KeyChain Extension and Integration Final Report

Practical Lab on Smartphone Security

Kjell Braden, Marvin Dickhaus, Cassius Puodzius

Winter term 2012/2013
February 10, 2013

The goal of this project is to extend the built-in key storage of Android 4.2 to support functionality such as signing and decrypting without revealing the needed key to any app.
The second goal is the integration of system apps to demonstrate the functionality. This group implemented symmetrically encrypted SMS for that purpose.

Contents

1 Introduction

Since Android API Level[1] 14 (4.0 Ice Cream Sandwich) the `KeyChain`[2] class exists in Android. The `KeyChain` class allows storage of asymmetric keys in a secure credential storage.

Whenever in the default implementation an authorized app calls to retrieve the private key, it'll get it. Our goal was to extend the given functionality further with a crypto oracle, as well as support for different key types, both symmetric and asymmetric. With that functionality, apps should call the API with the data they want en-/decrypted or signed/verified. In the process of en-/decryption and signing the API would return the respective byte data. With verifying the API would return a boolean indicating verification success or failure.

Besides the new API, we wanted to alter at least one system app to comply with our enhanced KeyChain. After suggestions from Sven Bugiel (our supervisor) we decided to implement symmetric encryption with the SMS app in lieu of the E-Mail app. More on that in section 5.

2 Initial Situation

At the starting point, we were confronted with a quite unfinished, not wholly tested API. Furthermore the KeyChain API lacks in documentation. So we had to figure out in what way it would be possible to implement our functionality. The following listing gives an overview of the need-to-know classes for this project:

KeyChain is the Android API for importing PKCS#12 containers (private keys, public key certificates and CA certificates) and providing grant-based access of the keys to apps. The PKCS#12 format requires the container to be encrypted.

- When an import is requested the API creates an Intent, which is handled in the system app `com.android.certinstaller` (source at `packages/apps/CertInstaller/CertInstaller.java` et al.). CertInstaller spawns a dialog and handles the container decryption process as well as stores the keys in the `keystore` daemon (see below).
- Keys are identified by aliases, which are chosen by the user when the CertInstaller is invoked with an import request.

[1] `http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels`
[2] `http://developer.android.com/reference/android/security/KeyChain.html`

- Private as well as public keys (in the form of certificates) can be requested:
  - If access to a private key was granted to the requesting app, it can be retrieved using `getPrivateKey()`.
  - Public keys can be retrieved through `getCertificateChain()`
- `KeyChain` also contains `AndroidKeyPairGenerator` since Android 4.2, which generates and stores key pairs in `keystore` automatically.

KeyChain app is a system application implementing most parts of the `KeyChain` API. It grants key access to apps and retrieves keys from the secure native `keystore` daemon (see below).

keystore is the native daemon (written in C++) that holds encrypted key information.

- The storage is encrypted with a master key which is derived from the unlock passphrase, PIN or pattern of the device[3].
- Unlocks on the *first successful unlock attempt* of the device, won't lock again *until the phone is powered off*. This means the keystore is protected for example from rooting, but not from live-debugging. (It is possible for apps to lock it manually, though.)
- The service supports two storage types:

  key is a RSA key pair. Once stored the private key cannot be exported again, more on this later.

  blob can be arbitrary data.
- `keystore` provides an OpenSSL engine called `keystore`, which should be able to retrieve PrivateKey objects from the daemon directly in Java code.

CertInstaller The `CertInstaller` is a system app that lets a user import and install key pairs and certificates from PKCS#12 container format files.

Detour: Java Cryptography Architecture (JCA)

The JCA is a *provider framework* for common cryptography operations such as hashing, sign/verify, encrypt/decrypt and key/certificate generation. Each of these operations have an algorithm- and implementation-independent interface, most of which operates on objects implementing the `javax.crypto.SecretKey`, `java.security.PublicKey` or `java.security.PrivateKey` interfaces, depending on the context.

This makes it possible for crypto code to be written largely independent on actual implementation e.g. of the keys.

