Getting Started with Arrow Shield96 Trusted Board and EmSPARK Security Suite

Getting Started Guide

Date September 15, 2020 | Version 1.0

TABLE OF CONTENTS

Getting Started Guide 1
1. Introduction 3
1.1. Prerequisites 3
2. Installation Procedure 4
2.1. Install Filesystem on the Board 4
2.2. Download the CoreLockr Kit and Extract the Contents in Linux Development Environment 6
3. Connecting to AWS IoT Core 6
3.1. Introduction 6
3.2. Linux Development Environment: Build Auxiliary Example Applications 9
3.3. Shield96: Customize Device Certificate and OEM Root Certificate in the TEE Certificate Store 10
3.4. AWS Console: Configure User's Account for AWS TLS Example 12
3.5. Linux Development Environment: Configure and Build the TLS AWS Example Application 18
3.6. Shield96: Execute the TLS AWS Example Application 19
Appendix A: Policy 25
Appendix B: Lambda Function 26

1. INTRODUCTION

The **Getting Started Guide** provides an overview of the steps to use the Shield96 Trusted Platform and the EmSPARK Security Suite package contents to connect and communicate with AWS IoT Core for IoT related tasks. The following tasks will be performed:

- Initial configuration and loading of the filesystem
- Download the CoreLockr Kit (SDK included with EmSPARK Security Suite Kit) with toolchain and examples and install it in a Linux development environment (not on the Shield96)
- Configure your AWS account for Just in Time Provisioning (JITP) with the Shield96 Trustboard
- Execute the AWS IoT Core Examples
  - JITP
  - Connect to AWS IoT Core
  - Publish to User defined Topic
  - Subscribe to User defined Topic
  - Interact with Device Shadow to control LED on Shield96

Short product description

The Shield96 Trusted Platform is a fused, locked SBC board preloaded with Sequitur Labs EmSPARK Security Suite. The board implements secure boot and a minimum, standard version of Linux preloaded to the QSPI present on the board. The board does not include a filesystem because it does not ship with a SD card. The Out of the Box experience includes the procedure to automatically download the Linux filesystem from the cloud. Once an SD card is inserted and the board is connected to the Internet via Ethernet the board automatically connects to a Sandbox Platform provided by Sequitur Labs which downloads securely to the board the latest firmware and the filesystem. The procedure includes the registration of the board to AWS IoT Core since IAWS IoT Core service is used for the firmware management. The Sandbox is built as a service on AWS. The Sandbox has further built-in functionality to provide various customers firmware update for the device lifecycle.

The EMSPARK Security Suite for the Shield96 Trusted Board provides secure boot, a preloaded TEE (Trusted Execution Environment) a number of Trusted Applications (TAs) and an SDK that includes the CoreLockr APIs abstracting all TA secure applications in easy to use C APIs in Linux.

Out of the box the board provides a secure enclave through the TrustZone\TEE and an immutable key pair that generates a CSR for the creation of a unique Device Certificates which acts as the device ID for communication with cloud applications – in this case with AWS IoT Core. The CoreLockr SDK provides APIs to generate more CSRs and thus to create other device IDs. The private keys cannot be extracted from the secure enclave.

1.1. Prerequisites

The following hardware and software are required to use the EmSPARK Suite:

- To install the filesystem on the Shield96
  - Shield96 Trusted Platform: Jumper for J3 must be in place.
  - SD Card to install the filesystem (U1 and U3 cards should NOT be used due to HW Limitations)
  - Network connection on the board
  - Host computer connected to the board
- To develop and build applications using the CoreLockr APIs

- CoreLockr Kit downloaded from https://github.com/ArrowElectronics/hd96/releases/download/20200721_shield96_emspark_/arrow_corelockr_package_2020-07-21_1.tar.gz - see section **2.2**
- Linux development environment to extract the CoreLockr Kit

Note: The CoreLockr kit contains all CoreLockr examples including connecting and communicating with AWS IoT Core.

2. INSTALLATION PROCEDURE

This section describes steps to install the filesystem on the board and to use the CoreLockr APIs in a Linux development environment.

2.1. Install Filesystem on the Board

To install the filesystem on the board, the process consists of these steps detailed in the following sections:

- Insert a blank SD Card on the board (No partitions defined)
- Connect the board to the host computer and start a serial console
- Power up the board
- Ensure the board has connectivity to a network

The board will register itself with AWS, configure the SD Card, and install the base filesystem when network connection is available and an SD Card is present.

2.1.1. Insert SD Card on the Board

The SD Card must be 4GB or larger. Insert a blank SD Card on the board. The installation process will partition, format and install the root filesystem on the card.

Important Notes:

- Inserting an SD Card that has a filesystem able boot the board will prevent the installation process.
- U1 and U3 cards should NOT be used due to HW Limitations.

