# Internet and Smart Card Application Deployment

Xavier Lorphelin

JSource, USA
xavier.lorphelin@jsource.com,
http://www.jsource.com

**Abstract.** This paper describes a new dynamic model for smart card application deployment and addresses two major constraints when deploying applications over the Internet: security and loading time. In this dynamic model, new applications can be downloaded to the card terminal as Java applets signed by a trusted entity. These applications are built on top of smartX framework, a new technology introduced by Gemplus that leverages Java and XML benefits for the dynamic deployment of smart card applications.

## Introduction

### The card application and the terminal application

This paper defines a smart card application in its large scope: the card-resident application and the terminal-resident application. Both application components have to be implemented in parallel. The terminal acts as a server and the card as a client: the terminal sends instructions and data to the card and the card returns data to the terminal. Once received by the card, the instructions sent by the terminal have to be mapped to native instructions that can be executed by the card. The card application processes these terminal instructions and returns the expected data to the terminal. The terminal application manages the sequence of instructions to be sent to the card and correctly interprets the card data.
Deploying a smart card application implies loading and personalizing the card-resident application and the terminal-resident application.

**Static model for smart card application deployment**

In a traditional deployment model, the smart card and the terminal are statically configured for a specific smart card application.
The card-resident application is typically loaded on the smart card during the manufacturing process and then personalized by the smart card issuer. There is one application per card and this is the same application for the card lifetime. The same initialization and personalization processes happen at the terminal level where the terminal-resident application needs to be installed before the deployment of the smart cards. Such terminal will only operate for the smart card applications it has been configured for. Note that a terminal can be loaded with different smart card applications, each application having its own functionality. For example, a terminal located at a hospital may have a purse application and a medical application. The patient has two smart cards: one card operates for the purse application, accepting debit or credit transactions, whereas the second smart card contains medical records for the patient. In this static model, the deployment of a new smart card application involves issuing new cards and reconfiguring the terminal infrastructure.

**Dynamic model for smart card application deployment**

Recent industry initiatives have made this static model obsolete. At the card level, new open platforms like Java Card, MultOS or Smart Card for Windows bring the promises of multi-application: multiple card-resident applications can be loaded on the same card. At the terminal level, the PC/SC and OCF initiatives define new frameworks to develop and deploy terminal applications. All these initiatives converge toward a new dynamic model for smart card application deployment.
In a dynamic model, the smart card and the terminal are multifunctional. This not only means that the card and the terminal can host multiple applications, but that these applications can be dynamically loaded and configured. Once the card is issued, new card-resident applications can be downloaded and personalized. The same applies to the terminal that can be dynamically configured to accept new smart card applications.

**The Internet: a dynamic world**

The Internet is an essential factor to enable the dynamic deployment of smart card applications. From the Web, you can access real-time information anytime, anywhere; you can buy products on-line; you can download new software to your personal computer.
The same logic applies to smart card applications: from the Internet, you will be able to download new card applications and new terminal applications. This model assumes the

terminal to be connected to the Internet, either directly for a stand-alone terminal or indirectly through your personal computer. Smart card applications deployed on the Internet are accessed through your favorite web browser. These smart card applications are dynamically downloaded to the terminal (or your personal computer) and to the card in case of post-issuance loading. In this model, the terminal application is compiled into a Java applet. The features and benefits of Java make it a natural programming language for applications loaded over the Internet. The card application format depends on the card operating system (Java Card, Multos or Windows for Smart Card).

Let's illustrate this model for a typical smart card application: credit/debit. Using your smart card, you can pay for the goods you are buying on-line. Assuming the credit/debit application is initially loaded on the card, you just need to insert your card in the terminal and type your password to authorize the transaction. The terminal downloads a Java applet that encapsulates the functionality to open a connection to the card and execute a debit or credit operation on the card. This process is independent of the terminal as long as the terminal is correctly configured to accept Java applets. In this example, a terminal application is dynamically downloaded to the terminal. The mechanism to download a card application to a smart card is out of the scope of this paper since there is yet no common standard for the card application format.

**Security and loading time**

This paper addresses two main constraints when deploying terminal applications on the Internet: security and loading time.

The security model for the Internet prevents Java applets from accessing local properties and resources (like the COM port to which your smart card reader is connected). The paper reviews the different security models to sign a Java applet: Microsoft Internet Explorer (IE), Netscape Navigator, Sun JDK 1.1, Sun JDK 1.2.

