SEfile™
Data-at-Rest Protection

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, error 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

SE*file*™ is a library that implements an encrypted virtual file system. Basically, **SE*file*™** gives you APIs that emulate the behavior of system calls such as `open()` and `close()`. The great advantage of **SE*file*™** is that, while standard OS system calls for the file system allow you to manage traditional files, the APIs of **SE*file*™** do exactly the same thing with files that are constantly encrypted thanks to the **SE*cube*™** device. **SE*file*™** works as a wrapper around the traditional file system interfaces of Windows and Unix, 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 use the secure virtual file system interface of **SE*file*™** instead of the standard file system interface of the OS.
If you need additional details about the **SE*file*™** library, please check out the documentation of the **SE*cube*™** Open Source SDK[1].

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 **SE*cube*™** 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*™** .

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

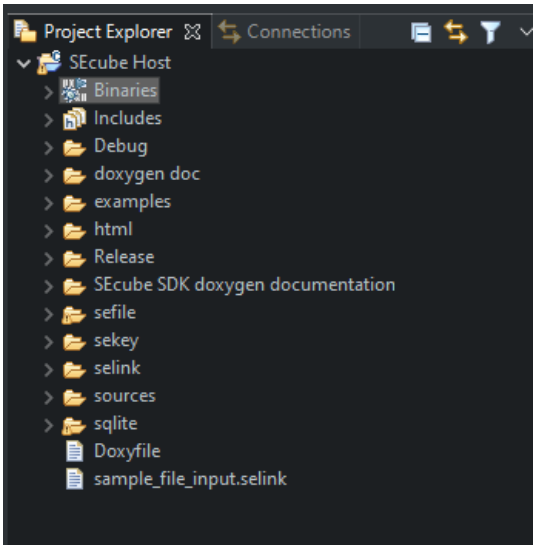

Figure 1: An example of how a **SE*cube*™** project can be structured.

3 SE*file*™ environment setup

SE*file*™ 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*file*™** :

1. go to https://www.secube.eu/resources/open-sources-sdk/ and download the Open Source **SE*cube*™** SDK;
2. extract the downloaded archive, you will find a folder named 'SEcube Host SDK 1.5.1';
3. go to https://www.secube.eu/resources/open-sources-sdk/ and download the **SE*file*™** SDK;
4. extract the downloaded archive and copy the 'sefile' folder into `SEcube Host SDK 1.5.1/environment/SEcube Host/` (see Figure 1 as a general example);
5. launch the Eclipse IDE setting the workspace to `SEcube Host SDK 1.5.1/environment/`, you will be able to see the 'SEcube Host' project.

Notice that the 'sefile' folder contains several files. Some of these are related to the Secure Database library implemented by means of **SE*file*™** and SQLite. The source code of the Secure Database library is stored in the 'sefile' folder simply because it shares a significant amount of code with **SE*file*™** itself; therefore, to avoid unnecessary code duplication, everything was placed in the same folder.
Important notice: if you do not want to use **SE*file*™** along with **SE*key*™** , you must follow these additional steps in order to complete the setup:

1. open the file named `SEfile.cpp` and comment the line where the `USING_SEKEY` constant is defined;

2. open the file named `environment.h`, you will find a global variable named `SEcube`. This is a pointer to the L1 object that is used to communicate with the **SE*cube*™** ; it is automatically initialized by **SE*key*™** but, since you are not going to use **SE*key*™** , you have to initialize it manually in your application by assigning to the pointer the address of the L1 object thay you created to communicate with the **SE*cube*™** . 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 SE*file*™ APIs

Here we provide a simplified and high-level overview about the **SE*file*™** APIs, notice that there are other functions inside the **SE*file*™** library that are used for internal purposes. The APIs listed here are everything you need to profitably use the library; however, please refer to source code comments to find out more details about the APIs and the other functions of **SE*file*™** . The comments are written using the Doxygen syntax so you can generate the documentation as you please.
Inside the source code of the **SE*file*™** library, you will also find APIs developed to be used exclusively with the SQLite database engine. The name of these functions always begins with 'securedb'. Some of these functions belong to the `SEfile` class, they should not be used explicitly because they are automatically called by the traditional SQLite C interface (i.e. `sqlite3_open()` internally calls `securedb_secure_open()`). The only **SE*file*™** functions reserved to SQLite that you may want to use directly are `securedb_ls()` and `securedb_recrypt()`.

```
uint16_t secure_init(L1 l1, uint32_t keyID, uint16_t crypto)
uint16_t secure_finit()
```