2.1.2. Connect the Board and Start the Console

To start the serial console which is the TEE console where occasionally the Secure World prints output messages:

- Connect a micro USB cable to J10 (debug) micro USB port on the board
- Connect the host machine to the serial port
  - 115200 bps
  - No parity
  - 8 bits
  - 1 stop bit
  - No flow control
- Connect a micro USB cable to the PC/power micro USB port on the board, if you would like to power the board separate from the console connection.

2.1.3. Power up the Board

When the board boots for the first time will do the following:

- Boot to the Linux RAM FS
- Check for network connection
- Check for SD Card

The initial setup checks for network connection and SD Card. If the checks succeed, then the board will register itself to AWS and retrieve the Root filesystem.

This process is automated but can be observed from the console connection.

The device is now able to run the example applications and modify the root filesystem.

The serial terminal will print output such as the following:

```
Checking: mmcblk1
Getparts:  /dev/mmcblk1 status:  2
Create partitions
...
eth0 at:  192.168.x.xxx
...
Enroll device at AWS ...
...
Enroll customer/device at AWS IoT ...
...
Retrieve device rootfs information from AWS ...
...
Payload server:  screechowl.seqlabs.com
Service port:    2270
Root filesystem: Shield96RootFS.tar.gz
...
Extract rootfs ...
```

Finally:

```
Welcome to Sequitur Labs CoreTEE
root@seqlabs_coretee:~#
```

2.1.4. Secure Boot Mode - Starting and Using the System

To start using the system on the board, start the console. After the board starts up:

- Access `username:password = root:root`.
- The board is configured to acquire an IP address using DHCP

The board is ready for your configuration:

- Required configuration:
  - The date on the board must be current for certificate management. When the board is used offline, the date must be configured. When the board is configured for remote access, verify that the date is current.
- Optional configuration:

- Configure additional user(s). In addition to `root`, users in the `coretee` group have access to the TEE clients and can execute applications using the CoreLockr APIs. If users are created to execute the example applications, then create the `coretee` group if not already created and add users to this group.
- Configure the board for remote access. The board is configured to acquire an IP address using DHCP. SSH is set up in the filesystem.

The board set up is complete and ready to execute operations in the TEE and execute applications using the EmSPARK Suite CoreLockr APIs. The following sections explain the process for TLS mutual authentication and session establishment with Amazon AWS IOT.

2.2. Download the CoreLockr Kit and Extract the Contents in Linux Development Environment

Download and extract the contents of the CoreLockr Kit in Linux development environment (not on the Shield96).

The CoreLockr Kit downloaded from https://github.com/ArrowElectronics/hd96/releases/download/20200721_shield96_emspark_/arrow_corelockr_package_2020-07-21_1.tar.gz contains everything needed to build applications using the CoreLockr APIs:

- APIs and Normal World Assets
- CoreLockr APIs (C libraries)
- OpenSSL Crypto Engine
- Code Examples
- Toolchain and Client API

If building in a 64-bit Linux development environment, install libraries to enable 32-bit support:

- Platform agnostic library list to install the necessary libraries for the toolchain to run
  - 32bit glibc
  - 32bit libstdc++
  - 32bit zlib (compression library)
- Debian based systems
  - dpkg --add-architecture i386
  - apt-get update
  - apt-get install libc6:i386 libstdc++6:i386 zlib1g:i386
- Red Hat based systems
  - dnf install glibc.i686 libstdc++.i686 zlib.i686

3. CONNECTING TO AWS IOT CORE

3.1. Introduction

AWS allows devices to use X.509 certificates signed and issued with 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 a 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 certificates managed in the TEE:

- EmSPARK Defined OEM CA: replace with a user defined OEM Root CA certificate
- Device Certificate signed with the EmSPARK Defined OEM Key: replace with a Device Certificate signed with the user's OEM Root Key

The 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 Device Certificate and update the OEM Root CA and Device Certificate in the TEE. The Suite includes a script (`run_aws_cert_generation.sh`) and example applications to accomplish that. 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:

- **AWS IoT Root Certificate,** the AWS cloud certificate is preloaded in the TEE (Secure enclave), `CLRSC_CLOUD_CERT`.
- **OEM Root Key,** this private key is generated by the user. It is used to sign the Device Certificate and to complete the AWS Custom CA Certificate registration process with AWS IoT.
- **OEM Root CA**, the user creates this certificate authority and using the provided tools saves it in the TEE, `CLRSC_OEM_CERT`. This certificate replaces the EmSPARK Defined OEM CA provisioned on the device.
- **Device Key,** the device private and public keys are generated during provisioning and stored in the TEE. Using the CoreLockr APIs, applications can access the Device Private and Public Key from the rich execution environment (e.g. Linux). The Device Private Key can be accessed for operations, even though applications in Linux do not have visibility of the private key attributes. The Device Public

Key is the key that the rich execution environment (Linux) can see. The Device Key is used to sign the Device Certificate Signing Request.

