F O R U M   N O K I A

MIDP 2.0: Tutorial On Signed MIDlets

Version 1.1; July 15, 2005

NOKIA

Contents

Change History

May 19, 2004	Version 1.0	Initial document release
July 15, 2005	Version 1.1	Chapter 5 and Appendix A added. Minor updates throughout the document.

1 Introduction

This document is a tutorial on how to create signed MIDlet suites, following the recommendations of MIDP version 2.0 [MIDP 2.0] and the Java™ Technology for the Wireless Industry [JTWI] specifications. It assumes that you are familiar with Java programming, and the basics of MIDP programming, for example, by having read the Forum Nokia document *Brief Introduction to MIDP Programming* [MIDPPROG]. Knowledge of public-key encryption and digital signatures is also recommended [PKCS], as well as their usage in the Java language [Java Certificates]. The document concentrates on providing practical advice and real-world cases about MIDlet signing.

The new security model introduced in MIDP 2.0 is relevant to MIDlet developers who need to use functions or APIs that are considered sensitive. These are, for example, APIs for network connections, messaging, and Push functionality. In addition, MIDP optional packages may contain additional restricted APIs.

Signing MIDlets is relevant because of the multitude of benefits for the developer and the end user. Because the signing procedure entails an extra effort in terms of cost and time to acquire the proper certificates and do the signing properly, the motivations to do so should be clearly stated.

Basically any MIDlet that uses a protected API will benefit from signing. Because of this, it should also be clear that certain types of applications do not really need signing. For example, a game that only uses graphical features and never, or very seldom, accesses the network, would not particularly benefit from signing. On the other hand, a MIDlet that often connects to the network, sends SMS messages, or accesses regularly the device PIM's data will have clear benefits from signing.

Signing a MIDlet brings several benefits, including:

- Depending on the security policy, certain functionality may not be accessible at all unless the MIDlet is signed. In some other cases, some features could be accessible to an unsigned MIDlet only if the user manually changes the security parameters. For example, access to write user data is denied by default for unsigned MIDlets.
- Security policies in certain devices, operators, or sales channels may reject the installation of unsigned MIDlets.
- User experience improves notably by avoiding security prompts when calling protected APIs.
- The installation of unsigned MIDlets gives a warning about the untrusted nature of the software. On the contrary, when installing a signed MIDlet, no warnings are given and with successful authentication, the installation proceeds seamlessly.
- Less restrictive security policy applies to signed MIDlets with the user having an option to make it more restrictive if he or she desires to do so.
- Ensures that modified versions of the software cannot be distributed under the signer's name.

In brief, the signing process brings important benefits to the developer and end users; even if it introduces an extra expense to the development process. This document is intended to explain the fundamentals of the signing process and give practical tips to successfully sign a MIDlet.

2 Security in MIDP 2.0

One of the main benefits of MIDP is the openness of the platform, which enables anybody to write software that can run on MIDP devices. MIDlet suites can be downloaded from the network in an anonymous fashion, and as such, there are some security and privacy issues that the user may be concerned about: Can a MIDlet read private data and send it to an unknown server? Can it make unauthorized calls that cost money to the user? Can rogue programs run on the device and potentially cause problems?

Besides the Java language safety features, such as garbage collection and array bounds checking, the MIDP specification adds some additional safety measures. In MIDP 1.0 [MIDP 1.0], the security restrictions are divided into low-level machine security and application-level machine security. Low-level security relates to the verification of the MIDlet suite's class files. The verification is partially done by the developer in the preverification step and partially on the device.

To further enhance security, MIDlet suites run in a "sandbox" that restricts the available APIs to a limited set. There are some additional restrictions, such as the absence of user-defined class loaders and the prohibition of adding new native functions. These restrictions are part of the application-level machine security [CLDC].

In MIDP version 2.0, the security model was enhanced to allow MIDlets access APIs that are considered sensitive. For example, making an HTTP connection is one of such sensitive operations because it may involve monetary costs for the user. MIDP 2.0 introduces the concept of *trusted* and *untrusted* MIDlets. An *untrusted* MIDlet suite has limited access to restricted APIs, requiring user approval depending on the security policy of the device. On the other hand, *trusted* MIDlet suites can acquire some permissions automatically depending on the security policy.

Permissions are used to protect APIs that are sensitive and require authorization. The MIDP 2.0 implementation has to check whether a MIDlet suite has acquired the necessary permission before invoking the API. Permissions have names starting with the package name in the same way as Java™ 2 Platform, Standard Edition (J2SE™) permissions. For instance, the permission to make an HTTP connection is called `javax.microedition.io.Connector.http`. Permissions are documented along the class or package documentation of the protected API.

2.1 Protection Domains

Protection domains are a key security concept in MIDP 2.0. A protection domain is a set of permissions and interaction modes. Those permissions can be either automatically granted or deferred until user approval. They are called *allowed* and *user* permissions respectively. When a MIDlet suite is installed, it is assigned to a given protection domain and acquires its permissions and interaction modes.

User permissions may require an explicit approval by the user. The user can either deny the permission or allow it. There are three interaction modes in which *user* permissions can be granted: *blanket*, *session*, and *oneshot*. When the *blanket* interaction mode is used, the MIDlet suite acquires the permission as long as the suite is installed, unless explicitly revoked by the user. The *session* mode requests the user authorization the first time the API is invoked and its validity is guaranteed while any of the MIDlets in the same suite are running. Finally, *oneshot* permissions request user approval every time the API is invoked. The protection domain determines which of the modes are available for each user permission as well as the default mode.

A MIDlet suite has to request permissions declaratively using the *MIDlet-Permissions* and *MIDlet-Permissions-Opt* attributes, either in the application descriptor or in the manifest file. *MIDlet-Permissions* contains permissions that are critical for the suite's functionality and *MIDlet-Permissions-Opt* indicates desired permissions, which are not so fundamental for the core functionality. For example, for an application's functionality it is critical to make HTTP connections to work. It may also use HTTPS connections for improved security, but it is not so vital. In this case, the application descriptor could look like this:

```
MIDlet-Permissions: javax.microedition.io.Connector.http
MIDlet-Permissions-Opt: javax.microedition.io.Connector.https
```

One of the requirements to install a MIDlet and assign it to a given domain is that the requested permissions should be a subset of the permissions given to the protection domain. As an example, the Series 60 MIDP Emulator Prototype 2.0 (SDK) includes a domain called *minimum*, which has no permissions at all. If a signed MIDlet that includes any permission request is to be installed in the *minimum* domain, the installation procedure will fail because the domain does not include any of those permissions. For the same reason MIDlet suites with misspelled *MIDlet-Permissions* properties will fail to install.

Each protection domain, except for the *untrusted* domain, is associated to a set of root certificates. When signing a MIDlet suite, it is necessary to use a public key certificate that can be validated to one of those root certificates. This association will be used to assign the MIDlet suite to a given protection domain. The relationship between root certificates and protection domain is that a domain can be associated to many root certificates, whereas a root certificate is associated to only one domain.

The MIDP 2.0 specification recommends four protection domains for GSM/UTMS devices: the *manufacturer*, *operator*, *trusted third party*, and *untrusted* domains. The *manufacturer* domain uses root certificates belonging to the device producer. The *operator* domain is used for the network operator MIDlets and may use root certificates available on storages such as SIM cards. The *trusted third party* domain will encompass well-known Certificate Authorities' (CA) root certificates. Finally, the mandatory *untrusted* domain has not an associated root certificate and is used for unsigned and MIDP 1.0 MIDlet suites.

Java Code signing certificates which have been acquired from well-known CAs and are included in the target devices, can be used to sign MIDlets. Alternatively, diverse developer programs, for example, from operators or other software publishers, offer signing services when an application is submitted to them.

Since the amount of domains and their associated permissions may deviate from the recommendations of the MIDP 2.0 addendum in some networks, you should seek information from the network operator your MIDlet suite is targeting to. Additionally, it may be worthwhile checking the set of root certificates available on a given device for Java authentication.

2.2 Untrusted MIDlet

The MIDP 2.0 specification defines as *untrusted* a MIDlet suite for which the origin and integrity of the JAR file cannot be verified by the device. This does not mean that the MIDlet cannot be installed or executed; it just means that the access to restricted operations requires explicit user permission. All MIDP 1.0 MIDlets are by default *untrusted*.

Untrusted MIDlets can call any API without permission if it is not protected. That includes all classes in the following packages:

```
java.util
java.lang
java.io
javax.microedition.rms
javax.microedition.midlet
javax.microedition.lcdui
javax.microedition.lcdui.game
javax.microedition.media
javax.microedition.media.control
```

In case an untrusted MIDlet suite tries to invoke a protected API and does not have the permission, for example if the user rejects the permission's prompt, a `SecurityException` will be thrown. The MIDlet should catch those exceptions and handle them properly.