These functions are used to setup the basic attributes of each **SE*file*™** object in association with a file encrytpted with **SE*file*™** . In particular:

- the `l1` parameter is a pointer to the L1 object that is used to communicate with the **SE*cube*™** connected to the host machine;
- the `keyID` parameter is a 4 byte unsigned integer that identifies the key to be used to encrypt or decrypt the data of the file;
- the `crypto` parameter is a 2 byte unsigned integer that identifies the algorithm to be used to encrypt or decrypt the data of the file.

These three attributes are specific to each **SE*file*™** object, obviously a dedicated **SE*file*™** object is required for each file that needs to be managed by **SE*file*™** . The `secure_init()` is used to initialize those attributes, the `secure_finit()` is used to reset those attributes to default values (i.e. *NULL* for the **SE*cube*™** pointer, 0 for the key and the algorithm).
Notice that the only attribute that you always need to setup is the `l1` pointer because **SE*file*™** must communicate with a **SE*cube*™** . On the other hand, the `keyID` and `crypto` attributes need to be set only when you want to create a file. If the file already exists and you simply want to access to it, then **SE*file*™** will automatically adjust the key and the algorithm according to the key ID and algorithm ID specified in the header sector of the file itself (provided that you have the right key stored on your **SE*cube*™**).

The usage of these APIs is not mandatory, you can simply use the constructors available for the **SE*file*™** object. Notice that the destructor of the **SE*file*™** object automatically calls the `secure_finit()` so that the user does not have to worry about resources deallocation.

```
uint16_t secure_open(char *path, int32_t mode, int32_t creation)
```

This function, given the name of a file as plaintext (relative path or absolute path), is used to open an existing file or create a new one. The name of the file is modified with the `crypto_filename()` function, which transforms it into its digest (64 hex chars) computed with SHA-256. Notice that there is also another function, called `secure_create()`, that is used automatically by the `secure_open()` to create a new file; the `secure_create()` should never be called directly because it is intended to be used exclusively by the `secure_open()`.
The `mode` parameter is used to specify read-only or read-write privilege, the `creation` parameter is used to specify the opening policy (i.e. `SEFILE_NEWFILE` forces the creation of the file, `SEFILE_EXISTING` opens an existing file). A real write-only mode has not been implemented since a dedicated `secure_write()` function exists. Notice that you must specify in advance if you want to create the file or if you want to open it; there is not a mode to open it if exists or create it if it does not exists (you can implement it by yourself generating the encrypted file name with `crypto_filename()` and checking if that file exists or not).
If a new file is created, the header sector is filled with appropriate information (i.e. the ID of the encryption key of the **SE*file*™** object upon which the method is called, the ID of the algorithm, the name of the file, etc.), then the header sector is encrypted and signed (except for the `key_id`, `algorithm`, and `nonce_pbkdf2`, as it is needed to check the signature of the header sector itself) before writing it to disk.
If an existing file is opened, the clear text part of the header sector is read to set the correct key ID and algorithm in the **SE*file*™** object, then the rest of the header is decrypted and the signature is checked; if everything is correct and the key can be used for decryption, the file can be used.
Independently from the actual behaviour of the `secure_open()` function, if it succeeds the file pointer is set to the first byte of the first sector placed after the header.
The following algorithm demonstrates how the `secure_open()` works.

