SEkey™
Key Management System

Technical Documentation

Release: February 2021

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

This manual covers the details and the features of **SE*key*™** , the Key Management System of the **SE*cube*™** Open Source Security Platform. In order to learn more about **SE*key*™** , about the **SE*cube*™** and about the capabilities of this Open Source Security Platform, please check out the **SE*cube*™** SDK documentation[1]. The **SE*cube*™** SDK documentation contains a specific section about **SE*key*™** , providing additional insights about the architecture of the Key Management System.

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 SEcube 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 Hardware requirements

SE*key*™ requires one computer for the administrator, one **SE*cube*™** for the administrator, one computer for each user and one **SE*cube*™** for each user. For example, if we plan to use the Key Management System in an environment of 50 users, then we will need 50 **SE*cube*™** devices and 50 computers for the users, plus one **SE*cube*™** device and one computer for the administrator.

4 Architectural requirements

There are also additional requirements that must be met. Because of the segregation in terms of roles inside the Key Management System, the administrator cannot act as a user and vice versa.

[1] https://www.secube.eu/resources/open-sources-sdk/

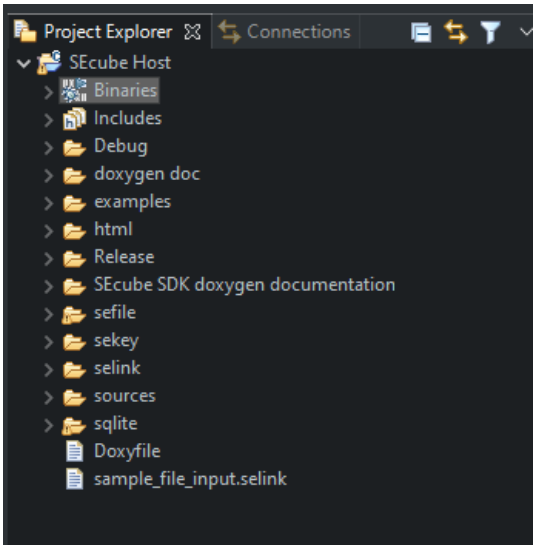

Figure 1: An example of how a **SE*cube*™** project can be structured.

Therefore, if you plan to implement a Key Management System where the administrator is also a user of the system, you will have to provide another **SE*cube*™** device (and another computer if you think so) to the person in charge of acting both as administrator and as a user.
Another requirement is that the users and the administrator must always be able to access with read/write privilege to a 'shared memory location'. Inside this shared memory, they will read and write encrypted files that are used to distribute asynchronously the data of the Key Management System from the administrator to the users (and for other stuff, like writing an encrypted **SE*key*™** logfile for each user).
The so-called 'shared memory location' can be any non-volatile memory easily accessible by all actors involved in the KMS, for example: a shared folder in a LAN, a folder hosted by a Cloud Service Provider (i.e., Google Drive, Dropbox, etc.), a NAS (Network-Attached-Storage). Notice that **SE*key*™** does not implement synchronization mechanism for accessing to the files stored on this non-volatile memory.

5 SE*key*™ environment setup

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++

Follow these steps in order to setup and compile **SE*key*™** :

1. go to https://www.secube.eu/resources/open-sources-sdk/ and download the Open Source **SE*cube*™** SDK;
2. extract the downloaded archive;
3. go to https://www.secube.eu/resources/open-sources-sdk/ and download the **SE*key*™** SDK;
4. extract the downloaded archive and copy the 'sekey' folder into the folder where you extracted the Open Source **SE*cube*™** SDK (see Figure 1 as a general example);

