

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	14
Obtaining SQLCipher and preparing the Android project	15
Implementing SQLCipher	16
Inspecting the contents of a database encrypted with SQLCipher	17

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. This protects the key store against unauthorized use. First, it avoids extracting the keys from the app processes and the Android device entirely in order to reduce unauthorized use of keys outside the device. Second, it allows apps to specify authorized uses of their passwords and applies these restrictions outside of app processes to reduce unauthorized use of key material on the device.

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 pair, and save and obtain it from the Android keystore. This key will require that the system be protected with a PIN, password or pattern, so that it is encrypted on the device. Keep in mind that if this option is disabled and the lock type is changed to none or to swipe, these keys will be deleted.

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>. Since this version has some errors, a corrected version of this code posted in the Aula Global of the subject will be used.

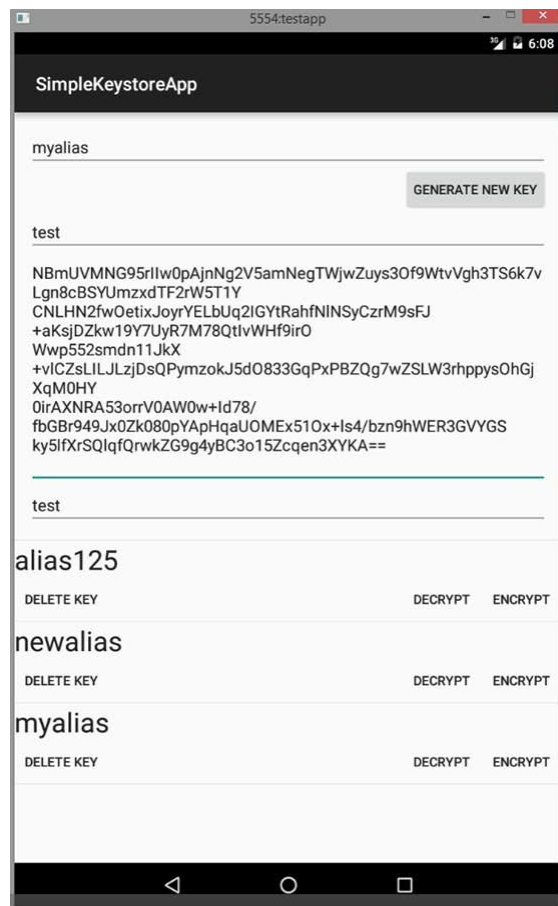


Image 2.1. Main screen

In order to better understand how this code works, we are going to modify the application. We copy the following into the MainActivity:

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())
            .setEncryptionRequired() //protect the key pair with the
secure lock screen credential
            .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(alias, null);
    if (!(entry instanceof KeyStore.PrivateKeyEntry)) {
        Log.e(TAG, " alias: " + alias + " is not a PrivateKey");
        return null;
    }

    return ((KeyStore.PrivateKeyEntry) entry).getPrivateKey();
}

```

4. We add functions in order to see if the system has some kind of locking mechanism:

```

public static boolean isDeviceSecured(Context context){
    KeyguardManager keyguardManager = (KeyguardManager)
context.getSystemService(Context.KEYGUARD_SERVICE);
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        return keyguardManager.isDeviceSecure();
    }else{
        return (isPatternEnabled(context) ||
isPassOrPinEnabled(keyguardManager));
    }
}

private static boolean isPatternEnabled(Context context){
    return (Settings.System.getInt(context.getContentResolver(),
Settings.System.LOCK_PATTERN_ENABLED, 0) == 1);
}

private static boolean isPassOrPinEnabled(KeyguardManager
keyguardManager){
    return keyguardManager.isKeyguardSecure();
}

```

5. We force the program to refresh the keys in case the screen lock security mechanism has been disabled while the application was not in foreground:

```

protected void onResume(){
    super.onResume();
    refreshKeys();
}

```

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.

Let's now make some changes to use our functions (mostly analogous to the text that will be replaced) in the application and see how they work:

1. We are going to change their function for obtaining the keys for ours, as shown in Figure 2.2.



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    /*try {
        keyStore = KeyStore.getInstance("AndroidKeyStore");
        keyStore.load(null);
    }
    catch(Exception e) {}*/
    loadKeyStore();
    refreshKeys();

    setContentView(R.layout.activity_main);
}
```

