Security in Networked Computing Systems PROJECT

A.Y. 2013/2014

This page intentionally left blank.

Contents

Chapter 1

Project specifications

On Android device it is possible to intercept SMS text message before they leave the phone or as soon as they are received. To avoid this privacy issue it is possible to implement an SMS encyption app to exchange SMS messages with other users sharing the same apps. The users have both a public and private key. When a user wants to start a private communication with a peer, a protocol to establish a session key is started. Design and analyze a protocol which satisfies the following requirements:

- At the end of the protocol, both peers share a session key for symmetric encryption of SMS messages.
- At the end of the protocol, both peers are sure that the other peer received the session key.
- Expiring PINs should be used to ensure freshness.

Develop a sample application that performs the protocol and shares encrypted messages. The key exchange protocol is performed through SMS text messages, while PINs are manually inserted by users.

Chapter 2

Protocol Analysis

The main aim of this chapter is to describe the key exchange protocol used by the application and to analyze such protocol with the BAN logic.

2.1 Legend of symbols

This section introduces a brief legend of the symbols used during the protocol analysis.

- N_A: A's nonce
- N_B: B's nonce
- $pubK_A$: A's public key
- $pubK_B$: B's public key
- K_{AB}: session key shared between A and B

2.2 Real protocol

This section shows the so called *real protocol*, that is the sequence of messages exchanged by the two parties involved in the protocol. The whole list of messages exchanged is presented in the followings lines:

1. M1: A→B: $\{N_A \mid A\}_{pubK_B}$
2. M2: B→A: $\{N_A \mid N_B \mid B \mid K_{AB}\}_{pubK_A}$
3. M3: A→B: $\{N_B\}_{K_{AB}}$

2.3 Idealized protocol

This section shows the *idealized protocol*, that is the 1^{st} step in the analysis of a protocol with the BAN logic.

1. M1: A→B: $\{N_A\}_{pubK_B}$
2. M2: B→A: $\left\{N_A \mid N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right\}_{pubK_A}$
3. M3: A→B: $\left\{N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right\}_{K_{AB}}$

2.4 Assumptions

1. $A \mid\equiv \#(N_A)$
2. $B \mid\equiv \#(N_B)$
3. $A \mid\equiv \overset{pubK_A}{\longmapsto} A$
4. $B \mid\equiv \overset{pubK_B}{\longmapsto} B$
5. $A \mid\equiv \overset{pubK_B}{\longmapsto} B$
6. $B \mid\equiv \overset{pubK_A}{\longmapsto} A$
7. $A \mid\equiv B \Rightarrow A \overset{K_{AB}}{\longleftrightarrow} B$
8. $B \mid\equiv A \overset{K_{AB}}{\longleftrightarrow} B$

2.5 Goals

The goals of the protocol are **key authentication** and **key confirmation**; in the following lines they are expressed in terms of BAN logic.

- Key Authentication
  1. $A \mid\equiv A \overset{K_{AB}}{\longleftrightarrow} B$
  2. $B \mid\equiv A \overset{K_{AB}}{\longleftrightarrow} B$
- Key Confirmation
  1. $A \mid\equiv B \mid\equiv A \overset{K_{AB}}{\longleftrightarrow} B$
  2. $B \mid\equiv A \mid\equiv A \overset{K_{AB}}{\longleftrightarrow} B$

2.6 Protocol analysis

This section is completely dedicated to the actual analysis of the protocol. The main aim is to prove the goals showed in the previous section, by appling the BAN logic to the assumptions.

