|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Document History** | | | | |
| **Version** | **Date** | **Author** | **Section** | **Changes** |
| 1.0 | 24-01-2017 | Anurag, Adarsha, Kavya | All | Integration of ORMLite with Sqlcipher |
| 1.2.0 | 10-07-2017 | Adarsha, Kavya,  Yogesh | All | SQLCipher version upgrade to 3.5.7 |

Secure Database Android Integration

|  |  |
| --- | --- |
| Author | Anurag, Adarsha, Kavya, Yogesh |
| Approved by |  |
| Email Id | [anurag.gautam@philips.com](mailto:anurag.gautam@philips.com) adarsha.shetty@philips.com  [kavya.g.kurpad@philips.com](mailto:kavya.g.kurpad@philips.com)  [yogesh.hs@philips.com](mailto:yogesh.hs@philips.com) |

1. INTRODUCTION 3

2. INTEGRATION 3

2.1 Maven repository Integration 3

2.2 Library Integration 3

Git source path: 3

Tfs Url Link: 3

2.3 Library versioning 4

getWriteDbPermission () 5

isOpen(): 6

getDao (): 7

close (): 8

2.4 Gradle dependencies 8

2.5 Prerequisites 8

3. How to execute a Secure Database 8

4. How to build new Secure database API 11

5. Proguard Rules for Secure DB 11

6. Frequently asked questions 12

a. What is ORMLite? 12

b. What is SQLCipher? 13

c. How SQLCipher integration different from SQLite 13

d. SQLCipher Advantages 13

e. SQLCipher Limitations 14

f. How internally SqlCipher working? 14

g. SQLCipher Security 15

7. Notes 15

# INTRODUCTION

This document provides an overview of integration procedure for secure database library in android mobile applications or any other components.

# INTEGRATION

## Maven repository Integration

**sqlcipher:3.5.7** No extra config required in build.gradle .

## Library Integration

# Git source path:

# 

# Tfs Url Link:

<http://tfsemea1.ta.philips.com:8080/tfs/TPC_Region24/CDP2/TEAM%20App%20Chassis/_git/ail-android-secure-db?path=%2F&version=GBdevelop&_a=contents>

Check out the code from above path, please refer sample demo app which depends on Secure Database library’s aar file.

Secure database has ORMLite and SqlCipher .

a. compile "com.j256.ormlite:ormlite-core:5.0"

b. compile “com.j256.ormlite:ormlite-android:5.0”

c. compile 'net.zetetic:android-database-sqlcipher:3.5.7@aar'

d. compile(group: 'com.philips.cdp', name: 'AppInfra', version: '1.7.0' + objcdp.getVersionSuffix(), ext: 'aar', changing: true){

transitive = true

}

## Library versioning

Library version can be obtained by using below API

**onCreate() and onUpgrade():**

The Secure DB integration is simple as Sqlite Database integration can be done by extending SQLiteOpenHelper. To implement Secure DB extend with SecureDbOrmLiteSqliteOpenHelper, this class will be your own DatabaseHelper class by extending SecureDbOrmLiteSqliteOpenHelper.

To create and upgrade a database in the android application, subclass of the SecureDbOrmLiteSqliteOpenHelper class should be created. In the constructor of subclass call the super() method of SecureDbOrmLiteSqliteOpenHelper specifying the context, instance of AppInfra , database name, CursorFactory, if anything else pass null,

app current database version and databaseKey. (This is the key for the database, based on this Secure DB will create own key with the help Appinfra’s Secure Storage Module “createKey” API, the created key used as a password for Database).

In this class, override the following methods to create and update your database.

* onCreate() - It is called the first time when database is created. Creation of table and initiaisation is done here.
* onUpgrade() – This method is called if the database version is increased in your application code. This method allows to update an existing database schema or to drop the existing database and recreate it via the onCreate() method.

The SecureDbOrmLiteSqliteOpenHelper class provides getWriteDbPermission() methods to get access to SqlcipherDatabase object write mode.

