

Exercises III:

Secure storage of persistent data

Mobile devices security

Table of Contents

Contents

| | |
|--|----------|
| 1 Introduction..... | 3 |
| 2 Android KeyStore provider..... | 4 |
| 3 Encrypting a DB with SQLCipher..... | 7 |
| 3.1 Obtaining SQLCipher and preparing the Android project..... | 7 |
| 3.2 Implementing SQLCipher..... | 8 |
| 3.3 Inspecting the contents of a database encrypted with SQLCipher..... | 8 |

1 Introduction

The goal of this section is to show the correct use of cryptography to securely store data in a device. We start by creating our own cryptography base, which will be implemented through a function to generate keys for symmetric encryption and latter storage of these keys by leveraging a system service called Android KeyStore.

Lastly, we will take a look at SQLCipher to assure that the databases used by the applications is encrypted, leading to a security increase in the application data.

2 Android KeyStore provider

In Android 4.3 a new facility to allow applications to store secret keys in the system key storage was added. The Android key storage restrict access to the keys only to the application that created it and is secured by the device PIN.

The Android key store is for certificates storage, so only public/private keys may be stored. Currently, symmetric keys, such as those for use with AES, cannot be stored. In Android 4.4 support for the Elliptic Curve Digital Signature Algorithm (ECDSA) was added to the key store. This section explains how to generate a new key, store and retrieve it using the Android key store.

Due to the fact that this functions was added in Android 4.3, make sure that the minimum version in the Android manifest is set to API 18.

Below are the steps to generate a key pair. The used code may be found in the link: <https://github.com/obaro/SimpleKeystoreApp>.

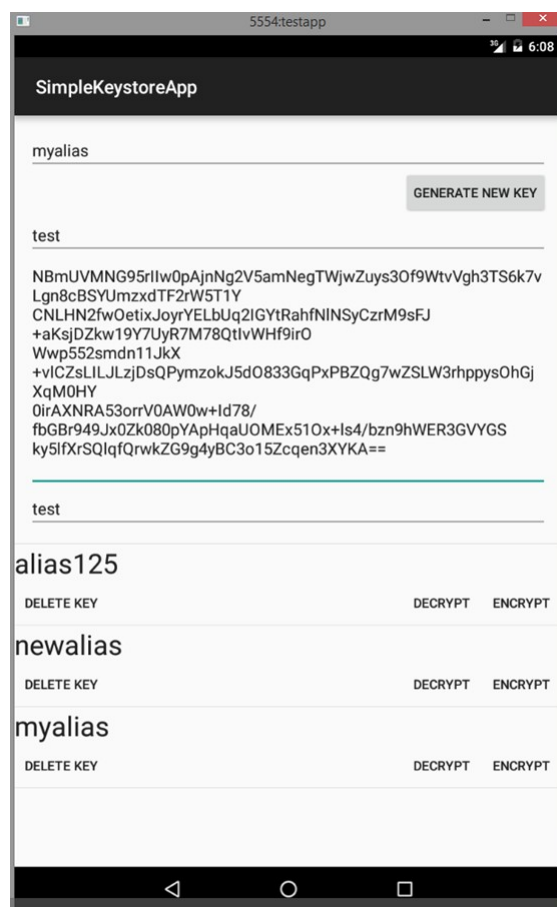


Image 2.1. Main screen

1. We create a handler for the KeyStore in our app:

```
public static final String ANDROID_KEYSTORE = "AndroidKeyStore";
public void loadKeyStore() {
    try {
        keyStore = KeyStore.getInstance(ANDROID_KEYSTORE);
        keyStore.load(null);
    } catch (Exception e) {
        // TODO: Handle this appropriately in your app
        e.printStackTrace();
    }
}
```

2. We generate and save the key pair:

```
public void generateNewKeyPair(String alias, Context context)
    throws Exception {
    Calendar start = Calendar.getInstance();
    Calendar end = Calendar.getInstance();
    // expires 1 year from today
    end.add(Calendar.YEAR, 1);
    KeyPairGeneratorSpec spec = new
    KeyPairGeneratorSpec.Builder(context)
        .setAlias(alias)
        .setSubject(new X500Principal("CN=" + alias))
        .setSerialNumber(BigInteger.TEN)
        .setStartDate(start.getTime())
        .setEndDate(end.getTime())
        .build();

    // use the Android keystore
    KeyPairGenerator gen =
    KeyPairGenerator.getInstance("RSA", ANDROID_KEYSTORE);
    gen.initialize(spec);

    // generates the keypair
    gen.generateKeyPair();
}
```