Algorithm 1 How a secure file is opened or created

```
function SECURE_OPEN(in path, in mode, in creation)
    if creation == SEFILE_NEWFILE then
        return SECURE_CREATE()
    end if
    // existing file (SEFILE_OPEN) from now on
    generate encrypted filename with crypto_filename()
    OS system call to open the encrypted file according to mode
    set the key ID and algorithm ID according to the header content
    check if the inherited key ID can be used for decryption
    decrypt header and check signature
    return
end function
```

Algorithm 2 How a secure file is created

```
function SECURE_CREATE(in path, in hFile, in mode)
    check if specified key can be used for encryption
    generate encrypted filename with crypto_filename()
    OS system call to create the encrypted file according to mode
    populate the header of the file
```

encrypt and sign the header sector
write the header sector to disk
move the file pointer to the first byte after the header sector
return
end function

```
uint16_t secure_close()
```

This function simply closes the file associated to the SEfile object and deallocates all the resources that were acquired. This function is called automatically by the destructor of the SEfile object, therefore you do not need to call it manually all the time (but you are suggested to, because it is good practice).

```
uint16_t secure_read(uint8_t *dataOut, uint32_t dataOut_len,
   uint32_t *bytesRead)
```

The `secure_read()` function works as the `read()` in Unix and the `ReadFile()` in Windows, adding all the needed operations related to the secure file management.
The number of bytes requested as clear text is provided in `uint32_t dataOut_len` while the actual number of read bytes is stored in `bytesRead`. In details, the operations performed are: starting from the position pointed by the file pointer the function extracts sequentially all the sectors related to the requested portion of data to be read, check for its integrity by looking at the signature, decrypts the sector and concatenates the read data in the output buffer `dataOut`. After that, the file pointer points after the last byte read. A read operation issued requesting a number of bytes that is not aligned to the sector size and is not a multiple of `SEFILE_LOGIC_DATA` will lead to performance degradation, since it still needs to decrypt the whole sector.
The implemented functionality is shown in the following algorithm.

Algorithm 3 How a secure file is read

function *SECURE_READ(out dataOut, in dataOut_len, out bytesRead)*
check if specified key can be used for decryption
set number of read bytes to zero
do
read, decrypt and verify signature of current sector
append decrypted data do dataOut
bytesRead = bytesRead + data read
dataOut_len = dataOut_len - data read
go on with next sector if required
while *dataOut_len > 0*
end function

```
uint16_t secure_write(uint8_t *dataIn, uint32_t dataIn_len)
```

The `secure_write()` function masks the `write()` in Unix and the `WriteFile()` in Windows, adding all the needed operations related to the secure file management. The function writes in the file the data sent as clear text in the buffer. In particular, the function divides the buffer into sectors, then it encrypts and signs each sector and writes it in the specified position in the file. After this operation, the file pointer points after the last byte written.
In this case, it has been chosen to not return the actual number of written bytes since if the operation fails in writing `dataIn_len` bytes it would result as an error.
If a `secure_write()` is issued requesting to write a number of bytes that is not aligned to the

sector size and is not a multiple of `SEFILE_LOGIC_DATA`, since it still needs to decrypt the whole sector, it will lead to performance degradation.
The implemented functionality is shown in the following algorithm.

Algorithm 4 How a secure file is written

```
function SECURE_WRITE(in dataIn, in dataIn_len)
   check if specified key can be used for encryption
   if file pointer not aligned to sector size then
      read, decrypt and verify signature of current sector
      store inside the buffer the sector to be written
   end if
   do
      append data from dataIn to output buffer
      encrypt and sign the sector to be written
      write the sector to disk
      decrement the amount of data still to be written
   while dataIn_len > 0
end function
```

```
uint16_t secure_seek(int32_t offset, int32_t *position, uint8_t
   whence)
```

This function moves the file pointer by `offset` bytes, taking care of the effective byte of user data and skipping the bytes related to the overhead introduced by **SE*file*™** itself (i.e. header sector, signature field and data length). The parameter `whence` is used to choose if the user wants move the file pointer from the beginning of the file, from the current position, or from the end of the file. The `position` parameter is used to store the logic value where the file pointer is set after issuing `secure_seek()`.
If the destination exceeds the file size, the file is resized by adding zeros until the specified position. This function has proper mechanisms to avoid jumping inside the header sector. The implemented functionality is shown in the following algorithm.