Loading time is a constraint that impacts the size of the applet. This paper introduces a new technology developed by Gemplus: smartX. By separating the application logic from the application process, smartX reduces the Java applet to the core logic and relies on XML dictionaries to encapsulate card-specific protocols. With this technology, an application is also independent of the type of smart card.

## Security model for smart card application deployment

### OpenCard Framework

For the purpose of this paper, terminal applications loaded over the Internet are built on top of OpenCard Framework (OCF). The OpenCard Framework is an open standard that provides interoperability of smart card applications across NCs, POS terminals, desktops, laptops, and set-top boxes (see [1] for OCF official web site). The OpenCard Framework provides developers with an interface for the development of terminal applications in Java. There are two main layers defined in the OCF architecture: the CardTerminal layer and the CardService layer.

The CardTerminal layer provides access to physical terminals and inserted smart cards. It is the responsibility of the terminal manufacturer to implement the CardTerminal layer for a specific terminal. There are two alternatives to physically accessing a terminal: through a PC/SC driver or through a pure Java driver. PC/SC standard defines a comprehensive and flexible solution for integrating smart cards with Windows platforms. The other alternative is a pure Java driver built on the Java Communications API from Sun Microsystems. For example, to access your Gemplus reader GCR410 through OCF, you can either select the generic CardTerminal implementation for PC/SC or the pure Java CardTerminal implementation provided by Gemplus.

The CardService layer provides access to the card application loaded on the smart card. A CardService implementation encapsulates the logic of a smart card application for a specific smart card operating system. CardServices are implemented by smart card manufacturers or by smart card application developers.

### Terminal configuration

The OpenCard Framework API consists in several packages that contain the core classes and the extensions. We assume these packages are initially installed on the terminal (or the personal computer). OpenCard Framework also needs to be correctly configured before being invoked. This configuration phase is critical since it determines which CardTerminal and CardService implementations are to be used. Configuration properties are listed in a property file (`opencard.properties`). The mechanism to load these properties is described in OCF reference implementation. This property file has to be located in either one of the variable paths defined by the following system properties: `user.home`, `user.dir` and `java.home`. Note that these system properties depend on the Java Virtual Machine you selected to run the Java applet. This paper will cover the following Java VMs: Microsoft VM, Netscape VM, Sun Java Runtime Environment 1.1, Sun Java Runtime Environment 1.2.

```
# OCF configuration for GCR410 pure Java driver
# OpenCard.terminals =
com.gemplus.opencard.terminal.GemplusCardTerminalFactory|
    mygcr|GCR410|COM1
# OCF configuration for PC/SC driver
OpenCard.terminals =
com.ibm.opencard.terminal.pcsc10.Pcsc10CardTerminalFactory
```

**Listing 1: CardTerminal configuration**

Part of the configuration requirements, your CLASSPATH variable needs to include the different packages required for your terminal application: OCF core and extensions packages, Java Communication API (if you selected a pure Java CardTerminal) and any other APIs relevant to your application. To configure your CLASSPATH under Windows 95/98, edit your `Autoexec.bat` file located in the main drive of your computer, update the CLASSPATH and reboot your machine. For Windows NT, select "Environment" under "System" in the Control Panel.
Finally, make sure any required DLL (like `Ocfpcsc1.dll` and `Ocfpcscm.dll` for OCF) is in a directory included in your PATH variable.

**Java applet and Internet security**

Java applets have restricted privileges when running on your local machine. They cannot read or write to a file, nor can they access the system properties (see [2] for additional details on the Java security model for applets). As explained in the previous section, OpenCard Framework relies on a property file (`opencard.properties`) to configure its run-time environment. Only signed Java applets will have access to this property file. Terminal applications loaded as Java applets and built on top of OCF therefore have to be initially signed.
Unfortunately for the smart card application developer, there are multiple security models (one for each Virtual Machine). Each security model defines its own mechanism for applet signing. An applet signed with Microsoft `signcode` tool will not execute on Netscape Navigator, and vice versa. Sun signing mechanism for JDK 1.1 and JDK 1.2 is also not supported by any of the native browsers. The next sections will briefly describe the signing process for each model, as well as the configuration required to run the terminal application.
The process to sign an applet can be summarized in three main steps:

- Create a certificate for a trusted entity.

This certificate can either be requested from a Certification Authority (CA) like VeriSign, or auto-generated for test purpose.