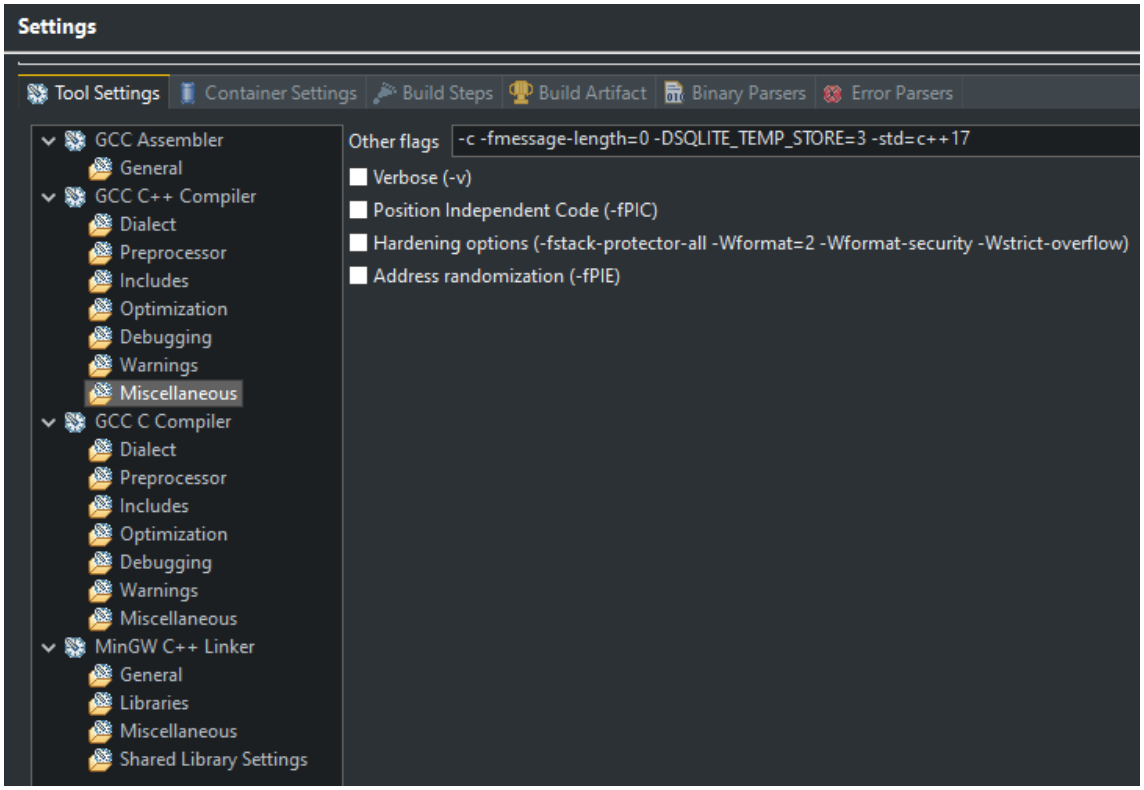

Figure 2: An example of how the compiler shoud be configured.

5. launch your IDE (i.e., Eclipse);
6. open the file named `SEkey.cpp`;
7. modify the value of the `root` variable (located at the beginning of the file) setting it to the path of the 'shared memory location' mentioned in Section 4;
8. if the 'shared memory location' is a disk or a folder of a 'Windows Network', uncomment the line that defines the parameter named `SHARED_WINDOWS_FOLDER`;
9. add the following compilation options to your compiler (i.e., G++ and GCC):
   `-DSQLITE_TEMP_STORE=3 -std=c++17`

An example of how the compiler should be configured is shown in Figure 2.

6 Rules about users, groups and cryptographic keys

The users, the keys and the groups managed by **SE*key*™** are identified by an ID. Each ID is unique, it must be assigned manually by the administrator to each individual entity (i.e., a group, a user, a key). Each ID must be compliant to the following rules:

- keys: all IDs must be in the form "K*" where "*" can be replaced by any number in the range (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 different categories accoding to their numbers:

- the value 0 (ID equal to K0) is not allowed;
- values in the range [1, 999] (i.e., K1, K10, K100, etc.) are reserved for manual key usage (for those who do not want to use **SE*key*™**);
- values in the range [1000, 1999] (i.e., K1000, K1001, K1002, etc.) are reserved for internal purposes of the **SE*cube*™** and of the **SE*cube*™** libraries;
- values in the range [2000, 4.294.867.293] can be freely assigned to keys managed by **SE*key*™** (you want to use this value range);
- values in the range [4.294.867.294, 4.294.967.294] are reserved for internal purposes of the **SE*key*™** library;
- value 4.294.967.295 is not allowed.

Modifying the values of these ID ranges may cause errors and unexpected behavior in the software.

