Demystifying Java Platform Security Architecture

Ramesh Nagappan, CISSP
nramesh@post.harvard.edu

core SECURITY PATTERNS

Overall Presentation Goal

Learn how to get started on using **Java Platform and its core Security Mechanisms** .

- J2SE, J2ME, Java Card, Applets, Java Web start
- Java Security Management tools
- Securing Java Code from Decompilation

Presentation Outline

- Java Platform Security Architecture
- Java Applet Security
- Java Web Start Security (JNLP Security)
- Java Micro Edition (J2ME) Security Architecture
- Java Card Security Architecture
- Java Platform Security – Key and Certificate Management tools
- Securing the Java code from Decompilation

Java Platform Security Architecture

Basics about Java

- Java Platform and its Programming language is introduced by Sun Microsystems during late 1995.
  - Invented by Dr.James Gosling at Sun Microsystems.
- Java offers a object-oriented programming and platform-independent application development environment.
  - Java delivers a architecture neutral, interpreted and executable bytecode.
  - Java enables delivering portable cross-platform application solutions.
  - Java applications can be accessed locally or dynamically loaded from a network.
  - Java applications are capable of running on any device platform
    - From smartcards to micro-devices, workstations to enterprise servers, mainframes to supercomputers and so on.

Core Java Technologies

- Sun Categorized the Java technologies under three key major editions.
  - Primarily to simplify software development and to support target deployment platform.
  - **Java Standard Edition** (also referred to a J2SE or JDK)
    - Commonly used as Java runtime environment (JRE) for running basic Java applications
  - **Java Enterprise Edition** (also referred to as J2EE or Java EE)
    - Set of standards and API technologies for developing multi-tier business applications.
  - **Java Micro Edition** (also referred to as J2ME or Java ME)
    - Set of standards and API technologies for Java enabled Micro-devices and embedded systems
    - **Java Card** is a sub-set of J2ME for supporting smartcards.

Java Security at the High-level

- Security has been an integral part of Java technology from day one.
  - It has been an evolving design goal of the Java community as Java is primarily targeted to network-centric applications.
- Java Security Architecture foundation provides a secure execution environment via:
  - The **Java Virtual Machine** (JVM)
    - The JVM defines a secure environment by enforcing stringent security measures for running Java applications
  - The **Java Programming Language**
    - The Java language provides several inherent features that ensures security and integrity of the Java application and its underlying JVM.

Security in Java Virtual Machine

- The Java Virtual Machine (JVM) is an abstract computing engine
  - It resides on the host computer and serves as the execution environment for executing the compiled Java code.
  - The JVM insulates the Java application from underlying differences of the operating systems, networks and system hardware.
- The JVM's built-in security architecture protects the Java environment from most security breaches.
  - The JVM security architecture acts as a **primary security layer** by protecting users and the environment from malicious acts.
  - The JVM enforces security via configurable policies, access control mechanisms and security extensions.
  - The JVM also allows users to securely download and execute untrusted Java programs from remote resources and over the network.

Security in Java Language

- The Java language is a general-purpose object-oriented programming language
  - It delivers platform-neutral compiled code that can be executed by a JVM.
- The Java language is designed to provide security of the application and its underlying runtime environment.
  - The Java language assures security of the application at all levels
    - From the basic Java language constructs to the Java runtime
    - From the supporting Java class libraries to the Java application
  - The Java language also offers several inherent features that contributes to the security of the application.

Security features in Java Language

- The key security features of the Java language that contributes to the security are:
  - Java defines all primitives with a specified size and all operations are defined to be executed in a specific order of execution.
  - Java language provides access control functionality on variables and methods in the object via namespace management for type and procedure.
    - Ex. public, private, protected, package etc.
  - Java language does not allow defining or de-referencing pointers.
    - Programmers cannot misuse or forge a pointer to the memory or create code defining offset points to memory.
  - Java object encapsulation supports "***programming by contract***" that allows reuse of the code that has already been tested.
  - Java is a strongly typed language – During compile time it does extensive type checking for type mismatches.

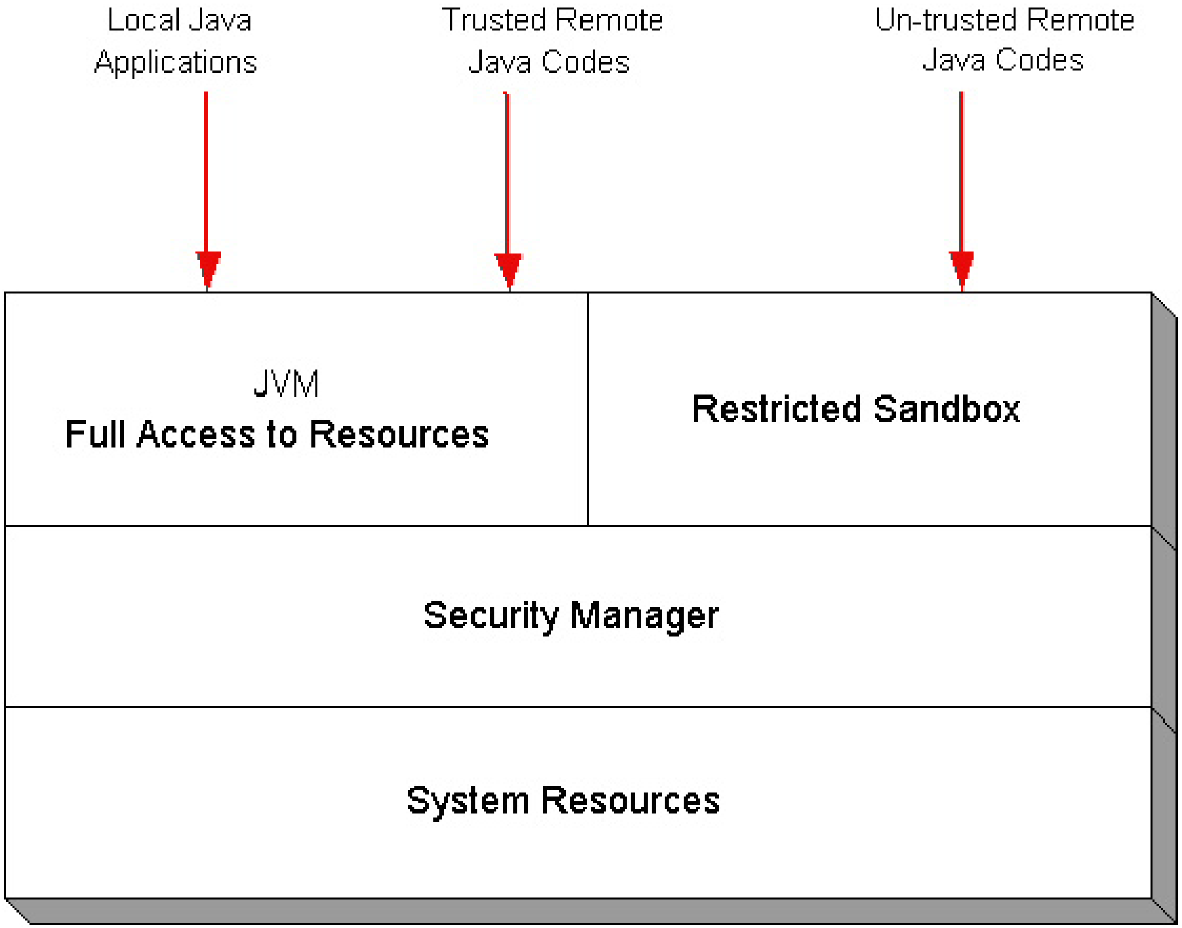
Security features in Java Language

- Continued:
  - Java allows declaring classes or methods as final.
    - Helps to protect the code from malicious attacks via creating sub-classes and substituting it for the original class and override methods.
  - Java Garbage Collection mechanism contributes to security of Java programs
    - By providing a transparent storage allocation and recovering unused memory without manual intervention.
    - Ensures program integrity during execution and prevents programmatic access to accidental and incorrect freeing of memory resulting a JVM crash.

