Secure Intelligence Gathering Using Smartphones

Doug Serfass
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR, USA
djserfass@ualr.edu

Mengjun Xie
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR, USA
mxxie@ualr.edu

***Abstract*—Equipped with camera, microphone, GPS and other sensors, together with convenient applications such as email clients, smartphones have a high potential to become a cost-effective intelligence gathering platform in a war zone. However, current smartphone applications are developed for civilian use. Due to lack of support for military requirements, such as confidentiality, smartphones may not be directly adopted in military environments. To bridge the gap, we propose a new model for secure intelligence gathering using smartphones. Based on the Android operating system, we develop Imsec, a proof-of-concept application which can securely capture, store, and transfer camera-generated images using smartphones. Our evaluation results demonstrate the feasibility and effectiveness of this new model.**

Encryption; Military computing; Mobile computing; Programming; Electronic mail

I. INTRODUCTION

A soldier on an urban battlefield is using his smartphone to take pictures of enemy locations. After his patrol, he has access to Wi-Fi and uses this opportunity to email his pictures to an officer in his company. Unknown to the soldier or the officer, the pictures are intercepted by a hacker who is contracted by the enemy to monitor local internet traffic. The hacker sends the pictures to the enemy. The enemy uses this intelligence to vacate the locations in the pictures. Could this network security breach have been prevented while still allowing the soldier to use his smartphone as an intelligence gathering tool? Yes. We propose a model that will defend against smartphone image attacks and implement this model on an Android smartphone. This paper consists of the following sections: Section II discusses recent related work on smartphone image security. Section III describes potential attacks against smartphone images. Section IV proposes defenses against these attacks. Section V describes a model based on our defense proposals. Section VI shows Java code for Imsec (Image security), an Android application based on our model. Section VII discusses the results of our study. Section VIII is our conclusions and suggestions for future research.

II. RELATED WORK

Nanjunda et al. [1] proposed opportunistic encryption as a method to optimize the tradeoff between cipher security and throughput loss. Their results showed that applying opportunistic encryption on JPEG compressed images resulted in an improved quality of received images and improved security compared to fixed block length encryption. Kovacevic et al. [2] describe the implementation of a system for secure data transfer in telemedicine applications based on a public key infrastructure. They conclude that, although their system focused on medical image transport, it can be modified to support secure transport of other data types. Shankar et al. [3] and Zhang et al. [4] introduced new encryption algorithms specifically designed for smartphones. Both studies conclude that their new encryption algorithms meet the unique requirements of small computing devices.

III. SMARTPHONE IMAGE ATTACKS

Attacks against smartphone images take place in three distinct locations: the smartphone itself; during transport between the smartphone and the server; and at the server. We discuss four attacks that take place at these locations.

A. Image Compromised on Smartphone

Smartphone is stolen and rooted. By combining a rooted Android smartphone with the Android Debug Bridge, an attacker gains complete access to the Linux file system on the smartphone. This access includes any camera-generated image stored on the smartphone. We used a rooted Android smartphone for application development.

B. Image Intercepted Enroute To (or From) Server

Any Internet traffic is subject to a Man-In-The-Middle (MITM) [5] attack. Regardless of the transport protocol used, any smartphone image transiting between smartphone and server is open to this exploit.

C. Password and Salt Compromised on Smartphone

Password and salt embedded in a file or Java class file on the smartphone. Smartphone is stolen and rooted. File is read or Java class is decompiled. Encrypted images stored on all smartphones for all users compromised. Independent of an attack, this is also an improper method for password and salt storage. Any change to the password and salt would require download of a new file or reinstallation of the application. It is also weak security to have one password and salt for all users.

D. Server Database Login Name and Password Obtained

Smartphone images, user passwords and user salts stored on a server database. Server database login name and password obtained. User passwords and salts obtained. Encrypted

978-1-4673-1375-9/12/$31.00 ©2012 IEEE

smartphone images on the server database for all users compromised. The main security flaw leveraged in this exploit is that encrypted smartphone images, user passwords and user salts are all stored together in one location on a server database.

IV. SMARTPHONE IMAGE DEFENSES

Below are the corresponding defenses against the attacks listed in the previous section.

