Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Course Project

Secure Cloud Storage
Project Report

Candidates:
Alekos Interrante Bonadia (S319849)
Lorenzo Sebastiano Mathis (S314875)
Pietro Mazza (S314897)

Referee:
Prof. Alessandro Savino

Contents

List of Figures

Abstract

In this project, we successfully integrated functionality for interacting with the SECube hardware module into Cryptomator, facilitating the generation and secure storage of cryptographic passphrases. Building upon an existing software foundation, we meticulously defined a process that allows the SECube module and the application to securely exchange strong keys by leveraging libraries provided by the SECube Project, alongside the Java Native Interface (JNI), which enabled seamless interaction between the Java and C++ components of the system.

Our subsequent efforts focused on automating the process of generating secure passphrases directly on the SECube module, intended for use within the Cryptomator platform. Given the constraints associated with the SECube's inability to export cryptographic keys generated directly on the hardware module, we implemented an alternative approach for password generation. This involved the initial creation of a cryptographic key via the module's `L1` library, utilizing the True Random Number Generator (TRNG) for robust entropy. In conjunction with this, two additional values—a plaintext and an initialization vector—were generated through less secure methods using the `L0` library which internally resort to system library (dev/urandom on Unix, BCryptGenRandom from bcrypt or CryptGenRandom from wincrypt or rand from c standard, with this priority order, on Windows). To generate the passphrase, we encrypted the plaintext using the AES-256 encryption algorithm in CTR mode, employing the TRNG-generated key and initialization vector.

Through this methodology, we were able to derive secure and exportable keys, which were then systematically stored within a secure SQLite database residing on the SECube module's internal memory. To achieve secure storage and retrieval of the keys, we utilized the `SEFile` library, which ensured that the cryptographic material remained protected and could only be accessed following successful authentication to the hardware module. Lastly, we exposed these key management and encryption functionalities to the Cryptomator application by creating a C++ library, which was interfaced through a JNI connector to enable smooth integration with Cryptomator's existing architecture.

CHAPTER 1

Introduction

To ensure the privacy of data uploaded to the cloud, a viable solution is to encrypt the data prior to sharing it. Cryptomator provides a comprehensive solution for this purpose, allowing users to create virtual drives whose contents are encrypted using AES-256. Once the Vault is locked, the data is encrypted, ensuring that files can be uploaded to the cloud securely and with full protection.
In order to securely generate and manage strong passwords that offer robust protection for our data, it is possible to employ a Key Management System (KMS). For this purpose, we utilize the SECube, which, despite being a Hardware Security Module (HSM), provides all the necessary functionalities to create strong cryptographic keys and securely store them. The functionalities we have implemented are based on predefined libraries, specifically leveraging the L1 library for key generation and the SEFile library (a Level 2 library) for secure storage.
Throughout the course of the project, we encountered three primary challenges: ensuring secure interaction between Java and C++, generating keys with the SECube that could be exported outside the hardware module, and securely storing the generated keys.

- To manage interactions between Java and C++, we employed the Java Native Interface (JNI), a foreign function interface framework that enables Java code running in the Java Virtual Machine (JVM) to invoke and be invoked by native applications and libraries written in languages such as C and C++.

- The SECube L1 library provides robust functionality for generating secure keys; however, it does not include a method for exporting these keys. As a solution, we developed the passphrases used in Cryptomator by starting with a pseudo-random payload generated via a method from the L0 library (which internally relies on system-provided PRNGs, such as the random libraries in Linux and Windows). This payload was subsequently encrypted using a secure key generated by the SECube's True Random Number Generator (TRNG), employing AES-256 encryption in CTR mode. The initialization vector (IV) is generated using the same process as the payload. Consequently, we obtain exportable keys that are also secure under IND-CPA (Indistinguishability under Chosen Plaintext Attack) since all three parameters are regenerated for each operation and using AES-CTR encryption algorithm.

- For secure storage of the generated keys, we rely on the SEFile library, which facilitates secure storage within the card in a database that can only be accessed after authentication using a custom version of SQLite natively encrypted (SEcubeDB library).

