

Developer Guide

Security Starter Kit with i.MX 8X and OPTIGA™ TPM 2.0

Date: December 3, 2020 | Version 1.4

FINAL



The Solutions People



CONTENTS

1	INTRODUCTION	6
1.1	Purpose of the Document	6
1.2	Intended Audience	6
1.3	Prerequisites	6
1.4	Scope of Detailed Design	6
2	ENVIRONMENT SETUP	7
3	HARDWARE SETUP	8
3.1	Hardware set up – Security Starter Kit withi. MK 8X and OPTI GA™ TPM M2.0	8
3.1.1	Hardware connection between i. MK 8X (A_ML board) and OPTI GA™ TPM M2.0	8
3.1.2	Powering up the Board	8
3.1.3	Open boards terminal - console (Mriti.com) on Linux PC	9
4	SOFTWARE SETUP	10
4.1	I. MK_8X-SSK and OPTI GA™ TPM M2.0 Yocto Environment Set up	10
4.1.1	Pre-requisite	10
4.1.2	Steps to build the BSP for the I. MK_8X-SSK and OPTI GA™ TPM 2.0	10
4.2	Keys and certificates information	12
4.3	OPTI GA™ TPM M2.0 Set up Script	12
4.4	TLS Mutual Authentication and Session Establishment Using H/w Security of TPM with Amazon AWS IoT	16
4.4.1	Linux Environment: Generate the required keys and certificate	18
4.4.1.1	AWS Custom Gateway CA Creation	18
4.4.1.2	Registering Your CA Certificate	19
4.4.2	Linux Environment: Secure Device Certificate and Private Key Gen.	21
4.5	AWS Greengrass Group Creation	26
4.6	AWS Console and Board: Set up AWS IoT for the Demo	31
4.6.1	Register the Device Certificate to the AWS IoT for the "thing"	31
4.6.2	AWS Console Create Publish/Subscribe Policy	34
4.6.3	Linux Environment: Configure the AWS Example Application that Connects to AWS	35
4.6.4	Lambda Functions on AWS IoT Greengrass	37
4.6.5	On Board: Execute the AWS Example Application	46
5	I MK8X SECURE BOOT	51
5.1	Preparing the environment to build a secure boot image	51
5.2	Building a Secure Signed Image	52
5.2.1	Programming SRK Hash	52
5.2.2	Prepare the boot image layout	54
5.2.3	Signing the boot image	56
5.2.4	Flash the signed image	56
5.2.5	Prepare the OS container image	56
5.2.6	Dumping SRK Hash values in host machine	57
5.2.7	Verify SECO events	58
5.2.8	Close the device	59
5.3	Measured Boot with OPTI GA™ TPM M2.0	59
5.3.1	Measured boot Step to verify Platform Integrity	59
5.3.2	Measuring Kernel Image Hash	59

5.3.3	Extending Measured Hash to PCR	59
5.3.4	Measuring Content from Specified PCR	59
5.3.5	Generating TPM Measured Boot Policy	59
5.3.6	Define NV-Area in TPM for PCR Storage	60
5.3.7	Extending PCR value to Secure NV RAM	60
5.3.8	Verifying Platform Integrity Check	60
6	APPENDIX	61
6.1	AI_M -96 Boards	61
6.2	AWS Greengrass	62
6.3	Tresor Mezzanine OPTI GA™ TPM2.0	63
6.4	Trusted Platform Module -TPM	64
6.5	i MK8X Secure Boot Overview	67
6.5.1	Secure AHAB boot architecture	67
6.5.1.1	The System Control Unit (SCU)	68
6.5.1.2	The Security Controller (SECO)	69
6.5.1.3	The Image Container	69
6.5.1.4	Secure boot flow	69
6.5.1.5	AHAB Secure Boot Process Overview	71
6.5.1.6	Architecture and image supporting secure boot	73
6.6	Measured boot Principles	74
7	REFERENCES	76

FIGURES

Figure 1: Security Starter Kit with i.MX8X and OPTI GA™ TPM2.0 Architecture	6
Figure 2: Hardware Connection Setup on AI_M board	8
Figure 3: AWS Greengrass Group	62
Figure 4: Tresor Mezzanine OPTI GA™ TPM2.0	63
Figure 5: Tresor Mezzanine connector OPTI GA™ TPM2.0	64
Figure 6: Root of Trust	65
Figure 7: Measured/Trusted Boot Process	66
Figure 8: Container layout	68
Figure 9: Secure boot flow overview	70
Figure 10: Secure boot image layout	71
Figure 11: Signature Block	72

TABLES

Table 1: Preloaded Keys and Certificates	17
--	----

ACRONYMS AND ABBREVIATIONS

Definition/Acronym/Abbreviation	Description
AI_M board	Artificial Intelligence and Machine Learning board featuring the NXP i.MX 8X MPU
AES	Advanced Encryption Standard
AHAB	Advanced High Assurance Boot
AWS	Amazon Web Services
BSP	Board Support Package
CA	Certificate Authority
CAA M	Cryptographic Acceleration and Assurance Module
CMS	Cryptographic Message Syntax
CSF	Command Sequence File
CSR	Certificate Signing Request
CST	Code Signing Token
DCD	Device Configuration Data
GG	AWS Greengrass
OS	Operating System
OTP	One-Time Programmable
PK	Public Key Infrastructure
SA	Signature Authority
SCFW	SCU Firmware
SDP	Serial Download Protocol
SECO	Security Controller
SPL	Secondary Program Loader
SRK	Super Root Key
SSK	Security Starter Kit
TPM	Trusted Platform Module
USB	Universal Serial Bus
TLS	Transport Layer Security
RSA	Rivest-Shamir-Adleman
IoT	Internet of Things
HSM	Hardware Security Module
PKCS#11	PKCS#11 (Public Key Cryptography Standards) defines an API to communicate with cryptographic security tokens such as smart cards, USB keys and HSMs
HW	Hardware
MQTT	Message Queuing Telemetry Transport
SSL	Secure Sockets Layer
SHA	Secure Hash Algorithm
SDK	Software Development Kit
ECC	Elliptic Curve Cryptography
ARN	Amazon Resource Name
SECO	Security Controller
FW	Firmware

NV RAM	Non Volatile Random Access Memory
API	Application Programming Interface
UUD	Universally Unique Identifier
SCP	Secure Copy Protocol
SCU	System Control Unit
IAM	AWS Identity and Access Management
TSS	TPM Software Stack

1 INTRODUCTION

1.1 Purpose of the Document

This guide describes - how to set up the OPTI GA™ TPM 2.0 on Arrow A_ML based Yocto platform with integrated TPM driver and Amazon Greengrass support. This is a hardware layer security for A_ML communication with AWS IoT Services.

1.2 Intended Audience

This document is for developers and end-users who want to use OPTI GA™ TPM 2.0, A_ML, AWS services to develop a gateway solution enabled with hardware layer security.

1.3 Prerequisites

Below are the list of Hardware and Software needed to enable demonstration of the AWS GG and OPTI GA™ TPM 2.0 security,

- Security Starter Kit Setup will require the following:
 - A_ML Board
 - Tresor Mezzanine board (with the OPTI GA™ TPM 2.0 installed)
 - SD card - 16GB
 - MicroUSB debug cable
 - Power Supply -
 - [MEAN WELL GST60A12-P1](#)
 - [5.5/2.1mm to 4.75/1.7mm cable DC plug converter](#)
- Linux PC (Mipi comfor serial console)
- Internet connectivity (Wi-Fi/Ethernet) of Board and Linux PC should be on same Network

1.4 Scope of Detailed Design

Integration of AWS IoT Greengrass with OPTI GA™ TPM 2.0 to provide hardware-based endpoint device security. This integration ensures the use of private key to establish device identity, which is securely stored in tamper-proof hardware devices, which prevents the device from being compromised, impersonated and other malicious activities.

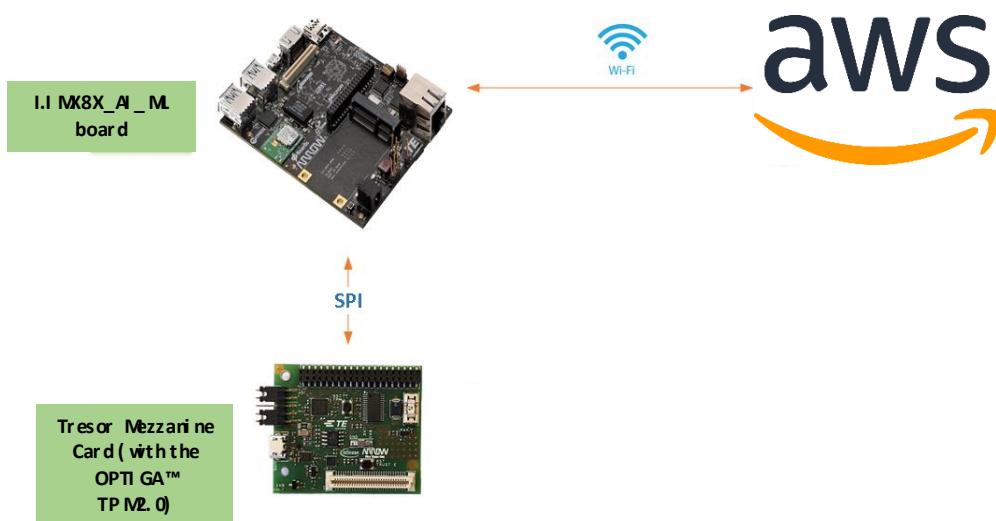


Figure 1: Security Starter Kit with MK 8X and OPTI GA™ TPM 2.0 Architecture

2 ENVIRONMENT SETUP

1. **Cloud Services** – Amazon Web Services (User must have an AWS Account Credentials before using this Guide)
2. **Gateway device** - AI_ML Board [AI_ML BOARD](#)
3. **Hardware security device** - [Tresor Mezzanine Infini neon OPTI GA™ TPM2.0](#). (TPM devices swapped with the Infini neon OPTI GA™ SLB 9670 TPM2.0)
4. **Power Supply cable** – 12V-5A 60WAC/DC Power Supply & 2 1/1.7mm plug converter
5. **Debug Cable** – MicroUSB debug cable
6. **HOST PC** – Linux as Operating System(Ubuntu 16.04)

3 HARDWARE SETUP

3.1 Hardware setup – Security Starter Kit with i.MX 8X and OPTIGA™ TPM 2.0 Hardware connection between i.MX 8X (AI_ML board) and OPTIGA™ TPM M2.0

The mezzanine will be mounted on top of the AI_ML board as shown in Figure 2. When the AI_ML board is powered up, the Power LED on the OPTIGA™ TPM M2.0 board turns on, indicating that the board is correctly connected. *Need to ensure the dpsw switch settings are configured for SD1 (0011). These settings are also on the silk screen of the board for your reference.*