### getWriteDbPermission ()

- open or create the database (needs read or write permission) in SQLCipher, for security reasons internally it needs password to connect. We have to call this method with the help of helper class instance.

For example:

Suppose my DatabaseHelpher class instance is **secureDataBaseHelper**.

**secureDataBaseHelper**.**getWriteDbPermission()**

We need to write our own class to handle all database CRUD(Create, Read, Update and Delete) operations. For more information refer How to execute a Secure Database section.

### isOpen():

Return true if the helper is still open. Once helper is closed this will return false*.*

For example:

Suppose my DatabaseHelpher class instance is **secureDataBaseHelper**.

**secureDataBaseHelper. isOpen()**

### getDao ():

*Get a DAO for our class. This uses the**DaoManager to cache the DAO for future gets.*

 DAOs are the one of the most important components in ORMLite ecosystem as they are the only way to access database tables. So, every table should expose a DAO for the application to access it when required.

For example:

Suppose my DatabaseHelpher class instance is **secureDataBaseHelper**.

Dao<T, ?> dao = (Dao<T, ?>) **secureDataBaseHelper**.getDao(classname.class);

Example:

Dao<T, ?> dao = (Dao<T, ?>) **secureDataBaseHelper**.getDao(ContactList.class);

Using this DAO instance we can do any CRUD operations.

### close ():

To *close any open connections.*

For example:

Suppose my DatabaseHelpher class instance is **secureDataBaseHelper**.

**secureDataBaseHelper**.**close()**

## Gradle dependencies

No dependencies.

## Prerequisites

No

# How to execute a Secure Database

SecureDbOrmLiteSqliteOpenHelper is the helper class for app/component, it is extended from SQLiteOpenHelper.

Note: SQLiteOpenHelper class is from net.sqlcipher.database, not android.database.

SecureDbOrmLiteSqliteOpenHelper should be extended by your application if you want to create or upgrade its database.

Create Java Module Pojo class and use proper annotation to create database fields.

**Example: AddressBook**

**public class** AddressBook **implements** Serializable {  
  
 */\*\*  
 \* Model class for address database table  
 \*/* **public static final** String ***ID\_FIELD*** = **"address\_id"**;  
  
 // Primary key defined as an auto generated integer  
 // If the database table column name differs than the Model class variable name, the way to map to use columnName

@DatabaseField(generatedId = **true**, columnName = ***ID\_FIELD***)  
 **public int addressId**;  
  
 @DatabaseField(columnName = **"first\_name"**)  
 **public** String **firstName**;  
  
  
 @DatabaseField(columnName = **"last\_name"**)  
 **public** String **lastName**;  
  
 @DatabaseField  
 **public** String **address**;  
  
 @DatabaseField(columnName = **"contact\_number"**)  
 **public** String **contactNumber**;  
  
 **public** AddressBook() {  
  
 }  
  
 **public** AddressBook(String firstName, String lastName, String address, String contactNumber) {  
 **this**.**firstName** = firstName;  
 **this**.**lastName** = lastName;  
 **this**.**address** = address;  
 **this**.**contactNumber** = contactNumber;  
 }  
  
}

1. Extend SecureDbOrmLiteSqliteOpenHelper to your application helper class. Import SecureDbOrmLiteSqliteOpenHelper from “com.philips.platform.securedblibrary.ormlite.sqlcipher.android.apptools.SecureDbOrmLiteSqliteOpenHelper”

Example: SecureDataBaseHelper **extends** SecureDbOrmLiteSqliteOpenHelper

1. Pass this value to super class constructor from your helper class

**super**(context,mAppInfraInterface, “dataBaseName”, **null**, databaseVersion, “keyName”);

context- your activity class context.

dataBaseName - Name of the database to create

null - Cursor factory or null if none.

mAppInfraInterface - AppInfraInterface instance

databaseVersion - version number of database in int.

keyName - name of the key to retrieve generate password.