- **Device Certificate,** using the provided script and example applications, the user creates this certificate signed with the user's OEM Root Key and saves it in the TEE, `CLRSC_DEVICE_CERT`. The execution of the TLS AWS example will register the Device Certificate with the AWS IoT cloud.

The AWS example preparation includes steps in a Linux development environment, on the Shield96 and in AWS console.

The following is an outline of the steps described in the next sections of this document to build the AWS example and successfully register to AWS IoT Core using JITR:

1. **On the Linux development environment**, build auxiliary example applications included in the CoreLockr Kit to manage and store certificates in the TEE, please see **3.2**
    - Build the CoreLockr Secure Certificate application called from the `run_aws_cert_generation.sh` script to execute certificate management commands in the TEE.
    - Build the Manifest Retrieval application called from the `run_aws_cert_generation.sh` script to store the user's new certificates in non-volatile-memory in the TEE.
2. **On the Shield96**, customize the OEM Root Certificate and Device Certificate in the TEE Certificate Store, please see **3.3**
    - Execute the `run_aws_cert_generation.sh` script which does the following
        - Generates or takes input of user's preexistent OEM Root Key and OEM Root CA
        - Extracts the device CSR from the TEE
        - Generates the Device Certificate
        - Uploads the updated OEM Root CA and Device Certificate in the TEE
        - Stores the updated certificates in non-volatile-memory

    The script uses the CoreLockr Secure Certificate example application and the Manifest Retrieval application.

3. **On the AWS Console**, configure user's account for AWS TLS example, see **3.4**
    - 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
4. **On the Linux development environment**, configure and build the TLS AWS example application, see **3.5**
    - Configure the specific MQTT host in the TLS AWS example application
    - Build the TLS AWS example application
5. **On the Shield96**, execute the TLS AWS example application, see **3.6**
    - AWS connection
    - AWS subscription and publishing events
    - AWS shadow functionality
    - AWS shadow interoperability

Note: 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, generation of the Device Certificate signed with the OEM Root Key that would be produced on a server is instead produced and signed on the device.

3.2. Linux Development Environment: Build Auxiliary Example Applications

Build example applications that use the CoreLockr APIs to manage and store certificates in the TEE. The `run_aws_cert_generation.sh` needs two auxiliary example applications, one using the CoreLockr Secure Certificates API and one using the Manifest Retrieval API, explained in the following sections.

3.2.1. Build the CoreLockr Secure Certificates Example Application

The `run_aws_cert_generation.sh` calls this application binary to execute certificate management commands in the TEE. This 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 to be used as Certificate Authorities (CAs). The application also illustrates how to update in the TEE the Device Certificate and OEM Root CA.

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 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.

The application uses the OpenSSL library to load certificates from the local filesystem, convert the certificates from X509 structures to a DER encoded byte string and pass the certificates to the CLRSC API. Among other operations, the example application performs the following:

- Update the OEM Root CA certificate in TEE Certificate Store with a certificate from the local filesystem
- Extract the OEM certificate from the TEE Certificate Store and write it in the filesystem as PEM encoded file

For description of the application complete functionality, please see the CORELOCKR_LIBRARIES_GUIDE.pdf included with the EmSPARK Suite and/or `corelockr_cert/examples/README.txt`.

The following sections provide software and data requirements specific for the TLS Mutual Authentication and Session Establishment with Amazon AWS example execution, and instructions for application building and installation.

Software and Data Requirements

- CoreLockr Secure Certificates example application, `corelockr/corelockr_cert/examples`
- Key provided with the example, `corelockr/corelockr_cert/examples/certs/clrsc_example_root_key.key`.

  This key should be renamed as `clrsc_signing_key.key` and corresponds to the EmSPARK Defined OEM Private Key and is used to sign the certificate management commands to update the OEM and device certificates.

  The private key used for signing certificate management commands usually is not found on a device. To avoid the configuration of an external system that sends signed commands to the board and expedite the example execution, this provided private key is used on the board.

The application uses the Root Private Key `clrsc_signing_key.key` to sign the following certificate management commands:

- Update OEM Root CA certificate in the TEE Certificate Store
- Update Device Certificate in the TEE Certificate Store

Note 1: The TEE Certificate Management requires each command to be signed with the OEM Private Root key (i.e. commands sent from a remote secure server). The preloaded OEM Root Certificate is used to verify the signature of the commands. This way the OEM has ownership of the TEE Certificate Store.

