SEcube™
RSA and X.509 library

Technical Documentation

Release: October 2021


HW
SW
Open
Source

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


HW
SW
Open
Source

Contents

1 Introduction

The **SE*cube*™** Open Source Security Platform[1] is built around the **SE*cube*™** device, a hardware security module with the capability of performing several algorithms, such as AES-256 and SHA-256. The **SE*cube*™** Open Source Security platform does not support, by default, asymmetric cryptography; the open source firmware of the **SE*cube*™** is entirely focused on symmetric cryptography. In order to overcome this issue, a side-project has been developed. The result is that it is now possible to add an asymmetric cryptography module to the standard **SE*cube*™** Open Source SDK, making the **SE*cube*™** capable of executing the RSA algorithm and of managing X.509 certificates. The support for asymmetric cryptography can be added to the **SE*cube*™** SDK by downloading the source code related to the current project, and extracting it to the folder of the **SE*cube*™** SDK, so that the standard source code gets replaced with the new one.
Please notice that the aim of this project is simply to add to the **SE*cube*™** the possibility of performing basic RSA operations and basic X.509 certificate management, without any intention to be compliant with state-of-the-art techniques and to compete with commercially-available products. If you are not familiar with the **SE*cube*™** Open Source Security Platform, please check out the official documentation[2].

2 Specifications

The asymmetric cryptography library for the **SE*cube*™** supports the following features:

- Injection of RSA keys from the external environment (i.e., the computer) to the **SE*cube*™** ;
- Generation of RSA keys directly inside the **SE*cube*™** ;
- Deletion of RSA keys from the **SE*cube*™** ;
- Creation and deletion of X.509 certificates (stored inside the **SE*cube*™**);
- Reading X.509 certificates from the internal flash memory of the **SE*cube*™** .

The RSA algorithm can be leveraged to implement the secure exchange of symmetric keys (i.e., AES-256 keys) over an untrusted channel.
X.509 certificates can be exploited to use RSA keys in a more structured and ordered manner.

The asymmetric cryptography library has been implemented porting the Mbed TLS library[3] into the firmware of the **SE*cube*™** . Mbed TLS supports countless other features and algorithms, meaning that there is a lot of overhead determined by a lot of code that is not used but must still be compiled into the firmware since it is required by the small set of features that are used by the **SE*cube*™** . There is a plan, however, to review this project in the future, in order to avoid using such a big library (Mbed TLS), using far more compact RSA and X.509 libraries.

3 Limitations

- The size of RSA keys is limited to the following values: 1024 bit, 2048 bit, 4096 bit, 8192 bit.

[1] https://www.secube.blu5group.com/
[2] https://github.com/SEcube-Project/SEcube-SDK
[3] https://tls.mbed.org/

- The maximum size of the data that can be processed by RSA encryption and decryption is equal to the size of the key (in bytes), minus 36 bytes needed for the internal implementation (overhead). `max_RSA_input_size = RSA_key_size - 36`
  If you need to encrypt/decrypt more data, you can split them in multiple segments and process each segment individually (not recommended because of poor performance, it is much better to use a symmetric algorithm and use RSA instead to exchange the symmetric key).

- RSA key generation is very slow, requiring up to 1000 seconds for 2048-bit keys (and even more for longer keys). The value `SE3_TIMEOUT_RSA_KEYGEN` is the timeout that is set by the computer when waiting for the reply of the **SE*cube*™** , it has been configured accordingly (1 hour timeout) in order to avoid errors. You are free to change it, experimenting with lower values (the constant is defined in the file named 'L0_base.h').

4 Library setup

The asymmetric cryptography library has been tested on the following platforms:

- Windows 10 64-bit (20H2, build 19042.1237), Eclipse 2020-12, Mingw-w64 (x86_64-8.1.0-win32-seh-rt_v6-rev0)

In order to setup the asymmetric cryptography library, you must follow these steps:

1. download the official **SE*cube*™** wiki[4];
2. follow the steps of the wiki to download and setup the **SE*cube*™** Open Source SDK (version 1.5.2 recommended);
3. once you are able to compile the code in the SDK and to run some examples, download the source code of the asymmetric cryptography library;
4. extract the content of the downloaded archive inside the folder of the SDK, replacing all required files;
5. compile the SDK again, both the host side and the firmware;
6. flash the firmware on the **SE*cube*™** ;
7. run the examples related to the asymmetric cryptography library.