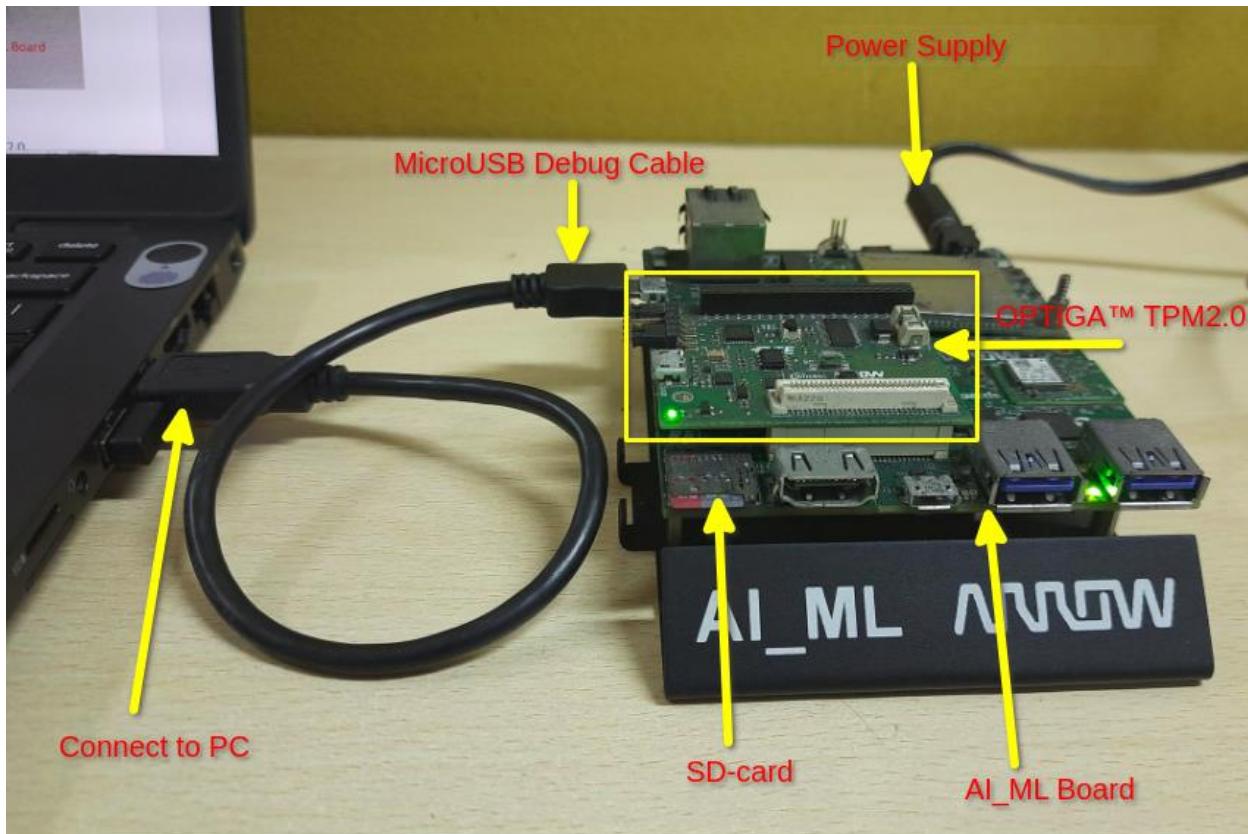


Figure 2: Hardware Connection Set up on AI_ML board

3.1.2 Powering up the Board

3.1.3 Open board's terminal - consde (Minicom) on Linux PC

2. Connect serial cable's USB end to Linux PC's USB
3. On Linux PC Launch Minicom utility as shown below (For debugging purpose)

Linux-PC ~\$ sudo minicom-s

- a. Baud rate 115200
 - b. Parity none
 - c. hardware flow control/software flow control none
 - d. Serial device /dev/ttyUSB0
 - e. save setup as def
5. After the A_ML board boots up, it will display below login console on Minicom terminal on Linux PC as shown below
 6. Username for board is "root" and password is "root" (if asked for)

```
NXP i.MX Release Distro 4.14-sumo imx8qxpaiml ttyLP2
imx8qxpaiml login: root
Last login: Wed Sep 16 12:58:47 UTC 2020 on tty7
root@imx8qxpaiml:~#
```

4 SOFTWARE SETUP

4.1 I. MK_8X-SSK and OPTI GA™ TPM2.0 Yocto Environment Set up

4.1.1 Pre-requisite

- Host PC(x86) having Linux Ubuntu 16.04 LTS installed (to build Yocto image)
- Basic understanding of Linux commands

4.1.2 Steps to build the BSP for the I. MK_8X-SSK and OPTI GA™ TPM2.0

[**Note** Default, image will be available in the SD-card. But If user wants to install the new image again on the SD-card or in case the image gets corrupted, then below steps can be handy]

1. Download the SSK A_ML release package from Arrow GitHub [Security-Starter-Kits-i. MK_8X-SSK.zip](#) on Linux HOST PC
2. Extract the [Security-Starter-Kits-i. MK_8X-SSK.zip](#)

```
Linux-PC $ unzip Security-Starter-Kits-i. MK_8X-SSK.zip
```

Extracting the zip file, one will find the below contents:

- [i. MK_8X-SSK Developers Guide.pdf](#) that has detailed description of all the components, examples and howto enable all features of the Kit.
- [i. MK_8X-SSK Quick Start Guide.pdf](#)
- i MK_8X_SSK_Pkg_Rel
 - Firmware_Image
 - SSK_AWS_Demo
 - SSK_Cert_Auth_Config
 - SSK_Suit_Configuration
 - Linux_Yocto_Build

1. Download [meta-efi-nfoclip-SSK-V1.0.tar.gz](#) from Arrow GitHub and copy to `i MK_8X_SSK_Pkg_Rel/linux_Yocto_Build/`.
2. Run the build script `build_script_aiml.sh` to complete Yocto environment set up.

```
Linux-PC $ cd i MK_8X_SSK_Pkg_Rel/linux_Yocto_Build/
Linux-PC $ ./build_script_aiml.sh
```

3. Please note that, if you re-build any module then it is better to re-build all modules, which are dependent on that module. For example, if you change anything in Linux kernel code and rebuild it using above commands then you must need to re-build kernel-module-aird, imx-gpu-sdk etc. packages to avoid conflicts.

```
Linux-PC $ bitbake <PACKAGE_NAME> -c cleanall
Linux-PC $ bitbake <PACKAGE_NAME>
```

```
Linux-kernel
```

```
Li nux-PC $ bitbake linux-imx -c d eanall
Li nux-PC $ bitbake linux-imx (Build Linux kernel only)
```

Same way

```
Li nux-PC $ bitbake u-boot-imx -c d eanall
Li nux-PC $ bitbake u-boot-imx (Build UBoot code only)
```

4. When build is successful, the final SD-card image resides at below location
bl d-xwayland-a i nh/t mp/deploy/images/imx8qxpai nh/_copied_to/Firmware_l mage
5. File name should be **fsl-image-qt5-imx8qxpai nh.sdcard bz2** which is soft link of original build image file **fsl-image-qt5-imx8qxpai nh-<TIMESTAMP>.rootfs.sdcard bz2**

[**Note:** Please refer the [Bitbake User Manual](#) for better understanding of bitbake files, recipes and layers as well as options to build an image.]

6. Once Yocto build is complete Create SD-card image.
7. Navigate to the directory “Firmware_l mage” in order to find built SD-card image.

```
Li nux-PC $ cd MK_8X_SSK_Pkg_Rel/Firmware_l mage /
Li nux-PC $ bunzip2 -dkf fsl-image-qt5-imx8qxpai nh-TIMESTAMP.rootfs.sdcard bz2
Li nux-PC $ ls /dev/<device>
```

8. Flash the SD-card using below command

```
Li nux-PC $ sudo dd if=fsl-image-qt5-imx8qxpai nh-TIMESTAMP.rootfs.sdcard of=/dev/<device> bs=1M
conv=fsync status=progress && sync
```

9. Unmount and eject the SD-card from the Li nux PC
10. Reinserting the SD-card to Li nux PC will mount below partitions, verify using below command

```
Li nux-PC $ sudo lsblk
|-mmcblk0p1 179:1 0 64M0 part / media/<username>/Boot_imx8qxpai
└mmcblk0p2 179:2 0 7G 0 part / media/<username>/rootfs_imx8qxpai
```

11. Copy below listed files to file system imx8qxpai_Suit_package

- **Li nux-PC \$ sudo cd MK_8X_SSK_Pkg_Rel**
- **Li nux-PC \$ sudo cp -r SSK_AWS_De mo/SSK_Suit_Configuration/**
/media/<username>/rootfs_imx8qxpai/home/root/

- Li nux-PC \$ sudo cp -r SSK_Cert_Auth_Config/AWS_Config/openssl.cnf / media/<user name>/rootfs_id/etc/ssl/
- Li nux-PC \$ sudo cp -r SSK_Cert_Auth_Config/AWS_Config/config.json / media/<user name>/rootfs_id/greengrass/config/
- Li nux-PC \$ sudo cp -r SSK_Cert_Auth_Config/AWS_ROOTCA/root CA key / media/<user name>/rootfs_id/greengrass/certs/
- Li nux-PC \$ sudo cp -r SSK_Cert_Auth_Config/AWS_ROOTCA/root CA pem / media/<user name>/rootfs_id/greengrass/certs/
- Li nux-PC \$ sudo cp -r SSK_Suit_Configuration/Tpm_Measured_Boot/Measured_boot.sh / media/<user name>/rootfs_id/etc/init.d/
- Li nux-PC \$ sudo cp -r SSK_Suit_Configuration/Tpm_Measured_Boot/rc-local.service / media/<user name>/rootfs_id/etc/systemd/system/
- Li nux-PC \$ sudo cp -r SSK_Suit_Configuration/Tpm_Measured_Boot/rc.local / media/<user name>/rootfs_id/etc/
- Li nux-PC \$ sync

12. The SD card is now ready for use on the A_ML board
13. Plug the SD card in the A_ML board and power up the board. The board will boot and it will display boot up logs and finally ask for login

4.2 Keys and certificates information

- Preloaded one root certificate (associated private key provided as a file) - **root CA pem**
- One root private key - **rootCA key**
- One AWS IoT root certificate - **root.ca.pem**
- Key pairs associated device certificate Pre-Flashed into Trusted Platform Module Chip – **Inside HS M**
- One Device certificate signed with the root private key and the associated private key needed for TLS mutual authentication with AWS IoT – [generated using **SSK_Suit_Configuration.sh**]

4.3 OPTI GA™ TPM2.0 Setup Script

After successful boot-up of the AI_ML board, the user needs to set up the hardware security chip OPTI GA™ TPM 2.0. This Setup script will set up/install prerequisite packages and TPM configuration (i.e. create Keypair, device certificate and store in OPTI GA™ TPM 2.0).