User Instructions: Building and Installing the Application

Change to `corelockr/corelockr_cert/examples` and execute `make` to build the executable `clrsc_ossl_ex`. Transfer the executable to the board to a directory of your choice where the `run_aws_cert_generation.sh` is located. Transfer to the same directory the `~/corelockr_cert/examples/certs/clrsc_example_root_key.key` which is the sample OEM Root Key associated with the EmSPARK Defined OEM Root CA certificate and rename it as `clrsc_signing_key.key`.

3.2.2. Build the Manifest Retrieval Example Application

The `run_aws_cert_generation.sh` will call this application binary. The application uses the `libclr_manifest.a` library to communicate with CoreTEE and retrieve an encrypted file that contains the user's OEM Root CA and Device Certificate updated in the TEE. The script will store the encrypted certificate file in non-volatile-memory.

User Instructions: Build and Install the Application

Change to `corelockr/utilities/manifest_retrieval` and execute `make` to build the `clr_manifest_example` executable. Transfer `clr_manifest_example` to the board to the same directory where `clrsc_ossl_ex` is located.

3.3. Shield96: Customize Device Certificate and OEM Root Certificate in the TEE Certificate Store

In this step, the user will generate a custom Device Certificate and OEM Root Certificate which will replace the EmSPARK Defined certificates. This section provides first an overview and then detailed instructions.

NOTE: On development boards, the custom Device Certificate and OEM Root Certificate will be updated in volatile memory only. After a board reboot, the EmSPARK Defined certificates preloaded in the TEE Certificate Store will replace the custom certificates.

Overview

In the Suite, `~/corelockr/examples/AWS/tools/` contains the `run_aws_cert_generation.sh` script along with a sample `~/conf/ca.conf` file, auxiliary text files and the `gen_csr` binary. The script uses the CoreLockr Secure Certificate `clrsc_ossl_ex` and the Manifest Retrieval

`clr_manifest_example` executables built in the previous steps. The script is designed to take an OEM CA, OEM Key and a CA configuration file (for signing the device CSR) to do the following:

1. Generate a new CA cert (OEM Root CA) and key (OEM Root Key) if user does not provide them
2. Use the CoreLockr Secure Certificates example to update the EmSPARK Defined OEM Root CA in the TEE with the new OEM Root CA
3. Replace the `clrsc_signing_key` signing the certificate management commands with the user's OEM Root Key
4. Generate the device CSR
5. Create a new Device Certificate signed with the user's signing key
6. Replace the Device Certificate in the TEE with the new one
7. Retrieve from the TEE an encrypted file which contains the user's OEM Root CA and new Device Certificate and store it in non-volatile-memory so that the user's certificates will be persistent after device reboots

The user can provide the OEM Root Key and OEM Root CA as inputs. If not available, the script will generate them.

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

Before executing the script:

- Customize `~/conf/ca.conf` or produce a CA configuration file used for signing the Device CSR. **Note that the example application requires that key and cert be ECDSA P256. Those parameters should not be changed to avoid errors during the application execution.**
- Generate the custom OEM Root Key and OEM Root CA if desired. If not available, the script will generate them.
- Transfer the script, customized `conf/ca.conf` directory, `gen_csr` binary, serial.txt and index.txt files to the board to the same location where CoreLockr Secure Certificates example application `~/corelockr_cert/examples/clrsc_ossl_ex` binary and `clrsc_signing_key.key` are located.

Execute the script which takes the following parameters:

- `-f` <conf file> : default "conf/ca.conf"
- `-c` <CA file> : default ""
- `-k` <Key file>: default ""

If in the script either the CA or the key are empty (default), then they are generated:

- The new CA file is: `aws_custom_ca_cert.pem`
- The corresponding key file is: `aws_custom_ca_key.pem`

If using preexistent OEM Root Key and OEM Root Certificate, then update the script and `conf/ca.conf` and provide the corresponding names when executing the script, e.g.

In `run_aws_cert_generation.sh`:

```
oem_cert_str="aws_custom_ca_cert.pem"
oem_key_str=" aws_custom_ca_key.pem"
```

Execute the script:

```
./run_aws_cert_generation.sh -k aws_custom_ca_key.pem -c
aws_custom_ca_cert.pem
```

When executing the script, 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.

The OEM Root CA file must be registered with AWS (please see **3.4.1 Register the CA to the AWS IoT**). The following section assumes the OEM Root CA filename is `aws_custom_ca_cert.pem`.

Behind the scenes, during the script execution, the verification of certificate management commands that alter the TEE Certificate Store is as follows:

- The `clrsc_ossl_ex` example expects that the signing key for certificate management commands be named `clrsc_signing_key.key`.
- Initially the signing key (`clrsc_signing_key.key`) is the renamed `clrsc_example_root_key.key` included in `~/corelockr_cert/examples/certs`.
- After the EmSPARK Defined OEM Root CA certificate is replaced in the TEE with the user's OEM Root CA (e.g. `aws_custom_ca_cert.pem`), the script creates a symbolic link to the new OEM Root Key (e.g. `aws_custom_ca_key.pem`) which becomes the signing key of subsequent certificate management commands.
- The new OEM Root Key signs the command that updates the Device Certificate
- The `clrsc_ossl_ex` example verifies the new OEM Root Key signature in the TEE using the public key extracted from the OEM Root CA certificate.

