Serge Chaumette et Jonathan Ouoba
LaBRI, Laboratoire Bordelais de Recherche en Informatique – UMR CNRS 5800,
Université Bordeaux 1 – serge.chaumette@labri.fr – jonathan.ouoba@labri.fr

JAVA CARD (U)SIM ET APPLICATIONS SÉCURISÉES SUR TÉLÉPHONES MOBILES

mots clés : *Java Card / (U)SIM / Java embarqué / téléphone mobile / sécurité*

Les cartes à puce sont aujourd'hui considérées comme des ressources sûres du fait de leur structure interne et des validations qu'elles subissent tout au long de leur cycle de vie. Il est donc possible de les utiliser comme éléments de base pour sécuriser différents types d'environnements : accès physiques à des locaux, transactions bancaires, systèmes embarqués, etc. Dans cet article, nous expliquons comment utiliser les cartes (U)SIM Java des téléphones mobiles afin d'apporter de la sécurité, en termes de confidentialité des données et des communications, dans le fonctionnement des applications développées pour ces matériels. Nous présentons les différentes technologies qui interviennent (Java Card, (U)SIM, Java ME) ainsi que la façon de les combiner. Nous terminons par un exemple d'application qui utilise tous les éléments présentés.

1. Java Card

1.1 Principe

Une carte à puce est une pièce de plastique contenant un circuit électronique intégré et en particulier un processeur, ce qui lui permet donc de manipuler des données. Elle peut être programmée pour réaliser certaines tâches et peut également stocker de l'information.

En fonction de leur OS (*Operating System*), les cartes à puce se partagent en deux grands groupes :

- à système de fichier fixé : la structure de fichiers embarquée (donc les applications) ne peut pas être modifiée et ces cartes offrent donc un jeu de fonctions fixe, défini par le constructeur. C'est le cas de certaines cartes bancaires.
- à système dynamique d'application : on peut ajouter (déployer) des applications sur ces cartes, dynamiquement, de façon contrôlée et sûre. Elles peuvent donc également être mises à jour. C'est le cas de certaines (U)SIM ou des cartes Java. On parle alors de cartes multi-applicatives.

Quel que soit le cas, ces cartes à puce sont considérées comme des éléments sûrs, du fait que l'ajout ou la modification d'un élément de leur structure interne est soit non autorisé (système de fichier fixé), soit encadré par des procédures strictes.

La carte Java est une carte à puce particulière, multi-applicative, qui utilise la technologie Java Card [1]. Il est possible d'y charger et d'y exécuter des applications écrites dans un dérivé du langage Java. De façon pratique, la machine virtuelle, appelée JCVM (*Java Card Virtual Machine*) dans la technologie Java Card, est découpée en deux parties : la première qui contient l'interpréteur de *bytecode* est située sur la carte, et la deuxième qui contient les autres fonctionnalités classiques d'une machine virtuelle (convertisseur, chargement de classes, vérification du bytecode) est située hors de la carte.

1.2 Les applets

Une *applet* [2] est une application écrite dans un dérivé du langage Java qui est chargée et qui s'exécute sur une carte Java. Conformément au découpage de la JCVM décrit ci-dessus, la carte Java contient un JCRE (*Java Card Runtime Environment*). Celui-ci fournit des mécanismes de sécurité assurant entre autres la séparation entre le système de la carte et les applications qui s'y exécutent. Le JCRE gère les ressources de la carte, l'exécution et la sécurité des applets. Les applets interagissent avec le JCRE à travers des API spécifiques. Chaque applet embarquée sur la carte est isolée des autres par un pare-feu (firewall) et dispose de son propre contexte d'exécution. Toute applet et le paquetage (*package*) à laquelle elle appartient sont identifiés par un numéro unique appelé AID (*Application ID*).

Il est important de noter que le langage Java Card utilisé pour développer les applets est un sous-ensemble du langage de programmation Java. Il y a donc certaines caractéristiques de Java qui ne sont pas disponibles en Java Card. Ce sont par exemple les types `char`, `string`, `double` et `float`, les tableaux à plusieurs dimensions, le chargement dynamique des classes et les *threads*. Certaines de ces limitations sont levées avec la version 3 de Java Card, mais nous utilisons ici la version 2 qui est celle essentiellement disponible actuellement.

1.3 Exemple de code commenté

Le code d'une applet respecte la structure d'un code Java classique. La classe principale dérive de la classe `javacard.framework.Applet`.

Le listing 1 présente un code simple qui ne réalise effectivement aucune opération, mais qui contient les éléments principaux que l'on retrouve dans une applet, à savoir :

- **le constructeur**, ici `example_applet` (ligne 16), utilise la méthode `register` qui permet de déclarer ou d'enregistrer l'applet auprès du JCRE de la carte ;
- **la méthode** `install` (ligne 28) provoque la création d'une instance de l'applet par le JCRE ;
- **la méthode** `process` (ligne 58) traite toutes les communications venant du CAD (*Card Acceptance Device*, le lecteur) qui contient la carte ;
- **la méthode** `select` (ligne 38) conduit le JCRE à notifier l'applet du fait qu'elle a été sélectionnée. Le JCRE lui redirigera ensuite les requêtes qui lui sont destinées ;
- **la méthode** `deselect` (ligne 47) notifie l'applet qu'une autre applet va été sélectionnée (rappelons que sur les cartes multi-applicatives plusieurs applets peuvent être chargées).