7 How to initialize the SE*cube*™ of the administrator for SE*key*™

Before being actively used in the Key Management System, each **SE*cube*™** device must be physically erased and reprogrammed with the latest firmware (https://www.secube.eu/resources/open-sources-sdk/).
The initialization of the **SE*cube*™** of the administrator is managed by the **SE*key*™** function named `sekey_admin_init()`. This function is not ‘ready to be used‘, because it requires a physical interaction of the person performing the initialization to disconnect and reconnect the **SE*cube*™** to the computer during the process.
Depending on your environment, you may want to modify this function in order to make it compatible with a GUI or a CLI, simply because you must be able to ask to the user to disconnect and reconnect the device (which is something that is application-dependent, so it cannot be tackled at the level of the **SE*key*™** API).
Here are the main steps that you need to follow:

1. erase and reprogram the **SE*cube*™** of the administrator with the firmware for **SE*key*™** ;
2. connect the **SE*cube*™** to the PC;
3. without performin any login on the **SE*cube*™** , call the `sekey_admin_init()` function;
4. if the function succeeds, ask to the user to disconnect the device;
5. reconnect the device;
6. login to the **SE*cube*™** in order to check if the PIN codes set by the initialization function are correct.

In case of any error, start again from step 1.
Important notice: when initializing the **SE*cube*™** of the administrator user, two PIN codes must be set. Each PIN code is made of 32 characters; there is a PIN code for logging in the **SE*cube*™** with administrator privilege and another to log in with user privilege. The administrator of **SE*key*™** must set unique PIN codes for each **SE*cube*™** device involved in the KMS; moreover, the administrator must not disclose to anyone the PIN codes of his **SE*cube*™** device.
In order to be able to act as administrator of the KMS, the administrator must always login with his admin PIN on the **SE*cube*™** before starting **SE*key*™** .

8 How to initialize the SE*cube*™ of a user for SE*key*™

The initialization of the **SE*cube*™** of a user for **SE*key*™** can be done only by the administrator of **SE*key*™** . Before being actively used in the Key Management System, each **SE*cube*™** device must be physically erased and reprogrammed with the latest firmware (https://www.secube.eu/resources/open-sources-sdk/).
The initialization of the **SE*cube*™** of a user can be done only if specific requirements are met:

1. the **SE*cube*™** device of the administrator must be initialized;
2. the **SE*cube*™** device of the user must be erased and reprogrammed with the correct firmware for **SE*key*™** ;
3. the administrator must add the user to **SE*key*™** (by means of `sekey_add_user()`) before initializing the **SE*cube*™** of the user.

If the prerequisites are met, the **SE*cube*™** of a user can be initialized according to this procedure:

1. connect the **SE*cube*™** of the administrator to the computer;
2. launch **SE*key*™** on the computer of the administrator;
3. call the function to initialize the **SE*cube*™** of the user (an example is provided by the `init_user_secube_wrapper()` function);

Similarly to the initialization of the administrator, also the initialization of the user requires interaction with the person physically doing the task. This interaction can be handled only at the application level, because it depends on the application using the **SE*key*™** API (i.e. a GUI or a CLI). Take a look at the code of the `init_user_secube_wrapper()` function in order to understand how you can implement your own initialization function.
Important notice: when initializing the **SE*cube*™** of a user, two PIN codes must be set. Each PIN code is made of 32 characters; there is a PIN code for logging in the **SE*cube*™** with administrator privilege and another to log in with user privilege. The administrator of **SE*key*™** must set unique PIN codes for each **SE*cube*™** device involved in the KMS; moreover, the administrator must not disclose to anyone the PIN code for logging in as administrator to the **SE*cube*™** of any user. Both PIN codes of the **SE*cube*™** devices of the users are automatically stored in a safe way by the **SE*cube*™** of the administrator.
Simply put, the administrator tells to a user only the PIN code to login to his **SE*cube*™** with user privilege level. The privilege level user to login on the **SE*cube*™** is exactly what **SE*key*™** uses to distinguish if **SE*key*™** itself is being run by the KMS administrator or by a normal user (for this specific reason, the administrator must always login with admin privilege on his **SE*cube*™** before starting **SE*key*™** ; conversely, the users must always login with user privilege on their **SE*cube*™** devices before starting **SE*key*™**).