The key names and locations could be modified in the initial lines of the script for a different location but the `clrsc_ossl_ex` example looks for `clrsc_signing_key.key` so it is not recommended to change them.

3.4. 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

3.4.1. Register the CA to the AWS IoT

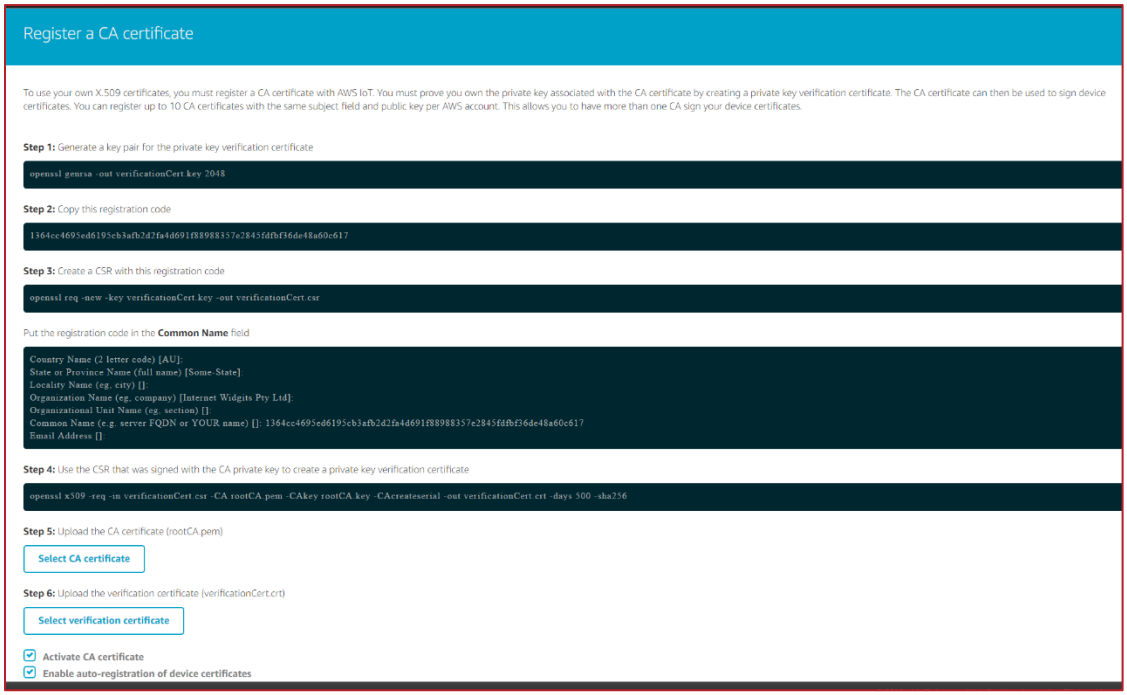
As per Amazon AWS,

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`.



3.4.2. AWS Console: Create a Policy

Create a policy. **Appendix A: 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

3.4.3. 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 A: 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 B: 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
- 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 "Phyton 3.8"
- "Permissions", select the desired role, however, selecting "Create a new role with basic Lambda permissions" is sufficient

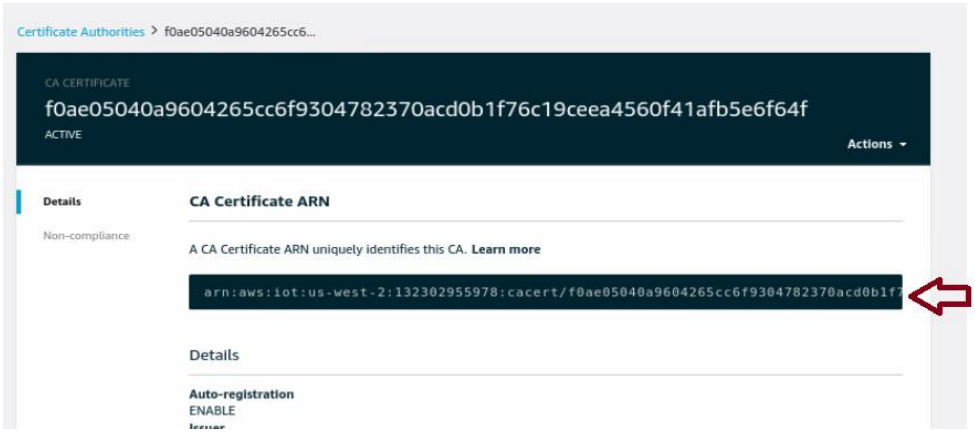
In "Configuration":