Java Platform Built-in Security

- Java provides a built-in security architectural foundation as part of the JVM.
  - Based on configurable policies and domains
  - Rule based class loading
  - Signing code via support for cryptography services.
  - Allows implementing security policies
    - For protecting/controlling access to resources
- Since inception : Java 1.0.x
  - Java introduced the notion of a ***Sandbox based security model***
  - Java 1.0.x sandbox security model helps running all Java applications locally within the resources available to the JVM.
    - Protects downloaded Java applets cannot access or alter the user’s resources beyond the sandbox
  - Java 1.1.x introduced ‘***signed applets***’, which allowed downloading and executing applets as trusted code after verifying applet signer’s information.

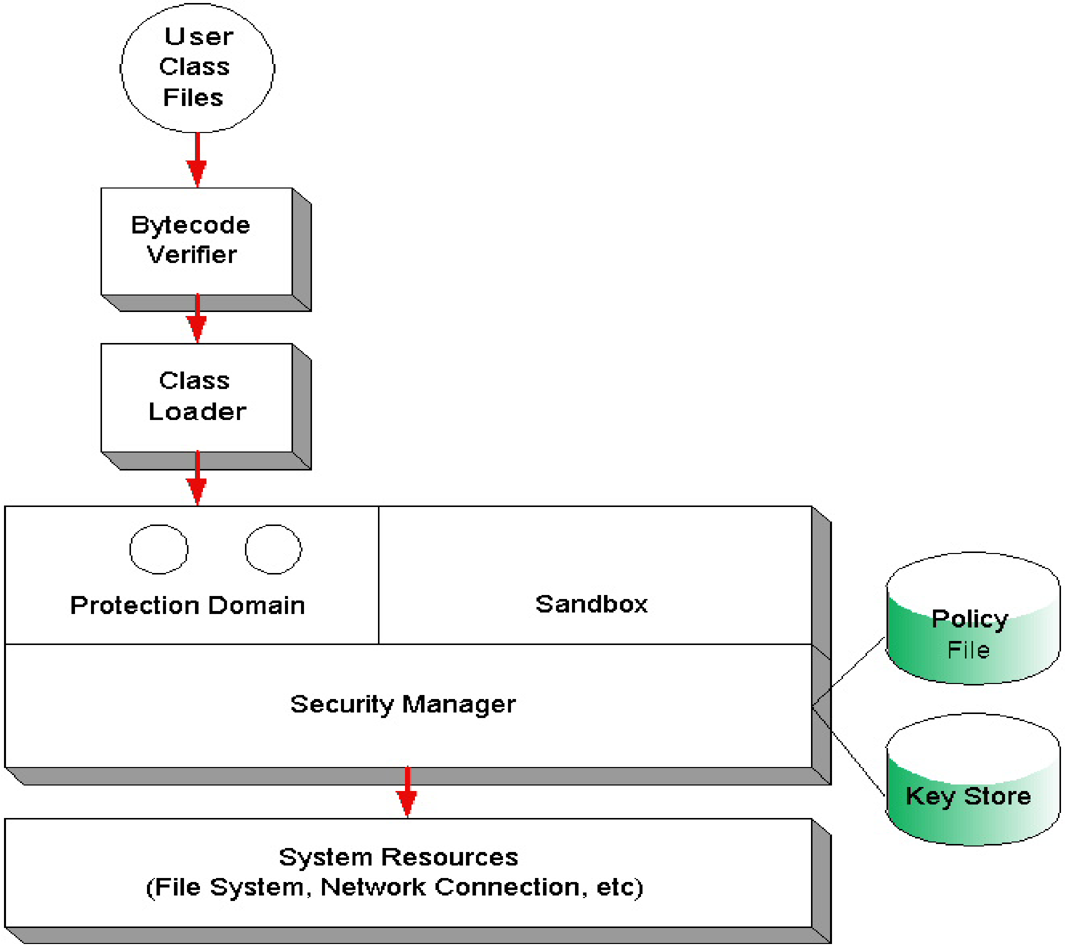
JDK 1.1 Security Model



Java 2 Platform Security Model

- Java 2 Platform (J2SE) introduced significant security enhancements to JDK 1.1.x.
  - Full-fledged support for cryptographic services
  - Tools for PKI management and digital certificates
  - Policy-driven restricted access control to JVM resources
  - Rule-based class loading and verification of bytecode
  - Policy driven access to Java applets downloaded by a Web browser
- In J2SE Security architecture, all code can be subjected to a ‘Security Policy’ – regardless of running locally or downloaded remotely
  - All code can be configured to make use of a ‘***Protection domain***’ (equivalent to a sandbox) and a ***Security policy***.
  - The Security policy dictates whether the code can be run on a particular protection domain or not.

J2SE Security Architecture Elements



J2SE Protection Domain

- J2SE introduced the notion of “Protection Domains” , that allows to enforce access control policies.
  - Configuring a ‘Protection Domain’ allows grouping of classes and instances by associating them with a “Security policy” containing set of “Permissions”.
  - Protection domains are determined by the current security policy defined for a Java runtime environment.
  - The ***java.security.ProtectionDomain*** class encapsulates the characteristics of a Protection Domain.
  - With out defining a Protection Domain - by default all “local” Java applications run unrestricted as trusted applications
- Protection Domains are generally categorized as two domains.
  - ***System Domain***: All protected resources such as file systems, networks
  - ***Application Domain***: The protected resources that are part of the single execution thread.

Permissions

- ‘Permissions’ determine whether access to a resource of the JVM is granted or denied.
  - ***Permissions*** give specified resources or classes running in the instances of the JVM – the ability to permit or deny certain runtime operations.
    - For example: An applet or application using a Security Manager can obtain access to a system resource only if it has permissions.
  - The Java Security API defines a hierarchy of Permission classes.
  - The ***java.security.Permission*** is the abstract class that represents access to a target resource.
  - The Permission class contain several sub-classes to represent different type of permissions.
- Example ‘Permission’ Classes
  - For wildcard permissions: *java.security.AllPermission*
  - For network permissions: *java.net.SocketPermission*
  - For file system permissions: *java.io.FilePermission*

Setting Permissions in a policy

- ‘Permissions’ can be defined using a security policy configuration file.
  - If a caller application requires access to a file located in a file system. The caller application must have the permissions granted to access the file object.
  - For example, to grant access to read a file in “c:\temp” the file permission can be defined in security policy file.

```
grant {
        permission  java.io.FilePermission
                                      “c:\\temp\\testFile”, “read” ;
        }
```

Policy

- In J2SE, Security policy defines the protection domains for Java applications.
  - The JVM makes use of a policy driven access control mechanism by dynamically mapping permissions defined in one of more “policy” files.
  - Java applications are configured with policy files describing access privileges such as read and write or making a connection to a host.
  - The user or administrator of the application usually configures the policy file.
    - In J2SE, the system-wide security policy file ‘***java.policy***’ is located at *<JRE-HOME>/lib/security/* directory.
    - The policy file location for the system is defined in the security properties file with a java.security located at *<JRE-HOME>/lib/security/* .
- The effective policy of the Java application environment will be the union of all permissions defined in all policy files.

