EmSPARK™ Suite

CoreLockr™ Libraries User Guide

Date August 31, 2022 | Version 3.4

TABLE OF CONTENTS

FIGURES

1. CoreLockr™ Libraries

This document is an overview of the EmSPARK™ Security Suite contents, CoreLockr™ APIs' capabilities and included example applications. The EmSPARK™ Suite enables and makes it easy for customers to securely store and isolate keys and certificates. TrustZone® with Security Suite separates domains for security purposes and provides chaining to the provisioned root of trust. The EmSPARK™ Suite provides the following advantages:

1. IP protection: unique keys are provisioned or generated on the device ensuring secrets used to protect the firmware are never exposed.
2. HW/SW security isolation: two operating systems are enabled – TEE (CoreTEE™) and Linux (Rich OS) – and HW security features are segregated in the secure domain (CoreTEE™).

The CoreLockr™ APIs are C libraries for development of Client Applications that execute in the Rich OS. The APIs rely on Trusted Applications (TAs) to execute operations in the TrustZone. EmSPARK™ incorporates a cryptographic engine running in CoreTEE™ that can be used in Linux through its APIs or via OpenSSL. In addition, the Suite supplies other C APIs for performing security-specific functions that are common in IoT applications. The Suite, through HW/SW isolation, allows customers to deploy for end products:

- IP protection
- Secure communication
- Secure payload verification
- Secure storage
- Keys / Certificates provisioning and storage
- Unique device certificate (unique identity) creation

Devices provisioned with the EmSPARK™ Security Suite are also EmPOWER™ enabled. EmPOWER™ is a SaaS solution that provides essential cloud services needed to secure, provision, update and manage devices. During provisioning, keys and certificates that support EmPOWER™ are installed on the device and as such are part of the suite contents explained in the next section. This document does not explain EmPOWER™, for information please contact Sequitur Labs.

1.1. Acronyms and Terminology

Certificate Store	Non-volatile storage of certificates in encrypted form. Managed using the Secure Certificates API.
Client Application	An application that runs in the Rich OS and uses the CoreLockr APIs to access facilities provided by TAs running in the TEE.
CoreTEE™	Sequitur's Trusted Execution Environment (TEE), or secure OS, enabled by ARM's TrustZone™ architecture.
EmPOWER™	SaaS solution that provides cloud services to secure, provision, update and manage intelligent edge devices.
Key Store	Non-volatile storage of keys in encrypted form. Keys within a key store are addressed by name.
Manifest (SLIP)	Encrypted component containing customer personalization data such as keys and certificates. They are installed on device along with the device firmware.
OOID	Opaque Object Identifier, which contains the OOInfo structure encrypted specifically for a target device.

OOInfo	Opaque Object Information structure.
Rich OS	Rich execution environment such as Linux, which runs outside the TEE. The Rich OS is considered un-trusted as compared to the TEE.
STP	Sequitur Trusted Package. A package in a custom DER-encoded format with contents authenticated using a provisioned key.
TA	Trusted Application. A TA runs inside the TEE and provides security related functionality to Client Applications running in the Rich OS or to other TAs running in the TEE.
TEE	Trusted Execution Environment or secure OS, enabled by ARM's TrustZone™ architecture.

1.2. EmSPARK™ Suite Contents

The Suite includes assets for the Rich OS and the Trusted Execution Environment, TEE:

- APIs and Rich OS Assets
  - CoreLockr APIs (C libraries)
  - OpenSSL Crypto Engine
  - Applications and code examples
  - Linux patches (TEE driver and service provider daemon) to enable CoreTEE functionality
  - Toolchain and Client API
- CoreTEE, Secure OS required to access TrustZone secured resources
- Trusted Applications (TAs) in the TEE coupled with the CoreLockr APIs in the Rich OS
  - Crypto Engine TA
  - Secure Certificates TA
  - Secure Storage TA
  - TLS IO TA

The Suite components are illustrated in **Figure 1 EmSPARK Architecture**. The figure depicts a logical view of the two worlds on the device, with CoreTEE in the TrustZone and the non-secure Rich OS. Applications using the CoreLockr APIs in the Rich OS request to execute security functions in the TrustZone. CoreTEE receives and processes the requests. The security functions are executed in the TrustZone and the result passed back to the applications in the Rich OS.

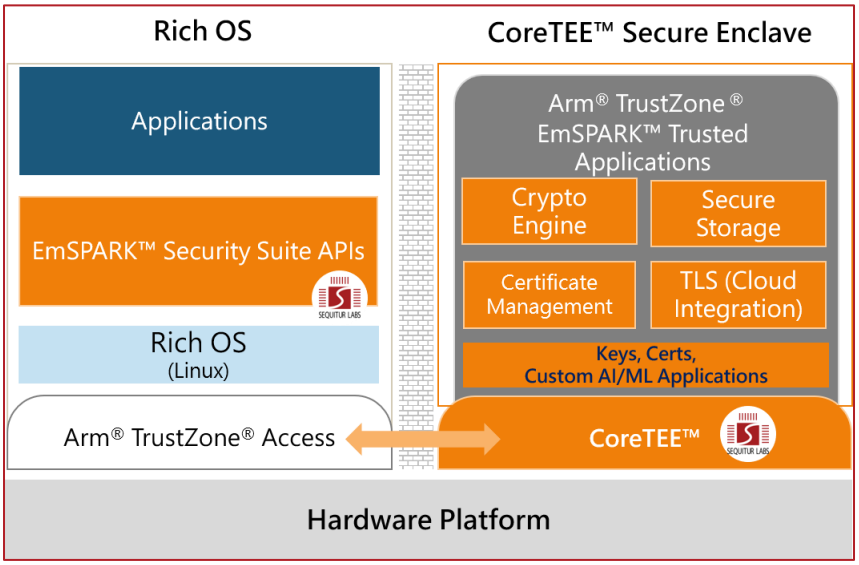


Figure 1 EmSPARK Architecture

1.3. CoreLockr™ APIs

The suite includes the following libraries:

- **CoreLockr Crypto API**, cryptographic functions and key management in the TEE
- **CoreLockr Crypto OpenSSL Engine API**, TEE based crypto engine via OpenSSL
- **CoreLockr Payload Verification and Key Utilities API**, verification of data from a trusted source and general key utility functions
- **CoreLockr TLS IO API**, interface of a TLS client running in the TEE with access to keys and certificates in the TEE
- **CoreLockr Secure Certificates API**, management of trusted authorities in the TEE, rotation of selected provisioned keys and certificates
- **CoreLockr Secure Storage API**, protection of data at rest

1.4. Preinstalled Keys and Certificates in the TEE

During device provisioning, keys and certificates configured in the firmware image are stored in the device non-volatile memory, in manifests. In a production environment, the OEM would configure the keys and certificates. The Evaluation Kit firmware image has preconfigured such keys and certificates. The associated private keys are supplied with the Kit to execute the examples.

Installing on devices the Evaluation Kit firmware provisions keys and certificates that support EmSPARK and EmPOWER™ services. For EmSPARK, the keys and certificates illustrate how a customer such as an OEM can access and manage them through applications developed using the CoreLockr APIs. Customer designs the usage scenarios. For EmPOWER, the keys and certificates illustrate how they are used in cloud services. **Table 1** and **Table 2** list the provisioned keys and certificates stored in non-volatile memory and managed in the TEE.

Table 1 – Provisioned Keys and Certificates for OEM Usage Scenarios

Cert/Key	Name of cert/key exposed by Security suite	Description
OEM Root Cert	CLRSC_OEM_ROOT_CERT	Cert containing the OEM Public Key, customer decides usage. In the Evaluation Kit, it is predefined and the associated OEM Root private key is provided as a file for execution of example applications that verify the OEM signature.
OEM Public Key	CLRC_OEM_PUBLIC_KEY	Public Key extracted from the OEM cert. Used for OEM signature verification, customer decides usage scenarios including authentication and integrity checking of Opaque Keys and Opaque Objects.
OEM Device Private Key	CLRC_OEM_DEVICE_PRIVATE_KEY	Device private key created in device TEE during provisioning. Unique per device. Immutable. Customer decides usage scenarios including decryption of payloads encrypted for the specific device and for TLS connection establishment.

OEM Device Public Key	CLRC_OEM_DEVICE_PUBLIC_KEY	The pair of the OEM Device Private key. Customer decides usage scenarios including generation of encrypted payloads such as Opaque Keys and Opaque Objects tailored to the specific device.
OEM Device Cert	CLRSC_OEM_DEVICE_CERT	Cert that can be used for TLS mutual authentication. Customer decides additional usage scenarios.
OEM Cloud IoT Root CA	CLRSC_OEM_CLOUD_CERT	Root CA of the Cloud IoT. Customer decides usage scenarios such as TLS mutual authentication.
OEM Cloud IoT Public Key	CLRC_OEM_CLOUD_PUBLIC_KEY	Public key extracted from Cloud IoT Root CA.
OEM Payload Cert	CLRSC_OEM_PAYLOAD_CERT	Cert containing the OEM Payload Public Key.
OEM Payload Public Key	CLRC_OEM_PAYLOAD_PUBLIC_KEY	Key used to authenticate update payloads such as firmware update payloads.
OEM Command Cert	CLRSC_OEM_COMMAND_CERT	Cert containing the OEM Command Public Key.
OEM Command Public Key	CLRC_OEM_COMMAND_PUBLIC_KEY	Key used to authenticate commands that change trust on the device, such as commands modifying the Certificate Store or updating the provisioned certificates.

Table 2 – Provisioned Keys and Certificates for Use with EmPOWER Sevices

Cert/Key	Name of cert/key exposed by Security suite	Description
EmPOWER Root Cert	CLRSC_EMPOWER_ROOT_CERT	Certificate used for EmPOWER cloud connectivity.
EmPOWER Public Key	CLRC_EMPOWER_PUBLIC_KEY	Public key contained in EmPOWER Root Certificate.
EmPOWER Device Private Key	CLRC_EMPOWER_DEVICE_PRIVATE_KEY	Device private key created in device TEE during provisioning. Unique per device. Immutable. Used to identify the device with EmPOWER cloud services.
EmPOWER Device Public Key	CLRC_EMPOWER_DEVICE_PUBLIC_KEY	The pair of the EmPOWER Device Private key.
EmPOWER Device Cert	CLRSC_EMPOWER_DEVICE_CERT	Cert used for TLS authentication with EmPOWER service.
EmPOWER Cloud Cert	CLRSC_EMPOWER_CLOUD_CERT	Used for TLS mutual authentication with EmPOWER cloud services
EmPOWER Cloud Public Key	CLRC_EMPOWER_CLOUD_PUBLIC_KEY	Public key extracted from EmPOWER Cloud Cert.

Applications in the Rich OS using the CoreLockr APIs access these keys and certificates by names defined in header files of the Crypto API and Secure Certificates API. Please see **2.3 Access to Provisioned Keys** and **8.1 Provisioned Certificates**.

In the Evaluation Kit, with exception of the OEM Device Key, OEM Device Certificate, EmPOWER Device Key and EmPOWER Device Certificate which are unique per device, all other keys are the same in all kits and are not meant to protect secrets but to be used with the examples or for testing.

2. CoreLockr™ Crypto API

The CoreLockr Cryptographic API and Crypto Engine TA allow easy access to cryptographic functions in the TEE and provides mechanisms to protect confidential information on a device. This section is an overview of the API functionality and example applications. For description of the API, please see `CoreLockr_Cryptographic_API.pdf`.

The API functionality includes:

- Key management, ephemeral keys and persistent keys managed in the TEE
- Key Store, device specific encrypted key storage
- Access to provisioned keys, access of provisioned credentials protected in the TEE
- Cryptographic operations, cryptographic operations executed in the TEE
- Tools to generate and use Opaque Keys, mechanism to transport keys to a device while protecting their confidentiality and integrity and store them in the device Key Store
- Tools to generate and use Opaque Objects, off-device encrypted objects for decryption on-device only when the decryption is enabled with device specific keys

In the Kit, `corelockr/corelockr_crypto` contains:

- `lib`, `libseqr_corelockr_crypto.so` library
- `include`, header files
- `ta`, `138A1951-2A00-BF5A-A463E61F402EBE1D.stp` associated TA
- Documentation, `CoreLockr_Cryptographic_API.pdf` describes the API
- `README.txt`, general API information
- `COPYRIGHT`, copyright notice
- Example applications

Section **2.8 Examples** describes the example applications. See **Appendix A: Supported Cryptographic Operations** for list of supported operations.

In the Kit, `corelockr/corelockr_opaque_keys` contains a script used for creating Opaque Key packages. Please see the **3 CoreLockr™ Crypto API – Opaque** Keys section.

In the Kit, `corelockr/corelockr_opaque_objects` contains scripts used for creating Opaque Objects. Please see the **4 CoreLockr™ Crypto API – Opaque** Objects section.

2.1. Key Management

Supported key types include AES, RSA, ECDSA, ECDH, DH, DSA and HMAC. The API supports ephemeral keys and persistent keys managed throughout the device life cycle. Ephemeral keys exist in memory within the loaded TA instance. An ephemeral key disappears when the TA is closed. Persistent keys are stored in the Key Store, a non-volatile storage of keys in encrypted form. Applications in the Rich OS reference keys in the TEE via key handles.

Capabilities of the CoreLockr Crypto engine to import and use keys in the TEE, and execution of operations with provisioned keys are illustrated with an example application and code sample.

2.2. Key Store

The Key Store is a non-volatile storage of keys in encrypted form. Each persistent key is stored with a name used to access and manage the key. Keys can be created in the TEE or imported to the TEE. Keys can be stored in the Key Store using Opaque Key mechanisms. The visibility the Rich OS has of the key private attributes is set at the time of key creation or importing.

Keys within the Key Store may be password protected. The same as keys, password objects are stored in the Key Store and have names. Loading a password object from the Key Store generates a handle.

2.3. Access to Provisioned Keys

The API allows access to provisioned keys such as the keys listed in **Table 1** and **Table 2**. Provisioned keys are accessed as *named* keys, the same as keys in the Key Store with no password. The key names are defined in the `corelockr_crypto.h` header file.

```
~/corelockr_crypto/include/corelockr_crypto.h
#define CLRC_OEM_PUBLIC_KEY              "com.seqlabs.oem_pub_key"
#define CLRC_OEM_CLOUD_PUBLIC_KEY        "com.seqlabs.oem_cloud_pub_key"
#define CLRC_OEM_PAYLOAD_PUBLIC_KEY      "com.seqlabs.oem_payload_pub_key"
#define CLRC_OEM_COMMAND_PUBLIC_KEY      "com.seqlabs.oem_command_pub_key"
#define CLRC_OEM_DEVICE_PUBLIC_KEY       "com.seqlabs.oem_device_pub_key"
#define CLRC_OEM_DEVICE_PRIVATE_KEY      "com.seqlabs.oem_device_key"
#define CLRC_EMPOWER_PUBLIC_KEY          "com.seqlabs.emp_pub_key"
#define CLRC_EMPOWER_CLOUD_PUBLIC_KEY    "com.seqlabs.emp_cloud_pub_key"
#define CLRC_EMPOWER_DEVICE_PUBLIC_KEY   "com.seqlabs.emp_device_pub_key"
#define CLRC_EMPOWER_DEVICE_PRIVATE_KEY  "com.seqlabs.emp_device_key"
```

The API function to load a provisioned key returns a key handle. In the case of the OEM Device Private Key, `com.seqlabs.oem_device_key`, and EmPOWER Device Private Key, `com.seqlabs.emp_device_pub_key`, which are generated during provisioning, applications running in the Rich OS have no visibility of the keys' private attributes. To use these private keys, applications reference them via handles.

2.4. Cryptographic operations

Cryptographic operations are executed in the TEE. Applications in the Rich OS reference operations via handles. For complete information about the supported API functions, please see `CoreLockr_Cryptographic_API.pdf`. Operations include:

- Symmetric encrypt/decrypt
- Asymmetric encrypt/decrypt
- Generate message authentication codes
- Sign and verify signatures
- Derive shared keys
- Generate cryptographic hashes
- Random number generation

2.5. Opaque Keys

This is a mechanism to transport keys in signed and encrypted opaque key packages. Device specific keys are used to encrypt opaque key packages. The Suite provides tools to produce Opaque Key packages containing keys generated in environments outside the device, and tools to import and store the keys in the TEE while preventing the Rich OS from accessing the contents of such keys. Section **3 CoreLockr™ Crypto API – Opaque** Keys explains concepts, creation and usage.

2.6. Opaque Objects

Opaque Objects protect applications and IP at rest. Opaque Objects facilitate transferring and storing encrypted payloads and enable their access for a specific device. There are two sides to the code: the server side where the Opaque Object is encrypted and its device-specific identifier is created, and the device side where the object is decrypted. Section **4 CoreLockr™ Crypto API – Opaque Objects** explains concepts, creation and usage.