A. *Encrypt Image Stored on Smartphone*

Encrypted images stored in a file or database on a smartphone cannot be compromised without also obtaining the user password and salt used for encryption. Only unencrypted information will be available to an attacker who steals and roots a smartphone.

B. *Encrypt Image Before Routing To (or From) Server*

Encrypted images intercepted during an MITM attack cannot be compromised without also obtaining the user password and salt used for encryption. Only unencrypted information will be available to an attacker who executes an MITM attack.

C. *Password and Salt on Server Database*

Obtain a user-entered password and salt over HTTPS from a server database. User can login at any time to a web application over HTTPS and modify his password and salt. If required, all distinct smartphone images can each be encrypted with a unique, user-specified, password and salt.

D. *(Password, Salt), (Address, Password) on Two Databases*

User (password, salt) and email (address, password) stored on two distinct server databases. Encrypted smartphone images stored as email message attachments in user inbox on a mail server. A successful attack would first require obtaining two database login names and passwords. Second, an attacker would also need to obtain encrypted smartphone images from a mail server.

V. DEFENSE-BASED MODEL

Based on our defense proposals, we create a model for secure intelligence gathering using smartphones. Our model is shown in Fig. 1.

Encrypted Image = Camera + HTTPS (Password, Salt, PasswordSalt_id)

Write Encrypted Image, PasswordSalt_id to Smartphone Database

Email Image = Wi-Fi+HTTPS (Email Address,Image_id,PasswordSalt_id)

Encrypted Image Stored on Mail Server

Download Image = Wi-Fi+HTTPS (Addr,Pwd,Image_id,PasswordSalt_id)

Write Encrypted Image, PasswordSalt_id to Smartphone Database

Decrypted Image = Read Database + HTTPS (Password, Salt)

Display Decrypted Image

Figure 1. Defense-based model.

This model describes the journey of an encrypted image from a smartphone to a mail server to a second smartphone. The user of the first smartphone (where the picture is taken) might be a soldier. The user of the second smartphone might be an intelligence officer charged with reviewing the decrypted images. An addition to the model would provide the intelligence officer with a computer-based interface for viewing decrypted images.

VI. IMSEC JAVA CODE

We present relevant Java code snippets developed for Imsec as we implemented our defense-based model. This code is not intended as a tutorial for creating an Android application. For information on this topic, the reader is directed to the Android Developers web site. All Java classes discussed in this section are documented in Oracle or Android Javadocs.

A. *Permissions*

Android application permissions are set in the AndroidManifest.xml file. Imsec required the entries as shown in Fig. 2.

```
<"android.permission.INTERNET"/>
<"android.permission.WRITE_EXTERNAL_STORAGE"/>
<"android.permission.CAMERA"/>
```

Figure 2. Imsec permissions.

Internet allows Imsec to open network sockets and is required for email. Write external storage allows Imsec to read and write to external storage (files). Camera allows Imsec to be able to access the camera device.

B. *Get Image During Application Development*

The image is delivered from the camera to Imsec as an instance of the class android.graphics.Bitmap. The camera is not available from the Android emulator during application development. We used the class com.tomgibara.android.camera.HttpCamera [6] to get an image during application development as shown in Fig. 3.

```
u = new
URL("http://www.birdhousesbymark.com/images/chickadee
.jpg");
h = (HttpURLConnection) u.openConnection();
h.setAllowUserInteraction(false);
h.setConnectTimeout(CONNECT_TIMEOUT);
h.setReadTimeout(SOCKET_TIMEOUT);
h.setInstanceFollowRedirects(true);
h.setRequestMethod("GET");
h.connect();
r = h.getResponseCode();
if(r == HttpURLConnection.HTTP_OK){
in = h.getInputStream();
bitmap = BitmapFactory.decodeStream(in);}
```

Figure 3. Java code to get an image during application development.