Example Policy file

- Policy configuration file specifying the permission for signed JAR file loaded from http://www.coresecuritypatterns.com and signed by “javaguy”
  - The signer “javaguy” is granted with read/write access to all files in /export/home test.

```
grant  signedBy “javaguy”
            codebase “http://www.coresecuritypatterns.com/*”   {
        permission  java.io.FilePermission
                     “/export/home/test/*”, “read,write” ;
      };
```

Java Security Manager

- In J2SE, Security Manager acts as the primary security guard against malicious operations.
  - Security Manager (***java.security.SecurityManager***) plays the role of enforcing the required security policy.
  - The *java.security.SecurityManager* consists of several *checkXXYYZZ()* methods to determine access privileges.
  - If there is a security violation, the JVM will throw an ***AccessControlException*** or ***SecurityException***.
  - To enforce a Java application to use a SecurityManager and security policy – startup the JVM with ***–Djava.security.SecurityManager*** and ***–Djava.security.policy*** as JVM arguments.
    - For example: *java –Djava.security.SecurityManager -Djava.security.policy=/export/ramesh/My.Policy MyJavaClass*

Using SecurityManager Class

- In a Java application, the Security Manager is set by the *setSecurityManager()* or obtained via *getSecurityManager()* methods in class System.
  - For example, to use SecurityManager programmatically in a Java application code.

```
. . .
     java.security.SecurityManager mySecurityManager =
                                  System.getSecurityManager();
        if  (mySecurityManager != null)  {
           mySecurityManager.checkWrite(fileName) ;
          }
 . . .
```

Java AccessController

- Access Controller allows performing dynamic inspections and deciding whether access to a resource is granted or denied.
  - In Java code, AccessController (***java.security.AccessController***) allows to encapsulate the location, codesource and permissions to perform an operation.
  - It makes use of ***CheckPermission(Permission)*** method to determine access to the resource.
  - If there is a security violation, the JVM will throw an ***AccessControlException***.

Using AccessController Class

- For example, to use AccessController for checking read and write permissions of a directory.

```
. . .  .
try {
      AccessController.CheckPermission
                    (new FilePermission(“/var/temp/*”, “read,write”));
                                        System.getSecurityManager();
    }  catch (SecurityException secx) {
              //Print…Does not access to directory
             }
. . .
```

Java Codebase

- Java allows to specify URL location of a class or JAR file using codebase.
  - In Java Security Policy file, codebase identifies the URL location with permissions for granting or denying access.

```
. . .
   grant   codebase “http://www.coresecuritypatterns.com/*”   {
              permission  java.io.FilePermission
                             “/export/home/test/*”, “read,write” ;
            };
```

Java CodeSource

- CodeSource allows representation of a URL from which a class was loaded and the certificate keys used to sign the class.
  - The CodeSource class and its two arguments for defining code location and certificate keys are specified as:

```
. . .
     CodeSource myCS = (URL url, java.security.cert.Certificate certs[]);
    . . .
```

Java Bytecode Verifier

- The Java Bytecode verifier is an integral part of JVM.
  - Allows to verify code prior to execution
  - Ensures that the code was produced consistent with Java specifications by a trustworthy compiler.
  - Also allows to detect inconsistencies related to array bound-checking and object casting through runtime enforcement.

Java ClassLoader

- The Java ClassLoader is responsible for loading Java classes into the JVM.
  - From a security standpoint, Classloaders can be used to establish security policies before executing untrusted code or to verify digital signatures.
  - To enforce security, Classloader coordinates with the *SecurityManager* and *AccessController* to determine security policies.
  - In J2SE, all Java applications have the capability of loading bootstrap classes, system classes and application classes using an internal class loader (also referred to as ***primordial class loader***).
    - The Primordial classloader uses a special classloader ***java.security. SecureClassLoader*** to protect JVM from malicious classes.
    - The *SecureClassLoader* has a protected constructor that associates the loaded class to a protection domain.
    - For example: *URLClassLoader* is a sub-class of the *SecureClassLoader*.

Using URLClassLoader

- For example, to use URLClassLoader for loading classes from a directory.

```
. . .
try {
    //Convert file location to URL
     URL url = file.toURL();
        URL[] urls = new URL[]{url};
    // Create a new class loader
         ClassLoader myClassLoader = new URLClassLoader(urls);
         Class  myClass = myClassLoader.loadClass(“com.csp.MySecClass);
       }  catch (MalformedURLException secx) {
        } catch (ClassNotFoundException) {
           }
```

Java Applet Security

Java Applet Security

- Java Applets are client-side applications downloaded from the Web.
  - Runs in either a Java-enabled Web browser or a Java appletviewer.
- Downloaded Applets are considered “untrusted” and restricted from access to resources in client host.
  - Prevents applets - from reading or writing files, making network connections or native calls, starting other programs or loading libraries.
- Applets can be considered “Trusted” based on following factors
  - Applets installed on a local file system or executed on a localhost.
  - ***Signed applets*** allows verification of originating source and signer’s information.
  - Signed applets can be trusted to run with the permissions granted in a security policy file.

Signed Applets

- J2SE introduced the notion of signed applets
  - Ensures that the applet origin and its integrity is guaranteed by a Certificate Authority (CA).
  - Allows to trust them with permissions granted via a ***client security policy*** file.
- J2SE bundle provides tools for signing applets and applications.
  - ***'jarsigner'*** tool allows attaching a digital signature to the applet.
  - To sign the applet, it is required to obtain a certificate capable of code signing.
  - The digital signature identifies the signer of the applet.
- Applet JAR file is signed using the private key of the applet creator.
  - The signature is verified at the client using the public key of the applet creator.
  - For production purposes, it is important to acquire public/private key certificates from a trusted CA.

Applet Signing Process

1. Compile the Applet source to an executable class
2. Package the compiled class into a JAR file

   Ex.
   ```
   jar cvf WriteFileApplet.jar  WriteFileApplet.class
   ```
3. Generate Key Pairs using keytool (For testing) or Obtain Public and Private keys from a CA (For production purposes).

   Ex.
   ```
   keytool –genkey -keystore mystore  -keypass mypwd –storepass mystorepwd
   ```
4. Sign the JAR file using the *jarsigner* utility and verify the signature on the JAR file.

   Ex.
   ```
   jarsigner –keystore mystore –storepass mystorepwd –keypass mypwd
                 -signedjar  SignedWriteFileApplet.jar  WriteFileApplet signapplet
   ```
5. Export the public key certificate – It is required to sent to the end user keystore requiring access to the applet.

Applet Signing Process

Continued …

5. Deploy the JAR and Certificate files

   Ex.
   ```
   <applet code=WriteFileApplet.class
       archive=“SignedWriteFileApplet.jar”  codebase=“….”></applet>
   ```

6. Import the Public key and Trusted CA certificates into the client keystore (Using keytool utility).
7. Create the Policy file that grants the applet to have the required permissions.
8. Run and test the applet for all defined permissions.

Java Web Start Security

Java Web Start

- Java Web Start (JWS) is a full-fledged Java client application, considered as a viable alternative to Java applet.
  - JWS based client applications be deployed, launched and updated from a Web server.
  - The underlying technology of JWS is **Java Network Launch Protocol (JNLP).**
  - JWS provides a mechanism for application distribution through a Web server.
  - Enables Java rich-client access to server application over a network.
  - Once a JWS application is downloaded – it does not need to be downloaded next time – the updates to the application is automatically done in an incremental fashion.
  - Since J2SE 1.4.x, JWS has been an integral part of Java bundle.

Securing Java Web Start Applications

- JWS applications runs outside of the Web browser using the sandbox features of the Java runtime.
- JWS allows defining security attributes for client-side Java applications.
    - To specify access to local resources, such as file systems, network connections etc.
    - The security attributes are specified using XML file referred to as ***JNLP descriptor*** file.
- JNLP descriptor defines the application access privileges to the local and network resources.
    - JNLP also allows the use of digital signatures for signing JAR files.
    - When downloading unsigned applications, JNLP displays a “Security Advisory” dialog box prompting the end user about any required action.