- Sign the Java applet packaged as a CAB or JAR file using the certificate.
- Insert the signed applet in an HTML document.

Listings 10-17 at the end of the paper provide the scripts (DOS batch files) that automate the process of creating a certificate and signing an applet for each security model described in this paper. Please feel free to use these scripts for your own terminal application implementation.

**Microsoft security model**

Using the `makecert` and the `cert2spc` tools, you can generate your own certificate for the Microsoft security model. The Java applet is packaged as a CAB file with the `cabarc` tool. Using your SPC certificate (Software Publisher Certificate), you can then sign your applet with `signcode`. When you sign your applet, you can select three permission levels: High, Medium and Low. These permission levels work in conjunction with Internet Explorer zones to determine what an applet can do (see [3] for information on Microsoft security model).
To insert your signed CAB file as an applet in an HTML document, you will need to use the parameter `cabbase` inside the tag `<applet>`.

```
<applet width="120" height="120" code="myApplet.class">
    <param name="cabbase" value="connect.cab">
</applet>
```

**Listing 2: HTML syntax for CAB package**

When this applet is loaded on Microsoft IE, the user will be prompted once for his authorization to run the applet. Note that if the Internet Explorer Zone is set to “Low”, any applet - signed or unsigned - will automatically run on your local computer without giving you a warning message.
With Microsoft IE, the `opencard.properties` file needs to be located in the `java.home` directory: `C:\Windows\Java\lib` or `C:\Winnt\Java\lib`. You might have to create the `lib` folder if it does not already exist.

**Netscape security model**

Netscape Signing Tool version 1.1 allows you to create object-signing certificates for testing purposes (see [4] for information on Netscape security model and object signing resources). Before generating your certificate, you must set a password for the

Communicator Certificate Database: click on the “Security” icon on the toolbar and select “Passwords”. This Communicator password will be used to protect your certificates.
The `signtool` option `-G` generates a new public-private key pair and certificate. To automatically install the newly generated certificate and keys in the Communicator Certificate Database, you can select the `-d` option followed by the Netscape directory that contains the key and certificate databases (respectively `key3.db` and `cert7.db`). These databases are located in `C:\Program Files\Netscape\Users\default`. Note that the auto-generated certificate is output to a file named `x509.cacert`.
When signing the Java applet, you first need to create a temporary directory where you copy the files you want to sign. Then you sign the whole directory using `signtool` and your test certificate. The same operation will package the applet as a JAR file and sign the JAR file.
To insert your signed CAB file as an applet in an HTML document, you will need to use the parameter `archive` inside the tag `<applet>`.

```
<applet width="120" height="120" code="myApplet.class">
    <param name="archive" value="connect.jar">
</applet>
```

**Listing 3: HTML syntax for JAR package**

When this signed applet is loaded on Netscape Navigator, the user is asked to authorize each privilege requested by the applet. With the Microsoft IE security model, the user is prompted only once. There is another major difference between the two security models: with Netscape, the user must first import the certificate used to sign the applet in his own Certificate Database. The HTML page hosting the applet should then have a link to the `x509.cacert` certificate.
To configure OCF for Netscape Navigator, you rename the file `opencard.properties` to `.opencard.properties` (add a dot to the file name) using the DOS command `ren` and you copy the renamed file in the `user.home` directory: `C:\Program Files\Netscape\Users\<user>`.

### JDK 1.1 security model

In the JDK 1.1 security model, a signed applet is given the same privileges as an application running on your local machine. You cannot specify which privilege to grant to the applet as in the Microsoft or Netscape security models.
With the `javakey` tool, you create a new trusted entity in the security database and a key pair (public and private keys) for this entity. You then generate a certificate for this entity using a certificate directive file (a simple text file that contains certificate information). Once