5 Asymmetric cryptography data structures

The library uses different data structures to store different kinds of data.

```
struct se3AsymmKey{
  uint32_t id;
  uint16_t length;
  std::unique_ptr<uint8_t[]> N;
  std::unique_ptr<uint8_t[]> E;
  std::unique_ptr<uint8_t[]> D;
  uint8_t type;
};
```

[4] https://github.com/SEcube-Project/SEcube-SDK

This class holds the information related to an RSA key pair (private key and public key).

- `id`: the ID of the RSA key pair to be stored inside the **SE*cube*™** flash memory;
- `length`: the length of the key in bytes (specifically, of the modulus and exponents);
- N: modulus of the key;
- E: public exponent;
- D: private exponent (secret);
- `type`: the type of the key (see `L1Key::RSAKeyType`).

```
class X509_certificate{
  uint32_t id;
  uint32_t issuer_key_id;
  uint32_t subject_key_id;
  std::string serial_number;
  std::string not_before;
  std::string not_after;
  std::string issuer_info;
  std::string subject_info;
};
```

This class holds the information related to a X.509 certificate.

- `ID`: the identifier of the certificate (no duplicates are allowed inside the **SE*cube*™** flash memory);
- `issuer_key_id`: ID of the RSA key of the issuer;
- `subject_key_id`: ID of the RSA key of the subject;
- `serial_number`: serial number issued by the certificate authority (hex format);
- `not_before`: time at which the certificate is first considered valid (YYYYMMDDhhmmss format);
- `not_after`: time at which the certificate is no longer considered valid (YYYYMMDDhhmmss format);
- `issuer_info`: comma-separated string containing OID types and values (i.e., ”C=UK, O=ARM, CN=mbed TLS CA”);
- `subject_info`: comma-separated string containing OID types and values (i.e., ”C=UK, O=ARM, CN=mbed TLS CA”).

```
class RSA_IO_data{
  std::unique_ptr<uint8_t[]> data;
  size_t data_size;
};
```

This class holds the information related to the output of an RSA operation (encryption or decryption).

- `data`: the buffer containing the output of the RSA operation (i.e., the encrypted data or the decrypted data);
- `size`: the size of the output.

6 Asymmetric cryptography APIs

Here we provide a simplified and high-level overview about the APIs of this library.

```
void Edit_asymm_key(se3AsymmKey& k, uint16_t op)
```

This function, depending on the value of the second parameter, is used to inject, generate or delete an RSA key pair.
The first parameter is a structure that holds the basic information about the RSA key pair (i.e., ID and key size). Notice that the first parameter must include also the private and public key values (specifically the modulus, the public exponent and the private exponent) in case of key injection. The value of the second parameter, `op`, must be equal to one of the following values:

- `SE3_KEY_OP_INJECT_ASYMM`: inject an RSA key pair inside the **SE*cube*™** flash memory;
- `SE3_KEY_OP_GENERATE_ASYMM`: generate an RSA key pair inside the **SE*cube*™** flash memory;
- `SE3_KEY_OP_DELETE`: delete an RSA key pair from the **SE*cube*™** flash memory.

In case of key deletion, it is sufficient to initialize only the `id` attribute of the first parameter, since the **SE*cube*™** will search for a key with the specified ID and delete it from the internal memory.

```
void Find_asymm_key(uint32_t key_id, bool& found)
```

This function, given a key ID specified as a 4-byte unsigned integer, searches for an RSA key with that ID in the **SE*cube*™** flash memory. If such key ID is found, the `found` parameter is set to `true`, otherwise to `false`.

```
void Get_asymm_key(se3AsymmKey& k)
```

This function, given the key ID set inside the parameter `k`, fill the first parameter with information related to the key with that ID that is stored inside the **SE*cube*™** (if such key exists). Notice that only the public exponent and the modulus of the key are exported from the **SE*cube*™** memory, the private exponent stays inside the **SE*cube*™** .

```
void RSA_enc_dec(std::shared_ptr<uint8_t[]> input, size_t
   inputLen, RSA_IO_data& output, se3AsymmKey& key, bool
   public_key, bool on_the_fly)
```