Le lecteur souhaitant effectivement exécuter un code d'applet pourra se référer à la section 4.4.

```
1.  package demo_applet ;
2.
3.  // paquetages importés
4.  // importation spécifique pour l'API Javacard
5.  import javacard . framework .*;
6.
7.  public class example_applet extends javacard . framework
8.  Applet {
9.
10. /**
11. Constructeur par défaut
12. @param baBuffer paramètres d'installation
13. @param sOffset offset de départ
14. @param bLength longueur en octets
15. */
16. protected example_applet ( byte [] baBuffer , short sOffset , byte
17. bLength ) {
18.       register ( baBuffer , ( short ) ( sOffset + 1), ( byte )
19.       baBuffer [ sOffset ]);
20. }
21.
22. /**
23. Methode pour installer l'applet.
24. @param baBuffer paramètres d'installation
25. @param sOffset offset de départ
26. @param bLength longueur en octets
27. */
28. public static void install ( byte [] baBuffer , short sOffset , byte
29. bLength ) {
30.       // création d'une instance de l'applet
31.       new example_applet ( baBuffer , sOffset ,bLength ).
32.       register ();
33. }
34.
35. /**
36. @le booléen retourné doit toujours être true .
37. */
38. public boolean select () {
39.       /** @todo : ACTION DE SELECTION A EXECUTER */
40.       // retourne l'état de la désélection
41.       return true ;
42. }
43.
44. /**
45. Invoquée par le système dans le processus de désélection.
46. */
47. public void deselect () {
48.       /** @todo : ACTION DE DESELECTION A EXECUTER*/
49. }
50.
51. /**
52. Méthode qui traite une APDU entrante
53. @voir APDU
54. @param apdu APDU entrante
55. @exception ISOException avec les octets de réponse
56. defini dans l'ISO 7816-4
57. */
58. public void process ( APDU apdu ) {
59.       /** @todo : METTRE L'ACTION A EXECUTER ICI */
60. }
61.
62. }
```

Listing 1 : Code applet

2. Java Card sur (U)SIM

2.1 Présentation

L'(U)SIM est une application spécifique [3] installée sur une carte à puce amovible (généralement insérée dans un téléphone mobile) et qui permet de se connecter à un réseau GSM/3G. Elle fournit aussi des mécanismes d'authentification et de stockage des profils des utilisateurs.

Par extension, on appelle carte Java (U)SIM une carte Java qui embarque une (U)SIM. En d'autres termes, c'est une carte à puce sur laquelle on peut naturellement charger et exécuter des applets Java Card, être connecté à un réseau GSM/3G et enfin avoir accès à certaines informations du profil utilisateur.

2.2 Spécificités

L'association des technologies Java Card et (U)SIM dans une carte permet d'avoir accès dans les applets embarquées, grâce à un jeu d'API spécifiques, à certaines fonctionnalités liées aux (U)SIM. On peut par exemple accéder au système de fichiers (fichier des contacts ou fichier des SMS), manipuler le code PIN, etc.

De façon générale, les cartes à puce ont un caractère passif c'est-à-dire qu'elles fonctionnent en mode client serveur, répondant aux requêtes provenant du CAD (*Card Acceptance Device*, le téléphone mobile dans notre cas) dans lequel elles sont insérées. Il existe des cartes dites « actives » qui envoient des commandes à exécuter au téléphone mobile. Ces commandes sont dites « proactives ». Il est possible avec ces commandes d'afficher des textes sur l'écran du téléphone, de lancer l'émission d'un appel, d'expédier des SMS, etc.

Tous ces éléments se retrouvent dans des applets Java Card spécifiques définies autour de l'(U)SAT. L'(U)SAT (*(U)SIM Application Toolkit*, standard 3GPP 31.111) permet donc à une applet d'utiliser la proactivité, et les API d'accès aux fonctionnalités des cartes (U)SIM.

3. Java embarqué sur téléphone mobile

3.1 Principe

Java embarqué est construit autour de la plate-forme Java ME (*Java Micro Edition*) qui est une technologie permettant de développer des applications pour les appareils de petite taille et donc à ressources réduites (PDA, téléphone mobile, etc.).

Plus précisément, Java ME [4] est un ensemble de technologies et de spécifications offrant un environnement d'exécution Java pour petits appareils. Les appareils cibles sont des équipements ayant des ressources limitées en matière de mémoire, de puissance de traitement, de capacité d'affichage et d'autonomie en énergie. Java ME est basé sur trois éléments :

- **une configuration** qui fournit les éléments de base et une machine virtuelle compatible avec un large ensemble d'équipements. Java ME possède deux configurations à savoir la CLDC (*Connected Limited Device Configuration*) pour les téléphones mobiles et la CDC (*Connected Device Configuration*) pour les PDA.
- **un profil** qui est lié à une configuration et qui propose des API ciblant une gamme éventuellement plus limitée d'équipements. On trouve par exemple la MIDP qui est un profil possible pour la configuration CLDC et la *Foundation Profile* pour la CDC. Ce sont les profils les plus répandus
- **des paquetages optionnels** qui sont un ensemble d'API très spécifiques et que l'on ne retrouve pas nécessairement dans toutes les configurations.

Comme indiquée ci-dessus, la configuration CLDC est spécialement conçue pour les appareils tels que les téléphones mobiles et elle est associée à la MIDP (*Mobile Information Device Profile*) qui comprend un ensemble d'API permettant de développer les applications Java ME. Nous utilisons la version 1.0 de CLDC et la version 2.0 de MIDP.

3.2 Les midlets

Une *midlet* est une application développée en utilisant la plateforme Java ME et qui s'exécute sur des appareils de petite taille, comme les téléphones portables. Un fichier JAD (*Java Application Descriptor*) peut également être associé à une midlet ; il fournit des informations complémentaires concernant l'application (signature de la midlet par un certificat, API particulières utilisées, autorisations données à la midlet, etc.).

