutable array of all the signers that signed a particular class
  - An applet could revise such an array to get whoever it wants as its signer
  - The fix: give caller a copy of the array rather than the array itself.

More Recent Troubles

- 2012 and 2013 were bad years for Java, owing to 0-day vulnerabilities that were being exploited in the wild
- Some were pretty complex, e.g., CVE-2013-0422:

  Multiple vulnerabilities in Oracle Java 7 before Update 11 allow remote attackers to execute arbitrary code by (1) using the public `getMBeanInstantiator` method in the `JmxMBeanServer` class to obtain a reference to a private `MBeanInstantiator` object, then retrieving arbitrary `Class` references using the `findClass` method, and (2) using the Reflection API with recursion in a way that bypasses a security check by the `java.lang.invoke.MethodHandles.Lookup.checkSecurityManager` method due to the inability of the `sun.reflect.Reflection.getCallerClass` method to skip frames related to the new reflection API, ...

- These problems led to disabling Java by default in popular browsers
Conclusion

- Java has lots to offer in terms of a language for writing secure programs (though it's still not perfect)

- It is far superior to C/C++ for writing secure code
  - Primarily due to type safety and rich permissions support

- Presence of security-relevant bugs has been dramatically reduced since Java’s introduction