CS 5523 Lecture 21:
Java Language Security

- *Discussion of laboratory 3*
- *Discussion of laboratory 4*
- *Java security strategies*
- *Evolution of security models in Java*
- *Class loaders*
- *Permissions and policies*
- *Security managers and access controllers*
- *Signed and guarded objects*

Java security strategies:

- *Safety built into the language:*
  - *Bounds checking*
  - *Type conversion*
  - *No pointer arithmetic*
- *Code is verified after byte-code translation*
- *Resource access is controlled by policy*
- *Code can be signed so that users can determine the source and whether or not it has been modified.*
- *The runtime system actively enforces security policies*

Java 1.0 security model:

All remote code runs in a sandbox which has very limited access to resources. Local code has full access to resources.

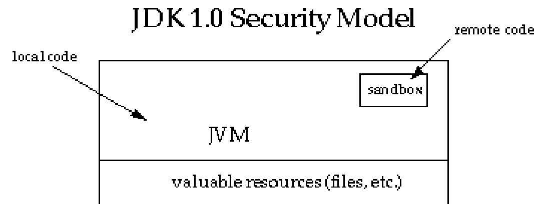


Diagram from Java™ 2 Platform Security Archicture

Java 1.1 security model:

Signed applets (delivered as signed jar files) are recognized as trusted code and can run on the full virtual machine with local code. Other remote code must run in the sandbox.

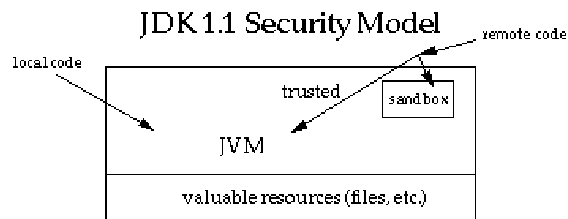


Diagram from Java™ 2 Platform Security Archicture

Java 2 security model:

- *Fine-grained access control*
- *Configurable security policy*
- *Extensible access control*
- *Extends checks to all programs*

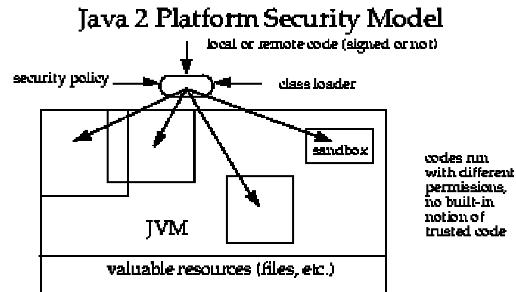


Diagram from Java™ 2 Platform Security Archicture

Java class loaders:

- *Convert source code into byte code for a target virtual machine*
- *Resolve classes (by loading all classes that a given class depends on)*

Three types of class loaders:

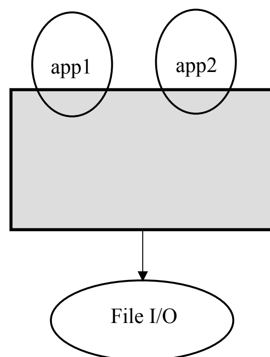
- *Bootstrap - for the system classes (rt.jar), written in C*
- *Extension – for standard extensions (jre/lib/ext), written in Java*
- *Application or system – for classes in CLASSPATH*

- *Class loaders always ask their parent to load and only load if parent cannot, (thus, system classes are loaded first).*

Java protection domains:

- *Enclose a set of classes (the principals) whose instances are granted the same set of permisssions.*
- *Two types of domains --- system and application*
- *External resources (e.g. file and networking) are only accessible through a system domain.*
- *The* `ProtectionDomain` *class holds a list of principals and the permissions.*

Domain transitions:



- *Applications should not acquire additional permissions when entering a system domain to perform operation*
- *When a system domain invokes an application method, it should not do so with additional permissions.*
- *Resource handling code invokes an* `AccessController` *to evaluate requests for resources*

Permissions:

- *A "permission" is a property checked by a security manager.*
- *The* `Permission` *class represents access to system resources.*
- *Example:*

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

represents a permission to read `/tmp/myfile`.

