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

This is a very brief overview about what you need to know to work with **SE*key*™** . Please check also the **SE*cube*™** SDK wiki[1], the example provided in the source code of **SE*key*™** , the Doxygen documentation of the code and, if you need further insights about the details of the KMS, the Master's Thesis[2] written by the developer of **SE*key*™** .

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.

3 What SE*key*™ is and what it is not

SE*key*™ is a software library that allows to implement a simple Key Management System specifically based on the **SE*cube*™** Security Platform, therefore it cannot be used without the **SE*cube*™** device. **SE*key*™** is not a complete software with a GUI or a command line interface. **SE*key*™** is, instead, a library that provides several APIs to implement a KMS according to the architecture that has been designed and described in the documentation of the open source SDK[3]. If you want to use **SE*key*™** , you need to build a dedicated application (i.e. a CLI or a GUI) that exploits the **SE*key*™** APIs to make the users able to actively use the KMS (i.e. the security administrator clicks a button of the GUI to create a key, the GUI calls the corresponding **SE*key*™** API). Notice that a dedicated GUI is currently under development.

4 SE*key*™ hardware requirements

SE*key*™ requires 1 host computer for the administrator, 1 **SE*cube*™** for the administrator, 1 host computer for each user and 1 **SE*cube*™** for each user. Notice that, if the administrator also needs to act as a user of the system, then the person who assumes the role of administrator needs 2

[1] https://www.secube.eu/resources/open-sources-sdk/
[2] https://webthesis.biblio.polito.it/14521/1/tesi.pdf
[3] https://www.secube.eu/resources/open-sources-sdk/

SE*cube*™ devices and, ideally, 2 host computers. One **SE*cube*™** will be used to act as administrator, the other one will be used to act as user (the same holds for the host computers).

5 SE*key*™ architectural requirements

The architecture of **SE*key*™** requires that the administrator and the users must be able to send each other files containing sensitive information about the keys, the users, and the groups of the KMS. These files are protected using **SE*file*™** . **SE*key*™** specifies how these files are generated and processed; however, it does not specify how these files are sent from one actor to the other. The only assumption that **SE*key*™** makes is that both the administrator and the users have access to a location, called `root` in the source code, that is where everybody can read and write files encrypted with **SE*file*™** . This is the location where the administrator will store the encrypted information that can be disclosed to every single user, this is where the users will write their log file so that the administrator will be able to read it. Notice that both the administrator and the users must have read and write access to this location.
SE*key*™ does not provide any protection for concurrent access to files stored at this location. If you plan to deploy **SE*key*™** in a concurrent context (i.e. a user may try to fetch from the "root" the most recent update for him while the administrator is writing it) then you have to develop concurrent access protection in the higher levels of your application or you can choose a location that inherently provides concurrent access protection. Notice that you should also provide a suitable authentication mechanism to the location where you store these file (i.e. a NAS accessible only from a local network or a cloud folder shared only with selected users).
Speaking at high-level, this location can be seen as a folder where several files are stored. The administrator writes them, then the users process them and delete them when they are not needed anymore.
At low-level, **SE*key*™** does not care about how you implement this architecture. While developing **SE*key*™** , different approaches have been tested: a shared folder in a Windows network, a folder accessible on a NAS on a private LAN, a folder shared by means of Dropbox. You can choose whatever approach you want (not limited to the three mentioned before), as long as you are compliant with the high-level view explained above.
In order to setup this essential aspect of the system, please follow these steps:

1. Open the file named `SEkey.cpp`
2. Modify the value of the `root` variable (located at the beginning of the file) setting it to the location where you want to store the files used by the administrator and by the users.
3. If you plan to use a Windows shared folder as `root` location, you also need to uncomment the line that defines the parameter named `SHARED_WINDOWS_FOLDER`.

