Lecture 11: Java Security model, part II

Lecture topics

- Java security model, part II
  - Class loaders
  - Security manager

The function of class loaders

- Read files containing Java bytecodes and convert them into class definitions
- Central to Java security
  - A lot of security-related checking has to happen at the time untrusted classes are loaded
  - Work in conjunction with access controller and security manager
- Class loaders can be extended to provide additional application-specific functionality
  - Handle protocols other than HTTP
  - Implement class file encryption
  - Implement special security policies
Lecture 11: Java Security model, part II

When a class is loaded

- Class loading of a class X is triggered by the first active use of class X
  - A new instance of X is created
  - A new statement
  - Reflection
  - Cloning
  - Deserialization
  - When a static method of X is invoked
  - When a static field of X is read or written
  - When a method in java.lang.Class or java.lang.reflect.* needs information about X
  - When a subclass of X is initialized

Why not do class loading at start-up?

- OK, so start-up will be even longer, but after that the program will run faster...
- Depending on inputs, some classes may never be actually loaded
- More importantly, sometimes classes to be loaded cannot be known until run time, if reflection is used:

```java
public class LoadAndExecute {
    public static void main(String [] args) {
        Class cl = Class.forName(args[0]);
        Runnable r = (Runnable) cl.newInstance();
        r.run();
    }
}
```
Lecture 11: Java Security model, part II

Class loader architecture

- Class loaders are created automatically by the JVM
  - Bootstrap class loader: Core classes
  - Extension class loader: Installed extension classes
  - System class loader: Classes on the class path
- User-defined class loaders created by the program
  - The system class loader is the parent class loader by default
  - Another parent class loader can be specified explicitly

Handling threads

- A context class loader is created for each thread
  - If a thread needs to load a class and a class loader is not specified explicitly (e.g., a new statement), the thread's context class loader is used
  - Default context class loader is the system class loader
  - A different context class loader can be specified by calling Thread.setContextClassLoader()
Lecture 11: Java Security model, part II

### Class loader delegation model

- When asked to load a class, a class loader first asks its **parent** (class loader that loaded it) to load the class
- If the parent succeeds, the class loader returns the class from the parent
- If the parent fails, the class loader attempts to load the class itself
- The reason for this is so that the same class is not loaded separately by two different loaders (that would have negative security implications)

```java
SomeClassLoader loader = new SomeClassLoader(url);
Class stringClass = cl.loadClass("java.lang.String");
```

### So, how do we decide/know which class loader loads which class?

- Let `c` be the result of call to `l.loadClass(...)`
  - `l` is the **defining class loader** of `c`
    - Assuming that no delegation happened
  - `l` is the **initiating class loader** of `c`
    - If it delegated loading `c` to some other class loader
- `l` will be asked to load all classes referenced by `c`
  - `l` can delegate these loads

```java
package com.evil.attack;
public class AttackClass {
    java.lang.String field;
    ...
}
```

![Diagram showing class loader delegation model](image)

![Diagram showing class loader decision](image)
A class with the same name should not be used across namespaces

- A bug from Java 1.1

```java
class AttackC {
    void m() {
        SecureC x = D.f();
        System.out.println(x.secret);
    }
}
```

```java
class D {
    SecureC f() {...}
}
```

```java
class SecureC {
    public int secret;
    ...
}
```

```java
class SecureC {
    private int secret;
    ...
}
```

L1 delegates loading of D to L2

Avoiding the problem

- JVM doesn’t trust user-defined loaders to return the same type for the given name
- JVM maintains loaded class cache
  - Maps loaded class names and initiating loaders to class types
- Upon loadClass method, JVM
  - Checks if the class with the given name has already been loaded; if yes, throws an exception
Class loaders and namespaces

- **What’s a namespace?**
  - The full name of a Java class is qualified by the name of the package to which the class belongs
    
    E.g. there is no standard class called `String` in the Java API, but there is the class `java.lang.String`

- **But what if different classes with the same fully qualified names are loaded from different URLs**

- **Namespaces are used to avoid ambiguity between classes with the same name**
  - Enforce package protection

- **Classes with different codebases are loaded by different class loader instances**
  - These instances may be of the same or different types

