SElink™ Data-in-Motion Protection

Technical Documentation

Release: February 2021

Proprietary Notice

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 user manual describes how to use the **SE***link*™ library. If you need additional details about the **SE***link*™ library, please check out the documentation of the **SE***cube*™ Open Source SDK[1].
SE*link*™ is a library that allows you to easily encrypt and decrypt data with the **SE***cube*™ without having to worry about low level security details. **SE***link*™ cannot be used without the **SE***cube*™ device.
SE*link*™ is not a tool that runs on top of preexisting software, it requires custom software to be developed specifically to use the **SE***link*™ APIs in order to encrypt and decrypt data. **SE***link*™ is intended to be used for the encryption and decryption of *data in motion*, for the *data at rest* scenario please have a look at **SE***file*™ .
SE*link*™ grants the confidentiality, integrity and authentication of data using AES256-HMAC-SHA256; moreover, it offers the possibility to serialize and deserialize data before/after their transmission over any network (base64 encoding is used). A simplified overview of how **SE***link*™ works is shown in Figure 1.

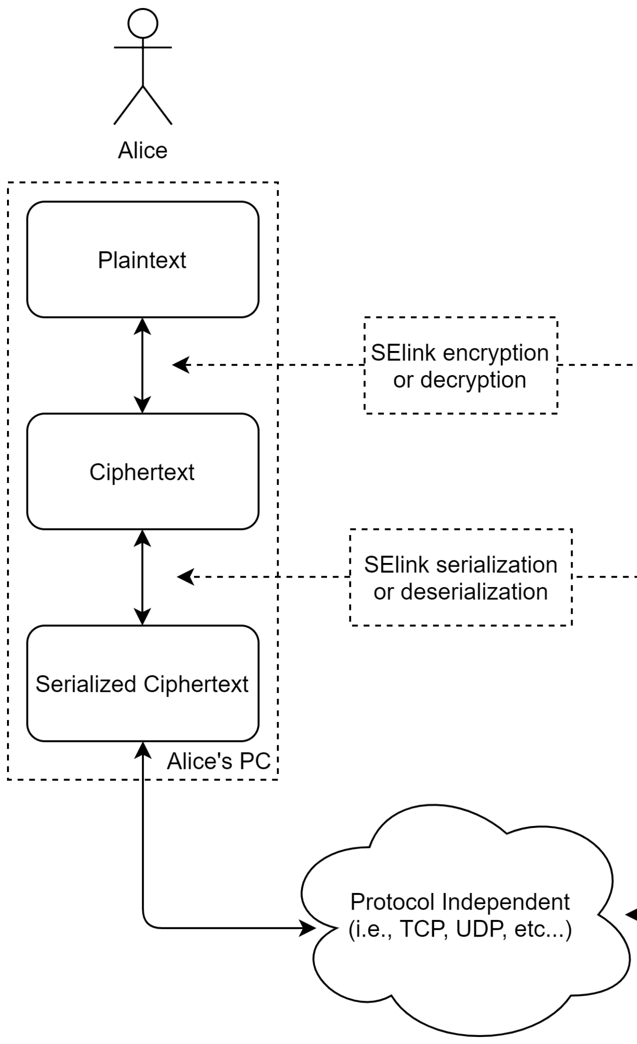

Alice
Bob
Plaintext
Plaintext
SElink encryption or decryption
Ciphertext
Ciphertext
SElink serialization or deserialization
Serialized Ciphertext
Serialized Ciphertext
Alice's PC
Bob's PC
Protocol Independent (i.e., TCP, UDP, etc...)

Figure 1: Simplified **SE***link*™ overview.

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

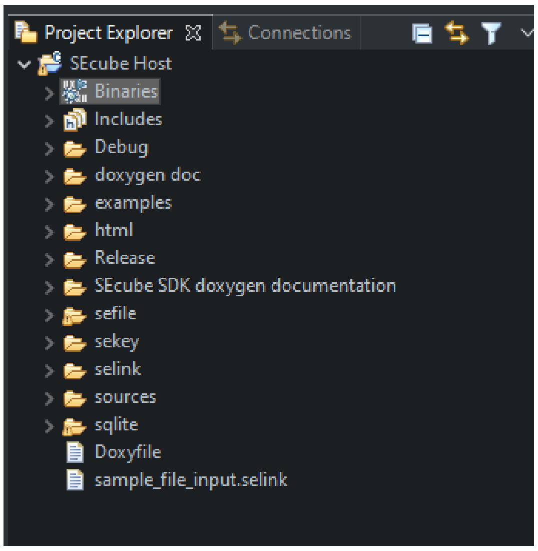
2 SE*cube*™ SDK libraries and dependencies

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 2 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*™ .



SEcube Host
- Binaries
- Includes
- Debug
- doxygen doc
- examples
- html
- Release
- SEcube SDK doxygen documentation
- sefile
- sekey
- selink
- sources
- sqlite
- Doxyfile
- sample_file_input.selink

Figure 2: An example of how a **SE***cube*™ project can be structured.

3 SE*link*™ hardware requirements

SE*link*™ simply requires a **SE***cube*™ device and a host computer. Each person interested in using **SE***link*™ must own a dedicated **SE***cube*™ device.

4 SE*link*™ architectural requirements

SE*link*™ does not require any specific underlying architecture in order to be used. The library encrypts and decrypts the data before they are used by applications at the higher levels, therefore it is completely independent from the protocol and the infrastructure you want to use to distribute the *data in motion*.

5 SE*link*™ environment setup

SE*link*™ 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***link*™ :

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***link*™ SDK;
4. extract the downloaded archive and copy the *selink* folder into the folder where you extracted the Open Source **SE***cube*™ SDK (see Figure 2 as a general example);
5. launch your IDE (i.e., Eclipse) in order to create a dedicated **SE***cube*™ project.