C. Get Image on Smartphone

The Java code to get an image on the smartphone is shown in Fig. 4.

```
String s =
Environment.getExternalStorageDirectory().getAbsolute
Path() + "/imsec.jpg";

f = new File(s);

Uri u = Uri.fromFile(f);

i = new
Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTU
RE);

i.putExtra(android.provider.MediaStore.EXTRA_OUTPU
T, u);

startActivityForResult(i, 0);

BitmapFactory.Options o = new
BitmapFactory.Options();

o.inSampleSize = 4;

bitmap = BitmapFactory.decodeFile(s,o);

f.delete();
```

Figure 4. Java code to get an image on the smartphone.

android.content.Intent is an abstract description of an operation to be performed. In this case, the operation is to capture an image from the camera. The startActivityForResult method is used to get a result back from an activity when it ends. The result of this method is that the camera image is written to the file imsec.jpg after the picture is taken. The field inSampleSize requests the decoder to subsample the original image, returning a smaller image to save memory. Setting this field to 4 returns an image that is 1/4 the width/height of the original, and 1/16 the number of pixels. Not setting this field caused an application out of memory exception.

D. Encryption

The encryption Java code is shown in Fig. 5.

```
pps = new PBEParameterSpec(salt, iterationCount);

pks = new PBEKeySpec(password);

skf =
SecretKeyFactory.getInstance("PBEWithMD5AndDES");

SecretKey sk = skf.generateSecret(pks);

Cipher ec =
Cipher.getInstance("DES/CFB8/NoPadding");

ec.init(Cipher.ENCRYPT_MODE, sk, pps);

FileOutputStream fos =

context.openFileOutput("imsec.tmp",
Context.MODE_PRIVATE);

cos = new CipherOutputStream(fos, ec);

bitmap.compress(Bitmap.CompressFormat.JPEG, 5,
cos);
```

Figure 5. Encryption Java code.

The user logs in to a web application to obtain his salt, password and PasswordSalt_id over HTTPS. This activity requires very little traffic and can be conducted over the smartphone carrier network (i.e., no Wi-Fi). The user is now ready to use the camera.

javax.crypto.spec.PBEParameterSpec specifies the set of parameters used with password-based encryption (PBE), as defined in the PKCS #5 [7] standard. javax.crypto.spec.PBEKeySpec uses the PBE mechanism defined in PKCS #5 to consume the lower order 8 bits of each password character.

javax.crypto.SecretKeyFactory represents a factory for secret keys. A String with the standard name of the requested secret-key algorithm is passed to the getInstance method. The Java Cryptography Architecture [8] provides information about standard secret-key algorithm names. The method returns a SecretKeyFactory object for the specified secret-key algorithm.

The PBEKeySpec is passed to the generateSecret method. The method returns a SecretKey object from the provided key specification.

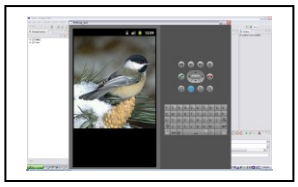
javax.crypto.Cipher provides the functionality of a cryptographic cipher for encryption and decryption. It forms the core of the Java Cryptographic Extension (JCE) framework. A String with the name of the transormation is passed to the getInstance method. The method returns a Cipher object that implements the requested transformation. The Cipher init method initializes the cipher for encryption.

android.content.Context is an abstract class whose implementation is provided by the Android system. It is an interface to global information about an application environment. The Context implementation class openFileOutput method is used to instantiate a java.io.FileOutputStream object and to open the file imsec.tmp for writing. The file creation mode MODE_PRIVATE specifies that the file imsec.tmp can only be accessed by the calling application (i.e., Imsec).

javax.crypto.CipherOutputStream is composed of the FileOutputStream and Cipher objects so that write methods first process data (i.e., encrypt it) before writing the data to the underlying FileOutputStream. This activity takes place when the android.graphics.Bitmap compress method is used.

The Bitmap compress method writes a compressed version of the Bitmap to the FileOutputStream. The first parameter specifies the format of the compressed image, in this case JPEG. The second parameter specifies the quality in the range [0,100] and is a hint to the compressor. 0 means compress for small size and 100 means compress for maximum quality.

For comparison, a source image [9] is shown in the Android emulator after encryption and compression (and subsequent decryption) in Fig. 6 with quality values of 100 (left) and 0 (right).



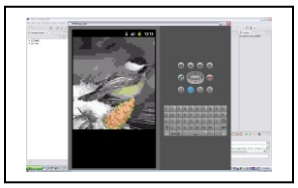


Figure 6. Quality of 100 and 0.

The quality value could be provided by the user. Selecting a low quality value would provide an addition to the model and would permit the user to email the image using the smartphone carrier network (i.e., no Wi-Fi).

