Developing secure Java Card applications

Jan Vossaert **Laurens Lemaire** **Denis Steckelmacher**
Vincent Naessens

January 9, 2017

1 Introduction

This tutorial covers the basic steps towards secure smart card application development with the *Java Card platform*. We introduce basic development tools for both the card and the host/middleware application. The exercises revolve around a setting with a smart card applet that contains information (i.e. name and serial number) which can be read by a host/middleware application. The applet, however, first requires the user to give his consent by entering his PIN code.

2 Smart card basics

Java Card technology adapts the *Java platform* for use on devices with limited memory and processing power. A common platform on which the *Java Card technology* is widely used is smart cards. Smart cards are often used in environments with strong security requirements. Examples are electronic identity cards, credit cards, membership cards . . . The card performs certain cryptographic operations such as encryptions and digital signatures.

2.1 Smart card communication

Smart cards communicate with host applications through a request-response protocol in which *application protocol data units* (`APDU`'s) are exchanged. Two types of `APDU`'s exist, namely `C`ommand and `R`esponse `APDU`'s. The former are sent by the host application to the card. The latter are sent by the card, as a response to a `C-APDU`, to the host application.

Mandatory header				**Optional body**		
CLA	INS	P1	P2	Lc	Data field	Le

Table 1: The C-APDU structure.

A `C-APDU` consists of a required header and an optional body, as illustrated in Table 1. The `CLA` byte defines an application-specific class of instructions. The `INS` byte defines a specific instruction. The `P1` and `P2` fields can be used to further qualify the instruction or provide input data. The other fields are optional: the `Lc` field defines the number of data bytes in the data field; the `Data` field can contain up to 255 bytes of data; and the `Le` field defines the maximum number of bytes in the data field of the `R-APDU`.

Optional body	**Mandatory trailer**	
Data field	SW1	SW2

Table 2: The R-APDU structure.

An `R-APDU` consists of an optional body and mandatory trailer. The `Data` field contains the response data, maximum 255 bytes, returned by the applet. The fields `SW1` and `SW2` provide feedback about the execution of the `C-APDU`. Several status words are predefined in the ISO7816 standard. The status word `0x9000` represents successful execution of the command.

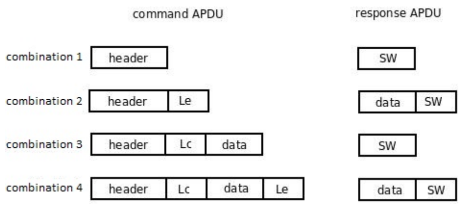


Figure 1: Overview of the possible C/R-APDU combinations.

2.2 Java Card Virtual Machine

Due to the memory and processing constraints of smart cards, the JCVM only supports a subset of the Java programming language. For example, the types `char`, `double`, `float`, `long` are not supported. Support for `int` is optional. Moreover, the Java core API classes and interfaces (`java.io, java.lang` and `java.util`) are not supported except for `Object` and `Throwable`. Further, threads, the security manager and object cloning are not available. For a full overview of the restrictions of the JCVM, we refer to the documentation in the JCDK.

2.3 Java Card Runtime Environment

The *Java Card runtime environment* (`JCRE`) is responsible for card resource management, applet execution, applet security ... It separates applets from the proprietary technologies of smart card vendors and provides `standard system` and `API`[1] interfaces for applets.

The JCRE also ensures that each applet is executed in an isolated environment and can, therefore, not influence other applets on the card. This is realized by the – so called – *applet firewall.* It is enforced by the JCVM during byte code execution.

The JCRE allows for multiple Java Card applets on one smart card. Each applet is uniquely identified by means of an *application identifier* (AID). An AID is a byte array consisting of around 12 bytes. The structure is defined in the ISO7816 standard.

1. More information about the Java Card API that can be used to develop applets can be found on `http://www.win.tue.nl/pinpasjc/docs/apis/jc222/`.

3 Getting Started

3.1 Windows

All files and directories referenced in this tutorial can be found in the *zip* file downloaded from Toledo. Add the plugin for Java Card development to eclipse by copying the *jar* files found in the *plugin* directory to the *dropins* directory of your eclipse installation. Start eclipse and open the workspace contained in the *zip*. Skip to section 3.3.

3.2 Linux

Install gpshell, ant, pcsc-lite and pcsc-ccid from the distribution. Note: on some distributions pcsc-ccid may be named libccid. Unzip the workspace file from Toledo on your system. Add the plugin for Java Card development to eclipse by copying the *jar* files found in the *plugin* directory to the *dropins* directory of your eclipse installation. Download Java Card Kit 2.2.2 for Linux from the Oracle website. Unzip the file somewhere, it contains 4 smaller zip files. Unzip all of them in the directory they are. Start eclipse and open the workspace.

