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

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by Blu5 View Pte Ltd. Other brands and names mentioned herein may be the trademarks of their respective owners. No use of these may be made for any purpose whatsoever without the prior written authorization of the owner company.

Disclaimer

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN “AS IS” BASIS AND ITS AUTHORS DISCLAIM ALL WARRANTIES, EXPRESS, OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PURPOSE. THE SOFTWARE IS PROVIDED TO YOU “AS IS” AND WE MAKE NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER WITH RESPECT TO ITS FUNCTIONALITY, OPERABILITY, OR USE, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PURPOSE, OR INFRINGEMENT. WE EXPRESSLY DISCLAIM ANY LIABILITY WHATSOEVER FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOSS REVENUES, LOST PROFITS, LOSSES RESULTING FROM BUSINESS INTERRUPTION OR LOSS OF DATA, REGARDLESS OF THE FORM OF ACTION OR LEGAL THEREUNDER WHICH THE LIABILITY MAY BE ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY LIKELIHOOD OF SUCH DAMAGES.

Contents

1 Introduction

SE*file*™ is a library that you can use, instead of the standard OS calls to the file system, to work on data stored on non-volatile memory. **SE***file*™ works as a wrapper around the traditional file system interfaces of Windows and Unix environments, adding a security layer provided by the **SE***cube*™ in order to grant confidentiality, integrity and authentication with AES-256-HMAC-SHA-256.
Basically, instead of using system calls like `read()` and `write()` you can use `secure_read()` and `secure_write()`, that work in a similar manner but provide security properties to your data. In conclusion, if you want to exploit **SE***file*™ to improve the security of your data, you need to write dedicated applications that are able to use the secure virtual file system interface of **SE***file*™ instead of the standard file system interface of the OS.

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 SE*file*™

Inside the folder of the source code of **SE***file*™ , you will find files related to **SE***file*™ itself but also to the Secure Database library. This is due to the fact that the Secure Database library is implemented using a partially customized version of **SE***file*™ ; however, some of the code is in common with the standard **SE***file*™ version therefore it has been decided to keep everything inside the same folder to minimize code duplication.
If you want to use **SE***file*™ along with **SE***key*™ , then you do not have to do anything. Instead, if you do not want to use **SE***key*™ and you only downloaded **SE***file*™ (and maybe also the Secure Database library), you must follow these steps:

1. Open the file named `SEfile.cpp` and comment the line where the `USING_SEKEY` constant is defined. Removing this definition, **SE***file*™ will skip the code that implies any reference to the APIs of **SE***key*™ (i.e. to check if a key is valid and can be used to encrypt data).

2. Open the file named `environment.h` and notice the global variable named `SEcube`. This is a pointer to the L1 object that is used to communicate with the **SE***cube*™ , this pointer is setup automatically by **SE***key*™ and it is used also in few functions of **SE***file*™ . Since you are not using **SE***key*™ , you need to setup this pointer manually. To do so, you simply need to assign to the `SEcube` global variable the address of the L1 object that you created in your `main()` function (remember to include the `environment.h` header file). Here is a simple example:

```
// this is in your main function
unique_ptr<L1> l1 = make_unique<L1>();
// other code here to login to the SEcube, etc...
SEcube = l1.get(); // you assign the pointer here, before using
   any SEfile API
```

4 Basic SE*file*™ example

Let us analyze a simple **SE***file*™ example. Imagine that we want to work on a text file, in particular we want to create it, write something to it, read what we wrote and close it. We can use the APIs of **SE***file*™ to perform these operations quite easily, remember that we need to work on an object of the **SE***file*™ class. Suppose that we want to encrypt this file using AES-256-HMAC-SHA-256 with the key having ID equal to 10 (of course we need a key with that ID stored in the **SE***cube*™).

```
unique_ptr<L1> l1 = make_unique<L1>();
// other code here to login to the SEcube, etc...
SEcube = l1.get(); // see section 3
SEfile myfile(l1.get(), 10, L1Algorithms::Algorithms::
   AES_HMACSHA256);
string filename = ''example.txt'';
string content = ''Hello World!'';
myfile.secure_open((char*)filename.c_str(), SEFILE_WRITE,
   SEFILE_NEWFILE); // force file creation
myfile.secure_seek(0, &pos, SEFILE_END); // append to the end of
    the file
myfile.secure_write((uint8_t*)content.c_str(), content.size());
myfile.secure_seek(0, &pos, SEFILE_BEGIN);
unique_ptr<char[]> filecontent;
uint32_t filedim;
secure_getfilesize((char*)filename.c_str(), &filedim, l1.get());
filecontent = make_unique<char[]>(filedim);
myfile.secure_read((uint8_t*)filecontent.get(), filedim, &
   bytesread);
myfile.secure_close();
```

5 How to use SQLite databases 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. 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
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 */
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()`.