E. Write Encrypted Image to Smartphone Database

After encryption and compression, the file imsec.tmp and the PasswordSalt_id are written to a SQLite smartphone database. The Java code for writing the encrypted image to the smartphone database is shown in Fig. 7.

```
FileInputStream fis;
fis = context.openFileInput("imsec.tmp");
bis = new BufferedInputStream(fis);
byte[] b = new byte[bis.available()];
bis.read(b);
cv = new ContentValues();
cv.put("name", "image name");
cv.put("passwordsalt_id", ps_id);
cv.put("in_out", "out");
cv.put("sent", "n");
cv.put("file", b);
dh = new DatabaseHelper(context);
SQLiteDatabase sdb;
sdb = dh.getWritableDatabase();
sdb.insert("images", null, cv);
```

Figure 7. Java code to write encrypted image to smartphone database.

The Context implementation class openFileInput method is used to instantiate a java.io.FileInputStream object and to open the file imsec.tmp for reading. java.io.BufferedInputStream read method is used to read FileInputStream into a byte array.

android.content.ContentValues is a map used to store a set of values. The put method takes a (key, value) pair and adds the value to the set. The key is a table column name and the value is a table column value.

The first put is an image name input by the user. The second put is the PasswordSalt_id. The third put designates whether this file is to be sent as an email attachment (out) or was received from an email attachment (in). The fourth put gives the sent status of an image (n is not sent, y is sent). The fifth put is the image now stored in an array of bytes.

DatabaseHelper is an inline class that extends android.database.sqlite.SQLiteOpenHelper, which is a helper class used to manage database creation and versioning. The getWritableDatabase method is used to open a database for writing and returns an android.database.sqlite.SQLiteDatabase object.

The SQLiteDatabase insert method inserts a row in the images database using the ContentValues map.

F. Email Encrypted Image

The Java code for emailing encrypted image(s) is shown in Fig. 8.

```
for(int i = 0;i < n;i++){
m = new Mail();
String[] to = {"email address"};
m.setTo(to);
m.setFrom("imsec@ualr.edu");
m.setSubject(image_id[i],passwordsalt_id);
m.setBody("imsec body");
m.addAttachment("imsec.tmp");
m.send();}
```

Figure 8. Java code to email the encrypted image.

The user waits until he is on a Wi-Fi network to email the encrypted images. The Wi-Fi network will allow Imsec to email high quality (and large file size) images in batch mode as quickly as possible. Note that this activity could be performed on the smartphone carrier network if the model is extended to allow the user to select low quality for images.

A query is issued to the smartphone database that returns a count of the number of unsent encrypted images: SELECT COUNT(*) AS n FROM images WHERE in_out='out' AND sent='n'. The user logs in to a web application to obtain his email address and a block of unique Image_ids (quantity n) over HTTPS. A unique Image_id will be used for the subject of every email. This ensures that, for all email addresses and inboxes, every email subject will uniquely identify one and only one encrypted image.

edu.ualr.imsec.mail.Mail extends javax.mail.Authenticator which is a class in javamail-android [10], a JavaMail port for Android. The setTo method is used to set the user email address. The setSubject method is used to set the Image_id that uniquely identifies this encrypted image. The PasswordSalt_id is read from the smartphone database and is appended to the subject. The resulting subject is of the form Image_id,PasswordSalt_id. The encrypted image is read from the smartphone database (see below) and written to the file imsec.tmp. The addAttachment method is used to attach the encrypted image to the email. The send method sends the email.

After the encrypted images have been emailed, the user may disconnect from Wi-Fi.

G. Download Encrypted Image

The Java code for downloading the encrypted image from an email message attachment is shown in Fig. 9.

```
    Store s = session.getStore("imaps");
    s.connect("imap.gmail.com","emailaddr","passw");
    Folder f = s.getFolder("INBOX");
    f.open(Folder.READ_ONLY);
    Message[] m = f.getMessages();
    for(int k = 0;k < n;k++){
     for(int i = 0;i < m.length;i++){
      String[] subject =
m[i].getSubject().split("\\,");
      if(image_id[k].equals(subject[0])){
       FileOutputStream fos;
       fos = context.openFileOutput("imsec.att");
       Object c = m[i].getContent();
       mp = (Multipart)c;
       for(int j=0;j < mp.getCount();j++){
        Part p = mp.getBodyPart(j);
        byte[] b = new byte[2048];
        InputStream is = p.getInputStream();
        while( (int r = is.read(b)) > 0 )
         fos.write(b, 0, r);
       //write image, subject[1] to database
    }}}}
```

Figure 9. Java code to download an encrypted image.