Une midlet est censée respecter le principe du « *write once, run everywhere* ». Cela signifie que toute midlet développée en respectant les spécifications d'une configuration et d'un profil donnés doit pouvoir s'exécuter sur tout équipement mobile compatible avec cette configuration et ce profil. Les midlets peuvent être téléchargées et installées directement par les utilisateurs de téléphones mobiles compatibles avec Java ME à partir de sites web ou de serveurs dédiés.

3.3 Exemple de code commenté

Le code d'une midlet reprend la structure d'un code Java classique. La classe principale dérive de la classe `javax.microedition.Midlet`.

Le Listing 2 présente le code d'un « *hello world* » qui contient tous les éléments essentiels que l'on retrouve dans une midlet :

- **le constructeur**, ici `example_midlet` (ligne 13), contient les éléments d'initialisation de la méthode d'affichage (invocation de `Display.getDisplay(this)`).
- **la méthode** `startApp` (ligne 25) est invoquée au lancement de l'application.
- **la méthode** `pauseApp` (ligne 22) est invoquée lors de la mise en pause de l'application (par la méthode `NotifyPaused()`).
- **la méthode** `destroyApp` (ligne 19) est invoquée lorsque l'application se termine, le paramètre `unconditional` de cette méthode précise si les ressources allouées à la midlet doivent être libérées lors de sa terminaison.

On notera également la présence de deux autres méthodes : `showScreen` (ligne 32) qui permet de porter à l'écran les éléments d'affichage construits par ailleurs et `display_alert` (ligne 37) qui est utilisée pour visualiser le message « Hello world ! »

```
1.  package demo_midlet ;
2.
3.  import javax . microedition . midlet . MIDlet ;
4.  import javax . microedition . midlet .
5.  MIDletStateChangeException ;
6.  import javax . microedition . lcdui .*;
7.  import java . util .*;
8.
9.  public class example_midlet extends MIDlet {
10. private Display display ;
11. private Form form ;
12.
13. public example_midlet () {
14.       display = Display . getDisplay ( this );
15.       // création du "form" principal avec ses composants
16.       form = new Form (" Midlet Example ");
17. }
18.
19. protected void destroyApp ( boolean unconditional ) {
20. }
21.
22. protected void pauseApp () {
23. }
24.
25. protected void startApp () {
26.       showScreen ();
27.       display_alert ( AlertType .INFO ," INFO ", "Hello world !",
28.       5000);
29. }
30. ç
31. // affichage de l'écran principal
32. public void showScreen () {
33.       display . setCurrent ( form );
34. }
35. ç
36. // affichage d'un message d'alerte pendant timeout millisecondes
37. public void display_alert ( AlertType type, String title , String text ,
38. int timeout ){
39. Alert alert = new Alert (title , text , null, type);
40. alert . setTimeout ( timeout );
41. display . setCurrent ( alert );
42. }
```

Listing 2 : Code midlet

4. Communication mobile/carte (U)SIM

4.1 Principe

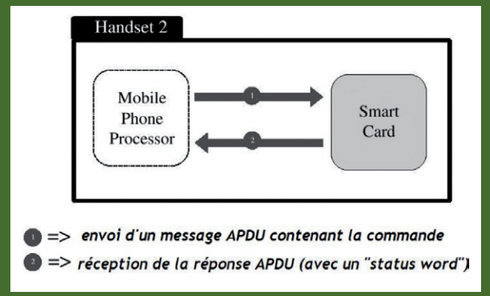
Rappelons que notre objectif est d'expliquer comment utiliser des cartes (U)SIM pour éventuellement sécuriser des applications sur téléphone mobile. Il faut donc comprendre les spécifications à exploiter pour faire communiquer ces deux matériels. Le principe de base est le même que pour une communication entre une carte à puce classique et un CAD, comme cela est défini par la norme ISO 7816-4.

Le protocole d'échange avec une carte à puce se fait en mode série et *half-duplex* de la façon suivante. La carte reçoit des commandes à exécuter via des APDU (*Application Processing Data Unit*). Une APDU est une suite d'octets respectant une norme précise formalisant la communication CAD/carte à puce. Elle comporte les champs suivants:

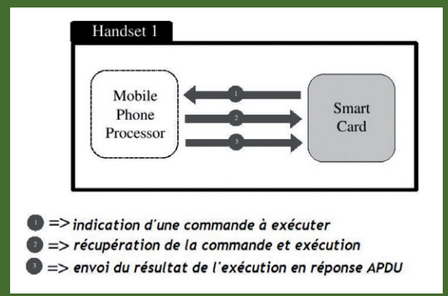
- `CLA`, octet de classe, généralement spécifique à l'application dans laquelle elle est utilisée ;
- `INS`, octet d'instruction, définit la commande à exécuter ;
- `P1`, paramètre 1, sur 1 octet, lié à l'instruction ;
- `P2`, paramètre 2, sur 1 octet, lié à l'instruction ;
- `Lc`, sur 1 octet, taille des données ;
- `Data`, 0-255 octets, données.
- `Le`, définit la longueur des données attendues dans la réponse à cette commande

Dans la figure 1 (page suivante), c'est la machine auquel le CAD est connecté (on dira simplement le CAD) qui initie la communication APDU. La commande est expédiée par le CAD, puis exécutée par la carte qui retourne un code (le *status word*) composé de deux octets (qui valent 90 et 00 en cas de succès de l'exécution de la requête).