9 SE*key*™ APIs

The library of **SE*key*™** offers many APIs; however, in the source code you will also find a lot of other functions that are used for internal purposes. The functions listed in this section are what you need to use the Key Management System, do not use other functions unless you are perfectly sure that you have understood the documentation and the meaning of what is done in the source code. For more details about the implementation of the **SE*key*™** APIs and other functions, please refer to source code comments. Moreover, the comments are written using the Doxygen syntax so you can generate the documentation as you please..

```
int sekey_start(L0& l0, L1 *l1ptr);
int sekey_stop();
```

These APIs are used respectively to start and stop **SE*key*™** . The first function takes as input parameters the pointers to the L0 and L1 object that are necessary to exploit the **SE*cube*™** device. Notice that those objects should be created by the higher levels, i.e. the 'main' of your application. These functions must be called when opening and closing the application that needs to work with **SE*key*™** .

```
int sekey_admin_init(L1& l1, vector<string>& pin, string&
   userpin, string& adminpin);
```

This function is used to physically initialize the **SE*cube*™** device of the **SE*key*™** administrator. Clearly, this function must be executed by the **SE*key*™** administrator only once, before starting to build the KMS (i.e. adding users, groups and keys). Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_init_user_SEcube(string& uid, string& userpin, string&
    adminpin, vector<string>& pin);
```

This function is used to physically initialize a **SE*cube*™** device dedicated to the user identified by the ID specified as first parameter. The second and third parameter are the two PINs to be set on the **SE*cube*™** , in particular, only the user PIN should be disclosed to the owner of the **SE*cube*™** . The last parameter is a list of PINs to be used to login to the **SE*cube*™** of the user before the initialization (the latest **SE*cube*™** firmware uses a default PIN which is 32 bytes long, each byte is equal to 0, so a list with a single PIN of all zeros should be enough).
Notice that the initialization of the **SE*cube*™** device of the user must be done after calling the `sekey_add_user()` function. However, you do not need to call this function immediately after the `sekey_add_user()`; you can delay it as long as you want, the only thing that matters is that you call this function before physically giving the **SE*cube*™** to the user.
Notice that this function can be executed exclusively by the administrator of **SE*key*™** .
For Linux users: when you connect the **SE*cube*™** of the user to the host computer, remember to mount the **SE*cube*™** otherwise the initialization might fail.

```
int sekey_add_user(string& user_id, string& username);
```

This function adds a new user to **SE*key*™** , using the required ID and username. If the ID is not available, an error is returned. The user, when is added, does not belong to any group by default. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_delete_user(string& userID);
```

This function deletes the user specified by the ID passed as parameter from **SE*key*™** . The user will be completely deleted from the system (i.e. from the list of users and from any group to which the user may belong); moreover, the **SE*cube*™** of the deleted user will receive a special instruction that will result in the complete deletion of any information related to **SE*key*™** from the **SE*cube*™** of the user (all the keys will be erased from the internal flash memory and the database containing the metadata of the KMS will be emptied). Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_add_user_group(string& userID, string& groupID);
```

This function adds the given user to the given group. When the user is added, the other members of the group are notified of the presence of a new user (the administrator automatically sends the details of the new user to the other members of the group). Similarly, the new user is provided with the details about all the other members of the group and with the encryption keys associated to that group. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_delete_user_group(string& user_id, string& group_id);
```

This function deletes the given user from the given group; however, the user is still part of **SE*key*™** . This means that all the keys of that group will be deleted from the **SE*cube*™** of the user, moreover, the user will loose contact with other users whose only common group was the one specified as parameter. Notice that this function can be executed only by the administrator of **SE*key*™** .

```
int sekey_user_change_name(string& userID, string& newname);
```

This simple function allows to change the username of the user specified by the ID passed as parameter. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_user_get_info(string& userid, se_user *user);
int sekey_user_get_info_all(vector<se_user> *users);
```

These functions retrieve the details about a specific user (first one) or about all users (second one), filling the object passed as second parameter. For more information about the attributes of the object of class `se_user`, check the Doxygen documentation. Notice that, when a user calls this function, he cannot retrieve the details about any user of the system but only about the users that he knows, meaning the user that have at least a common group with him.
Notice that these functions can be used by the administrator and by the users.