After message M1

- Appling the following postulate to **M1** and **4**, we obtain **9**

$$\frac{P \mid\equiv \stackrel{K}{\longmapsto} P, P \triangleleft \{X\}_K}{P \triangleleft \{X\}}$$

$$\frac{B \mid\equiv \stackrel{pubK_B}{\longmapsto} B, B \triangleleft \{N_A\}_{pubK_B}}{B \triangleleft \{N_A\}} \qquad \mathbf{9}$$

After message M2

- Appling the following postulate to **M2** and **3**, we obtain **10**

$$\frac{P \mid\equiv \stackrel{K}{\longmapsto} P, P \triangleleft \{X\}_K}{P \triangleleft \{X\}}$$

$$\frac{A \mid\equiv \stackrel{pubK_A}{\longmapsto} A, A \triangleleft \left\{N_A \mid N_B \mid A \stackrel{K_{AB}}{\longleftrightarrow} B\right\}_{pubK_B}}{B \triangleleft \left\{N_A \mid N_B \mid A \stackrel{K_{AB}}{\longleftrightarrow} B\right\}} \qquad \mathbf{10}$$

- Appling the following postulate to **1**, **M1** and **M2**, we obtain **11**

$$\frac{P \mid\equiv \#\left(X\right), P \mid\sim \{X \mid Y\}_{pk_q}, P \triangleleft \{X \mid Z\}_{pk_p}}{P \mid\equiv Q \triangleleft \left(X \mid Y\right), P \mid\equiv Q \mid\sim \left(X \mid Z\right)}$$

$$\frac{A \mid\equiv \#\left(N_A\right), A \mid\sim \{N_A\}_{pubK_B}, A \triangleleft \left\{N_A \mid N_B \mid A \stackrel{K_{AB}}{\longleftrightarrow} B\right\}_{pubK_A}}{A \mid\equiv B \triangleleft \left(N_A\right), A \mid\equiv B \mid\sim \left(N_A \mid N_B \mid A \stackrel{K_{AB}}{\longleftrightarrow} B\right)} \qquad \mathbf{11}$$

- Appling the following postulate to **1**, we obtain **12**

$$\frac{P \mid\equiv \#(X)}{P \mid\equiv \#(X \mid Y)} \qquad \textbf{P1}$$

$$\frac{A \mid\equiv \#(N_A)}{A \mid\equiv \#\left(N_A \mid N_B \mid A \xleftrightarrow{K_{AB}} B\right)} \qquad \textbf{12}$$

- Appling the nonce verification rule to **12** and **11**, we obtain **13**

$$\frac{P \mid\equiv (X), P \mid\equiv Q \mid\sim X}{P \mid\equiv Q \mid\equiv X} \qquad \textbf{NVR}$$

$$\frac{A \mid\equiv \#\left(N_A \mid N_B \mid A \xleftrightarrow{K_{AB}} B\right), A \mid\equiv B \mid\sim \left(N_A \mid N_B \mid A \xleftrightarrow{K_{AB}} B\right)}{A \mid\equiv B \mid\equiv \left(N_A \mid N_B \mid A \xleftrightarrow{K_{AB}} B\right)} \qquad \textbf{13}$$

- Appling the following postulate to **13**, we obtain **14**

$$\frac{P \mid\equiv Q \mid\equiv (X \mid Y)}{P \mid\equiv Q \mid\equiv X} \qquad \textbf{P2}$$

$$\frac{A \mid\equiv B \mid\equiv \left(N_A \mid N_B \mid A \xleftrightarrow{K_{AB}} B\right)}{A \mid\equiv B \mid\equiv \left(A \xleftrightarrow{K_{AB}} B\right)} \qquad \textbf{14}$$

- Appling the jurisdiction rule to **14** and **7**, we obtain **15**

$$\frac{P \mid\equiv\mid\equiv X, P \mid\equiv Q \Rightarrow X}{P \mid\equiv X}$$

$$\frac{A \mid\equiv B \mid\equiv \left(A \xleftrightarrow{K_{AB}} B\right), A \mid\equiv B \Rightarrow \left(A \xleftrightarrow{K_{AB}} B\right)}{A \mid\equiv \left(A \xleftrightarrow{K_{AB}} B\right)} \qquad \textbf{15}$$