In Nokia's MIDP 2.0 devices, MIDP 1.0 MIDlets will get an `IOException` instead of a `SecurityException` when the MIDlet cannot acquire the permission. This is to ensure backward compatibility with MIDP 1.0 MIDlets that do not expect a `SecurityException` to be raised, for example, when opening an HTTP connection.

It is also worth noticing that the Nokia's UI API is not protected. This includes the classes in the `com.nokia.mid.sound` and `com.nokia.mid.ui` packages.

2.3 Trusted MIDlets

If the device can verify the authenticity and integrity of the MIDlet suite and assign it to a protection domain, the MIDlet suite is said to be *trusted*. A *trusted* MIDlet suite will have its requested permissions granted according to its protection domain. For example, if the `javax.microedition.io.Connector.http` permission was requested and the protection domain has set the permission as *allowed*, no user confirmation will be needed to open an HTTP connection.

Do not confuse the concept of *trusted* MIDlet suite with the *trusted* protection domain. Each *trusted* MIDlet suite is assigned to a particular protection domain depending on the authorization mechanism.

To sign your MIDlet you will need a code-signing certificate conforming to the X.509 Public-Key Infrastructure (PKI) specification [X.509]. The device will use a set of root certificates to validate the MIDlet suite's certificate. Among them it is expected to find the manufacturer's root certificate and well-known CA's root certificates. Depending on the CA's policy, the certificate can include any number of intermediate certificates that should also be included in the MIDlet.

All the certificates used to sign the MIDlet are to be included in the suite's `JAD` file using the *MIDlet-Certificate-<n>-<m>* attributes. Besides the certificates, the SHA1 digest of the JAR file signed with the suite's certificates is stored in the JAD file on the *MIDlet-Jar-RSA-SHA1* attribute.

The process of verifying whether a MIDlet suite is trusted is done when the suite is downloaded or installed. The application manager checks the JAD file and if it contains a *MIDlet-Jar-RSA-SHA1* attribute, it will initiate the authentication and authorization procedures.

During the authentication, it reads the chain of certificates in the JAD file written in the attributes *MIDlet-Certificate-<n>-<m>* (where *n* and *m* are numbers indicating the certificate chain), and tries to validate the certificate with one of the root certificates.

If the certificate chain can be validated to a root certificate, the device will extract the public key from the MIDlet's suite certificate and use it to decrypt the *MIDlet-Jar-RSA-SHA1* attribute. The resulting value will be the SHA1 digest of the MIDlet JAR. The MIDP implementation will then calculate the same digest value from the JAR file.

If both digests are equal, the MIDlet suite is authenticated and it will be allocated to the protection domain assigned to the root certificate. If one or more requested permissions on the *MIDlet-Permissions* attribute are not in the protection domain, the installation will not be allowed to continue. On the other hand, if some of the requested *MIDlet-Permissions-Opt* are not in the protection domain, the installation can proceed.

2.4 Function Groups

Instead of making the user manage each individual permission requested by a MIDlet suite, permissions can be grouped by functionality in Function Groups. The user will then give permissions to Function Groups, for example the “Net Access” Function Group when using network features, rather than explicitly for the `javax.microedition.io.Connector.http` permission. Using a higher-level concept like Function Groups instead of single permissions is better suited for user interaction in small devices.

The MIDP 2.0 and JTWI specifications have defined the following Function Groups:

- *Net Access*: Contains permissions related to network data connections.
- *Messaging*: Set of permissions related to sending or receiving messages like SMS.
- *Auto Invocation*: Permissions related to automatically starting a MIDlet, for example by Push Registration.
- *Local Connectivity*: Permissions related to connection via local ports like IrDA or Bluetooth.
- *Multimedia Recording*: Permissions that allow to record images, audio, video, and so on.
- *Read User Data*: Set of permissions to read user’s data, such as phone book or calendar entries.
- *Write User Data*: Permissions related to writing user’s data.

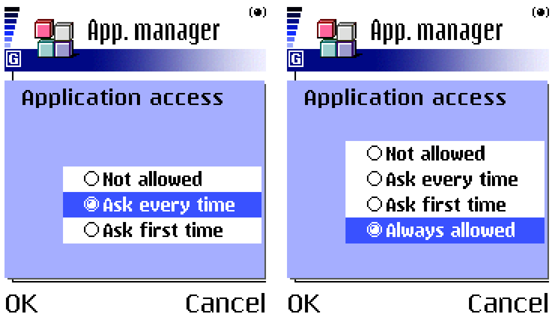
The availability of these Function Groups depends on the device’s capabilities. For instance, “Multimedia Recording” would not be available for devices without media capturing facilities or for those where the Media API is not available.

The list of Function Groups presented in this document may be extended in the future when new optional APIs are developed.

Function Groups also determine which interaction modes are available for the *trusted* and *untrusted* domains. For example, in the *untrusted* domain, “Net Access” can be set as *session* or *denied*, with blanket being disabled. On the other hand in the *trusted* domain, *oneshot*, *blanket*, and *denied* are allowed.

In addition of the above, Function Groups contain exclusion rules to forbid some combinations of permissions that are not allowed simultaneously. For instance, it would not be desirable to have *blanket* permission for “Net Access” and “Auto Invocation” at the same time. If this were the case, a MIDlet would set itself for periodical restart and make network connections without user permission, creating unsolicited charges for the user.

An example of Function Groups usage can be seen in the Nokia 6600 screenshots shown in Figure 1. They show the ‘Network Access’ Function Group found in the Application Manager application. The screenshot on the left is for an unsigned MIDlet suite, and the one on the right is for the same MIDlet signed with a Verisign certificate. Clearly the difference is that with the signed MIDlet the user can allow the MIDlet to open network connections without user’s prompts.



App. manager

Application access

○ Not allowed
[✓ Ask every time]
○ Ask first time

OK Cancel

App. manager

Application access

○ Not allowed
○ Ask every time
○ Ask first time
[✓ Always allowed]

OK Cancel

Figure 1: Network Access Function Group for an unsigned and signed MIDlet suite (Nokia 6600 screenshots)

3 Example of a Signed MIDlet

3.1 Description

A very simple example is described here to show how to request permissions and how the signing procedure works. The MIDlet simply loads a document using an HTTP connection. The actual content of the document is not relevant. The important point is that the MIDlet needs to call a sensitive operation. This chapter will examine both the scenario of building an *untrusted* MIDlet as well as the scenario of building a *trusted* MIDlet. The MIDlet has only one class, `SignedMIDlet`, which sets a textbox and adds two commands for closing the MIDlet and loading a page. The page's URL is set in the application descriptor. When the page is loaded, the amount of bytes read is displayed on the screen. In case of an error, the error message is displayed. The network operation is done in a separate thread.

The example was developed using the Nokia Developer's Suite for J2ME™ 3.0 (NDS for J2ME™ 3.0) and Series 60 Prototype 2.0 (SDK) emulator.

3.2 SignedMIDlet.java

This is the only class of the MIDlet suite, and it sets a form to be displayed. The `Load` command will attempt to open an HTTP connection and count the target page size in bytes. Any errors will also be displayed.

```
import java.io.*;
import javax.microedition.io.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class SignedMIDlet
  extends MIDlet
  implements Runnable, CommandListener
{
  private final TextField text
    = new TextField("", "", 256, TextField.ANY);
  private final Form form = new Form("HTTP Result");
  private final Command exitCommand
    = new Command("Exit", Command.EXIT, 1);
  private final Command okCommand = new Command("Load", Command.OK, 1);


  public SignedMIDlet() {}


  public void startApp()
  {
    if (Display.getDisplay(this).getCurrent()==null)
    {
      form.setCommandListener(this);
      form.addCommand(exitCommand);
      form.addCommand(okCommand);
      form.append(text);
      Display.getDisplay(this).setCurrent(form);
    }
  }


  public void pauseApp() {}


  public void destroyApp(boolean unconditional) {}
  public void run()
  {
    // do an action that requires permission, e.g. HTTP connection
    try
    {
      String url = getAppProperty("url");
      HttpConnection connection =
          (HttpConnection) Connector.open(url);
      InputStream in = connection.openInputStream();
      int counter = 0;
      int ch;
      while ((ch = in.read()) != -1)
      {
        counter++;
      }
      in.close();
      connection.close();
      text.setString("Bytes read: " + counter);
    }
    catch (IOException e)
    {
      text.setString("IOException: " + e.getMessage());
    }
    catch (SecurityException e)
    {
      text.setString("SecurityException: " + e.getMessage());
    }
  }


  public void commandAction(Command command, Displayable displayable)
  {
    if (command == exitCommand)
    {
      notifyDestroyed();
    }
    else if (command == okCommand)
    {
      new Thread(this).start();
    }
  }

}
```

3.3 Permissions

The MIDlet needs to inform which permissions it wants to acquire. In the case of `SignedMIDlet`, the `javax.microedition.io.Connector.http` permission will be requested. In NDS for J2ME™ 2.1, you can add the permissions under the **Create Application Package** option in the **Permissions** tab, as shown in Figure 2. The **Permissions** tab is displayed only if you select a MIDP 2.0 emulator as the default and the MIDP 2.0 is selected as the MicroEdition-Profile in the Attributes tab.