CHAPTER 2

Background

2.1 Cryptomator

Cryptomator is an open-source encryption software that provides a secure and convenient way to encrypt and decrypt files and folders. It works by creating virtual disks, known as ”vaults,” that are encrypted using the AES-256 algorithm. When a user opens a vault, its contents are decrypted and can be accessed as if they were on a regular disk. Once the vault is closed, its contents are re-encrypted, making them inaccessible to unauthorized parties.

Cryptomator offers several key features:

- **Cross-platform compatibility:** It runs on Windows, macOS, Linux, Android, and iOS, allowing users to access their encrypted data from various devices.
- **Password-based encryption:** Vaults are protected by a strong passphrase, providing a simple and secure way to access encrypted data.
- **Cloud integration:** Cryptomator can be used to encrypt data before storing it in cloud storage services like Dropbox, Google Drive, or OneDrive, ensuring that the data remains private even if the cloud service is compromised.
- **Synchronization:** Changes made to encrypted files are automatically synchronized across multiple devices, keeping data up-to-date.
- **Key Derivation Functions (KDFs)** Cryptomator uses Key Derivation Functions (KDFs), specifically PBKDF2 with HMAC-SHA-256, to enhance the security of password-based encryption. This approach ensures that even if a vault password is compromised, it remains computationally difficult to recover the encryption key, providing an additional layer of security for the encrypted data.
- **Metadata Encryption** In addition to encrypting the content of files, Cryptomator also encrypts metadata, such as file names and timestamps. This feature ensures that sensitive information about the files remains protected, preventing unauthorized access to metadata and offering an extra layer of privacy.

2.2 SECube

SECube is a hardware security module (HSM) that provides robust cryptographic capabilities to protect sensitive data. It is a security-oriented 3D SiP (Sys-tem in Package) that integrates three main components: an ARM Cortex M4 microcontroller, a flexible FPGA, and an EAL5+ certified smart card. These components work together to offer a comprehensive security platform capable of handling various cryptographic operations and secure key management tasks. The SECube platform is designed to be open-source, facilitating ease of integration and customization for various security applications.

2.2.1 L0 Library

The L0 library provides the fundamental communication functionalities required to interact with the SECube device. This layer primarily handles the following:

- **Provisioning APIs:** Used for initializing and setting up the SECube device, including the assignment of a serial number.
- **Communication APIs:** These APIs manage the sending and receiving of command and data packets between the host and the SECube device.
- **Commodities APIs:** These functions assist in discovering and selecting connected SECube devices, and handling other low-level operations.

2.2.2 L1 Library

The L1 library builds on the L0 layer to provide more advanced functionalities necessary for secure applications. Key features of the L1 library include:

- **Authentication:** Implements multi-factor login processes, allowing administrators and users to authenticate securely.
- **Key Management:** Facilitates manual key management operations such as adding, deleting, and searching for cryptographic keys within the SECube device.
- **Cryptographic Operations:** Provides APIs for encryption, decryption, and hashing using supported algorithms like AES-256 and SHA-256.
- **Device Diagnostics:** Enables diagnostic checks on the SECube device, including listing available keys and supported algorithms.

2.2.3 L2 Library

The L2 library abstracts the complexities of the lower layers, providing higher-level APIs that simplify the development of secure applications. The L2 library includes the following specialized components:

SEFile

SEFile is a secure file system interface that ensures data-at-rest protection. It integrates with the SECube's cryptographic capabilities to encrypt and decrypt files transparently. SEFile also supports a custom version of SQLite3, which uses the SECube to encrypt database entries, ensuring that all stored data remains confidential and tamper-proof.

SELink

SELink is a communication protocol designed to protect data-in-motion. It facilitates secure data exchange between SECube devices and hosts over TCP/IP. SELink uses the cryptographic features provided by the L1 library to establish and maintain secure communication channels, ensuring that transmitted data is encrypted and authenticated.

SEKey

SEKey is a distributed key management system that uses SECube devices as nodes in a secure network. It manages cryptographic keys across multiple devices, ensuring that keys are generated, distributed, and stored securely. SEKey simplifies the process of key management in complex systems, allowing for secure and efficient encryption and decryption operations across distributed environments.