Example: **public** SecureDataBaseHelper(Context context, String dataBaseName, **int** databaseVersion, String keyName) {  
 **super**(context, mAppInfraInterface, dataBaseName, **null**, databaseVersion, keyName);  
   
}

1. Inside onCreate() of helper class create table.

@Override  
 **public void** onCreate(SQLiteDatabase database, ConnectionSource source) {  
 **try** {  
 TableUtils.*createTable*(source, **tableName**);  
 } **catch** (SQLException e) {  
 }  
  
 }

Example:

@Override  
 **public void** onCreate(SQLiteDatabase database, ConnectionSource source) {  
 **try** {  
 TableUtils.*createTable*(source, **AddressBook.class**);  
 } **catch** (SQLException e) {  
 }  
  
 }

Helper class sample code:

**public class** SecureDataBaseHelper<T> **extends** SecureDbOrmLiteSqliteOpenHelper {  
   
  
 **public** SecureDataBaseHelper(Context context, String dataBaseName, **int** databaseVersion, String databaseKey) {  
 **super**(context, dataBaseName, **null**, databaseVersion, databaseKey);  
   
  
 }  
  
  
 @Override  
 **public void** onCreate(SQLiteDatabase database, ConnectionSource source) {  
 **try** {  
 TableUtils.*createTable*(source, **tableName**);  
 } **catch** (SQLException e) {  
 }  
 }  
  
 @Override  
 **public void** onUpgrade(SQLiteDatabase database, ConnectionSource source, **int** oldVersion, **int** newVersion) {  
 **try** {

TableUtils.*createTable*(source, **tableName**);  
 TableUtils.*dropTable*(source, **tableName**, **true**);  
 } **catch** (SQLException e) {  
 }  
 }  
}

1. create object instance of your application helper class.

Example: SecureDataBaseHelper secureDataBaseHelper = new SecureDataBaseHelper(this, mAppInfraInterface, DATABASE\_NAME, DATABASE\_VERSION, DATABASE\_PASSWORD\_KEY);

1. Call getWriteDbPermission() from helper class instance to get database writable permission.

Example:

secureDataBaseHelper. getWriteDbPermission();

# How to build new Secure database API

Please refer sample demo application for more details

# Proguard Rules for Secure DB

Have to add below proguard rules to our application.

*#GMS***-keep** class com.google.android.gms.\* { **public \*;** }  
**-dontwarn** com.google.android.gms.\*\*  
  
  
*# The support library contains references to newer platform versions.  
# Don't warn about those in case this app is linking against an older  
# platform version. We know about them, and they are safe.***-dontwarn** android.support.\*\*  
  
**-keepattributes** InnerClasses,Exceptions  
  
  
*#ormlite***-keep** public class com.j256.ormlite.\*\* {**\*;**}  
**-keep** class com.j256.ormlite.\*\* { **\*;** }  
**-keep** interface com.j256.ormlite.\*\* { **\*;** }  
**-dontwarn** com.j256.ormlite.\*\*  
**-dontwarn** org.slf4j.\*\*  
**-dontwarn** org.apache.log4j.\*\*  
**-dontwarn** org.apache.commons.logging.\*\*  
**-dontwarn** org.apache.commons.codec.binary.\*\*  
**-dontwarn** javax.persistence.\*\*  
**-dontwarn** javax.lang.\*\*  
**-dontwarn** javax.annotation.\*\*  
**-dontwarn** javax.tools.\*\*  
  
*#sqlcipher***-keep** class net.sqlcipher.\*\* {**\*;**}  
**-keep** interface net.sqlcipher.\*\* { **\*;** }  
**-keep** class net.sqlcipher.database.\*\* {**\*;**}  
**-keep** interface net.sqlcipher.database.\*\* { **\*;** }  
**-keep** enum net.sqlcipher.\*\*  
**-keepclassmembers** enum net.sqlcipher.\*\* { **\*;** }  
  
*#Secure DB***-keep** public class com.philips.platform.securedblibrary.SecureDbOrmLiteSqliteOpenHelper.\*\*{ **public \*;**}

# Frequently asked questions

# What is ORMLite?

ORMLite is lightweight Java ORM which provides support for android Sqlite Database. ORMLite stands for Object Relational Mapping Lite(ORMLite) and it provides light weight functionality to store and retrieve Java Objects. It helps to avoid complexity and obtain more standard Object Relational Mapping.