After message M3

- Appling the message meaning rule to **8** and **M3**, we obtain **16**

$$\frac{P \mid\equiv Q \overset{K_{AB}}{\longleftrightarrow} P, P \triangleleft \{X\}_K}{P \mid\equiv Q \mid\sim X}$$

$$\frac{B \mid\equiv A \overset{K_{AB}}{\longleftrightarrow} B, B \triangleleft \left\{N_B \mid \left(A \overset{K_{AB}}{\longleftrightarrow} B\right)\right\}_{K_{AB}}}{B \mid\equiv A \mid\sim \left(N_B \mid \left(A \overset{K_{AB}}{\longleftrightarrow} B\right)\right)} \qquad \mathbf{16}$$

- Appling the postulate **P1** to **2**, we obtain **17**

$$\frac{B \mid\equiv \#\,(N_B)}{B \mid\equiv \# \left(N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right)} \qquad \mathbf{17}$$

- Appling the nonce verification rule **NVR** to **17** and **16**, we obtain **18**

$$\frac{B \mid\equiv \# \left(N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right), B \mid\equiv A \mid\sim \left(N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right)}{B \mid\equiv A \mid\equiv \left(N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right)} \qquad \mathbf{18}$$

- Appling the postulate **P2** to **18**, we obtain **19**

$$\frac{B \mid\equiv A \mid\equiv \left(N_B \mid A \overset{K_{AB}}{\longleftrightarrow} B\right)}{B \mid\equiv A \mid\equiv \left(A \overset{K_{AB}}{\longleftrightarrow} B\right)} \qquad \mathbf{19}$$

2.7 Meeting the goals

- The *key authentication* is met with the assertions **15** and **8**.
- The *key confirmation* is met with the assertions **14** and **19**.

Chapter 3

Implementation

This chapter presents the main implementative choices taken during the development of the application; it also shows the salient portions of the code such as, for example, the class that realizes the ciphers, the functions used during the course of the protocol, etc.

3.1 Hypotheses and implementative choiches

Below lists the working hypothesis, under which the application runs correctly.

- **A** knows a priori its own public and private key, **B**'s public key and vice versa.
- Keys are stored in the mass memory, more precisely in the folder *'.. SSMSkeys'*, and they have **.key** extension.
  Their own key pair is formed by the file *'private.key'* and *'public.key'*.
  The public key of the recipient with a generic number DESTPHONE is stored in the file *'DESTPHONE.key'*.
  Once generated, the session key between **A** and **B** is stored in the same folder as *'PHONE1_PHONE2.key'*, where *PHONE1* and *PHONE2* are respectively **A**'s and **B**'s phone numbers.
- The session key has lifetime of a single SMS, then it expires becoming no more valid.
- In order to semplify the implementation, it suppose that, during the protocol handshake, each message to read from the inbox folder is the

last sent by the other entity. In this way it is easy to recover the message; otherwise we have to plan a function able to distinguish normal SMSs from encrypted ones.

- The algorithm for asymmetric encryption is RSA with 1024 bits keys. That used for symmetric encryption is AES with 128 bits keys and CBC as encryption mode.

- Through SMS we can send only text strings; so the encryption function produces encrypted bytes which are then encoded into a Base64 text string. This solution is better, in terms of length, than sending a string composed by the concatenation of the integer encoding of each single byte.

- The algorithm for symmetric encryption uses an *initialization vector* **IV** which is knew a priori by both the parties. It is composed by the following bytes:

```
byte[] IV = { 0x0a, 0x01, 0x02, 0x03, 0x04, 0x0b, 0x0c, 0x0d, 0
    x0a, 0x01, 0x02, 0x03, 0x04, 0x0b, 0x0c, 0x0d };
```

- The user has to wait for the SMS and then the message can be read correctly. On the contrary there will be generated an error. Errors are handled as easy as possible: when an error occours, the app returns on the main activity.

3.2 Code

The following pages set out the most important parts of the code of the application. Are not reported, however, the pieces of code related to the graphical interface (GUI) because they are not considered essential.