Dans le cas d'une commande proactive, c'est-à-dire lorsqu'une carte (U)SIM initie une opération (comme vu précédemment), le schéma des échanges APDU est alors différent (Figure 2).



1 *Schéma classique d'envoi d'une commande APDU*



2 *Schéma d'envoi d'une commande proactive*

La carte indique au CAD (en réalité le CAD fait du polling sur la carte) qu'il y a une commande à exécuter, ce dernier récupère l'information et exécute l'action demandée. Le résultat de l'exécution est ensuite retourné à la carte.

4.2 Communication midlet/applet

Dans la configuration qui nous intéresse ici, le téléphone mobile embarque une midlet et la carte (U)SIM une applet. Il s'agit donc de réaliser une communication entre une midlet et une applet. Ces échanges utilisent le protocole de communication CAD/carte à puce classique défini plus haut.

Dans la mise en œuvre et de façon pratique, la midlet se charge d'ouvrir le canal de communication avec la carte en spécifiant un certain nombre de paramètres (AID de l'applet cible, canal logique à utiliser). L'échange de données se fait ensuite de façon classique (schéma d'échanges APDU) en utilisant les API réservées à cet effet.

4.3 API à utiliser

Pour l'échange de commandes APDU avec la carte à puce en Java ME, on utilise la classe **javax.microedition.Apdu** qui fait partie des paquetages définis dans la JSR-177. Il est également important de noter que toute midlet utilisant les fonctionnalités de la JSR-177 en matière de communication APDU doit être signée par un certificat d'un organisme de certification « connu du téléphone » (Thawte ou Verisign par exemple).

Au niveau de l'applet, il n'y a pas d'API spécifique; la méthode `process` reçoit les commandes APDU venant du CAD de manière classique, lance le traitement approprié et renvoie la réponse APDU.

4.4 Exemple de code commenté

Dans cet exemple, nous mettons en œuvre une petite application qui, au niveau de la midlet, provoque le lancement par l'applet d'une commande proactive, en l'occurrence l'affichage d'un message sur l'écran du téléphone mobile.

Le listing 3 présente le code de la midlet correspondante qui utilise les API de la JSR-177 pour communiquer avec l'applet associée. La structure de la midlet reste classique avec les méthodes de base. Les nouveaux éléments au niveau de la classe principale `example_midlet_apdu` (ligne 11) sont le tableau d'octets `tempBuffer` (ligne 19) qui contient la commande APDU à exécuter, ainsi que la méthode `thread_apdu` (ligne 55) qui a pour rôle de lancer le thread (classe `Send_APDU`) en charge de gérer la communication avec l'applet.

La classe `Send_APDU` (ligne 64) encapsule les appels à l'API nécessaires à la communication midlet/applet. L'objet principal en est `javax.microedition.apdu.APDUConnection` (ligne 5). Il est tout d'abord utilisé dans la méthode `openAPDUConnection` (ligne 76) pour ouvrir la connexion apdu. Les paramètres de cette méthode sont le type de connexion (apdu dans notre cas) et la cible (dans notre exemple le *target* renvoie à l'AID de l'applet avec laquelle on veut communiquer). L'objet `APDUConnection` est utilisé ultérieurement dans la méthode `closeAPDUConnection` (ligne 88) chargée de fermer la connexion APDU lorsque tous les traitements sont terminés. Enfin, la méthode `sendAPDU` (ligne 98) reçoit en paramètre le tableau d'octets de la commande APDU à exécuter. Elle envoie la commande en invoquant `exchangeAPDU` (ligne 102) et retourne le résultat sous la forme d'un tableau d'octets. La méthode `run` (ligne 116) qui est exécutée au lancement du thread ouvre la connexion APDU, envoie la commande, récupère le résultat, puis ferme la connexion.

Le listing 4 présente le code de l'applet associée. Ce code reste classique. La méthode `process` (ligne 83) qui est exécutée à la réception d'un message APDU, lance la procédure qui conduira finalement à l'exécution de la commande proactive d'affichage d'un message à l'écran du téléphone mobile (message contenu dans le tableau d'octets `messageToDisplay`, ligne 19). L'invocation de `reg.setEvent` (ligne 89) dans la méthode `process` provoque à son tour l'invocation par le *runtime* de la méthode `processToolkit` en réaction à certains événements bien précis. Dans cet exemple, `processToolkit` (ligne 71) lance la méthode `displayText` (ligne 106) lors de la survenue de l'événement `EVENT_PROACTIVE_HANDLER_AVAILABLE` (lignes 72 et 74). Cet événement intervient lorsque le canal de communication entre l'applet et la midlet est libre pour envoyer une commande proactive.

`displayText` (ligne 106) récupère au niveau du système les informations concernant le *handler* (`ProactiveHandler`, ligne 107) de gestion des commandes proactives et invoque la méthode `displayMessage` (ligne 93) chargée de l'envoi du texte à afficher au téléphone mobile. `displayMessage` formate le message avec `initDisplayText` (ligne 97) et l'envoie avec `send` (ligne 99).