---

Once the decision is made that the class must be loaded

- **Class loader reads in an array of bytes**
- **Bytecode verification is performed**
- **A class object is constructed from bytecodes**
- **Checks that the class name is the same as the requested class name**
- **Base classes and classes referenced by static initializers are also loaded**
  - Other referenced classes are loaded when the class instances actually reference them at runtime
Lecture 11: Java Security model, part II

**What is loaded when by whom (Java 2)**

- An internal class loader (part of the JVM) loads the Java API classes when the JVM starts up
- An instance of `URLClassLoader` loads classes from the classpath
  - At run time, when referenced
- If RMI is used, an instance of `RMIClassLoader` loads classes from the codebase
  - Not a subclass of `ClassLoader`
  - Provides only static methods
- Secure class loader (`SecureClassLoader`) associates protection domains with each class it loads
  - Uses `AccessController`
  - Application programmers wishing to create their own class loaders are advised to subclass this loader or its subclasses

**Classfile verification -- motivation**

- The code may not have been generated by a trusted compiler
- The code may not have been originated from Java source
- Other classes may have changed since this class was compiled
- Load-time verification reduces run-time overheads
Lecture 11: Java Security model, part II

**Verification at load time**

- Verify that the file is a .class file and that all section/attributes are well-formed and of the proper length
- Verify the non-code portions of the class
  - Final methods/classes are not overridden
  - Every class has a superclass
  - The constant pool is valid and contains the necessary values
- Verify the code
  - The size and types of the operand stack are fixed at any point and no overflow or underflow occurs
  - Local variables are valid and of the correct type
  - Methods are invoked with the appropriate arguments
  - Fields are assigned values of the correct type
  - Arguments to JVM opcodes are of correct types

**Verification at run time**

- Verify that the current class can access fields and methods in other classes
  - Attempt to perform these checks only once
- Verify that an array access is within bounds
- Verify that an object assigned to a variable or field is of compatible type
  - That’s why you see ClassCastException from time to time...
- ...

Page <##>
Class loaders and security enforcement

- Class loaders are a piece of Java security architecture that is in charge of associating classes with their origins
  - URLs, digital signatures
- Note that class loaders by themselves do not enforce security policy
  - They just provide the security manager with sufficient information to enable that

Class loaders and JAR files

- JAR files are archives (ZIP format) that may include many class files
  - Can be digitally signed
- All classes in a JAR are loaded at once
Lecture 11: Java Security model, part II

**Custom class loader skeleton**

```java
import java.security.SecureClassLoader;
public class CustomClassLoader extends SecureClassLoader {
    public CustomClassLoader() {
        super(Thread.currentThread().getContextClassLoader());
    }

    public CustomClassLoader(ClassLoader parent) {
        super(parent);
    }

    protected Class findClass(String name) throws ClassNotFoundException {
        ... byte [] bytecode = retrieveClass(name);
        return defineClass(name, bytecode, 0, bytecode.length);
    }

    private byte [] retrieveClass(String name) {...}
}
```

**Security manager**

- An object that is in charge of "guarding the sandbox"
  - Prevents untrusted classes from doing dangerous things
- Default security manager is provided
  - Simply uses AccessController
- By default, applications have no security manager
- Default trusted classes:
  - Core API
  - Classes loaded from the classpath
Lecture 11: Java Security model, part II

What the security manager is supposed to do for untrusted code

- Prevent installation of new class loaders
  - Avoids spoofing library classes
- Protect threads and thread groups from each other
- Control the ability to shut down the VM
- Control access to other application processes
- Control access to system resources
  - print queues
  - Clipboard
  - event queues
  - system properties
  - Windowing system
- Control file system operations
- Control network socket operations
- Control access to Java packages

Accessing the security manager

- Only one security manager can be active at any given time
- Use the static method of System to obtain the active security manager:
  
  SecurityManager manager = System.getSecurityManager();
  if (manager != null) {
    manager.checkPermission(...);
  }

- A new security manager is set by calling another static method of System:
  
  System.setSecurityManager(new RMISecurityManager());

- The ability to set a security manager should be protected by permissions
Some methods from SecurityManager API

- checkPermission
  - The default implementation is a call to AccessController.checkPermission
- checkPrintJobAccess
- checkWrite
- checkExec
- ...

Lecture 11: Java Security model, part II