3.2.1 *Ciphers.java*

```
package com.example.sssm;


//******************
// import directives
//******************
```

```java
import javax.crypto.Cipher;
import java.security.Key;
import javax.crypto.spec.IvParameterSpec;
import android.util.Base64;


//*************************
// cipher classes definition
//*************************
   // AsymmetricCipher class
class AsymmetricCipher
{
   private Cipher cipher;

   public AsymmetricCipher(String xform) throws Exception
   {
      this.cipher = Cipher.getInstance(xform);
   }

   public String encrypt(String plainText, Key key) throws Exception
   {
      String cipherText = "";
      // encrypt
      cipher.init(Cipher.ENCRYPT_MODE, key);
      byte[] cipherTextBytes = cipher.doFinal(plainText.getBytes());
      cipherText = Base64.encodeToString(cipherTextBytes, Base64.DEFAULT);
      return cipherText;
   }

   public String decrypt(String cipherText, Key key) throws Exception
   {
      String plainText = "";
      byte[] cipherTextBytes = Base64.decode(cipherText, Base64.DEFAULT);
      // decrypt
      cipher.init(Cipher.DECRYPT_MODE, key);
      byte[] plainTextBytes = cipher.doFinal(cipherTextBytes);
      // decode into a plaintext string
      for(int i=0; i<plainTextBytes.length; i++)
         plainText += (char)plainTextBytes[i];
      return plainText;
   }
```

```
}
   // SymmetricCipher class
class SymmetricCipher
{
   private Cipher cipher;
   private IvParameterSpec ips;
   private byte[] IV;
   private Key key;

   public SymmetricCipher(Key key, String xform, byte[] IV) throws
       Exception
   {
      this.cipher = Cipher.getInstance(xform);
      this.IV = IV;
      this.key = key;

      if(this.IV != null)
         this.ips = new IvParameterSpec(IV);
   }

   public void setKey(Key key)
   {
      this.key = key;
   }

   public String encrypt(String plainText) throws Exception
   {
      String cipherText = "";
      // encrypt
      if(IV != null)
         cipher.init(Cipher.ENCRYPT_MODE, key, ips);
      else
         cipher.init(Cipher.ENCRYPT_MODE, key);
      byte[] cipherTextBytes = cipher.doFinal(plainText.getBytes());
      cipherText = Base64.encodeToString(cipherTextBytes, Base64.
          DEFAULT);
      return cipherText;
   }

   public String decrypt(String cipherText) throws Exception
   {
      String plainText = "";
```

```
    byte[] cipherTextBytes = Base64.decode(cipherText, Base64.
        DEFAULT);
    // decode into bytes
    if(IV != null)
      cipher.init(Cipher.DECRYPT_MODE, key, ips);
    else
      cipher.init(Cipher.DECRYPT_MODE, key);

    byte[] plainTextBytes = cipher.doFinal(cipherTextBytes);
    // decode into a plaintext string
    for(int i=0; i<plainTextBytes.length; i++)
      plainText += (char)plainTextBytes[i];
    return plainText;
  }
}
```