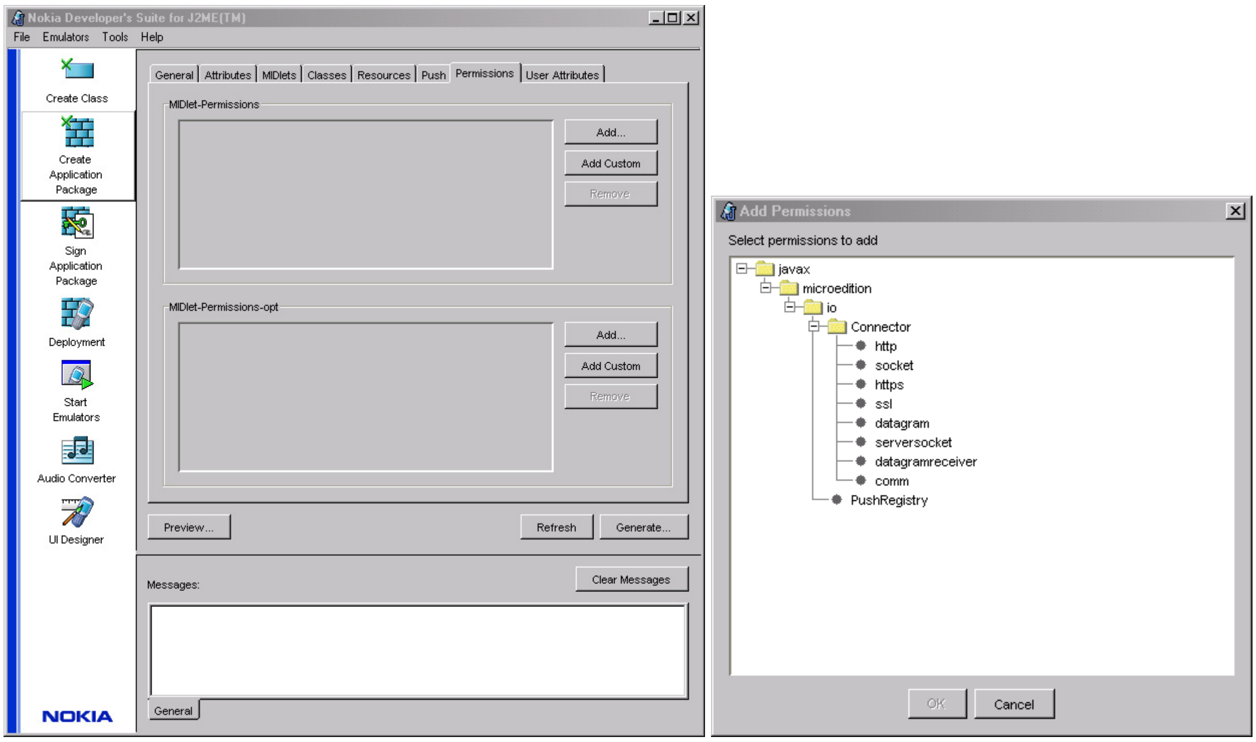


Figure 2: The Create Application Package and Permissions dialog

Alternatively, the requested permissions can be written directly to the JAD application descriptor.

3.4 Signing Procedure

To sign your MIDlet's JAR you need to obtain a code-signing certificate. Note, however, that SSL certificates are not acceptable to sign code. As a developer, you have to be aware what certificates will be acceptable for your target device. In this example, the JAR will be signed by using a certificate which has been signed by Verisign and purchased through the normal channels.

In this example a certificate signed by Verisign was used. Verisign offers several types of code signing certificates (http://www.verisign.com/products/signing/code/), each for a different application. For MIDlet suites signing the *Sun Java Signing Digital ID* type is required. Remember that CAs will usually charge a fee for issuing the certificate and they have a limited validity period. Verisign will have to verify your identity usually by requesting some proof of you identity and organization.

When signing the JAR file, the certificate will include an identifier of you as the developer or your company, so that the MIDlet can be associated with the signer. This is also an important input for technical non-repudiation.

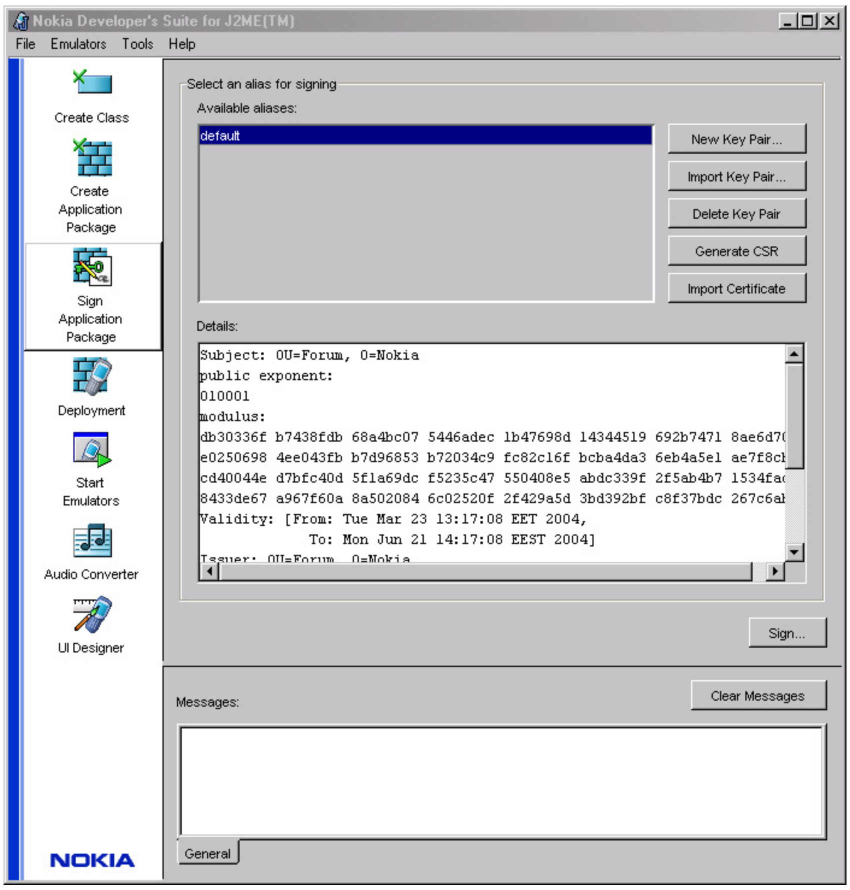
For experimentation purposes, it is possible create your own “self-signed” certificate and load it to the emulator. However, this approach will not work in actual Nokia devices since the set of root certificates is closed.

The signing procedure can be done using the NDS for J2ME™ 2.1 GUI or command line utilities. Both procedures are explained below.

3.4.1 Signing procedure using the NDS for J2ME™ 2.1 GUI

When using the GUI to sing the MIDlet's JAR, you would proceed as follows (steps 2 to 5 are done only once per each certificate):

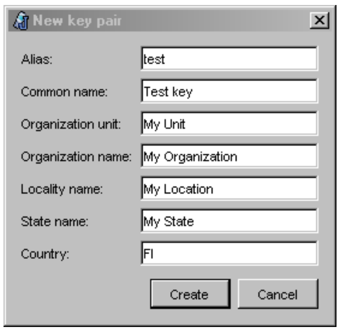
1. Go to the **Create Application Package** tool, add the needed permissions, and package your application.
2. Go to the **Sign Application Package** tool. This will display the screen shown in Figure 3.



```
Subject: OU=Forum, O=Nokia
public exponent:
010001
modulus:
db30336f b7438fdb 68a4bc07 5446adec 1b47698d 14344519 692b7471 8ae6d7
e0250698 4ee043fb b7d96853 b72034c9 fc82c16f bcba4da3 6eb4a5e1 ae7f8c
cd40044e d7bfc40d 5f1a69dc f5235c47 550408e5 abdc339f 2f5ab4b7 1534fa
8433de67 a967f60a 8a502084 6c02520f 2f429a5d 3bd392bf c8f37bdc 267c6a
Validity: [From: Tue Mar 23 13:17:08 EET 2004,
             To: Mon Jun 21 14:17:08 EEST 2004]
Issuer: OU=Forum, O=Nokia
```

Figure 3: The Sign Application Package screen

3. Here you can create a new key using the **New Key Pair...** button. This will display the window shown in Figure 4 requesting values for the key's attributes alias, domain, and company name.