3. Next, we obtain the key with the given alias:

```
public PrivateKey loadPrivteKey(String alias) throws Exception
{ if (keyStore.isKeyEntry(alias)) {
    Log.e(TAG, "Could not find key alias: " + alias);
    return null;
}
    KeyStore.Entry entry = keyStore.getEntry(KEY_ALIAS, null);
    if (!(entry instanceof KeyStore.PrivateKeyEntry)) {
        Log.e(TAG, " alias: " + alias + " is not a PrivateKey");
        return null;
    }
    return ((KeyStore.PrivateKeyEntry) entry).getPrivateKey();
}
```

The KeyStore class has been since the API level 1. To access the new key store in Android we use the special constant: 'AndroidKeyStore'.

According to the Google documentation, there is an strange issue with the KeyStore class which requires to invoke the method load(null) even though we are not going to load the KeyStore from the input stream, if not done the application might be interrupted.

When generating the key pair, we populate an instance of KeyPairGeneratorSpec.Builder with the needed details, including and the alias which will be used to retrieve the key later on. In this example, we have create a validity period of 1 year from the current date and a default serial 'TEN'.

Loading the key given the alias key is as simple as keyStore.getEntry("alias", null); from here, what we do is a cast to PrivateKey so we may use it to encrypt / decrypt.

The KeyChain class API also was updated in Android 4.3 to allow the developers know if the device is compatible with the hardware certificates store or not. This means that the device is compatible with a security element to store certificates. This is an interesting improvement, because it allows to securely store certificates even if the device is in debug mode.

However, not all devices are compatible with this hardware feature. The LG Nexus 4, a popular device, uses the ARM TrustZone hardware protection.

Other relevant information:

- <https://developer.android.com/intl/es/training/articles/keystore.html>
- <https://developer.android.com/intl/es/reference/android/security/keystore/KeyGenParameterSpec.html>
- www.androidauthority.com/use-android-keystore-store-passwords-sensitive-information-623779/
- La clase KeyStore en la guía de referencia de Android para desarrolladores: <https://developer.android.com/reference/java/security/KeyStore.html>
- El ejemplo del API KeyStore en: <https://developer.android.com/samples/BasicAndroidKeyStore/index.html>
- El artículo de *Nikolay Elenkov* "The Credential storage enhancements in Android 4.3" en: <http://nelenkov.blogspot.co.uk/2013/08/credential-storageenhancements-android-43.html>
- ARM TrustZone en: <http://www.arm.com/products/processors/technologies/trustzone/index.php>

3 Encrypting a DB with SQLCipher

SQLCipher is a security extension to the SQLite database platform which facilitates the creation of ciphered databases. It uses the Codec API from SQLite to insert a call in the pagination system which interacts with the database immediately before a read or write in the storage.

The main characteristics of SQLCipher are:

- **Transparent:** When in use, the app doesn't need to know the internal security of the database. The applications use the standard API for SQLite to handle data.
- **"On the fly":** SQLCipher encrypts and decrypts blocks named pages as it is needed, it does not work with the whole database at once. This allows SQLCipher to:
 - Open and close the database rapidly
 - The performance is excellent, event with very big databases
 - Works with the indexing of SQLite (for example, a search using an index can last only a 5% longer than with an standard database)

All the documentation regarding SQLCipher may be found in the link:

<https://www.zetetic.net/sqlcipher/documentation/>

3.1 Obtaining SQLCipher and preparing the Android project

In order to use SQLCipher in an Android Studio project, we can add the dependency automatically by navigating to *"File > Project Structure > app > Dependencies"*, or else we could perform the following steps:

1. Download the library in the following link:
<http://repo1.maven.org/maven2/net/zetetic/android-database-sqlcipher/3.5.6/android-database-sqlcipher-3.5.6.aar>
2. Copy the file in the app/libs directory in our project created by Android Studio (select the 'project' view)
3. Add the library to the app/build.gradle file in the dependencies section:

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    testCompile 'junit:junit:4.12'  
    compile 'com.android.support:appcompat-v7:23.1.1'  
    compile 'net.zetetic:android-database-sqlcipher:3.5.6@aar'  
}
```

Code 3.1. build.gradle file

3.2 Implementing SQLCipher

Once we have configured the project so it includes the SQLCipher library, the following modifications are going to be performed in order to cipher the database.

1. We need to change all the import directives for handling the database:
 - Change `android.database.sqlite.*` for `net.sqlcipher.database.*`
2. Before creating or executing a select against the database is needed to load the library using:
 - `SQLiteDatabase.loadLibs(this);`

The API from `net.sqlcipher.database` and the default one for SQLite is the same except for the methods used to create and open the database, which needs an additional parameter. The parameter is the key used to encrypt the database.

It is left to the developer how the key used to encrypt is generated. It may be generated using a PRNG for each application or any other technique that adds more security introduced by the user.

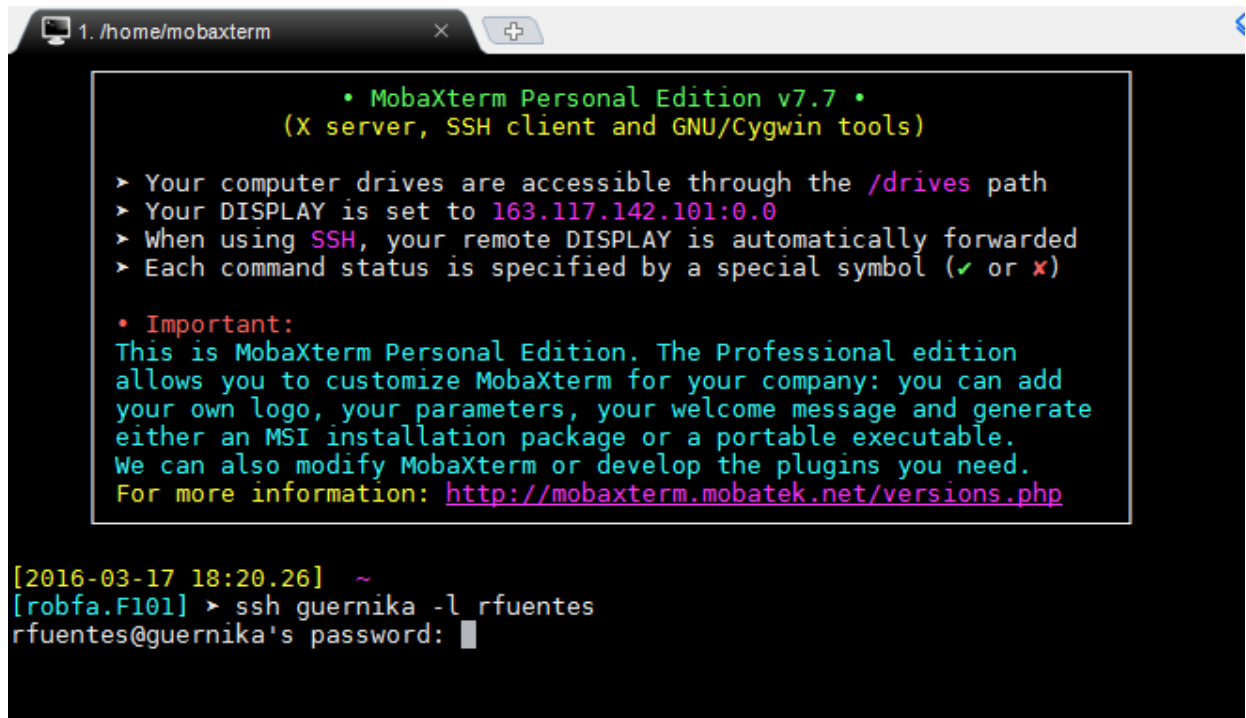
SQLCipher transparently encrypts and decrypts with the given key. It also makes usage of the Message Authentication Code (MAC) to assure both: confidentiality and authenticity; detecting if the data has been accidentally or maliciously modified.

3.3 Inspecting the contents of a database encrypted with SQLCipher

Due to the fact that a database created using SQLCipher is encrypted, we can no longer browse its contents using tools like `sqlite3`.

In order to browser the contents of a database created using SQLCipher in the lab computers we need to perform the next steps (we omit details from command used in previous modules):

1. Assuming that we are working with the Android emulator from Android Studio, we extract the file of the database to the PC using the adb command.
2. Once we have the file, we open the program 'MobaXterm Personal Edition'. We login with the lab user credentials in the server guernika.lab.inf.uc3m.es using the ssh protocol, as shown in the image below.



