

---

---

# Bluetooth

## *Part 11: Java ME Security*

**Kjell Jørgen Hole**

UiB



Last updated 30.03.09

Mail: [Kjell.Hole@ii.uib.no](mailto:Kjell.Hole@ii.uib.no)

URL: [www.kjhole.com](http://www.kjhole.com)

---

## Outline

[KJhole.com](http://KJhole.com)

- Java ME security mechanisms in
  - KVM (Kilobyte Virtual Machine)
  - CLDC (Connected Limited Device Configuration)
  - MIDP (Mobile Information Device Profile)
- JSR-82 security
  - security changes after connection establishment
- JSR-177, the Security and Trust Services API (SATSA)

# Java ME Security

11.3

## Levels of Java ME Security

*KJhole.com*

**Low-level/KVM security.** A pre-verifier and a KVM byte code verifier ensure that class files do not execute in any way not allowed by the KVM specification

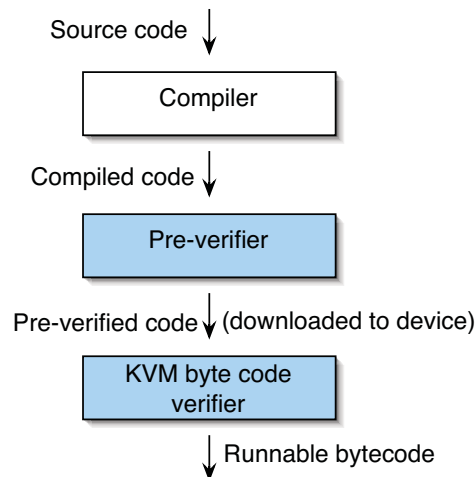
**Application-level/CLDC security.** Guarantees that Java applications can only access those libraries, system resources and other components defined by the CLDC platform

**End-to-end/MIDP security.** Safe delivery of data and code between servers and clients

11.4

# KVM Security: Byte Code Verification

---



**Figure 11-1** The pre-verifier and the KVM both verify the byte code produced by the compiler before the code is allowed to execute

11.5

## Pre-verifier

---

[KJhole.com](http://KJhole.com)

- The pre-verifier carries out off-line verification of all Java classes to enforce object, stack and control-flow safety
- *Stack maps* associated with the byte code are generated and stored in class files
- A stack map contains type information about all variables that should be on the stack at certain points in the code
- Class loading restrictions ensure that system classes (e.g. classes in `java.lang`) cannot be replaced during the pre-verification

11.6

- The KVM byte code verifier utilizes the stack maps to do less resource intensive byte code verification than the standard JVM
- Essentially, the KVM verifies the stack map instead of the whole byte code, thus, reducing the CPU and memory consumption

11.7

## CLDC Security: What is a Sandbox?

**Sandbox.** Closed environment in which an application can only access those libraries defined by the configuration, profiles, and other classes supported by the device

- An application cannot escape from a sandbox or access libraries or resources that are not part of the predefined functionality
- A malicious or erroneous application in a sandbox cannot gain access to system resources

11.8

- The application programmer can only utilize a predefined set of APIs that are defined by CLDC, the available profiles, and the classes made available by the device manufacturer
- User defined class loaders are not allowed. Hence, no program can alter the class loading procedures or order. This protects all system libraries from being overridden, bypassed or replaced

11.9

- Downloading, installing, and management of applications take place in such a way that the application programmer cannot modify or bypass the standard class loading mechanisms of the KVM
- The set of native function calls available to any application running on CLDC is closed. Thus, a developer cannot download any new code containing additional native functions

11.10

- An application programmer cannot override, modify, or add any classes to the protected system packages, i.e., system classes are protected from downloaded applications
- A Java application can only load classes from its own JAR (Java ARchive) file. This restriction ensures that
  - Java application on a device cannot interfere with each other or steal data from each other

11.11

## MIDP Security—Sensitive APIs

---

- MIDP introduces the concept of **trusted** and **untrusted** MIDlet suites
- Only trusted MIDlet suites can access **sensitive APIs** giving access to certain capabilities of the Java ME device
- The networking API and the `PushRegistry` class (automatic launching of MIDlets) are sensitive in MIDP 2.0

11.12

## Permissions and Protection Domains

---

- A trusted MIDlet suite's access to sensitive APIs are protected by [permissions](#)
  - an *User permission* requires that the user approves the access to an API
  - an *Allowed permission* gives a MIDlet direct API access
- A [protection domain](#) defines a set of permissions for trusted MIDlets

11.13

## More on Protection Domains

---

[KJhole.com](http://KJhole.com)