Figure 2.2. Replacing the key retrieval function

2. Likewise, we are going to use our key generation function, in the same way as in the previous case (Figure 2.3), but this time we will also add the check for the existence of a locking mechanism and we will notify the user if it does not exist:



```
public void createNewKeys(View view) {
    String alias = aliasText.getText().toString();
    try {
        // Create new key if needed
        if (!keyStore.containsAlias(alias) && isDeviceSecured( context: this)) {
            /*Calendar start = Calendar.getInstance();
            Calendar end = Calendar.getInstance();
            end.add(Calendar.YEAR, 1);
            KeyPairGeneratorSpec spec = new KeyPairGeneratorSpec.Builder(this)
                .setAlias(alias)
                .setSubject(new X500Principal("CN=Sample Name, O=Android Authority"))
                .setSerialNumber(BigInteger.ONE)
                .setStartDate(start.getTime())
                .setEndDate(end.getTime())
                .build();
            KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA", "AndroidKeyStore");
            generator.initialize(spec);
            KeyPair keyPair = generator.generateKeyPair();*/
            generateNewKeyPair(alias, getBaseContext());
        }
        if (!isDeviceSecured( context: this)){
            Toast.makeText( context: this, text: "Device not secured. Add PIN, password or pattern to the lock screen to generate keys", Toast.LENGTH_LONG).show();
        }
    } catch (Exception e) {
        Toast.makeText( context: this, text: "Exception " + e.getMessage() + " occurred", Toast.LENGTH_LONG).show();
        Log.e(TAG, Log.getStackTraceString(e));
    }
    refreshKeys();
}
```

Figure 2.3.Key generation substitution function and checking the locking mechanism

3. Lastly, we replace the key loading for decryption. To avoid errors, we also remove the string "AndroidOpenSSL" from the Cipher.getInstance method, thus making it to take care of finding the appropriate security provider by itself.

([https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html#getInstance\(java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html#getInstance(java.lang.String)))(Figure 2.4):

```
public void decryptString(String alias) {  
    try {  
        PrivateKey privateKeyEntry= loadPrivateKey(alias);  
        //KeyStore.PrivateKeyEntry privateKeyEntry = (KeyStore.PrivateKeyEntry)keyStore.getEntry(alias, null);  
        //AndroidKeyStoreRSAPrivateKey privateKey = (AndroidKeyStoreRSAPrivateKey) privateKeyEntry;  
  
        Cipher output = Cipher.getInstance("RSA/ECB/PKCS1Padding");  
        output.init(Cipher.DECRYPT_MODE, privateKeyEntry);  
  
        String cipherText = encryptedText.getText().toString();  
        CipherInputStream cipherInputStream = new CipherInputStream(  
            new ByteArrayInputStream(Base64.decode(cipherText, Base64.DEFAULT)), output);  
        ArrayList<Byte> values = new ArrayList<>();  
        int nextByte;  
        while ((nextByte = cipherInputStream.read()) != -1) {  
            values.add((byte)nextByte);  
        }  
  
        byte[] bytes = new byte[values.size()];  
        for(int i = 0; i < bytes.length; i++) {  
            bytes[i] = values.get(i).byteValue();  
        }  
  
        String finalText = new String(bytes, offset: 0, bytes.length, charsetName: "UTF-8");  
        decryptedText.setText(finalText);  
    } catch (Exception e) {  
        Toast.makeText( context: this, text: "Exception " + e.getMessage() + " occurred", Toast.LENGTH_LONG).show();  
        Log.e(TAG, Log.getStackTraceString(e));  
    }  
}
```

Figure 2.4. Replacing the keyload function and getInstance

Observing the behavior of the application, we see that it allows us to create new keys whenever the screen is blocked by PIN, password or pattern (Figure 2.5).



Figure 2.5. Key generation with activated locking screen mechanism

Whereas if there is no screen locking mechanism or it is set to swipe, the existing keys disappear (Image 2.6) and do not allow us to create new ones (Image 2.7).



Figure 2.6. Keys deleted due to disabling screen lock security mechanism



Figure 2.7. The application does not allow to create new keys without screen lock security mechanism (PIN, password or pattern)

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:
 - o Open and close the database rapidly
 - o The performance is excellent, event with very big databases
 - o 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:

1. We add the following lines to the file build.gradle (Module: app) in the dependencies section (Figure 3.1):

```
implementation 'net.zetetic:android-database-sqlcipher:4.3.0@aar'  
implementation "androidx.sqlite:sqlite:2.0.1"
```



```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation 'androidx.appcompat:appcompat:1.1.0'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'com.android.support:cardview-v7:26.1.0'  
    implementation 'com.android.support:appcompat-v7:26.1.0'  
    implementation 'net.zetetic:android-database-sqlcipher:4.3.0@aar'  
    implementation "androidx.sqlite:sqlite:2.0.1"  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'  
}
```