the applet is packaged into a JAR file using the `jar` tool, you can then sign the JAR file with a signing directive text file that specifies the identity of the signer and the certificate to use for this signer (see [5] for a detailed example on how to sign an applet for JDK 1.1).
Since Microsoft or Netscape does not support the JDK 1.1 security model, Sun introduced the Java Plug-in 1.1 (see [7]). This Plug-in allows to bypass the native Virtual Machine of the browser and to specify another Java Runtime Environment. The HTML file that hosts the Java applet must be converted to a correct format to automatically trigger the Java Plug-in. Fortunately, Sun also provided a tool, the HTML converter, that automates this conversion.
One major constraint with this model is that it requires additional configuration steps. The user first needs to install the Java Plug-in on his local machine. Then, through the Java Plug-in Control Panel, he selects the Java Runtime Environment (JRE) that will replace the native Virtual Machine of the browser. Before running the signed applet, the user must also declare the entity that signed the applet as trusted and import the certificate of this entity using the `javakey` tool. When the JRE loads a signed applet, it looks into its security database (`.obj` file) to verify if the entity that signed the applet is trusted. If the entity is trusted, the JRE retrieves the certificate attached to this entity and verifies the signature of the applet. One general error causing this verification to fail is the identity database not being located where the JRE expects it. By default, the JRE looks for the database in the `user.home` directory: `C:\Winnt\Profiles\<username>` for Windows NT platforms. It is however possible to specify another location for the identity database. If you edit the `java.security` file located in `lib/security` folder of the selected JRE, you can specify where the identity database is located by adding the following line:

```
Identity.database = <JRE path>/lib/security/mydb.obj
```
(this is the recommended location for the identity database)

The `opencard.properties` file can either be installed in the `user.home` directory or in the `lib` folder of the selected JRE.
All these extra configuration steps make this model limited for large deployment of smart card applications.

**JDK 1.2 security model**

JDK 1.2 security model differs from the “all” or “nothing” approach of the 1.1 model. Like the Microsoft or Netscape models, JDK 1.2 allows granting privileges in a very fine-grained matter.
The process to create a certificate is also simplified. One `keytool` instruction declares a trusted entity, generates the keys and creates the certificate. The new keystore architecture replaces the identity database that `javakey` created and managed. Note that you can now

protect your keystore and your private keys with passwords. Once the applet is packaged into a JAR file using the `jar` tool, you can then sign the JAR file with the `jarsigner` tool (see [6] for a detailed example on how to sign an applet for JDK 1.2).
Like JDK 1.1, the 1.2 security model is still not supported by your favorite browser and Sun released the Java Plug-in 1.2 and the HTML converter tool for the 1.2 security model. You will therefore need to convert your HTML file that hosts your signed applet.
Unfortunately, configuring your local machine for the JDK 1.2 security model is as complicated as for Java 1.1. Before running the signed applet, the user must first install the Java Plug-in, select the JRE 1.2 and import the certificate of the trusted entity that signed the applet. There is an additional step: the user needs to update the policy file attached with the keystore to specify which privileges are granted to the entity that signed the applet. This `java.policy` file is initially located in the `lib/security` folder of the selected JRE 1.2.

```
// this keystore is to store our certificates
keystore ".keystore";

// gives full privileges to applets signed by XLorphelin
grant signedBy "XLorphelin" {
    permission java.security.AllPermission;
};
```

**Listing 4: Policy file for JDK 1.2 security model**

To verify a signed applet, the JRE first refers to its `java.security` file located in `lib/security`. This file specifies where the `java.policy` file is located. The default is to have a single system-wide policy file in the `lib/security` folder and a policy file in the user's home directory (`C:\Winnt\Profiles\<username>` for Windows NT platforms). This policy file provides the name and location of the keystore as well as determines what actions foreign applets signed by a specific entity are allowed to carry out on the user's local machine. When using JDK 1.2 security model to load a terminal application, the user must carefully install the `java.policy` file and the attached keystore in the expected directory. Note that the `java.policy` file must be rename `.java.policy` (don't forget the dot at the beginning of the file name) if located in the `user.home` directory.
The `opencard.properties` file can either be installed in the `user.home` directory or in the `lib` folder of the selected JRE.

### OCF 1.1.1 support for native browsers

In the Microsoft and Netscape security models, an applet must explicitly request the privileges in its source code. You need to modify your Java code to request the relevant

permission before you exercise this permission in your code. To enable a permission in Microsoft IE, you call the static method `assertPermission()` on the class `com.ms.security.PolicyEngine` to grant a specific permission. In Netscape Navigator, you call the static method `enablePrivilege()` on the class `netscape.security.PrivilegeManager`.
To avoid vendor specific calls in the OCF pure Java code, the latest 1.1.1 release from the OpenCard Consortium introduced a new class: `opencard.core.util.SystemAccess`. Any operation that requires a permission from the native browser is processed through this class. By default, the `SystemAccess` class does not provide any browser functionality. OCF 1.1.1 provides two extensions of this parent class to implement browser-specific security model:

- `opencard.opt.netscape.NetscapeSystemAccess`
- `opencard.opt.ms.MicrosoftSystemAccess`