This function executes the RSA algorithm (unlike symmetric algorithm, with RSA there is no difference between encryption and decryption, it is the same operation) on the array of data that is provided as first parameter. The second parameter is the size of the input, the third parameter is a data structure that contains the output data and its size, the fourth parameter is the RSA key to be used, the fifth parameter is a flag that is `true` when the provided key is a public key and `false` when it is private, the sixth parameter is a flag that is `true` when the provided `key` parameter

includes also the key value (the modulus and the exponents).
When the last parameter is `false`, the `key` parameter can be filled specifying only the ID of the key to be used because, in this case, the key is already stored inside the **SE*cube*™** flash memory. On the contrary, when `on_the_fly` is `true`, the caller of the function is expected to provide the `key` parameter filled with all the information required to execute the operation (modulus, private exponent, public exponent).
Notice that a public key (modulus and public exponent) should be used only for encryption and signature verification. On the contrary, a private key (modulus and private exponent) should be used only for decryption and signature computation.

```
RSA_sign(const std::shared_ptr<uint8_t[]> input, const size_t
   inputLen, std::shared_ptr<uint8_t[]> &signature, size_t &
   signature_size, const se3AsymmKey& key, bool on_the_fly)
```

This function is used to sign a message, which is passed as first parameter; the signature is computed using the RSA algorithm and an asymmetric (private) key. The second parameter is the length of the message to be signed, the third parameter is the signature (output), the fourth is the signature length, the fifth is the key to be used and the seventh is the flag to specify if the key has to be used 'on the fly' (key value in input as fifth parameter) or if the key is stored inside the **SE*cube*™** . The behaviour of the last two parameters is the same as the `RSA_enc_dec()`.

```
void RSA_verify(const std::shared_ptr<uint8_t[]> input, const
   size_t inputLen, const std::shared_ptr<uint8_t[]> signature,
   const size_t signature_size, const se3AsymmKey& key, bool
   on_the_fly, bool &verified)
```

This function is used to verify a signature previously computed. The first two parameters are the original message and its length, the fourth and fifth parameters are the signature to be verified and its length, the sixth and seventh parameters are the key to be used and the flag for the behavior of the function (see `RSA_sign()` and `RSA_enc_dec()`), the last parameter (output) is set to `true` if the signature is valid and to `false` if the signature is not valid.

```
void Edit_certificate(const L1Commands::CertOpEdit op, const
   X509_certificate info)
```

This function is used to manage X.509 certificates. The first parameter, `op`, must be equal to one of the following values:

- `SE3_CERT_OP_STORE`: store the certificate provided as second parameter inside the flash memory of the **SE*cube*™** ;
- `SE3_CERT_OP_DELETE`: delete the certificate provided as second parameter from the flash memory of the **SE*cube*™** ; in this case the second parameter can be initialized specifying only the ID of the certificate (because the ID is the value that is used to distinguish a certificate from another one inside the **SE*cube*™**).

```
void Find_certificate(uint32_t certId, bool& found)
```

Search for a certificate inside the **SE*cube*™** given the certificate ID specified as first parameter. If found, the second parameter is set to `true`, else to `false`.

```
void Get_certificate(const uint32_t cert_id, std::string &cert)
```

Retrieve the certificate with the ID specified as first parameter from the internal memory of the **SE*cube*™** (if such a certificate exists). The certificate is exported as a string (second parameter) and can be immediately saved as a '.pem' file.

```
void List_certificates(std::vector<uint32_t>& certList)
```

Retrieve the IDs of the certificates stored inside the **SE*cube*™** , filling the vector passed as parameter.

7 FAQ

- **Why do I get an error trying to encrypt or decrypt data?** Check if the key you are trying to use actually exists inside the **SE*cube*™** device flash memory, check if the data size is compliant with the limitations mentioned in this document, check if you are providing the correct parameters to the RSA API.
- **Should I inject or generate RSA keys?** In general, it is better to generate RSA keys, because they are created inside the **SE*cube*™** so the private key is never exposed outside of the device.
- **Why is the key generation procedure so slow?** Generating an RSA key requires some time, especially when the hardware resources are limited. The time that is required depends on the length of the key (i.e., 2048 bit keys might require up to 1000 seconds to be generated by the microcontroller of the **SE*cube*™**).