6 How to compile SE*key*™

SE*key*™ has been compiled and tested on the following platforms:

- Windows 10 64-bit (10.0.18363 build 18363), Eclipse 2019-12, Mingw-w64 (x86_64-8.1.0-win32-seh-rt_v6-rev0)
- Ubuntu 18.04.4 LTS 64-bit, Eclipse 2019-12, Linux GCC/G++

You must compile **SE*key*™** using the flag `–DSQLITE_TEMP_STORE=3` (both for C and C++ source code). Moreover, we highly recommend to compile the SQLite library apart from the rest of the code, in order to avoid false warnings and errors reported by the IDE (this was the case of Eclipse,

other IDE might not have the same problem). If you compile SQLite apart from the rest of the code, remember to link it statically to your code. These are the steps you need to follow:

1. Exclude the file `sqlite3.c` (inside the sqlite folder) from the normal compilation of your IDE.
2. Open a terminal inside the folder named 'sqlite'.
3. Compile the SQLite library using this command: `gcc -v -DSQLITE_TEMP_STORE=3 -o sqlite3.o -c sqlite3.c`
4. Generate the archive using this command: `ar -rcs libsqlite3.a sqlite3.o`
5. Link statically the file `libsqlite3.a` to your code.
6. Delete the sqlite3.o file.

If you decided to compile SQLite apart from the rest of the code, notice that you must repeat this process every time that you change something in any of these files: `sqlite3.c, sqlite3.h, SEfile.h, SEkey_config.h`.

7 ID nomenclature for users, groups, and keys of SE*key*™

The users, the keys and the groups managed by **SE*key*™** are identified by a unique ID. This ID is not assigned automatically by **SE*key*™** , it can be decided by the administrator of the system. This choice was done in order to allow the administrator to pick a meaningful ID; however, the administrator can also exploit the APIs of **SE*key*™** to automatically retrieve the IDs currently used and generate a new ID which is available (i.e. find the current highest ID and increment it by 1). The only constraint on the IDs used in the KMS is that they must be compliant with the following rules:

- Keys: all IDs must be in the form "K*" where "*" can be replaced by any number (i.e. K1, K2, K983, etc.).
- Users: all IDs must be in the form "U*" where "*" can be replaced by any number (i.e. U1, U2, U983, etc.).
- Groups: all IDs must be in the form "G*" where "*" can be replaced by any number (i.e. G1, G2, G983, etc.).

Notice that the IDs for the keys stored inside any **SE*cube*™** are divided in three categories.

- The range [1, 100] is reserved for keys with special purposes, for example keys that can be used with **SE*file*™** without the support of **SE*key*™** . These keys cannot be managed with **SE*key*™** . The IDs 1 and 2 (meaning K1 and K2) are already used for internal purposes of **SE*key*™** , ID 0 is not available.
- The range [101, 4 294 867 295] is available for the keys managed by **SE*key*™** . You can use IDs such as "K101", "K900" or "K2020". Notice that the internal flash memory of the **SE*cube*™** cannot store $2^{32} - 1$ keys.
- The range [4 294 867 296, 4 294 967 295] cannot be used by **SE*key*™** because it is reserved for internal purposes of SEkey. This range of IDs is assigned to symmetric keys that are used to protect the encrypted files that the administrator of SEkey must send to the users. Notice that each user has 2 unique symmetric keys shared with the administrator that fall in this range. These keys cannot be managed with **SE*key*™** .

You can modify the ranges explained above working on the file `SEkey.h` as long as you do not use IDs 1 and 2.

8 SE*cube*™ initialization procedure for SE*key*™

The **SE***cube*™ devices used in the KMS must be "brand new", the best procedure is to erase the internal memory of every **SE***cube*™ and then flash the firmware specific for **SE***key*™ . After the memory of the **SE***cube*™ has been erased, the device must be initialized again. There is a different procedure for the administrator of **SE***key*™ and for the users.