Alias:	test
Common name:	Test key
Organization unit:	My Unit
Organization name:	My Organization
Locality name:	My Location
State name:	My State
Country:	FI

Figure 4: Create a new key pair

4. You can use this key as your self-signed certificate. In this example case, the **Generate CSR** button was used in NDS for J2ME™ 2.1 to generate a Certificate Signing Request (CSR) file. This file was sent to Verisign to purchase a code-signing certificate. Normally all CAs will proceed to verify your identity by different means and then issue the certificate.
5. Once the CA returns the certificate, it can be imported to NDS for J2ME™ 2.1 using the **Import Certificate** button.

6. Now that the certificate has been imported, you can press the **Sign...** button and select the JAD file to be signed.
7. Once this is done you can open the JAD file that now includes three new attributes encoded using the base 64 method. These include two certificates and the JAR's signature. Two certificates are obtained due to Verisign's practice of including an intermediate certificate.

```
MIDlet-Certificate-1-1:
MIIEHjCCA4egAwIBAgIQce1yiTZcQTou7Hh+fx0kEzANBgkqhkiG9w0BAQUFADCBpzEXMBUG
A1UEChMOVmVyaVNpZ24sIEluYy4xHzAdBgNVBAsTFlZlcmlTaWduIFRydXN0IE5ldHdvcmsx
OzA5BgNVBAsTMlRlcm1zIG9mIHVzZSBhdCBodHRwczovL3d3dy52ZXJpc2lnbi5jb20vcnBh
IChjKTAxMS4wLAYDVQQDEyVWZXJpU2lnbiBDbGFzcyAzIENvZGUgU2lnbmluZyAyMDAxIENB
MB4XDTAzMDgxNDAwMDAwMFoXDTA...

MIDlet-Certificate-1-2:
MIIDpjCCAw+gAwIBAgIQbaJ66Skutt3AqAAdR247aTANBgkqhkiG9w0BAQUFADBfMQswCQYD
VQQGEwJVUzEXMBUGA1UEChMOVmVyaVNpZ24sIEluYy4xNzA1BgNVBAsTLkNsYXNzIDMgUHVi
bGljIFByaW1hcnkgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkwHhcNMDExMjAzMDAwMDAwWhcN
MTExMjAyMjM1OTU5WjCBpzEXMBUGA1UEChMOVmVyaVNpZ24sIEluYy4xHzAdBgNVBAsTFlZl
cmlTaWduIFRydXN0IE5ldHdvcmsx...

MIDlet-Jar-RSA-SHA1:
2dQkKhAfMmy/WVIUdh4/O80R1RFWFuA6qi3ImzxolQfi+PnX4cU0PWonLdB86G/WQ/aRdqlc
6/Z0ibi+JYmZUek5zoFCp7nijxoP...
```

3.4.2 Signing procedure using command line utilities

It is possible to do the same signing procedure using command line utilities (Steps 1 to 4 are done only once per each certificate):

1. The key is created using the keytool utility (included in J2SE) with a command such as the following:

```
keytool -genkey -alias SignedMIDlet -keyalg RSA -keystore midlets.sks
Enter keystore password:  midlets
What is your first and last name?
  [Unknown]:  Test Key
What is the name of your organizational unit?
  [Unknown]:  My Unit
What is the name of your organization?
  [Unknown]:  My Company
What is the name of your City or Locality?
  [Unknown]:  My Location
What is the name of your State or Province?
  [Unknown]:  My State
What is the two-letter country code for this unit?
  [Unknown]:  FI
Is CN=Test Key, OU= My Unit, O="My Company", L=My Location, ST=My State,
C=FI correct?
  [no]:  yes

Enter key password for <SignedMIDlet>
        (RETURN if same as keystore password)
```

This will create a new keystore *midlets.sks* with the password *midlets* and a new key with the given distinguished name fields. There will be a new *midlets.sks* file in your current directory. Alternatively, you could omit the `-keystore` command and store the key in the default Java keystore.

You can list all the stored keys with the following command:

```
keytool -list -keystore midlets.sks
Enter keystore password:  midlets
Keystore type: jks
Keystore provider: SUN
```

```
Your keystore contains 1 entry

signedmidlet, Dec 6, 2003, keyEntry,
Certificate fingerprint (MD5):
C7:8C:F1:63:17:62:0A:43:6A:F7:F1:5F:E1:EC:66:73
```

8. The CSR file can be generated using the following command:

```
keystore -certreq -alias SignedMIDlet -keystore <keystore
path>\midlets.sks -keypass midlets -file request.csr
```

9. The generated CSR file can then be used to purchase a code-signing certificate from a CA.
10. The certificate returned by the CA can then be imported. Notice that the keystore needs to contain already the certificate used to generate the CSR; otherwise it cannot be imported.

```
keytool -import -file <cert_file> -alias SignedMIDlet -keystore
<keystore path>\midlets.sks -keypass midlets
```

11. The JadTool.jar provided with NDS for J2ME™ 2.1 can perform the actual JAR signing with the following command:

```
java -jar <NDS2.1 path>\bin\lib\JadTool.jar -addjarsig -keypass midlets
-alias SignedMIDlet -keystore <keystore path>\midlets.sks -inputjad <jad
path>\Signed.jad -outputjad <jad path>\Signed.jad -jarfile <jar
path>\Signed.jar
```

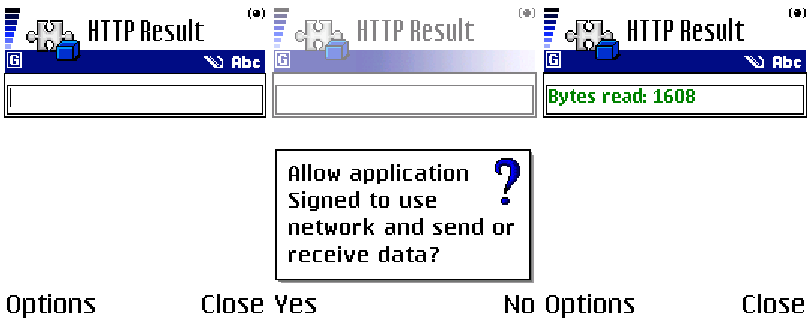
12. Now Signed.jad contains the original properties plus the *MIDlet-Jar-RSA-SHA1* attribute with the digital signature. The next step is to add the certificate using again JadTool.jar with the following command:

```
java -jar <NDS2.1 path>\bin\lib\JadTool.jar -addcert -alias SignedMIDlet
-keystore <keystore path>\midlets.sks -inputjad <jad path>\Signed.jad -
outputjad <jad path>\Signed.jad
```

4 Study Cases

4.1 Untrusted MIDlet

To run the example as an *untrusted* MIDlet, install the MIDlet suite in a MIDP 2.0 (in this tutorial a Nokia 6600 has been used) device or emulate the processes in NDS for J2ME™ 2.1 as usual with the **Start Emulators / Emulate** command, without signing the suite. Because the MIDlet is untrusted, the user is prompted for permission when the MIDlet tries to open an HTTP connection. If the user grants permission, the result will look as shown in Figure 5.

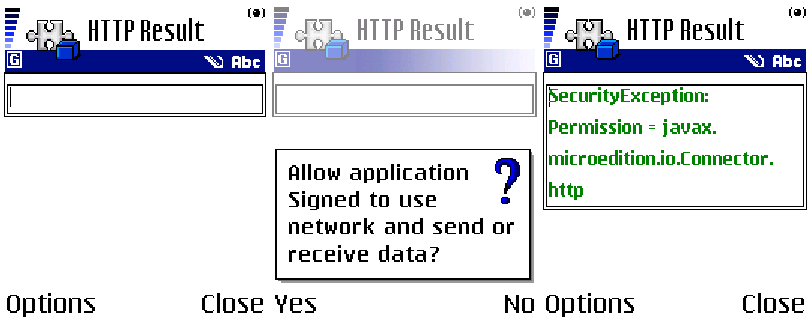


Allow application Signed to use network and send or receive data?

Bytes read: 1608

Figure 5: Untrusted MIDlet with user permission granted (Nokia 6600 screenshots)

If the user denies access to the sensitive API, a `SecurityException` will be thrown as shown in Figure 6.