ORMLite add objects of java using annotations and it has powerful abstract Database Access Object classes. It provides simple and flexible query using QueryBuilder. Auto generates SQL to create and drop database tables and it has basic support for database transactions.

More Guide Line about ORMLite Read below link:

http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite\_1.html#Getting-Started

# What is SQLCipher?

SQLCipher is used for a full database encryption. What *full* implies is that the DB file is entirely encrypted, not only separate rows or tables. It is a Modified version of SQLite and AES encryption by default.

More Guide Line about SQLCipher Read below link:

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

<http://lomza.totem-soft.com/tutorial-add-sqlcipher-to-your-android-app/>

# How SQLCipher integration different from SQLite

There is small chang in import:  
  
   import net.sqlcipher.database.SQLiteDatabase;  
   import net.sqlcipher.database.SQLiteOpenHelper;

import net.sqlcipher.database.SQLiteDatabase;  
  import net.sqlcipher.database.SQLiteQueryBuilder  
  
instead of  
  
  import android.database.sqlite.SQLiteDatabase;  
  import android.database.sqlite.SQLiteOpenHelper;

  import android.database.sqlite.SQLiteDatabase;  
  import android.database.sqlite.SQLiteQueryBuilder;  
  
  
Method signatures of both libraries  are same so no modification is required in function calls.  
  
There is a small difference between SQLite and SQLCipher. db.getReadableDatabase and db.getWriteableDatabase functions in android's SQLite  functions receives no argument while SQLCipher's functions take (String) password as an argument.

This password is used to encrypt when writing and decrypt when reading from database.

# SQLCipher Advantages

SQLCipher is often the go to choice for Android apps for the following reasons:

* Strong encryption (256-bit AES)
* Mature technology
  + Maintained and supported by its developers and the open source community
* Supports virtually the same API as standard Android database functions

# SQLCipher Limitations

It Adds 3MB to APK Size per CPU architecture.

# How is SqlCipher working internally?

When we extend SecureDbOrmLiteSqliteOpenHelper and pass necessary input to the constructor like context, AppInfra instance and data base name, data base keyName, database version then it will first load the SqlCipher library SQLiteDatabase.loadLibs(**this**).

**SQLiteDatabase.loadLibs(this):**

This must occur before any other database operation else will throw a runtime exception - java.lang.UnsatisfiedLinkError: No implementation found for void net.sqlcipher.database.SQLiteDatabase.dbopen(java.lang.String, int) (tried Java\_net\_sqlcipher\_database\_SQLiteDatabase\_dbopen and Java\_net\_sqlcipher\_database\_SQLiteDatabase\_dbopen\_\_Ljava\_lang\_String\_2I) and it's because we need to load a few native libraries first. So, add this line to the onCreate() method: SQLiteDatabase.loadLibs (this)

Note: The import of net.sqlcipher.database.SQLiteDatabase  instead of android.database.sqlite.SQLiteDatabase.

After loading the library to make secure Android database connection it will look for keyName, this keyName is the same user passed data base key to the SecureDbOrmLiteSqliteOpenHelper class constructor.

Once received, the keyName will use this to create key. The createKey API is same as Secure Storage createKey API. With the help of this API, it will generate password key and Stored in Secure Storage.

Once key is generated, it will use this key for database connection with the help of Secure storage getKey () API. Once receiving the key value, it will pass it to database connection and create database.

To perform create, insert, delete, update operation user needs permission. So users have to call getWriteDbPermission(). This method internally gets the getKey value and passes to getWritablePermission (keyvalue) to establish permission. Once the permission is obtained the user can do any data base operation.

# SQLCipher Security

* Customizable encryption algorithm based on OpenSSL libcrypto.
* Individual pages encrypted with own initialization vector.
* Message authentication code (MAC) per page to detect tempering.
* Hashed passphrase (PBKDF2) for key.

# Notes

1. Please refer sample application for more details.