- Administrator: in this case the **SE***cube*™ must be initialized, then it must be disconnected and reconnected to the PC in order to make the initialization effective. This implies an interaction with the software at the higher level (i.e. a GUI where the administrator sees a popup asking to connect, disconnect and reconnect the device). This interaction cannot be tackled at the level of the APIs of **SE***key*™ , it depends on the higher levels. Check the documentation of the `sekey_admin_init()` API for more information, here are the main steps.
  1. Prerequisite: the **SE***cube*™ of the administrator must be erased and re-programmed with the firmware for **SE***key*™ .
  2. Connect the **SE***cube*™ to the PC, do not login to the **SE***cube*™ .
  3. Call the `sekey_admin_init()` API, if it fails erase the device and re-program again the **SE***cube*™ .
  4. If the API succeeds, ask to the user to disconnect the device.
  5. Reconnect the device.
  6. Execute the login to the **SE***cube*™ .

  Notice that an initialization example for the **SE***cube*™ of the administrator is provided in the file named `minimal_working_example.cpp`.
- Users: in this case the sequence of actions is more complex because both the **SE***cube*™ devices of the administrator and of the user are involved. The prerequisite is the same: the device of the user must be erased and programmed with the correct firmware. Moreover, the **SE***cube*™ of the administrator must be already initialized. Notice that the user who is owner of the **SE***cube*™ to be initialized must be already registered to **SE***key*™ (i.e. the `sekey_add_user()` API was already executed and it was successful). Notice also that you can initialize the **SE***cube*™ of a user whenever you want, as long as that user has been already added to **SE***key*™ (i.e. you can add the user to **SE***key*™ , then you initialize his **SE***cube*™ few days later when he actually needs to use **SE***key*™). An example named `init_user_secube_wrapper()` is provided to show you how the initialization of the **SE***cube*™ of a user should be done.

9 How to manage the life cycle of encryption keys

The ultimate goal of a KMS is to manage the life cycle of encryption keys. Notice that you have several APIs to do that, in particular these APIs are very important (return type and parameter omitted for simplicity):

```
sekey_add_key()
sekey_activate_key()
sekey_key_change_status()
```

Notice that there is not any API to delete or destroy a key. This is due to the fact that the metadata of a certain key cannot be deleted from **SE***key*™ . The actual value of the key, however, can be deleted easily. You simply need to use the `sekey_key_change_status()` API passing as

parameter the 'destroyed' status as new status of the key. This API therefore will delete the key from every **SE*cube*™** containing it.
Check the documentation about these APIs in the wiki and in the Doxygen resources for more informations.

10 Metadata stored by SE*key*™ other than the key values

SE*key*™ is based on a lot of metadata that are used to manage the keys, the groups and the users of the system. In particular, here is the schema of the encrypted SQLite database that stores these data:

- Users table: ID, name, SEcube serial number, SEcube user pin, SEcube admin pin (this value is stored only inside the database stored in the SEcube of the administrator, the users do not know their admin PIN), k1 (the ID of the unique key used by the administrator to encrypt the updates for this user), k2 (the ID of the unique key used by the SEcube of the administrator to wrap the keys to be sent to the SEcube of this user), key algo (the algorithm to be used for the update files), init flag (0 if the SEcube of the user has not been initialized yet, 1 otherwise), update counter (counter that keeps the user aligned with the updates sent by the administrator).
- Groups table: ID, name, current, number of users, current number of keys, maximum number of keys, algorithm of the keys, default cryptoperiod of the keys.
- UserGroup table: ID of the user, ID of the group (this table associates each group with its members).
- SeKeys table: ID of the key, name of the key, owner of the key, current key status, algorithm of the key, length of the key, generation time, activation time, expiration time, cryptoperiod, deactivation time, key type, compromise time, destruction time, suspension time.
- Recovery table: ID of the user who needs a SEkey data recovery, serial number of the SEcube of the user (this table is used to store those who need to recover their data, for example because SEkey is not working properly on their SEcube).