Protecting Stored Data on Android Devices

Raluca-Alina COJOCARU
IT&C Security Master Program
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
ROMANIA
jhoalline@yahoo.com

Abstract: According to the latest statistics, the Android has become the world’s most popular operating system used for smartphones. Not only its popularity continues to increase, but also the number of malicious hackers who want to compromise mobile applications for their own gain. This paper reviews the protections offered by the Android Platform for storing application data, and provides means to enforce them.

Key-Words: permission, sandbox, isolation, cryptography, privacy.

1. Introduction

The Android operating system is organized as a stack of different layers running on top of each other, each layer providing services to the layer sitting above it. The heart of the Android OS is the Linux 2.6 kernel modified by Google to add some further architectural features. This layer enforces some of the basic Android separation between applications. Above this layer sits the native libraries and Android Runtime. They are modules of code which are compiled down to native machine code and run outside of the Dalvik VM, directly on the processor within the Android device. These native libraries run as processes within the underlying Linux kernel. Also running as a process within Linux Kernel is the Android Runtime. It consists of Dalvik Virtual Machine and Core Java Libraries. The Dalvik VM is a mobile-optimized virtual machine designed to run fast in low resource environments. The Dalvik VM allows multiple instances of virtual machines to be created simultaneously, providing security, isolation, memory management and threading support. Each application runs in its own instance of Android Runtime which contains both an instance on Dalvik VM and a set of the Android Core Libraries. [1]

1.1 Linux security model

The foundation of the Android platform is the Linux kernel and much of its security is a result of this fact. Central to Linux security is the concept of users and groups. Each user in a Linux system is assigned a unique user ID (UID) when it is created. Users can be added to groups and to each group a unique ID (GID) is assigned.

Each resource in a Linux system has an owner, identified by an UID. The owner has primary responsibility for that resource and can alter permissions on it. The resource has a group defined, which is the GID of the group of users having permissions over it.

A Linux resource has three sets of permissions: owner - that apply only to its owner, group - that apply to its group of users, and world – that apply to anyone that is not the owner or is not in the group that has defined permissions over it. Each set of permissions can include these rights: Read – which allows to read the content of the file, Write - which allows to write or update the file, Execute - which allows to execute the code.

The Principle of Least Privilege applies to Linux, stating that entities have only the minimum access to resources they need. Thus, an entity having a read permission does not imply it has a write permission, or vice versa. So not having granted a certain permission to an entity means that it does not have it.
At install time, the Linux creates a unique per-package UID and assigns it to the newly installed app. The app will run under this UID and it will be the owner of all the app resources. Linux prevents different applications having different UIDs from accessing the process or memory of other apps, providing the basis for the separation between applications. Android also provides a Unix-based mechanism for isolating app files. At install time Android creates for the app a directory under the path /data/data/App_package_name. Android sets the app’s UID as the owner of this directory. A set of owner permissions is created and assigned to this folder. No group or global permissions are configured. Inside the app’s directory Android creates some other directories, such as /file or /database and all the resources the app creates are placed in the appropriate folder. Each file the app creates is private unless the developer does not declare it public explicitly.

1.2 Android permissions

Android isolates apps to prevent them to use other app resources, but sometimes they need to share data to fulfill their jobs. To allow these changes Android created a permissions request model. An app can access other app data or services if it is granted the appropriate permission set. At install-time the user is prompted the list of permissions the app claims it needs. He needs to read each permission description, understand it and make an informed decision whether to accept them or not. The permission request model is an “all or nothing” process: either the user accepts the list of all permissions the application needs and the application is then installed, or the user refuses to grant them and the installation fails. Android permission request model has two main goals: to inform the user of all the sensitive and dangerous things the app can do before the application is installed and to allow the containment of an attack on a legitimate application [1]. The user has a very important role as he has to read the permissions and understand them. The protection offered by the Android permissions model is effective if and only if this happens. Android app developers have also an important role as they have to minimize the number of permissions requested by the app, not to fatigue the user, and not to give the app more privileges that it needs [1].

2. Protecting app stored data

All of the protections described previously rely on the mechanisms of the running Android system. They assume that the system is up and running and all accesses to stored data will be through these controlled channels. But this is not always the case. The basic access control provided by the OS is not always enough to protect against the risk of compromise. Depending on how sensitive is the data the application has to store and manipulate, further protection mechanisms should be used. It may include not at all sensitive data such as the game scores of a game, or very sensitive data as user’s credential for accessing his Bank page. One common approach to prevent sensitive or personal data disclosure would be to minimize the use of applications that access this data. But this is not enough when dealing with very sensitive data. Another tip would be not to store your personal data on mass storage, as an SD card, because no special protection is offered to this type of storage, it can even bypass the Linux Kernel basic isolation.