[3] In order to import and use certificates that are not in the trusted certificates from the Android base (known as user certificates), respectively use the `keystore`, a PIN, passphrase or pattern has to be created, that will unlock the device.

3 Requirements

To achieve our goal, we need to be able to securely store and generate both symmetric and asymmetric keys, regardless of the key type.

Apps should be able to do crypto operations with specific keys. In order to access those keys, the user should be required to give his permission. The operations necessary can be distinguished by the key type.

- For private keys, operations include decryption and signing of data.
- For public keys it's encryption and verification.
- With symmetric keys, apps need to be able to both en-/decrypt, as well as sign and verify, which means generating and checking *message authentication codes (MAC)* in this case.

These operations must not leak any kind of private key parameter or any part of the symmetric key used.

For convenience, it would be a good idea to create and store key exchange (eg. Diffie-Hellman) parameters as well, and to automatically derive and store the resulting symmetric key on receiving the remote's parameters.

3.1 Workflow with the existing KeyChain API

In the KeyChain API there is `AndroidKeyPairGenerator`, `AndroidKeyStore` and `KeyChain`.

The `AndroidKeyPairGenerator` implements JCA's `KeyPairGenerator` interface and sends a request to generate a key to the `keystore` daemon. This way, the keys' types are hard coded to *2048 bit RSA*. The `AndroidKeyStore` implements the `KeyStore` interface. It retrieves the key references from the daemon, but does no permission checking. If the requesting app has never been granted to access the requested key, the access simply fails. Additionally, it only supports only asymmetric keys.

Key references means that the returned key objects don't carry the cryptographic parameters themselves, but instead they are handles that the OpenSSL JCA provider can use to run the crypto operations on the keys in the `keystore`, so the Java code (and specifically, the app's code) never sees any crypto parameters at all. The `KeyChain` allows to retrieve `PrivateKey` using the `AndroidKeyStore`, but requests permission to access a key beforehand. The user will see a dialog telling him which app tries to access a key, can choose the one he wants to use and the app gets access to that granted key.

3.2 Proposed Implementation Ideas

3.2.1 Reuse KeyChain API

Our initial plan was to reuse as much of the KeyChain API as possible. Nevertheless we would need to add support for symmetric keys and other asymmetric key types in both storage and key generation as well as support for key exchange handling.

This turned out to be harder than we thought, as we would need to rewrite significant amount of C++ code in the `keystore` daemon for supporting anything else than RSA. Also we would have to rewrite the protocol the `keystore` uses to communicate with the Java framework.

On top of that, there turns out to be a bug somewhere in Android's OpenSSL engine, which causes *dalvik* to *segfault* as soon as encryption or decryption is being run with the returned key references.

3.2.2 The manual way

After various attempts to reuse existing `KeyChain` code, we got to a working solution by doing everything in our own code and store the keys as blobs in the `keystore`. This means:

1. Manually generate the key pair, manually store it in the `keystore` as a PKCS#8 (not PKCS#12) encoded blob and the certificate as PEM encoded blob.
2. For encryption, manually load the certificate from `keystore`, parse it, feed it to `java.security.Cipher`.
3. For decryption, manually load the private key from `keystore`, parse it, feed it to `java.security.Cipher`.
4. For symmetric crypto, we simply store the secret key bytes as blobs in `keystore`.

4 Implementing the framework

The framework respectively the back end consists of the `KeyChain` app for all the crypto operations on the one hand and the management of the keys on the other hand.

4.1 Key Chain

To reduce responsibility of the apps it makes sense to let the system app (`KeyChain`) handle everything related to key management:

1. Generate keys
2. Delete keys
3. Import/Export of key *pairs*[4]
4. Grant key access to apps

Our first sketch didn't quite reflect symmetric encryption, so there was some more work due in refining our API, with the final result being as follows.

[4]Users should be able to export their keys for use on other devices.