JNLP Settings for JWS Security

- JNLP descriptor uses XML elements to describe JWS applications and its security.
- To enforce security, <security> element is used to specify permissions.
- JNLP allows to define two permission options:
  - ***<all-permissions>*** for applications requiring full-access to client resources.
  - ***<j2ee-application-client-permissions>*** for selected set of permissions – socket permissions, file access, clip-board access and so forth.

This application has requested read/write access to a file on the local filesystem. Allowing this action will only give the application access to the file(s) selected in the following file dialog box. Do you want to allow this action?

☐ Do not show this advisory again

JWS Security Advisory Dialog

Sample JNLP Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
 <jnlp spec="1.0+" codebase="file:///c:/rameshn/jnlp/" >
          <information>
                          <homepage href="/jdc" />
          </information>
                 <offline-allowed/>
                  <security>
                         <j2ee-application-client-permissions/>
                 </security>
             <resources> <j2se version="1.2+" />
                        <jar href="/MySignedJNLP.jar"/>
            </resources>
      <application-desc main-class=“MyJNLP" />
  </jnlp>
```

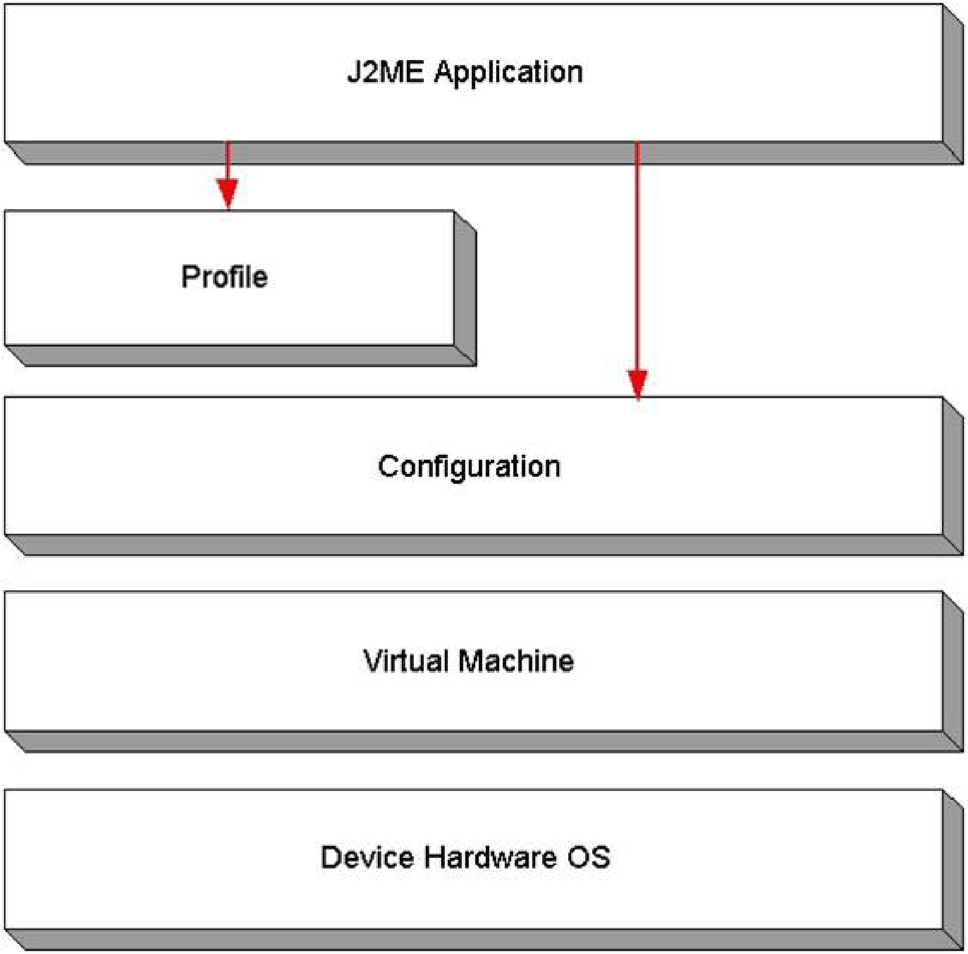
* *It is also important to sign the JAR file using JARSIGNER tool before JNLP deployment.*

J2ME Platform Security

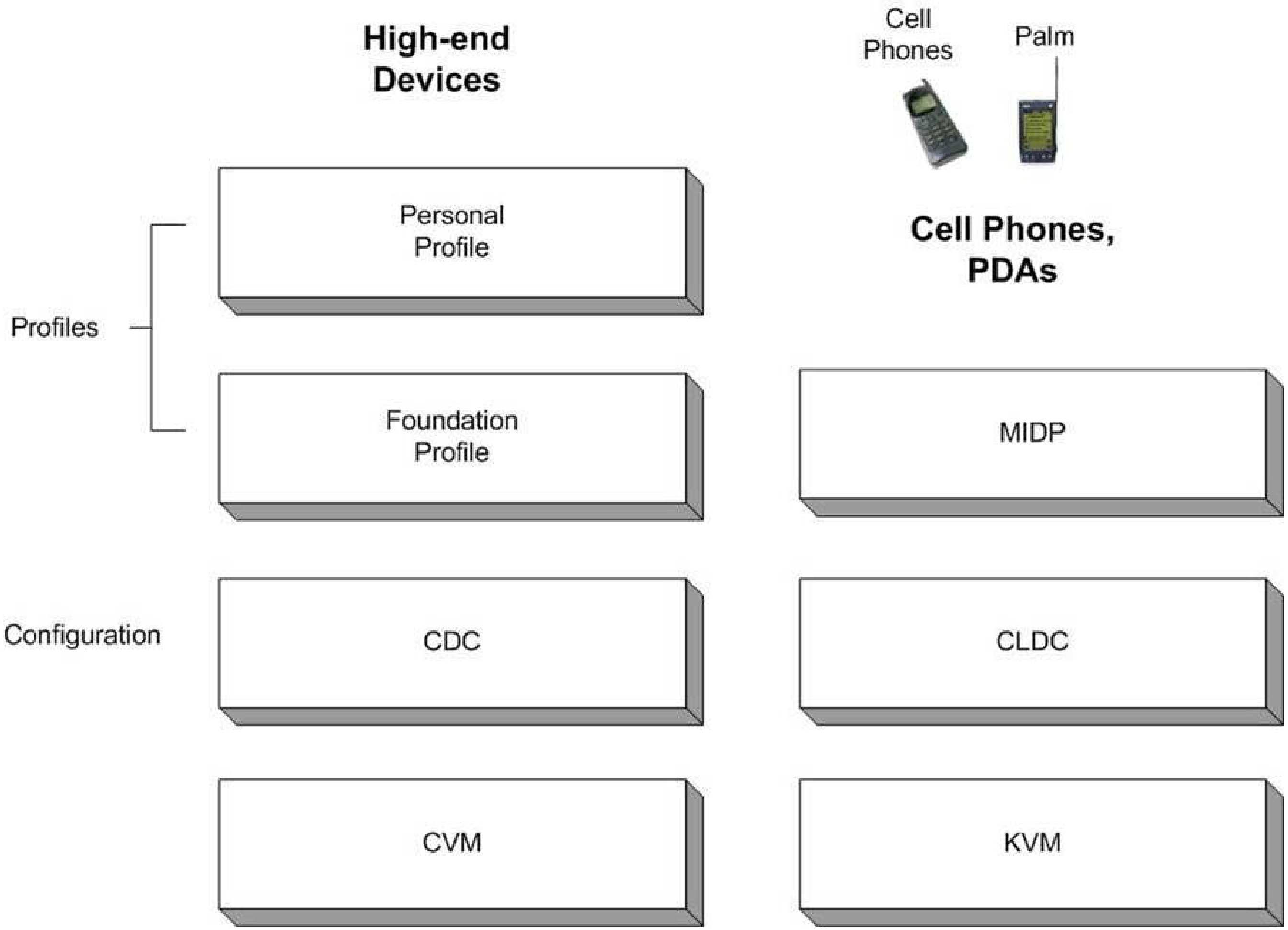
Java 2 Micro-edition (J2ME)

- J2ME is designed to deliver the benefits of Java technology for micro-devices and embedded systems.
  - Devices with limited constraints to Memory size, display size, processing power, network bandwidth and battery life.
  - Slimmed down version of J2SE.
  - Defines a notion of configurations and profiles to represent device characteristics as per industry specifications.
- J2ME defines configurations to satisfy the needs of broad range of devices.
  - ***Connected Device Configuration (CDC)*** targets high-end consumer devices with high-bandwidth network and atleast 2Mb Memory.
  - ***Connected Limited Device Configuration (CLDC)*** targets low-en devices with only 128-512 kb of memory.

J2ME Platform Architecture



J2ME Configuration and Profiles



High-end Devices

Cell Phones

Palm

Cell Phones, PDAs

Profiles

Personal Profile

Foundation Profile

MIDP

Configuration

CDC

CLDC

CVM

KVM

CDC Security

- CDC offers all security features typical to J2SE Security.
  - J2ME runtime built-on on CDC may utilize the standard JVM bundled with J2SE or Compact Virtual Machine (CVM).
- Similar to J2SE, CDC Security features include:
  - All code runs in a sandbox without exposing the user’s device to risk.
  - All classes loaded with full byte-code verification and Java language features.
  - Signed classes are verified for integrity and originating source.
  - Security policy provides fine-grained access control over the resources using set of permissions and policies.
  - Support for Java cryptography to secure programs, data, communication and data retrieval.

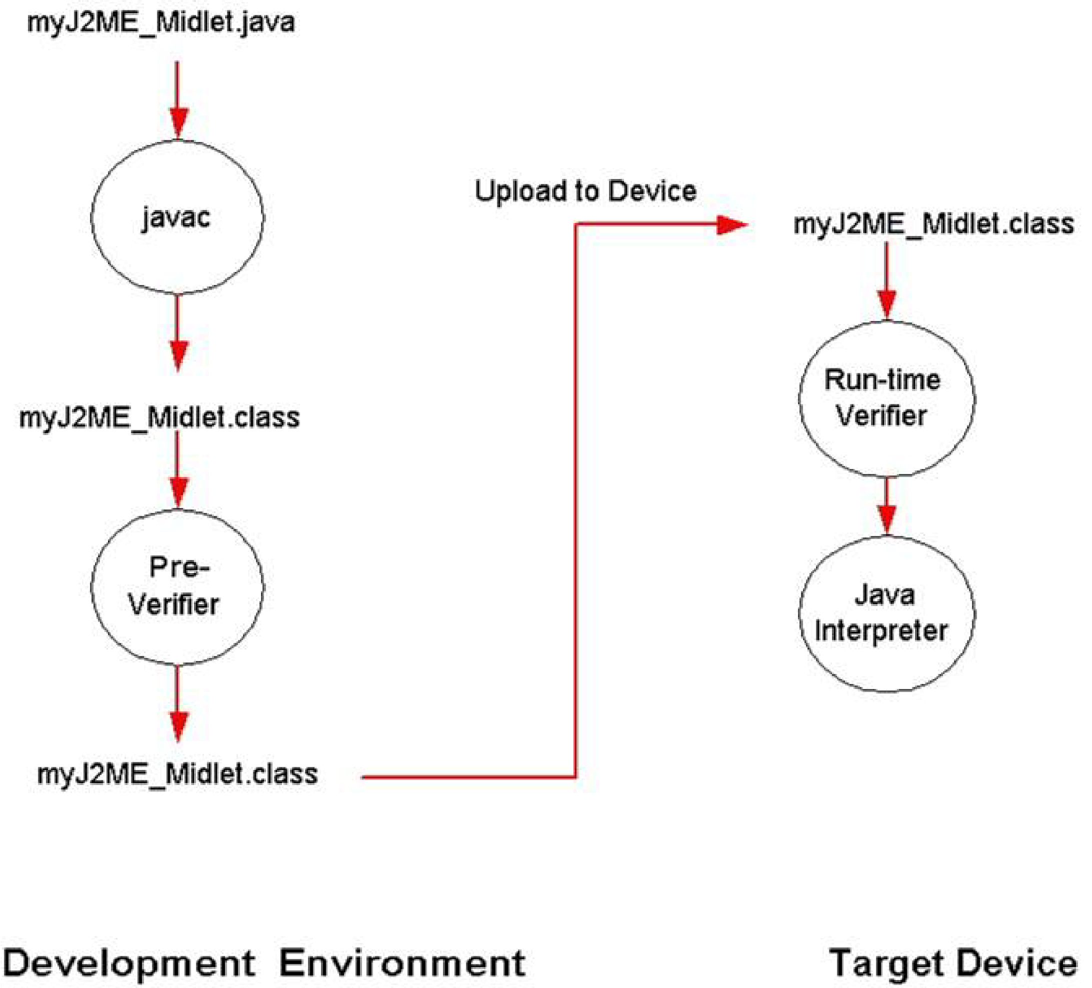
CLDC Security

- CLDC features a sub-set of JVM with limited API and supporting libraries.
  - CLDC runs on top of Sun’s ***K Virtual Machine (KVM)*** designed specifically for supporting resource limited devices.