2.3 JNI (Java Native Interface)

JNI is a programming interface that allows Java code to interact with native code (C/C++). This enables Java applications to leverage the performance and capabilities of native libraries. In the context of integrating Cryptomator and SECube, JNI can be used to:

- **Call native functions:** Java code can call native functions implemented in C/C++ to perform cryptographic operations or interact with the SECube device.
- **Pass data between Java and native code:** Data can be passed between Java and native code using JNI, allowing for efficient data processing and manipulation.
- **Access native resources:** JNI can be used to access native resources, such as hardware devices or system libraries, that are not directly accessible from Java.

CHAPTER 3

Implementation Details

Throughout the execution of the project, we were required to make certain foundational assumptions regarding the security mechanisms of the software and hardware systems we employed. First, we assumed the method used for storing `se3Key` within the smart card to be secure, ensuring that the communication between the device and the system was immune to potential Man-In-The-Middle (MITM) attacks. Furthermore, we presumed that the software we utilized had the capacity to manage cryptographic keys securely, even if only temporarily. With these assumptions firmly in place, our focus shifted toward more complex challenges, including the secure generation of cryptographic keys robust enough to ensure a high level of security. We also emphasized the secure storage, management, and utilization of these keys outside the hardware device in which they were initially generated, and the development of a secure interface between Java (which constitutes the core of Cryptomator) and C++ (the core of SECube).

One of the principal challenges we encountered was related to SECube's inherent inability to export cryptographic keys that are generated internally. This limitation led us to explore two potential approaches:

1. The first approach consists of modifying the base L0 library of SECube to expose a new functionality for the generation of random values through the device's physical True Random Number Generator (TRNG).

2. The second approach consists of developing a mechanism that relies on the key generated using the TRNG without export that key and store securely the result on the device memory.

The first solution, however, would have necessitated modifying the device's firmware and developing a secure protocol for exchanging cryptographic data with the board. Given the complexity of this task, we ultimately opted for a secure key generation process that leveraged the TRNG embedded in the SECube device.

To implement this solution, we utilized SECube's L1 library, which integrates the TRNG to generate a strong 32-bit cryptographic non-exportable key. Additionally, we relied on SECube's L0 library to generate a payload and an initialization vector (IV), respectively 32 and 16 bytes. The L0 library relies on system libraries for these operations: on UNIX-based systems, the '/dev/urandom' source was used; while on Windows systems, we employed the 'BCryptGenRandom' function from the 'bcrypt.h' library (if it fails, the system would fallback to 'CryptGenRandom' from the 'wincrypt.h' library, and, as a last resort, to the standard C library's 'rand' function). The inclusion of these fallback mechanisms, particularly the additional layers before reaching the 'rand' function, was intended to minimize the use of this latter function, which is widely regarded as insecure for cryptographic purposes.

Using the values generated —the key, payload, and IV— we proceeded to encrypt the payload with the key through the 'L1Encrypt' function from SECube's L1 library. This encryption was performed using the AES-256 encryption algorithm in CTR (Counter) mode, which is already integrated into the SECube system. The result of this encryption process was a 256-byte string. Since the key, payload, and IV are regenerated for each new password, this method ensures that the output is resistant to chosen plaintext attacks (IND-CPA), thereby offering a strong defense against such cryptographic threats and providing an adeguate level of entrophy in the output.

This process allowed us to generate a 64-character hexadecimal string, which we regard as a cryptographically strong password based on the values and methodology used in its generation. Importantly, this password can be utilized not only within the secure hardware environment of the SECube but also in external applications, such as Cryptomator, by leveraging the computational capabilities of the host system.

The generated password is securely stored using the 'SEFileDatabaseStoreKey' function, which is built upon the level-2 SEFile library. This storage process includes associating the password with its respective size and a unique ID linked to the Cryptomator-generated vault. When unlocking a Cryptomator vault, the system internally calls the private function 'SEFileDatabaseRetrieveKey', which, using the vault's ID, retrieves the corresponding cryptographic key for access.

This process ensures a robust and secure interaction between hardware-level key generation and external software applications, balancing the high security of hardware cryptography with the flexibility and computational power of external systems.