When running OCF under Microsoft IE or Netscape Navigator, the correct `SystemAccess` implementation needs to be set before the method `SmartCard.start()` is invoked. In Listing 5, this configuration is performed in the `init()` method of the applet for Microsoft IE security model.

```
Public void init() {
../..
opencard.core.util.SystemAccess sys =
    new opencard.opt.ms.MicrosoftSystemAccess();
opencard.core.util.SystemAccess.setSystemAccess(sys);
../..
}
```

**Listing 5: Initialize OCF security model**

## smartX & smart card application

### OCF model for smart card application deployment

We already saw that OCF is a suitable platform to deploy smart card applications over the Internet. OCF open architecture allows the user to select which CardTerminal and CardService implementations are to be used. For a given terminal, there are currently two possible CardTerminal implementations: the generic PC/SC CardTerminal or a pure Java CardTerminal implementation based on the Java Communications API. However, there are many possible CardService implementations, even for the same application (like credit/debit). Remember that a CardService is specific to the card operating system. A

CardService implementation for the Gemplus GPK card would be different than the implementation for the Schlumberger Multiflex card. Open standards like Java Card, MultOS, or Smart Card for Windows will solve this compatibility problems (at least reducing the number of operating systems to less than five), however these standards are far from being widely implemented by the whole smart card industry.
When you deploy your terminal application, you may want to accept as many different cards as possible. For each type of card, you will need to write the corresponding CardService. These CardServices need to be bundled with the Java applet, thus increasing the size of the applet. This will negatively impact the loading time of the applet. Remember that a person surfing the Web does not wait an average more than five seconds for the HTML page to be loaded on his browser! This model also does not clearly define how a new CardService - for a new type of smart card - would be dynamically downloaded to a terminal.

**Description of smartX architecture**

Gemplus introduces a new technology that solves the loading time & interoperability constraints inherent to the OCF model. This technology, smartX, combined with OCF on the terminal-side and open systems on the card-side will accelerate the migration to smart card applications being dynamically deployed on the Internet.
smartX defines a complete framework to develop and deploy smart card applications (see [8] for smartX official web site). In smartX architecture, the application process - the logic of the application - is dissociated from the application protocol. The application process consists in high-level functions (also called processes) executed by the terminal: for example, credit or debit. The application process relies on the application protocol to transform a process into a sequence of card-specific instructions, more commonly called APDUs.
Since it is hardware independent, the application process is only implemented once by the application developer. The application protocol is very similar to the concept of CardService introduced by OCF. However, this application protocol is not implemented in Java like CardService. For the description of application protocols, smartX relies on SML (Smart Markup Language), a markup language based on XML (eXtended Markup Language). The SML document describing an application protocol is called a dictionary. A dictionary contains one or several card profiles (one profile per smart card type) and each profile contains one or several card processes. Each card process is implemented for a given card profile using card-specific commands.
One main advantage of XML is that it is fully integrated with the Internet. The latest versions of the popular web browsers now include the functionality to read and process XML documents. Like HTML, XML is a markup language, but it presents one big advantage compared to HTML: with XML, you can define your own set of tags and your

own syntax. In this perspective, SML is merely an implementation of XML for the smart card industry. For example, SML provides the tag <Apdu> to describe an APDU command. As with HTML, you can have tag attributes and embedded tags inside a tag. Listing 6 details the structure of an APDU command described inside an SML dictionary.

```
<Apdu Id = "SelectFile">
<Apdu Id="selectFile"
<Command>
<Header Class = "0xA0" Ins = "0xA4" P1 = "0"  P2 = "0"
    Lc = "2" Le = "0x20"/>
<In>fileID</In>
</Command>
<Response Status = "NormalEnding" Notify = "DoNotNotify">
<Out>fileInformation</Out>
</Response>
</Apdu>
```

**Listing 6: SML structure of a card instruction (APDU)**

Application protocols described inside SML documents present the same advantages as HTML pages:

- *Downloadable*: SML dictionaries can be accessed and downloaded from the Internet through a regular HTTP connection.
- *Portable*: SML dictionaries are platform-independent.
- *Editable*: you can easily edit and modify an SML document using your preferred text editor.

**smartX model for smart card application deployment**

In this model, we assume OCF and the smartX engine are initially installed and configured on the target terminal. As explained in the previous section, the terminal application consists in two blocks: the application process and the application protocol.

