Java Security

"Securing Java", by Gary McGraw and Edward W. Felten, Wiley, 1999.

Java was designed with some security aspects in mind:

1. At the low level, Java byte code can be verified for JVM code conformance. This ensures that the code behaves correctly.
2. Security can be introduced using the `java.lang.SecurityManager` class.

   A security policy is put in place by instantiating a `SecurityManager` object and registering it with `System.setSecurityManager()`.

3. Another important aspect of security concerns the way classes are loaded, as defined by the `java.lang.ClassLoader` class.

   A subclass of this can be implemented to enforce specific protocols.

   In particular, it is important to distinguish between system classes, which can be trusted, and classes that are imported, which cannot be trusted.

Java introduced the idea of a **sandbox**: make untrusted code run inside a protected area which limits its ability to do risky things.

In JDK 1.0, system classes and classes loaded from the `CLASSPATH` were trusted, whereas classes loaded from a URL were untrusted.

Also, browsers imposed very stringent restrictions on imported applets:

  no access to the local file system;

  no network access to any computer other than the host from which the applet originated;

  no access to system properties or commands.

On the other hand, applications usually had the null security manager.

The very strict limitations on applets in JDK 1.0 meant that they were not very useful. A way needed to be found for applets to escape from the sandbox in a controlled way.

JDK 1.1 introduced the concept of signed applets. An applet became trusted if it was digitally signed by a subject that the user (or browser) trusted.

However, there is still a once-and-for-all decision about whether an applet is trusted, and the security is all-or-nothing.

Also, in JDK 1.1, local applications run outside the sandbox and there is no standard way to run them in the sandbox.

In JDK 1.2 (aka Java 2), all code is subject to the security policy.

This determines the privileges allowed to a class as it is loaded by the classloader.

There is a common security API and framework for applets and applications.

There is fine-grained access control with a well-defined access control mechanism.

The security manager is now concrete (rather than abstract) with an option to treat local applications and applets consistently, by using the `java -D` command line parameter.

In JDK 1.2, each class belongs to a single Protection Domain that is determined by the source of the code and the permissions given.

There is one security policy per JVM — this can be changed if you have the appropriate permission.

The permissions granted in a Protection Domain are used to determine access to system resources. Most permissions have a target, e.g., a filename, and an action, e.g., read.

When multiple domains are in the call stack, all must have the permission for access to be granted.

However, system code can have a privileged domain that stops the checking going further up the call chain.

Code Signing

Unfortunately, there are four different, incompatible signing mechanisms.

Sun's JDK 1.1 uses `javakey` to manage keys and certificates, and sign Java archive (`jar`) files.

Sun's Java 2 uses `keytool` to manage keys and certificates, and `jarsigner` to sign `jar` files.

Netscape uses its Object Signing technology with `jar` files.

Microsoft uses its Authenticode signing of Cabinet (`cab`) files.

The resulting situation is a complete mess!

Security Holes

One area of insecurity which remains is “denial of service attacks”.

There are no mechanisms in Java to prevent a malicious piece of Java code from using up all available resources, e.g., memory, process slots, etc.

Although this may be inconvenient, usually it cannot do any serious damage.

As far as I know, the Java security concepts are theoretically sound. Unfortunately, implementation “bugs” have resulted in insecurities.

Type Confusion

A type confusion attack is where one object type is passed off as another type.

In Java, an object instance is represented by a block of memory with the data fields laid out one after another.

An object reference is represented by a pointer to the memory address storing the object.

If one can create two pointers to the same object with different tags then the typing rules of Java can be bypassed completely.

```
class TowerOfLondon {
   private Jewels theCrownJewels;
}

class OpenHouse {
   public Jewels myJewels;
}
```

If the types `TowerOfLondon` and `OpenHouse` could be confused, then `theCrownJewels` could be accessed where they shouldn't be visible.

Perhaps worse is that private methods could be called where they shouldn't be callable.

So, type confusion completely destroys any hope of security.

One such type confusion attack used the fact that, for any type `T`, Java allows the type `array of T`.

In the JVM these array types have names beginning with `[` (which is not allowed as the first character of an ordinary class name).

One beta release of Netscape allowed a class to declare its own type name to be an array type name so that the JVM installed this name in its internal table.

This, in turn, allowed full system penetration.

(The fix is obvious.)

More Details

Permissions are subclassed from the (abstract) class `java.security.Permission`.

Already defined permissions include:

```
java.io.FilePermission
java.net.SocketPermission
java.util.PropertyPermission
java.lang.RuntimePermission
java.awt.AWTPermission
```

For example:

```
perm = new java.io.FilePermission("/tmp/abc", "read");
```

Each (concrete) permissions class must define the method `implies`.

This defines when one permission implies another.

For example, `java.io.FilePermission("/tmp/*", "read")` implies `java.io.FilePermission("/tmp/joe.txt", "read")`.

Some permissions have hidden implications, e.g., granting permission to write to the entire file system implies that the code could replace the JVM, and hence do anything!

Every piece of code is given an identity using the `java.security.CodeSource` class.

The identity is defined in terms of the code's source and signature (if any).

The security policy is represented by an object instantiated from `java.security.Policy`.

The policy is defined by mappings from identities to sets of permissions.

By default, the policy is specified within one or more policy configuration files which are read at JVM startup.

Examples:

```
grant {permission java.io.FilePermission "/tmp/*", "read";};

grant signedBy "Fred" { permission Foo.Bar; };

grant codeBase "http://java.sun.com/*" {
    permission java.io.FilePermission "/tmp/*", "read";
};

grant codeBase "http://java.sun.com/*", signedBy "Li" {
    permission java.io.FilePermission "/tmp/*", "write";
    permission java.io.SocketPermission "*", "connect";
};
```

A security policy may be specified when running an application, by using the `java -D` flag.

For example:

```
java -Djava.security.manager \
     -Djava.security.policy=/dcs/fwl/java.policy application
```

The first argument ensures that the default security manager is installed.

Access control is implemented by the `checkPermission` method in the class `java.security.AccessController`.

This takes a `Permission` object as parameter, and returns silently if permission is granted, or otherwise throws an `AccessControlException`.

For example:

```
AccessController.checkPermission(perm);
```

The `doPrivileged` method can be used to inform the AccessController that its body of code is “privileged”.

That is, the code is solely responsible for requesting access to its available resources, no matter what other code caused it to do so.

The idea is to encapsulate potentially dangerous operations that require extra privilege into the smallest possible self-contained code block.

The normal use of `doPrivileged` is:

```
somemethod() {
   // normal code here
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // privileged code goes here
         return null;
      }
   });
   // normal code here
}
```

Cryptography

Cryptography can be used not just to ensure confidentiality, but also for: identification, authentication, and data integrity.

Java provides facilities for cryptography via the "Java Cryptography Architecture" (JCA) and the "Java Cryptography Extension" (JCE).

Together, the JCA and JCE provide a complete, platform-independent cryptography API.

The JCE is separate because of US export control regulations. However, the API is available globally, and there are a number of implementations of the JCE available outside of the US.

Authentication and Authorization

Java 2 provides facilities for enforcing access control depending on **where** the code comes from. However, it still lacks these facilities based on **who** is running the code.

The proposed Java Authentication and Authorization Service (JAAS) augments the JDK with such facilities.

See: `http://java.sun.com/security/jaas/`

Summary

Java already provides security features which are much better than for other languages and systems.

There are still some security concerns with Java.

Some of the facilities are rather rudimentary and not easy to use.

Code signing is a hopeless mess.

Software bugs very often render the security useless.

Digital signatures, authentication and authorisation must be the way to go.