2.7. Opaque Keys and Opaque Objects Usage

From the usage schema, the following are differences between Opaque Keys and Opaque Objects:

- Opaque Keys are saved to the persistent Key Store and can be managed and used as any other key in the Key Store.
- Opaque Objects are encrypted and can be copied to any device. However, to decrypt them, an Opaque Object Identifier generated for a specific device is required.
- Opaque Object Identifiers produce transient keys that are only used to decrypt the Opaque Object.

2.8. Examples

This section describes the key management and Key Store examples. Please see **3 CoreLockr™ Crypto API – Opaque Keys** and **4 CoreLockr™ Crypto API – Opaque Objects** for additional examples.

2.8.1. Key Management and Provisioned Key Access Example

The application uses the CoreLockr Crypto API to execute the following functions:

- Compute a random number
- Import and use keys in the TEE
- Compute an HMAC
- Encrypt and decrypt data using the AES-128-CBC algorithm
- Sign and verify data using ECC keys
- Execute operations with a key preinstalled in the TEE

Software and Data Requirements

- CoreLockr Crypto example application, `corelockr/corelockr_crypto/example`
- Supplied EC private key file, `seq_oem_ca_key.der`
- Provisioned EC public key available in the TEE, `com.seqlabs.oem_pub_key`

The private key associated with the OEM Root Certificate is required for signing payloads that will be verified in the TEE with the OEM Public Key. The example provides `seq_oem_ca_key.der` associated with the OEM Root Certificate `com.seqlabs.oem_pub_key` preinstalled in the TEE. This private key file usually not found on the device is provided to make easy the example execution.

Building and Installing

In Linux development environment, change to the `corelockr/corelockr_crypto/example` directory and execute `make`. The compilation creates the `clrc_demo` executable. Transfer `clrc_demo` and `seq_oem_ca_key.der` to the board to a directory of your preference.

Flow and Code Walkthrough

See `corelockr/corelockr_crypto/example/README.txt`.

Executing the Example

Application options:
`-k [SSL ECC private key file]` (required)
`-p [Preloaded ECC public key name]` (optional). If omitted, the private key file is used for verification.

Change to the directory where the `clrc_demo` application and `seq_oem_ca_key.der` were transferred.

To sign with the ECC private key file and verify with the ECC public key saved in the TEE, execute:

```
./clrc_demo -k seq_oem_ca_key.der -p com.seqlabs.oem_pub_key
```

where `com.seqlabs.oem_pub_key` is the name of the OEM Public Key saved in the TEE, as explained in **2.3 Access to Provisioned Keys**.

2.8.2. Key Store Example

This example illustrates functionality of the Key Store. Using the Crypto API, the application creates a key in the TEE and saves it in the Key Store as a named key and with a password. Then, the application loads the key from the Key Store and uses it for crypto operations. Application functionality:

- Create an ECDSA key: ECC P256
- Use the created key to sign an input string
- Save the key in the Key Store
  - Set the password object in the Key Store
  - Save the named key in the Key Store with the associated password object
- Load the named key from the Key Store providing a password
- Use the named key to verify the signature
- Clean up
  - Delete the named key
  - Delete the password object from the store

Note that key names are unique in the Key Store. The example deletes the named key and password object to avoid `CLRC_ERROR_EXISTS` in subsequent application executions.

Software and Data Requirements

CoreLockr Crypto example application,
`corelockr/corelockr_crypto/example_key_store_with_password`

Building and Installing

In Linux development environment, change to `corelockr/corelockr_crypto/example_key_store_with_password` and execute `make`. The compilation creates the `clrc_key_store_password` executable. Transfer the executable file to the board in a directory of your choosing.

Flow and Code Walkthrough

See `corelockr/corelockr_crypto/example_key_store_with_password/README.txt`.

Executing the Example

Change to the directory where the `clrc_key_store_password` application was transferred and execute it:

```
./clrc_key_store_password
```

The application prints messages for different functions and computation results, including:

```
Creating ECDSA key
Printing out ECDSA key attributes
Signing with the created key
Creating password object "ecc-256-pw" in the store
Saving key in the key store
   Saved named key "ecc-256-key" in the key store,
   with associated password object "ecc-256-pw"
ECDSA verifying using key loaded from the key store
Deleting "ecc-256-key" from the key store
Deleting password object "ecc-256-pw" from the store
```

3. CoreLockr™ Crypto API – Opaque Keys

When keys generated outside the device need to be transferred to the device protecting their confidentiality and integrity, Opaque Keys are the solution. On a device-specific basis, Opaque Keys are made available to the TEE without allowing the Rich OS to see the key contents. Opaque Keys facilitate scenarios such as sending license keys to the device for feature enablement, symmetric keys used for confidential data transfer between devices, and asymmetric key pairs for associated certificates.

The EmSPARK suite provides tools to generate packages containing such keys in environments outside the device, and tools to import and store the keys in the TEE while preventing the Rich OS from accessing the contents of the Opaque Keys.

Kit contents

- The CoreLockr Crypto API documentation lists the supported Opaque Key types and algorithms and describes the `clrcSaveOpaqueKeyEx()` function that verifies the signature of an encrypted key package, decrypts and saves the packaged key into the Key Store,
`CoreLockr_Cryptographic_API.pdf`

In addition, the Suite includes in `corelockr/corelockr_opaque_keys`:

- A reference implementation script to create Opaque Key packages, code documentation and usage help, `make_opaque_key_package.sh`.
- Documentation describing the Opaque Key package structure and approach for encryption and MAC operations for those who wish to create packages with their own software, see `opaque_key_package_format.txt`.
- Instructions and sample key to execute example.

3.1. Creating and Storing Opaque Keys

Opaque Key packages are created in environments outside the device, such as servers. When transferred to the device, the Opaque Key package can be verified, decrypted and unpacked directly to the CoreLockr Key Store using the CoreLockr Crypto API. In this section, **Figure 2** depicts a high-level flow of the Opaque Key creation on a server and storage on the device. The next subsections describe how to create an Opaque Package and how to store the key contained in the Opaque Package into the Key Store on the device.

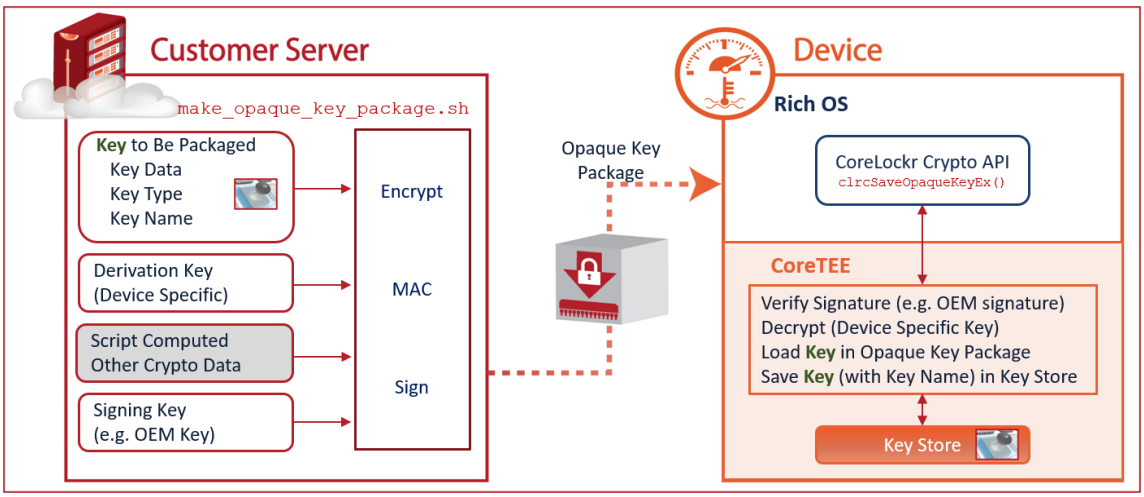


Figure 2 Opaque Keys

3.1.1. Creating Opaque Key Packages

The `corelockr/corelockr_opaque_keys/make_opaque_key_package.sh` script is used to create Opaque Key packages, as follows:

- The script packages a key into an opaque bundle. The key is either an asymmetric key in DER format, or a symmetric key in raw binary format.
- The key is combined with additional key information (key type, key name in the persistent storage and key data) into a DER-encoded SEQUENCE
- The DER-encoded SEQUENCE is encrypted and has its MAC tag computed (in accordance with the ECIES standard).
- The keys for the encryption and MAC are computed from a device specific ECC public key, an ephemeral key pair, and a random number.
- The encrypted payload, MAC tag, public component of the ephemeral key pair, random number, and algorithm IDs are all DER-encoded and concatenated.
- That data is signed, and the signature is prepended to the data within a wrapping SEQUENCE.

The script may be used on a server where the required utility programs are available. The script takes the following inputs:

a. Packaged key, the key to be transferred to the device
- Name that will be used to store the packaged key in the device Key Store
- Key type
- Path of the key file

b. OEM private key for signing the Opaque Key package, the signature will be verified on the device using the provisioned OEM Public Key
- Path of the key file

c. ECDH shared secret derivation key which will be used for decryption on the device, it shall be device specific such as the OEM Device Public Key file or another ECC public key file already stored in the device Key Store
- Path of the key file

d. Name of the cipher algorithm

e. Name of the MAC algorithm

f. Path to the output Opaque Key package file

The script has complete information of input values.

Saving Opaque Key on Device Key Store

On the device, the CoreLockr Crypto API `clrcSaveOpaqueKeyEx()` function verifies the signature of an encrypted key package, decrypts and loads the key contained in the package, and saves the key into the Key Store under the key name provided when creating the Opaque Key package.

```
ClrcResult clrcSaveOpaqueKeyEx(const uint8_t *keyPkg,
     uint32_t keyPkgLength,
     ClrcKeyHandle hDeviceKey,
     ClrcPasswordHandle  hPassword);
```

The key package shall be created using `make_opaque_key_package.sh`, or following the same process as described in the shell script. For information about `clrcSaveOpaqueKeyEx()`, please see `CoreLockr_Cryptographic_API.pdf`. After the key is stored in the Key Store, applications can access the key in the same manner as any other key managed in the TEE.

3.2. Opaque Key Example

The example illustrates the sequence from creating an Opaque Key Package using `make_opaque_key_package.sh` to storing the Opaque Key in the Key Store as a named key with a password using the Key Utilities example. It also illustrates the use of the Device Public Key extracted from the TEE as a file and used as the ECDH shared secret derivation key during the Opaque Key package creation.

The following sequence includes operations executed off-device and on the device:

- Create Opaque Key package using `make_opaque_key_package.sh`
- Store Opaque Key in the TEE as a Named Key with a password using `clrpv_key_utility`
- Print attributes of named key

The example uses the key utilities example provided with the CoreLockr Payload Verification API. For building instructions see **5.2 Key Utilities**.

a. Create Opaque Key package
Opaque Key packages are DER-encoded structures containing key information (key type, key name in the persistent store, and key data). The key information is encrypted inside the package, so no key information is accessible until decrypted. The MAC tag for the encrypted information is also added to the package. The encrypted key information, MAC tag, and the remaining information required for decryption are hashed and signed to ensure the integrity and verify the source of the package.

- On the device, obtain the device specific key, e.g. Device Public Key. Using `clrpv_key_utility` extract Device Public Key `com.seqlabs.oem_device_pub_key` to a file

```
./clrpv_key_utility -E -n "com.seqlabs.oem_device_pub_key" -k
device_pub_key.der
```

- Transfer `device_pub_key.der` to the development environment to create the Opaque Key package, `~/corelockr/corelockr_opaque_keys/`
- The Opaque Key package creation uses the OEM key to sign the package. The sample OEM key is provided with Crypto API example, located at `corelockr/corelockr_crypto/example/ seq_oem_ca_key.der`.
- The provided `ecdsa256.der` is a sample private key file to be transferred to the device

```
openssl ec -in ecdsa256.der -text -noout -inform DER
```

- Off-device, e.g. in the development environment, create the Opaque Key package from the example key `ecdsa256.der`

```
./make_opaque_key_package.sh -n "opaque.ecdsa256.test1" \
-t KEY_ECDSA_KEYPAIR -K ecdsa256.der \
-S ../corelockr_crypto/example/seq_oem_ca_key.der \
-D device_pub_key.der -c AES_CTR -m HMAC_SHA256 \
-o opaque_key.dat
```

The `-o` output Opaque Key package is `opaque_key.dat`. The other switches represent:
`-n` name that the packaged key will have in the Key Store `opaque.ecdsa256.test`
`-t` type of the key to be packaged as defined in `make_opaque_key_package.sh`
`-K` path to the key file to be packaged
`-S` path of the key file used for signing, the OEM key
`-D` path to the device-specific ECC public key file, in this case the Device Public Key

b. Transfer the Opaque Key package to the device
- Copy `opaque_key.dat` to the directory where `clrpv_key_utility` is located.

c. On the device, store the Opaque Key in the Key Store as a named key and with a password, using `clrpv_key_utility`
- Create a password object

```
./clrpv_key_utility -S -p "OKMyPW:myokpassword"
```

- Store the Opaque Key in `opaque_key.dat` with the password (the name of the key is encoded in the file). The application uses `clrcSaveOpaqueKeyEx()` to store the Opaque Key into the Key Store.

  ```
  ./clrpv_key_utility -O -k "opaque_key.dat" -p "OKMyPW:myokpassword"
  ```

- Print the public attributes of the stored key and confirm they are the same as in `ecdsa256.der`

  ```
  ./clrpv_key_utility -P -n "opaque.ecdsa256.test1" -p
  "OKMyPW:myokpassword"
  ```

- If desired to execute the example again, delete the password object and the stored named key, as shown in Key Utilities Example.

4. CoreLockr™ Crypto API – Opaque Objects

Opaque Objects facilitate transferring and storing encrypted payloads that can only be decrypted on a device-specific basis. Opaque Objects are the solution when objects generated outside the device need to be transferred and executed on the device protecting their confidentiality and integrity. Opaque Objects verification and decryption is only possible in the TEE without allowing the Rich OS access to the required keys.

Kit Contents

The CoreLockr Crypto API documentation describes the Opaque Object Decoding functions and the supported algorithms, `CoreLockr_Cryptographic_API.pdf`.

In addition, the Suite includes in `corelockr/corelockr_opaque_objects`:

- A reference implementation script to create an Opaque Object, `make_opaque_object.sh`.
- A reference implementation script to generate the Opaque Object Identifier, `make_opaque_object_identifier.sh`.
- Documentation of the Opaque Object Identifier package format for those who wish to create Identifiers with their own software, `opaque_object_identifier_package_format.txt`.
- Documentation of the cryptographic operations and keys, `README.txt`.
- Example application.

4.1. Creating and Decrypting Opaque Objects

Opaque Objects consist of two components:

- **Opaque Object**, the object itself is a data bundle encrypted using a standard AES-256 algorithm. It is created with an associated Opaque Object Information structure (**OOInfo** structure).
- **Opaque Object Identifier (OOID)**, which contains the OOInfo structure encrypted specifically for a target device. The OOID is required to access the decrypted contents of the Opaque Object.

Because the Opaque Object is encrypted, it can be created on a server and installed on any number of devices. The associated OOInfo structure contains the key for decrypting the object, the type of cipher, the digest and size of the cleartext data bundle, and various optional usage policies.

The OOInfo structure is itself encrypted into a separate package called the Opaque Object Identifier (OOID). To give a particular device access to the decrypted contents of the Opaque Object, the OOID

is created on the server, encrypted for a specific device and then transferred to the device. The encryption of the OOInfo inside the OOID uses a separate key from that used to encrypt the Opaque Object. On the device, the OOID is used with the Opaque Object to obtain the decrypted data from the latter.

Encrypted specifically for a target device, the OOID cannot be decrypted on another device. The key to decrypt the OOID is specified during its creation. The OOID is encrypted using Integrated Encryption and a public key from the device. When decrypted, the OOID is used to decrypt the Opaque Object.

Figure 3 depicts a high-level flow of the Opaque Object creation on a server and decryption on the device:

- On a server, the Opaque Object and the Opaque Object Information Structure (OOInfo) are generated. The Opaque Object may be transferred to a device.
- On the server, the Opaque Object Identifier is generated taking as inputs the Opaque Object Information Structure, the signing key and the device specific key.
  - The signing key can be the OEM private key whose signature is verified on the device using the provisioned OEM Public Key. Alternatively, the signing key can be another key whose public component is available in the device Key Store to verify the signature. Such key may be ECDSA, RSA or DSA.
  - The device specific key can be the OEM Device Public Key file or another ECC public key file already stored in the device Key Store.
- On the target device where the device specific key is in the TEE, the Crypto API function will require the Opaque Object and the Opaque Object Identifier in order to decrypt the object. For detail on the cryptographic operations to create Opaque Objects and OOID, keys, OOInfo structure and OOID format, please see `opaque_object_identifier_package_format.txt` and `README.txt`.