3.2.2 *KeyStorage.java*

The class *KeyStorage* allows to load/store keys from/into files. There are two constructors: one is used for load/store keys of an asymmetric cipher; the other one is used for load/store keys of a symmetric cipher.

```
package com.example.sssm;


//*****************
// import directives
//*****************
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;


//*******************
```

```java
// key storage class
//********************
   // KeyStorage class
class KeyStorage
{
   private String publicKeyPath;
   private String privateKeyPath;
   private String sharedKeyPath;
   private String publicKeyFilename;
   private String privateKeyFilename;
   private String sharedKeyFilename;
   private String algorithm;

   // **constructors**
   public KeyStorage(String publicKeyPath, String privateKeyPath,
       String publicKeyFilename, String privateKeyFilename)
   {
      this.publicKeyPath = publicKeyPath;
      this.privateKeyPath = privateKeyPath;
      this.publicKeyFilename = publicKeyFilename;
      this.privateKeyFilename = privateKeyFilename;
   }

   public KeyStorage(String sharedKeyPath, String sharedKeyFilename,
       String algorithm)
   {
      this.sharedKeyPath = sharedKeyPath;
      this.sharedKeyFilename = sharedKeyFilename;
      this.algorithm = algorithm;
   }

   // **asymmetric**
   public KeyPair loadKeyPair()
   {
      return new KeyPair(this.loadPublicKey(),this.loadPrivateKey());
   }

   public PrivateKey loadPrivateKey()
   {
      PrivateKey privateKeyR = null;

      try
      {
```

```
      // Read Private Key.
      File filePrivateKey = new File(privateKeyPath +
          privateKeyFilename);
      FileInputStream fis = new FileInputStream(privateKeyPath +
          privateKeyFilename);
      byte[] encodedPrivateKey = new byte[(int) filePrivateKey.
          length()];
      fis.read(encodedPrivateKey);
      fis.close();

      // Reconstruct
      KeyFactory keyFactory = KeyFactory.getInstance("RSA");
      PKCS8EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec
          (encodedPrivateKey);
      privateKeyR = keyFactory.generatePrivate(privateKeySpec);
    }
    catch(Exception e) {}

    return privateKeyR;
  }

  public PublicKey loadPublicKey()
  {
    PublicKey publicKeyR = null;

    try
    {
      // Read Public Key.
      File filePublicKey = new File(publicKeyPath +
          publicKeyFilename);
      FileInputStream fis = new FileInputStream(publicKeyPath +
          publicKeyFilename);
      byte[] encodedPublicKey = new byte[(int) filePublicKey.
          length()];
      fis.read(encodedPublicKey);
      fis.close();

      // Reconstruct
      KeyFactory keyFactory = KeyFactory.getInstance("RSA");
      X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(
          encodedPublicKey);
      publicKeyR = keyFactory.generatePublic(publicKeySpec);
    }
```

```
        catch(Exception e) {}

        return publicKeyR;
    }

    //**symmetric**
    public void saveSharedKey(SecretKey key) throws Exception
    {
        FileOutputStream fos = new FileOutputStream(sharedKeyPath +
            sharedKeyFilename);
        fos.write(key.getEncoded());
        fos.close();
    }

    public SecretKey loadSharedKey() throws Exception
    {
        File fileSharedKey = new File(sharedKeyPath + sharedKeyFilename
            );
        FileInputStream fis = new FileInputStream(sharedKeyPath +
            sharedKeyFilename);
        byte[] encodedSharedKey = new byte[(int) fileSharedKey.length()
            ];
        fis.read(encodedSharedKey);
        fis.close();
        return new SecretKeySpec(encodedSharedKey, algorithm);
    }
}
```

3.2.3 Handshake functions

The key establishment protocol, as said before, is basically a 3-steps handshake. Below is the code that is executed for each message.
Since the text to send is longer than 160 chars (that is the maximum length of a standard SMS), we can't use the *sendTextMessage()* function. Infact, we have first to split the string in multiple parts with the *divideMessage()* function and then send the parts obtained using the *sendMultipartTextMessage()*.
If an error occurs, it is reported and the application returns to the Main Activity.

Send message #0

```
private void sendMsg0(String destPhone) throws Exception
   {
      AsymmetricCipher ac = new AsymmetricCipher("RSA/ECB/
          PKCS1Padding");
      KeyStorage myAsymStorage;
      PublicKey bPublicKey;

      // retrieve B's public key
      File path = Environment.getExternalStorageDirectory();
      String keysPath = path.getAbsolutePath() + "/SSMSkeys/";
      myAsymStorage = new KeyStorage(keysPath, "", destPhone + ".key"
          , "");
      bPublicKey = myAsymStorage.loadPublicKey();
      // retrieve my phone number
      Scanner s = new Scanner(new FileReader(keysPath + "myPhone"));
      myPhone = s.next();
      s.close();
      // prepare SMS text
      String plainText = nonceA + "|" + myPhone;
      // encrypt plaintext
      String cipherText = ac.encrypt(plainText, bPublicKey);
      // send message
      SmsManager smanager = SmsManager.getDefault();
      ArrayList<String> parts = smanager.divideMessage(cipherText);
      smanager.sendMultipartTextMessage(destPhone, null, parts, null,
           null);
   }
```