1. Enable the Wi-Fi internet connectivity using mobile hotspot or router.
2. Use Minicom console to run the script, as mentioned below

```
root @ mx8qxpai nh:~#cd /greengrass/certs
root @ mx8qxpai nh:~#openssl genrsa -out rootCA.key 2048
root @ mx8qxpai nh:~#openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 7000 -out
rootCA.pem -subj /C="IN"/ST="GUJ"/L="AHMEDABAD"/O="Arrow"/OU="eic"
root @ mx8qxpai nh:~# cd ~/SSK_Suit_Configuration
root @ mx8qxpai nh:~#/SSK_Suit_Configuration.sh
```

```
root@imx8qxpaiml:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh
+++++
Welcome to Security Starter Kit AI_ML Board Auto Flashing Tool
+++++
-----Prerequisite-----
--> Please Enable Your Mobile Or Router Hotspot for internet connectivity <--
--> Have you enable the hotspot? y/n <--
y
```

3. If internet connectivity is enabled, press "Y". This will fetch the SSID of Wi-Fi list. Enter the SSID name and password details, example shown as below

```
--> [WIFI] List of available Wifi devices in Range... <--
SSID: ei-SecureWiFi
SSID: ei-GuestWiFi
SSID: Sai_Financial
SSID: ei-SecureWiFi
SSID: ei-GuestWiFi
SSID: TP-Link_8A6D
SSID: KVMS_Private
SSID: Echo
SSID:
      * SSID List
SSID: haresh.vithlani
SSID:
      * SSID List
SSID: KVMS_EIC
SSID: Rahul
SSID: abzaveri
SSID: hello
SSID: ei-SecureWiFi
SSID: ei-GuestWiFi
SSID: DIRECT-bd-HP_M227f_LaserJet
SSID: TP-Link_8A6C_5G
SSID: ORBI28
      * SSID List
SSID: ORBI70
      * SSID List
SSID: ei-SecureWiFi
SSID: ei-GuestWiFi
SSID:
SSID: ei-SecureWiFi
SSID: ei-GuestWiFi

--> Can you see your wifi devices:SSID? y/n <--
y
--> Please Enter the Name of your Wifi-Device SSID <--
hello
--> Can you please Provide the Password of your Wifi-Device <--
12341234
Successfully initialized wpa_supplicant
```

4. Now it will start installing the package. Running for the first time it will take ~20 minutes.

```
--> Installing Required Python Packages for Implementing Security Setup on Board with TPM2.0 <-->
--> Process will take Time...Please Wait for 20mins <-->
```

5. After completion of script, it creates Keypair and device certificate using TPM

```
--> Create Device Certificates Signed By RootCA <-->
Signature ok
subject=/C=IN/ST=GUJ/L=AHM/O=Arrow/OU=eic/CN=SSK
Getting CA Private Key
--> TPM2.0-Device Certificate Created Successfully <-->
--> Board is ready to Use for Demo <-->
```

6. Verify the Steps using below commands [Enter #P/N 1234]

```
root@mx8qxpai:~# ./SSK_Suit_Configuration.sh setup_result
```

```
ultt@imx8qxpaiml:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh setup_result
--> Running TPM Self-Test <-->
--> Checking TPM Self-Test Result <-->
status: success
data: 001fe18b000000009fbb
--> Verifying Token Handle Created or Not <-->
handle: 0x81000000
name-alg:
  value: sha256
  raw: 0xb
attributes:
  value: fixedtpm|fixedparent|sensitiveorigin|userwithauth|restricted|decrypt
  raw: 0x30072
type:
  value: rsa
  raw: 0x1
exponent: 0x0
bits: 2048
scheme:
  value: null
  raw: 0x10
scheme-halg:
  value: (null)
  raw: 0x0
sym-alg:
  value: aes
  raw: 0x6
sym-mode:
  value: cfb
  raw: 0x43
sym-keybits: 128
rsa: c8f3b0e05898ed357de7e273436d015a4a461d6f94890a4f211a1eede2e13ea6819bfff67179e9590e64alc5b7f38a4f4519273a418ad2c79f58f330bd8d961200d83a21a953b9
```

```

---> Verifying Token Module Created with Provide Label or Not <---
p11-kit-trust: p11-kit-trust.so
library-description: PKCS#11 Kit Trust Module
library-manufacturer: PKCS#11 Kit
library-version: 0.23
tpm2_pkcs11: libtpm2_pkcs11.so
library-description: TPM2.0 Cryptoki
library-manufacturer: tpm2-software.github.io
library-version: 42.42
token: greengrass
    manufacturer: Infineon
    model: SLM9670
    serial-number: 0000000000000000
    hardware-version: 1.38
    firmware-version: 1.11
flags:
    rng
    login-required
    user-pin-initialized
    token-initialized

---> Validating TPM2.0 Security Keys are Loaded inside the Protected/Shielded Area <---

---> Please Provide User-Pin Below(Provided in guide) <---
Object 0:
    URL: pkcs11:model=SLM9670;manufacturer=Infineon;serial=0000000000000000;token=greengrass;id=%32%35%66%65%35%32%64%66%36%30%30%65%66%30%63%62
Token 'greengrass' with URL 'pkcs11:model=SLM9670;manufacturer=Infineon;serial=0000000000000000;token=greengrass' requires user PIN
Enter PIN:
    Type: Private key (RSA)
    Label: greenkey
    Flags: CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
    ID: 32:35:66:65:35:32:64:66:36:30:30:65:66:30:63:62

---> Done Exiting TPM-Full Test after Production Flash Process <---

```

Additional Steps:

[**Note:** If user wants to clear the TPM then below steps will help for debugging]

7. TPM Clear command

In case of some mistakes when following the steps or any error occurred while configuring set up, the User can reset the TPM2.0 with below command.

```
root@imx8qxpaiml:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh tpm_clear
```

```

root@imx8qxpaiml:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh tpm_clear
---> TPM2.0 Cleared with Tokens and Labels <---
0x1500016:
---> Clearing Done Exiting <---
root@imx8qxpaiml:~/SSK_Suit_Configuration# 
```

8. If TPM is clear, then user will get below logs. Again, execute the above steps from 1 to 6 for set up again as described in 4.3 else follow the section 4.4.1 and 4.4.2

```
ultt@imx8qxpaiml:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh setup_res
---> Running TPM Self-Test <---
---> Checking TPM Self-Test Result <---
status: success
data: 001fe18b000000009fb0

---> Verifying Token Handle Created or Not <---
---> Verifying Token Module Created with Provide Label or Not <---
p11-kit-trust: p11-kit-trust.so
library-description: PKCS#11 Kit Trust Module
library-manufacturer: PKCS#11 Kit
library-version: 0.23

---> Validating TPM2.0 Security Keys are Loaded inside the Protected/Shielded Area <---
---> Please Provide User-Pin Below(Provided in guide) <---
No matching objects found

---> Done Exiting TPM-Full Test after Production Flash Process <---
root@imx8qxpaiml:~/SSK_Suit_Configuration#
```

4.4 TLS Mutual Authentication and Session Establishment Using H/wSecurity of TPM With Amazon AWS IoT

Amazon cloud allows customers to use device certificates signed and issued by their own certificate authority (CA) to connect and authenticate with AWS IoT. This is an alternative to using certificates generated by AWS IoT and fits better for customers' needs. This method is used 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.

When establishing TLS connectivity, AWS IoT authenticates the connecting device by extracting the device certificate and verifying its signature against a customer preloaded root certificate. Similarly, the device needs to verify the server certificate against the stored AWS IoT root certificate to confirm the authenticity of the server to which it connects. TLS mutual authentication requires the device to prove the ownership of its private key used to form the device certificate and this is being done by signing some data packets with the private key.

The AI_ML Gateway with OPTI GA™ TPM 2.0 facilitates the creation and signing of a device certificate. The device certificate is intended for establishing TLS connections with mutual authentication. The Kit provides an example of how to use the device certificate and OPTI GA™ TPM 2.0 based crypto for establishing a TLS connection with Amazon AWS IoT (usually used for running MQTT protocol that runs on top of TLS).

The example requires the user to create an AWS account, create an OEM Root CA and upload it to AWS. The Device Certificate needs to be signed with the private key that created the OEM Root CA so the two certificates are chained. It needs to be noted that Amazon AWS does not allow the activation of the same OEM Root CA for multiple AWS accounts.

The example guides the user to perform the following steps:

- Create an AWS Custom CA register the Custom CA in AWS, provide verification to AWS and create an AWS Device Certificate
- Save the created AWS Custom CA and AWS Device key and cert in OPTI GA™ TPM 2.0.
- This way the AWS CAs and the AWS Device Certificate are unique and can be used with AWS for the Evaluation Kit by multiple users

To set and test the TLS mutual authentication and connectivity to AWS IoT, users will generate their own AWS Custom CA private key and certificate and AWS device key and certificate, which must be ECDSA 256 (actively using RSA 256 Technique).

Cert/key	Name of cert/key exposed by OPTIGA™ TPM and AI_ML	Description
Cloud IoT Root CA	root.ca.pem	Root CA of the Cloud IoT. It is used for TLS mutual authentication.
Gateway Root Certificate/Key	rootCA.pem rootCA.key	Gateway Root Certificate. For the Evaluation Kit this cert is predefined. The associated private key is provided as a file for execution of the payload verification example application.
Gateway Verification Certificate/Key	verificationCert.crt verificationCert.key	Gateway Verification Certificate. For the Evaluation Kit this cert is predefined. The associated private key is provided as a file for execution of the payload verification of Root Certificate with provided AWS.
Gateway/Device Private Key	Stored securely under TPM	Created and Stored under PKCS11 Handle with TPM Accessible only
Gateway/Device Certificate	aws_device_cert.pem	Device Credentials are accessible with private key being verified through TPM

Table 1: Preloaded Keys and Certificates

The AWS cloud certificate is preloaded in the AWS.

The example uses the following keys and certificates:

1. **AWS IoT Root Certificate:** Comes preloaded with AWS
2. **AWS Custom Gateway CA Key:** the user generates this private key. It is used to sign the AWS Device Certificate and to complete the AWS Custom CA Certificate registration process with AWS IoT
3. **AWS Custom Gateway CA Certificate:** the user creates this certificate and using the openssl tool. This certificate needs to be uploaded to the AWS IoT cloud
4. **AWS Device Private Key:** Private key is generated by user stored inside TPM securely, not exposed to outside world
5. **AWS Device Certificate:** the user creates this certificate. It must be created and signed with the AWS Custom Gateway CA Key. It is used during the AWS device registration step.