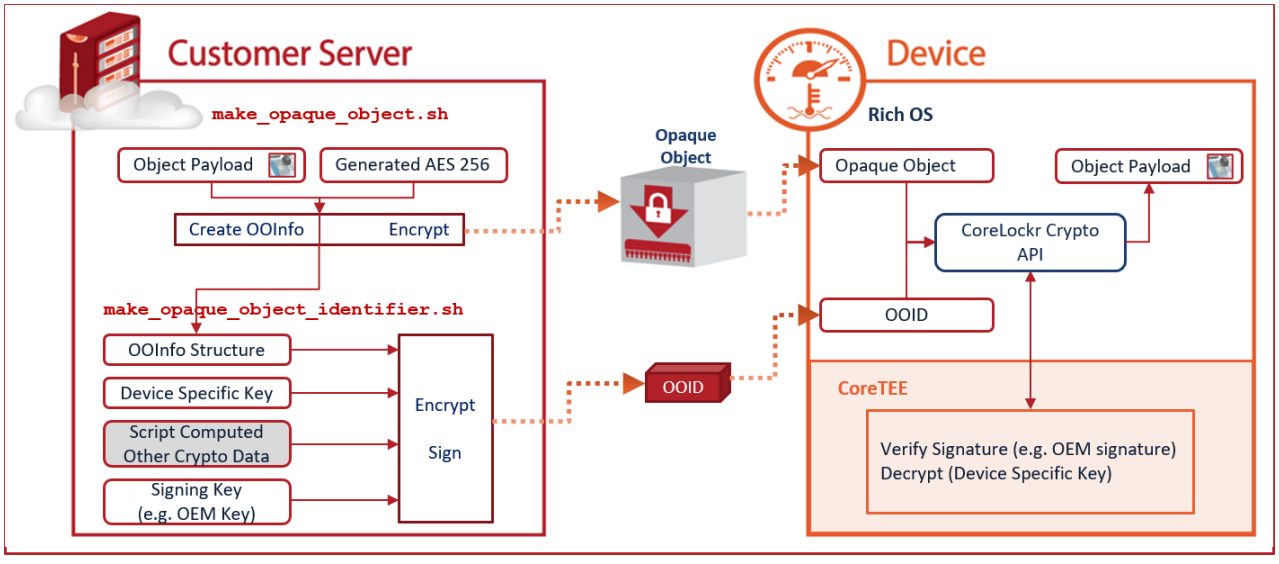


Figure 3 Opaque Objects

4.2. Opaque Object Example

The example creates an Opaque Object of a file (README.txt) and then the associated OOID. The client application on the device takes these two inputs. The client application simply decrypts the file contained in the Opaque Object. To illustrate the functionality, the example uses the following scripts and client application described also in the `README.txt` file provided with the example:

- Create an Opaque Object, the example uses the `make_opaque_object.sh` script
- Create a device-specific identifier for the Opaque Object, `make_opaque_object_identifier.sh` script
- Then the example uses the client application to decrypt the Opaque Object on the device.

This section provides requirements, building instructions and additional references. The example requires a sequence of steps starting in Linux development environment, then on the device, again in Linux environment and finally on the device.

Software and Data Requirements

- CoreLockr Opaque Objects example application, `README.txt` and scripts, `corelockr/corelockr_opaque_objects`
- Payload Verification example application, `corelockr/corelockr_payload_verification/example`
- Sample OEM private key, `corelockr/corelockr_crypto/example/seq_oem_ca_key.der`
- Provisioned Device Public Key, extracted from the TEE on the device

Building

In addition to create the Opaque Objects, this example requires the Payload Verification Key Utilities example application, described below. In Linux development environment:

a. Build the Payload Verification Key Utilities example application in this directory according to the instructions in **5.2 Key Utilities** and transfer the `clrpv_key_utility` executable to the board

```
corelockr/corelockr_payload_verification/example_key_utility/
```

b. The example execution will instruct how to build the application and to create the Opaque Object and Opaque Object Identifier

```
corelockr/corelockr_opaque_objects/make_opaque_object_identifier.sh
```

4.2.1. Executing the Example

Execute the following steps on the **device** in Linux to extract from the TEE the device public key:

a. Change to the directory where the `clrpv_key_utility` application was transferred.

b. Execute the following command to extract the OEM Device Public Key:

```
./clrpv_key_utility -E -n "com.seqlabs.oem_device_pub_key" -k dev-pub.der
```

Transfer the extracted key to the Linux development environment and place it in this example directory

```
~/corelockr/clrc_opaque_objects/example/
```

Execute the following steps in **Linux development environment** within the example directory to build the example application, create the Opaque Object and then generate the OOID.

c. Change to:

```
~/corelockr/clrc_opaque_objects/example/
```

d. Run "`make`" to build the `clrc_opaque_objects_demo` application.

e. Execute the script to create the Opaque Object from the `example/README.txt` file:

```
../make_opaque_object.sh -i README.txt
```

A random `AES-256` key will be generated, and the default cipher algorithm `AES_CTR` will be used for encrypting the Opaque Object. Two outputs are generated:

- The Opaque Object, `README.txt.enc`
- A configuration file with information necessary for creating the identifier, `README.txt.oocfg`

Run `../make_opaque_object.sh -h` to view more information about the script.

f. In this example, the creation of the Opaque Object identifier uses the OEM Device Public Key to encrypt the AES key generated in step **e.**, and the OEM signing key to sign the identifier. The device public key is acquired as in step **b.** above. The OEM signing key for the evaluation package is located at:

```
corelockr/corelockr_crypto/example/seq_oem_ca_key.der.
```

Execute the following command to create the Opaque Object identifier (OOID):

```
../make_opaque_object_identifier.sh -i README.txt.oocfg \
-S ../../corelockr_crypto/example/seq_oem_ca_key.der \
-D dev-pub.der
```

The identifier will be written to the file `README.txt.ooid`. The default cipher `AES_CTR` and digest `HMAC_SHA256` algorithms will be used for that.

Executing the script with the `-h` option shows help information, the available options and the full lists of cipher and HMAC algorithms which can be used to create the identifiers.

g. Transfer the `clrc_opaque_objects_demo` application and `README.txt.enc` and `README.txt.ooid` files to the device. Transfer also the original `README.txt`, which in this example is compared the decrypted file to illustrate they are identical.

Execute the following steps on the **board** in Linux.

h. Change to the directory where the `clrc_opaque_objects_demo` application and `README.txt*` files were transferred to.

i. Execute the following command to decrypt the Opaque Object:

```
./clrc_opaque_objects_demo -k README.txt.ooid -i README.txt.enc \
-o README.txt.dec
```

`README.txt.dec` contains the decrypted output of the Opaque Object.

j. Execute the following command to compare the decrypted output with the original file:

```
cmp README.txt README.txt.dec
```

If the files are identical as expected, no output will be generated by the `cmp` command.

5. CoreLockr™ Payload Verification and Key Utilities API

This API has payload verification functions and key utility functions. It is often necessary to know that data accessed on a remote device is from a trusted source and has not been tampered with. This is usually accomplished by signing the data with a private key at the source, and then verifying it with the corresponding public key on the device. The CoreLockr Payload Verification system does exactly that, with the added security of keeping the public key within a persistent store that prohibits tampering.

The key utilities are not strictly necessary for creating and verifying signed payload packages. However, they can be useful for testing and provisioning keys on devices. There are utilities for printing out the contents of loaded keys (useful for debugging) and utilities to save keys loaded in the TEE to a memory buffer or a file, useful for public keys.

The API functionality includes:

- Payload Verification
  - Creation of signed payload packages
  - Verification and decoding of signed payload packages
- General key utility functions
  - Importing and exporting asymmetric key DER encoded files into the CoreLockr Crypto system
  - Functions for converting between DER-encoded and CoreLockr Crypto format signatures and public keys
  - Printing out the contents of loaded keys for debugging

In the Kit, `corelockr/corelockr_payload_verification` contains:

- `lib`, two versions of the library are available: `libclrpv.a` for building applications for the board, and `libclrpv_x86_64.a` for building applications for Linux x86_64 systems (for creating packages)
- `include`, header files
- `docs`, library documentation
- `bin`, a shell script that uses OpenSSL for creating payload packages in case the API library is unavailable in the system
- `payload_package_format.txt`, description of the encoded package format for those who wish to create package files with their own software
- `README.txt`, general API information
- `COPYRIGHT`, copyright notice
- `example`, application with source code
- `example_key_utility`, application with source code

Sections **5.1 Payload Verification Example** and **5.2 Key Utilities Example** describe the example applications.

5.1. Payload Verification Example

The API facilitates creation of signed payload packages and verification and decoding of signed payload packages. Packages created at the source contain the payload data, signature and hash identifier, all encoded together in DER format.

5.1.1. Background

The example application illustrates the ability of the EmSPARK Suite to verify data using a key preinstalled in the TEE. The application uses the CoreLockr Payload Verification API which provides functions for creating and verifying signed payload packages. The API supports ECC, RSA, DSA and DH keys and multiple hashing algorithms. The Example Application Execution has two phases:

- Creation of a signed payload package
  - Creation of payload in a system external to the board. The Kit provides a shell script that uses OpenSSL for creating payload packages, `corelockr_payload_verification/bin/make_payload_package.sh`. The script signs the digest of a payload file and DER encodes it.
  - Creation of payload on the board. To simplify the example set up and avoid the configuration of an external system for creating signed packages, the application that executes on the board can be used for both creating a signed package and then for verifying it. The application creates a signed payload package using the OpenSSL crypto library.
- Verification of a signed payload package
  - Using the underlying CoreLockr Crypto, the example application executing on the board verifies the signed packages

For simplicity, the example code requires the use of ECC keys and forces the use of the SHA1 hash. The application permits the execution of the following scenarios:

- Successful verification of a payload package signed with an authorized key
- Failed verification of a payload package signed with an unauthorized key
- Encoding of a package (optional)

Software and Data Requirements

- Payload Verification example application, `corelockr/corelockr_payload_verification/example`
- OEM Payload Private Key associated with the OEM Payload Certificate flashed on the device, used for signing the payload packages. The example provides `seq_payload_ca.key` in PEM format and `seq_payload_ca_key.der` in DER format (ECC P256) which are associated with OEM Payload Public Key provisioned on the device to verify payloads, see **Table 1 – Provisioned Keys and Certificates for OEM Usage Scenarios**
- OEM Payload Public Key to verify signed packages. This key is already saved in the TEE and can be accessed by name: `com.seqlabs.oem_payload_pub_key`
- Script to create payload packages, `corelockr_payload_verification/bin/make_payload_package.sh`
- Payload files, which need to be available on the board before executing the application

Building and Installing

Change to the `corelockr/corelockr_payload_verification/example` directory. Execute `make` to build `clrpv_demo`. Transfer the executable along with payload sample file to the board.

5.1.2. Executing the Example

On the device, change to the directory where `clrpv_demo` is located to execute the application commands in the Linux command line.

Create a Signed Payload Package in a System External to the Board

To create a signed package in an external system using the `make_payload_package.sh` script in `corelockr_payload_verification/bin`, identify:

- Path of the private key used for signing, in PEM format. In this example, the provided `seq_payload_ca.key` in PEM format is associated with the OEM Payload Certificate
- Path of the payload file, `test-payload` is provided with the example application
- Path and name to the output package file
- Hashing algorithm, md5, sha1, sha224, sha256, sha384, sha512

Execute the script in the system where the payload file will be generated, e.g.

```
./make_payload_package.sh -k seq_payload_ca.key -d sha1 -p test-payload -o
signed-on-server-payload
```

where `-k` precedes the key file name, `-d` the hashing algorithm, `-p` the name of the payload to be signed and `-o` the name of the signed payload, which in the example is `signed-on-server-payload`. Transfer the signed packages to the board.

Create a Signed Package on the Board

To simplify the Example Application Execution, the application on the board can create a signed payload package using the OpenSSL crypto library. The creation of a signed package requires:

- Path of the private key, in PEM or DER format, used for signing
- Path of the payload file
- Path to the output package file
- Hashing algorithm, the example application uses SHA1

To create a package, make sure you have a sample payload file (`test-payload` is provided) to be signed, and execute:

```
./clrpv_demo -E -p test-payload -k seq_payload_ca.key -s encoded-pkg
```

where "`-E`" indicates the application option to encode the payload file name `test-payload` (this can be replaced by another existing payload file name), "`-k`" is the private key `seq_payload_ca_key.der`, and "`-s`" precedes the signed payload package file name to be created, `encoded-pkg`.

If the encoding is successful, the output below is printed and `encoded-pkg` is created:

```
Package creation returned 0 (0x00000000)
```

Verify a Package, Successful Verification Scenarios

The Payload Verification API has functions for decoding a signed package, extracting the payload and writing an output file. For package verification the Payload Verification API requires:

- Signed package path
- DER format key

The example application can verify packages using either a public key preinstalled in the TEE or a key file in DER format. To verify packages using the preloaded key in the TEE execute:

```
./clrpv_demo -V -k com.seqlabs.oem_payload_pub_key -s signed-on-server-payload -p new-payload

./clrpv_demo -V -k com.seqlabs.oem_payload_pub_key -s encoded-pkg -p new-payload2
```

where "`-V`" tells the application to verify the encoded package `signed-on-server-payload` or `encoded-pkg` and "`-p`" to extract the payload in a new payload file `new-payload` or `new-payload2`. "`-k`" precedes `com.seqlabs.oem_payload_pub_key`, the name of the OEM Payload Public Key managed in the TEE.

In this case the verification succeeds because both packages were signed with the private key corresponding to the public DER key saved in the TEE and used for verification (and the packages have not been tampered with). The following output is printed on the console and `new-payload` file is created:

```
Verification returned 0 (0x00000000)
```

The standard command-line tool "`diff`" can be used to test that the extracted `new-payload` and `new-payload2` files are identical to the original `test-payload` file.

Verify a Package, Verification Failure Scenarios

When there is a mismatch between the signed package and the key used for signing, the verification fails and the corresponding error is printed on the console. For example, attempts to verify a package whose signature or contents have been altered return errors. For description of the encoded package format see `payload_package_format.txt`.

Note that error codes can be returned from the Payload Verification API, from the underlying CoreLockr Crypto API calls, as well as the usual `errno` return codes from system calls. See the API documentation for list of return codes.

5.2. Key Utilities Example

Key utilities include functions for converting between DER-encoded and CoreLockr Crypto format keys and signatures.

5.2.1. Background

The example application uses the CoreLockr Payload Verification API and the CoreLockr Crypto API to illustrate a variety of scenarios, including the following for key store and Opaque Keys:

- Store a password object with password
- Store a private key associated with the stored password
- Print the attributes of the stored key
- Extract the components of the stored key
- List the name of the password required by the stored key
- List the names of the keys using the stored password

- Delete the stored key
- Delete the stored password object
- Extract from the TEE the public key of a provisioned key such as the OEM Device Public Key
- Store an Opaque Key in an Opaque Key package with a password

Note: the scenario that stores an Opaque Key to the key store requires that an Opaque Key package be created and transferred to the device. Please see the **3 CoreLockr™ Crypto API – Opaque Keys** section.

Software and Data Requirements

`~/corelockr/corelockr_payload_verification/example_key_utility` example source files and sample key

Building and Installing

Change to the `~/corelockr_payload_verification/example_key_utility` directory. Execute `make` to build `clrpv_key_utility`. Transfer the executable along with payload sample files to the board.

5.2.2. Executing the Example

On the board, change to the directory where `clrpv_key_utility` and sample key are located to execute the application commands in the Linux command line.

The application README.txt and the binary provide information of the switches and input data required for its execution.