Send message #1

```
private int sendMsg1(String destPhone) throws Exception
  {
    AsymmetricCipher ac = new AsymmetricCipher("RSA/ECB/
        PKCS1Padding");
    KeyStorage myAsymStorage, mySymStorage;
    PrivateKey myKey;
    PublicKey aPublicKey;
    String cipherText;

    // retrieve A's public key
    File path = Environment.getExternalStorageDirectory();
    String keysPath = path.getAbsolutePath() + "/SSMSkeys/";
    myAsymStorage = new KeyStorage(keysPath, keysPath, destPhone +
        ".key", "private.key");
    aPublicKey = myAsymStorage.loadPublicKey();
    // retrieve my private key
    myKey = myAsymStorage.loadPrivateKey();

    // retrieve cipher text SMS: it is the last received message
        from the sender whose phone number is equal to 'destPhone'
    Cursor cursor = getContentResolver().query(Uri.parse("content
        ://sms/inbox"), null, null, null, null);
    // check if there messages; if no messages are found, it
        returns -1
    if(!cursor.moveToFirst())
      return -1;
    // scroll all messages and find the last sent by 'destPhone';
        if no message from destPhone is sent, it returns -2
    for(;;)
    {
      String tmpSender = cursor.getString(cursor.
          getColumnIndexOrThrow("address"));
      if(tmpSender.equals(destPhone))
      {
        // message found! it retrieves the cipher text and breaks
            the loop
        cipherText = cursor.getString(cursor.getColumnIndexOrThrow
            ("body"));
        break;
      }
```

```java
      // try the next message; if no more messages are available, it returns -2
      if(!cursor.moveToNext())
        return -2;
    }
    // decrypt first and split the message; msgFields[0] contains A's nonce, msgFields[1] contains A's phone number
    String plainText = ac.decrypt(cipherText, myKey);
    String[] msgFields = plainText.split("[\\x7C]"); // 'x7C' ASCII code for vertical bar '|'
    // check for sender equality; if destPhone is not equal to the number contained in the message, it returns -3
    if(!destPhone.equals(msgFields[1]))
      return -3;

    // prepare SMS text
      // retrieve my phone number
    Scanner s = new Scanner(new FileReader(keysPath + "myPhone"));
    myPhone = s.next();
    s.close();
      // generate the session key
    KeyGenerator kg = KeyGenerator.getInstance("AES");
    kg.init(128);
    SecretKey key = kg.generateKey();
      // save the session key
    mySymStorage = new KeyStorage(keysPath, myPhone + "_" + destPhone + ".key", "AES");
    mySymStorage.saveSharedKey(key);
      // encode key into a DEC string
    byte [] encodedKey = key.getEncoded();
    String encodedKeyString = "";
    for(int i=0; i<encodedKey.length; i++)
    {
      encodedKeyString += encodedKey[i];
      if(i == (encodedKey.length - 1))
        continue;
      encodedKeyString += " ";
    }
      // prepare text to encrypt
    plainText = msgFields[0] + "|" + nonceB + "|" + myPhone + "|" + encodedKeyString;
      // encrypt plaintext
    cipherText = ac.encrypt(plainText, aPublicKey);
```

```java
        // send message
        SmsManager smanager = SmsManager.getDefault();
        ArrayList<String> parts = smanager.divideMessage(cipherText);
        smanager.sendMultipartTextMessage(destPhone, null, parts, null, null);
        return 0;
    }
```