6 The SE*link*™ class

The **SE***link*™ API heavily resorts on the features provided by the L1 layer. In order to manage the data encrypted by **SE***link*™ , the API is based on a ‘SElink‘ object that encapsulates a ‘SE-cube_ciphertext‘ object, the same that is used in L1 API. On top of this, **SE***link*™ introduces also a couple of methods to serialize and deserialize the data, simply because **SE***link*™ is focused on *data in motion*. The serialization is done using the base64 encoding.

```
class SElink{
public:
  SEcube_ciphertext ciphertext;
  void serialize(unique_ptr<char[]>& serialized_data, size_t&
     serialized_size);
  void deserialize(unique_ptr<char[]>& buffer_b64, size_t
     buffer_b64_size);
};
```

In conclusion, a ‘SElink‘ object is, by all means, equal to the ‘SEcube_ciphertext‘ object already used by L1 APIs. The only different is that this new object also embeds the ability to be serialized and deserielized. Moreover, the encryption/decryption functions of **SE*link*™** are based on the ‘SElink‘ object.

7 SE*link*™ APIs

This section provides a brief overview about the **SE*link*™** APIs. For more details about their implementation, please refer to the extensive comments in the source code. You can also process the code through the Doxygen engine in order to generate the documentation.

```
int selink_encrypt_manual(L1 *l1, shared_ptr<uint8_t[]>
   plaintext, size_t plaintext_size, SElink& ciphertext,
   uint32_t key);
```

This function is used to encrypt a plain text with **SE*link*™** . The ‘manual‘ in the name of the function means that the caller is supposed to provide a key ID (an unsigned integer on 4 bytes) to the function, if the **SE*cube*™** contains a key with the provided ID, then the plain text is encrypted. If the function succeeds, the ‘SElink‘ object passed as fourth is filled with the cipher text and other attributes. Then, this object can be serialized in order to be sent to the receiver (serialization might not be always required). Finally, notice that the caller is supposed to provide the plain text as an array of bytes, passing also the size of the array.

```
int selink_encrypt_auto(L1 *l1, shared_ptr<uint8_t[]> plaintext,
    size_t plaintext_size, SElink& ciphertext, vector<std::
   string>& recipient);
```

This function is, by all means, equal to the previous one. The only difference is that the user is not supposed to provide the ID of the key to be used for encryption. The user, in fact, should only provide the list of recipients. This list might include one or multiple users from **SE*key*™** , or even an entire group from **SE*key*™** . The list must be filled with the IDs of the recipients. When issued, this function will automatically resort to **SE*key*™** to find the most suitable key for encryption, then it will fill the `SElink` object passed as fourth parameter with the encrypted data. As usual, serialization can be performed after encryption.

```
int selink_decrypt(L1 *l1, shared_ptr<uint8_t[]>& plaintext,
   size_t& plaintext_size, SElink& ciphertext);
```

This function is used to decrypt a payload that was encrypted by one of the two funcitons described above. Notice that the caller must provide the cipher text object, **SE*link*™** will decrypt the cipher text and it will automatically verify the signature, providing therefore integrity and authentication of the data. In case of success, the plain text will be stored in the byte array passed as second parameter. In case of error, for example because the signatures of the data do not match, an error is returned.

```
void serialize(unique_ptr<char[]>& serialized_data, size_t&
   serialized_size);
```

Serialize the encrypted data stored by a `SElink` object using the base64 encoding. This method belongs to the `SElink` class. The serialized buffer is stored in the first parameter, the size of the serialized buffer is stored in the second parameter.

```
void deserialize(std::unique_ptr<char[]>& buffer_b64, size_t
   buffer_b64_size);
```

Deserialize a data buffer that was previously serialized with the corresponding method of the SElink class. The first parameter is the buffer containing the serialized data, the second parameter is the size of the serialized data. After completion, this method stores the deserialized data inside the SElink object upon which the deserialize function is called. After deserialization, the 'SElink' object can be passed as input to the selink_decrypt() function in order to decrypt the data. This method belongs to the SElink class.

8 SE*link*™ example

This is a very basic example about how to use **SE***link*™ to encrypt and decrypt data. A complete example is provided in 'examples' folder of the **SE***cube*™ Open Source SDK[2]; notice that the source code shown below omits many instructions that are needed in order to use the **SE***cube*™ (i.e., login on the device).

```
/* SENDER */
unique_ptr<L1> l1 = make_unique<L1>();
/* L1 object is used to execute the login on the SEcube before
   using any other API */
shared_ptr<uint8_t[]> plaintext_ptr(new uint8_t[plaintext_size])
   ; /* the plaintext must be initialized with the data that
   must be encrypted */
SElink ciphertext;
int rc = selink_encrypt_manual(l1.get(), plaintext_ptr,
   plaintext_size, ciphertext, key); /* encrypt the plaintext */
if(rc != SELINK_OK){
  // error!
}
size_t serialized_size = 0;
unique_ptr<char[]> serialized;
ciphertext.serialize(serialized, serialized_size); /* serialize
   the data */

/* RECEIVER */
unique_ptr<L1> l1 = make_unique<L1>();
/* L1 object is used to execute the login on the SEcube before
   using any other API */
unique_ptr<char[]> deserialized_ptr = make_unique<char[]>(
   serialized_size); /* buffer with the serialized data */
SElink deserialized; /* object containing the ciphertext */
deserialized.deserialize(deserialized_ptr, serialized_size); /*
   deserialize the data */
shared_ptr<uint8_t[]> decrypted;
int rc = selink_decrypt(l1.get(), decrypted, decrypted_size,
   deserialized); /* decrypt the data */
if(rc != SELINK_OK){
  // error!
}
```

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