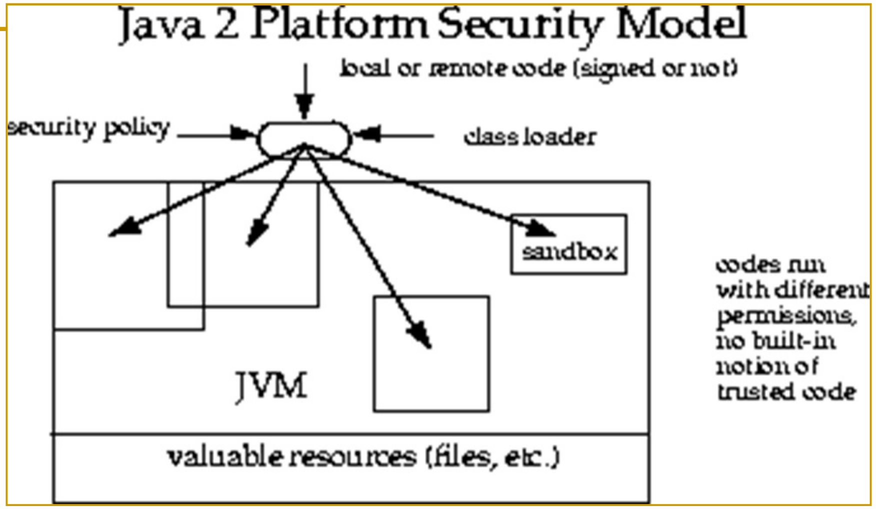
Java Virtual Machine Security

Java 2 Security Model



- Java Virtual Machine (JVM)
  - Java programs are implemented by a set of Java classes
    - From different sources
    - Not necessarily trusted
  - Secure sandbox for executing Java programs
- Security capabilities
  - Easily configurable security policy
  - Easily extensible access control structure
  - Extension of security checks to all Java programs

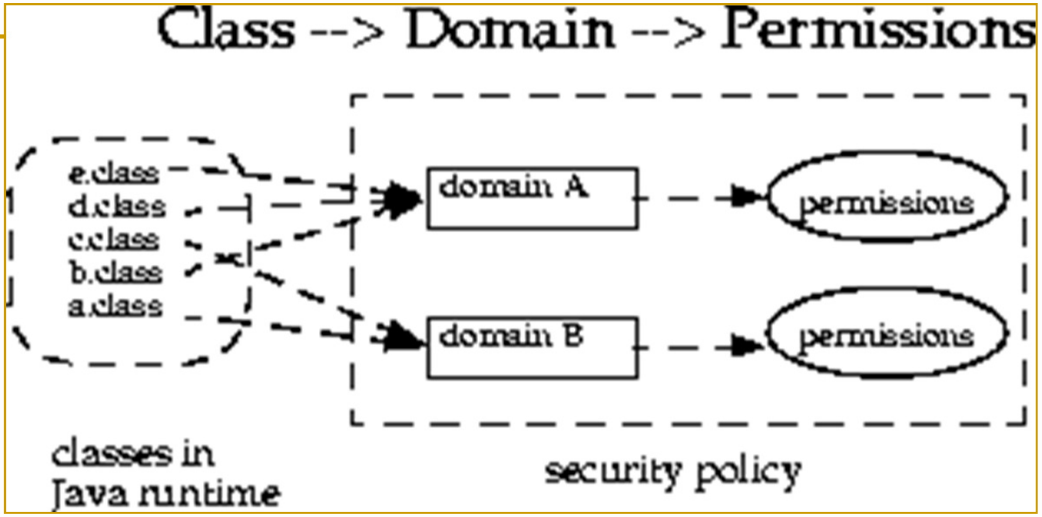
JVM sandbox model

- Creates a barrier around a Java execution environment
  - Applications are executed within a sandbox bounds
  - Cannot affect resources outside the sandbox
    - i.e. can only access resources available to the sandbox
- Basic rules of sandbox
  - Remote resource protection
    - Enforced by remote system
  - Local resource protection
    - Enforced by local security manager
  - JVM code and data protection
    - Enforced by static and dynamic check

Java Run-time Environment (JRE): Security-related features

- Loads required classes
  - Usually upon a class method invocation
- Verifies the correctness of loaded classes
  - Checks consistency and integrity
- Compiles bytecodes
  - Only for invoked methods
    - Just-in-time
  - Keeps original bytecodes for enforcing run-time validations
- Correct memory management
  - Automatic memory allocation and garbage collection
- Checks the correct execution of classes' code
  - Run-time integrity validations
    - Null pointer (ref)
    - Type checking
    - Dynamic (down)casting
    - Array bounds, etc.
  - Run-time security validations
    - Access control
      - Public, Package, Protected and Private access levels
      - Other permissions for Protection Domains
- Confinement
  - Isolation of Protection Domains