- The concept of a protection domain is deliberately vague, leaving implementors considerable latitude in their implementation
- Many implementations may have a single protection domain for signed MIDlets only

11.14

**Untrusted MIDlet suite.** MIDlet suite for which the origin and the integrity of the JAR file cannot be reliably determined by the device

- An untrusted MIDlet suite executes in a restricted environment where access to sensitive APIs either is not allowed or is allowed with explicit user permission
- MIDlet suites compliant with MIDP 1.0 are considered as untrusted in MIDP 2.0

11.15

- Untrusted MIDlet suites must, without explicit confirmation by the user, get access to:
  - `javax.microedition.rms`
  - `javax.microedition.midlet`
  - `javax.microedition.lcdui`
  - `javax.microedition.lcdui.game`
  - `javax.microedition.media`
  - `javax.microedition.media.control`

11.16

- With explicit confirmation by the user, untrusted MIDletsuites get access to:
  - `javax.microedition.io.HttpConnection`
  - `javax.microedition.io.HttpsConnection`

11.17

**Trusted MIDlet suite.** Security for Trusted MIDlet suites is based on protection domains. Each protection domain defines the permissions that may be granted to a MIDlet suite in that domain

- The protection domain owner specifies how the device identifies and verifies that it can trust a MIDlet suite (utilizing an X.509 PKI is one possibility)
- A trusted MIDlet suite is bound to one protection domain with the permissions that authorize access to sensitive APIs

11.18

- The basic authorization of a MIDlet suite is established by the relationships between the following elements:
  - A protection domain consisting of a set of *Allowed* and *User* permissions
  - Permissions requested by the MIDlet suite in `MIDlet-Permissions` and `MIDlet-Permissions-Opt` attributes of the JAD (Java Application Descriptor) file
  - The user who may be asked to grant permissions

11.19

- The `MIDlet-Permissions` attribute lists critical permissions needed to execute the MIDlet
- The `MIDlet-Permissions-Opt` attribute lists non-critical permissions not needed to run the MIDlet

11.20

## Application Signing (1)

KJhole.com

- MIDlet suite signing based on a X.509 PKI (Public Key Infrastructure) is optional in MIDP 2.0
- The signature lets the user verify that a MIDlet comes from a trusted entity and has not been tampered with by a third party
  - the MIDlet suite is protected by signing the JAR file
  - the signature and certificates are added to the JAD file as attributes

11.21

## Application Signing (2)

KJhole.com

- A device uses the attributes in the JAD file to verify the signed JAR file
- A root certificate\* bound to a protection domain on the device is used to complete the authentication
- **Remark** Not all mobile phones support signed applications

\*Certificate generated by a Certification Authority

11.22

## Secure Communication (1)

KJhole.com

- In MIDP 2.0, *secure* HTTP (HTTPS) communication must be implemented by one or more of the following specifications:
  - HTTP over TLS (RFC 2818) with TLS Protocol version 1.0 (RFC 2246)
  - SSL version 3.0
  - WTLS (Wireless Transport Layer Security)
  - WAP (Wireless Application Protocol) TLS Profile and Tunneling Specification

11.23

## Secure Communication (2)

KJhole.com

- A developer cannot know which specifications are implemented on a device without actually testing it
- WTLS does *not* provide end-to-end encryption
  - secure connection between phone and the WAP gateway
  - secure connection between gateway and final destination
  - gateway has access to unencrypted data
  - gateway is operated by mobile network provider

11.24

## Secure Communication (3)

KJhole.com

- A secure connection in MIDP 2.0 requires the server to have a valid certificate for authentication
- There is no support for certificate based authentication of the client, hence the client must be authenticated on the application level

11.25

## Storage System

KJhole.com

- The Record Management System (RMS) provides persistent storage in the form of [record stores](#)
- Each MIDlet Suite can have one or more record stores
- A record store is identified by a unique name, concatenation of vendor name, MIDlet suite name, and record store name
- A record store consists of a header and a body containing a number of byte arrays called records

11.26

- MIDlets can share record stores in MIDP 2.0
- The MIDlet creating a record store can choose to share it
- Sharing information is stored in the record store's header
  - default mode is private (no sharing)
- Encrypted storage is not supported in MIDP 2.0. Attacker can use equipment to read memory in mobile phone

11.27

- The MIDlet itself must encrypt the stored data
- Java ME doesn't provide a cryptographically strong PseudoRandom Number Generator (PRNG)
  - hence, the PRNG in Java ME is not suitable for generating encryption keys
- Some companies specify their own light-weight cryptographic schemes
  - unfortunately, it is very difficult to construct a cryptographically strong algorithm