- *The default permissions are contained in a* `client.policy` *file.*
- *Collections of permissions are represented by the abstract class* `java.security.PermssionCollection` *for a homogeneous collections and the final class* `java.security.Permissions` *for heterogeneous collections*

`Permission` is an abstract class with subclasses:

- `AllPermission`
- `BasicPermission`
- `FilePermission`
- `PrivateCredentialPermission`
- `Service Permission`
- `SocketPermission`
- `UnresolvedPermission`

`BasicPermission` is an abstract class with subclasses

- AudioPermission
- AuthPermission
- AWTPermission
- DelegationPermission
- LoggingPermission
- NetPermission
- PropertyPermission
- ReflectedPermission
- RuntimePermission
- SecurityPermission
- SerializablePermission
- SQLPermission

Examples of types of permissions:

- java.io.FilePermission: read, write, execute
- java.net.SocketPermission: accept, connect, listen, resolve
- java.lang.RuntimePermission: createClassLoader, setSecurityManager, setFactory, modifyThread, ...
- java.awt.AWTPermission: accessClipboard, accessEventQueue, listenToAllAWTEvents, ...
- java.net.NetPermission: requestPasswordAuthentication, setDefaultAuthenticator, specifyStreamHandler

Permission checks:

Permission properties are checked by a security manager. All of the permissions are checked by one of the following calls in the security manager:

```
void checkPermission (Permission p)

void checkPermission(Permission p, Object context)
```

Security managers:

- *A class that controls whether specific operations are allowed*
- *The default for running applications is no security manager*
- *The default for the `appletviewer` is the `AppletSecurity` manager*
- *You install a security manager by calling `System.setSecurityManager`*
- *Permissions are controlled with policy files*

Security managers can check allowed operations:

Check an operation by calling the security manager:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
     security.checkXXX(argument,  . . . );
}
```

For example,

```
     checkAccept(host, port)
```

throws `SecurityException` *if the thread is not permitted to accept a connection on port from host.*

Security manager context:

As of Java 2 SDK v1.2, the default implementation of the check methods in `SecurityManager` *is to call the* `SecurityManager checkPermission` *method to determine if the calling thread has permission to perform the requested operation.*

Note that the `checkPermission` *method with just a single permission argument always performs security checks within the context of the currently executing thread.*

Policy objects:

- *Represent permissions available for code from various sources*
- *Code can always read files from its* `CodeSource`
- *The* `CodeSource` *includes a URL and certificates containing public keys to verify signed code originating from location*
- *Use* `getPolicy` *to retrieve current* `Policy` *object*
- `Policy` *objects can be specified in a policy configuration file.*
- *The policy configuration file may specify a keystore to be used to verify signers specified in the grant entries of the file.*

The `AccessController` class:

- *Decide whether or not access to resource is allowed based on policy*
- *Mark code as privileged*
- *Make a snapshot of current context to be used in other contexts*
- *Should be used in code that controls access to system resources if it is going to use specific security model*

The `GuardedObject` class:

- *Used when consumer of object different from producer and cannot get a context*
- *A* `GuardedObject` *encapsulates the resource and checks the security.*
- *Its encapsulated resource is retrieved by the* `getObject` *method that automatically checks the guard before returning the object.*

The `SignedObject` class:

- *Contains the* `Serializable` *object to be signed and its signature*
- *The signed object is a deep copy of an original object.*

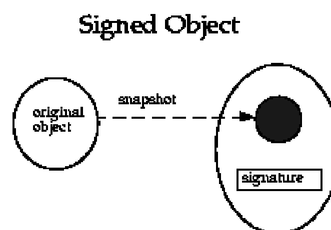


Diagram from Java™ 2 Platform Security Archicture

Signing:

```
Signature signingEngine = Signature.getInstance(algorithm,provider);
SignedObject so = new SignedObject(myobject, signingKey, signingEngine);
```

Verification:

```
String algorithm = so.getAlgorithm();
Signature verificationEngine = Signature.getInstance(algorithm, provider);
so.verify(verificationEngine);
```

Java byte code verification:

- *Checks that variable are initialized before use*
- *Verifies that method calls match types*
- *Verifies that the rules for accessing private data and methods are followed*
- *Checks that local variable accesses are in the runtime stack*
- *Checks that the runtime stack does not overflow*

These checks are also performed at compile time, but class files could be modified!

For next time:

- *Read CDK 8.1-8.2*