[Note: If you followed the steps as per [Section 4.3 - OPTIGA™ TPM2.0 Setup Script](#) then please ignore below [Section 4.4.1, 4.4.2](#)]

4.4.1 Linux Environment: Generate the required keys and certificate

To use your own X.509 device certificates, it is essential to register a CA certificate with AWS IoT. The CA certificate can then be used to sign device certificates. You can register up to 10 CA certificates with the same subject field per AWS account per AWS region. This allows you to have more than one CA - sign your device certificates.

[Note: The registered CA certificate must sign Device certificates. It is common for a CA certificate to be used to create an intermediate CA certificate. If you are using an intermediate certificate to sign your device certificates, you must register the intermediate CA certificate. Use the AWS IoT root CA certificate when you connect to AWS IoT even if you register your own root CA certificate. The AWS IoT root CA certificate is used by a device to verify the identity of the AWS IoT servers]

Earlier, AWS IoT released support for customers in case user wants to use their own device certificates signed by their preferred Certificate Authority (CA). This is in addition to the support for AWS IoT generated certificates. The CA certificate is used to sign and issue device certificates, while the device certificates are used to connect a client to AWS IoT. Certificates provide strong client-side authentication for constrained IoT devices. During TLS handshake, the server authenticates the client using the X.509 certificate presented by the client.

With this feature, customers with existing devices in the field or new devices with certificates signed by a CA other than AWS IoT can seamlessly authenticate with AWS IoT. It also provides manufacturers the ability to provision device certificates using their current processes and then register those device certificates to AWS IoT. For example, if a customer's manufacturing line lacks internet connectivity, they can provision their devices offline with their own CA issued certificates and later register them with AWS IoT.

This exercise will walk you through an end-to-end process of setting up a client that uses a device certificate signed by your own CA. First, you will generate a CA certificate that will be used to sign your device certificate. Next, you will register the CA certificate and then register the device certificates. After these steps, your device certificate will be ready to connect AWS IoT service.

4.4.1.1 AWS Custom Gateway CA Creation

Let us begin by creating your first sample CA certificate using OpenSSL in a terminal. In reality, you will have the signing certificates issued by your CA vendor in the place of this sample CA. This sample CA certificate is used later in the walk-through to sign a device certificate that will be registered with AWS IoT:

[Note: If you do not have a CA certificate, you can use [OpenSSL tool](#)]

To create a CA certificate

1. Generate a key pair on board at /greengrass/certs.

```
root@mx8qxpai-nh:~# cd /greengrass/certs/
root@mx8qxpai-nh:~# openssl genrsa -out rootCA.key 2048
```

2. Use the private key from the key pair to generate a CA certificate

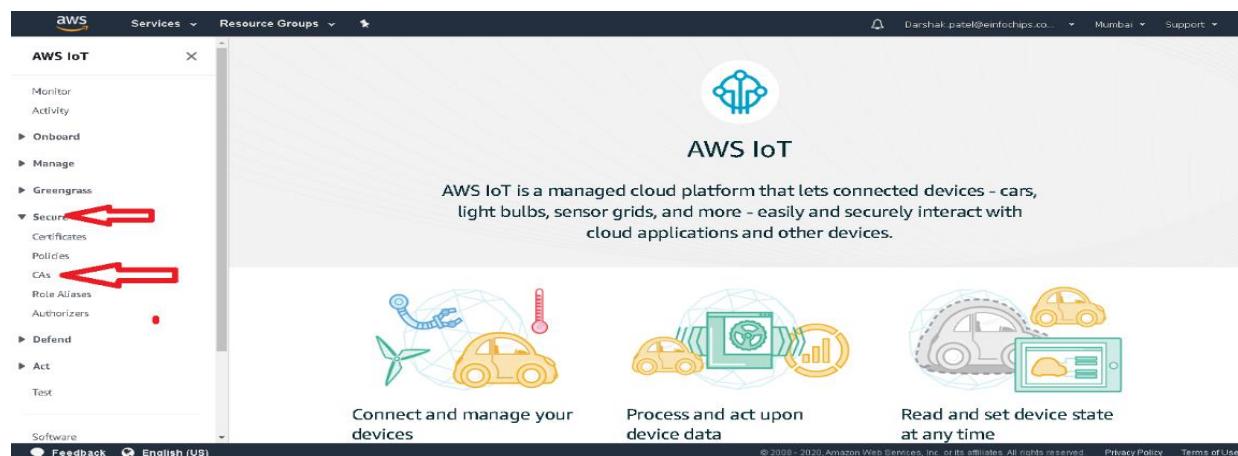
```
root CA.pem
```

4.4.1.2 Registering Your CA Certificate

[Note: A CA certificate cannot be registered with more than one account in the same AWS Region. However, a CA certificate can be registered with more than one account if the accounts are in different AWS Regions. A CA certificate is used to register Device certificates, which are signed by CA certificate.]

To register a CA certificate

Get a registration code from AWS IoT. This code is used as the Common Name of the private key verification certificate. One can retrieve the registration code using the AWS CLI or from the [AWS IoT Console >> SECURE >> CA >> Register Certificate section](#)



Register a CA certificate

To use your own X.509 certificates, you must register a CA certificate with AWS IoT. You must prove you own the private key associated with the CA certificate by creating a private key verification certificate. The CA certificate can then be used to sign device certificates. You can register up to 10 CA certificates with the same subject field and public key per AWS account. This allows you to have more than one CA sign your device certificates.

Step 1: Generate a key pair for the private key verification certificate

```
openssl genrsa -out verificationCert.key 2048
```

Step 2: Copy this registration code

```
1c88ceefdaf69a90f579e3abe85784b1c6d5b8e40c10743cfcc59fd28e432e56
```

1. Generate a key pair for the private key verification certificate

2. Create a CSR for the private key verification certificate. Set the Common Name field of the certificate to your registration code. Fetched from above [AWS IoT Console >> SECURE >> CA >> Register Certificate section](#)

User needs update some information, including the Common Name for the certificate

3. Use the CSR to create a private key verification certificate

```
CACreateCsr -out verificationCert.crt -days 500 -sha256
```

4. In the navigation pane, go to Secure option on IoT Console then select [Secure >> CA >> "Register your CA certificate"](#), and upload your sample CA certificate and verification certificate

To use your own X.509 certificates, you must register a CA certificate with AWS IoT. You must prove you own the private key associated with the CA certificate by creating a private key verification certificate. The CA certificate can then be used to sign device certificates. You can register up to 10 CA certificates with the same subject field and public key per AWS account. This allows you to have more than one CA sign your device certificates.

Step 1: Generate a key pair for the private key verification certificate

```
openssl genrsa -out verificationCert.key 2048
```

Step 2: Copy this registration code

```
1c88ceefdaf69a90f579e3abe85784b1c6d5b8e40c10743cfcc59fd28e432e56
```

Step 3: Create a CSR with this registration code

```
openssl req -new -key verificationCert.key -out verificationCert.csr
```

Put the registration code in the **Common Name** field

Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []: 1c88ceefdaf69a90f579e3abe85784b1c6d5b8e40c10743cfcc59fd28e432e56
Email Address []:

Step 4: Use the CSR that was signed with the CA private key to create a private key verification certificate

```
openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out verificationCert.pem
```

Step 5: Upload the CA certificate (rootCA.pem)

Select CA certificate

Step 6: Upload the verification certificate (verificationCert.crt)

Select verification certificate

Activate CA certificate
 Enable auto-registration of device certificates

Cancel Register CA certificate

4.4.2 Linux Environment: Secure Device Certificate and Private Key Gen

You can use a CA certificate, registered with AWS IoT, to create a device certificate. The device certificate must register with AWS IoT before use

1. Changes have been made in AML- meta-security build image
2. Check if any persistent handle is already present with OPTIGA™ TPM 2.0.

```
root @ mx8qxpai nh:~# tpm2_list_persistent
- handle: 0x81000000
  name: alg
  value: sha256
  raw: 0xb
  attributes:
    value: fixed|fixedparent|sensitive|dataorigin|user|withauth|restricted|decrypt
    raw: 0x30072
.
```

- To clear the earlier token and handle inside OPTIGA™ TPM 2.0 Chipset, use the command

```
root @ mx8qxpai nh:~# tpm2_evictctrlr -a o-c 0x81000000 -p 0x81000000
```

- Install python packages.

```
root @ mx8qxpai nh:~# pip install --upgrade pip pip install pyyaml==5.3.1; sleep 1; pip install cryptography==2.9.2; sleep 1; pip install paramiko==2.7.2; sync;sync
```

- Clone the script file to board

```
root @ mx8qxpai nh:~# git clone https://github.com/tpm2-software/tpm2-pkcs11
root @ mx8qxpai nh:~# cd tpm2-pkcs11/
root @ mx8qxpai nh:~# tpm2-pkcs11/# git checkout a82d0709c97c88cc2e457ba111b6f51f21c22260
```

- Run the script to generate keys and token inside TPM 2.0 inside given directory.

```
root @ mx8qxpai nh:~# cd ~/tpm2-pkcs11/tools
root @ mx8qxpai nh:~# ./tpm2_ptod.py init --pobj=pin=1234 --path=/opt/tpm2-pkcs11/
Created a primary object of id 1

root @ mx8qxpai nh:~# ./tpm2_ptod.py addtoken --pid=1 --pobj=pin=1234 --sopin=1234 --userpin=1234 --label=greengrass --path=/opt/tpm2-pkcs11/
Created token label: greengrass

root @ mx8qxpai nh:~# ./tpm2_ptod.py addkey --algorithm=rsa2048 --label=greenkey --userpin=1234 --keylabel=greenkey --path=/opt/tpm2-pkcs11/
Added key as label: "greenkey"
```

7. Create Soft link to the resource manager libraries for listing token from OPTI GA™ TPM2.0 and displaying it on board's console

```
root @ mx8qxpai nh:~# cd /usr/lib/
root @ mx8qxpai nh:/usr/lib# ln -s libtss2-tcti-ta br md.so.0 libtss2-tcti-ta br md.so
```

8. To check the URL's of token generated, p11tod and p11-kit is needed along with other packages opensc. Use the command **p11-kit list-modules** to list the HSM modules available with tokens.

```
root @ mx8qxpai nh:~# p11-kit list-modules
p11-kit-trust: p11-kit-trust.so
  library-description: PKCS#11 Kit Trust Module
  library-manufacturer: PKCS#11 Kit
  library-version: 0.23
  tpm2_pkcs11: libtpm2_pkcs11.so
    library-description: TPM2.0 Cryptoki
    library-manufacturer: tpm2-software.github.io
    library-version: 42.42
    token: greengrass
      manufacturer: Infineon
      model: SLB9670
      serial-number: 0000000000000000
      hardware-version: 1.16
      firmware-version: 7.40
      flags:
        rng
        logon-required
        user-initialized
        token-initialized
```

9. Use command “p11tod --list-tokens” to see the token with its URL.

```
root @ mx8qxpai nh:~# p11tod --list-tokens
Token 0:
  URL: pkcs11: model=SLB9670; manufacturer=Infineon; serial=0000000000000000; token=greengrass
  Label: greengrass
  Type: Hardware token
  Manufacturer: Infineon
  Model: SLB9670
  Serial: 0000000000000000
  Module: libtpm2_pkcs11.so
```

10. Use command “p11tod --list-prvkeys pkcs11: manufacturer=Infineon” to see the PKCS listing OPTI GA™ TPM2.0 private and public keys.