11.28

# JSR-82 Security

11.29

## JSR-82 Security

*KJhole.com*

- We've earlier seen how a server can configure the security by including the following parameters in the connection string argument to `Connector.open()`
  - `authenticate`
  - `encrypt`
  - `authorize`
- Similarly, a client can set the parameters `authenticate` and `encrypt`

11.30

- The `RemoteDevice` class contains methods to
  - request a change in security for a connection at any time
  - interrogate the current security settings for a connection
- The methods to change the security are intended to be used in situations where an increased level of security is required only for a bounded set of operations

11.31

## Security Changes

- The following example shows how to change the security of a client connection after the connection is already established
- Initially, the connection was established with no authentication, encryption or authorization
- The example adds authentication and encryption to the connection to send one message, then withdraws the encryption request after the first message is sent

11.32

```

String encryptedMsg = "This message will be sent encrypted";
String clearMsg = "This message will be sent unencrypted";
OutputStream os = null;
StreamConnection con = null;
RemoteDevice remDev;
ServiceRecord record;    // Example from JSR-82 Specification Ver. 1.1

// Use the SDP client methods to obtain a
// ServiceRecord from an SDP server
    :
// Create a string including optional parameters that can be used by
// the client to connect to the service described by the ServiceRecord
String clientsConnString =
    record.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT,false);

```

11.33

```

try {
    con = (StreamConnection) Connector.open(clientsConnString);
    remDev = RemoteDevice.getRemoteDevice(con);
    if (!remDev.isEncrypted()) {

        // The connection to remDev is not currently encrypted,
        // so turn on encryption
        if (!remDev.authenticate() || !remDev.encrypt(con, true)) {

            // quit since unable to turn on encryption
            return;
        }
    }
}

```

11.34

```
// If we reach this point, then the server device has been
// authenticated, and all communications between the client
// device and the server device over con (or any other
// connection) are being encrypted.
os = con.openOutputStream();

// Send encrypted data to the server device
os.write(encryptedMsg.getBytes());
```

11.35

```
// Withdraw the request for encryption
if (remDev.encrypt(con, false)) {

    // Send unencrypted data to the server device
    // since successful in turning off encryption
    os.write(clearMsg.getBytes());
} else {

    // Send encrypted data to the server device
    // since unable to turn off encryption
    os.write(encryptedMsg.getBytes());
}
os.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
```

11.36

```
} finally {  
    if (con != null) {  
  
        // No need to do remDev.encrypt(con, false)  
        // before closing the connection  
  
        try {  
            con.close();  
        } catch (Exception e) {  
        }  
    }  
}
```

11.37

# JSR-177 Security

## Security And Trust Services API (SATSA)

11.38

- SATSA relies on a Security Element (SE) implemented in software or hardware
  - software component
  - dedicated hardware
  - removable [smart card](#)
- The SE realization is transparent to the developer, the interaction with the SE is handled by the SATSA implementation
- See [java.sun.com/products/satsa/](http://java.sun.com/products/satsa/)

11.39

- Smart cards are of particular interest to developers writing MIDlets for smartphones because
  - keys and certificates can be stored on the card
  - data can be signed without private key leaving card
  - high-end cards are tamper-resistant
  - PIN or password required to access information on card

11.40

- The SATSA specification defines four APIs
- Two APIs add functionality for smart card interaction

**SATSA-APDU** supports communication using the Application Protocol Data Unit (APDU) protocol defined by ISO7816-4 specification

**SATSA-JCRMI** enables high-level communication utilizing the Java Card Remote Method Invocation Protocol (JCRMI)

11.41

- The remaining two APIs do not require a smart card, they can be used with any SE

**SATSA-PKI** enables a MIDlet to request digital signatures from an SE, providing authentication and possibly non-repudiation. Client certificate management and key generation are also provided

**SATSA-CRYPTO** offers message digests, digital signature verification, and chipers

11.42

- SATSA is distributed as part of the Java Runtime Environment (JRE) in some smartphones
- The mobile phone OS must therefore be fully trusted because the JRE depends on services from the OS
- The SATSA specification states that both SATSA itself and applications using SATSA must trust the OS

11.43

## SATSA Shortcomings

---

- Client certificates stored by SATSA cannot be used for authentication over HTTPS links
- SATSA generates signed messages on a standardized format, but it's cumbersome to validate the signatures
- SATSA doesn't validate certificates
- Cryptographic keys stored on a smart card can not be used for encryption, only signing

11.44

- Developing “secure” Java ME applications is difficult and time consuming due to
  - limitations in the Java ME security APIs
  - bugs in real devices
- Implementing authentication and encryption based on public-key certificates remain complex tasks