The application process that encapsulates the logic of the application is compiled into a Java applet signed by a trusted entity. The application protocol is described inside an SML dictionary and is card-specific. Once the Java applet is downloaded, the smartX engine identifies the smart card inserted in the terminal. A simple identification consists in verifying the historical bytes of the card ATR (Answer To Reset). After correct identification, the smartX engine dynamically downloads the SML dictionary that contains the application protocol for the card inserted inside the terminal. With this dynamic mechanism, you minimize the loading time since you only download the dictionary relevant

to the card inside the terminal. In the OCF model, you had to download with the applet all the CardService implementations. With smartX, a terminal is also not limited to a predefined set of smart cards. As long as you provide the correct SML dictionary, a terminal can dynamically accept a new smart card that was not originally supported by the application. All these advantages make smartX a platform of choice for developing and deploying smart card applications on the Internet.

**Downloading SML dictionaries**

After identifying the smart card, the smartX engine needs to know the location of the dictionary to be downloaded. You would prefer not to hardcode this location inside the application process to avoid modifying and recompiling your terminal application each time you need to update this location or add a new dictionary. The solution consists in defining the dictionary locations in an external document, the same way OCF defines its properties inside a property file. Rather than using a text file, smartX relies on a specific SML document called the “boot sequence”. This document specifies the dictionary location for each card profile.

```
<Profile Type = "GemClub" Version = "0.1">
<Signature Type = "Card">0x80 0x66 0xA2 0x06 0x02 0x01 0x32
    0x0E</Signature>
<Process Name = "getFile">
<Apdu Id = "returnFilename">
<Command>
<Header Class = "0x00" Ins = "0x00" P1 = "0"  P2 = "0"
    Lc = "0" Le = "0"/>
<In>"http://www.smartxml.com/developer/demo/ClubLoyalty.xml"</In>
</Command>
<Response Status = "NormalEnding" Notify = "DoNotNotify">
<Out>"Void"</Out>
</Response>
</Apdu>
</Process>
</Profile>
```

**Listing 7: Boot sequence document**

In Listing 7, the dictionary for the card GemClub is located on the Internet. This dictionary could also be installed on the local terminal, or it could be stored inside the smart card. Note that the location is specified as a comment inside the `<In>` tag. This is a special SML syntax to declare an instruction that needs to be processed by the application, not by the smart card. The initial location of the boot sequence can be defined inside the application

process code or as a parameter for the applet inside the HTML document. To add support for a new smart card, you just need to update the boot sequence by adding a new profile type with the location of the corresponding SML dictionary.

**Implementing the application process.**

The application process is the core of the terminal application. It is implemented as a Java applet that can be dynamically downloaded to a terminal. This applet has to be signed by a trusted entity since the application process relies on OCF framework to open a communication to the terminal. smartX provides the class `com.gemplus.smartxx.framework.OcfTerminal11` to interface with the OCF layer. Remember to set the correct `SystemAccess` implementation before you create an instance of the `OcfTerminal11` class.
Once the connection to the terminal is opened, the application process can communicate to the smart card through a virtual smart card object. This virtual smart card abstracts the smart card inserted in the terminal. A process called on this virtual smart card is transformed at runtime by the smartX engine into card-specific instructions (or APDUs) defined inside the SML dictionary. smartX provides two different implementations for the virtual smart card: a generic implementation and a proxy implementation.
In the generic implementation, the virtual smart card is declared as an instance of the interface `gemplus.smartx.framework.RunProcess`. You can then invoke any card process like “credit” by calling the method `runProcess()` on this virtual smart card and passing the name of the process as an argument. In listing 8, the argument `processArguments` is an array of Strings that contains the arguments to run the process “credit” (like the amount to be credited, the date of the transaction,…).

```
// create an instance of the terminal
OcfTerminal11 terminal = new OcfTerminal11();
../..
// instantiate the virtual smart card
RunProcess virtualCard = new RunProcessImpl(terminal);
// call a card process
try { responseProcess = virtualCard.runProcess("credit",
    processArguments); }
catch(ProfileException pe) { }
catch(RunProcessException rpe) { }
```