- CLDC features a limited security model including:
  - New class verification mechanism.
  - No user-defined class loaders
  - No support for thread groups or daemon threads
  - No support for weak references
  - Limited error handling
  - No finalization
  - No reflection
  - New connection framework for networking

CLDC Security levels

- CLDC features two levels of security.
    - Low-level KVM Security
    - Application level Security
- Low-level KVM security
    - Application running in the KVM cannot disrupt the device anyway
    - Security is guaranteed by a “***Pre-verification process***” that rejects invalid classes.
    - After “Pre-verification” the KVM does an in-device verification process.
- Application-level Security
    - The KVM defines a ***sandbox*** that ensures all Java classes are verified and guaranteed to be valid.
    - Limits a pre-defined set of APIs for the application as required by the CLDC specification and the supporting device profile.
    - Application is restricted from using its own classloader.

KVM Pre-verification Process



J2ME Profiles

- J2ME Profiles define a set of Java API technologies suited to meet a specific device class or targeted class of devices..
  - Profiles are built on top of J2ME Configurations
  - ***Mobile Information Device Profile (MIDP)*** is built on top of CLDC
  - Foundation Profile (FP) is built on top of CDC
- MIDP
  - Combines with CLDC to provide an execution environment and application functionality
  - Includes user interface, application management, network connectivity etc.
  - MIDP applications are packaged similar to Java applet referred to as ***MIDlet***
- MIDlet
  - A J2ME application designed to run on a mobile device.
  - A MIDlet suite consists of one or more MIDlets packaged as JAR file including a ***Java Application Descriptor (JAD)*** file.

MIDlet Security

- MIDlet suites are restricted to operate within a sandbox based security model.
  - Helps to avoid any risks to the device resources.
  - MIDP 2.0 introduced the notion of ***Trusted MIDlets and Signed MIDlets***
    - Ensures consistent security mechanism defined by a domain policy.
    - MIDlet suites can be cryptographically signed and verified for integrity.
- Trusted MIDlets
  - Based on the J2SE concept of Protection Domains.
  - Each Protection Domain associates a MIDlet with set of permissions and interaction modes.
- Signed MIDlets
  - Similar to signed applets, MIDlets are signed and trusted via digital signature and PKI support.
  - Signer of the MIDlet is responsible for distributing and supporting the MIDlets.

Permissions and Interaction Mode

- A Trusted MIDlet contains "allowed" and "user" *permissions*.
  - The "allowed" permissions define a set of actions allowed without user interaction.
  - The "user" permissions define a set of permissions that require explicit user approval.
    - The user permissions are defined to grant allow or deny permissions to specific functions via three types of ***Interaction Modes.***
    - Interaction modes are determined by a security policy.
- Interaction modes
  - Blanket mode - The MIDlet is valid for every invocation until its permission is revoked by the user or deleted from the device.
  - Session mode – The MIDlet is valid for very invocation until the session terminates.
  - Oneshot mode: The MIDlet is valid for single invocation of a restricted method.

Example: MIDlet policy