Another threat to disclosure of user’s personal data is when the user has full root access to the system. This allows complete and unrestricted access to the memory and storage while the device is running. The isolation provided to files, databases, and other resources simply does not exist when other applications can run with the root identity.

2.1 Defense in depth principle

Depending on the sensitivity of the user’s data multiple layers of security controls
which reinforce each other should be used. For example, if the Android system is working properly, offering all the protections described so far and if proper permissions set is requested on data, other apps not having these permissions will not be able to access that data. Otherwise, as in the case of rooted devices, where all OS protections are bypassed by any application running as a super user, sensitive data should also be scrambled, for example hashed or encrypted, so that an unauthorized application gaining access to it will not be able to do anything with it.

The protection mechanisms applied to data must be chosen according to the risk of compromising that data. It is very important before applying any protection mechanisms to carefully consider what type of data manages the app.

### 2.2 Using Cryptography

In addition to providing data isolation, supporting full-file system encryption, and providing secure communications channels, Android provides a wide array of algorithms for protecting data using cryptography.

The cryptographic algorithm, used either for symmetric encryption or asymmetric encryption, should be a standard implementation. It is not best practice at all for a developer to use his own cryptographic algorithm.

Encryption is a cryptographic operation that is used to provide confidentiality for sensitive information [7]. They use one key both for encrypting and decrypting data. Modern symmetric encryption algorithms work very fast to encrypt and decrypt data and they offer a very good level of protection. The only concern with them is to safeguard the key. If the key get lost or compromised the messages cannot be recovered.

Today’s NIST recommendation for symmetric key encryption is the AES algorithm (Advanced Encryption Standard) with a different key strength. The strength of the symmetric encryption is generally based upon the length of the key used. The longer the key, the stronger the protection offered by the encryption. As for 2011, the accepted sizes for AES are AES-128, AES-192, AES-256 and three-key Triple DES [7].

Asymmetric key encryption uses two different keys: one for encryption and the other for decryption. The keys have a special mathematical relationship that allows messages encrypted with one key to be decrypted by the other.

The strength of the asymmetric encryption is also based on the length of the key used. The key lengths are not comparable between asymmetric and symmetric algorithms; RSA keys generally need to be at least 2048 bits long to offer protection comparable to that of AES-256. Asymmetric algorithms are very slow compared to symmetric algorithms, so the data being encrypted or decrypted should be minimized [7].

A hash function is a function that takes a message of any length and produces a fixed-size hash called a digest. The digest looks like a random sequence of bits. SHA-256 is a popular hashing function that produces 256 bit digests, recommended by NIST [7]. The digest can be seen as a digital fingerprint of the message.

The hash functions are one-way functions, as you can go one way (message to digest), but you cannot go back (digest to message).

Hash functions can be used for example to validate the user's credentials. Instead of directly storing user’s credentials which are very sensitive data, we can hash them and store the digest. Whenever the user is prompted to fill his password, the hash of the newly typed password is computed and the resulting digest in compared with the one we stored. If they match then the user is authenticated. Another way to validate user’s credentials is using encryption. An encrypted value of the password is stored. When a user tries to authenticate, this value is decrypted and compared with the user’s input. If they match the user must have supplied the correct password.

Hash functions make it possible to securely store and validate user’s credentials. If some unauthorized person manages to get the stored credentials, he
can get no knowledge about the password.

3. Android Privacy App for protecting user’s logs

The goal of the Privacy Application is to provide the user a way to protect its SMSs and call logs in a private space. The user should authenticate itself each time he uses the application. He should provide at application startup a username and password. Once authenticated, the user will have the possibility to see his personal logs and to apply other preferences and restrictions with respect to a phone number or another entity. The application makes use of cryptographic algorithms for user’s authentication and for encrypting user’s private data.

To fulfill its job the app has to be granted a list of uses-permissions. Most of the permissions required by the application are related to its main job: listening for phone calls and SMS events. These system permissions are requested to the user at install-time to be granted.

The READ_PHONE_STATE permission enables the application to receive an Intent which notifies the application that a telephony event has occurred. An application Broadcast Receiver with a very high priority (it guarantees that that application is the first informed about the occurrence of an event) listens for this type of notifications. Further information about the Telephony state can be read from the Telephony Manager.

```xml
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

The RECEIVE_SMS permission, as the READ_PHONE_STATE permission, enables the application to be notified when a message has been received.

```xml
<uses-permission android:name="android.permission.RECEIVE_SMS" />
```

In Android almost all the application’s components should be declared and configured in the AndroidManifest.xml file. The application defines a list of Activity components to deliver to the user as UI content. In Android, each application which displays some graphic components should have a launcher activity that is first displayed when the application starts. In our case, the start-up activity, as defined in the manifest, is SplashScreenActivity.