Figure 3.1. Section of the build.gradle file with the added lines

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);  
  
or  
  
private Context context;  
...  
SQLiteDatabase.loadLibs(context);
```

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.

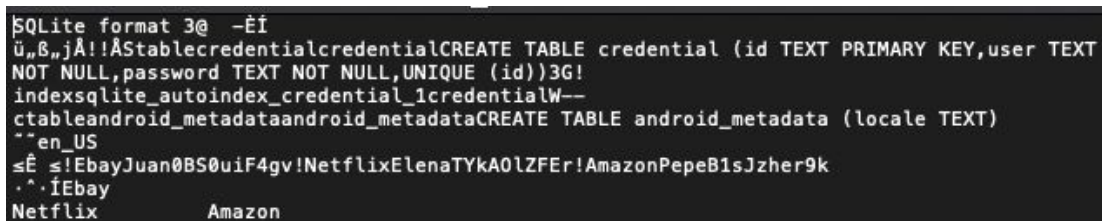
The following link presents example usage code.

(<https://www.zetetic.net/sqlcipher/sqlcipher-for-android/>)

3.3 Inspecting the contents of a database encrypted with SQLCipher

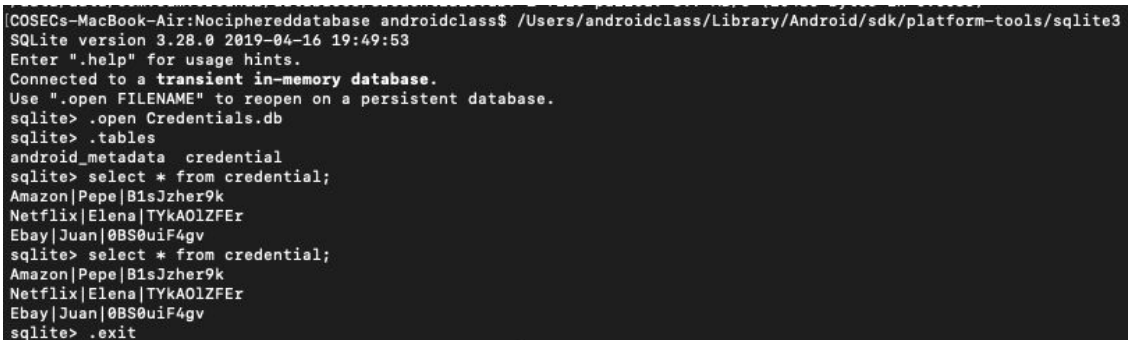
Due to the fact that a database created using SQLCipher is encrypted, we can no longer browse its contents using tools like sqlite3.

In order to query an encrypted database, the password is required. When opening an unencrypted database with a simple text editor, we can see the information (Figure 3.2), such as the user ID and password of the three entries, despite its bad formatting. It is also possible to access the data much more clearly using sqlite3 (Figure 3.3) or the DB Browser for SQLite tool (Figure 3.4). The database used for the example is located within the “Notciphereddatabase” folder within the files available in the Aula Global.



```
SQLite format 3@ -E!  
U„B„jÄ!!ÄStablecredentialcredentialCREATE TABLE credential (id TEXT PRIMARY KEY,user TEXT  
NOT NULL,password TEXT NOT NULL,UNIQUE (id))3G!  
indexsqlite_autoindex_credential_1credentialW--  
ctableandroid_metadataandroid_metadataCREATE TABLE android_metadata (locale TEXT)  
--en_US  
sÊ s!EbayJuan0BS0uiF4gv!NetflixElenaTYkA0lZFEr!AmazonPepeB1sJzher9k  
.*.iEbay  
Netflix          Amazon
```

Figure 3.2. Unencrypted database opened with a text editor