Pour tester le fonctionnement de ces codes, il est possible d'utiliser les outils de développement présentés dans la section 5.

```
1.  package demo_midlet_apdu ;
2.
3.  import javax . microedition . midlet . MIDlet ;
4.  import javax . microedition . midlet .
    MIDletStateChangeException ;
5.  import javax . microedition . apdu . APDUConnection ;
6.  import javax . microedition .io. Connector ;
7.  import javax . microedition . lcdui .*;
8.  import java .io .*;
9.  import java . util .*;
10.
11. public class example_midlet_apdu extends MIDlet {
12.
13. protected APDUConnection apduConnectionRef = null ;
14.
15. private Display display ;
16. private Form form ;
17. private Thread tAPDU ;
18. private Send_APDU sAPDU ;
19. private byte [] tempBuffer = {
20. ( byte )0 xA0 , ( byte )0 xC4 ,
21. ( byte )0 x00 , ( byte )0 x00 ,
22. ( byte )0 x00 };
23.
24. public example_midlet_apdu () {
25.       display = Display . getDisplay ( this );
26.       // creation du "form" principal
27.       form = new Form (" Midlet APDU Example ");
28. }
29.
30. // affichage de l'écran principal
31. public void showScreen () {
32. display . setCurrent ( form );
33. }
34.
35. protected void destroyApp ( boolean unconditional ) {
36. }
37.
38. protected void pauseApp () {
39. }
40.
41. protected void startApp () {
42. showScreen ();
43. thread_apdu ( tempBuffer );
44. }
45.
46. // affichage d' un message d'alerte
47. public void display_alert ( AlertType type ,String title ,
    String text , int
48. timeout ){
49. Alert alert = new Alert (title , text ,null , type );
50. alert . setTimeout ( timeout );
51. display . setCurrent ( alert );
52. }
53.
54. // lancement du thread de gestion des commandes apdu
55. private void thread_apdu ( byte [] apdu_send ) {
56. sAPDU = new Send_APDU (this , apdu_send );
57. tAPDU = new Thread ( sAPDU );
58. tAPDU . start ();
59. }
60.
61. }
62.
63. // classe d'envoi et de réception de commande apdu
64. class Send_APDU implements Runnable {
65.
66. private demo_midlet parent ;
67. private APDUConnection apduConnectionRef = null ;
68. private byte [] data = null ;
69. private byte [] response = null ;
70.
71. Send_APDU ( example_midlet_apdu parent ,byte [] data ) {
72. this . parent = parent ;
73. this . data = data ;
74. }
75.
76. private void openAPDUConnection () {
77. try {
78. // ouverture de la connexion SATSA APDU
79. apduConnectionRef =( APDUConnection ) Connector. open ("
80. apdu :0;target =A0
81. .00.00.00.18.50.00.00.00.00.00.00.00.52.41.44.41 ");
82. } catch ( Exception ex) {
83. parent . display_alert ( AlertType .INFO ," INFO "," Error
    opening
84. APDU ...  " ,2000);
85. }
86. }
87.
88. private void closeAPDUConnection () {
89. try {
90. // fermeture de la connexion SATSA APDU
91. apduConnectionRef . close ();
92. } catch ( Exception ex) {
93. parent . display_alert ( AlertType .INFO ," INFO "," Error
    closing
94. APDU ... " ,2000);
95. }
96. }
97.
98. private byte [] sendAPDU ( byte [] messApdu ) {
99. byte [] result = null ;
100. try {
101. // Envoi de la commande APDU
102. result =apduConnectionRef . exchangeAPDU (messApdu );
103. parent . display_alert ( AlertType .INFO ," INFO "," APDU
104. command sent ... " ,2000);
105. } catch ( IOException ioErr ) {
106. parent . display_alert ( AlertType .INFO ," INFO "," Can 't
     send
```

```
107.  APDU ... " ,2000);
108.  }
109.  catch ( IllegalArgumentException argErr ) {
110.  parent . display_alert ( AlertType .INFO ," INFO "," Illegal APDU
111.  ..." ,2000);
112.  }
113.  return result ;
114.  }
115.
116.  public void run () {
117.  openAPDUConnection ();
118.  response = sendAPDU ( data );
119.  closeAPDUConnection ();
120.  }
121.
122.  }
```