CHAPTER 4

Results

The result of this project is the integration of Cryptomator with the SECube device for the generation and storage of the vault passphrases.

On the bottom of the ”Add vault” screen there are the option list with all the connected SECube devices and the text field where the user can insert the User PIN of the selected device.

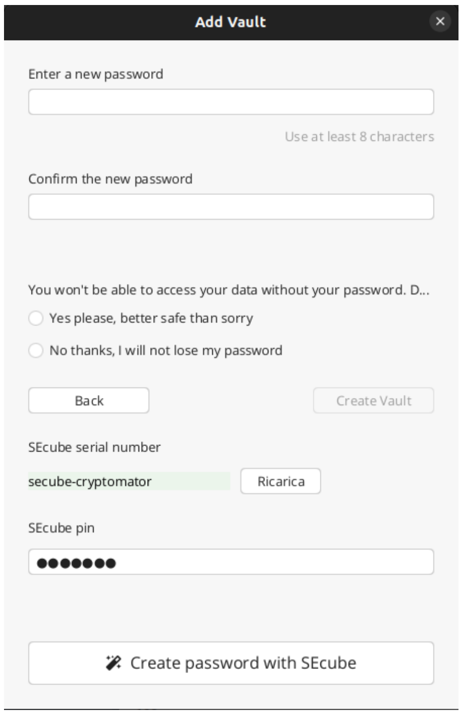
Add Vault
Enter a new password
Use at least 8 characters
Confirm the new password
You won't be able to access your data without your password. D...
Yes please, better safe than sorry
No thanks, I will not lose my password
Back
Create Vault
SEcube serial number
secube-cryptomator
Ricarica
SEcube pin
●●●●●●●
Create password with SEcube

Figure 4.1: Add vault screen

If the vault is created with SECube, the unlock screen ask to the user to select a connected SECube device from the list and enter the User PIN.

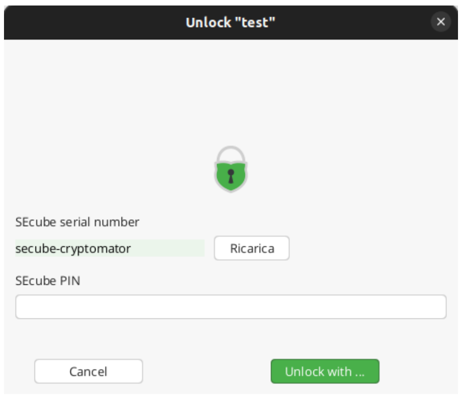

Figure 4.2: Unlock vault screen

4.1 Known Issues

- **MacOS**: due to the impossibility to get a MacOS system (we try to use the QuickEmu environment to run a MacOS virtual machine but with very low results), the connector is untested on this operative system.

4.2 Future Work

- **Improving the TRNG usage**: the current implementation relies on the TRNG and the capability of AES algorithm to produce very good outputs in terms of entropy. A future work can be the implementation of a direct interface to the TRNG modifying the SECube firmware and L0 capabilities to expose a API to use the TRNG directly.
- **Vault ID generation**: at the moment, Cryptomator generates the IDs relying on the `Pseudo` RNG provided by Java. These IDs are a `9 bytes` random sequence, that can be considered enough to avoid duplicate IDs. A future improvement can be to use UUID technologies (UUIDv1 combines devices information and time information to generate the UUID; since we can assume that the same device cannot be capable to generate the two different vaults, the output can be considered universally unique).

Bibliography

[1] Cryptomator Team, *Cryptomator User Guide*,
https://docs.cryptomator.org/en/1.5/desktop/setup/

[2] SECube Project, *SECube SDK Documentation*,
https://github.com/SEcube-Project/SEcube-SDK/blob/master/wiki/wiki_rel_012.pdf

[3] SECube Project, *SEFile Documentation*,
https://github.com/SEcube-Project/SEfile/blob/master/wiki/wiki-sefile.pdf

[4] SECube Project, *SELink Documentation*,
https://github.com/SEcube-Project/SElink/blob/master/wiki/wiki-selink.pdf