```
(COSECs-MacBook-Air:Nociphereddatabase androidclass$ /Users/androidclass/Library/Android/sdk/platform-tools/sqlite3  
SQLite version 3.28.0 2019-04-16 19:49:53  
Enter ".help" for usage hints.  
Connected to a transient in-memory database.  
Use ".open FILENAME" to reopen on a persistent database.  
sqlite> .open Credentials.db  
sqlite> .tables  
android_metadata credential  
sqlite> select * from credential;  
Amazon|Pepe|B1sJzher9k  
Netflix|Elena|TYkA0lZFEr  
Ebay|Juan|0BS0uiF4gv  
sqlite> select * from credential;  
Amazon|Pepe|B1sJzher9k  
Netflix|Elena|TYkA0lZFEr  
Ebay|Juan|0BS0uiF4gv  
sqlite> .exit
```

Figure 3.3. Unencrypted database opened with sqlite3



Figure 3.4. Unencrypted database opened with DB Browser for SQLite

However, let's see what happens when trying to open a database encrypted with SQLCipher in each case. The database to be used in this example is available in the Aula Global files in the "Ciphereddatabase" folder. When opened with a normal text editor, no information can be seen (Figure 3.5). When it is opened with sqlite3, this program is not able to identify it as a database (Figure 3.6).

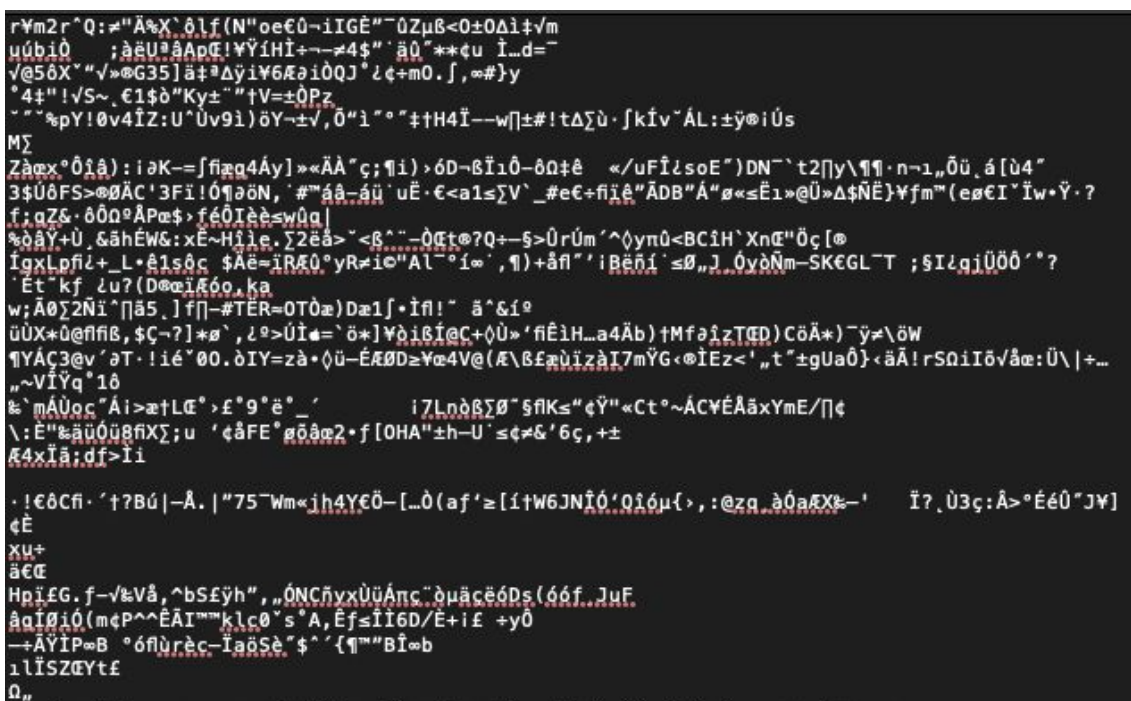


Figure 3.5. Encrypted database opened with a text editor