- "Add trigger" select "AWS IoT"
  - Select Custom IoT rule
  - In Rule, select Create a new rule identifying the CA registered in **3.4.1 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 A: Policy

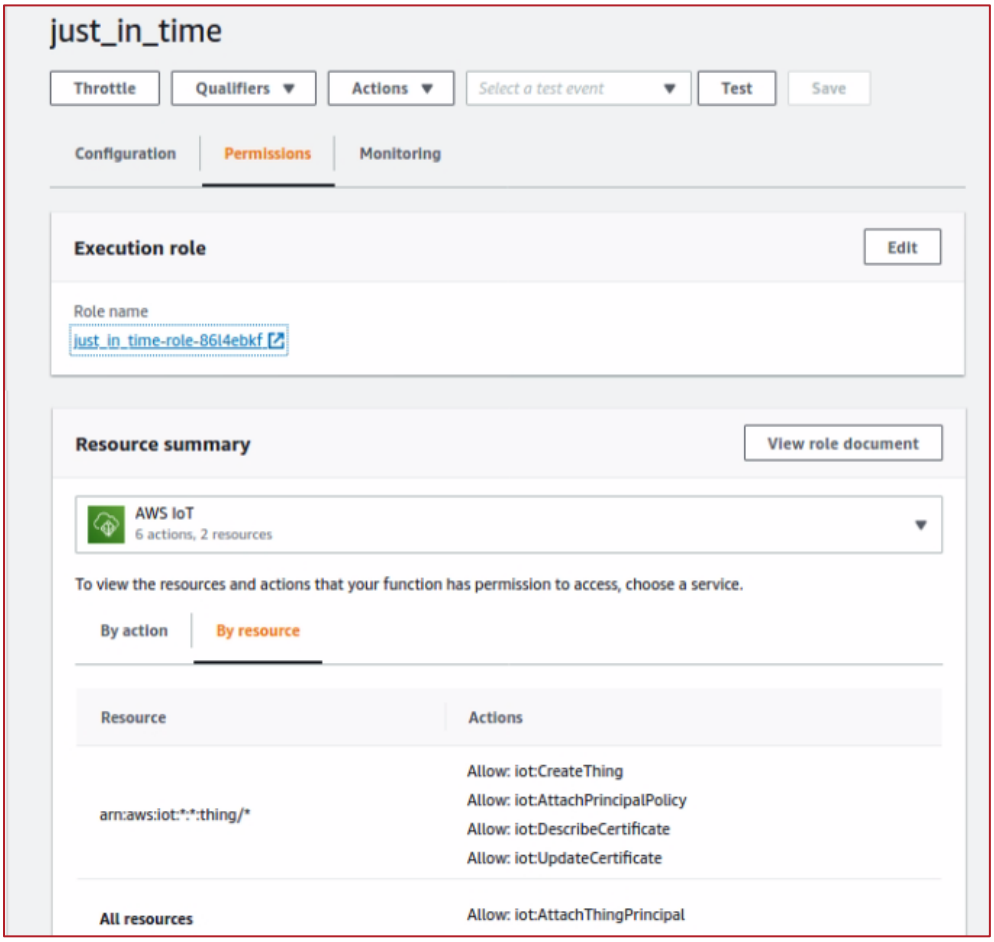
Sample policy:

```
{
  "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 B: 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



3.5. 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, in this case Amazon Web Services. The application uses the CoreLockr TLS IO API, CoreLockr Crypto API and the 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 provided, the Suite provides a patch against the 3.0.1 version:

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

Please see `corelockr/examples/AWS/README.txt` for additional information about the example.

References

- CoreLockr TLS IO documentation, corelockr/corelockr_tlsio/docs/html/index.html

Software and Data Requirements

- TLS AWS example application, `corelockr/examples/AWS`

User Instructions: Configure the Application

In `aws_iot_config.h` included in `corelockr/examples/AWS/aws-iot-c/samples/linux/sli_dev`, the `AWS_IOT_MQTT_HOST` must be set to the AWS URL configured Endpoint

User Instructions: Build and Install the Application

In Linux development environment, change to `corelockr/examples/AWS/aws-iot-c/samples/linux/sli_dev/`.

Executable `make` to build the application binary, `sli_test`.

Transfer the sli_test executable to the board to a directory of your choice.

3.6. Shield96: Execute the TLS AWS Example Application

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, e.g.