```
int sekey_add_group(string& groupID, string& group_name,
   group_policy policy);
```

This function adds a new group to **SE*key*™** , using the required ID, name, and security policy. If the ID is not available, an error is returned. Notice that the new group, by default, is empty and does not own any key. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_delete_group(string& groupID);
```

This function deletes the given group from **SE*key*™** . The users belonging to the group will simply be removed from the group and will not have access to its keys anymore. The keys associated to the group will not be deleted, they will be deactivated and their owner will be set to ‘zombie‘ (i.e.

like with processes in an operating system). The keys will still be usable, by the administrator, to decrypt old data; the administrator may also decide to delete all the keys of the group if they are not needed anymore. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_group_change_name(string& groupID, string& newname);
```

Simply change the name of an existing group. Notice that the other attributes of the group (i.e. ID and involved users) are not changed. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_group_change_max_keys(string& groupID, uint32_t
   maxkeys);
```

Simply change the maximum number of encryption keys that can be associated to an existing group. Notice that the other attributes of the group (i.e. ID and involved users) are not changed. If the specified number of keys is lower than the current number of keys, the change does not take place. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_group_change_default_cryptoperiod(string& groupID,
   uint32_t cryptoperiod);
```

Change the default cryptoperiod used for the encryption keys of the specified group. Notice that the change is applied only for keys that will be activated after this change, the encryption keys that were activated in the past are not affected by this change therefore their expiration date will stay the same as before. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_group_get_info(string& groupID, se_group *group);
int sekey_group_get_info_all(vector<se_group> *groups);
```

These functions retrieve the details about a specific group (first one) or about all groups (second one), filling the object passed as second parameter. For more information about the attributes of the object of class `se_group`, check the Doxygen documentation. Notice that, when a user calls this function, he cannot retrieve the details about any group of the system but only about the groups to which he belongs. These functions can be used by the administrator and by the users.

```
int sekey_add_key(string& key_id, string& key_name, string&
   key_owner, uint32_t cryptoperiod, se_key_type keytype);
```

This function adds a new key to **SE*key*™** , using the required parameters. If the ID is not available, an error is returned. Notice that the caller must provide also the owner of the key, meaning the ID of the group that is associated to the key (i.e. the key 'K11' is associated to the group 'G7'). The *cryptoperiod* is optional, the caller can insert '0' to use the default cryptoperiod of the group owner of the key. Finally, notice that the key type must be always '*symmetric_data_encryption*'. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_activate_key(string& key_id);
```

This function is used to activate the key specified by the ID passed as parameter. Upon activation, the expiration time is set depending on the cryptoperiod of the key. Notice that a key can be activated only if its status is 'pre-active' or 'suspended' (in this last case it is a re-activation). Once

a key is activated, it can be used to encrypt data. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_key_change_status(string& key_id, se_key_status status
   );
```

This API is used to change the status of an encryption key, according to Figure **??**. Notice that to change the status to 'active', there is a dedicated function. The status of a key must be changed by the administrator according to the conditions of the environment, for example if the administrator suspects that a key has been found by an unauthorized entity, then the key must be set to the 'compromised' state. Similarly, to destroy a key its status can simply be set to 'destroyed' and it will be erased from all **SE*cube*™** devices that have it in their flash memory. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_key_change_name(string& key_id, string& key_name);
```

This function is simply used to change the label of an encryption key. The other attributes of the key remain unchanged. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_key_get_info(string& key_id, se_key *key);
int sekey_key_get_info_all(vector<se_key> *keys);
```

These functions retrieve the details about a specific key (first one) or about all keys (second one), filling the object passed as second parameter. For more information about the attributes of the object of class `se_key`, check the Doxygen documentation. Notice that, when a user calls this function, he cannot retrieve the details about any key of the system but only about the keys that are stored on his personal **SE*cube*™** , meaning the keys associated to the groups to which he belongs. Notice that these functions can be used by the administrator and by the users.

```
int sekey_find_key_v1(string& chosen_key, string& source_user_id
   , string& dest_user_id, se_key_type keytype);
int sekey_find_key_v2(string& chosen_key, string& source_user_id
   , string& group_id, se_key_type keytype);
int sekey_find_key_v3(string& chosen_key, string& source_user_id
   , vector<string>& dest_user_id, se_key_type keytype);
```

These APIs allow to find the most secure key to be used in different scenarios:

- the first function is used in a 1-to-1 communication (i.e. 'U1' wants to send an encrypted message to 'U2') , therefore both parties must share at least one common group;
- the second function is used in a 1-to-N communication, where the sender is a single user and the recipients are all the members of a specific group;
- the third function is used in a 1-to-N communication, where the sender is a single user and the recipients are a set of users. Notice that, if all the users involved (sender and recipients) do not share at least one common group, this function will fail because it will be impossible to find a single key that can be used for all recipients.