```
domain: O="MIDlet Underwriters, Inc.", C=US
allow: javax.microedition.io.HttpConnection
oneshot(oneshot): javax.microedition.io.CommConnection
alias: client_connections
 javax.microedition.io.SocketConnection,
     javax.microedition.io.SecureConnection,
     javax.microedition.io.HttpConnection,
     javax.microedition.io.HttpsConnection
 domain: O=Acme Wireless, OU=Software Assurance
 allow: client_connections
 allow: javax.microedition.io.ServerSocketConnection,
            javax.microedition.io.UDPDatagramConnection
oneshot(oneshot): javax.microedition.io.CommConnection
domain: allnet
blanket(session): client_connections
oneshot: javax.microedition.io.CommConnection
```

Signed MIDlet

- Signing a MIDlet suite is just a process of applying the digital signature to the JAR file.
  - Process of adding signer’s public key certificates and digital signature to the JAR file.
  - Adds new attributes to the JAD file – as Base64 encoded values of the certificare.

    MIDlet-Certificate: <Base64 encoded value of certificate>

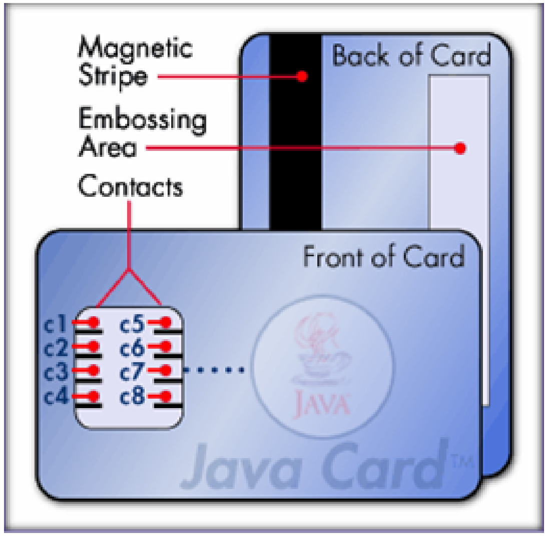
    MIDlet-Jar-RSA-SHA1: <Base64 encoded value of signatures>.
- The *J2ME wireless toolkit* enables a signer to either sign a MIDlet suite (JADTool)
  - The ***JADTool*** utility allows to use certificates and keystores from a J2SE keystore.

    Example: *java –jar JADTool.jar –addcert or -addjarsig*

Java Card Platform Security

Java Card Technology Basics

- Java Card technology enables smart cards and other devices with limited memory to run Java applications.
  - Designed based on the smartcard specifications – ISO 7816 defines the communication between application and a smartcard through "APDU – Application protocol Data Unit".
  - Helps adoption of Smartcard technology in Cellular phones, ATM/Credit cards, PDAs etc.
  - Brings Java advantages to smartcards offering application portability, platform-independence and secure execution environment.
  - Sun Microsystems provides a ***Java Card development toolkit for smartcard*** application development.



The JavaCard
(Source: Sun Microsystems)

Java Card Runtime Environment

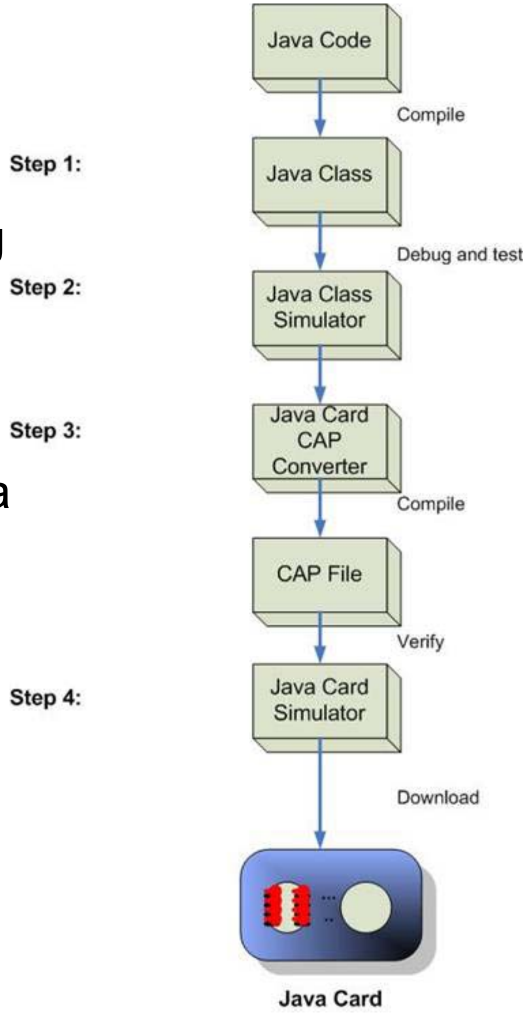
- The Java Card Technology defines a Java runtime environment for smartcards.
  - The ***Java Card runtime environment (JCRE)*** runs on top of smartcard hardware and its native smart card system.
  - Java Card applications are implemented as “***Java Card Applets***” built using Java Card APIs.
  - JCRE acts as an intermediary between the native smart card system and the Java card applet.
  - Using JCRE, the host application sends a command APDU abd Java Card applet responds with a response APDU.
    - The command APDU is transmitted to the JCRE which is sent to appropriate Java Card Applet for processing – which in turn send a response APDU to the JCRE.
- JCRE provides a secure environment and high-level API interface to support smartcard applications.

JCRE Security

- The JCRE provides a *secure execution environment* with a virtual firewall between smartcard applications on the card .
  - The Java Card API allows to develop secure smartcard applications – by enforcing security evaluation of the application by inheriting a subset of Java language features.
  - The JCRE store the objects and data in memory. During a power loss or failure the platform make objects/data are store to its previous state.
  - The JCRE also bring the notion of sandbox – implemented via ***applet firewall*** mechanism called a context.
    - Applets are forced to execute and its data access is allowed within a context only.
    - Applets residing in different context can share objects using secure object-sharing mechanisms.
  - JCRE embraces support for PKI and use of digital signatures.
  - JCRE allows multi-application support for smartcards that allows multiple application coexist on a card without sacrificing security.

Java Card Applet Development

- The Java Card Applet development process contributes to ***Java Card security*** – The steps are: .
  1. Development of Java Card applet – implementing and compiling any Java class.
  2. Resulting Java Classes are tested using a Java Card simulator environment.
  3. The simulator tested class files are converted to a ***Converted Applet (CAP)*** file using Java Card CAP converter tool. The resulting file is a Java Card Applet.
  4. The Java Card Applet is further tested using an emulator tool – of the smartcard vendor.
  5. The tested applet will be downloaded to the Java Card using smartcard vendor-provided tool.



Java Platform Security
Key and Certificate Management Tools

Java Security Management Tools

- J2SE provides a set of security management tools
  - To administer security policies
    - Support for reading and editing Java policy files
  - PKI Management
    - Support for managing keys and digital certificates
  - Signing JAR files
    - Support for signing and verifying JAR files and its integrity
- In a J2SE bundle, the Java platform offers the following to support security and cryptographic related functions:
  - ***Keystore***
  - ***Keytool***
  - ***Policytool***
  - ***Jarsigner.***

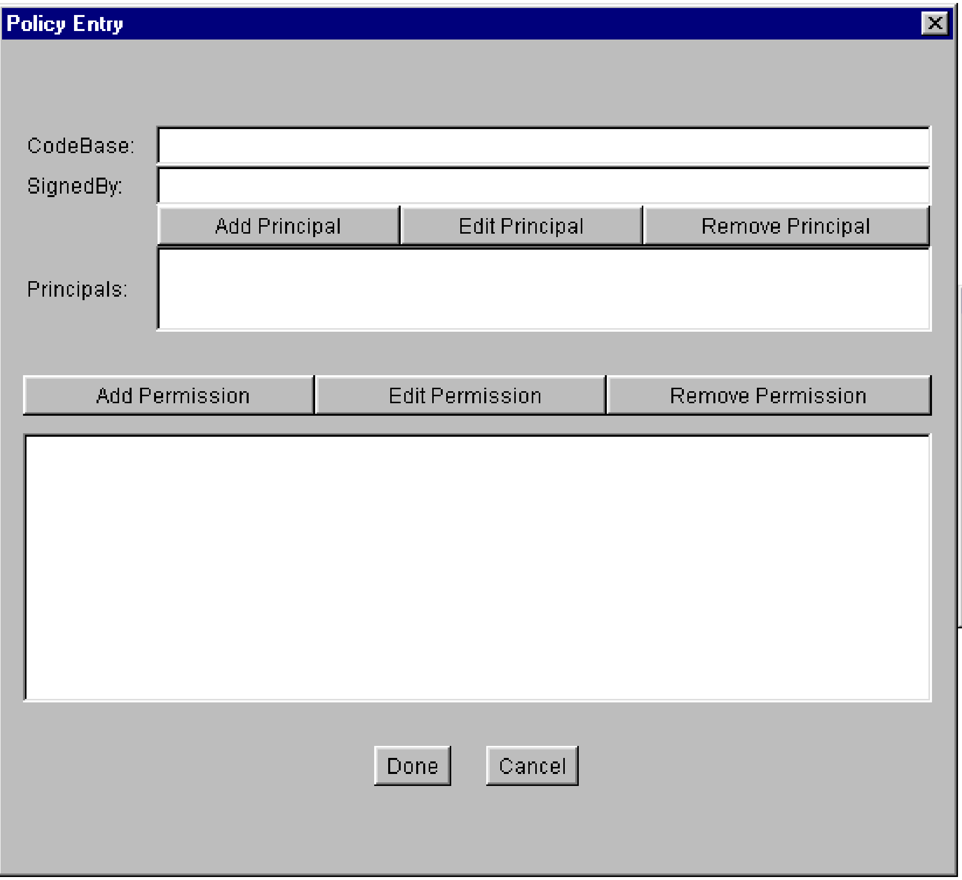
Java Keystore

- The keystore is a protected database
  - Adopts **PKCS#12** (RSA Cryptographic standard for key storage)
  - Stores keys and trusted certificate entries in a password protected file.
    - Intended for verifying or proving an identity of a person or application
    - Contains private key and chain of certificates to establish authentication with public keys.
  - Each entry in the keystore is identified by an unique alias
  - Keystore entries are stored in a **.keystore** file by default in <JRE>/lib/security directory (unless specified).
  - All trusted certificates are store in a **.cecerts** file by default in <JRE>/lib/security directory (unless specified).
- With J2SE 5.0, Java offers support for using smartcards and cryptographic devices as keystores
  - Via the support for **PKCS#11** (RSA Cryptographic Token Interface)

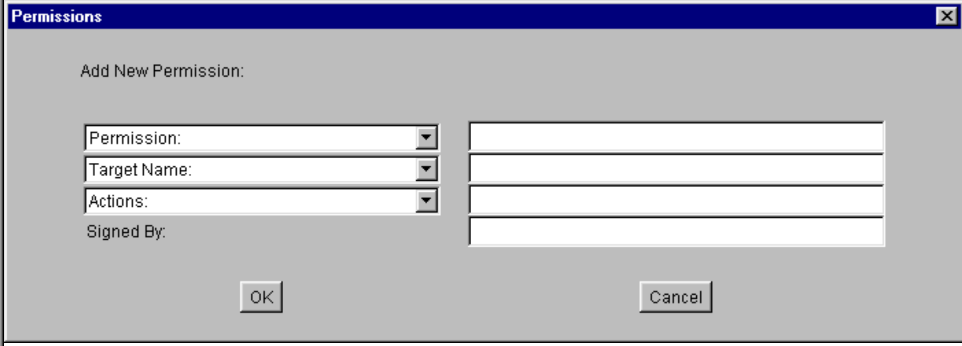
Java Policytool

- GUI tool for creating and viewing Java Security Policy configuration.
  - Use ‘**policytool**’ command
  - Allows to add Policy entries
    - Specify codebase
    - Add Principals
    - Add Permissions
    - Grant or Deny access

Policytool Options



Adding a Policy Entry



Adding Permissions

Java Jarsigner tool

- Java utility for digitally signing the Java archives (JAR)
  - Uses the signer’s private key (from Java keystore) for applying digital signature
    - After signing, the JAR file also include the copy of signer’s public key
  - The ***JARsigner*** also allows to verify a signed JAR file
  - Example (Signing a JAR file)