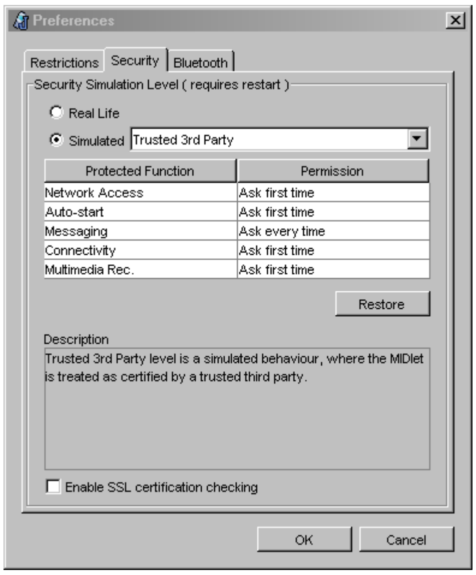
Allow application Signed to use network and send or receive data?

SecurityException: Permission = javax. microedition.io.Connector. http

Figure 6: Untrusted MIDlet with user permission denied

Notice that the Function Group “Net Access” is by default set to “Ask every time” for the *untrusted* domain. This means that every time the *Load* command is invoked, the user will be prompted for access. It is possible to change this behavior in the device as shown in Figure 1.

Using the Concept SDK this is implemented by changing the default domain the MIDlet suite is run into, for example, to the *Trusted 3rd Party* domain. In that case the user will be prompted only once while the MIDlet suite is running. The different domains available for the Concept SDK, and its Function Groups settings, are shown in the Figure 7.



Security Simulation Level (requires restart)

Real Life
[✓ Simulated] Trusted 3rd Party

Protected Function	Permission
Network Access	Ask first time
Auto-start	Ask first time
Messaging	Ask every time
Connectivity	Ask first time
Multimedia Rec.	Ask first time

Description
Trusted 3rd Party level is a simulated behaviour, where the MIDlet is treated as certified by a trusted third party.

Enable SSL certification checking

Figure 7: Protection domains

If you change the security domain to *minimum*, a `SecurityException` will be thrown without a user prompt as shown in Figure 6. This is due to the fact that the *minimum* domain has no permissions assigned to it.

Using the *Real Life* mode, the Concept SDK will attempt to perform the authorization and authentication procedure. Since the MIDlet suite was not signed, the emulator will run the suite under the *untrusted* protection domain.

4.2 Trusted MIDlet

To make a *trusted* MIDlet suite, you first have to sign the MIDlet suite as explained in Section 3.4, “Signing Procedure.” When the MIDlet is installed in a MIDP 2.0 device, the authentication and authorization procedure is carried on, and if successful, the MIDlet suite will be installed with the appropriate rights.

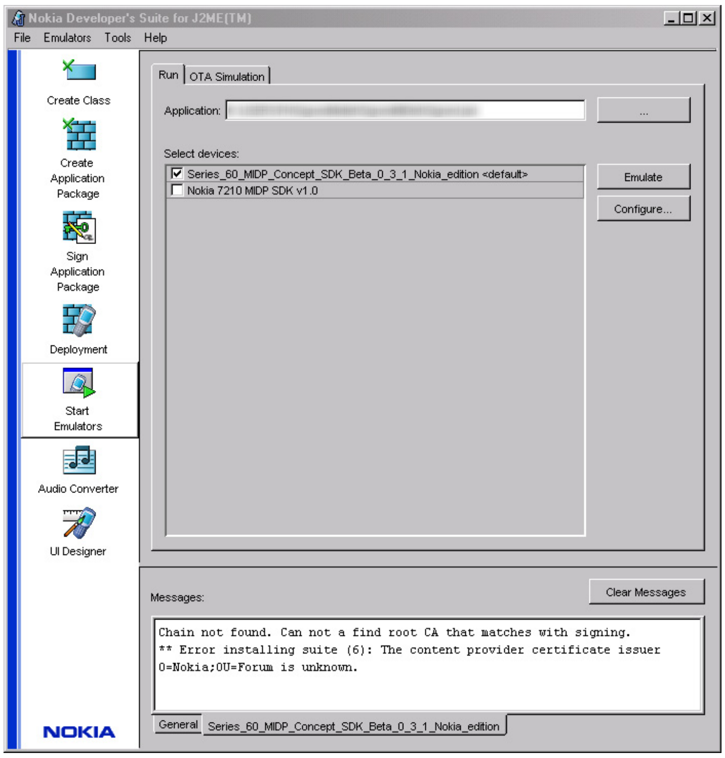
The Concept SDK contains a security simulation level called *Real Life* that can be used to test signed MIDlets. Only using it, the Concept SDK will execute the authentication and authorization process on the executed MIDlet suites. In that case, the Application Management Service is in charge of the process of verifying the *MIDlet-Jar-RSA-SHA1* attribute and run the suite in the protection domain associated with the root certificate.

4.3 Potential Problems and Mismatches

This section discusses common problems found when signing a MIDlet suite. Most of these problems are created during the signature procedure but may be spotted only at deployment time. Screenshots are included of the same situation in both the Concept SDK emulator and the Nokia 6600.

A commonly encountered problem when installing a signed MIDlet is that its certificate cannot be authenticated. This would happen, for instance, if you would actually sign the suite with a self-signed certificate. It may also happen if the device does not include the certificate of your CA, or you have missed to put all the intermediate certificates in the JAD file, or if the root certificate of your CA is present on the device but disabled for Java authentication.

In that case, there will be an error message as shown in Figure 8.

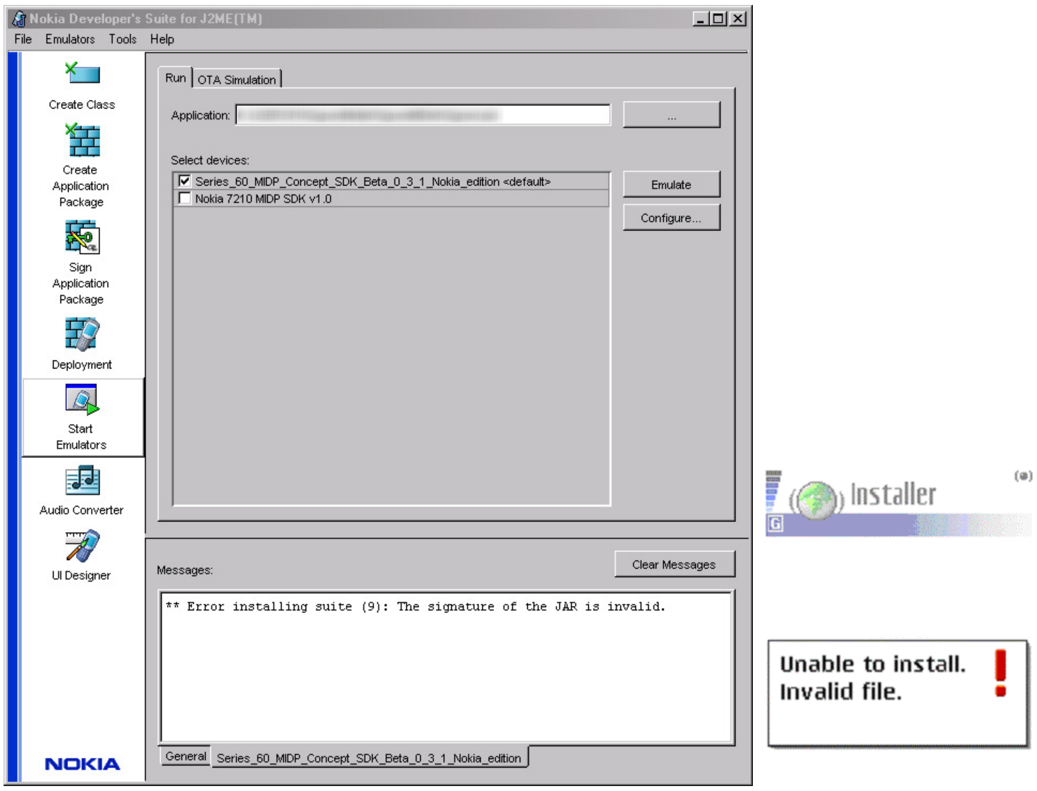


```
Chain not found. Can not a find root CA that matches with signing.
** Error installing suite (6): The content provider certificate issuer
O=Nokia;OU=Forum is unknown.
```

Figure 8: Unknown certificate

Note: With signed applets, the behavior when encountering an unknown certificate depends on the browser and local settings. In the end, the users can accept to trust an applet signed with an unknown certificate if he or she wishes to. In MIDP, this is not allowed. If the device cannot recognize the MIDlet suite, it will not be installed at all. This highlights the need for a certificate issued by a Certificate Authority rather than a self-signed one.