```

COSECs-MacBook-Air:Cipheredatabase androidclass$ /Users/androidclass/Library/Android/sdk/platform-tools/sqlite3
SQLite version 3.28.0 2019-04-16 19:49:53
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open Credentials.db
sqlite> .tables
Error: file is not a database
sqlite> .exit
COSECs-MacBook-Air:Cipheredatabase androidclass$

```

Figure 3.6. Encrypted database opened with sqlite3

Instead, when trying to access it with the DB Browser for SQLite tool, it will ask for a password. In this case the password is:

m7rof368gv4cnmknqhe3m1ra9

The password is entered, selecting the default encryption type of SQLCipher 4 (Figure 3.7), and it is already possible to view the data from said database in plain text format (Figure 3.8).

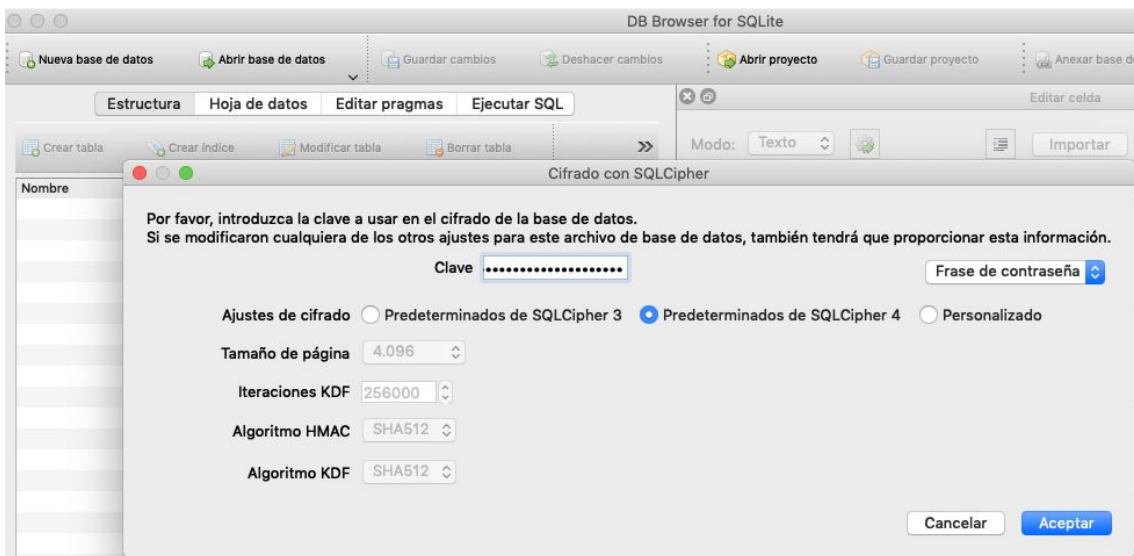


Figure 3.7. Encrypted database opened with DB Browser for SQLite asking for the password

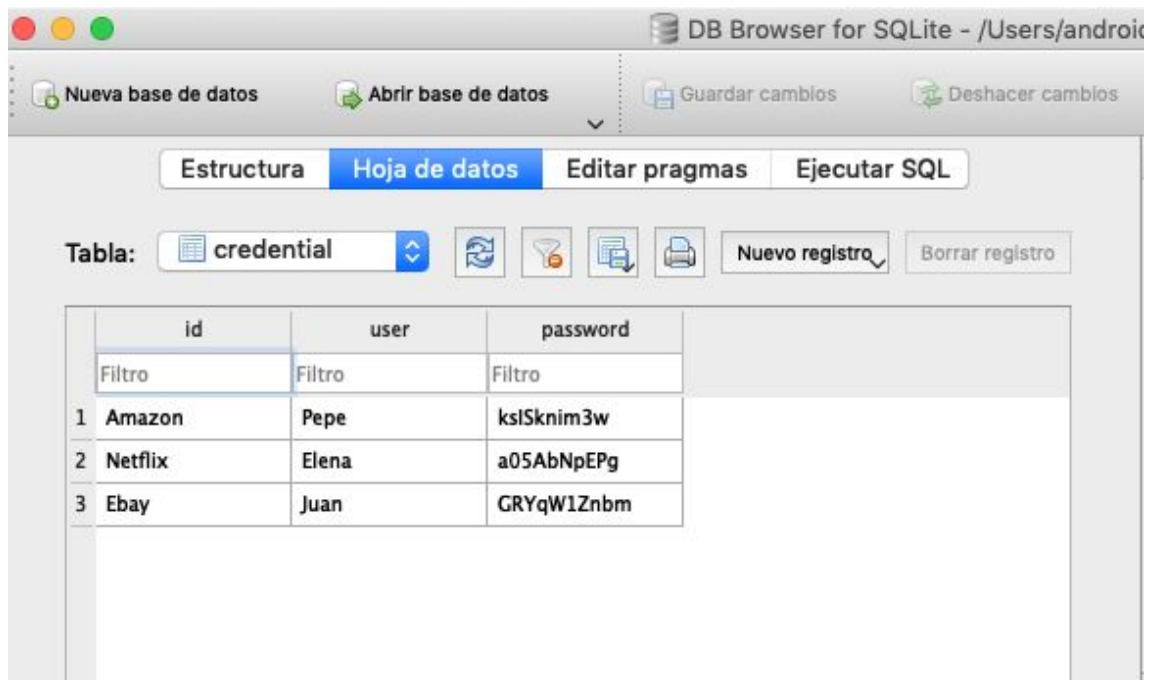


Figure 3.8. Encrypted database opened with DB Browser for SQLite