Key Utilities Example

Store a password object named “MyPW” with password “mypassword”

```
./clrpv_key_utility -S -p "MyPW:mypassword"
```

Store a DER-encoded ECDSA private key in the file “ecdsa.der” under the name “MyEcdsaKey” and using the stored password

```
./clrpv_key_utility -S -n "MyEcdsaKey" -t KEY_ECDSA_KEYPAIR -k ecdsa.der -p
"MyPW:mypassword"
```

Print the public attributes of the stored key

```
./clrpv_key_utility -P -n "MyEcdsaKey" -p "MyPW:mypassword"
```

Extract the public components of the stored key

```
./clrpv_key_utility -E -n "MyEcdsaKey" -p "MyPW:mypassword" -k ecdsa-pub.der
```

List the name of the password required by the stored key

```
./clrpv_key_utility -L -n "MyEcdsaKey"
```

List the names of the keys using the stored password

```
./clrpv_key_utility -L -p "MyPW:mypassword"
```

Delete the stored key

```
./clrpv_key_utility -D -n "MyEcdsaKey" -p "MyPW:mypassword"
```

Delete the stored password object

```
./clrpv_key_utility -D -p "MyPW:mypassword"
```

Extract from the TEE the public key of a provisioned key such as the OEM Device Public Key

```
./clrpv_key_utility -E -n "com.seqlabs.oem_device_pub_key" -k
device_pub_key.der
```

Opaque Key Example

Store Opaque Key in the TEE as a named key and a password using `clrpv_key_utility`. For information to execute this option, see **Opaque Key** Example in **3 CoreLockr™ Crypto API – Opaque Keys.**

6. CORELOCKR™ CRYPTO OPENSSL ENGINE API

The EmSPARK Suite supports the use of the TEE based crypto engine via OpenSSL. The CoreLockr Crypto OpenSSL Engine executes cryptographic operations in the TEE using underneath the CoreLockr Crypto API. The CoreLockr Crypto OpenSSL Engine API has functions that allow:

- Loading the Engine
- Setting the Engine behavior
- Configuring the Engine's capabilities

After the Engine is loaded, OpenSSL EVP libraries can automatically use it for calculations in applications. Some functionality is available when loading the engine on the OpenSSL command line.

In the Kit, `corelockr/corelockr_ssl` contains:

- `lib`, `libclrc_util.a` library
- `include`, header files
- `ssl_engine`, the `libclrc.so` OpenSSL engine
- `docs`, for library documentation see `~/docs/html/index`
- `README.txt`, general API information
- `COPYRIGHT`, copyright notice
- Example applications

In this document **6.1 OpenSSL with Crypto in TrustZone for Secure Communication** describes an example of a server and a client establishing a TLS/SSL connection, **6.2 OpenSSL with Crypto in TrustZone for Cryptographic Functions** describes an application that loads the Engine to execute cryptographic operations called from the OpenSSL EVP libraries, **and 6.3 OpenSSL Using Named Keys Stored in the TEE** illustrates how to load keys stored in the Key Store into OpenSSL. This tutorial includes examples of OpenSSL command-line commands in **6.4 OpenSSL Command Line**.

The Engine supports the algorithms supported by the Crypto API. Note that DES algorithms may not be enabled in OpenSSL.

6.1. OpenSSL with Crypto in TrustZone for Secure Communication Example

This example consists of a server and a client establishing a TLS/SSL connection.

6.1.1. Background

For simplicity of setting up the example and avoiding the configuration of a server in a different system, both client and server run on the device. The CoreLockr Crypto OpenSSL Engine API is used to load the engine in the client application and make it the default for doing crypto operations within the TLS stack. The client connects to the server and, after the handshake completes, sends and receives a brief text message. The example includes code to build the server.

Software and Data Requirements

- Client application, `corelockr/corelockr_ssl/example/client`
- Server application, `corelockr/corelockr_ssl/example/server`
- Client and server certificates, `corelockr/corelockr_ssl/example/client/certs` and `corelockr/corelockr_ssl/example/server/certs`

Building and Installing

Change to `corelockr/corelockr_ssl/example/` directory and execute `make` to build the client and the server application executables. Client and server are preconfigured to run on the board and read their certificates from a known location.

Transfer the client to the board:

- executable, `client/clrc_ssl_demo_client`
- configuration file, `client/demo_client.conf`
- certificates, consisting of client private key (EC P-256) and CA certificates, `client/certs/*`

Also transfer the server to the board:

- executable, `server/clrc_ssl_demo_server`
- configuration file, `server/demo_server.conf`
- certificates consisting of the server private key and CA certificates, `server/certs/*`

See `corelockr_ssl/example/README.txt` for additional information.

6.1.2. Executing the Example

Start the Server

On the board, change to the directory where the `clrc_ssl_demo_server` application was transferred to and execute:

```
./clrc_ssl_demo_server
```

The server program is listening at the configured port for the client to connect to it.

Start the Client

On the board, in another shell, execute the following command to start the client:

```
./clrc_ssl_demo_client
```

The client application prints messages such as

```
CoreLockr Cipher Probe started
Starting SSL test
```

The client application may be executed with the "`-p`" print option which prints the available TLS suites and engine ciphers i.e. `clrc` cipher list.

See the Client / Server communication output.

Eventually the client application prints to the console:

```
Got chat response: Hi from server!
```

The server application prints to the console:

```
Got chat: Hi from client!
Sent chat: Hi from server!
Ssl_read: SSL_ERROR_ZERO_RETURN
```

The last message is not an error; it means that the connection was closed.

The server code does not use the CoreLockr Crypto OpenSSL Engine in this demonstration. The client code does use the engine under the hood. For flow and code walkthrough, see `corelockr_ssl/example/README.txt`.

6.2. OpenSSL with Crypto in TrustZone for Cryptographic Functions Example

The CoreLockr Crypto OpenSSL Engine (`clrc`) can be loaded in OpenSSL to perform cryptographic operations in the TEE:

- In applications using the OpenSSL EVP API
- In the OpenSSL command line utility

The example application loads the Engine and executes cryptographic operations using the OpenSSL EVP libraries. **6.4 OpenSSL Command Line** presents examples of loading the Engine in the OpenSSL command line using the "`-engine`" switch.

6.2.1. Background

This example uses the OpenSSL EVP API. The application loads the `clrc` engine and uses it to perform cryptographic operations using the EVP cipher routines. The example illustrates how to load the `clrc` engine. The application executes sample operations such as random number generation, computation of message digests, symmetric encryption and decryption and CMAC computations.

Software and Data Requirements

CoreLockr EVP example application, `corelockr/corelockr_ssl/example_evp`

Building and Installing

Change to `~/corelockr_ssl/example_evp` and execute `make` to build the executable `clrc_evp_sample`. Transfer `clrc_evp_sample` to the board. For additional instructions for building and executing the application, and explanation on flow and code walkthrough to load the engine please see `corelockr_ssl/example_evp/README.txt`.

6.2.2. Executing the Example

Change to the directory where the application binary was copied and execute it:

```
./clrc_evp_sample
```

The application prints to the console the computed results of crypto operations such as:

```
RNG
Message digests: SHA256
Encrypt: AES-256-CTR
Decrypt: AES-256-CTR
CMAC-AES-128-CBC
```

Load Engine Code Walkthrough

When the application starts and before executing cryptographic operations, the application calls the `clrcLoadEngine()` function of the CoreLockr Crypto OpenSSL Engine API to load the CoreLockr Crypto OpenSSL Engine. The engine code is implemented as a shared library named `libclrc.so`. This file must be dynamically loaded by the OpenSSL library before it can be used. This sample function accomplishes that:

```
int load_engine() {
int ret = 0, hide_private = 0;
extern evp_test_params params;
ret = clrcLoadEngine(params.dyn_lib, ENGINE_METHOD_ALL, hide_private);
return (ret == 1) ? 0 : -1;
}
```

Detailed `load_engine()` function overview:

- `int ret = 0, hide_private = 0;`

  The OpenSSL engine control API provides a way to pass information to an engine that cannot be passed via the EVP interface. In the case of the CoreLockr Crypto OpenSSL engine, one command is currently available: the `HIDE_PRIVATE` command. It is used for setting a flag in the engine code that determines whether the private values of keys will be exposed outside of the engine. Setting the flag to a non-zero value has two effects:

  - OpenSSL key structures do not contain the private values
  - Any new key added to the underlying CoreLockr Crypto system has the `CLRC_ATTR_EXPORT_AS_PLAIN` attribute disabled so that the private values cannot be read using the attribute fetching functions.

- `extern evp_test_params params;`

  The `libclrc.so` library is defined as a parameter to be used in the EVP test, e.g.

```
evp_test_params params = {
 .dyn_lib = "./libclrc.so",
};
```

- `ret = clrcLoadEngine(params.dyn_lib, ENGINE_METHOD_ALL, hide_private);`

  The `clrcLoadEngine()` function loads the engine and sets the default behavior, where:

  - `params.dyn_lib` is the full path to the `libclrc.so` engine
  - `ENGINE_METHOD_ALL` indicates the engine methods to be enabled
  - `hide_private` is a flag for disabling access to keys' private values

    Returns 1 on success, or 0 on error and the OpenSSL error stack is updated.

    On success, the CoreLockr Crypto OpenSSL engine shared library is loaded, the specified engine methods are registered, and the flag for hiding private values in keys is set.

    From this point, the EVP functions set up the context with the CoreLockr Crypto OpenSSL engine.

6.3. OpenSSL Using Named Keys Stored in the TEE Example

The CoreLockr Crypto OpenSSL Engine (`clrc`) enables applications using the OpenSSL EVP libraries to access named keys saved in the Key Store in the TEE. This example consists of three interdependent applications whose functions are:

- Create and store an ECDSA key in the Key Store, using the Crypto API for key store management. Save the key under the name `ECC-256-KEY` in the key store.
- Load a named key from the Key Store into OpenSSL, using the EVP libraries for crypto operations with the key. The application loads `ECC-256-KEY` from the Key Store.
- Delete a named key from the Key Store using the Crypto API. The application deletes `ECC-256-KEY` from the Key Store.

6.3.1. ECDSA Key Creation and Storing in the Key Store

This application illustrates how to create a key and save it in the store as a named key that other applications can use. The application uses the CoreLockr Crypto API to create an ECDSA key and store it in the Key Store with no password and named as `ECC-256-KEY`. The **6.3.2 Named Key Use with OpenSSL** example uses the `ECC-256-KEY` key from the Key Store.

For CoreLockr Crypto API documentation, please see
`corelockr/corelockr_crypto/CoreLockr_Cryptographic_API.pdf`

Software and Data Requirements

- Store named key example,

`corelockr/corelockr_ssl/example_key_store_with_openssl/store_named_key`

Building and Installing

In Linux, change to the
`~/corelockr_ssl/example_key_store_with_openssl/store_named_key` directory and

execute `make`. The compilation creates the `clrc_store_named_key` executable. Transfer `clrc_store_named_key` to the board in a directory of your choosing.

Executing the Example

Change to the directory where the application binary was copied and execute:

```
./clrc_store_named_key
```

The application prints the created key attributes and result of storing the key in the key store, e.g.:

```
Creating ECDSA key:
     Printing out ECDSA key attributes
Saving the named key in the key store:
     Successfully saved ECC-256-KEY key in the key store, without  password
```

Note that if the `ECC-256-KEY` name already exists in the Key Store, the application exits with an error. In such a case, to delete the key name from the key store, execute the `clrc_delete_named_key` application explained in **6.3.3 Named Key Deletion from the Key Store**.

Flow and Code Walkthrough

For information see
`~/corelockr_ssl/example_key_store_with_openssl/store_named_key/README.txt`

6.3.2. Named Key Use with OpenSSL

OpenSSL can use keys stored in the TEE. This application illustrates how to load a named key from the Key Store into OpenSSL. The application uses the CoreLockr OpenSSL engine API to load the "`clrc`" OpenSSL engine and uses the OpenSSL EVP libraries to execute the following functions:

- Load `ECC-256-KEY`, an ECDSA key previously stored in the TEE, into an EVP_PKEY via the engine
- Use the ECDSA private key for signing data
- Use the ECDSA public key for verifying the signature

Software and Data Requirements

`~/corelockr_ssl/example_key_store_with_openssl/clrc_use_named_key`

Building and Installing

Change to `corelockr_ssl/example_key_store_with_openssl/clrc_use_named_key/` directory and execute `make` to build the application binary, `clrc_use_named_key`. Transfer `clrc_use_named_key` to the board.

Executing the Example

The application loads `ECC-256-KEY` from the key store to execute cryptographic operations. To save the key in the key store, previously execute `clrc_store_named_key`, explained in **6.3.1**. Change to the directory where the binary was transferred and execute it:

```
./clrc_use_named_key
```

When the named key exists, the application prints messages:

```
Load CoreLockr Cryptographic Engine
Successfully loaded private key: ECC-256-KEY
Computed signature using loaded private key
Successfully loaded public key: ECC-256-KEY
Successful signature verification
```

Flow and Code Walkthrough

For information see
`~/corelockr_ssl/example_key_store_with_openssl/clrc_use_named_key/README.txt`

6.3.3. Named Key Deletion from the Key Store

This application deletes the `ECC-256-KEY` key from the key store. The application uses the CoreLockr Crypto API to delete the key. If the key name exists, the application deletes it, otherwise it prints an error message.

For CoreLockr Crypto API documentation, please see
`corelockr/corelockr_crypto/CoreLockr_Cryptographic_API.pdf`

Software and Data Requirements

- Delete named key example,
  `corelockr/corelockr_ssl/example_key_store_with_openssl/ delete_named_key`

Building and Installing

In Linux, change to the `corelockr/corelockr_ssl/example_key_store_with_openssl/delete_named_key` directory and execute `make`. The compilation creates the `clrc_delete_named_key` executable. Transfer `clrc_delete_named_key` to the board in a directory of your choosing.

Executing the Example

Change to the directory where the application binary was copied and execute:

```
./clrc_delete_named_key
```

The application prints the result of deleting the key from persistent storage, e.g.

```
ECC-256-KEY key has been deleted from the key store
```

Flow and Code Walkthrough

See `~/corelockr_ssl/example_key_store_with_openssl/ delete_named_key`

6.4. OpenSSL Command Line

The CoreLockr Crypto OpenSSL Engine can be loaded on the OpenSSL command-line utility to execute some operations. If `libclrc.so` is stored within the default location (e.g. `/usr/lib/arm-linux-gnueabihf/openssl-1.0.0/engines/` on Ubuntu ARM systems), then it can be loaded by the OpenSSL utility using the "`-engine clrc`" option, for example, secure hashing:

```
echo Sample | openssl sha256 -engine clrc
```

The OpenSSL command line cannot use key tokens and therefore is unable to access keys stored in the TEE.

7. CoreLockr™ TLS IO API

The Transport Layer Security (TLS) protocol is used to provide secure communication over the internet. It is designed to protect the secrecy and integrity of the communication from external factors. The CoreLockr TLS IO API is the Rich OS interface to an implementation of TLS client code that runs within the TEE. Running the TLS code within the TEE provides some security from snooping and tampering on the device itself.

The CoreLockr TLS IO provides access to named keys and certificates that exist within the CoreLockr secure storage system, such as provisioned keys and certificates, Key Store and Certificate Store. It also provides access to cryptographic operations within the CoreLockr Crypto system in the TEE. The CoreLockr Secure Certificates system is used to verify the peer's certificates during the mutual authentication step when the TLS communication is being established.

Kit Contents

In the Kit, `corelockr/corelockr_tlsio` contains:

- `lib`, `libseqr_corelockr_tlsio.so` library
- `include`, header file
- `ta`, `A37F954E-303F-6D5D-B81685157929ADA2.stp` associated TA
- `docs`, for library documentation see `~/docs/html/index`
- `README.txt`, general API information
- `COPYRIGHT`, copyright notice
- Example application

7.1. Communication with a Server Example

The example uses the CoreLockr TLS IO API in the client application to communicate with an OpenSSL-based server. The client connects to the server and, after the handshake completes, sends and receives a brief text message.

There are two mutual authentication scenarios demonstrated in this example:

1) The client uses the provisioned device key and certificate,

2) The client uses an externally created key and certificate.

In the first scenario, the device key and certificate are specified using the names under which they are stored in secure storage. The key contents are not exposed to the Rich OS. In the second scenario, the key and certificate are provided as PEM-format files from the Rich OS. In both cases, the TLS encryption takes place within the CoreLockr TLS IO Trusted Application, and all of the secrets associated with that are kept in the TEE.

The server program is the same as is used **in 6.1 OpenSSL with Crypto in TrustZone for Secure Communication** example. However, different configuration files and certificates/keys are provided for the two scenarios in this example. For scenario 1, the server requires the OEM root certificate for verifying the device's certificate. Its own certificate was signed with the same CA, so the device verifies it with the provisioned OEM Root Certificate. For scenario 2, the server and device certificates were signed with the same external CA, and both sides require the `ca-ext.crt` certificate to verify each other's certificate.

Building and Installing

- Run "`make`" within the example directory to build the `client/clr_tls_demo_client` and `server/clrc_ssl_demo_server` applications
- Transfer the `client/clr_tls_demo_client`, `client/demo_client.conf.prov`, `client/demo_client.conf.ext`, `server/clrc_ssl_demo_server`, `server/demo_server.conf.prov` and `server/demo_server.conf.ext` files to the board in a directory of your choosing. Transfer the private EC client and server keys, and the client, server and CA certificates to a certs/ directory in that same location.

Flow and Code Walkthrough

For information see `~/corelockr_tlsio/example/README.txt`.

Executing the Example

These steps must be performed on the board in Linux.