Note: Provide PIN 1234 if asked

```
root @ mx8qxpai nh:~# p11tod --list-pkcs11-manufacturer=rfi neon
      Object 0
URL:
pkcs11: model =SLB9670; manufacturer=rfi neon; serial=0000000000000000; token=greengrass; id=%36%37%36%30%39%61%62%36%65%65%36%39%34%33%30; object=greenkey
      Token 'greengrass' with URL
'pkcs11: model =SLB9670; manufacturer=rfi neon; serial=0000000000000000; token=greengrass'
requires user PIN
Enter PIN
      Type: Private key (RSA)
      Label: greenkey
      Flags: CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
      ID: 36:36:37:36:30:39:61:62:36:65:65:36:39:34:33:30
```

[11](#) Creation of soft link to "libpkcs11.so"

```
root @ mx8qxpai nh:~# cd /usr/lib/engines/
root @ mx8qxpai nh:~# ln -s pkcs11.so libpkcs11.so
root @ mx8qxpai nh:~# export PKCS11_MODULE_PATH=/usr/lib/libtpm2_pkcs11.so
```

[12](#) Edit openssl.conf for enabling PKCS11 interface for TPM (Edit only highlighted points)

```
root @ mx8qxpai nh:~# vi /etc/ssl/openssl.conf
#
# OpenSSL example configuration file
# This is mostly being used for generation of certificate requests.
#
# This definition stops the following lines choking if HOME isn't
# defined.
HOME      =
RANDFILE   = $ENV::HOME/.rnd

openssl_conf = openssl_init

# Extra OBJECT IDENTIFIER information
#oid_file    = $ENV::HOME/.oids
oid_section  = new_oids
```

[13](#) Edit below contents at bottom of openssl.conf file, open /etc/ssl/openssl.conf

```
[openssl_init]
engines=engine_section
```

```
[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
dynamic_path = /usr/lib/engines/libpkcs11.so
MODULE_PATH = /usr/lib/pkcs11/libpm2_pkcs11.so
init = 0
```

14. Generating Certificate Signing Request with openssl

```
root@mx8qxpai-nh:~# openssl req -engine pkcs11 -new -key
"pkcs11: model=SLB9670; manufacturer=rfineon; token=greengrass; object=greenkey; type=private; pin-value=1234" -keyform engine -out /greengrass/certs/deviceCert.csr

Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]: GUARAT
Locality Name (eg, city) []: AHM
Organization Name (eg, company) [Internet Widgets Pty Ltd]: EC2
Organizational Unit Name (eg, section) []: KB
Common Name (e.g. server FQDN or YOUR name) []: SSK
Email Address []:

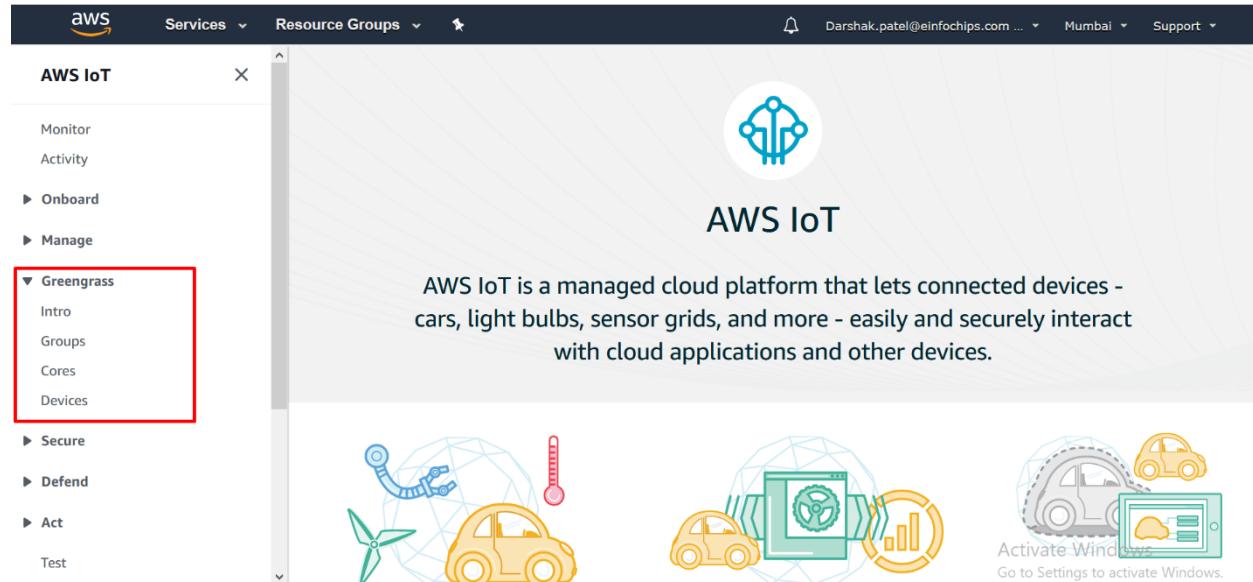
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: 1234
An optional company name []: ARROW
```

15. Registering Device Certificates manually

```
root@mx8qxpai-nh:~# cd /greengrass/certs/
root@mx8qxpai-nh:~# openssl x509 -req -in deviceCert.csr -CA root CA.pem -CAkey root CA.key -CAcreateserial -out aws_device_cert.pem -days 500 -sha256
```

4.5 AWS Greengrass Group Creation

- 1 Create a Greengrass group using your AWS account.
- 2 Sign in to the AWS Management Console on your computer and open the AWS IoT console. Choose Get started, if this is the first attempt to open this console
 - In the navigation pane, choose Greengrass.



[**Note:** If you don't see the Greengrass node, change to an AWS Region that supports AWS IoT Greengrass. For the list of supported regions, see [[AWS IoT Greengrass](#)] in the Amazon Web Services General Reference.]

- 3 On the Welcome to AWS IoT Greengrass page, choose <[Create a Group](#)>.
 - Greengrass will need your permission to access other services. If prompted with this dialog box, choose 'Grant permission' to allow the console to create or configure the Greengrass service role for you.

Greengrass needs your permission to access other services

AWS IoT Greengrass works with other AWS services, such as AWS IoT and AWS Lambda. Greengrass needs your permission to access these services and read and write data on your behalf. [Learn more](#)

When you grant permission, Greengrass does the following:

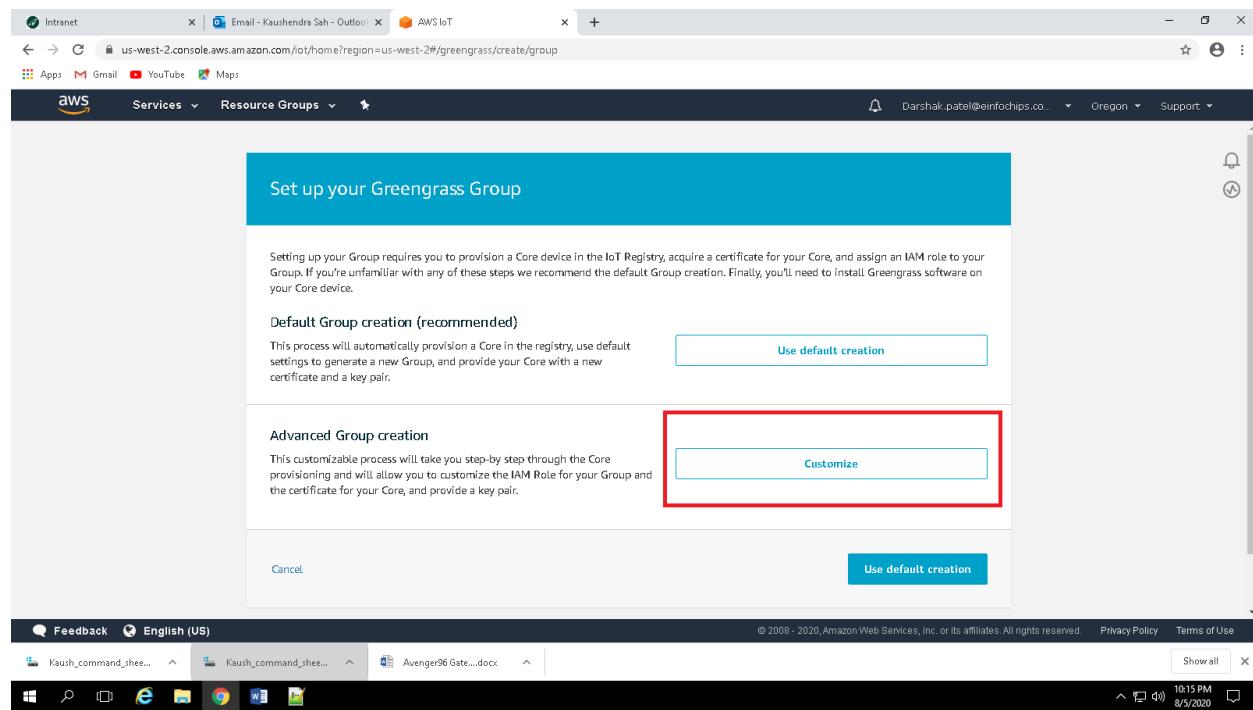
- Creates a service role named Greengrass_ServiceRole, if one doesn't exist, and attaches the [AWSGreengrassResourceAccessRolePolicy](#) managed policy to the role.
- Attaches the service role to your AWS account in the AWS Region that's currently selected in the console.

This step is required only once in each AWS Region where you use Greengrass.