Listing 3 : Code midlet

```
1.    package demo_applet_apdu ;
2.
3.    /*
4.    Imported packages
5.    */
6.    import uicc . toolkit .*;
7.    import uicc . usim . access . USIMConstants ;
8.    import uicc . access .*;
9.    import javacard . framework .*;
10.   import javacard . security .*;
11.   import javacardx . crypto .*;
12.
13.   public class example_applet_apdu extends javacard . framework .
14.   Applet implements ToolkitInterface , uicc . toolkit .
      ToolkitConstants {
15.   // variables obligatoires
16.   private ToolkitRegistry reg ;
17.   final byte DCS_8_BIT_DATA = 0 x04 ;
18.   private FileView uiccView ;
19.   private byte [] messageToDisplay = {
20.   ( byte ) 0x61 , ( byte ) 0x62 ,
21.   ( byte ) 0x63 , ( byte ) 0x64 ,
22.   ( byte ) 0x31 , ( byte ) 0x32 ,
23.   ( byte ) 0x33 , ( byte ) 0 x34 };
24.
25.   /**
26.   Constructeur de l'applet
27.   */
28.   public example_applet_apdu ( byte [] bArray ,short bOffset ,
      byte
29.   bLength ) {
30.           // Enregistrement de l'applet
31.           register ( bArray , ( short ) ( bOffset + 1),( byte )
      bArray [ bOffset
32.           ]);
33.           // Référence de l'applet
34.           // avec l'objet ToolkitRegistry
35.           reg = ToolkitRegistrySystem . getEntry ();
36.           tempBuffer =JCSystem . makeTransientByteArray (( short )
      155
37.           , JCSystem . CLEAR_ON_RESET );
38.           // référence de l'objet File View
39.           uiccView = UICCSystem . getTheUICCView (JCSystem .
40.           CLEAR_ON_RESET );
41.   }
42.
43.   /**
44.   Methode d'installation invoquée par le JCRE à l'installation
45.   @param bArray tableau d'octet pour l'AID
46.   @param bOffset début de l'AID
47.   @param bLength longueur de l'AID
```

```
48.   */
49.   public static void install ( byte [] bArray ,short bOffset , byte
      bLength ) {
50.           // création de l'applet Java SIM toolkit
51.           applet_test StkCommandsExampleApplet = new
52.           example_applet_apdu ( bArray ,bOffset , bLength );
53.   }
54.
55.   /**
56.   Methode invoquée par la plateforme GSM
57.   */
58.   public Shareable getShareableInterfaceObject (AID clientAID ,
      byte
59.   parameter ) {
60.           if (( parameter == ( byte ) 0 x01 )&& ( clientAID ==
      null )) {
61.           return (( Shareable ) this );
62.           }
63.           return null ;
64.   }
65.
66.   /**
67.   Method invoquée par la plateforme SIM tollkit
68.   @param event représentation en octets
69.   de l'événement déclenché
70.   */
71.   public void processToolkit ( short event ) {
72.           if ( event ==EVENT_PROACTIVE_HANDLER_AVAILABLE ){
73.           reg . clearEvent
74.           (EVENT_PROACTIVE_HANDLER_AVAILABLE );
75.           displayText ();
76.           }
77.   }
78.
79.   /**
80.   Methode invoquée par le JCRE
81.   @param apdu objet APDU entrant
82.   */
83.   public void process ( APDU apdu ) {
84.           // ignore la commande
85.           // de sélection de l'applet
86.           if ( selectingApplet ()) {
87.           return ;
88.           }
89.           reg . setEvent (EVENT_PROACTIVE_HANDLER_AVAILABLE);
90.
91.   }
92.
93.   public void displayMessage ( ProactiveHandlerproHdlr , byte []
94.   messageBuffer , boolean flag ) {
95.           // reconstruction du proactive handler
96.           //et affichage du message défini
97.           proHdlr . initDisplayText (( byte ) 0,DCS_8_BIT_DATA ,
98.           messageBuffer ,( short ) 0, ( short ) ( messageBuffer .
      length ));
99.           proHdlr . send ();
100.    // arrêt de l'exécution de l'applet
101.    if ( flag ) {
102.          ToolkitException . throwIt (( short ) 0 x0000 );
103.    }
104.  }
105.
106.  private void displayText () {
107.    ProactiveHandler proHdlr =
108.    ProactiveHandlerSystem . getTheHandler ();
109.    displayMessage ( proHdlr ,messageToDisplay , false );
110.  }
111.
112.  }
```

Listing 4 : Code applet

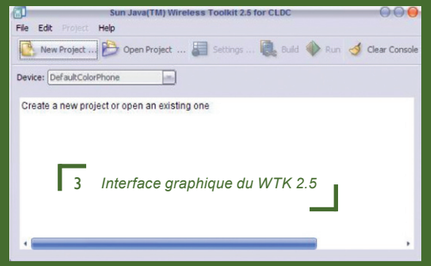
5. Outils de développement

5.1 Sun Wireless Toolkit

Le Sun WTK (*Sun Wireless Toolkit*) est un ensemble d'outils permettant de développer des applications Java compatibles avec les appareils (téléphones mobiles généralement) supportant la plateforme Java ME. Cette suite logicielle permet donc de compiler et de vérifier (au sens du vérificateur de bytecode) les midlets créées, mais il ne propose pas d'éditeur pour saisir le code des applications. Il est à noter qu'il est cependant possible de l'intégrer dans des IDE (*Integrated Development Environment*), tels que Eclipse et Netbeans. Le système de gestion de projets permet également de définir les spécificités des midlets développées en termes de version de CLDC et de MIDP, d'API optionnelles à utiliser, d'autorisations, etc.

On notera aussi que le WTK possède un environnement de simulation assez complet et qui permet de tester des éléments spécifiques comme l'émulation OTA (*over-the-air*) et l'émulation du *push registry* (pour le lancement automatique de la midlet lors de la survenue d'un événement spécifique, la réception d'un SMS par exemple). La signature des midlets est également supportée grâce à un système de gestion des certificats émis par des autorités de certification.

À la fin du processus de développement, le WTK produit le fichier JAR (Java ARchive) et le fichier JAD qui doivent être installés sur le téléphone mobile pour faire fonctionner la midlet.

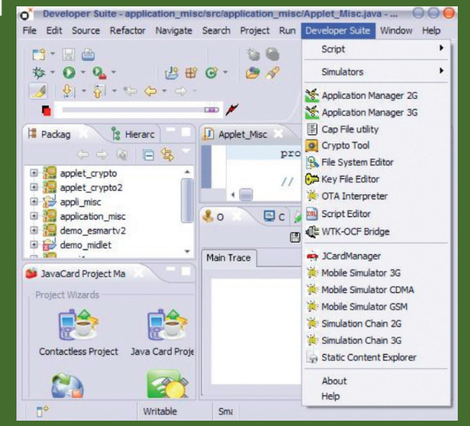


3 *Interface graphique du WTK 2.5*

5.2 Developer Suite de Gemalto

Developer Suite est une suite logicielle de développement d'applets Java Card (U)SAT de la société Gemalto. C'est un module qui s'intègre à l'IDE Eclipse. Nous l'utilisons dans sa version 3.1.