Protection domain



- A set of classes whose instances are granted the same set of permissions
  - Determined by the policy currently in effect

- JRE maintains a mapping from code (classes and instances) to their protection domains

- Instantiation of Protection Domains
  - ProtectionDomain (CodeSource, PermissionCollection);
  - ProtectionDomain (CodeSource, PermissionCollection, ClassLoader, Principal[]);
  - CodeSource (URL, Certificate[]);

Permissions

- Definitions of what is allowed or denied
  - Subclasses of interface java.security.Permission
- Examples
  - BasicPermission
    - Hierarchical name and arbitrary (or boolean) action
    - RuntimePermission, AWTPermission, ManagementPermission, NetPermission, PropertyPermission, etc.
  - FilePermission
    - Pathname & action (read, write, execute, delete)
  - SocketPermission
    - Host + port + action (accept, connect, listen, resolve)

Security policies

- Each JRE maintains an installed security policy
  - It determines the set of granted/denied authorizations
  - Subclass of java.security.policy
- Installed policy
  - There is always a policy installed (Policy Policy.getPolicy())
    - JRE includes a default policy reference implementation
      - Policy specified within one or more configuration files
      - [java_home]/lib/security/default.policy
    - Can be referenced by caller with getPolicy permission
  - Can be overwritten (void Policy.setPolicy(Policy))
    - Requires a setPolicy permission
    - The source location for the policy information utilized by the Policy object is up to the Policy implementation

Security manager

- At most one per JVM
  - Implement a security policy for an application
    - What is allowed and denied
  - It helps to check whether an action is allowed before requesting it
    - In the context of the calling thread
- Class SecurityManager of java.lang
  - Default run-time security manager
  - Can be redefined
    - but requires runtime permission setSecurityManager
    - This prevents malicious classes to overrun an installed security manager
  - Many checkXXX methods
    - For checking authorization for specific actions

      void checkRead(String file)
    - Uses the AccessController class and the method checkPermissions

AccessController

- An abstract class used for:
  - Decide whether an access to a critical system resource is to be allowed or denied
    - According to the security policy currently in effect

```
FilePermission p = new FilePermission( "/temp/testFile", "read" );
AccessController.checkPermission( p );
```

  - Mark code as being *privileged*
    - Affecting subsequent access determinations
  - Obtaining a *snapshot* of the current calling context so access-control decisions from a different context can be made with respect to the saved context

Dynamic class loading: Class loaders

- Primordial class loader
  - Critical part of VM
    - Trusted VM component, defined in JVM specification
  - Prevents name spoofing of java.* library classes

- Additional class loaders
  - Defined by users/applications
    - They can help application to locate and download classes' contents
    - But the bytecodes of classes are installed by the VM class loader
  - Each one defines separate namespace environment
    - Each class is tagged with class loader that loaded it
    - Classes in one namespace cannot interact with classes in other namespaces
  - Allows different versions of same class name to co-exist
    - Typically associated with code from different origins

Dynamic class loading: Overview (1/2)

- Class loading security policies
  - No class loading of packages java.* other than from the canonical local repository
    - To avoid the replacement of the basic Java classes
    - Primordial class loader ensures this
  - Classes from different network servers do not interact
    - Different domains
    - No interference between "programs" of different sources

Dynamic class loading: Overview (2/2)

- Class loading steps
  - Locate the requested binary class
    - .class file
  - Parse/translate into internal data structures for emulation
  - Enforce the naming conventions
    - Domain, package, classes, fields/methods
    - Accessibility levels: public, private, package
  - Check correctness of binary class
    - File integrity check
    - Class integrity check
    - Bytecode integrity check
    - Runtime integrity check
  - Perform any translation of code and metadata
    - Make the method ready to be run
  - Initialize memory and pass control to emulation engine

Dynamic class loading: Class loader checks

- File integrity check
  - Magic number, proper formats used
  - Component declared and actual sizes
- Class integrity check
  - Has superclass and is not final
  - No override of final superclass method
  - Methods and fields have legal names and signatures
- Bytecode integrity check
  - Data-flow analysis
  - Stack checking
  - Static type checking for method arguments and bytecode operands
- Runtime integrity checks
  - Verifications on method calls

Dynamic class loading: Class loader checks

- Consistency checks
  - Check binary class format
    - Magic number, proper formats used
    - Component declared and actual sizes
  - Check method calls
    - Number and type of arguments match between caller and callee
  - Check and resolve fully qualified references
    - On demand
    - Replaced with a direct reference for performance
- Integrity checks
  - Verify the integrity of bytecode program
  - Static type checking
    - Including up-casting
  - Check branch instructions within boundaries