```
jarsigner
         -keystore /home/ramesh/.keystore
         -storepass mystorepasswd
         -keypass   mykeypasswd
         -signedjar  mysignedjar.jar
              myunsignedjar.jar  myPrivateKeyAlias
```

  - Example (Verifying a signed JAR file)

```
jarsigner
          -keystore /home/ramesh/.keystore
                       -verify  -certs  mysignedjar.jar
```

Java Keytool

- Keytool is a key and certificate management tool
  - Intended to support authentication services and verifying data integrity.
  - Supports administering public/private key pairs and associated certificates.
  - Allows to create Java keystore (JKS) or Java Cryptographic Extension Keystore (JCEKS).
  - Supports generation of Certificate signing requests (CSRs)
  - Support self-signed certificates
  - Supports storing keys and certificates in Java keystore
  - Maintenance of stored entries in a Java keystore.
  - Supports X.509v3 Certificate standard with ASN.1 standard encoding and DER formats.

Creating a Java Keystore

- Keytool creates a keystore as a file named .keystore in the user's home directory.
  - Access to keystore is protected by a password
  - By default, J2SE bundles installs a keystore in <JRE>/lib/security directory.
  - Using keytool, keystores can be created to support applications or for supporting end-users.
- A keystore is created whenever we try to add entries to non-existent keystore.
  - A keystore name can be specified using the –keystore <keystore-name> option.
  - The following options automatically create a keystore when it does not -
    - *-genkey* option is used to generate private/public key pairs
    - *-import* option is used to import a trusted certificate
    - *-identitydb* option is used to import data from legacy JDK 1.1 keystore.

Generating private/public key pairs

- Keytool allows creating public/private key pairs for testing Java applications with PKI.
  - Each generated entry contains a private key and its associated certificate chain.
    - The first certificate in the chain contains the public key corresponding to the provate key.
    - The public key is wrapped' in as X.509 certificate (***Issued as a Self-signed certificate***).
  - By default, the generated key pairs are added to the keystore.
- Example:

```
keytool –genkey –alias mykeyalias –keyalg RSA  -keypass keypasswd
                    -keystore mykeystore –storepass mystorepasswd.
```

    - -genkey option is used to generate private/public key pairs
    - -keyalg option represents the key alogorithm
  - NOTE: *Self-signed certificates must be used for testing purposes only. For production usage, acquire certificates from a trusted CA -Verisign, Entrust, etc.*

Exporting and Importing Certificates

- Exporting the public key certificate is required to support trusted interactions with the client application
  - This is done by exporting the Public key certificate from the keystore.
  - For example:

```
$ keytool -export -alias myalias -file mycertificate.cer -keystore mykeystore
Enter keystore password:  mystorepass
Certificate stored in file <mycertificate.cer>
```

- On the client-side, Import the trusted certificate in client keystore.
  - The client makes use of the Public key certificate by importing them.
  - When a certificate is imported – the keytool utility verifies the certificate for its integrity using the list of trusted CA certificates stored in .cacerts file.
  - For example:

```
$ keytool -import -alias myclientalias -file     \
          mycertificate.cer -keypass clientkeypass  -keystore clientstore -storepass clientpass
```

Creating Certificate Signing Request

- Keytool allows to create Certificate *Signing or Authentication* Request for obtaining certificates from a Certificate Authority (CA).
  - This helps to sent CSRs and obtain CA signed certificates for production use.
  - For example:

```
keytool -certReq -keystore mykeystore
   -file myCSR.csr -alias mycsralias
```

Listing and Printing Keystore entries

- To list keystore entries and also look at the contents of an entry

```
$ keytool –list –keystore mykeystore –storepass mykeystorepasswd
```

- To print certificate information and display contents of a binary certificate

```
$ keytool  -printcert  -file  mycertificate.cer
```

- To delete a keystore, just use the operating system command for deleting (.keystore) files.

Securing Java Code From Decompilation

Securing the Java Code – Why ?

- ***Reverse engineering*** is well-known security problem for Java applications.
  - Java bytecode generated by a Java compiler contains much symbolic instruction, actual Java source and debugging information
  - Using reverse-engineering mechanisms – it is possible to ***disassemble and decompile the executable Java bytecode*** into actual Java source code.
- The reverse-engineering risks and vulnerabilities include:
  - Modifying the original code and data
  - Determine the flow of Java program execution
  - Determine the algorithms in use
  - Constructing a fraudulent application from the decompiled source.
  - Stealing the intellectual property
  - Allows a hacker to apply code-level security breaches ☹

Reverse Engineering

- Reverse engineering is a process of ***decompilation for extracting actual source code*** from bytecode.
  - Disassembling the executable classes to an intermediary assembly code and then decompiling to higher-level abstraction of the bytecode.
  - The noticeable difference of resulting source code is just absence of comments.
  - Several commercial and freeware tools available for Java bytecode decompilation.
  - For example, using freeware “JAD” (http://www.kpdus.com/jad.html)

```
jad -r  -d /home/ramesh/directory_for_sourcecode   MyJava.class
```

Preventing Decompilation

- Decompilation of Java executable can be restricted by any of the following ways:
  - **Code Authentication**
    - Adopts evaluation and verification of executable code for trusted sources, runtime checks, predictable behavior and output.
  - **Encryption and Decryption**
    - Using encryption and decryption of executable code in transmission to ensure code is not accessible or tampered.
    - Limits portability of the application but works well in server-side invocation scenarios.
  - **Code Obfuscation**
    - Transformation mechanism that changes the program and generate Java code with obscure references.
      - Common methods include Structural/Layout transformation, Data transformation, String encryption, Watermarking.
    - Most popularly adopted by Java developers.

Code Obfuscation

- Java Code Obfuscation is a process of transforming the executable in a manner it affects Java bytecode decompilation
  - Decouples the relationship between the executable and the original source.
  - Most common code obfuscation techniques are based on the following:
    - ***Structural or Layout transformation***, which transforms the lexical structure of the cod by scrambling and renaming the identifiers of methods and variables.
    - ***Data transformation***, which affects the data structures represented in the program (example – changing the order of data in a list).
    - ***Control transformation***, which affects the flow control represented in the program (example – grouping of inline procedures, order of execution).
    - ***Tamperproofing and Preventive transformation*** – which makes the decompiler to fai and makes the generated code unusable.
    - ***String Encryption*** encrypts all string literals with in the executable code.
    - ***Waterproofing*** embeds a secret message in the executable that identifies the copy.
  - Several Commercial and Freeware code obfuscators available (*Intentionally avoided to mention here*).
  - Little performance overhead is common but no portability issues.

Further reading (*Shameless Plug*)

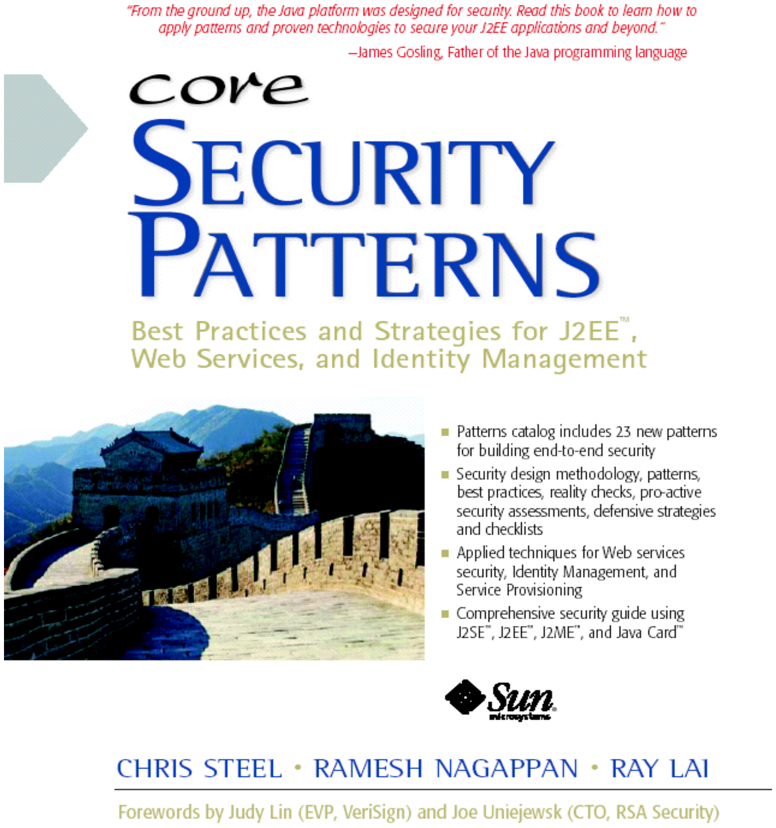
Core Security Patterns

Chris Steel, Ramesh Nagappan & Ray Lai

Prentice Hall, September 2005

&

http://java.sun.com



"From the ground up, the Java platform was designed for security. Read this book to learn how to apply patterns and proven technologies to secure your J2EE applications and beyond."

—James Gosling, Father of the Java programming language

core

SECURITY PATTERNS

Best Practices and Strategies for J2EE™, Web Services, and Identity Management

- Patterns catalog includes 23 new patterns for building end-to-end security
- Security design methodology, patterns, best practices, reality checks, pro-active security assessments, defensive strategies and checklists
- Applied techniques for Web services security, Identity Management, and Service Provisioning
- Comprehensive security guide using J2SE™, J2EE™, J2ME™, and Java Card™

Sun microsystems

CHRIS STEEL • RAMESH NAGAPPAN • RAY LAI

Forewords by Judy Lin (EVP, VeriSign) and Joe Uniejewsk (CTO, RSA Security)

Thank You

Ramesh Nagappan CISSP
nramesh@post.harvard.edu