The user waits until he is on a Wi-Fi network to download the encrypted images. The Wi-Fi network will allow Imsec to download high quality (and large file size) images in batch mode as quickly as possible. Note that this activity could be performed on the smartphone carrier network if the model is extended to allow the user who emailed the encrypted images to select low quality for images.

The user logs in to a web application to obtain over HTTPS the email address and password of the user who created and emailed the encrypted images. The user also selects from a list of image names in order to obtain over HTTPS a block of unique Image_ids (quantity n). The email address, password and Image_ids are then used to download the encrypted images.

The javax.mail.Store object uses the Internet message access protocol (IMAP) to connect to the specified address using a simple authentication scheme that requires the user email address and password. The Store object instantiates a javax.mail.Folder object corresponding to the email address inbox. The Folder object opens in read only mode. The Folder object uses the getMessages method to get all javax.mail.Message objects from the Folder object.

The outer for loop with iteration variable k loops over all keys. The inner for loop with iteration variable i loops over all Message objects. The split method splits the message subject into an array where subject[0]=Image_id and subject[1]=PasswordSalt_id. If the current Image_id matches subject[0], then the user selected this encrypted image in the web application and the encrypted image is downloaded to the file imsec.att. Though not shown, the file imsec.att and the PasswordSalt_id (subject[1]) would also be written to the smartphone database with column in_out set to in.

After the selected encrypted images have been downloaded, the user may disconnect from Wi-Fi.

H. Read Encrypted Image from Smartphone Database

The Java code for reading the encrypted image from the smartphone database is shown in Fig. 10.

```
    dh = new DatabaseHelper(context);
    SQLiteDatabase sdb;
    sdb = dh.getReadableDatabase();
    Cursor c;
    c = sdb.query("images", null, "id=1", null, null,
null, null);
    c.moveToNext();
    byte[] b = c.getBlob(4);
    FileOutputStream fos;
    fos = context.openFileOutput("imsec.dbf",
Context.MODE_PRIVATE);
    fos.write(b);
```

Figure 10. Java code to read encrypted image from smartphone database.

The user has downloaded all selected encrypted images and is ready to view the decrypted images. The first step in this process is to read the encrypted image from the smartphone database and write the encrypted image to a file.

The getReadableDatabase method is used to open a database for reading and returns a SQLiteDatabase object. The SQLiteDatabase object uses the query method to read the images table: SELECT * FROM images WHERE id=1. The query method returns an android.database.Cursor object.

The Cursor object uses the moveToNext method to move to the next row in the result set. The Cursor object uses the getBlob method to copy the encrypted image from the fourth column in the result set to a byte array. The FileOutputStream object writes the byte array to the file imsec.dbf.

I. Decryption

The Java code for decrypting the encrypted image is shown in Fig. 11.

```
    Cipher dc =
Cipher.getInstance("DES/CFB8/NoPadding");
    dc.init(Cipher.DECRYPT_MODE, sk, pps);
    FileInputStream fis;
    fis = context.openFileInput("imsec.dbf");
    cis = new CipherInputStream(fis, dc);
    bitmap = BitmapFactory.decodeStream(cis);
```

Figure 11. Decryption Java code.

The user logs in to a web application and uses the PasswordSalt_id to obtain the salt and password (of the user who created the encrypted images) over HTTPS. The same Java code that used the password, salt and iterationCount to prepare to compress and encrypt the bitmap is also used to decrypt the image to a bitmap. This code is not repeated in Fig. 11.

The Cipher object uses the getInstance method to generate a second Cipher object that implements the specified transformation. The Context implementation class openFileInput method is used to instantiate a java.io.FileInputStream object and to open the file imsec.dbf for reading.

javax.crypto.CipherInputStream is composed of the FileInputStream and Cipher objects so that read methods first process data (i.e., decrypt it) before reading the data to the underlying FileInputStream. This activity takes place when the android.graphics.BitmapFactory decodeStream method is used.