```
1. /home/mobaxterm

• MobaXterm Personal Edition v7.7 •
(X server, SSH client and GNU/Cygwin tools)

> Your computer drives are accessible through the /drives path
> Your DISPLAY is set to 163.117.142.101:0.0
> When using SSH, your remote DISPLAY is automatically forwarded
> Each command status is specified by a special symbol (✓ or ✗)

• Important:
This is MobaXterm Personal Edition. The Professional edition
allows you to customize MobaXterm for your company: you can add
your own logo, your parameters, your welcome message and generate
either an MSI installation package or a portable executable.
We can also modify MobaXterm or develop the plugins you need.
For more information: http://mobaxterm.mobatek.net/versions.php

[2016-03-17 18:20.26] ~
[robfa.F101] > ssh guernika -l rfuentes
rfuentes@guernika's password: █
```

Image 3.1. Login into guernika.lab.inf.u3m.es

3. We drag and drop the file to the desired location in the server, as shown in the next image:

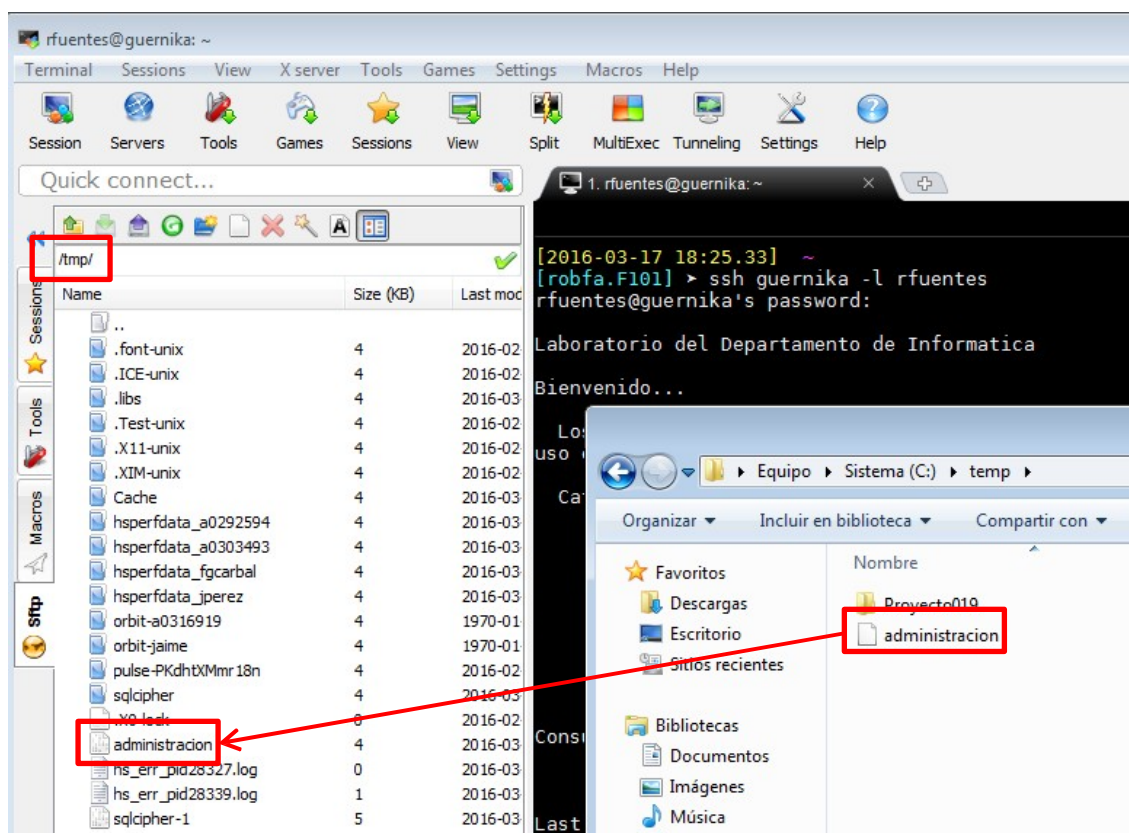


Image 3.2. File transfer to the Linux server

4. We execute the next command in the terminal to open and explore the encrypted database

```
rfuentes@guernika:~$ cd /tmp
rfuentes@guernika:/tmp$ sqlcipher
administracion SQLCipher version 3.8.10.2 2015-
05-20 18:17:19
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> PRAGMA key = 'contraseña_usada_para_cifrar';
sqlite> .schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE articulos(codigo int primary
key,descripcion text,precio real);
sqlite> select * FROM articulos;
1|adfadfad|12.0
2|adfadf|12.0
3|adfadf|12.0
4|pepe|100.0
sqlite>
```

Code 3.3. Command used to explore an encrypted database

Note: The complete API of the SQLCipher can be reached at: <https://www.zetetic.net/sqlcipher/sqlcipher-api/>