A further problem, which the security framework is especially designed to detect, is the tampering or corruption of a MIDlet JAR file. If for any reason the JAR file has been modified, for example during the OTA transmission or while stored in a device, the integrity check depicted in Section 2.3, "Trusted MIDlets" will fail. This protects the user and the developer of maliciously or accidentally modified MIDlets. This is emulated by manually modifying the manifest file, and therefore making the SHA1 digest incorrect. Figure 9 shows this error in both the emulator and an actual device.

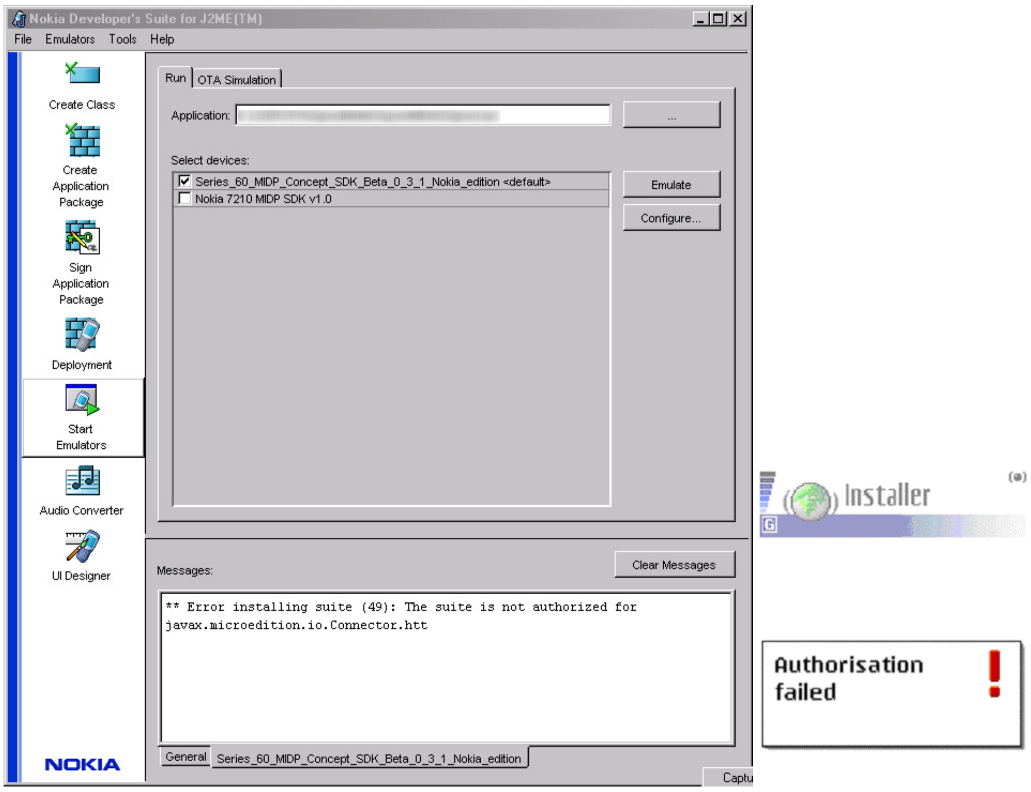


```
** Error installing suite (9): The signature of the JAR is invalid.
```

Unable to install.
Invalid file.

Figure 9: Tampered JAR file (emulator and Nokia 6600 screenshots)

Another potential problem is a mismatch between the permissions requested by the MIDlet suite and the protection domain's permissions. This can be tested in the emulator, for example, by misspelling the requested permission to `javax.microedtion.io.Connector.htt`. In this case, even tough the application certificate is recognized and the MIDlet suite's integrity accepted, the requested permissions cannot be fulfilled, and the installation is aborted, as shown in Figure 10 for the emulator and device.



```
** Error installing suite (49): The suite is not authorized for
javax.microedition.io.Connector.htt
```

Figure 10: Not enough permission (emulator and Nokia 6600 screenshots)

In the case of optional permissions, the MIDlet can be installed even if the domain does not contain those permissions (or they are misspelled).

5 Practical Tips

5.1 Certificate Acquisition

One of the main tasks confronted by developers whishing to sign their MIDlets is how to obtain a certificate suitable for their application. To decide the best approach, it is necessary to determine the target market. If the application is to be distributed via a particular developer program, such as Java Verified™ or programs provided by operators and portals, a good approach is to get a certificate from the program. In many cases the program will sign itself a MIDlet as part of their programs.

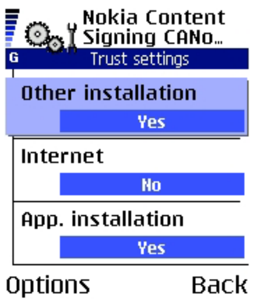
For an application aimed at other channels it is necessary to discover which root certificates are available in the target devices. This is not necessarily straightforward since there are variations on the set of certificates among different device manufacturers, operators, and device models. The best approach is to check the target device for the list of root certificates available and acquire the appropriate one. Entities emitting certificates should also be aware of their availability on different devices.

Once the CA has been selected, it is necessary to purchase the certificate, usually for a fee, for a given period. The CA will normally verify the developer’s identity to ensure that he or she is who he or she has claimed. Once this is done, the certificate is produced and delivered electronically, typically via e-mail or a Web page.

Note that a CA has normally many available certificate types for different purposes. Even for code signing there are several types, depending on the usage. The correct type for MIDlet signing, if not explicitly said, is a code-signing certificate for Java applications.

5.2 Certificates’ Trust Settings

Related to the previous section, it is relevant to note that a single root certificate may be used to sign certificates whose usage goes beyond application signing. In particular if they are used, for example, to connect to secure Web sites using SSL. In Nokia devices root certificates have a trust setting which defines what is allowed for a certificate signed with a given root certificate. This is under the option of trust settings, and defines that a root certificate can be used, for example, to validate MIDlets but not to do SSL transactions. This is, for example, shown in Figure 11 for the Nokia Content Signing certificate. In this case certificates signed with this one can be used for signing MIDlets and other applications but not for Internet communication.


Nokia Content Signing CANo...
Trust settings

Other installation
Yes

Internet
No

App. installation
Yes

Options Back

Figure 11: Root certificate trust settings

In certain cases it is possible that you can obtain a certificate from a CA whose root certificate is present on the device, but the trust setting is not correct. This could happen, for example, if the certificate is valid for installing Symbian applications but not Java applications.

Depending on the device, the trust setting could be manually modified and thus allow installation of MIDlets signed with it. However, this approach is not so intuitive and is therefore difficult to use for mass-market MIDlets.

5.3 Certificate Chains

Depending on the policy of the CA, the certificate provided to the developer may not be directly signed using the CA's root certificate but using one or more intermediate certificates instead. If the intermediate certificate is not included in the JAD file, the installation will be rejected since the device knows only about the root certificate and not about the intermediate ones.

The solution is to include the intermediate certificates in the JAD files and the signing tools should do this automatically. However, care must be taken to ensure that all the certificates are available for the tool to sign. For example, certificates bought from Thawte will come with the certificate chain if the "PKCS#7 Certificate Chain" option is selected when downloading it. Verisign, on the other hand, will always include the intermediate certificates.

It is worthwhile to examine the certificate/certificate chain received from a CA for the content it is going to be used for. Make sure that when importing certificates to the keystore, intermediate certificates are included. Examine the JAD file after MIDlet signing to see if the number of certificate attributes in the JAD file corresponds to the number of certificates in the chain provided by the CA. Also note that some CAs provide complete certificate chains including the root certificate. If this is the case, the root certificate must not be added to JAD.

5.4 Certificate Expiration

During the authentication procedure, the certificate chain is verified by checking that all the certificates are valid and being used correctly. This includes certificate integrity checking, proper usage properties, and validity period. In case any certificate in the chain fails, the MIDlet suite will not be installed.

With respect to the validity period, is important to consider that purchased certificates normally have a restricted period, often one or two years. Only during that time is possible to use the certificate to sing and install MIDlet suites. This implies that suites signed with a certificate cannot be installed after or before this period. It is important to keep this in mind to obtain new certificates when appropriate and sign again the MIDlet suites that need it.

However, any MIDlet suites already installed in a device will continue to function normally after the signing certificate is expired.

5.5 Certificate Revocation

A certificate owner, or the CA who signed it, can revoke certificates due to a number of reasons. In that case the certificate is no longer valid and MIDlets signed with them should not be installed. Nokia devices supporting the Online Certificate Status Protocol (OCSP) will verify the validity of the certificate during the installation.