[5] SECube Project, *SEKey Documentation*,
https://github.com/SEcube-Project/SEkey/blob/master/wiki/wiki-sekey.pdf

APPENDIX A

User Manual

A.1 User Manual Overview

The application is tested on Linux and Windows 11 platform

Cryptomator project is migrated from Maven to Gradle (KDL) tool. The `build.gradle.kts` script is responsible for all the deployment operations:

- **Native Headers Generations**: the task `generateNativeHeader` create the header file for the connector class. If you are using IntellijIdea, you can find a launch configuration to launch this task automatically.
- **Native Library Building process**: the building process relies on CMake. It build the native library described in the `src`
  `main`
  `cpp`
  `CMakeList` file.
- **Java Building Process**: it builds the Cryptomator Java code base. If you are using IntellijIdea, you can find a launch configuration to launch this task automatically, the configuration will launch the previous all task sequentially.
- **Run**: it will run Cryptomator. If you are using IntellijIdea, you can find a launch configuration to launch this task automatically, the configuration will launch the previous all task sequentially.

If you are not using IntellijIdea, you can run the program by using gradle:

1. `$ cd /path/to/cryptomator/folder`
2. `$ ./gradlew run`

A.2 Required Toolchains and Documentation

A.2.1 Cryptomator

Cryptomator is used in this project for secure encryption and decryption of files. Please refer to the Cryptomator User Guide for detailed instructions on installing and using Cryptomator.

To run the project with Gradle, you need:

1. Java 17, the jdk should be registered in enviroment variables at `JAVA_HOME`, othewise the main CMakeList file must be updated.
2. Gradle (tested on version 8.6) and gradle wrapper correctly installed (if not present in `gladle/wrapper` go in that folder and type `$ gradle wrapper`.
3. The SECube building requirements

A.2.2 SECube Toolchain and Libraries

The SECube hardware security module and its associated libraries are crucial components of this project. The following steps outline how to set up the SECube environment:

- Install the SECube toolchain as described in the SECube SDK Documentation.
- For detailed information on the libraries used (L0, L1, and L2), refer to the respective sections in the documentation. The relevant sections include:
  - L0 Library: See Chapter 4.2.1 in the SECube SDK Documentation.
  - L1 Library: See Chapter 4.2.2 in the SECube SDK Documentation.
  - L2 Libraries:
    - **SEFile:** See the SEFile Documentation.
    - **SELink:** See the SELink Documentation.
    - **SEKey:** See the SEKey Documentation.

The subfolder `/src/main/cpp` contains a CLion configuration is ready to use to build the library or the example project.

The requirements are:

- CMake toolchain
- (on linux) the normal cpp toolchain.
- (on windows) visual studio for C++ desktop application, note that the various element should be put in the environment variables to work (`nmake`, `cl` and other)

To build the library only without using the Gradle script:

1. `$ cd /src/main/build/cpp`
2. `$ cmake -S . -B build`
3. `$ cmake --build build`

Note: if you do not have the `org_cryptomator_secube_SECubeConnector.h` file, you need to generate it with `javac -h`. For more details, see the task `generateNativeHeaders` in the gradle file.

A.2.3 Reproducing the Demo

To reproduce the demo, follow these steps:

1. Unzip or clone the project repository
2. `$ cd /project/folder`
3. `$ gradlew run`
4. The Cryptomator project will autoconfigure, build all native and Java source, then run.
5. Connect the SECube board using the USB cable (connected to the J5000 port)
6. Create a new vauld or unlock a previous created one.

This guide assume that:

1. The devices is correctly initialized, if not you can use `ERROR_CODE SECubeFactoryInitRoutine()` function provided by the `SECubeConfiguration` API.
2. The secure database is already created. If not, you can use `ERROR_CODE SECubeDBKeyInitRoutine()` to (re)generate the database master encryption key and `ERROR_CODE SECubeDBInitRoutine()` to (re)create the SQLite3 database.

APPENDIX B

API

The SEcube API is a level 3 library providing the functionality:

- to **generate onboard a passphrase storing it** inside the secure database for a given vault ID.
- to **retrieve the previous generated passphrase** for a given vault ID.
- to **get the list of devices connected** to the host (this is an interface to the L0 functionality).