3.3 Configuration

Two active projects should have appeared: *Middleware* and *JavaCard*. The *Middleware* project contains the host application and the *JavaCard* project contains the applet. In case of project errors, try *cleaning* both eclipse projects. In the *JavaCard* project a basic applet is present that can verify PIN codes and release a serial number. The *Middleware* project contains code for communicating with simulated and deployed applets. Also the *selectApplet* and *validatePIN* commands are implemented.

Now configure the plugin by selecting *preferences* from the *JCWDE* menu (see Figure 2) and linking to the Java Card development kit (see Figure 3). Further, similarly modify the *exportpath* in the *CAPGenerationScript.txt* found in the *JavaCard* project.

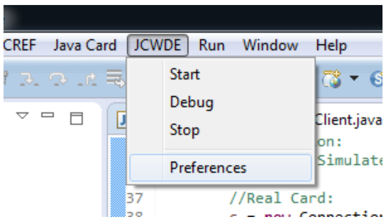
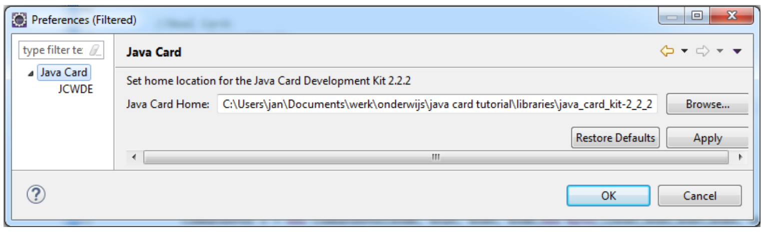


Figure 2: Configuring the Java Card eclipse plugin.



Java Card

Set home location for the Java Card Development Kit 2.2.2

Java Card Home: C:\Users\jan\Documents\werk\onderwijs\java card tutorial\libraries\java_card_kit-2_2_2

Figure 3: Configuring the Java Card eclipse plugin.

4 Testing the demo application

4.1 Starting the simulator

To run applets in simulation the `JCWDE` simulator is used. This simulator also allows execution in debug mode where breakpoints can be set in the code. Note that contrary to a real card application, when the simulator is stopped, the card is entirely reset. All state information is lost. The simulator can be started through the eclipse interface (see Figure 4). Select the first applet an press *OK*. The simulator is now started.

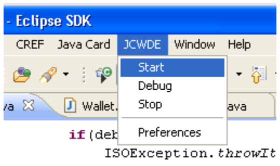


Figure 4: Starting the Java Card simulator.

4.2 Running the host application

Since Java version 1.6, a smart card access framework is integrated by default. This allows Java applications to communicate with applets through APDUs in a user friendly way.

Open *Client.java* in the *be.msec.client* package of the *Middleware* project. Have a look at the code. The program is now configured to work with the simulator. Start the simulator (see Figure 4) and execute the middleware application. Which commands were sent to the applet? Was the PIN validation successful? Successful execution of commands result in response APDUs with status 0x9000h.

4.3 Extending the demo application

Add a second APDU, below the *validatePIN* APDU, that requests the serial number of the card. (Hint: have a look at the applet code to determine what the correct INS, CLA,. . . values are.) Test your implementation by adding support in the middleware application. Is the serial number successfully retrieved? What is the serial number? What happens if you remove the *validatePIN* command and try again?

To personalize the applet, implement a *getName* command in the applet. The name should only be released after successful verification of the PIN. Test your implementation by adding support in the middleware application.

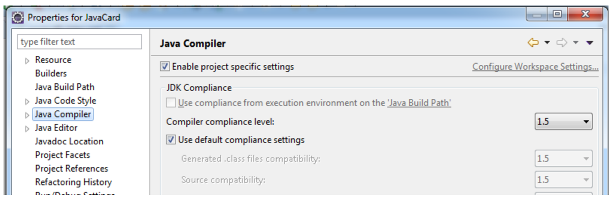
5 Card deployment

The card environment is generally more restrictive than the simulator. Applets might, therefore, not work directly on the card. However, most issues can be resolved by casting *integers* to *shorts* or *bytes*. Also, the simulator does not restrict memory allocation. Applications that require large amounts of memory might, therefore, work in the simulator but fail on the card. Generally, these problems result in the `6F00` status word.

To deploy applets on the card the *GPShell*[2] application is used. The entire process of compiling the Java Card application and deploying it on the card is controlled by the *ant* script (i.e. *projectBuilder.xml*) contained in the *JavaCard* project.

Before the *ant* script can be used, some parameters need to be set. First, set the *JC_HOME* system environment variable to link to the Java Card development kit. Now modify the *cardDir* parameter in the *ant* script to also reference to the Java Card sdk. The Java Card sdk requires that the project is compiled with a Java compiler version 1.5 or lower. Set the JDK compiler compliance level accordingly as shown in Figure 5. Now configure the *ant* script to run in the same JRE as the workspace. Therefore, open the *ant* script in the editor, right click, choose *Run As* and then select *External Tools Configurations...*. Now navigate to the *JRE* tab and then select *Run in the same JRE as the workspace* as shown in Figure 6. Finally, select the *ant* target to be executed by choosing *Run As*, *Ant Build...* and selecting *instantiate* (see Figure 7).