5.6 MIDlet Updates

One important case to analyze is the MIDlet update scenario. A first consideration is that it is not possible to update a trusted MIDlet with an untrusted one. It is, however, possible to update an untrusted MIDlet with a trusted one and it is also possible to update a trusted MIDlet with another trusted MIDlet.

In case of updating a trusted MIDlet with another trusted MIDlet, and assuming that the protection domain is preserved across the update, the permissions and the permissions mode ought to be preserved. Therefore, if the user has manually given a *session* mode for a particular function group, this setting will be preserved across updates.

6 Release Notes

This tutorial was tested with the Nokia 6600 mobile device using the maintenance release software version 4.09.1 and higher.

Notice also that the signing procedure depicted in the tutorial is meant to work with NDS for J2ME™ 2.1 and it is not guaranteed to work in older versions or in Sun's WTK.

7 Terms and Abbreviations

Term or abbreviation	Meaning
CA	Certificate Authority
Concept SDK	Series 60 MIDP Concept SDK Beta 0.3.1, Nokia Edition
CSR	Certificate Signing Request
J2SE™	Java™ 2 Platform, Standard Edition
MIDP	Mobile Information Device Profile
NDS for J2ME™	Nokia Developer's Suite for J2ME™
OCSP	Online Certificate Status Protocol

8 References

[MIDP 2.0] Mobile Information Device Profile 2.0, Java Community Process, 2002,
http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html

[JTWI] *Java™ Technology for the Wireless Industry*, Java Community Process, 2003,
http://www.jcp.org/en/jsr/detail?id=185

[MIDPPROG] *MIDP 1.0: Introduction to MIDlet Programming*, Forum Nokia, 2002,
http://www.forum.nokia.com

[PKCS] *Public-Key Cryptography Standards*, RSA, http://www.rsasecurity.com/rsalabs/pkcs/index.html

[Java Certificates] *X.509 Certificates and Certificate Revocation Lists (CRLs)*, JavaSoft, 2001
http:java.sun.com/j2se/1.4.1/docs/guide/security/cert3.html

[MIDP 1.0] *Mobile Information Device Profile (MIDP)*, Java Community Process, 2000,
http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html

[CLDC] *J2ME Connected, Limited Device Configuration*, Mobile Information Device Profile 2.0, Java Community Process, 2000, http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html

[X.509] *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, IETF, January 1999,
http://www.ietf.org/rfc/rfc2459.txt

Appendix A Summary of Protected Methods

This appendix lists all methods that require a permission to be invoked. The list is limited to the MIDP-related APIs made public at the time of writing.

A.1 MIDP 2.0

The following packages do not contain methods requiring permissions:

```
java.util
java.lang
java.io
javax.microedition.rms
javax.microedition.midlet
javax.microedition.lcdui
javax.microedition.lcdui.game
javax.microedition.media
javax.microedition.media.control
javax.microedition.pki
```

API call	Permission(s)
`javax.microedition.io.Connector.`	
`open("http://<address>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.http`
`open("https://<address>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.https`
`open("datagram://<address>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.datagram`
`open("datagram://:<port>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.datagramreceiver`
`open("socket://<address>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.socket`
`open("socket://:<port>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.serversocket`
`open("ssl://<address>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.ssl`
`open("comm:<port>",<ANY>,<ANY>)`	`javax.microedition.io.Connector.comm`
`javax.microedition.io.PushRegistry.`	
`registerConnection(connection, midlet, filter)`	`javax.microedition.io.PushRegistry` `+ connection specific permission`
`registerAlarm(midlet, time)`	`javax.microedition.io.PushRegistry`

A.2 Mobile 3D Graphics API (JSR-184)

This API does not contain any method requiring security permissions.

A.3 Bluetooth API (JSR-82) (Excluding OBEX)

API call	Permission(s)
`javax.microedition.io.Connector.`	
`open("btspp://<Server ADDR>…")`	`javax.microedition.io.Connector.bluetooth.client`
`open("btl2cap://<Server ADDR>…")`	`javax.microedition.io.Connector.bluetooth.client`
`open("btspp://localhost:…)`	`javax.microedition.io.Connector.bluetooth.server`
`open("btspp:/localhost:…")`	`javax.microedition.io.Connector.bluetooth.server`

A.4 Wireless Messaging API (JSR-120)

API call	Permission(s)
`javax.microedition.io.Connector.`	

open("sms:// …", WRITE)	javax.microedition.io.Connector.sms javax.wireless.messaging.sms.send
open("sms:// …", WRITE, boolean)	javax.microedition.io.Connector.sms javax.wireless.messaging.sms.send
open("sms:// …", READ)	javax.microedition.io.Connector.sms javax.wireless.messaging.sms.receive
open("sms:// …", READ, boolean)	javax.microedition.io.Connector.sms javax.wireless.messaging.sms.receive
open("sms:// …", READ_WRITE)	javax.microedition.io.Connector.sms javax.wireless.messaging.sms.send javax.wireless.messaging.sms.receive
open("sms:// …", READ_WRITE, boolean)	javax.microedition.io.Connector.sms javax.wireless.messaging.sms.send javax.wireless.messaging.sms.receive
open("cbs:// …", READ)	javax.microedition.io.Connector.cbs javax.wireless.messaging.cbs.receive
open("cbs:// …", READ, boolean)	javax.microedition.io.Connector.cbs javax.wireless.messaging.cbs.receive

A.5 Wireless Messaging API 2.0 (JSR-205)

API call	Permission(s)
javax.microedition.io.Connector.	
open("mms:// …", WRITE)	javax.microedition.io.Connector.mms javax.wireless.messaging.mms.send
open("mms:// …", WRITE, boolean)	javax.microedition.io.Connector.mms javax.wireless.messaging.mms.send
open("mms:// …", READ)	javax.microedition.io.Connector.mms javax.wireless.messaging.mms.receive
open("mms:// …", READ, boolean)	javax.microedition.io.Connector.mms javax.wireless.messaging.mms.receive
open("mms:// …", READ_WRITE)	javax.microedition.io.Connector.mms javax.wireless.messaging.mms.send javax.wireless.messaging.mms.receive
open("mms:// …", READ_WRITE, boolean)	javax.microedition.io.Connector.mms javax.wireless.messaging.mms.send javax.wireless.messaging.mms.receive

A.6 Mobile Media API 1.1 (JSR-135)

API call	Permission(s)
javax.microedition.media.control.RecordControl.	
setRecordLocation(String)	javax.microedition.media.control.RecordControl
setRecordStream(OutputStream)	javax.microedition.media.control.RecordControl
javax.microedition.media.control.VideoControl.	
getSnapshot(String type)	javax.microedition.media.control.VideoControl. getSnapshot

Note: Other methods in the Mobile Media API may be used to play protected content, such as network-stored content. In that case security permissions may also apply.

A.7 Location API (JSR-179)

API call	Permission(s)
javax.microedition.location.LocationProvider.	
getLocation()	javax.microedition.location.Location
setLocationListener()	javax.microedition.location.Location
addProximtyListener()	javax.microedition.location.ProximityListener

`javax.microedition.location.Orientation.`	
`getOrientation()`	`javax.microedition.location.Orientation`
`javax.microedition.location.LandmarkStore.`	
`getInstance()`	`javax.microedition.location.LandmarkStore.read`
`listLandmarkStores()`	`javax.microedition.location.LandmarkStore.read`
`addLandmark()`	`javax.microedition.location.LandmarkStore.write`
`deleteLandmark()`	`javax.microedition.location.LandmarkStore.write`
`removeLandmarkFromCategory()`	`javax.microedition.location.LandmarkStore.write`
`updateLandmark()`	`javax.microedition.location.LandmarkStore.write`
`addCategory()`	`javax.microedition.location.LandmarkStore.category`
`deleteCategory()`	`javax.microedition.location.LandmarkStore.category`
`createLandmarkStore()`	`javax.microedition.location.LandmarkStore.management`
`deleteLandmarkStore()`	`javax.microedition.location.LandmarkStore.management`

A.8 PDA Optional Packages (JSR-75)