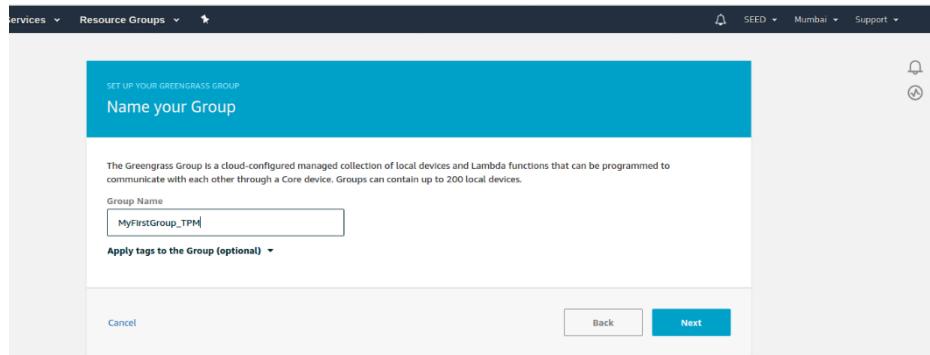
[Cancel](#)

[Grant permission](#)

4. On the Set up your Greengrass group page, choose “Customize” to create a group in AWS IoT Greengrass

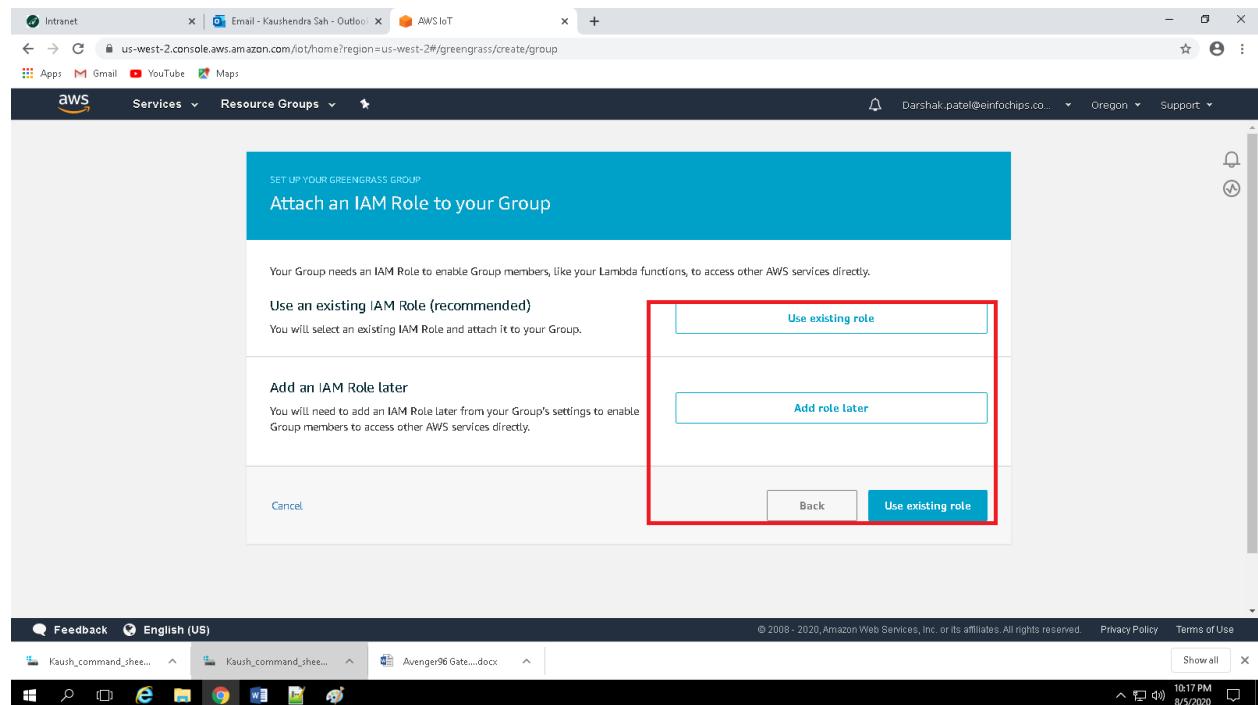


5. Enter a name for your group (for example, **MyFirstGroup_TPM**), and then choose Next



6. Add IAM Role to the Greengrass Group

Click on “Use existing role” and select the Greengrass_ServiceRole.



SET UP YOUR GREENGRASS GROUP

Use an existing IAM Role

Your Group needs an IAM Role to enable Group members, like your Lambda functions, to access other AWS services directly.

Select an IAM Role with a Greengrass Role Type

Search Role name

Greengrass_ServiceRole

[Cancel](#) [Back](#) [Next](#)

7. Select the **Stream Manager** option as **Customize** to **Disable** it

SET UP YOUR GREENGRASS GROUP

Stream manager

Stream manager enables the Core to ingest and process data streams and export them to cloud targets. Stream manager requires the Java 8 Runtime to be installed on the Core device. [Learn more](#)

Enable with default settings [Use defaults](#)

You can change the default values later in the Group's settings.

Customize settings [Customize settings](#)

You'll choose to enable or disable stream manager and optionally configure settings in the next step.

[Cancel](#) [Back](#) [Use defaults](#)

SET UP YOUR GREENGRASS GROUP

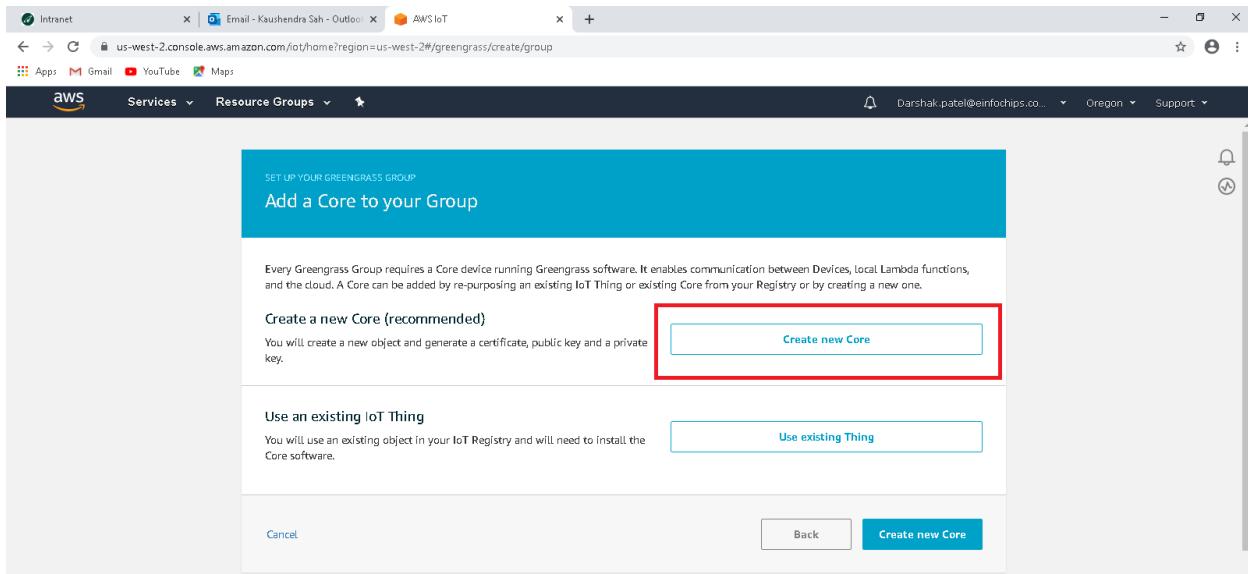
Configure stream manager

Stream manager enables the Core to ingest and process data streams and export them to cloud targets. Stream manager requires the Java 8 Runtime to be installed on the Core device. [Learn more](#)

Enable
 Disable

[Cancel](#) [Back](#) [Next](#)

- Select the Greengrass Core as per the Gateway Device running within **MyFirstGroup_TPM_Core**



SET UP YOUR GREENGRASS GROUP

Add a Core to your Group

Every Greengrass Group requires a Core device running Greengrass software. It enables communication between Devices, local Lambda functions, and the cloud. A Core can be added by re-purposing an existing IoT Thing or existing Core from your Registry or by creating a new one.

Create a new Core (recommended)
 You will create a new object and generate a certificate, public key and a private key.

Use an existing IoT Thing
 You will use an existing object in your IoT Registry and will need to install the Core software.

[Create new Core](#) [Use existing Thing](#)

[Cancel](#) [Back](#) [Create new Core](#)

- Select Security type as customize to upload the Gateway Device certificate Generated by OPTI GA™ TPM 2.0 and signed with Registered Root CA with AWS as depicted in [Section 4.4.1](#)

SET UP YOUR GREENGRASS GROUP

Set up your security

Your Core needs a certificate and a policy to access AWS IoT Greengrass, and you need to set permissions to and provision and IAM role for your Group. If you're unfamiliar with any these steps we recommend the automated set up.

1-Click

This will generate a certificate, public key and private key using AWS IoT's root CA, generate a default policy, and create a new IAM role with default permissions.

Use Defaults

Advanced setup

This will allow you to handle certificate signing request (CSR) based on a private key you own, customize your own policy, and use an existing IAM role or create a new one.

Customize

Cancel **Back** **Use Defaults**

4.6 AWS Console and Board: Setup AWS IoT for the Demo

The user must go through the following steps to set and test the TLS connectivity with AWS IoT

1. Create an Amazon AWS account
2. Sign into the AWS IoT Console
3. Create (Register) a "thing" in the Thing Registry
4. Register the CA to the AWS IoT.

4.6.1 Register the Device Certificate to the AWS IoT for the "thing"

On the AWS console, after the AWS Custom CA Certificate has been registered and activated, for the "thing" that has been created, the user must click again on Security as shown in the screenshot below

1. Then click on "View other options" and then "Use my certificate" (click on "Get started").

Create a certificate

A certificate is used to authenticate your device's connection to AWS IoT.

One-click certificate creation (recommended)

This will generate a certificate, public key, and private key using AWS IoT's certificate authority.

[Create certificate](#)

Create with CSR

Upload your own certificate signing request (CSR) based on a private key you own.

[Create with CSR](#)

Use my certificate

Register your CA certificate and use your own certificates for one or many devices.

[Get started](#)

- As shown in the screenshot below the user will have to click and select the CA that was last registered and then click on the bottom blue button "Next".

Select a CA

Select or register the CA certificate used to sign your device certificates. To use device certificates that are not signed by a registered CA, just select Next. [Learn more](#).

Registered CAs

Search CA certificate



3d8d56d4997296baa360a2196baef2261fb932fbfb1741eb45598ab57146f656

[View](#)

[Register CA](#)

[Cancel](#)

[Next](#)

- As per the screenshot below the user has now the option to select to **upload** and **register the Device Certificate**

Register existing device certificates

You can upload up to 10 device certificates at one time. If you selected a CA, make sure you upload only certificates signed by that CA. [Learn more](#).

Existing certificates

You have not selected any device certificates to upload yet.