```
./sli_test -p 8883 -l /sys/devices/platform/leds/leds/zigbee/brightness -t 4
```

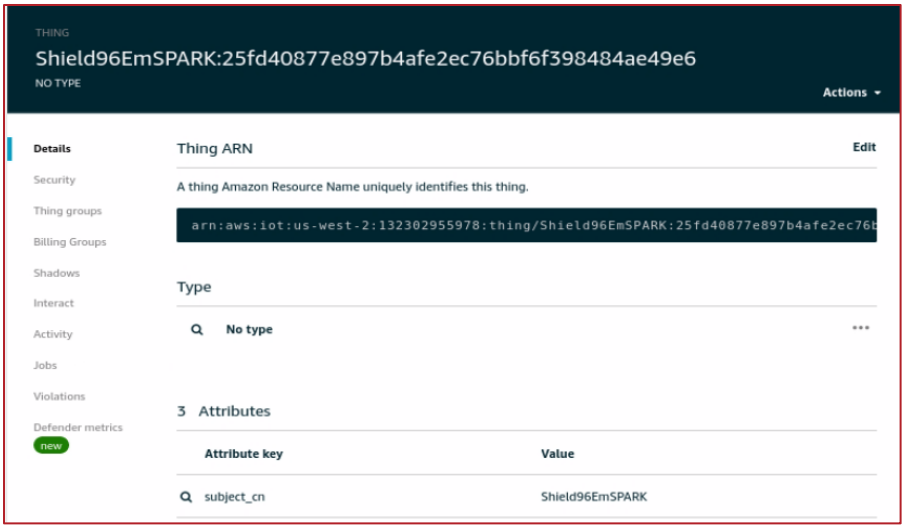
3.6.1. AWS connection

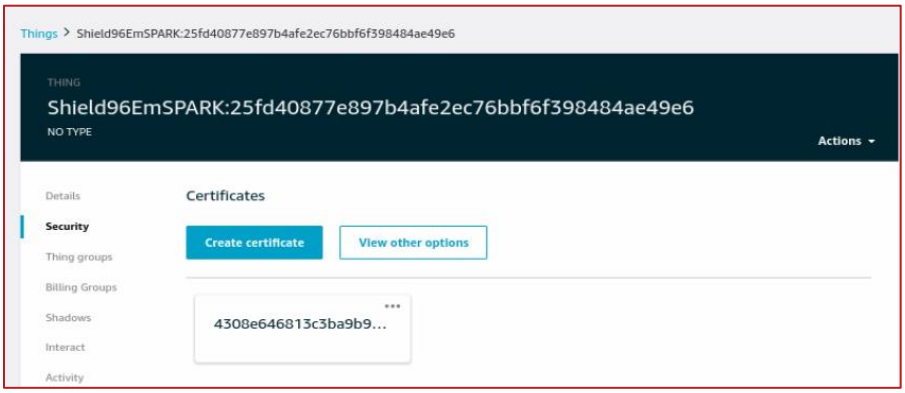
When the device connects to AWS for the first time, the Just in Time Registration is performed. Based on the Lambda Function configured in **3.4.3 AWS Console: Create a Lambda Function,** the Thing and Device Certificate are created in AWS.

On the device, the application prints the following messages:

```
./sli_test -t1
AWS IoT SDK Version 3.0.1- [Device ID:
Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6]

[iot_tls_init] - Host is av98lkolvdy9i-ats.iot.us-west-2.amazonaws.com
Connecting...
Setting TLS IO State to OPEN
Setting TLS IO State to OPEN
Continuing...
Disconnecting
```

On AWS, the `Shield96EmSPARK` device thing and certificate can be seen in IoT Core as shown in the figures:





3.6.2. AWS Subscription and Publishing Events

AWS Subscription

```
./sli_test -t2
AWS IoT SDK Version 3.0.1- [Device ID:
Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6]

[iot_tls_init] - Host is av98lkolvdy9i-ats.iot.us-west-2.amazonaws.com
Connecting...
Setting TLS IO State to OPEN
Continuing...

Subscribing to TOPIC
[$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/TestSu
bPub]...
{ "message" : "Publishing message on: QOS0" }...
{ "message" : "Publishing message on: QOS1" }...
```