A.8.1 File Connection API

API call	Permission(s)
`javax.microedition.io.Connector.`	
`open(“file://…”)`	`javax.microedition.io.Connector.file.read` `javax.microedition.io.Connector.file.write`
`open(“file://…, Connector.READ)`	`javax.microedition.io.Connector.file.read`
`open(“file://…, Connector.WRITE)`	`javax.microedition.io.Connector.file.write`
`open(“file://…”, Connector.READ_WRITE)`	`javax.microedition.io.Connector.file.read` `javax.microedition.io.Connector.file.write`
`openDataInputStream("file://...")`	`javax.microedition.io.Connector.file.read`
`openDataOutputStream("file://...")`	`javax.microedition.io.Connector.file.write`
`openInputStream("file://...")`	`javax.microedition.io.Connector.file.read`
`openOutputStream("file://...")`	`javax.microedition.io.Connector.file.write`
`javax.microedition.io.file.FileConnection.`	
`setFileConnection() // when original was opened in READ MODE`	`javax.microedition.io.Connector.file.read`
`setFileConnection() // when original was opened in WRITE MODE`	`javax.microedition.io.Connector.file.write`
`setFileConnection() // when original was opened in READ_WRITE MODE`	`javax.microedition.io.Connector.file.read` `javax.microedition.io.Connector.file.write`
`availableSize()`	`javax.microedition.io.Connector.file.read`
`usedSize()`	`javax.microedition.io.Connector.file.read`
`directorySize(boolean)`	`javax.microedition.io.Connector.file.read`
`fileSize()`	`javax.microedition.io.Connector.file.read`
`canRead()`	`javax.microedition.io.Connector.file.read`
`canWrite()`	`javax.microedition.io.Connector.file.read`
`isHidden()`	`javax.microedition.io.Connector.file.read`
`setReadable(boolean)`	`javax.microedition.io.Connector.file.write`
`setWritable(boolean)`	`javax.microedition.io.Connector.file.write`
`setHidden(boolean)`	`javax.microedition.io.Connector.file.write`
`list()`	`javax.microedition.io.Connector.file.read`
`list(java.lang.String filter, boolean includeHidden)`	`javax.microedition.io.Connector.file.read`
`create()`	`javax.microedition.io.Connector.file.write`
`mkdir()`	`javax.microedition.io.Connector.file.write`
`exists()`	`javax.microedition.io.Connector.file.read`
`isDirectory()`	`javax.microedition.io.Connector.file.read`
`delete()`	`javax.microedition.io.Connector.file.write`
`rename(java.lang.String)`	`javax.microedition.io.Connector.file.write`
`truncate(long)`	`javax.microedition.io.Connector.file.write`
`lastModifed()`	`javax.microedition.io.Connector.file.read`
`javax.microedition.io.file.FileSystemRegistry.`	
`addFileSystemListener()`	`javax.microedition.io.Connector.file.read`

listRoots()	

Note: Other security restrictions may be imposed on Connector.open(), for example, when accessing critical files.

A.8.2 PIM API

API call	Permission(s)
javax.microedition.pim.PIM.	
openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY)	javax.microedition.pim.ContactList.read
openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE)	javax.microedition.pim.ContactList.read javax.microedition.pim.ContactList.write
openPIMList(PIM.CONTACT_LIST, PIM.WRITE_ONLY)	javax.microedition.pim.ContactList.write
openPIMList(PIM.CONTACT_LIST, PIM.READ_ONLY, java.lang.String)	javax.microedition.pim.ContactList.read
openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE, java.lang.String)	javax.microedition.pim.ContactList.read javax.microedition.pim.ContactList.write
openPIMList(PIM.CONTACT_LIST, PIM.WRITE_ONLY, java.lang.String)	javax.microedition.pim.ContactList.write
listPIMLists(PIM.CONTACT_LIST)	javax.microedition.pim.ContactList.read
openPIMList(PIM.EVENT_LIST, PIM.READ_ONLY)	javax.microedition.pim.EventList.read
openPIMList(PIM.EVENT_LIST, PIM.READ_WRITE)	javax.microedition.pim.EventList.read javax.microedition.pim.EventList.write
openPIMList(PIM.EVENT_LIST, PIM.WRITE_ONLY)	javax.microedition.pim.EventList.write
openPIMList(PIM.EVENT_LIST, PIM.READ_ONLY, java.lang.String)	javax.microedition.pim.EventList.read
openPIMList(PIM.EVENT_LIST, PIM.READ_WRITE, java.lang.String)	javax.microedition.pim.EventList.read javax.microedition.pim.EventList.write
openPIMList(PIM.EVENT_LIST, PIM.WRITE_ONLY, java.lang.String)	javax.microedition.pim.EventList.write
listPIMLists(PIM.EVENT_LIST)	javax.microedition.pim.EventList.read
openPIMList(PIM.TODO_LIST, PIM.READ_ONLY)	javax.microedition.pim.ToDoList.read
openPIMList(PIM.TODO_LIST, PIM.READ_WRITE)	javax.microedition.pim.ToDoList.read javax.microedition.pim.ToDoList.write
openPIMList(PIM.TODO_LIST, PIM.WRITE_ONLY)	javax.microedition.pim.ToDoList.write
openPIMList(PIM.TODO_LIST, PIM.READ_ONLY, java.lang.String)	javax.microedition.pim.ToDoList.read
openPIMList(PIM.TODO_LIST, PIM.READ_WRITE, java.lang.String)	javax.microedition.pim.ToDoList.read javax.microedition.pim.ToDoList.write
openPIMList(PIM.TODO_LIST, PIM.WRITE_ONLY, java.lang.String)	javax.microedition.pim.ToDoList.write
listPIMLists(PIM.TODO_LIST)	javax.microedition.pim.ToDoList.read
javax.microedition.pim.Contact.	
commit()	javax.microedition.pim.ContactList.write
javax.microedition.pim.Event.	
commit()	javax.microedition.pim.EventList.write
javax.microedition.pim.ToDo.	
commit()	javax.microedition.pim.ToDoList.write
javax.microedition.pim.ContactList.	
removeContact(Contact)	javax.microedition.pim.ContactList.write
items()	javax.microedition.pim.ContactList.read
items(PIMitem)	javax.microedition.pim.ContactList.read
items(java.lang.String)	javax.microedition.pim.ContactList.read
itemsByCategory(java.lang.String)	javax.microedition.pim.ContactList.read
addCategory(java.lang.String)	javax.microedition.pim.ContactList.write
deleteCategory(java.lang.String)	javax.microedition.pim.ContactList.write
renameCategory(java.lang.String currentCategory,	javax.microedition.pim.ContactList.write

java.lang.String newCategory)	
javax.microedition.pim.EventList.	
removeEvent(Event)	javax.microedition.pim.EventList.write
items(int searchType, long startDate, long endDate, boolean initialEventOnly)	javax.microedition.pim.EventList.read
items()	javax.microedition.pim.EventList.read
items(PIMitem)	javax.microedition.pim.EventList.read
items(java.lang.String)	javax.microedition.pim.EventList.read
itemsByCategory(java.lang.String)	javax.microedition.pim.EventList.read
addCategory(java.lang.String)	javax.microedition.pim.EventList.write
deleteCategory(java.lang.String)	javax.microedition.pim.EventList.write
renameCategory(java.lang.String currentCategory, java.lang.String newCategory)	javax.microedition.pim.EventList.write
javax.microedition.pim.ToDoList.	
items()	javax.microedition.pim.ToDoList.read
items(PIMitem)	javax.microedition.pim.ToDoList.read
items(java.lang.String)	javax.microedition.pim.ToDoList.read
itemsByCategory(java.lang.String)	javax.microedition.pim.ToDoList.read
addCategory(java.lang.String)	javax.microedition.pim.ToDoList.write
deleteCategory(java.lang.String)	javax.microedition.pim.ToDoList.write
renameCategory(java.lang.String currentCategory, java.lang.String newCategory)	javax.microedition.pim.ToDoList.write

A.9 SIP API (JSR-180)

API call	Permission(s)
javax.microedition.io.Connector.	
open("sip:…")	javax.microedition.io.Connector.sip
open("sip:…", <ANY>)	javax.microedition.io.Connector.sip
open("sip:…", <ANY>, <ANY>)	javax.microedition.io.Connector.sip
open("sips:…")	javax.microedition.io.Connector.sips
open("sips:…", <ANY>)	javax.microedition.io.Connector.sips
open("sips:…", <ANY>, <ANY>)	javax.microedition.io.Connector.sips

A.10 J2ME™ Web Services (JSR-172)

This API does not contain any method that explicilty requires security permissions. However, when invoking a remote interface, permission may be required to open the network connection.

A.11 Security and Trust Services API (JSR-177)

API call	Permission(s)
javax.microedition.io.Connector.	
open("apdu:"[<slot>]";target=SAT")	javax.microedition.apdu.sat
open("apdu:"[<slot>]";target="<AID>)	javax.microedition.apdu.aid
open("jcrmi:"[<slot>]";aid="<AID>)	javax.microedition.jcrmi
javax.microedition.securityservice.CMSMessageSignatureService	
Authenticate (byte[] byteArrayToAuthenticate, ..)	javax.microedition.securityservice. CMSMessageSignatureService

Evaluate This Document

Please spare a moment to help us improve document quality and recognize the documents you find most valuable, by rating this resource.