**Listing 8: Generic implementation for the application process**

In the proxy implementation, the virtual card is declared as an instance of the interface `VirtualCard`. This interface and its implementation, `VirtualCardImpl`, act as a proxy to the application protocol: you can directly invoke a card process like “credit” by calling the

method `credit()` on the virtual smart card. Note that the two classes `VirtualCard` and `VirtualCardImpl` have to be generated by the application developer (Gemplus provides a useful tool that automates this step).

```
// instantiate smartX engine
RunProcess processEngine = new RunProcessImpl(terminal);
// instantiate the virtual card
virtualCard = new VirtualCardImpl(processEngine);
// call a card process
try { responseProcess = virtualCard.credit(processArguments); }
catch(ProfileException pe) { }
catch(RunProcessException rpe) { }
```

**Listing 9: Proxy implementation for the application process**

## References


[1] OpenCard Framework Web Site: http://www.opencard.org/

[2] Java Security Model: http://java.sun.com/sfaq/

[3] Microsoft Security Model:
http://www.microsoft.com/java/security/default.htm

[4] Netscape Object Signing Resources:
http://developer.netscape.com/software/signedobj

[5] Signed Applet Example for JDK 1.1: http://java.sun.com/security/signExample/

[6] Signed Applet Example for JDK 1.2:
http://java.sun.com/security/signExample12/

[7] Java Plug-in from Sun Microsystems: http://java.sun.com/products/plugin/

[8] smartX Web Site: http://www.smartxml.com

```
rem *** Script to create a certificate (Microsoft) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the directory where Microsoft signing tools are installed:
set SIGN_DIR=C:\Progra~1\Microsoft\SDK-Java.31\Bin\PackSign
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\microsoft

rem Set Path to include Microsoft signing tools
set PATH=%PATH%;%SIGN_DIR%

rem Create a certificate using makecert
makecert -sk XLKey -n "CN=Xavier Lorphelin" XLCert.cer

rem Turn certificate into a Software Publisher Certificate (SPC)
cert2spc XLCert.cer XLCert.spc

rem Delete environment variables
set SIGN_DIR=
set SECUR_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 10: Generating a certificate for Microsoft security model**

```
rem *** Script to sign a CAB file (Microsoft) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the name of the CAB file:
set CAB_NAME=connect.cab
rem the directory where Microsoft signing tools are installed:
set SIGN_DIR=C:\Progra~1\Microsoft\SDK-Java.31\Bin\PackSign
rem the directory that contains the Java classes to be signed:
set CLASSES_DIR=D:\Data\xavier\gdc99\code\classes
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\microsoft
```

```
rem Set Path to include Microsoft signing tools
set PATH=%PATH%;%SIGN_DIR%

rem Create a CAB file using cabarc
cd %CLASSES_DIR%
cabarc -r -p n %CAB_NAME% *.*

rem Sign CAB file using signcode
move %CAB_NAME% %SECUR_DIR%
cd %SECUR_DIR%
signcode -j javasign.dll -jp low -spc XLCert.spc -k XLKey
    %CAB_NAME%

rem Delete environment variables
set CAB_NAME=
set SIGN_DIR=
set CLASSES_DIR=
set SECUR_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 11: Signing an applet for Microsoft security model**

```
rem *** Script to create a certificate (Netscape) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the directory where Netscape signing tool is installed:
set SIGN_DIR=D:\Data\xavier\gdc99\code\security\netscape
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\netscape
rem the Netscape directory that contains key3.db and cert7.db:
set DB_DIR=C:\Progra~1\Netscape\Users\default

rem Set Path to include Netscape signing tool
set PATH=%PATH%;%SIGN_DIR%

rem Create Test certificate
cd %SECUR_DIR%
signtool -G XLCert -d %DB_DIR%
```

```
rem Delete environment variables
set SIGN_DIR=
set SECUR_DIR=
set DB_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 12: Generating a certificate for Netscape security model**

```
rem *** Script to sign a JAR file (Netscape) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the name of the JAR file:
set JAR_NAME=connect.jar
rem the directory that contains the Java classes to be signed:
set CLASSES_DIR=D:\Data\xavier\gdc99\code\classes
rem the directory where Netscape signing tool is installed:
set SIGN_DIR=D:\Data\xavier\gdc99\code\security\netscape
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\netscape
rem the Netscape directory that contains key3.db and cert7.db:
set DB_DIR=C:\Progra~1\Netscape\Users\default

rem Set Path to include Netscape signing tool
set PATH=%PATH%;%SIGN_DIR%

rem Copy key3.db and cert7.db in the certificate directory
cd %SECUR_DIR%
copy %DB_DIR%\key3.db %SECUR_DIR%
copy %DB_DIR%\cert7.db %SECUR_DIR%

rem Sign JAR file using signtool
md signdir
xcopy %CLASSES_DIR% signdir /s
signtool -k XLCert -Z %JAR_NAME% -d. signdir
rd signdir /s

rem Delete environment variables
set JAR_NAME=
set CLASSES_DIR=
set SIGN_DIR=
```