The '*keytype*' parameter must be '*symmetric_data_encryption*' (it was added explicitly to make the KMS able to handle also other types of keys in the future). Notice that in the first and in the

third function, the most important rule to find the most secure key is to find an active key (i.e. usable for encryption) belonging to the smallest group containing all the users involved. This does not automatically imply that only the involved users have access to that key, for example two users may have only a group in common, which includes 3 other users; therefore also these users will have access to the chosen key and will be able to decrypt the data. Creating small groups of 2 users to enable secure 1-to-1 communication is considered to be a responsibility of the administrator. Notice that these functions can be used only by the users of **SE*key*™** .

```
int sekey_readlog(string* sn, string& output);
```

This function opens the log file generated by the **SE*cube*™** device with the serial number passed as first parameter and copies its content, as cleartext, into the string passed as second parameter; the string then can be printed to a normal text file for further analysis. Notice that the log file is generated automatically by the administrator and by each user of **SE*key*™** , the log file is useful to record which actions have been performed in a specific moment. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_recovery_request(string& user_id, string&
   serial_number);
```

This function can be used to recover the integrity of the data of **SE*key*™** concerning a specific user. Let us imagine that a user has deleted the encrypted file used by **SE*key*™** from the MicroSD of his **SE*cube*™** or, for some reason, **SE*key*™** is not working properly on the **SE*cube*™** of the user. The administrator can try to solve the problem calling this function that, given the ID of the user and the serial number of his **SE*cube*™** , sends again to the user all the data that he is allowed to access (keys, groups, users, etc.). Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_check_expired_keys();
```

This is a simple function that iterates over all the keys stored within **SE*key*™** , checking if the expiration time of the key has been reached or not. If it has been reached and the key has not been deactivated yet, then it deactivates the key. Notice that this function has no effect if the key is already in the status ‘deactivated‘, ‘compromised‘, or ‘destroyed‘. This function is used extensively within the APIs of **SE*key*™** and **SE*file*™** to ensure that expired keys are never used when they are not supposed to be used. However, this function can be safely called also by higher levels from time to time (but it is not strictly necessary); notice that it can be called by the administrator and the users.

```
int sekey_update_userdata();
```

This function is called automatically by **SE*key*™** , but the higher levels may safely call it from time to time (but it is not strictly necessary). Its purpose is to check if the administrator has sent any new data related to **SE*key*™** , concerning the current user. This function can be called exclusively by the users of **SE*key*™** , it cannot be called by the administrator.

10 Use Cases

10.1 Administrator Use Cases

Among all the management use cases, the ones related to the creation of new objects (groups, users, keys) are the most complete. For this reason, they are described here, while the ones related to the modification and elimination of objects, that are similar and simpler, are not shown.

Use case UC1: Add Group
Application: **SE*key*™**
Actor: Security Admin
Pre-conditions: Security Admin is already authenticated and logged in.
Post-conditions: A new group is added to **SE*key*™** .
Main scenario:

1. Security Admin calls the API to add a new group to the KMS, providing the required parameters (i.e. the name of the group, the security policy, etc.);
2. The API validates the correctness of the request and checks for collisions in the current KMS configuration (i.e. the administrator requested a group ID that is already in use);
3. If the request of the administrator is valid, the API modifies the database containing the metadata of **SE*key*™** adding a new group that is empty and is not owner of any key;
4. A suitable result code is returned to the administrator, who then can assess if the operation was completed or not.