- Change to the directory where the example applications were transferred to.
- If the Trusted Application was built to use the CoreLockr Secure Certificates API for verification, then the `certs/ca-ex.crt` certificate must be added to the Certificate Store on the device for scenario 2 to work properly. The `clrsc_example` application from the `corelockr_cert` example can be used to add the certificate. Build that example if necessary, and then run the following command

```
<path-to-example-application>/clrsc_example a certs/ca-ext.crt
```

The CA certificate for scenario 1 was provisioned on the board, so adding it to the secure certificate store is not necessary.

- Execute the following command to start the demo server for scenario 1:

```
./clrc_ssl_demo_server -f demo_server.conf.prov
```

  or for scenario 2:

```
./clrc_ssl_demo_server -f demo_server.conf.ext
```

  The server program is listening at the configured port for the client to connect to it. The server is configured to use different ports for the two scenarios, so both instances can be left running at the same time.

- In another shell, execute the following command to start the demo client for scenario 1:

```
./clr_tls_demo_client -o demo_client.conf.prov
```

  or for scenario 2:

```
./clr_tls_demo_client -o demo_client.conf.ext
```

  The client application should eventually print:

```
"Got chat response: Hi from server!"
```

  The server application should print:

```
"Got chat: Hi from client!
 Sent chat: Hi from server!
 Ssl_read: SSL_ERROR_ZERO_RETURN"
```

The last message is not an error; it means that the connection was closed.

8. CoreLockr™ Secure Certificates API

The CoreLockr Secure Certificates library and accompanying Trusted Application (TA) leverage the capabilities of CoreTEE™ to provide secure storage and management of X.509 v3 Certificate Authority certificates (CAs). This section provides an overview of the API functionality and example applications. The CoreLockr TLS IO API uses the CoreLockr Secure Certificates system to verify the peer's certificates during the mutual authentication step when the TLS communication is being established. The API has functionality to:

- Verify certificates against known Certificate Authorities stored in the Certificate Store and provisioned certificates
- Manage Certificate Authority certificates: add, update and delete CA certificates stored in the Certificate Store in the TEE
- Manage a Certificate Revocation List, CRL
- Manage the signing key of certificate management commands
- Enable rotation of certificates provisioned on the device
- Extract from the TEE the provisioned certificates
- Extract from the TEE the Device Certificate Signing Request (CSR) generated during provisioning

Management commands modifying the Certificate Store or the provisioned certificates must be signed with an authorized key, this way the OEM has ownership of the certificates managed in the TEE. The authorized signing key is the OEM Command Key. Such commands are generated on a secure server and sent to the device for verification and execution.

On the device, the Secure Certificates API verifies the signature before executing such commands. On the device, the API verifies the signature using the OEM Command Public Key (`CLRC_OEM_COMMAND_PUBLIC_KEY`) contained in the OEM Command Certificate in the TEE (`CLRSC_OEM_COMMAND_CERT`). Please see **Table 1 – Provisioned Keys and Certificates for OEM Usage Scenarios**.

Note that commands to verify certificates against known CAs in the TEE do not need to be signed.

There are two classes of certificates in the CoreLockr Secure Certificates API:

- Provisioned certificates, these certificates are provisioned in non-volatile memory when the device firmware is installed. These certificates can be accessed and managed through the device lifecycle using proper verifications
- Certificate Store, these certificates exist entirely within the TEE's persistent object store. The Certificate Store includes Certificate Authorities managed during runtime using the Secure Certificates API. It also includes the certificate revocation list.

The two classes are managed differently within the API. The sections below explain the differences.

Kit Contents

In the Kit, `corelockr/corelockr_cert` contains:

- `docs`, library documentation
- `lib`, `libseqr_corelockr_cert.so` library
- `include`, header files
- `ta`, `222A521C-62CB-0653-BCE5DD727660FAF0.stp` associated TA
- `README.txt`, general API information
- `COPYRIGHT`, copyright notice
- `examples`, example application with source code, explained in **8.1**
- `README_named_certificates.txt`, description of STP structure and STP file creation to update provisioned certificates
- `~/bin/make_named_cert_stp.sh`, reference script to generate STP files for updating provisioned certificates

In `corelockr/examples/AWS` is an example application explained in **8.4 Connecting to AWS IoT Core**.

8.1. Provisioned Certificates

These certificates are provisioned in non-volatile memory when the device firmware is installed. The Secure Certificates API allows access to these certificates by a name defined in the `clrsc_ta_commands.h` header file, i.e.

```
~/corelockr_cert/include/clrsc_ta_commands.h
#define CLRSC_OEM_CLOUD_CERT      "com.seqlabs.oem_cloud_cert"
#define CLRSC_OEM_ROOT_CERT       "com.seqlabs.oem_root_cert"
#define CLRSC_OEM_PAYLOAD_CERT    "com.seqlabs.oem_payload_cert"
#define CLRSC_OEM_COMMAND_CERT    "com.seqlabs.oem_command_cert"
#define CLRSC_OEM_DEVICE_CERT     "com.seqlabs.oem_device_cert"
#define CLRSC_OEM_DEVICE_CSR      "com.seqlabs.oem_device_csr"
#define CLRSC_EMPOWER_CLOUD_CERT  "com.seqlabs.emp_cloud_cert"
#define CLRSC_EMPOWER_ROOT_CERT   "com.seqlabs.emp_root_cert"
#define CLRSC_EMPOWER_DEVICE_CERT "com.seqlabs.emp_device_cert"
#define CLRSC_EMPOWER_DEVICE_CSR  "com.seqlabs.emp_device_csr"
```

Provisioned certificates are updated using the `clrscUpdateNamedCertificate()` function. Updating a named certificate also updates the public key it contains. Such public keys can be accessed using the CoreLockr Crypto API, as explained in **2 CoreLockr™ Crypto** API. The device certificates whose associated keys are generated during provisioning are exceptions, `CLRSC_OEM_DEVICE_CERT` and `CLRSC_EMPOWER_DEVICE_CERT`. Updating these device certificates only updates the certificate data but not the corresponding device key.

Note: The EMSPARK_KEYS_CERTS.pdf describes the certificates and keys for provisioning on the device, their origin and configuration. It also describes how such certificates and keys can be accessed from the Rich OS after provisioning. This document is provided with the EmSPARK Development Kit or upon request.

For reasons of security, when updating provisioned certificates, the new certificate file, its name as declared in `clrsc_ta_commands.h`, command and OEM Command Key signature must be provided in the form of an STP file. The `README_named_certificates.txt` explains the STP file format and

storage of the provisioned certificates. The supplied `bin/make_named_cert_stp.sh` shell script is a reference implementation for creating the STP files.

The OEM Command Key can be changed through the device life cycle. The Secure Certificates API allows updating the OEM Command Public Key by replacing the `CLRSC_OEM_COMMAND_CERT` certificate. To make this change, as with any management command, the command updating the `CLRSC_OEM_COMMAND_CERT` must be signed with the current OEM Command Key and verified on the device.

Updating any of the provisioned certificates requires `root` privileges. The command to update the provisioned certificates writes back to the partition where the manifest containing the certificates is stored. Writing to this partition is restricted to `root`.

Figure 4 illustrates the command flow to update a provisioned certificate. On a customer server, a STP file is created containing the certificate name, certificate contents, command and signature of that data. The STP file shall be transferred to the device. On the device, an application uses the Secure Certificates API to verify the signature against the OEM Command Public Key available in the TEE. The API update function validates and authenticates the STP file contents. If the verification succeeds, the command updating the provisioned certificate is executed. The command updates the certificate in the manifest and its copies as described in `README_named_certificates.txt`.

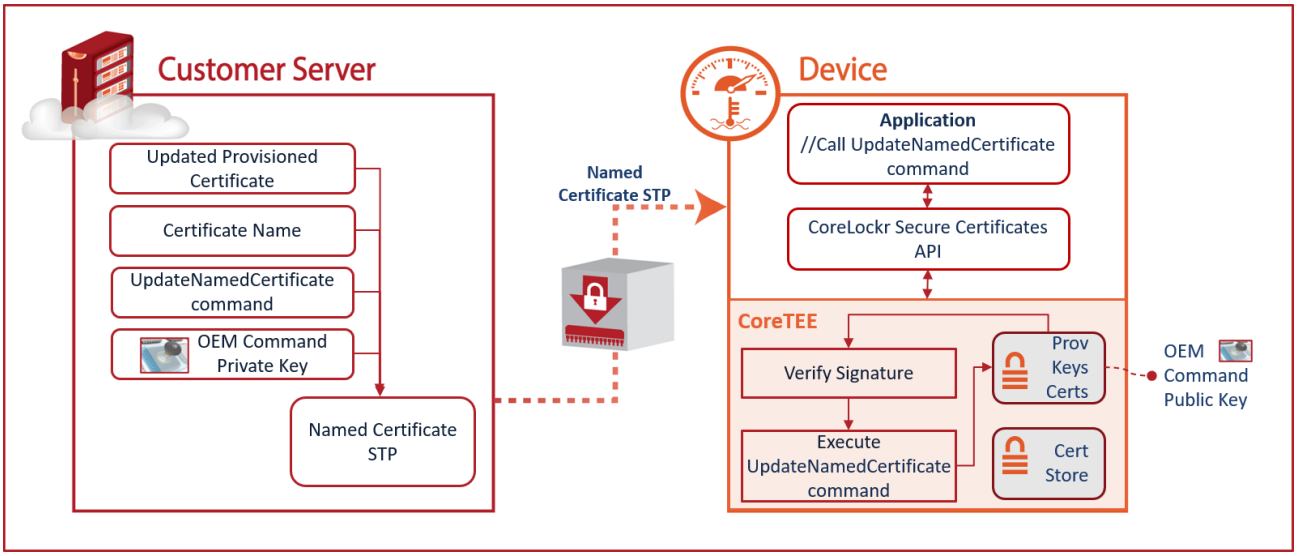


Figure 4 Provisioned Certificate Management Flow

8.2. Certificate Store

The Certificate Store includes Certificate Authority certificates managed through the device life cycle. It also includes the certificate revocation list (CRL). The Certificate Store uses an offline CRL, therefore, the OEM must add certificates to this Revocation List manually. Any attempt to connect this CRL to an online source would traverse the Rich OS (non-secure world) breaking the isolation provided by the TEE.

Commands modifying the Certificate Store must be signed with the Command Verification Private Key. On the device, the OEM Command Verification public key is used to verify that the command is legitimate and can be executed. These commands include certificate authority management functions such as `clrscAddCertificateAuthority()`, `clrscDeleteCertificateAuthority()`, `clrscUpdateCertificateAuthority()`, and certificate revocation functions, `clrscAddCertificateRevocation()`.

Commands to verify certificates against known CAs stored in the Certificate Store do not need to be signed.

Figure 5 illustrates the flow of a management command. On a customer server, a command modifying the Certificate Store and the certificate in question are signed with the OEM Command Private key (the DER encoded certificate and command are hashed and the hash signed). The signed management command and CA certificate shall be transferred to the device. On the device, an application uses the Secure Certificates API to verify the signature against the OEM Command Public Key available in the TEE. If the verification succeeds, the command modifying the Certificate Store is executed.

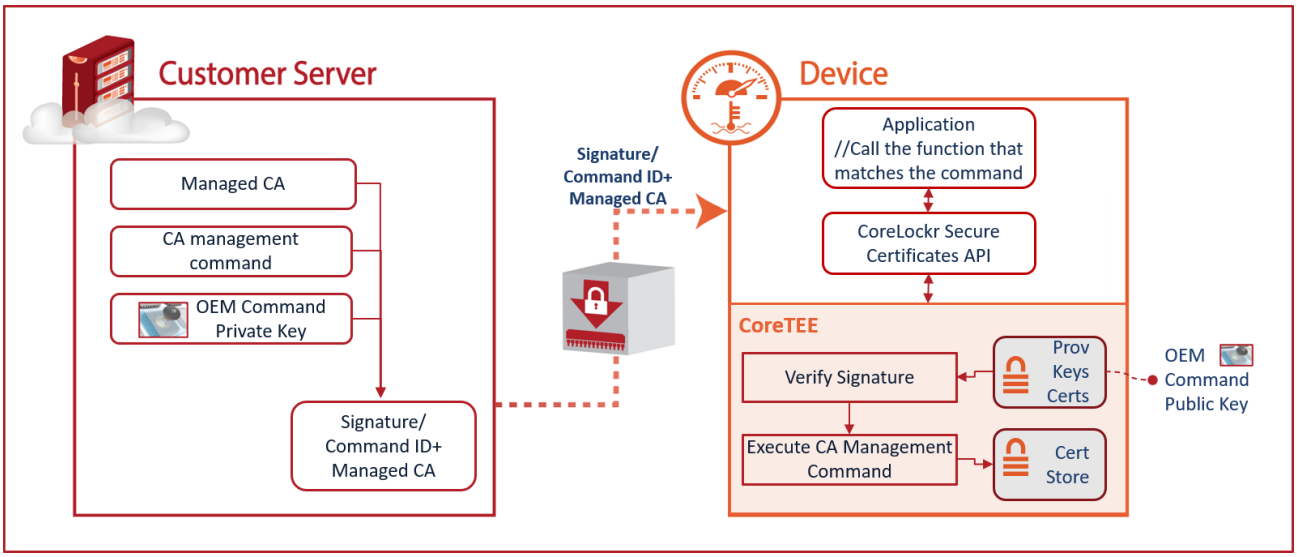


Figure 5 Certificate Store Management Flow

8.3. Certificate Authority Management Example

The example application illustrates capabilities of the EmSPARK Suite and the CoreLockr Secure Certificates (CLRSC) library to provide secure storage and management of X.509 v3 certificates.

In order to simplify the configuration and building of the example, operations usually performed on a secure server and securely sent to the board are instead executed on the board. For instance, commands for certificate authority (CA) management that would be produced and signed on a server and sent to the device to be executed are instead produced and signed on the board.

8.3.1. Background

The application performs the following operations:

- Load Certificate Authority certificates and certificates from the local file system
- Delete Certificate Authority certificates

- Update Certificate Authority certificates
- Verify certificates
- Add certificates to the Certificate Revocation List (CRL) in the TEE
- Extract certificates preinstalled in the TEE and write them as files in PEM format
- Update the provisioned certificates preinstalled in the TEE

The application uses the OpenSSL library to load certificates from the local file system, convert the certificates from X509 structures to a DER encoded byte string and pass the certificates to the CLRSC API.

Software and Data Requirements

- CoreLockr Secure Certificates example application, `corelockr/corelockr_cert/examples`
- Key and certificate files provided with the example
  - OEM Command Private Key `clrsc_example_command_key.pem`: used to sign the certificate management commands
  - OEM Command Certificate: installed during provisioning and loaded in the TEE, exposed to the API as `CLRSC_OEM_COMMAND_CERT`
  - Certificate authority sample: `clrsc_example_ca_cert.pem`
  - Certificates issued by sample CA `clrsc_example_ca_cert.pem`: `clrsc_example_server_01.pem` and `clrsc_example_server_02.pem`
  - `clrsc_example_ca_key.pem` private key to enable the user to generate additional certificates

In the example, the OEM Command Private Key corresponds to an OEM private key used to sign commands. This private key used for signing the certificate management commands usually is not found on the device. To make easy to set up the test application and avoid the configuration of an external system that sends signed commands to the board, the private key is provided and the example application on the board uses it to sign the commands.

The application uses the OEM Command Private Key `clrsc_example_command_key.pem` to sign the following commands sent to the TEE (such commands usually are received from a secure server):

- Load Certificate Authority certificates and certificates from the local file system
- Delete Certificate Authority certificates
- Update Certificate Authority certificates
- Add certificate to the Certificate Revocation List (CRL) in the TEE. This is specific for certificates.

Building and Installing

Change to `corelockr/corelockr_cert/examples` and execute `make` which builds the executable `clrsc_example`. Transfer the executable to the board to a directory of your choice. Transfer to the same directory the `certs` directory containing the certificates. For additional instructions see `corelockr_cert/examples/README.txt`.

8.3.2. Executing the Example

To manage certificates, verify that the date on the board is current and execute the commands as instructed below.

Add a Certificate Authority to the TEE

To add a certificate authority to the TEE execute

```
./clrsc_example a certs/clrsc_example_ca_cert.pem
```

where "a" adds the CA certificate, and "certs/clrsc_example_ca_cert.pem" is the path to the file containing the certificate.

The application prints messages, including a confirmation:

```
Successfully saved certificate to TEE
```

Verify the Certificate Authority

To verify a certificate authority execute

```
./clrsc_example v certs/clrsc_example_ca_cert.pem
```

where "v" verifies clrsc_example_ca_cert.pem against the known certificate authorities.