Algorithm 5 How a secure file pointer is moved

```
function SECURE_SEEK(in offset, out position, in whence)
   retrieve the size of the file using get_filesize()
   if offset > file size then
      move the file pointer to the last sector using OS system call
      add as many bytes equal to zero as necessary to reach a file size equal to the offset
      return position = current file pointer position
   end if
   computer file pointer destination according to whence
   move the file pointer to destination using OS system call
   return position = destination
end function
```

```
uint16_t secure_truncate(uint32_t size)
```

This function resizes the file to `size` bytes. It takes care of the sectors and leaves the file pointer to the end of the file (after the last byte of user data).
If the specified file is bigger than the original, sectors are filled with zeros, otherwise data in excess are lost. The implemented functionality is shown in the following algorithm.

Algorithm 6 How a secure file is truncated

```
function SECURE_TRUNCATE(in newsize)
    retrieve the size of the file using get_filesize()
    if newsize > file size then
        return SECURE_SEEK(newsize - file size, nullptr, end of file)
    end if
    compute which sector will become the last one of the file
    move file pointer to the computed sector
    read, decrypt and verify the last sector
    keep only the data of that last sector that must be preserved by the truncation
    truncate the file using the OS system call
    write back the previously saved data with the secure_write()
end function
```

```
uint16_t secure_sync()
```

This function is used in case it is needed to be sure that the OS buffers are correctly flushed to the physical file.

```
uint16_t get_secure_context(std::string& filename, std::string *
    keyid, uint16_t *algo)
```

This function, given the clear text name of a file, returns the ID of the key and the ID of the algorithm used by **SE*file*™** to encrypt and authenticate that file. This is useful in many situations, for example when working with other functions like `secure_ls()`.

```
uint16_t secure_recrypt(std::string path, uint32_t key, L1 *
    SEcubeptr)
```

This function is used to decrypt and encrypt again, with a new key, a file managed by **SE*file*™** that was encrypted with a key that is considered not secure anymore. This function ideally should be used together with the **SE*key*™** KMS; however, it can be easily used also without having the KMS running (as long as you resort to keys which are not in the range of IDs managed by the KMS).
The function takes as parameters the clear text name of the file, the key to be used for the new encryption and the pointer to the L1 object used to communicate with the **SE*cube*™** .
If the function succeeds, the old file will be replaced with a new file whose content is identical and encrypted with the new key; the old file will be deleted. If the function fails, no changes are applied.
Notice that the same function is available also for encrypted SQLite databases under the name of `securedb_recrypt()`.

```
uint16_t crypto_filename(char *path, char *enc_name, uint16_t *
    encoded_length)
```

This function computes the encrypted name of the file specified at position `path` and writes the result to `enc_name`; the quantity of bytes written is saved in `encoded_length`. The filename is computed using the SHA-256 algorithm, so there is no decryption function to obtain its clear text name unless the header sector is decrypted. Since the service which computes the SHA-256 works with 32 Bytes block, its result is always on 32 bytes, and it is represented as hexadecimal values in ASCII encoding, meaning that for each byte there will be 2 character, resulting in a 64

characters length.
In any case, this function takes care of parsing `path` so in `enc_name` will be copied everything that comes before a "/" or "\" character to compute just the hash of the filename to encrypt.

```
secure_getfilesize(char *path, uint32_t * position, L1 *
   SEcubeptr)
```

This function is used to retrieve the total logic size (how many bytes of valid data, excluding the **SE*file*™** overhead) of an encrypted file pointed by `path`, the result is stored in `position`. The `SEcubeptr` parameter is a pointer to the L1 object used to communicate with the **SE*cube*™** connected to the host machine. This function does not need to be called upon a **SE*file*™** object but can be normally used simply passing the clear text name of the file. Notice that the logic size of the file will always be smaller than the physical size given the overhead introduced by **SE*file*™** .
The implemented functionality is shown in the following algorithm.