1. `encryptData(keyId, algorithm, padding, plaintext, IV)` → `ciphertext`
2. `decryptData(keyId, algorithm, padding, ciphertext, IV)` → `plaintext`
3. `sign(keyId, algorithm, plaintext)` → `signature`
4. `verify(keyId, algorithm, plaintext, signature)` → `isValid`
5. `storePublicCertificate(keyId, certificate)` → `void`
6. `generateSymmetricKey(keyId, algorithm, keysize)` → `void`
7. `retrieveSymmetricKey(keyId, algorithm)` → `key`
8. `importSymmetricKey(keyId, key)` → `void`
9. `deleteSymmetricKey(keyId)` → `void`
10. `mac(keyId, algorithm, plaintext)` → `mac`
11. `generateKeyPair(keyId, algorithm, keysize)` → `publicKey`
12. `keyAgreementPhase(keyId, keyAlgorithm, agreementAlgorithm, publicKey, isLastPhase)` → `publicKey`

Keys itself are referenced by an string alias (*keyId*). Every key has exactly one alias assigned in the `keystore`. This alias must be unique.

The functions `encrypt` and `decrypt` are used for symmetric as well as for asymmetric crypto.

The functions `sign`, `verify` and `storePublicCertificate` are exclusive to asymmetric crypto.

`generateKeyPair` and `keyAgreementPhase` are intended for key agreement protocols like the Diffie-Hellman key exchange. This way, you can create key agreement parameters (which are handled as `PrivateKey` and `PublicKey` objects), transmit them to your communication partner. You then feed in the received parameters using `keyAgreementPhase` which can return another set of parameters if the protocol is not finished yet, or, if the protocol is finished, it will immediately derive the shared secret key and store it under the alias previously used by the key exchange parameters.

The other functions are just for symmetric crypto operations.

The API is made available to apps from the framework in the `android.security` package, but every call is actually forwarded using the Binder IPC to the `KeyChain` app which runs with system permissions and is allowed to do any operation on the `keystore` daemon. The `KeyChain` app is therefore responsible for checking if requesting apps are allowed to access specific keys before obeying to their requests.

4.2 Key Management App

In addition to all the new crypto operations, keys have to be assigned to contacts. This is realized in the new *Key Management* system app. After a key was created or imported, it can be assigned to contacts. Each assignment gets a new string identifier (called *usage type*) which is arbitrary but application defined. For example a crypto e-mail could get *mail-crypt* as usage type. Now whenever an encrypted e-mail is sent to a contact, the program only has to check if a key with this usage type is present. If not, no key for this usage is defined and so encryption can't happen. This way the user can decide which key to use for which application, and easily replace keys once they are obsolete.

Each usage type has to be unique per contact. It is also possible to assign the same key more than once to a contact, provided that the usage type is different each time. For example, a user could use the same key for e-mail and SMS encryption.

The key assignments are stored with a MIME type of `vnd.android.cursor.item/key` in the `ContactsContract.Data` table of the *Android contacts provider*[5].

5 Integrating the SMS app with our API

5.1 Considerations

At first we thought about integrating asymmetric cryptography with the standard e-mail app. Sven Bugiel brought to our attention that extending the default e-mail app would be quite complicated and would bear a huge overhead. Instead he suggested to take on SMS as this would be more feasible. As asymmetric crypto bears a huge overhead for SMS (in terms of message size and therefore cost per SMS) we decided to implement symmetric cryptography with the SMS app instead.

In this scenario, we assumed that the symmetric key would already be exchanged with the other party. So we just had to handle sending and receiving SMS.

The greatest inconvenience with system apps is that one first has to build the image once you developed something. As compiling Android even on powerful systems takes hours, we decided against altering the system app and instead overlaying it. This means:

- For writing encrypted messages, a separate UI is needed.
- For receiving encrypted messages, broadcasts need to be captured.
- In every case, the unencrypted messages are stored in the communication log with the respective contact.

This also means, other than when sending encrypted SMS, the user doesn't have to differ from his known behavior.

[5] `http://developer.android.com/guide/topics/providers/contacts-provider.html`

5.2 Implementation Details

5.2.1 Sending

Since outgoing messages can't be intercepted and modified, there needs to be separate custom activity for composing encrypted messages. The layout is very simple. It consists out of a contact field, a text field and a send button. See figure 1.