After successful verification, the application prints:

```
Result of CLRSC verify function is: 1. Certificate is: VERIFIED
```

The verification of the clrsc_example_ca_cert.pem CA certificate is successful because the certificate is self-signed and the certificate was already added in the TEE. Attempting to verify an unknown CA certificate will return "NOT VERIFIED".

Verify Certificates against the CA

To verify the example certificates against the CA execute

```
./clrsc_example v certs/clrsc_example_server_01.pem
./clrsc_example v certs/clrsc_example_server_02.pem
```

The verification of these certificates is successful because the issuing CA (clrsc_example_ca_cert.pem) was previously added in the TEE.
Verification of a certificate issued by an unknown CA returns "NOT VERIFIED".

Add a Certificate to the Certificate Revocation List (CRL)

To add a certificate to the CRL execute

```
./clrsc_example r certs/clrsc_example_server_01.pem
```

which produces this output:

```
Successfully add certificate to CRL in TEE
```

After added to the CRL, the certificate does not verify. Execution of

```
./clrsc_example v certs/clrsc_example_server_01.pem
```

produces this output

```
Result of CLRSC verify function is: 0. Certificate is: NOT VERIFIED
```

Entries to the CRL have effect only over certificates. Adding CA certificates to the CRL simply prevents the CA from being verified, but certificates issued with such CA still verify. This is standard functionality.

After adding a certificate to the revocation lists, an attempt to add it to the TEE again will return an error.

Update a Certificate

A certificate may be updated

```
./clrsc_example u certs/clrsc_example_server_02.pem
```

which returns

```
Successfully updated certificate in TEE
```

Revoke a Certificate Authority

A Certificate Authority may be removed from known Certificate Authorities in the TEE

```
./clrsc_example d certs/clrsc_example_ca_cert.pem
```

After the CA removal, the certificates issued with the removed CA do not verify.

Extract Certificates from the TEE

Certificates saved in the TEE during the firmware flashing can be extracted and written as files in PEM format. The following certificates saved in the TEE have a known name and assigned an ID in the application:

```
ID = 0 - Get OEM Cloud Certificate
ID = 1 - Get OEM Root Certificate
ID = 2 - Get OEM Payload Certificate
ID = 3 - Get OEM Command Certificate
ID = 4 - Get OEM Device Certificate
ID = 5 - Get OEM Device CSR
ID = 6 - Get EmPower Cloud Certificate
ID = 7 - Get EmPower Root Certificate
ID = 8 - Get EmPower Device Certificate
ID = 9 - Get EmPower Device CSR
```

To generate a file of the OEM Cloud Certificate which is option "`ID = 0`" execute

```
./clrsc_example g 0
```

This creates `clrsc_oem_cloud_cert.pem` located in the directory where the application was executed. If the device was flashed with the provided certs in the Kit, `clrsc_oem_cloud_cert.pem` corresponds to the AWS IoT root CA certificate.

To extract the OEM Root Certificate, execute

```
./clrsc_example g 1
```

which creates `clrsc_oem_cert.pem`

To extract the OEM Device Certificate, execute

```
./clrsc_example g 4
```

which creates `clrsc_device_cert.pem`

Return Codes

The CoreLockr Secure Certificates API returns status codes from the TEE. Please refer to the `coretee_dev_kit/tc_sdk/include/tee_client_api.h` or the API documentation for return codes.

8.4. Connecting to AWS IoT Core

This example illustrates how to use the EmSPARK Suite and mbedTLS to connect to an AWS server. This section explains concepts and how to configure the example. The contents are located in `corelockr/examples/AWS` which includes:

- Application source code
- AWS embedded C SDK Version 3.0.1
- Patch against the 3.0.1 version
- Sample files to generate a user's CA

8.4.1. Background

AWS allows devices to use X.509 certificates signed and issued by a customer defined certificate authority (CA) to connect and authenticate with AWS IoT Core. This is one of the methods allowed for authentication by "things" using MQTT protocol. MQTT is using TLS as a secure transport mechanism. In IoT, each "thing" needs to be uniquely identified by the cloud application and that is realized by using device certificates as identifiers. More information can be found in the following references:

- https://docs.aws.amazon.com/iot/latest/developerguide/client-authentication.html
- https://docs.aws.amazon.com/iot/latest/developerguide/iot-authorization.html
- https://docs.aws.amazon.com/iot/latest/developerguide/life-cycle-events.html

Once a device is registered with the AWS IoT Core, the authentication is performed using a standard TLS mutual authentication based on the X.509 certificate associated to the thing. To register a device this example uses the Just in Time Provisioning, which will check an unknown device certificate's signing Certificate Authority (CA). If the CA is in the list of CA's on the IoT Core, then the registration process is performed. More information can be found here:

https://docs.aws.amazon.com/iot/latest/developerguide/jit-provisioning.html.

The device certificate is intended for establishing TLS connections with mutual authentication. This example illustrates how to use the device certificate and TrustZone based crypto for establishing a TLS connection with Amazon AWS IoT Core. The example will demonstrate the TLS Connection, using MQTT over TLS, and interacting with Device Shadow.

During initial device provisioning, the EmSPARK Suite creates the OEM Device Key and signs a Device Certificate with the *EmSPARK Defined OEM Key*. To execute TLS mutual authentication and session establishment with Amazon AWS IOT, the user will update two provisioned certificates managed in the TEE with user's generated certificates:

- **OEM Root CA**, the EmSPARK defined OEM CA provisioned on the device will be replaced with a user defined OEM Root CA certificate.
- **OEM Device Certificate** generated during provisioning and signed with the EmSPARK defined OEM Key will be replaced with a device certificate signed with the user defined OEM Root Key.

The OEM Device Certificate needs to be signed with the private key that created the OEM Root CA so the two certificates are chained. The EmSPARK Suite provides the tools to generate the new OEM Device Certificate and update the OEM Root CA and Device Certificate in the TEE. The sections below describe how to perform the needed steps.

The example requires the user to create an AWS account and upload the OEM Root CA to AWS. This is due to the fact that Amazon AWS does not allow the activation of the same OEM Root CA for multiple AWS accounts and the EmSPARK Defined OEM CA is the same on all devices.

The example uses the following keys and certificates:

1. **OEM Cloud Certificate,** during provisioning, the Evaluation Kit installs the ***AWS IoT Root Certificate*** required for communication with AWS, `CLRSC_OEM_CLOUD_CERT`.
2. **OEM Root Key,** this private key is generated by the user. It is used to sign the OEM Device Certificate and to complete the AWS Custom CA Certificate registration process with AWS IoT.
3. **OEM Root CA**, the user creates this certificate authority and saves it in the TEE, `CLRSC_OEM_ROOT_CERT`. This certificate replaces the EmSPARK `CLRSC_OEM_ROOT_CERT` provisioned on the device and the associated OEM Public Key it contains.
4. **OEM Device Key,** the device private and public keys are generated during provisioning and stored in the TEE, `CLRC_OEM_DEVICE_PRIVATE_KEY` and `CLRC_OEM_DEVICE_PUBLIC_KEY`. Using the CoreLockr APIs, applications can access the OEM Device Private and OEM Public Key from the Rich OS (i.e. Linux). The OEM Device Private Key can be accessed for operations, even though applications in Linux do not have visibility of the private key attributes. The OEM Device Public Key is the key that the Rich OS (Linux) can see. The OEM Device Key is used to sign the Device Certificate Signing Request, `CLRSC_OEM_DEVICE_CSR`.
5. **OEM Device Certificate,** the user generates the `CLRSC_OEM_COMMAND_CERT` signed with the user's OEM Root Key and saves it in the TEE. The execution of the TLS AWS example will register the Device Certificate with the AWS IoT cloud.
6. **OEM Command Public Key,** installed during provisioning, `CLRC_OEM_COMMAND_PUBLIC_KEY` is used to verify the commands that replace the OEM Root Certificate and OEM Device Certificate in the Certificate Store.

The example preparation includes steps in a Linux development environment, on the device and on AWS Console. The following is an outline of the steps described in the next sections of this document:

- **In the Linux development environment**, build the CoreLockr Secure Certificate example application included in the CoreLockr Kit to manage and store certificates in the TEE, please see **8.4.2**
- **On the Device**, extract the OEM Device Certificate Signing Request
- **In the Linux development environment**, generate custom OEM Root Key and Certificate and OEM Device Certificate, and prepare the files needed for installation on the device
- **On the Device**, customize the OEM Root Certificate and OEM Device Certificate in the TEE, these steps use the CoreLockr Secure Certificate example application, please see **8.4.3**
- **On the AWS Console**, configure user's account for AWS TLS example, see **8.4.6**
  - AWS IoT, register the OEM Root CA certificate
  - AWS IoT, create a policy to manage access in AWS
  - AWS Lambda, create a Lambda function to configure actions during the Just in Time Registration
- **In the Linux development environment**, configure and build the TLS AWS example application, see **8.4.7**

- **On the Device**, execute the TLS AWS example application, see **8.4.8**

Note: In order to simplify the configuration and building of the example, some operations usually performed on a secure server and securely sent to the device are instead executed on the device. For instance, signing the commands to update the certificates on the device that would be produced on a server is instead signed on the device.

8.4.2. Linux Development Environment: Prepare Application and Key for Certificate Updates

The TLS AWS example application uses User's keys and certificates, therefore some certs installed during provisioning need to be replaced. The CoreLockr Secure Certificates API enables the rotation of those certificates. These instructions will use the Certificate Authority Management example -**8.1**- to update the OEM Root Certificate and OEM Device Certificates in the TEE. To accomplish this:

- Build the Certificate Authority Management application binary, `clrsc_example`, explained in **8.1**
- Transfer the `clrsc_example` executable to the device

8.4.3. Device: Extract OEM Device Certificate Signing Request

Use the CoreLockr Secure Certificates example to extract the OEM Device CSR from the TEE:

- Change to the directory where `clrsc_example` was transferred
- Extract the OEM Device CSR from CoreTEE using the `clrsc_example` example

```
./clrsc_example g 5
```

  Where `g 5` extracts the certificate signing request (`ID = 5 – Get OEM Device CSR`). The application generates a DER encoded file, `clrsc_oem_device_csr.der`.

- Transfer the `clrsc_oem_device_csr.der` to the Linux development environment, `~/corelockr/examples/AWS/tools/` directory

8.4.4. Linux: Prepare User's OEM Root Certificate and OEM Device Certificate

Because these are provisioned certificates, to update them on the device requires that an STP file be provided. This allows the operation to be validated and authenticated before it is permitted to continue. This example instructs how to generate the certificates and how to generate the STP files.

To generate the certificates, `~/corelockr/examples/AWS/tools/` contains a sample CA configuration file, `~/AWS/tools/conf/ca.conf`, and auxiliary text files to generate a CA. The user can use these files to generate a custom OEM Root Certificate and OEM Device Certificate.

To generate the STP files, the example instructs the use of `~/corelockr/corelockr_cert/bin/make_named_cert_stp.sh`. The script takes the following arguments:

- Signing key, `-k`: the private key associated with `CLRC_OEM_COMMAND_PUBLIC_KEY` provisioned on the device. This key is used to authenticate commands that change trust on the device. In the Evaluation Kit, the sample private key file associated with `CLRC_OEM_COMMAND_PUBLIC_KEY` is supplied with the Secure Certificates example application.
- ID or name under which the certificate is accessed, `-n`: name of the provisioned certificate defined in `~/corelockr/corelockr_cert/include/clrsc_ta_commands.h`.
- Path to the certificate file, `-c`: the new certificate file in PEM or DER encoding.

- The path to the output STP file, `-o`: file to be transferred to the device containing the certificate name, new certificate contents and signature of that data.

Here is an overview and then detailed instructions.

Overview

1. Generate a new OEM Root Key and OEM Root CA
2. Create a new Device Certificate signed with the user's new OEM Root Key
3. Generate STP files containing the certificates and transfer them to the device

NOTE: The description in the following sections assumes that the **OEM Root Key** is a customer's sample key used for testing purposes.

User Instructions: Detail

- Change to `~/corelockr/examples/AWS/tools` where `clrsc_oem_device_csr.der` was transferred

- Generate the custom OEM Root Key and OEM Root CA. To generate a self-signed CA certificate, in `~/AWS/tools/`:
  - Customize `~/AWS/tools/conf/ca.conf` or produce a CA configuration file used for signing the Device CSR. Note that the example requires that key and cert be ECDSA P256. Those parameters should not be changed to avoid errors during the application execution.

  - Generate new OEM Root Key, in this example the filename is `aws_custom_ca_key.pem`:

    ```
    openssl ecparam -out aws_custom_ca_key.pem -name prime256v1 -genkey
    ```

  - Generate custom OEM Root CA cert, e.g. `aws_custom_ca_cert.pem`:

    ```
    openssl req -x509 -config conf/ca.conf -newkey
    ec:aws_custom_ca_key.pem -sha256 -nodes -out aws_custom_ca_cert.pem -
    outform PEM
    ```

    The user must register the OEM Root CA file with AWS (please see **Register the CA to the AWS IoT** section in **8.4.6**). The following section assumes the OEM Root CA filename is `aws_custom_ca_cert.pem`.

- Generate the custom OEM Device Certificate
  - Convert the OEM Device CSR transferred from the device from DER to PEM encoding

    ```
    openssl req -in clrsc_oem_device_csr.der -inform DER -out
    clrsc_oem_device_csr.pem -outform PEM
    ```

  - Generate the new OEM Device Certificate by signing the PEM encoded OEM Device CSR and using the user's new OEM Root Key and OEM Root CA

    ```
    openssl ca -verbose -config  conf/ca.conf -policy signing_policy -
    extensions signing_req -out new_device_cert.pem -infiles
    clrsc_oem_device_csr.pem
    ```

Where `conf/ca.conf` is user configured and expects `aws_custom_ca_key.pem` and `aws_custom_ca_cert.pem` key and certificate names. The output is `new_device_cert.pem`.

Answer "`y`" to the questions "`Sign the certificate`" and "`commit?`". To comply with the TLS mutual authentication protocol, the new Device Certificate is signed with the new OEM Root Key.

- Generate the STP files containing the certificates generated in the previous step and that replace OEM Root Certificate and OEM Device Certificate. In `~/tools/`:
  - Create a symbolic link in which the source is the key to sign Certificate Store management commands, OEM Command Key, provided with the Secure Certificates API example `~/corelockr_cert/examples/certs/clrsc_example_command_key.pem` and destination is `clrsc_command_signing_key.pem`, e.g.:

```
ln -s ../../../corelockr_cert/examples/certs/clrsc_example_command_key.pem
clrsc_command_signing_key.pem
```

  - Generate the STP file to update the OEM Root Certificate

```
../../../corelockr_cert/bin/make_named_cert_stp.sh -k
clrsc_command_signing_key.pem -n "com.seqlabs.oem_root_cert" -c
aws_custom_ca_cert.pem -o new_oem_root_cert.stp
```

    Where `-k clrsc_command_signing_key.pem` is the private key that authorizes commands, `-n "com.seqlabs.oem_root_cert"` is the name of the OEM Root Certificate defined name, `-c aws_custom_ca_cert.pem` is the new OEM Root Certificate filename and `-o new_oem_root_cert.stp` is the STP file to transfer to the device.

  - Generate the STP file to update the OEM Device Certificate

```
../../../corelockr_cert/bin/make_named_cert_stp.sh -k
clrsc_command_signing_key.pem  -n "com.seqlabs.oem_device_cert" -c
new_device_cert.pem  -o new_oem_device_cert.stp
```

- Transfer to the device the STP files containing the certificate name, certificate contents and signature of that data, e.g. `new_oem_root_cert.stp` and `new_oem_device_cert.stp`.

8.4.5. Board: Customize the Device Certificate and OEM Root Certificate

After transferring the STP files to the device, the user replaces the provisioned certificates with the custom certificates. Here is an overview of steps on the device and then detailed instructions.