La Developer Suite est dotée d'un environnement de simulation assez bien fourni offrant de nombreuses fonctionnalités. Il est possible de simuler plusieurs types de cartes (U)SIM, et d'explorer leur contenu. Il est également possible de tester le fonctionnement des applets (U)SAT grâce au *Mobile Simulator*, ainsi que de tester l'envoi et la réception de SMS. Un point intéressant est l'intégration du WTK qui permet de faire des tests sur la communication midlet/applet avant de déployer les applications sur de véritables terminaux. Il est enfin important de noter la présence d'un *Application Manager* qui facilite le chargement des applets développées sur de véritables terminaux.



4 *Interface graphique de la Developer Suite 3.1*

6. Choix du matériel

6.1 Les téléphones

Les téléphones mobiles que l'on peut utiliser doivent implémenter la JSR-177 et supporter les applets signées par des CA (*Certificate Authority*) connues, comme Thawte ou Verisign, les midlets utilisant la JSR-177 devant être signées.

Notre choix s'est porté sur le Nokia 5610 Xpressmusic qui remplit toutes les conditions et sur lequel nous avons effectivement réalisé des communications midlet/applet.

6.2 Les cartes (U)SIM

Le challenge consiste à trouver des cartes (U)SIM connectées à un réseau GSM/3G (afin de pouvoir bénéficier des services réseau comme l'envoi de SMS) et sur lesquelles il est possible de charger et d'exécuter des applets Java Card et même des (U)SAT.

Les cartes (U)SIM R5/R6 de Gemalto (fournies avec le kit de développement de la Developer Suite) représentent une bonne solution, même si étant de test elles ne sont pas liées à un réseau GSM. Elles permettent néanmoins de tester l'envoi de commandes APDU d'une midlet vers une applet ainsi que certaines commandes proactives des (U)SIM.

7. Exemple

Les sources de l'exemple étant un peu trop volumineuses, elles sont disponibles à l'URL suivante : **http://www.labri.fr/perso/chaumett/papers/misc2008/usim**.

L'exemple que nous avons développé est un code permettant d'enregistrer des données confidentielles (coordonnées bancaires, mots de passe, etc.) dans un téléphone. Les informations sont stockées de façon sécurisée sur la carte (U)SIM et on peut les communiquer (par SMS par exemple) à une tierce personne toujours de façon sûre.

Cette application reprend tous les éléments que nous avons présentés dans cet article. Elle utilise également les possibilités cryptographiques des cartes Java pour chiffrer les informations confidentielles et ainsi pouvoir les transmettre sans les divulguer à un observateur éventuel.

Le scénario de l'application est simple. Lorsque l'utilisateur veut communiquer ses informations confidentielles, il se sert d'un bouton `send` qui déclenche la récupération de l'information chiffrée au niveau de la carte (U)SIM et son envoi par SMS. De manière symétrique, lors de la réception d'un SMS contenant une information sensible communiquée par une tierce personne, le message est automatiquement transféré à la carte pour déchiffrement avant d'être affiché sur l'écran du téléphone mobile. L'application est divisée en deux parties : la midlet sur le téléphone et l'applet sur la carte (U)SIM.

Le code de la midlet (listing 5, voir l'URL citée plus haut) est divisé en quatre blocs, à savoir, la classe principale `Midlet_Misc`, la classe `Send_APDU`, la classe `Receive_SMS` et la classe `Send_SMS`. Cette midlet est chargée de proposer une interface à l'utilisateur qui lui permette de déclencher l'envoi de l'information confidentielle à transmettre (chiffrée) par SMS. La midlet communique avec l'applet appropriée, récupère l'information chiffrée et l'expédie au contact indiqué. Les différentes classes qui ont été définies sont les suivantes :

- La classe `Midlet_Misc` est la classe principale et elle présente une structure classique de midlet. Les différents éléments à noter se situent d'abord au niveau de son constructeur qui définit les éléments de l'interface graphique (`stringItem` pour l'affichage des informations et les boutons `commandExit`, `commandSend` et `commandReceive` pour l'exécution des différentes actions). La méthode `initApp` est exécutée au démarrage de l'application ; elle contient une séquence de lancement du thread chargé de recevoir les SMS (à travers la classe `Receive_SMS`). La méthode `sendSMS`, elle, se charge tout simplement d'envoyer une chaîne de caractères par SMS en utilisant la classe `Send_SMS`. La méthode `fill_apdu` formate dans un tableau d'octets la commande APDU à envoyer à la carte (U)SIM pour réaliser une opération de déchiffrement. Cette APDU contient l'information à déchiffrer dans son champ de données. La méthode `receivedSMS` est chargée de déclencher les opérations à exécuter (formatage de la commande APDU et envoi de cette commande à la carte) lors de la réception par SMS d'un message contenant une information confidentielle. Enfin, la méthode `commandAction` déclenche les actions liées à chaque bouton de l'interface, `CMD_EXIT` qui ferme l'application, `CMD_SEND` qui récupère l'information dans la carte (en utilisant la classe `Send_APDU`) avant qu'elle ne soit envoyée par SMS et `CMD_RECEIVE` qui permet de lancer les opérations de déchiffrement au niveau de la carte (en utilisant là aussi la classe `Send_APDU`). La variable `smsPort` contient le numéro de port utilisé dans la classe `Receive_SMS` pour récupérer les messages SMS destinés à l'application. Le tableau d'octets `encryptCommand` quant à lui contient la commande APDU utilisée pour récupérer l'information confidentielle à communiquer via SMS.
- La classe `Send_APDU` est quasiment identique à celle utilisée dans l'exemple de communication midlet/applet de la section 4.2. Elle présente tout de même une particularité au niveau de la méthode `run`. Lorsque la commande APDU concerne le cryptage de l'information confidentielle (tag 0xC1) le message crypté est ensuite envoyé par SMS (par la méthode `parent.sendSMS`) ; dans le cas du tag 0xC3 (décryptage) l'information est simplement affichée en clair sur l'écran par la méthode parent.`setMessage`.
- La classe `Receive_SMS` gère la réception de tous les messages transférés par SMS sur le port `smsPort`. Dans le constructeur, on trouve l'ouverture de la connexion avec l'instruction `(MessageConnection)Connector.open(smsConnection)` associée à l'objet `smsconn` de type `MessageConnection`. La méthode `closeConnection` ferme la connexion SMS à la fin de tous les traitements. La méthode `run` quant à elle contient une boucle qui permet de recevoir, puis de traiter chaque SMS. Chaque message est reçu en utilisant `smsconn.receive` pour ensuite être traité par l'invocation de `parent.receivedSMS`.
- La classe `Send_SMS` permet l'envoi par SMS d'une chaîne de caractères à une destination donnée. Toutes les actions sont regroupées dans la méthode `run`. On ouvre tout d'abord la connexion comme dans la classe `Receive_SMS`. L'adresse de destination est ensuite définie avant que le message ne soit envoyé avec `smsconn.send`. La connexion est enfin refermée en invoquant `smsconn.close`.

