SEcube™ -based SQLite Database Library

Technical Documentation

Release: February 2021


HW
SW
Open
Source

Proprietary Notice

The present document offers information subject to the terms and conditions described hereinafter. The authors reserve the possibility to change the content and information described in this document and to update such information at any time, without notice. Despite the attention that has been taken in preparing this document, typographical errors or omissions may have occurred.

Authors

Matteo FORNERO *(Researcher, CINI Cybersecurity National Lab)* matteo.fornero@consorzio-cini.it
Nicoló MAUNERO *(PhD candidate, Politecnico di Torino)* nicolo.maunero@polito.it
Paolo PRINETTO *(Director, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(PhD candidate, Politecnico di Torino)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Disclaimer

Contents

1 Introduction

The **SE*cube*™** -based SQLite Database is a library built on top of **SE*file*™** and SQLite that allows to work with traditional SQLite databases which inheriting all the security properties granted by **SE*file*™** . Basically, this is the result of the **SE*file*™** library applied to the SQLite database engine.
The **SE*cube*™** -based SQLite Database is made of two components: SQLite and **SE*file*™** . SQLite provides the database engine while **SE*file*™** provides the security features. Notice that the SQLite database engine provided here has been slightly modified in order to work with **SE*file*™** ; therefore, you also need to download **SE*file*™** in order to get it working properly.
The greatest advantage of this library is that it provides security features that are almost transparent to the user, in fact you can manage the Secure Database resorting almost exclusively to the traditional SQLite C interface that is well documented on the SQLite website [1].

2 SE*cube*™ SDK libraries and dependencies

The **SE*cube*™** SDK consists of several libraries that run on a host computer to which the **SE*cube*™** is connected. These libraries have been developed according to a hierarchical structure, therefore there are specific dependencies between them.
Table 1 summarizes the requirements of the **SE*cube*™** libraries (Y required, N not required). Each row identifies a library, each column identifies a dependency from another library. For example, **SE*key*™** depends on L0, L1, **SE*file*™** and the Secure Database.

	L0	L1	SEfile	SElink	SEkey	SEcure DB
L0	-	N	N	N	N	N
L1	Y	-	N	N	N	N
SEfile	Y	Y	-	N	optional	optional
SElink	Y	Y	N	-	optional	N
SEkey	Y	Y	Y	N	-	Y
SEcure DB	Y	Y	Y	N	N	-

Table 1: Requirements and dependencies of **SE*cube*™** libraries.

Notice that there are optional dependencies. In the case of **SE*file*™** , you do not need to include also the **SE*key*™** library if you do not plan to use the Key Management System; similarly, you do not need to include the Secure Database library if you do not plan to use encrypted SQLite databases. In the case of **SE*link*™** , you do not need to include the **SE*key*™** library if you do not plan to use the Key Management System, resorting instead on manual key management.
In Figure 1 you can see how a **SE*cube*™** project can be structured inside the IDE. This screenshot, in particular, was taken from the project that is actually used to develop the SDK. You can easily recognize, in fact, the folder related to the SDK (named 'sources', containing L0 and L1 APIs), the folder of **SE*key*™** , **SE*link*™** , the Secure Database and **SE*file*™** .

3 How to setup the Secure Database

In order to work with the Secure Database, you also need to download the **SE*file*™** source code[2]. If you also want to take advantage of key management features, then you can download the **SE*key*™** library[3]. Once all the necessary source code has been downloaded, it is suggested to compile the

[1] https://www.sqlite.org/cintro.html
[2] https://www.secube.eu/resources/open-sources-sdk/
[3] https://www.secube.eu/resources/open-sources-sdk/

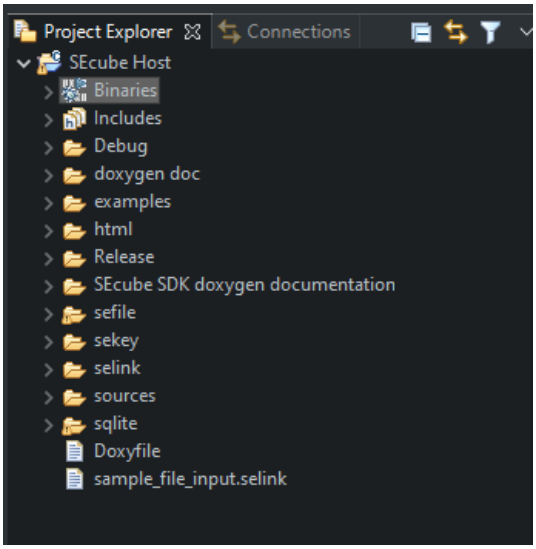

Figure 1: An example of how a **SE*cube*™** project can be structured.

code using the `-DSQLITE_TEMP_STORE=3` option. In order to complete the setup, you also need to check the **SE*file*™** configuration; please refer to the **SE*file*™** user manual for more information[4].

4 How to use the SE*cube*™ -based SQLite Database encrypted with SEfile

Inside the folder of **SE*file*™** , you will notice a file called `environment.h`. This file contains the declaration of three global variables, we focus on the variable called `databases`. This is an array of pointers to **SE*file*™** objects, each one is used to handle a file containing a SQL database encrypted with **SE*file*™** . If you are also using **SE*key*™** , this vector already contains a pointer, which points to the **SE*file*™** object used to manage the encrypted SQL database used by **SE*key*™** to store its metadata. If your application requires to use another SQLite database encrypted with **SE*file*™** , then you must carefully follow these steps:

1. create a `unique_ptr` to a **SE*file*™** object;
2. setup the security context you want to use for the database (i.e., set the pointer to the L1 **SE*cube*™** object, setup also the key ID and the algorithm if you need to create the file of the database, otherwise they will be inherited automatically if the file already exists);
3. set the `name` attribute of the `handleptr` attribute of your **SE*file*™** object to the clear-text name of the file of your database;
4. insert the `unique_ptr` you created into the databases array (use `std::move()`);
5. start working with your database using the `sqlite3*` pointer to the db connection.

Notice that the **SE*file*™** object will be automatically removed from the vector of databases when you call the `sqlite3_close()` API. When you have completed the steps listed above, you can start working with your database using all the standard APIs described in the documentation of the SQLite C interface [5]. You do not need to use custom functions or other functions implemented

[4] https://www.secube.eu/resources/open-sources-sdk/
[5] https://www.sqlite.org/cintro.html

by **SE*file*™** or other libraries, you can simply use the database as if you were working on an normal database. Here is an example.

```
unique_ptr<L1> l1 = make_unique<L1>();
/* other code here to login on the SEcube, etc. */
SEcube = l1.get(); // see section 3 of the SEfile getting
   started guide
sqlite3 *db;
unique_ptr<SEfile> dbfile = make_unique<SEfile>();
uint32_t key_id = 999;
dbfile->secure_init(l1.get(), key_id, L1Algorithms::Algorithms::
   AES_HMACSHA256);
char dbname[] = `test`;
memcpy(dbfile->handleptr->name, dbname, strlen(dbname));
databases.push_back(std::move(dbfile));
sqlite3_open(dbname, &db);
/* other code here to work on the database using all the
   traditional SQLite C interface APIs */
sqlite3_close(db);
```

Notice that you should not use directly the APIs of **SE*file*™** specific for the SQLite database engine. Those APIs are automatically called by SQLite itself, the only APIs of **SE*file*™** related to SQLite that you may consider are the `securedb_ls()`, the `securedb_recrypt()`, and the `securedb_get_secure_context()`.