If the card is inserted in the card reader, the applet can now be deployed by executing the *ant* script. Once the applet has been succesfully deployed, reconfigure the host application to send the commands to the card and test the implementation on the card. This requires the commenting and uncommenting of some lines.



☑ Enable project specific settings

JDK Compliance

☐ Use compliance from execution environment on the 'Java Build Path'

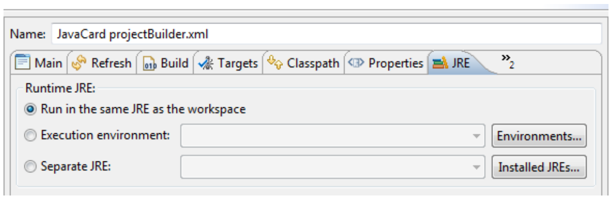
Compiler compliance level: 1.5

☑ Use default compliance settings

Generated .class files compatibility: 1.5

Source compatibility: 1.5

Figure 5: Running Java application in eclipse.



Name: JavaCard projectBuilder.xml

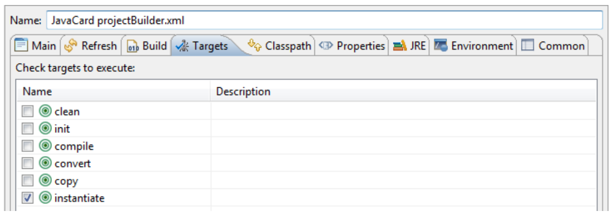
Runtime JRE:

● Run in the same JRE as the workspace

○ Execution environment:

○ Separate JRE:

Figure 6: Configure eclipse to run the ant script.



Name: JavaCard projectBuilder.xml

Check targets to execute:

Name	Description
☐ clean	
☐ init	
☐ compile	
☐ convert	
☐ copy	
☑ instantiate	

Figure 7: Configure eclipse to run the ant script.

2. More information can be found on `http://sourceforge.net/p/globalplatform/wiki/GPShell/`.

6 Additional assignments

Below are two additional assignments. Before starting with the assignments, replace the code of the applet with the code found in the *assignments.txt* file. This code contains the modulus (*privModulus*) and exponent (*privExponent*) of an RSA 512 bits key. It also contains a byte array representation of a digital certificate (*certificate*) with the corresponding public key.

Assignment 1: *Cryptographic operations on the card.*
To verify the authenticity of the card a private key is placed on the card. This key is used to digitally sign challenges sent by the host application. Use the private key to add the signing functionality to the applet. How to build RSAKey objects and perform signing operations is explained respectively in Listing 1 and 2. Also, rewrite *Client.java* to generate a 20 byte challenge and send it to the card. Visualize the challenge and the signature in the host application. Code for generating random byte arrays in Java is given in Listing 3.

```
short offset = 0;
short keySizeInBytes = 64;
short keySizeInBits = 512;
RSAPrivateKey privKey = (RSAPrivateKey)KeyBuilder.buildKey(
  KeyBuilder.TYPE_RSA_PRIVATE, keySizeInBits, false);
privKey.setExponent(privExponent, offset, keySizeInBytes);
privKey.setModulus(privModulus, offset, keySizeInBytes);
```

Listing 1: Building RSA private keys on smart cards

```
Signature signature = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
signature.init(privKey, Signature.MODE_SIGN);
short sigLength = signature.sign(input, offset, length, output, 0);
```

Listing 2: Generation of a digital signature on a smart card

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
byte[] bytes = new byte[20];
random.nextBytes(bytes);
```

Listing 3: Generating random byte arrays of length 20 in Java

Assignment 2: *Transferring large amounts of data.*
To verify that the signature is correct, a digital certificate is also stored on the card. The byte array representation of this certificate (*certificate* in the applet code), however, is larger than 255 bytes. Therefore, write a method in the applet and host application that will retrieve the certificate from that card in blocks of 240 bytes.

Once the certificate is retrieved by the host application, build a certificate object (code given in Listing 4), let the card sign a challenge and the client verify the digital signature (code given in Listing 5).

```
CertificateFactory certFac = CertificateFactory.getInstance("X.509");
InputStream is = new ByteArrayInputStream(byteArrayRep);
X509Certificate cert = (X509Certificate) certFac.generateCertificate(is);
```

Listing 4: Building a certificate object from the byte array representation in Java

```
Signature signature = Signature.getInstance("SHA1withRSA");
signature.initVerify(cert.getPublicKey());
signature.update(challenge);
boolean ok = signature.verify(sig);
```

Listing 5: Verifying digital signatures in Java