Send message #2

```java
private int sendMsg2() throws Exception
    {
        AsymmetricCipher ac = new AsymmetricCipher("RSA/ECB/PKCS1Padding");
        KeyStorage myAsymStorage;
        PrivateKey myKey;
        String cipherText;

        // retrieve my private key
        File path = Environment.getExternalStorageDirectory();
        String keysPath = path.getAbsolutePath() + "/SSMSkeys/";
        myAsymStorage = new KeyStorage("", keysPath, "", "private.key");
        myKey = myAsymStorage.loadPrivateKey();

        // retrieve cipher text SMS: it is the last received message from the sender whose phone number is equal to 'destPhone'
        Cursor cursor = getContentResolver().query(Uri.parse("content://sms/inbox"), null, null, null, null);
        // check if there messages; if no messages are found, it returns -1
        if(!cursor.moveToFirst())
        return -1;
        // scroll all messages and find the last sent by 'destPhone'; if no message from destPhone is sent, it returns -2
        for(;;)
        {
            String tmpSender = cursor.getString(cursor.getColumnIndexOrThrow("address"));
            if(tmpSender.equals(destPhone))
            {
```

```java
            // message found! it retrieves the cipher text and breaks the loop
            cipherText = cursor.getString(cursor.getColumnIndexOrThrow("body"));
            break;
        }
        // try the next message; if no more messages are available, it returns -2
        if(!cursor.moveToNext())
            return -2;
    }
    // decrypt first and split the message;
    // msgFields[0] contains my nonce (A's nonce)
    // msgFields[1] contains B's nonce
    // msgFields[2] contains B's phone number
    // msgFields[3] contains the session key encoded in bytes expressed with their integer representation and separated by white spaces
    String plainText = ac.decrypt(cipherText, myKey);
    String[] msgFields = plainText.split("[\\x7C]"); // 'x7C' ASCII code for vertical bar '|'
    // check for sender equality; if destPhone is not equal to the number contained in the message, it returns -3
    if(!destPhone.equals(msgFields[2]))
        return -3;
    // check for nonce equality; if nonceA is not equal to that contained in the message, it returns -4
    if(!msgFields[0].equals(nonceA))
        return -4;

    //reconstruct the shared key
    String[] bytesString = msgFields[3].split(" ");
    byte[] bytes = new byte[bytesString.length];
    for(int i=0; i<bytes.length; i++)
        bytes[i] = Byte.parseByte(bytesString[i]);
    sharedKey = new SecretKeySpec(bytes, "AES");
    // generate the symmetric cipher
    sc = new SymmetricCipher(sharedKey, "AES/CBC/PKCS5Padding", IV);

    // prepare SMS text
        // prepare text to encrypt
    plainText = new String(msgFields[1]);
```

```
      // encrypt plaintext
    cipherText = sc.encrypt(plainText);
      // send message
    SmsManager smanager = SmsManager.getDefault();
    ArrayList<String> parts = smanager.divideMessage(cipherText);
    smanager.sendMultipartTextMessage(destPhone, null, parts, null, null);
    return 0;
  }
```

Receive message #2

```
private int receiveMsg2() throws Exception
  {
    String plainText, cipherText;

    // retrieve cipher text SMS: it is the last received message from the sender whose phone number is equal to 'destPhone'
    Cursor cursor = getContentResolver().query(Uri.parse("content://sms/inbox"), null, null, null, null);
    // check if there messages; if no messages are found, it returns -1
    if(!cursor.moveToFirst())
      return -1;
    // scroll all messages and find the last sent by 'destPhone'; if no message from destPhone is sent, it returns -2
    for(;;)
    {
      String tmpSender = cursor.getString(cursor.getColumnIndexOrThrow("address"));
      if(tmpSender.equals(destPhone))
      {
        // message found! it retrieves the cipher text and breaks the loop
        cipherText = cursor.getString(cursor.getColumnIndexOrThrow("body"));
        break;
      }
      // try the next message; if no more messages are available, it returns -2
      if(!cursor.moveToNext())
        return -2;
```

```
    }
    // decrypt
    plainText = sc.decrypt(cipherText);
    // check for nonce equality; if they are different, it returns
        -3
    if(!plainText.equals(nonceB))
      return -3;
    return 0;
  }
```