```xml
<activity android:name="com.ism.privacyapp.SplashscreenActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

SplashscreenActivity is a simple activity which displays the applications logo. It is displayed for a few seconds, then the LoginActivity shows. LoginActivity is the screen where the user must authenticate.

```xml
<activity android:name="com.ism.privacyapp.LoginActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

READ_CONTACTS permission allows the application to access the list of phone contacts. This allows the user to add a contact in his personal list.

```xml
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

```xml
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```
The user must fill his username and password, and if he is logging in for the first time he should check the "I'm new" option. The password should comply with the password policy which the user can review.

Each time the user tries to sign in, the password is first verified. If the password does not respect the policies, an error is displayed. Password validation is achieved by the PasswordValidator class using a regex pattern.

```java
private static final String PASSWORD_PATTERN = "((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).{6,20})";
```

If the user is logging in for the first time his credentials are stored into the application’s database. The credentials are not stored as plain text, they are first applied a hash function and the resulting digests, for both the username and password, are stored into the database. Next time the user signs in he types his credentials. To the new values the same hash algorithm is applied and the resulting digests are compared against those stored into the database.

The hash algorithm used is declared in the /assets/security.properties together with the other security parameters. Once authenticated, the user is passed to the application space. It's now when the encryption key is derived starting from the user's password.

The key will be used for encrypting and decrypting data throughout all the application. It is not best practice to hardcode the encryption key into the application because the application can be reversed engineered. There are multiple tools that can be used to recover all of an application’s content starting from the APK file.

Android offers a mechanism for key derivation starting from the password. The user will need to provide its password for deriving the key whenever the key must be used. The key derivation is achieved by using a key derivation function (KDF) designed for Password-based encryption (PBE). KD process is composed of multiple iterations of hashing, performed on the password combined with a salt value. The resulting value is then used as a key for subsequent encryption computations.

The PBE algorithm, as declared below, uses SHA-256 for hashing and AES for encryption in CBC mode. It uses 1000 iterations of hashing together with a salt value to be concatenated to the value being hashed. The algorithm will produce a 256-bits key.

```
# Hashing algorithm used throughout the project.
hashAlgorithm = SHA-256
```

```
# The algorithms and parameters we use for key derivation (KDF - Key Derivation Function), designed for Password Based Encryption (PBE)
# - hashing algorithm : SHA-256
```
# encryption algorithm : AES
# encryption mode : CBC

PBE_keyDerivAlgorithm = PBEWithSHA256And256BitAES-CBC-BC

# Number or iterations of the hashing function during the KD process
PBE_iterations = 1000

# Symmetric key generated with KDF of 256-bit
PBE_keySize = 256

# Salt used for the key derivation with PBG algorithm
PBE_salt = ISM security master

To obtain an encryption key using KDF:

```java
PBEKeySpec pbeKeySpec = new PBEKeySpec(
    password.toCharArray(),
    mSalt.getBytes(mEncodingForma, mIterations, mKeySize);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(mPBAlgorithm);
SecretKey tempKey = keyFactory.generateSecret(pbeKeySpec);
SecretKey secretKey = new SecretKeySpec(tempKey.getEncoded(), "AES");
```

Other important components used by the application are:

- **PhoneStateReceiver** broadcast receiver, configured to receive all Telephony events, as described before. Each telephony state update event is broadcasted by the system as an Intent (IPC). The intent action broadcasted is android.intent.action.PHONE_STATE. Some common types of events are:
  - **CALL_STATE_RINGING** - ringing, a new call arrived and is ringing or waiting. In the latter case, another call is already active.

The manifest definition for PhoneStateReceiver:

```xml
<receiver android:name="com.ism.privacyapp.receiver.PhoneStateReceiver" >
<intent-filter
android:priority="999">
<action
android:name="android.intent.action.PHONE_STATE" />
</intent-filter>
</receiver>
```

**SMSReceiver** broadcast receiver, configured to listen to all messages received in the device.

```xml
<receiver android:name="com.ism.privacyapp.receiver.SMSReceiver">
<intent-filter
android:priority="999">
<action
android:name="android.provider.Telephony.SMS_RECEIVED" />
</intent-filter>
</receiver>
```

### 4. Conclusion

According to the latest statistics the Android has become the world’s most popular operating system used for smartphones. Not only its popularity continues to increase, but also the number of malicious hackers who want to compromise mobile applications for their own gain.

Android offers some very strong protections for their apps, such as app sandboxing, secure inter-process communication, a strong permissions model, file system isolation and cryptography.

Unfortunately, all of these protections can be bypassed in some situations, for example by rooted systems. To avoid possible data disclosure in such cases, the app has to use other protection...
mechanisms to enforce those offered by the Android OS. One such approach is to use cryptography.

Acknowledgement

Parts of this research have been published in the Proceedings of the 7th International Conference on Security for Information Technology and Communications, SECITC 2013.

References