```
set SECUR_DIR=
set DB_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 13: Signing an applet for Netscape security model**

```
rem *** Script to create a certificate (SUN JDK 1.1) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the directory where SUN signing tools are installed:
set SIGN_DIR=D:\Data\java\jdk1.1.8\bin
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\plugin1.1

rem Set Path to include JDK signing tools
set PATH=%SIGN_DIR%;%PATH%

rem Create identity in security database
cd %SECUR_DIR%
javakey -cs XLorphelin true

rem Create key pair (public & private)
javakey -gk XLorphelin DSA 512 XL_pub XL_priv

rem Generate certificate
javakey -gc XLcertdir.txt

rem Delete environment variables
set SIGN_DIR=
set SECUR_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 14: Generating a certificate for Java 1.1 security model**

```
rem *** Script to sign a JAR file (SUN JDK 1.1) ***
rem You need to customize the certificate-related names
```

```
rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the name of the JAR file:
set JAR_NAME=connect.jar
rem the directory that contains the Java classes to be signed:
set CLASSES_DIR=D:\Data\xavier\gdc99\code\classes
rem the directory where SUN signing tools are installed:
set SIGN_DIR=D:\Data\java\jdk1.1.8\bin
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\plugin1.1

rem Set Path to include JDK signing tools
set PATH=%SIGN_DIR%;%PATH%

rem Create a JAR file
cd %CLASSES_DIR%
jar cvf %JAR_NAME% *.*

rem Sign JAR file using javakey
move %JAR_NAME% %SECUR_DIR%
cd %SECUR_DIR%
javakey -gs XLsigndir.txt %JAR_NAME%
del %JAR_NAME% /q
ren %JAR_NAME%.sig %JAR_NAME%

rem Delete environment variables
set JAR_NAME=
set CLASSES_DIR=
set SIGN_DIR=
set SECUR_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 15: Signing an applet for Java 1.1 security model**

```
rem *** Script to create a certificate (SUN JDK 1.2) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the directory where SUN signing tools are installed:
```

```
set SIGN_DIR=D:\Data\java\jdk1.2.1\bin
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\plugin1.2

rem Set Path to include JDK signing tools
set PATH=%SIGN_DIR%;%PATH%

rem Create identity with new keypair and self-signed certificate
cd %SECUR_DIR%
keytool -genkey -alias XLorphelin -dname "cn=Xavier Lorphelin,
    ou=JSource, o=JSource, c=US" -keypass jsource –storepass
    jsource

rem Export the self-signed certificate
keytool -export -alias XLorphelin -file XLCert.cer

rem Delete environment variables
set SIGN_DIR=
set SECUR_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 16: Generating a certificate for Java 1.2 security model**

```
rem *** Script to sign a JAR file (SUN JDK 1.2) ***
rem You need to customize the certificate-related names

rem Save previous PATH value
set PATH_SAVE=%PATH%

rem Set environment variables
rem the name of the JAR file:
set JAR_NAME=connect.jar
rem the directory that contains the Java classes to be signed:
set CLASSES_DIR=D:\Data\xavier\gdc99\code\classes
rem the directory where SUN signing tools are installed:
set SIGN_DIR=D:\Data\java\jdk1.2.1\bin
rem the directory where to store the certificates:
set SECUR_DIR=D:\Data\xavier\gdc99\code\security\plugin1.2

rem Set Path to include JDK signing tools
set PATH=%SIGN_DIR%;%PATH%

rem Create a JAR file
cd %CLASSES_DIR%
```

```
jar cvf %JAR_NAME% *.*

rem Sign JAR file using jarsigner
move %JAR_NAME% %SECUR_DIR%
cd %SECUR_DIR%
jarsigner -verbose -storepass jsource -keypass jsource
    %JAR_NAME% XLorphelin

rem Delete environment variables
set JAR_NAME=
set CLASSES_DIR=
set SIGN_DIR=
set SECUR_DIR=

rem Restore default PATH
set PATH=%PATH_SAVE%
```

**Listing 17: Signing an applet for Java 1.2 security model**