[Select certificates](#)

[Cancel](#)

[Register certificates](#)

[Done](#)

4. Select the AWS Device Certificate, `aws_device_cert.pem` generated in section 4.4.2. Upload it to the AWS IoT “thing”, and press the **Register certificate** blue button (check the “**Activate all**” radio button)

Register existing device certificates

You can register device certificates signed by your CA certificate. Note that you must first register your CA certificate before uploading device certificates. You can upload up to 10 device certificates at a time.

Existing certificates

[Deactivate all](#) [Revoke all](#)

aws_device_cert.pem



[Remove](#)

[Select certificates](#)

[Done](#)

[Register certificates](#)

5. At this stage, the AWS IoT cloud has a “thing” ready to allow a device to connect to it: the device is registered with an **active certificate**

4.6.2 AWS Console Create Publish/Subscribe Policy

For this example, device needs to be attached to a policy that allows them to subscribe and publish. To accomplish this:

- 1 Create a policy that allows subscription and publishing to a topic such as shown in the example policy below

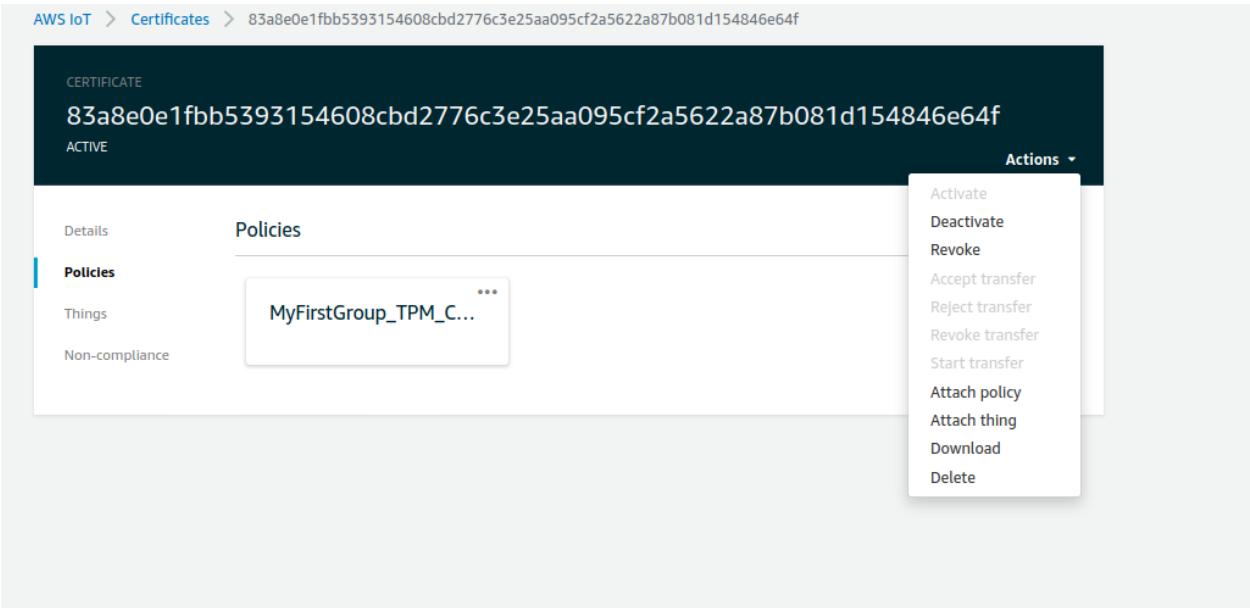
Policy document

The policy document defines the privileges of the request. [Learn more](#)

Version 1 [Edit policy document](#)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Connect",
        "iot:Receive"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:/*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

- 2 Attach the device certificate to the policy, e.g.



4.6.3 Linux Environment: Configure the AWS Example Application that Connects to AWS

To enable and use the OPTIGA™ TPM 2.0 as Hardware Security Integration for Gateway (Device Demo) with AWS

1. Please ensure all certificate and keys generated from [section 4.4.1](#) and [4.4.2](#) are placed inside `/greengrass/certs/`
2. Enable it in the AWS IoT Greengrass config. Edit `/greengrass/config/config.json` and replace the configuration with the content based on your OpenSSL configuration and location of the keys. A complete example of the AWS IoT Greengrass configuration with the setup completed as described in the preceding sections resembles the following:

```
root @ nx8qxpai ~:~# vi /greengrass/config/config.json
{
    "coreThing": {
        "thingArn": "arn:aws:iot:<AWS_REGION>:<AWS_ACCOUNT_NUMBER>:thing/<GG_Thing_Name>",
        "iotHost": "XXXXXXXXXXXXXXXXXX-ats.iot.us-east-1.amazonaws.com",
        "ggHost": "greengrass-ats.iot.ap-south-1.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes"
        }
    },
    "managedReservation": false,
    "crypto": {
        "PKCS11": {
            "OpenSSLEngine": "/usr/lib/engines/pkcs11.so",
            "P11Provider": "/usr/lib/pkcs11/libpnm_pkcs11.so",
            "SlotLabel": "greengrass",
            "SlotUserPin": "1234"
        },
        "privatePaths": {
            "IoTCertificate": {
                "privateKeyPath": "pkcs11:model=SLB9670;manufacturer=rfineon;token=greengrass;object=greenkey;type=private;pin-value=1234",
                "certificatePath": "file:///greengrass/certs/aws_device_cert.pem"
            }
        },
        "caPath": "file:///greengrass/certs/root.ca.pem"
    }
}
```

One can find ARN by navigating to the created thing in AWS IoT -> Manage -> Things -> Select thing and copy the thing ARN in the Details tab as shown below

The screenshot shows the AWS IoT Things console. On the left, there's a navigation sidebar with sections like Monitor, Activity, Onboard (Get started, Fleet provisioning templates), Manage (Things, Types, Thing groups, Billing groups, Jobs, Tunnels), Greengrass (Intro, Groups, Cores, Devices), Secure, Defend, Act, Test, Software, Settings, Learn, and Documentation. The main area shows a 'THING' named 'MyFirstGroup TPM_Core' under 'NO TYPE'. It has tabs for Details, Security, Thing groups, Billing Groups, Shadows, Interact, Activity, Jobs, Violations, and Defender metrics. Under Details, there's a 'Thing ARN' field containing 'arn:aws:iot:us-east-1:123456789012:myFirstGroup TPM_Core' and an 'Edit' button. Below that is a 'Type' field with 'No type' selected.

One can find endpoint by navigating to AWS IoT -> Settings

The screenshot shows the AWS IoT Settings console. The left sidebar is identical to the previous one. The main area is titled 'Settings' and contains a 'Custom endpoint' section. It says 'This is your custom endpoint that allows you to connect to AWS IoT. Each of your Things has a REST API available at this endpoint. This is also an important property to insert when using an MQTT client or the AWS IoT Device SDK.' Below that, it says 'Your endpoint is provisioned and ready to use. You can now start to publish and subscribe to topics.' There's an 'Endpoint' field containing 'arn:aws:iot:us-east-1:123456789012:myFirstGroup TPM_Core' with an 'ENABLED' status indicator.

4.6.4 Lambda Functions on AWS IoT Greengrass

This section will describe how to create and deploy a Lambda function that sends MQTT messages from your AWS IoT Greengrass core device. The module describes Lambda function configurations, subscriptions used to allow MQTT messaging, and deployments to a core device.

Create and Package a Lambda Function

In this step, you will:

- **Download** the AWS IoT Greengrass Core SDK for Python to your computer (and not AWS IoT Greengrass core device) from <https://github.com/aws/aws-greengrass-core-sdk-python/>
- Create a Lambda function deployment package that contains the function code and dependencies.
- Use the Lambda console to create a Lambda function and upload the deployment package.
- Publish a version of the Lambda function and create an alias that points to the version.

1. **Downloaded** the AWS IoT Greengrass Core SDK for Python to your computer.

```
Linux-PC $ git clone https://github.com/aws/aws-greengrass-core-sdk-python.git
Linux-PC $ cd aws-greengrass-core-sdk-python/
```

2. The Lambda function in this module uses:
 - The greengrassHelloWorld.py file in examples/HelloWorld. This is your Lambda function code. Every five seconds, the function publishes one of two possible messages to the hello/world topic.
 - The greengrasssdk folder. This is the SDK.
3. Copy the Greengrass SDK folder into the HelloWorld folder that contains greengrassHelloWorld.py.

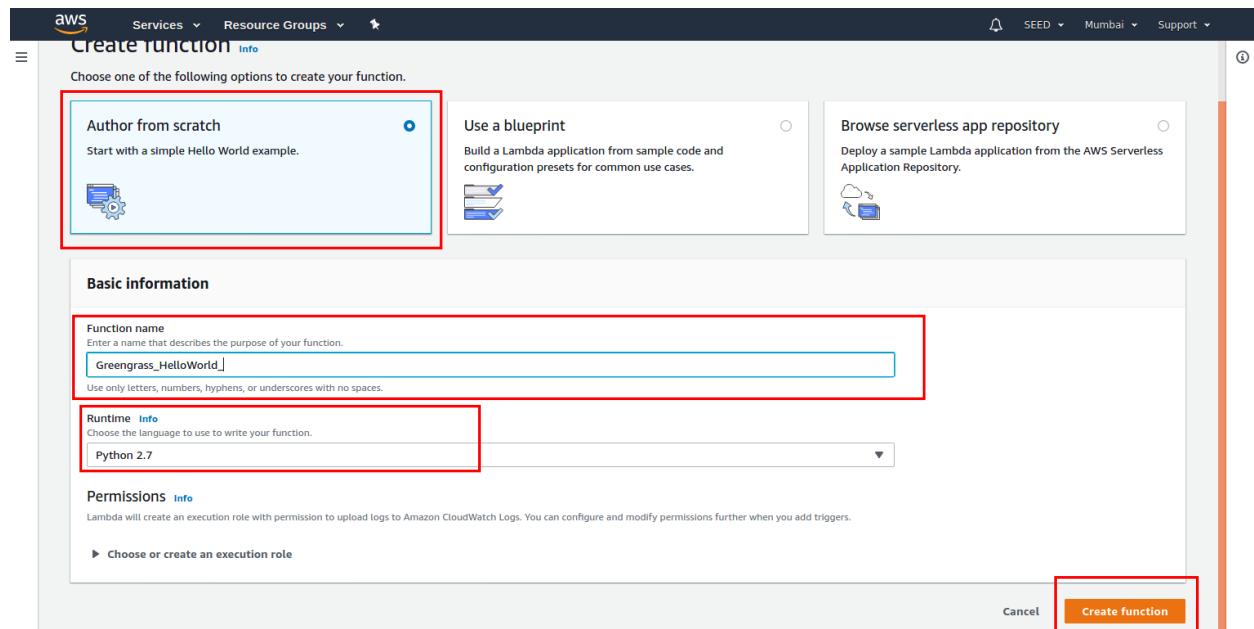
```
Linux-PC $ cp -r greengrasssdk/ examples/HelloWorld/
```

