# SEcube™
# Open Security Platform

## Introduction

Release: May 2020

# Proprietary Notice

The present document offers information, which is subject to the terms and conditions described hereinafter.
While care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein and to update such information at any time without notice. The opinions expressed are in good faith and while every care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinion expressed are in good faith and while every care has been taken in preparing this document.

## *Authors*

**Matteo FORNERO** *(Researcher, CINI Cybersecurity National Lab)* fornero.matteo@gmail.com
**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 Secure SQLite Database is a library built on top of **SE*file*™** and SQLite that allows to work with traditional SQLite databases which inherits 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 Secure 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*™ libraries overview and dependencies

The libraries for the **SE*cube*™** that are listed on the **SE*cube*™** website are interconnected and some of them cannot work without the others. In particular:

- **SE*key*™** requires also **SE*file*™** and the Secure Database.
- **SE*file*™** can work standalone if you do not plan to use **SE*key*™** and/or the Secure Database.
- The Secure Database requires **SE*file*™** to work correctly.

Notice that all these libraries require the APIs of L0 and L1 (**SE*cube*™** host-side SDK). Be careful about downloading all the source code you need for your target, here are few examples:

- If you want to use the **SE*cube*™** simply to implement a secure database (an encrypted SQL database with SQLite), then you must download the Secure Database library and **SE*file*™** .
- If you want to use the **SE*cube*™** to encrypt generic files, you do not care about key management and you are not interested in the Secure Database, then you simply need to download the source code of **SE*file*™** .
- If you need key management features (i.e. because you need to encrypt thousands of files with **SE*file*™** and you need to use many different keys) then you must download the **SE*key*™** source code, along with **SE*file*™** and the Secure Database.

Depending on the source code that you download, please read carefully the documentation provided in the ‘getting started‘ guidelines provided with the source code itself.

# 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. If you also want to take advantage of key management features, then you can download the **SE*key*™** library. Once all the necessary source code has been downloaded, it is suggested to compile the 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*™** getting started guide for more information (it is provided with the **SE*file*™** source code).

[1] https://www.sqlite.org/cintro.html

# 4 How to use the Secure 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 exploited 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 SEfile object.
2. Setup the security context you want to use for the database (i.e. set the pointer to the L1 SEcube 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 SEfile 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 once 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 [2]. You do not need to use custom functions or other functions implemented by **SE*file*™** or other libraries, you can simply use the database as if you were working on an unencrypted 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()`.

[2] https://www.sqlite.org/cintro.html