J. Display Decrypted Image

The Java code for displaying the decrypted image is shown in Fig. 12.

```
   Rect r1 = new Rect(0, 0, bitmap.getWidth(),
bitmap.getHeight());

   Rect r2 = new Rect(r1);

   r2.bottom = bitmap.getHeight() * r1.right /
bitmap.getWidth();

   r2.offset(0,(r1.bottom - r2.bottom)/2);

   canvas.drawBitmap(bitmap, null, r2, paint);
```

Figure 12. Java code to display decrypted image.

This code preserves the aspect ratio of the image. android.graphics.Rect creates a new rectangle, r1, where the width (of the image) is the X coordinate of the right side of the rectangle and the height (of the image) is the Y coordinate of the bottom of the rectangle. A second Rect constructor creates a rectangle, r2, initialized with the values in rectangle r1 (which is left unmodified). r2.offset offsets r2 by adding 0 to the rectangle's left and right coordinates and (r1.bottom – r2.bottom)/2 to the rectangle's top and bottom coordinates. android.graphics.Canvas uses the drawBitmap method to draw the specified bitmap, scaling/translating automatically to fill r2.

VII. RESULTS

We have used our Android application to prove that our model is correct. Our model and our Android application will defend against all possible smartphone image attacks.

If the smartphone is stolen and rooted, our encryption software will ensure that images on the smartphone cannot be viewed. Encryption also defends against the MITM attack conducted by the fictional hacker in the Introduction.

To defend against the third attack, our model dictates that the password and salt used for encryption and decryption are stored on a server database and accessed through a web application. We did not implement this defense in software. Android provides the Java classes to create an HTTPS web application.

Our model defends against the fourth attack by physically separating user information, for example password and salt, from encrypted images. User information is stored on database servers while encrypted images are stored on a mail server. Our email software has shown a mail server can be used for this purpose.

VIII. CONCLUSION

Smartphones are an excellent example of Commercially available Off-The-Shelf (COTS) hardware that can be readily adapted for military use. Soldier familiarity with these devices will ensure that applications developed for this platform will require minimal training. Our model can be used to develop such an application for secure intelligence gathering.

Additional work is to implement the smartphone web application described in our model. Future research is to analyze the performance, storage requirements and memory utilization of our Android application.

REFERENCES

[1] C. Nanjunda et al., “Robust Encryption for Secure Image Transmission Over Wireless Channels,” in 2005 IEEE Int. Conf. on Communications, 2005 © IEEE. doi: 10.1109/ICC.2005.1494554

[2] S. Kovacevic et al., “System for Secure Data Exchange in Telemedicine,” in 9th Int. Conf. on Telecommunications, 2007 © IEEE. doi: 10.1109/CONTEL.2007.381881

[3] T. Shankar et al., “Image Encryption for Mobile Devices,” in 2010 IEEE Int. Conf. on Communication Control and Computing Technologies, 2010 © IEEE. doi: 10.1109/ICCCCT.2010.5670766

[4] W. Zhang et al., “Research on Image-Text Encryption Techniques in Mobile Communications,” in 2010 Second WRI Global Congr. on Intelligent Systems, 2010 © IEEE. doi: 10.1109/GCIS.2010.184

[5] C. Newman. (1999, June). *Network Working Group, Request for Comments: 2595, Category: Standards Track, Using TLS with IMAP, POP3 and ACAP* [Online]. Available: http://www.ietf.org/rfc/rfc2595.txt

[6] T. Gibara. (2010, May). *Live Camera Previews in Android* [Online]. Available: http://www.tomgibara.com/android/camera-source

[7] RSA Laboratories. (2006). *PKCS #5 v2.1: Password-Based Cryptography Standard* [Online]. Available FTP: ftp.rsasecurity.com Directory: pub/pkcs/pkcs-5v2/pkcs5v2_1.pdf

[8] Oracle. *Java Cryptography Architecture (JCA) Reference Guide* [Online]. Available: http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

[9] Bird Houses by Mark. *Chickadee* [Online]. Available: http://www.birdhousesbymark.com/images/chickadee.jpg

[10] *javamail-android* [Online]. Available: http://code.google.com/p/javamail-android/