Algorithm 7 How a secure file size is computed

function *SECURE_GETFILESIZE(in path, out position, in SEcubeptr)*
  open the file pointed by path using secure_open()
  move the file pointer to the last sector of the file using OS system call
  if *number of sectors of the file = 1* ***then***
    return *position = 0*
  end if
  read, decrypt, verify the last sector of the file
  *position = ((total file size / sector size) - 1) * valid bytes in each sector + valid bytes in last sector*
  close the file pointed by path using secure_close()
  return *position*
end function

```
secure_ls(string& path, vector<pair<string, string>>& list, L1 *
   SEcubeptr)
```

This function is used to list the content of a directory containing encrypted files and/or directories. The `path` parameter tells to the function where to search, the `list` parameter stores as first element of each pair the name of the file or directory as it appears to the user (i.e. the encrypted file name of a file managed by **SE*file*™**) and as second element the actual name of the file or directory. The third parameter is the pointer to the L1 object used to communicate with the **SE*cube*™** .
Notice that this function works with any file or directory. In particular, if the name to list is not recognized as a name belonging to the 'nomenclature' of **SE*file*™** , it is simply copied as it is. If this function finds encrypted files managed by **SE*file*™** APIs specific for the SQLite database engine, then their names will not be decrypted; to list their real names use instead the `securedb_ls()` function. The implemented functionality is shown in the following algorithm.

Algorithm 8 How to discover the names of encrypted files in a folder

function *SECURE_LS(in path, out list, in SEcubeptr)*
  retrieve list of files and directories within specified directory using OS system call
  do
    if *current element in list is a directory* ***then***
      decrypt directory name
      if *decryption successful* ***then***
        add decrypted name to list

else
add original name to list
end if
else if *current element in list is a file* ***then***
open the file and try to decrypt the header
if *decryption successful* ***then***
add clear text name to list
else
add original name to list
end if
end if
while *all files and directoris within specified directory have been processed*
end function

```
uint16_t secure_mkdir(string& path, L1 *SEcubeptr, uint32_t key)
```

This function masks the `mkdir()` function of the Unix environment and the `CreateDirectory()` function of Windows, but it does not implement the whole functionalities of those functions. Since directories are created using a wrapper to the OS system call, it is not possible to achieve a mechanism like the one employed for regular files, so it has been decided to use this encryption scheme, leveraging to `crypt_dirname()`, just for the name of the directory: the first 8 characters are the hexadecimal representation in ASCII of the key ID, the rest is obtained computing the AES-256-ECB of the name specified as clear text. The `SEcubeptr` parameter is, as usual, the pointer to the L1 object used to communicate with the **SE*cube*™** ; the `key` parameter is the ID of the key to be used to encrypt the name of the directory.

5 Basic SE*file*™ example

Here is a very simple **SE*file*™** example. Suppose that you want to create a text file, write some text, read what you wrote and close the file. You can use the APIs of **SE*file*™** to perform these operations very easily. Everything you need to do is setup a 'SEfile' object with the required parameters (L1 object pointer, ID of the key to be used for encryption, algorithm to be used). A detailed example can be found in the 'examples' folder of the **SE*cube*™** Open Source SDK[2].

```
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;
```

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

```
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();
```

6 How to use SQLite databases encrypted with SE*file*™

Inside the 'sefile' folder mentioned in Section 3, there is a file called `environment.h`. This file contains the declaration of 3 global variables, we are interested in the variable named '`databases`'. This is an array of pointers to '`sefile`' 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 '`sefile`' object used to manage the encrypted SQL database used by **SE*key*™** to store its metadata. If your want to create 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 (the pointer to the L1 object, the key ID and the algorithm);
3. initialize the '`name`' attribute of the '`handleptr`' attribute of your '`sefile`' object with the name of the database file you need to create or open (the name is always the cleartext name, not the encoded name generated by **SE*file*™**);
4. insert the `unique_ptr` into the databases array with `std::move()`;
5. finally, you can begin working with your database using the standard SQLite C interface and the `sqlite3*` pointer to the database.

Notice that the '`sefile`' object will be automatically removed from the list of databases when you call the `sqlite3_close()` function. 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()`.