B.1 Public API

-
  ```
  vector<pair<string, string>> EnumerateSECubeDevices()
  ```

  It returns a vector containing the pair (mount point, serial number) for all connected SECube devices. If no device is connected, the function will return an empty vector.

-
  ```
  void GenerateSecurePassword(
      const string &vaultID,
      const string &serial,
      const string &pin,
      const uint8_t * password,
      int size
  );
  ```

  It generate a random sequence of `size` bytes and put them into `password` using the `SECubeGenerateSecurePassword` function and storing it inside the database for the key `valutID` using `SEFileDatabaseStoreKey`.
  The parameters `serial` and `pin` are used to login into the selected device as user.

  The function can throw:

  - `SECubeLoginException` if the login fails.
  - `SECubeKeyException` if the key generation process fails.
  - `SECubeDatabaseException` if the database operations fails.
  - `SECubeException` if any generic error raises.

-
  ```
  void RetrieveSecurePassword(
      const std::string &vaultID,
      const std::string &serial,
      const std::string &pin,
      const uint8_t * password,
      int size
  );
  ```

  It retrieves a previously generated sequence of `size` bytes from the secure database for the key `vaultID` and puts it in the `password` buffer using the `SEFileDatabaseRetrieveKey` function. The parameters `serial` and `pin` are used to log into the selected device as a user.

  The function can throw:

  – `SECubeLoginException` if the login fails due to `L1Login` failure.

  – `SECubeException` if the L1 object is `null` or if the `serial` and `pin` strings are too long.

B.2 Private API

-
  ```
  void SECubeLogin(const std::string &serial, const std::string &pin);
  ```

  Perform the login as user on the SECube device identified with `serial` using `pin`. The device will be selected in the `L1` object; eventual previous sessions will be forced out.

  The function can throw:

  – `SECubeLoginException` if the login fails.

  – `SECubeDatabaseException` if the database operations fails.

  – `SECubeException` if any generic error raises.

-
  ```
  std::string SEFileDatabasePath(const char * databaseName);
  ```

  Return the full path to the database file stored in the SECube device memory. The file name is defined by the `DB_NAME` constant inside the `configuration.h` file, the mount point is retrieved using the `L0` utilities.

-
  ```
  void SEFileDatabaseRetrieveKey(
      const char * vaultId,
      const char uint8_t * key,
      int size
  );
  ```

  Retrieve the key sequence associated to the `vaultId` from the secure database using the SEcureDB APIs. It will copy the first `size` bytes inside the `key` buffer.

  The function can throw:

  – `SECubeDatabaseException` if the database operations fails.

  – `SECubeException` if SEFile `secure_init` fails or the `L0`/`L1` are `null`.

-
  ```
  void SEFileDatabaseStoreKey(
      const char * vaultId,
      const uint8_t * key,
      int size
  );
  ```

  Store the key sequence for the primary key `vaultId` into the database using the SEcureDB APIs. It will store the first `size` bytes from the `key` buffer.

  The function can throw:

  – `SECubeDatabaseException` if the database operations fails.

  – `SECubeException` if SEFile `secure_init` fails or the `L0`/`L1` are `null`.

-
  ```
  void SECubeGenerateSecurePassword(const uint8_t * password, int size);
  ```

  It generates a new secure bytes sequence of `size` length and puts it inside the `password` buffer. The generation procedure is:

  1. Clean up old temporary keys if any exist, then generate a new `se3Key` relying on the onboard TRNG.
  2. Generate the AES `IV` using the `L0Support::Se3Rand`, on Linux it will use `dev/urandom`.
  3. Generate a random `plaintext` with length `size` using the `L0Support::Se3Rand`.
  4. Encrypt the `plaintext` using **AES-256-CTR** with key and IV previously generated.
  5. Copy the result inside the `password` buffer.

  The function can throw:

  – `SECubeLoginException` if no user is logged in.

  – `SECubeKeyException` if key generation or the encryption operation fails.

-
  ```
  void SECubeGenerateKey(uint32_t keyId);
  ```

  It generates a new secure `se3Key` using the onboard TRNG with id `TMP_KEY_ID` defined in `configuration.h`.