```
**********Subscribe callback
Topic:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/TestSub
Pub{ "message" : "Publishing message on: QOS0" }
Payload - { "message" : "Publishing message on: QOS0" }
```

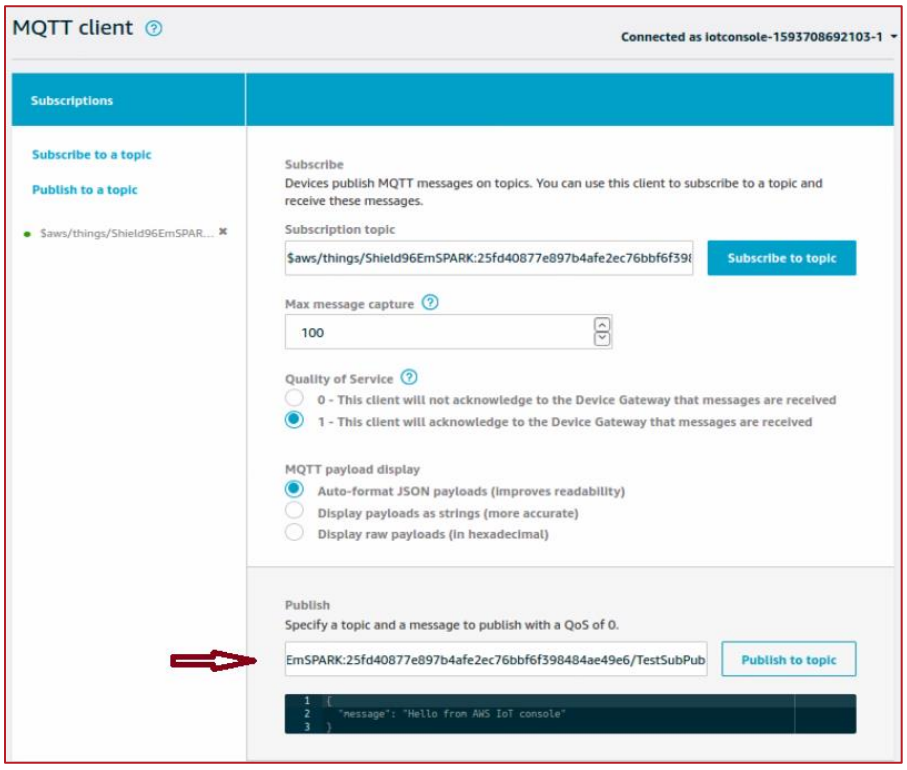
Publishing Messages

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/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/TestSub
Pub
```



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

```
Topic:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/T
estSubPub{
  "message": "Hello from AWS IoT console"
}
Payload - {
"message": "Hello from AWS IoT console"
```

3.6.3. AWS shadow functionality

```
./sli_test -t3
AWS IoT SDK Version 3.0.1- [Device ID:
Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6]

[iot_tls_init] - Host is av98lkolvdy9i-ats.iot.us-west-2.amazonaws.com
Shadow Connect...
Setting TLS IO State to OPEN
Subscribing to SHADOW...
Subscribe:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/shadow/
update/get/accepted
Subscribe:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/shadow/
update/get/rejected
Subscribe:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/shadow/
update/delta
Subscribe:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/shadow/
update/documents
Subscribe:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/shadow/
update/accepted
Subscribe:
$aws/things/Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6/shadow/
update/rejected
Sleep..
Publishing...
Shadow published 'update' successfully

Yield...
Disconnecting
```

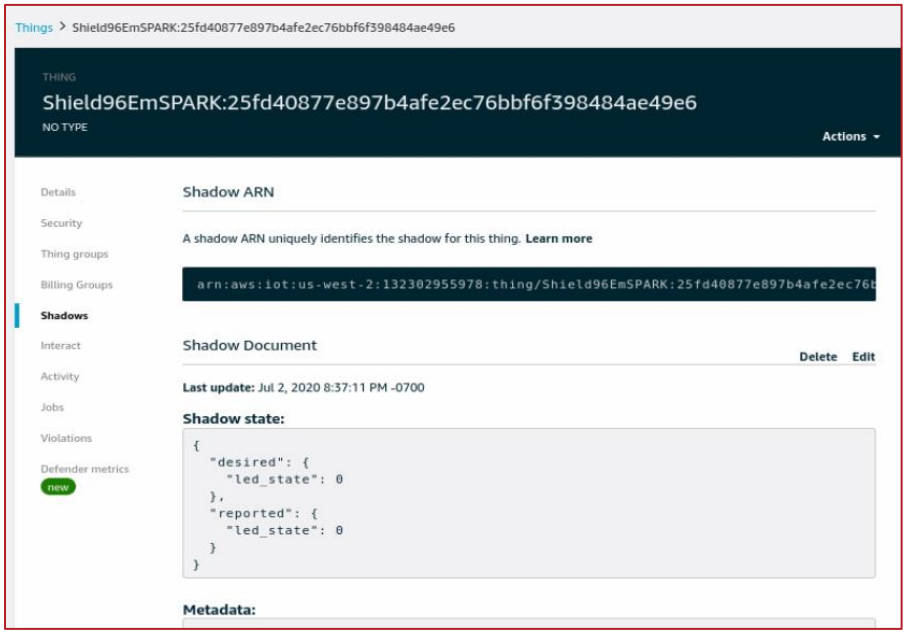
3.6.4. AWS shadow interoperability

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:
Shield96EmSPARK:25fd40877e897b4afe2ec76bbf6f398484ae49e6]

[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, as shown in the figure.



APPENDIX A: 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 B: 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")
    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
June 18, 2020	1.0	Julia Narvaez	Produced document for release.