KeyChain Extension and Integration Midterm Report

Practical Lab on Smartphone Security

Kjell Braden, Marvin Dickhaus, Cassius Puodzius

Winter term 2012/2013
December 13, 2012

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 like Contacts or E-Mail with the newly provided features.

Contents

1 Introduction

Since Android API Level[1] 14 (4.0 Ice Cream Sandwich) the KeyChain[2] class exists in Android. The KeyChain class provides access to private keys and their corresponding

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

certificate chains in credential storage.

Whenever in the current implementation an authorized app calls to retrieve the private key, it'll get it. Our goal is to extend the given functionality further with a crypto oracle. 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 string. With verifying the API would return a boolean indicating verification success.

Besides the new API, we want to alter at least one system app to comply with our enhanced KeyChain.

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.

2.1 Class Overview

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 allows the container to be encrypted and/or signed.

- 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.
- Private as well as public keys (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 deamon (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 keypair. 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 load private keys.

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

2.2 Proposed Implementation Ideas

Before we could get any real work done, we needed an angle on how we could implement our enhancements.

Using `keystore`'s own `key` storage format sounded promising, as it already applies non-exportability and provides sign/verify operations.

2.2.1 AndroidKeyPairGenerator & KeyChain

At first we tried to reuse as much code as possible.

1. Use `android.security.AndroidKeyPairGenerator` for key generation and storage.
2. Use `KeyChain.getCertificateChain()` and `java.security.Cipher` for encryption.
3. Use `KeyChain.getPrivateKey()` and `java.security.Cipher` for decryption.

The results were rather frustrating. `AndroidKeyPairGenerator` isn't really well developed as of yet.

- It's not registered as a `java.security.KeyPairGenerator` provider, so we either have to change the ROM or each app will have to register it on itself.

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

- It's not configurable in terms of private key parameters (key size, key type[4])

In conclusion: While we could live with the rest, fixed key size is not quite what we were hoping for. Also, for some reason, `KeyChain.getPrivateKey()` does not work - the `keystore` fails with `KEY_NOT_FOUND`.

Detour: Importing keys with CertInstaller

When importing a PKCS#12 file the `CertInstaller` works like this:

1. Ask the user for the container's password.
2. Decrypt the container & parse contents to `java.security.PrivateKey`.
3. Request an alias from the user to provide further identification of the key pair.
4. Re-encode the keys.
5. Send keys to `com.android.settings.CredentialStorage`, which in turn sends the key to the `keystore` daemon for storage.

2.2.2 KeyPairGenerator & CertInstaller

Instead of using the Android specific `KeyPairGenerator`, we could use the default `KeyPairGenerator`[5] provided by Java and send the key pair to `CertInstaller` to handle the storage.

Unfortunately, we were unable to skip the first two steps - we would have had to provide our keys as encrypted PKCS#12 containers. Since we already had the contents of the container (because we were generating it with `KeyPairGenerator`), it wouldn't make sense to pack and encrypt them, just to decrypt and unpack them again.

Thus we implemented step 3 to 5 on our own and let `CredentialStorage` take care of the storage[6], this left us with working key generation and storage.

The problem now is that `KeyChain.getPrivateKey()` won't work. It turns out that, the way `getPrivateKey()` works, it prepares the alias by checking some permissions and adding the system uid (`1000`) to the name. The `keystore` daemon does the same under some circumstances, this being one of them. In the end the `keystore` daemon tried to read `1000_1000_USRPKEY_alias` instead of `1000_USRPKEY_alias`.

Conclusion: We can't use `KeyChain.getPrivateKey()` either.

[4] At this point, the key type is hard coded to RSA by the `keystore` anyway. While we could store the keys as blobs to support more key formats, the current Android `java.security.Security` providers only support RSA for asymmetric encryption and RSA and DSA for signatures, so we wouldn't gain much.

[5] http://docs.oracle.com/javase/6/docs/api/java/security/KeyPairGenerator.html

[6] For this we had to move our code from the `KeyChain` app to the `CertInstaller` app because the CredentialStorage refuses to talk to anybody else.

2.2.3 CertInstaller & OpenSSL engine

The keystore offers a custom OpenSSL engine which provides access to `PrivateKey` objects backed by the storage.

1. Use our code in `CertInstaller` as outlined above.
2. For decryption, load private key directly from the `keystore` OpenSSL engine (like `KeyChain` does).
3. For encryption, load public key using `KeyChain.getCertificateChain()`

Observation: The *decryption crashes the process* (SIGSEGV, not a java exception)
Conclusion: `keystore`'s `key` type is broken. From the `keystore` code it seems to only allows access to the public key - it returns the public key data even when a private key was requested. The included OpenSSL Java-API parses this as a `PrivateKey` and returns it. Any private-key-crypto operation obviously crashes.

2.2.4 Do everything manually

Obviously none of these approaches worked out very well. As a result we're going to do 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`.

As a result, this is the only working constellation we could acquire.

3 Implementing the framework

Once we decided how to generate and store the keys, we needed to determine how our interface should look. 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*[7]
4. Grant key access to apps

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

We need the following functionality to be available to our app:

1. `encrypt(keyId, plaintext, blockCipherMode)` → `ciphertext`
2. `decrypt(keyId, ciphertext, blockCipherMode)` → `plaintext`
3. `sign(keyId, plaintext)` → `signature`
4. `verify(keyId, plaintext, signature)` → `isValid`
5. `requestKey()` → `keyId`
6. `storeCertificate(keyId, certificate)` → `void`[8]

Keys itself are referenced by an string alias.

On `requestKey()`, the `KeyChain` app displays a key selection dialog to the user, informing him which app requested it and giving him the option to generate a new key. Once confirmed, the requesting app is granted usage of the selected key.
We have implemented this API and an userspace app to test this basic functionality.

4 Perspective - or - Extend the system apps

In our further planning and development, we want to integrate the contacts, and stock e-mail app with native support for our crypt oracle. The steps for this purpose have yet to be evaluated.

[8]Existing methods in `android.security.KeyChain` can be used for retrieving the certificate / public key