1. For the sake of simplicity, the Secure Certificates example application uses the provided sample private key file to sign on the board the commands updating the certificates
2. Use the CoreLockr Secure Certificates example to update the EmSPARK Defined OEM Root CA in the TEE with the new OEM Root CA and store it in non-volatile-memory so that the user's certificate will be persistent after device reboots
3. Use the CoreLockr Secure Certificates example to update the OEM Device Certificate in the TEE with the new one and store it in non-volatile-memory

User Instructions: Detail

Execute the following steps to update in the TEE the OEM Root CA and OEM Device Certificate:

- Change to the directory where the Secure Certificates example application is located, `~/corelockr/corelockr_cert/examples/clrsc_example`
- Transfer to this directory the command signing key located in the Kit in `~/corelockr/corelockr_cert/examples/certs/clrsc_example_command_key.pem`
- The `clrsc_example` example expects that the signing key for certificate management commands be named `clrsc_command_signing_key.pem`, thus create a symbolic link

  ```
  ln -s clrsc_example_command_key.pem clrsc_command_signing_key.pem
  ```

- Replace in the TEE the provisioned OEM Root CA certificate with the new OEM Root CA contained in the STP file. Because updating any of the provisioned certificates requires root privileges, as `root` execute

  ```
  ./clrsc_example n new_oem_root_cert.stp
  ```

  Where `n` updates the named certificate identified in the STP file. Successful execution prints messages such as `Successfully updated the named certificate`, corresponding to `com.seqlabs.oem_root_cert` defined in `clrsc_ta_commands.h`

  ```
  #define CLRSC_OEM_ROOT_CERT "com.seqlabs.oem_root_cert"
  ```

  Updating the OEM Root CA also updates the OEM Root Public Key in the TEE, i.e. the key value defined in `~/corelockr/corelockr_crypto/include`

  ```
  #define CLRC_OEM_PUBLIC_KEY       "com.seqlabs.oem_pub_key"
  ```

- Replace the OEM Device Certificate in TEE with the new certificate contained in the STP file, as `root` execute

  ```
  ./clrsc_example n new_oem_device_cert.stp
  ```

  Where `n` updates the named certificate and contents in the STP file. Successful execution prints messages such as `Successfully updated the named certificate`, corresponding to `com.seqlabs.oem_device_cert` defined in `clrsc_ta_commands.h`

  ```
  #define CLRSC_OEM_DEVICE_CERT   "com.seqlabs.oem_device_cert"
  ```

  Now, the OEM Root CA and OEM Device Certificate are updated with the user created certs.

8.4.6. AWS Console: Configure User's Account for AWS TLS Example

The user must go through the following steps to set and test the TLS connectivity with AWS IoT (step by step instructions are provided by the Amazon website. Some steps are detailed below with screenshots: http://docs.aws.amazon.com/iot/latest/developerguide/iot-gs.html):

- Create an Amazon AWS account
- Sign in to the AWS IoT Console
- Register the CA to the AWS IoT
- Create a Publish/Subscribe Policy
- Create a Lambda Function

1. Register the CA to the AWS IoT

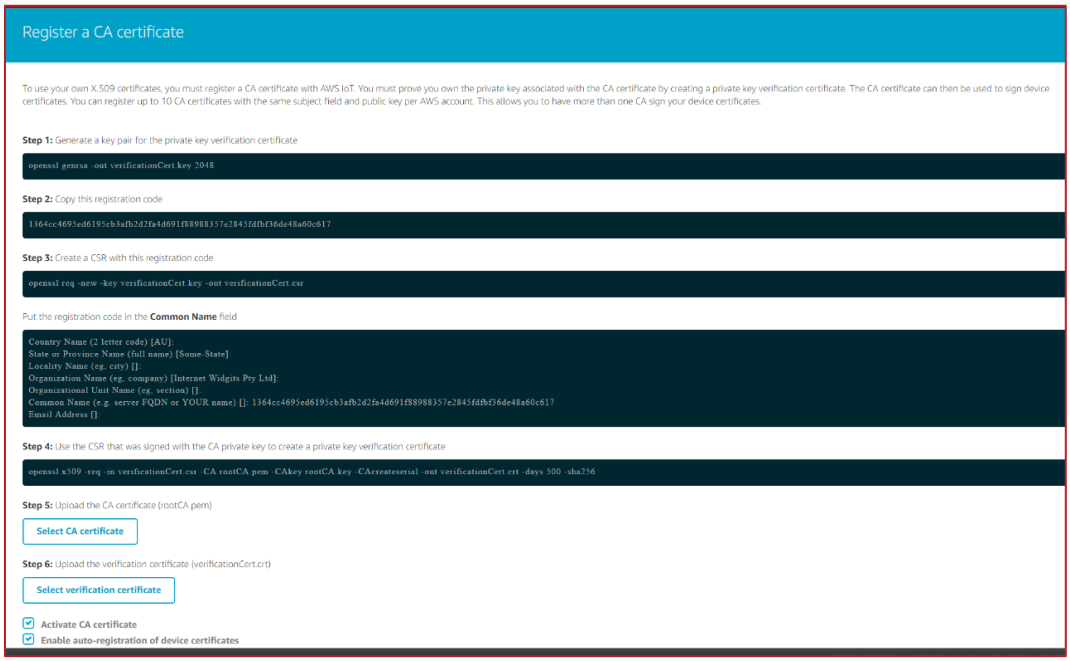
As per Amazon AWS,

to register your CA certificate, you must get a registration code from AWS IoT, sign a private key verification certificate with your CA certificate, and pass both your CA certificate and a private key verification certificate to the register-ca-certificate CLI command. The Common Name field in the private key verification certificate must be set to the registration code generated by the get-registration-code CLI command. A single registration code is generated per AWS account. You can use the register-ca-certificate command or the AWS IoT console to register CA certificates.

To register the CA certificate, follow the instructions in the screenshot below and check the two boxes in order to load and activate the CA.

In Step 4, "Use the CSR that was signed with the CA private key" use the AWS Custom CA Certificate and AWS Custom CA Key, i.e. "`-CA aws_custom_ca_cert.pem -Cakey aws_custom_ca_key.pem`".

In Step 5, "Select CA certificate", upload the AWS Custom CA Certificate, i.e. `aws_custom_ca_cert.pem`.



2. AWS Console: Create a Policy

Create a policy. **Appendix B: Policy** provides a sample global policy for the following actions:

- Connect
- Update the thing shadow
- Publish
- Subscribe

In the policy, configure the Amazon Resource Names (ARNs) region and AWS account-id for your account. In the example policy,

- `us-west-1` corresponds to the region

- `123456789012` corresponds to the ID of the AWS account that owns the resource

AWS information about Amazon Resource Names (ARNs):
https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html

3. *AWS Console: Create a Lambda Function*

The device will self-register the first time it connects to AWS. To configure the AWS actions during the Just in Time Registration, the user creates a Lambda function. **Appendix C: Lambda Function** is a python file ready to use as the Lambda function code that does the following:

- Get environment and event data
- Get device certificate information
- Create a thing (thing name is `thing_name = cn_string + ":" + subj_key_id`)
  - In this documentation `Device_Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c`
- Attach policy to device certificate
- Activate the certificate to allow connections from that device
- Attach certificate to thing

On the AWS console, select Lambda from the Services options. In the “Create function” page enter the required information:

- “Function name”: user’s choice. This example uses “`just_in_time`”
- “Runtime” option select “Python 3.8”
- “Permissions”, select the desired role, however, selecting “Create a new role with basic Lambda permissions” is sufficient

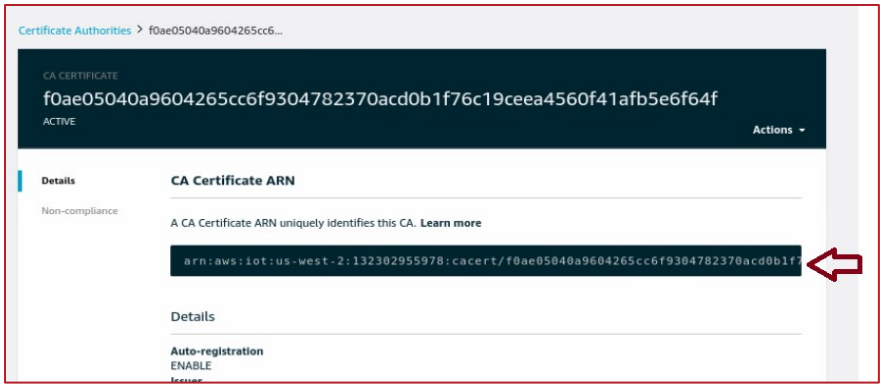
In “Configuration”:

- “Add trigger” select “AWS IoT”
  - Select Custom IoT rule
  - In Rule, select Create a new rule identifying the CA registered in **Register the CA to the AWS IoT**, e.g. JITR_CA
  - Enter description
  - Enter query statement for the CA, e.g.

```
SELECT * FROM
'$aws/events/certificates/registered/f0ae05040a9604265cc6f9304782370
acd0b1f76c19ceea4560f41afb5e6f64f'
```

Where `f0ae05040a9604265cc6f9304782370acd0b1f76c19ceea4560f41afb5e6f64f` is the CA Certificate ARN in the **IoT Core** CA Certificate page as illustrated in the figure, e.g.

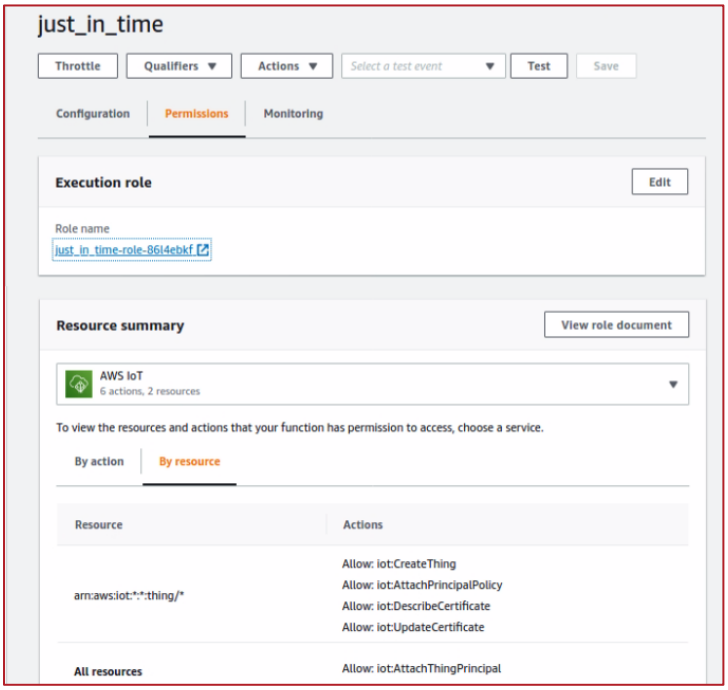
```
arn:aws:iot:us-west-
2:132302955978:cacert/f0ae05040a9604265cc6f9304782370acd0b1f76c19
ceea4560f41afb5e6f64f
```



  - Ensure the rule is enabled
- "Function code": use the python source code included in **Appendix C: Lambda Function**

In "Permissions", the role has permissions to the following, as shown in the image:

- iot.attach_principal_policy
- iot.create_thing
- iot.update_certificate
- iot.attach_thing_principal
- iot.list_thing_principals
- iot.describe_certificate



8.4.7. Linux Development Environment: Configure and Build the TLS AWS Example Application

This example application, `sli_dev`, demonstrates the capabilities of the EmSPARK Suite for supporting integration with cloud services such as Amazon Web Services. The application uses the CoreLockr TLS IO API, CoreLockr Crypto API, CoreLockr Secure Certificates API and keys and

certificates stored in the TEE Key Store and Certificate Store. This example connects to an AWS server and is intended for proof of concept only, not for commercial use.

The AWS embedded C SDK was modified to use the CoreLockr TLSIO API for TLS communication with the AWS servers. The AWS SDK was originally downloaded from here:

https://github.com/aws/aws-iot-device-sdk-embedded-C

Version 3.0.1 (SHA ID: d039f075e1cc2a2a7fc20edc6868f328d8d36b2f)

In addition to the full git repo, the Suite provides a patch against the 3.0.1 version, located in `corelockr/examples/AWS/aws-iot-c/`:

`0001-Sequitur-CoreLockr-TLSIO-AWS-Example.patch`

For additional information about the example, please see `corelockr/examples/AWS/README.txt`.

To prepare the application:

- Configure the specific MQTT host in the TLS AWS example application
- Build the TLS AWS example application

User Instructions: Building and Installing the Application

- Configure the Application
  - In the development environment, in `corelockr/examples/AWS/aws-iot-c/samples/linux/sli_dev/aws_iot_config.h` set `AWS_IOT_MQTT_HOST` to the AWS URL configured Endpoint
- Build the application executable
  - In `corelockr/examples/AWS/aws-iot-c/samples/linux/sli_dev` execute `make` to build the application binary, `sli_test`
- Transfer `sli_test` to the board to a directory of your choice.

8.4.8. Board: Execute the TLS AWS Example Application

The application scope includes:

- AWS connection
- AWS subscription and publishing events
- AWS shadow functionality
- AWS shadow interoperability

On the board, change to the directory where `sli_dev/sli_test` is located to execute it. The application will use the Device Certificate and the OEM Root CA in the TEE.

Usage for `sli_test`:

-h – Host Address
-p – Port
-l – Path to LED light control
-t – Test Type:
1 – CONNECTION
2 – SUBSCRIBE_PUBLISH
3 – SHADOW FUNCTIONAL
4 – SHADOW INTEROP

The application prints output on the device terminal. AWS events can be seen on multiple modules including AWS IoT Core and CloudWatch Logs.

The `-t` switch option is required. The `-h`, `-p` and `-l` switches are optional and intended to change the host, port and LED path configured in `aws_iot_config.h` for a given test type.

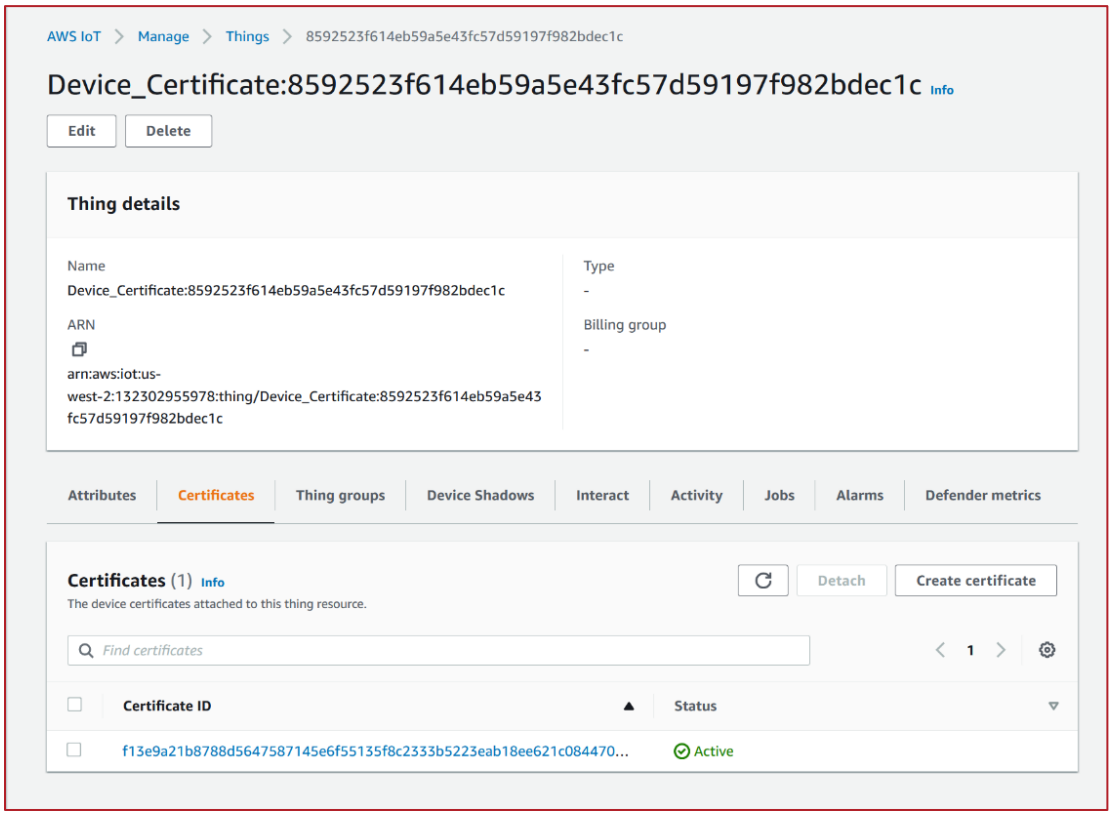
1. *AWS connection*

When the device connects to AWS for the first time, the Just in Time Registration is performed. Based on the Lambda Function configured in **AWS Console: Create a Lambda Function,** the Thing and Device Certificate are created in AWS.