  The function can throw:

  – `SECubeLoginException` if no user is logged in.

  – `SECubeKeyException` if key generation fails.

-
  ```
  void SECubeDeleteKey(uint32_t keyId);
  ```

  It deletes the `se3Key` with id `TMP_KEY_ID` defined in `configuration.h`.

  The function can throw:

  – `SECubeLoginException` if no user is logged in.

  – `SECubeKeyException` if key deletion fails.

B.3 Utilities

The file `SECubeUtils.cpp/h` provides some utility functions for passphrase conversion, handling SQLite3 errors, and performing some operations across JNI.

-
  ```
  void JNIDebugPrint(JNIEnv *env, const char *msg);
  ```

  It prints a debug message on Java stdout (`System.out.println`).

-
  ```
  std::string uint8ToHex(const uint8_t *data, size_t len);
  ```

  It converts the `data` bytes buffer with `len` length into a hexadecimal `string` which will be returned.

-
  ```
  void SQLite3ErrorHandler(sqlite3 *db, const char *msg);
  ```

  It handles a SQLite3 error by extracting the database error using the SQLite3 APIs and combining it with a custom message, if provided. It will throw a `SECubeDatabaseException` with the resulting message.

-
  ```
  void JNIThrowException(JNIEnv *env, std::exception &e);
  ```

  It gets a C++ exception, maps it to the corresponding Java exception class, and throws it via JNI to the Java runtime environment.

B.4 SECube Connector

The SECube Connector is the piece of code written partially in Java, partially in C++ that connects Crytomator to the SECube APIs.
In Java, it is represented by the `SECubeConnector` class that provides a unified interface to the native functions. In C++, it is represented by the `SECubeConnector.cpp` file that implements the native functions.

The Java implementation is a thread-safe singleton class providing a unified entry-point for all the operations. The native methods are kept private, and the features are exposed by wrapper methods.

Its public interface is composed of three methods:

1.
   ```
   public Map<String, String> GetDevices()
   ```

   The `GetDevices` method returns a map containing, for each connected device, the mount point and the serial number. If no device is connected, the returned map will be empty.

2.
```
public String GenerateSecurePassword(
        String vaultID, String SECubeSerial, String SECubePIN
    )
```

The `GenerateSecurePassword` method requests the SECube device to generate a new passphrase for the given `vaultID`, which will be stored onboard and returned as a `String`. The `SECubeSerial` and `SECubePIN` are the credentials to log in as user to the SECube device.

The method throws various types of `SECubeException`:

- `SECubeLoginException` if the login fails.
- `SECubeKeyException` if the key generation process fails.
- `SECubeDatabaseException` if the database operations fail.
- `SECubeException` if any generic error occurs.

3.
```
public String GetSecurePassword(
        String vaultID, String SECubeSerial, String SECubePIN
    )
```

The `GetSecurePassword` method requests the SECube device to retrieve the passphrase associated with the given `vaultID` that is stored onboard and returned as a `String`. The `SECubeSerial` and `SECubePIN` are the credentials to log in as user to the SECube device.

The method throws various types of `SECubeException`:

- `SECubeLoginException` if the login fails.
- `SECubeDatabaseException` if the database operations fail.
- `SECubeException` if any generic error occurs.

These public methods rely on the corresponding native methods declared in the same Java class and implemented in the `SECubeConnector.cpp` C++ file.

The C++ implementation of the connector is completely contained inside the `SECubeConnector.cpp` file, providing the function declarations in the `org_cryptomator_secube_SECubeConnector.h` header generted automatically by the `javac` compiler. These functions are responsible for:

- Converting the parameters from the JNI format into a C++ compatible format, for example from `jstring` to `const char *`.
- Invoking the SECubeAPI functions.
- Building the return value in a JNI/Java compatible format, for example from the `vector<pair<string,string>>` used by the C++ API to store the device list to the `Map<String,String>` used in Java.