Le code de l'applet (listing 6, voir l'URL citée plus haut) est lui aussi très classique. Il comprend les éléments présents dans toute applet Java Card, et intègre de plus les méthodes de chiffrement et de déchiffrement 3DES de tableaux d'octets. Les spécificités à noter se situent tout d'abord au niveau du constructeur `Applet_Misc` avec l'invocation de la méthode `initCrypto()`. Cette méthode permet d'initialiser tous les éléments nécessaires à l'utilisation d'une clé 3DES (dont la valeur est stockée dans le tableau d'octets `user3DESCipheringKeyValue`). La méthode `initCrypto()` est aussi chargée de l'initialisation des objets `cipher3DESEnc` et `cipher3DESDec` avec la clé 3DES. Ces objets contiennent respectivement les méthodes pour chiffrer et déchiffrer les données. La méthode `padUserData` permet de formater correctement les données à chiffrer, la longueur de ces dernières devant être un multiple de 8 octets. Le reste du traitement au niveau de l'applet se retrouve dans la méthode `process`. Dans le cas de la réception d'une commande APDU avec le champs INS de type `INS_CRYPT (0xC1)`, l'information confidentielle est chiffrée avant d'être retournée dans la réponse APDU. L'information est auparavant formatée (longueur multiple de 8) avec `padUserData`, puis la méthode `cipher3DESEnc.doFinal(apduData, (short) 0x00, nbData, apduDataEncrypted, (short) 0x00)` effectue les opérations de chiffrement et le résultat est produit dans le tableau `apduDataEncrypted`. Si au contraire la commande APDU reçue dans la méthode `process` est du type `INS_DECRYPT (0xC3)`, les données contenues dans la commande sont alors déchiffrées avec `cipher3DESDec.doFinal(apduData, (short) 0x00, nbData, apduDataEncrypted, (short) 0x00)` pour ensuite être renvoyées en réponse APDU avec `apdu.sendBytesLong(apduDataEncrypted, (short)0x00,nbData)`.

Conclusion

L'utilisation des cartes à puce Java et de leurs possibilités cryptographiques dans le contexte d'applications embarquées pour téléphones mobiles ouvre de nombreuses possibilités. Il devient en effet possible de sécuriser les données liées aux applications afin de s'assurer que les personnes non autorisées n'y ont pas accès. Il est également possible de sécuriser les communications en les chiffrant. Ce chiffrement peut être mis en œuvre quelle que soit la technologie de communication utilisée : Bluetooth, WIFI, GSM, 3G.

De plus, la mise en œuvre des technologies utilisées est relativement simple et accessible, et les équipements compatibles devraient se généraliser dans un avenir proche.

Dans le cadre de nos activités de recherche, nous utilisons ces technologies en particulier au sein du projet Multilevel Java Card Grid. Ce projet a pour objectif de proposer une architecture de communication multi-niveau et sécurisée utilisant des téléphones mobiles **[5]**, **[6]**. Les utilisations potentielles de ce système sont nombreuses. Le paiement par téléphone mobile, le stockage et la communication d'informations sensibles sont autant d'exemples de domaine d'application.

Références

[1] *The Java Card technology*, **http://java.sun.com/javacard/index.jsp**

[2] *Java Card applet developer's guide*, **http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html?page=1**

[3] *USIM/USAT support*, **http://www.accessdevnet.com/index.php/ACCESS-Linux-Platform-Native-Development/ALP_Telephony_MobileServices.html#1013158**

[4] *The Java Micro Edition technology*, **http://java.sun.com/javame/technology/index.jsp**

[5] CHAUMETTE (S.), MARKANTONAKIS (K.), MAYES (K.) ET SAUVERON (D.), *The Mobile Java Card Grid*, e-Smart, 2006.

[6] CHAUMETTE (S.), OUOBA (J.), *The Multilevel Java Card Grid, invited poster*, WISTP 2007, Greece, 2007.

- SAUVERON (D.), La technologie Java Card : présentation de la carte à puce, La Java Card, RR-1259-01, LaBRI, Université Bordeaux 1, 2001.
- ouoba (J.), *The Multilevel Java Card Grid*, *Master Thesis Report*, LaBRI, University Bordeaux 1, 2007.
- NEMEC (J.), *Simagine 2008 Training, Developing Java STK, NFC and SCWS applications*, Gemalto, Meudon, 2007.