On the device, the application prints the following messages:

```
./sli_test -t1
AWS IoT SDK Version 3.0.1- [Device ID: Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c]
Connecting...
Continuing...
Disconnecting
```

On AWS, the `Device_Certificate` device thing and certificate can be seen in IoT Core as shown in the figure:



2. AWS Subscription and Publishing Events

AWS Subscription

```
./sli_test -t2
AWS IoT SDK Version 3.0.1- [Device ID: Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c]
Connecting...
Continuing...
Subscribing to TOPIC [$aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/TestSubPub]...
{ "message" : "Publishing message on: QOS0" }...
{ "message" : "Publishing message on: QOS1" }...
****************Subscribe callback
Topic: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/TestSubPub{ "message" :
"Publishing message on: QOS0" }
Payload - { "message" : "Publishing message on: QOS0" }
Looping for subscriptions...
```

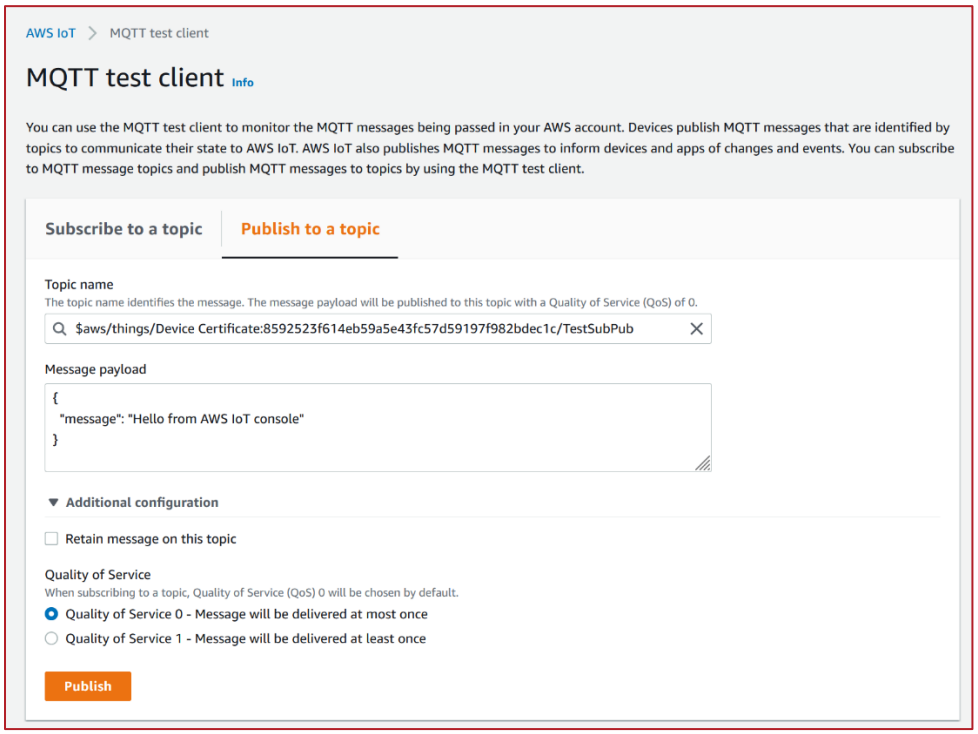
Publishing Messages

On the AWS console, select **Test.** On the **MQTT client** page, enter the topic for which the device is registered and select **Subscribe to topic**. In this example, the topic is:

```
$aws/things/${iot:ClientId}/TestSubPub
```

e.g.

```
$aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/TestSubPub
```



To send messages from the AWS console to the device, enter the topic on the **Publish** text box and select the **Publish to topic** button.

On the board, observe the messages from the AWS console

```
*****************Subscribe callback
Topic: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/TestSubPub{
  "message": "Hello from AWS IoT console"
}
Payload - {
  "message": "Hello from AWS IoT console"
}
```

3. AWS shadow functionality

```
./sli_test -t3

AWS IoT SDK Version 3.0.1- [Device ID: Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c]

Shadow Connect...
Subscribing to SHADOW...
Subscribe: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/shadow/update/get/accepted
Subscribe: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/shadow/update/get/rejected
Subscribe: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/shadow/update/delta
Subscribe: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/shadow/update/documents
Subscribe: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/shadow/update/accepted
Subscribe: $aws/things/Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c/shadow/update/rejected
Sleep..
Publishing...
Shadow published 'update' successfully

Yield...
Disconnecting
```

4. AWS shadow interoperability

The application uses the device shadow to retrieve and update a device LED state. The application controls `/sys/devices/platform/leds/leds/user/brightness`. The path to the LED can be modified in the `aws-iot-config.h` file.

During execution, on the device observe the green LED. To interact with the application, enter `1`, `0` or `x` when requested. The application prints messages like these:

```
./sli_test -t4

AWS IoT SDK Version 3.0.1- [Device ID: Device
Certificate:8592523f614eb59a5e43fc57d59197f982bdec1c]
[iot_tls_init] - Host is av98lkolvdy9i-ats.iot.us-west-2.amazonaws.com
Shadow Connect...
Setting TLS IO State to OPEN
Subscribing to SHADOW...
led_state from shadow : 1
Current LED state is: ON
Please enter the desired state [1,0] - or x to exit:
..0....
Setting DESIRED state to: OFF
Setting _REPORTED_ STATE to: OFF
Current LED state is: OFF
Please enter the desired state [1,0] - or x to exit:
```

On AWS IoT Core, the shadow state is updated.

9. CoreLockr™ Secure Storage API

The CoreLockr Secure Storage API protects data at rest. It enables saving data as encrypted persistent objects through the TEE. The persistent objects are encrypted with device specific keys.

The API functionality includes:

- Creation and writing of encrypted files
- Opening and reading from encrypted files
- Use of a password for these operations

In the Kit, `corelockr/corelockr_storage` contains:

- `lib`, `libclrf.a` library
- `include`, header files
- `ta`, `5840EE82-131E-4259-BB1F2A9286DA48A8.stp` associated TA
- `docs`, for library documentation see `~/docs/html/index`
- `README.txt`, general API information
- `COPYRIGHT`, copyright notice
- `example`, application

This document describes the sample application in **9.1 Secure Storage**.

9.1. Secure Storage Example

This example application illustrates features of the EmSPARK Suite, Secure Storage API, to protect data at rest (it does not protect data from the operating system). The application reads a secret from the console and writes the data as a persistent object through the TEE. Optionally a password can be provided. Then, the application prints the contents of the encrypted file. If a password was provided during the file creation, it is required to show the file contents.

9.1.1. Background

Software and Data Requirements

Secure Storage example application, `corelockr/corelockr_storage/example`

Building and Installing

Change to the `corelockr/corelockr_storage/example/` directory and execute `make` to build the application, `storage_example`. Transfer the executable to the board.

9.1.2. Executing the Example

To write/create an encrypted file (i.e. a persistent object in the TEE's store), the example application receives the following parameters:

```
storage_example w <filename> "<data>" [password (optional)]
```

where "`w`" is the application command for writing a file, `filename` is the name of the written persistent object in the TEE's store and "`data`" is the secret to be stored. `password` is an optional parameter that if provided becomes required for decryption of contents.

To show the encrypted contents:

```
storage_example r <filename> [password (optional)]
```

where "`r`" is the application command for reading, `filename` is the name used when the encrypted file was written and `password` is required if used during encryption.

Create and Write Encrypted Files

On the board, change to the directory where `storage_example` was transferred and execute

```
./storage_example w myfilename "This is a secret" mypassword
```

In this case, the file is written using a password. If writing the file succeeds, the application prints:

```
Secret written to: myfilename
```

The file name provided by the user (`myfilename`) is not visible in the filesystem, file names are only numbers and their contents are encrypted.

To create/write a file with no password, execute:

```
./storage_example w myfilename2 "This is a new secret"
```

Successful operation prints the output on the console, i.e.

```
Secret written to: myfilename2
```

Note that an attempt to overwrite without a password a file name that was originally created with a password will return an error. For example, executing:

```
./storage_example w myfilename "This is a secret"
```

returns:

```
Could not open myfilename: 0xffff0102
```

Decrypt Files and Print Data

To show the contents of the file encrypted with a password execute:

```
./storage_example r myfilename mypassword
```

If the decryption is successful, the application prints on the console:

```
Secret retrieved: This is a secret
```

In the previous step, the file was created with a password, therefore the password is required. If the correct password is not provided the corresponding error is printed on the console, e.g.

```
Could not open myfilename: 0xffff0102
```

For a complete list of return codes, see the Secure Storage API documentation.

To show the contents of the file encrypted without a password execute:

```
./storage_example r myfilename2
```

which returns

```
Secret retrieved: This is a new secret
```


APPENDIX A: SUPPORTED CRYPTOGRAPHIC OPERATIONS

The following table details the cryptographic operations supported with the EmSPARK Suite.

Certificate management
Store certificate in TEE
Delete certificate from TEE
Update certificate in TEE
Verify signature of a certificate
Add certificate to CRL

Key management
Create key in TEE keystore
Load key from a TEE keystore
Save key in a TEE keystore
Delete key in a TEE keystore

Cryptographic operations	
Hashing	MD5
	SHA1
	SHA224
	SHA256
	SHA384
	SHA512
Symmetric crypto functions	AES_ECB_NOPAD
	AES_CBC_NOPAD
	AES_OFB
	AES_CTR
	AES_CFB_128

	AES_XTS_NOPAD
	DES3_ECB_NOPAD
	DES3_CBC_NOPAD
	DES3_OFB
	DES3_CFB_128
MAC functions	AES_CMAC
	HMAC_MD5
	HMAC_SHA1
	HMAC_SHA224
	HMAC_SHA256
	HMAC_SHA384
	HMAC_SHA512
Authenticated encryption	AES_GCM
	AES_CCM
Asymmetric signature functions	RSASSA_PKCS1_V1_5_MD5
	RSASSA_PKCS1_V1_5_SHA1
	RSASSA_PKCS1_V1_5_SHA224
	RSASSA_PKCS1_V1_5_SHA256
	RSASSA_PKCS1_V1_5_SHA384
	RSASSA_PKCS1_V1_5_SHA512
	RSASSA_PKCS1_PSS_MGF1_SHA1
	RSASSA_PKCS1_PSS_MGF1_SHA224
	RSASSA_PKCS1_PSS_MGF1_SHA256
	RSASSA_PKCS1_PSS_MGF1_SHA384
	RSASSA_PKCS1_PSS_MGF1_SHA512
	DSA_SHA1
	DSA_SHA224
	DSA_SHA256
	ECDSA_P192
	ECDSA_P224
	ECDSA_P256

	ECDSA_P384
	ECDSA_P521
Asymmetric Encryption Functions	RSAES_PKCS1_V1_5
	RSAES_PKCS1_OAEP_MGF1_SHA1
	RSAES_PKCS1_OAEP_MGF1_SHA224
	RSAES_PKCS1_OAEP_MGF1_SHA256
	RSAES_PKCS1_OAEP_MGF1_SHA384
	RSAES_PKCS1_OAEP_MGF1_SHA512
	RSA_NOPAD
Key Derivation	DH_DERIVE_SHARED_SECRET
	ECDH_P192
	ECDH_P224
	ECDH_P256
	ECDH_P384
	ECDH_P521

APPENDIX B: POLICY

Sample policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:us-west-
1:123456789012:client/${iot:ClientId}"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:UpdateThingShadow",
        "iot:GetThingShadow"
      ],
      "Resource": "arn:aws:iot:us-west-
1:123456789012:topic/$aws/things/${iot:ClientId}/shadow/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": "arn:aws:iot:us-west-
1:123456789012:topic/$aws/things/${iot:ClientId}/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Subscribe"
      ],
      "Resource": [
        "arn:aws:iot:us-west-
1:123456789012:topicfilter/$aws/things/${iot:ClientId}/shadow/*",
        "arn:aws:iot:us-west-
1:123456789012:topicfilter/$aws/things/${iot:ClientId}/*"
      ]
    }
  ]
}
```

APPENDIX C: LAMBDA FUNCTION

```
import os
import base64
import binascii
import json
import boto3
import botocore

iot = boto3.client('iot')
client = boto3.client('iot-data')

ZT_THING_TYPE_NAME = 'sequitur-zero-touch-kit'

def lambda_handler(event, context):

    # Get evironment and event data
    region = os.environ['AWS_DEFAULT_REGION']
    account_id = event['awsAccountId']
    certificate_id = event['certificateId']

    print("Received event: " + json.dumps(event, indent=2))

    # Get device certificate information
    response = iot.describe_certificate(certificateId=certificate_id)
    certificate_arn = response['certificateDescription']['certificateArn']

    # Convert the device certificate from PEM to DER format
    pem_lines = response['certificateDescription']['certificatePem'].split('\n') #
split PEM into lines
    pem_lines = list(filter(None, pem_lines))      # Remove empty lines
    raw_pem = ''.join(pem_lines[1:-1])             # Remove PEM header and footer
and join base64 data
    cert_der = base64.standard_b64decode(raw_pem)  # Decode base64 (PEM) data into
DER certificate

    # Find the subjectKeyIdentifier (quicker than a full ASN.1 X.509 parser)
    subj_key_id_prefix = b'\x30\x1D\x06\x03\x55\x1D\x0E\x04\x16\x04\x14'
    subj_key_id_index = cert_der.index(subj_key_id_prefix) +
len(subj_key_id_prefix)
    subj_key_id =
binascii.b2a_hex(cert_der[subj_key_id_index:subj_key_id_index+20]).decode('ascii')
    print('Certificate Subject Key ID: {}'.format(subj_key_id))


    # Find CN in subject name.
    cn_id_prefix = b'\x06\x03\x55\x04\x03'
    cn_id_index = cert_der.index(cn_id_prefix) + len(cn_id_prefix) + 1
    cn_id_len = int.from_bytes(cert_der[cn_id_index:cn_id_index+1], "little") + 1
    issuer_cn_bytes = cert_der[cn_id_index+1:cn_id_index+cn_id_len]
    issuer_cn_string = issuer_cn_bytes.decode("utf-8")
    print('Certificate issuer  CN: {}'.format(issuer_cn_string))
```

```
    # 2nd call, index at num-bytes
    cn_id_index = cert_der.index(cn_id_prefix, cn_id_index) + len(cn_id_prefix) + 1
    cn_id_len = int.from_bytes(cert_der[cn_id_index:cn_id_index+1], "little") + 1
    cn_bytes = cert_der[cn_id_index+1:cn_id_index+cn_id_len]
    cn_string = cn_bytes.decode("utf-8")
    cn_string=cn_string.replace(' ','_')
    print('Certificate subject CN: {}'.format(cn_string))

    # extract Serial Number
    sn_id_prefix = b'\xa0\x03\x02\x01\x02\x02'
    sn_id_length_index = cert_der.index(sn_id_prefix) + len(sn_id_prefix)
    sn_id_length =
int.from_bytes(cert_der[sn_id_length_index:sn_id_length_index+1], "little")
    sn_id_bytes = cert_der[sn_id_length_index+1:sn_id_length_index+sn_id_length+1]
    serial_number_string = binascii.b2a_hex(sn_id_bytes).decode('ascii')
    print('Serial number: {}'.format(serial_number_string))


    # Thing name and MQTT client ID will be the subject key ID
    thing_name = cn_string + ":" + subj_key_id
    client_id = thing_name

    thing_attributes = {
        'attributes': {
            'serial_number' : serial_number_string,
            'initialized' : '0',
            'subject_cn' : cn_string
        }
    }


    # Create a thing (no error if it already exists)
    response = iot.create_thing(
        thingName=thing_name,
        attributePayload=thing_attributes)


    # Attach policy to device certificate. Certificates must have a policy
    # before they can be activated.

    Iot.attach_principal_policy(
        policyName='GlobalDevicePolicy',
        principal=certificate_arn)

    # Activate the certificate to allow connections from that device
    response = iot.update_certificate(
        certificateId=certificate_id,
        newStatus='ACTIVE')

    # Attach certificate to thing
    response = iot.attach_thing_principal(
        thingName=thing_name,
        principal=certificate_arn)
```

CHANGE HISTORY

DATE	VERSION	RESPONSIBLE	DESCRIPTION
November 15, 2019	1.0	Julia Narvaez	Produced document for release.
June 2, 2020	1.1	Julia Narvaez	Added Opaque Keys in CoreLockr Crypto API section and Key Utility example description in CoreLockr Payload Verification API section.
July 9, 2020	1.2	Julia Narvaez	Added CoreLockr TLS IO API and updated AWS example.
August 4, 2020	1.3	Julia Narvaez	Updated AWS example.
August 11, 2020	1.4	Julia Narvaez	Updated numbering of some examples.
September 24, 2020	1.5	Julia Narvaez	Added EmSPARK architecture graphic.
November 4, 2020	2.0	Julia Narvaez	Added Opaque Objects in CoreLockr Crypto API section.
November 30, 2020	2.1	Julia Narvaez	Simplified AWS example process
September 21, 2021	3.0	Julia Narvaez	Expanded library descriptions and EmPOWER enabled updates.
March 3, 2022	3.1	Julia Narvaez	In CoreLockr Secure Certificates section, added user privilege requirements to update provisioned certificates.
March 18, 2022	3.2	Julia Narvaez	Updated introduction.
August 2, 2022	3.3	Julia Narvaez	Updated CoreLockr Secure Certificates section with new procedure to update provisioned certificates, expanded Crypto API Key Store and Provisioned Keys content and clarified Opaque Objects creation process.
August 31, 2022	3.4	Julia Narvaez	Updated description of Secure Storage API.