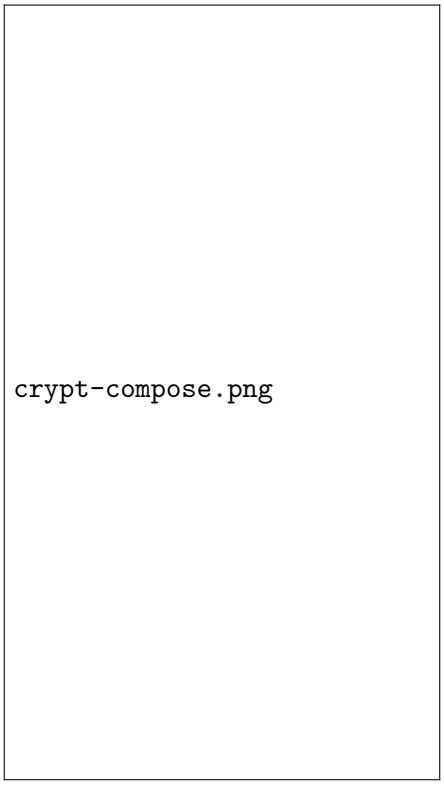

Figure 1: The layout of encrypted SMS composing.

Contacts are only selectable if they have a key with the *usage type* of `sms-symmetric` defined. After composing is finished, the user sends the SMS with the send button. The text will then be encrypted with the given symmetric key and encoded in Base64. To identify encrypted SMS when they are received, the @-character will be appended at the beginning and the end.

Outgoing messages are stored in the (undocumented) SMS content provider `content://sms`, so they can appear in the regular conversation history and receive status updates (for example a delivery notification). Then they are encrypted and sent, in multiple parts, if necessary, using the Android API `SmsManager`.

5.2.2 Receiving

For receiving SMS, apps need to intercept the `SMS_RECEIVED` broadcast with a high priority, before any other app can handle them. After being processed a notification is shown and the decrypted message will be stored in the SMS content provider as well.

Once the user has assigned a `sms-symmetric` key to a contact, decrypting SMS woks without any interference of the user. Basically, the steps of decrypting are reverse to encrypting. If a broadcast for a new incoming SMS is received, then:

1. check if the SMS is encrypted after all parts were received (@-char at the beginning and end)
2. check if the Sender has a key assigned, if yes then
3. abort broadcast
4. decrypt SMS
5. store decrypted SMS in the conversation log.

6 Key Exchange Scenarios

Since apps running as the system user (ie. with user id 1000) are not generally allowed to write to the sd card, exporting keys from the key management app is implemented by using a `ACTION_SEND` intent with a MIME type of `*/*`. This way, the operating system automatically offers to send the key file for example as an attachment via e-mail or via bluetooth. It should be easy to extend this functionality to Near Field Communication.

The receiving party receives the key file and can import it in their key management app. Before importing though, this should display key fingerprints (or any other hash visualization scheme[6]) to allow verification between both parties.

7 Responsibilities

8 Further development

As this project was intended as a proof-of-concept, there are quite a few angles, that leave room for improvement. Some ideas of further development are:

1. When symmetric cryptography is used, MACs have to be compared in user code. The KeyChain could provide a `verifyMac` method instead.
2. Currently when keys are exchanged between parties, the keys are exposed to non system apps. Instead, NFC could be used to exchange keys directly between two `keystore` instances and therefor never be exposed to the outside world.

[6] as discussed in [1]

3. The encrypted SMS app is missing some basic functionalities. Some enhancements that come into mind:
   a) Warn the user, if the encrypted SMS is longer than 160 characters, because it cost more money to send them.
   b) Implement a character counter, that encrypts the text on-the-fly and displays used characters of the encrypted SMS.
4. A further security consideration would be to store the SMS encrypted on the phone. Of course there are several possible ways to do so.
   a) Encrypt SMS with the master key derived from the pin/pattern that locks the phone. This would be the way, the keystore is encrypted, too.
   b) Encrypt with a application specific pin.

References

[1] Adrian Perrig and Dawn Song, *Hash Visualization: a New Technique to improve Real-World Security*. 1999.