APPENDIX C

Device Configuration and Examples

The SECubeAPI can be used both with the Java connector in the form of shared library and as standalone static library. At the moment, the CMakeFile supports both compilation configurations. The choice between the two is done by setting a CMake parameter in the file; if the `CMAKE_BUILD_CONFIGURATION_TYPE` is set to:

- `"Cryptomator"`, it will be compiled as the shared library usable by Cryptomator or any other Java program (with the appropriate adjustments to the names).
- `"Example"` or any other value, it will be compiled as an executable with the main in `example.cpp` file.

In the second case, the `SECubeConnector` will be excluded from the sources, and the two utility methods used to deal with JNI will also be excluded. The `SECubeConfiguration.cpp/h` module is included in the sources list.

C.1 SECubeConfiguration utilities

The SECubeConfiguration module contains various useful functions that can be used to configure, manage, and debug a SECube device:

1.
```
ERROR_CODE SECubeDeviceList(
    vector<pair<string, string>>& devices,
    bool select = false
);
```

   This function is equivalent to the `EnumerateSECubeDevices()`, it also provides the possibility to ask the user which SECube device to select and use from that moment by setting the `select` flag to `true` (by default, `false`).

2.
```
ERROR_CODE SECubeFactoryInitRoutine();
```

   This function starts the procedure to set up a SECube device from a factory reset. It asks the user for the **serial number**, the **admin pin**, and the **user pin**. It also performs some tests on the pins.

3. `ERROR_CODE SECubeDBKeyInitRoutine();`

   This function starts the procedure to generate a new `se3Key` for the secure database. The key will be generated with a length of **256 bits** and it uses the `DB_KEY_ID` provided in the `configuration.h`. **Do not change this declaration after setting up the key, it is the same declaration used by the library in "cryptomator mode"**. If a key is already present with that ID, it will ask the user if they want to regenerate the key.

---

4. `ERROR_CODE SECubeDBInitRoutine();`

   This function starts the procedure to set up the secure database assuming that the key is already present. If a file with the same name (`DB_NAME` in `configuration.h`) already exists, it will ask the user if they want to delete it and recreate a new database from scratch. **Note that all the keys stored in the database will be lost!!!**.

---

5. `ERROR_CODE SECubeDBDebugRead();`

   This function will read all the database content and print it to `stdout`, censoring the passphrase bytes value.

C.2 Database structure

The database is composed of a single table with the following schema:

```
TABLE vaults (
    vault_id TEXT PRIMARY KEY,
    key BLOB NOT NULL
)
```

The queries performed by the APIs are the following statements:

- `SELECT key FROM vaults WHERE vault_id = ?`

- `INSERT INTO vaults (vault_id, key) VALUES (?, ?)`

C.3 Example

```
#include <iostream>
#include "SECubeConfiguration.h"
#include "SECubeAPI.h"
#include "SECubeUtils.h"

using namespace std;

int main() {
    // Listing and selecting the device
    vector<pair<string, string>> devices;
    SECubeDeviceList(devices, true);

    // Trying to read the database
    SECubeDBDebugRead();

    // Testing the APIs

    vector<pair<string, string>> devs = EnumerateSECubeDevices();
    for (auto &dev: devs) {
        cout << "Device: " << dev.first << " - " << dev.second << endl;
    }

    string vaultID = "vault1";
    string serial = devs[0].second;
    string pin = "user1234";
    uint8_t password[32] = {0};

    try {
        GenerateSecurePassword(vaultID, serial, pin, password, 32);

        cout << "Generated password" << endl;
        cout << "Raw: ";
        for (unsigned char i : password) {
            cout << i;
        }
        cout << endl;

        string hexPass = uint8ToHex(password, 32);
        cout << "Hex: " << hexPass << endl;

        uint8_t password_r[32] = {0};
        RetrieveSecurePassword(vaultID, serial, pin, password_r, 32);

        cout << "\n\n-----------\n" << endl;
        cout << "Retrieved password" << endl;
        cout << "Raw: ";
        for (unsigned char i : password) {
            cout << i;
        }
        cout << endl;

        string hexPass_r = uint8ToHex(password, 32);
        cout << "Hex: " << hexPass_r << endl;

    }
    catch (SECubeException &e) {
        cout << e.what() << endl;
    }
}
```