Use case UC2: Add User
Application: **SE*key*™**
Actor: Security Admin
Pre-conditions: Security Admin is already authenticated and logged in
Post-conditions: A new user is added to **SE*key*™** . An encrypted update file is prepared for the **SE*cube*™** of the new user.
Main scenario:

1. Security Admin calls the API to add a new user to the KMS, providing the required parameters (i.e. the name of the user, the desired ID, etc.);
2. The API validates the correctness of the request and checks for collisions in the current KMS configuration (i.e. the administrator requested a user ID that is already in use);
3. If the request of the administrator is valid, the API modifies the database containing the metadata of **SE*key*™** adding a new user who still does not belong to any group;
4. A suitable result code is returned to the administrator, who then can assess if the operation was completed or not.

Use case UC3: Add Key
Application: **SE*key*™**
Actor: Security Admin
Pre-conditions: Security Admin is already authenticated and logged in
Post-conditions: A new key is added to **SE*key*™** . A specific encrypted update file is prepared for the **SE*cube*™** of each user belonging to the group owner of the new key.
Main scenario:

1. Security Admin calls the API to add a new key to the KMS, providing the required parameters (i.e. the name of the key, the desired ID, the group owner of the key, etc.);
2. The API validates the correctness of the request and checks for collisions in the current KMS configuration (i.e. the administrator requested a key ID that is already in use);
3. If the request of the administrator is valid, the API modifies the database containing the metadata of **SE*key*™** adding a new key;

4. After storing the metadata of the new key in the encrypted database, the API issues a request to the firmware of the **SE*cube*™** to internally generate a key of the specified length using the True Random Number Generator of the **SE*cube*™** ;

5. The **SE*cube*™** of the administrator generates the key using the TRNG, then the key is stored inside the flash memory of the **SE*cube*™** without being exposed outside of the device;

6. The API checks if the key was successfully generated; if so then it prepares a specifically encrypted update file for each member of the group that has been specified as owner of the new key, inserting into the update file the metadata of the key;

7. The API must also insert the actual key value inside the update, in order to distribute the key to the entitled users. This is done issueing a request to the firmware of the **SE*cube*™** which returns to the API the value of the new key, wrapped using AES-256 with another *encryption key* that is symmetric and unique for each pair of user and administrator;

8. The API, without having access to the actual value of the new key because it is encrypted, writes it into the encrypted update file for each involved user (notice the double encryption layer around the value of the key, the first one provided with the wrapping of the **SE*cube*™** and the second one provided by **SE*file*™**);

9. A suitable result code is returned to the administrator, who then can assess if the operation was completed or not.

10.2 Users Use Cases

The purpose of the user is to exploit group keys in order to encrypt and decrypt data to be exchanged with other users. A user can decide to encrypt some data by choosing as recipient another user, an entire group to which he belongs or a set of users.

Use case UC4: Encrypt Data
Application: **SE*key*™**
Actor: User
Pre-conditions: User is already authenticated and logged in
Post-conditions: **SE*key*™** returns to the user the ID of the key to be used for encryption
Main scenario:

1. The User calls one of the APIs of **SE*key*™** to retrieve the most secure key to be used providing the recipient(s) (i.e. a single user, unicast communication);

2. **SE*key*™** checks the validity of the requests, searches the encrypted database for the smallest group containing both the sender and the receiver, then looks for the active key (i.e. usable for encryption) with the shortest cryptoperiod (assuming that a shorter cryptoperiod implies higher security);

3. If a suitable key is found, the API returns the ID of that key to the user, otherwise an error code is returned;

4. The user checks the return code and, if possible, specifies that key ID as a parameter to the encryption API to encrypt the required data.

Use case UC5: Retrieve known users
Application: **SE*key*™**
Actor: User

Pre-conditions: User is already authenticated and logged in
Post-conditions: **SE*key*™** returns to the user the list of known users
Main scenario:

1. The User calls the API of **SE*key*™** to retrieve the list of known users, meaning the users who have at least one group in common with the current user;
2. **SE*key*™** selects the metadata (i.e. the name and the ID) of all the users from the encrypted database stored in the **SE*cube*™** device of the current user (notice that the database contains exclusively users with at least one group in common);
3. **SE*key*™** fills a list containing all the details about the known users, then it returns an appropriate result code;
4. The user checks the return code and, if possible, uses the details of the known users for his own purposes (i.e. displaying a list of 'contacts' that are available for an encrypted communication).