4. To create the Lambda function deployment package, save greengrassHelloWorld.py and the Greengrass SDK folder to a compressed zip file named hello_world_python_lambda.zip. The python file and Greengrass SDK folder must be in the root of the directory.



```
Linux-PC $ cd examples/HelloWorld/
Linux-PC $ zip -r hello_world_python_lambda.zip greengrasssdk/greengrassHelloWorld.py
```

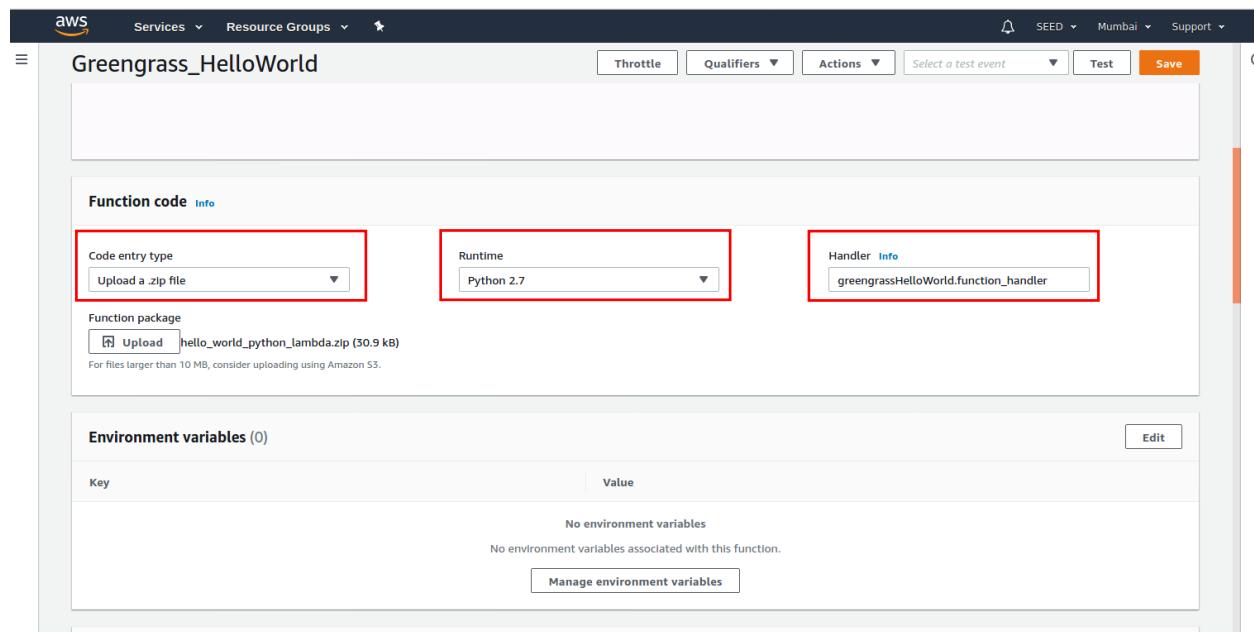
- Open the Lambda console and choose **Create function**.
- Choose **Author from scratch**.
- Name your function **Greengrass_HelloWorld**, and set the remaining fields as follows:
- For Runtime, choose **Python 2.7**.
- Click on **Create function** at bottom.



5. Upload your Lambda function deployment package:

On the Configuration tab, under Function code, set the following fields:

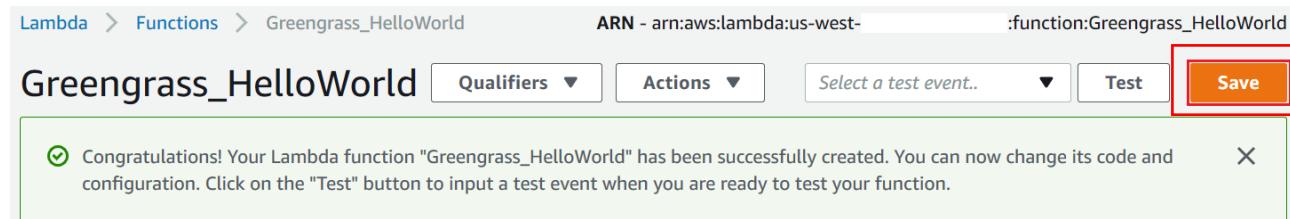
- For Code entry type, choose **Upload a .zip file**
- For Runtime, choose **Python 2.7**
- For Handler, enter **greengrassHelloWorld.handler**



- Choose **Upload**, and then choose **hello_world_python_lambda.zip** (The size of your **hello_world_python_lambda.zip** file might be different from what is shown here.)

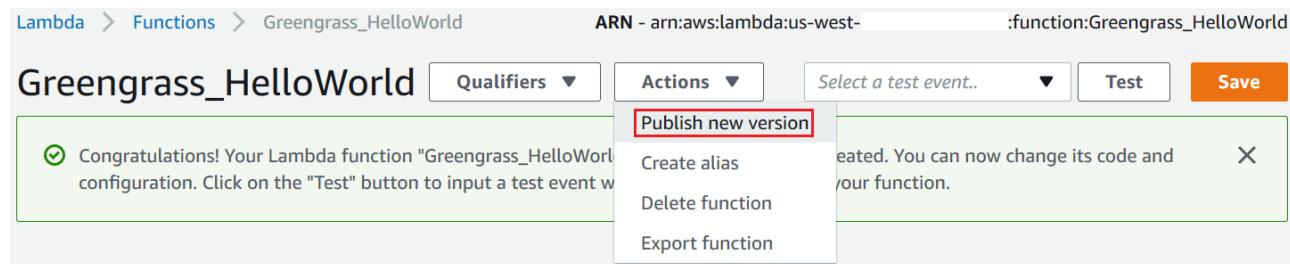
The screenshot shows the AWS Greengrass HelloWorld function configuration interface. At the top, there are tabs for Throttle, Qualifiers, Actions, Select a test event, Test, and Save. The main area is titled "Function code" with an "Info" link. It shows the "Code entry type" as "Upload a .zip file" and the "Runtime" as "Python 2.7". The "Handler" is set to "greengrassHelloWorld.function_handler". A red box highlights the "Function package" section, which contains a "Upload" button and a file entry for "hello_world_python_lambda.zip (30.9 kB)". Below this, a note says "For files larger than 10 MB, consider uploading using Amazon S3.". The "Environment variables" section shows "0" items, with an "Edit" button. It includes a table with columns "Key" and "Value", and a message stating "No environment variables". A "Manage environment variables" button is also present.

2. Choose **Save**

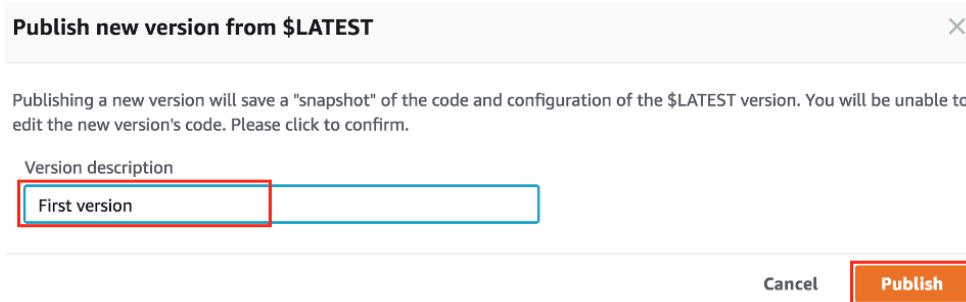


3. Publish the Lambda function:

- From Actions, choose **Publish new version**

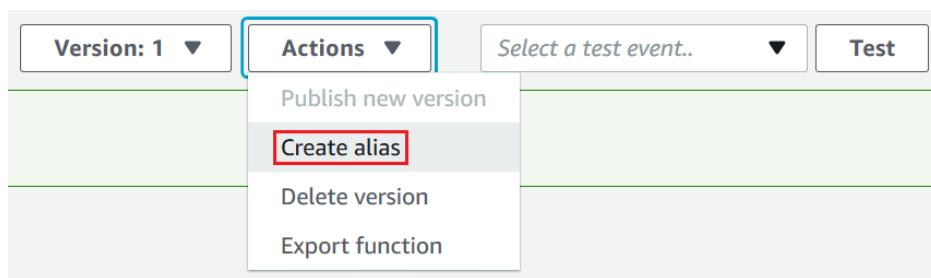


4. For Version description, enter **First version**, and then choose **Publish**



5. Create an alias for the Lambda function version:

- From Actions, choose **Create alias**.



- Name the alias **GG_HelloWorld**, set the version to **1** (which corresponds to the version that you just published), and then choose **Create**.

- Note: AWS IoT Greengrass does not support Lambda aliases for \$LATEST versions.

Create a new alias

An alias is a pointer to one or two versions. Choose each version that you want the alias to point to.

Name*
GG_HelloWorld

Description

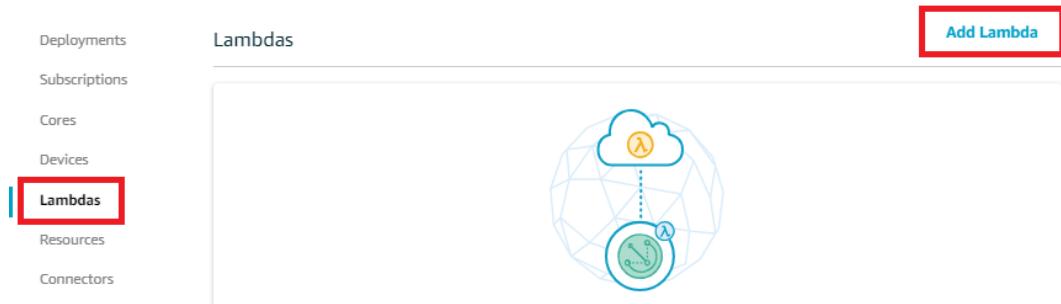
Version*
1

You can shift traffic between two versions, based on weights (%) that you assign. Click [here](#) to learn more.

Additional version

Cancel **Create**

6. In the AWS IoT console under Greengrass, choose Groups, and then choose the group that was created using the steps as mentioned above.
 - On the group configuration page, choose Lambdas, and then choose Add Lambda.



- Choose Use existing Lambda.

Add a Lambda to your Greengrass Group

Local Lambdas are hosted on your Greengrass Core and connected to each other and devices by Subscriptions, but they can also be deployed individually to your Group.

Create a new Lambda function

You will be taken to the AWS Lambda Console and can author a new Lambda function.

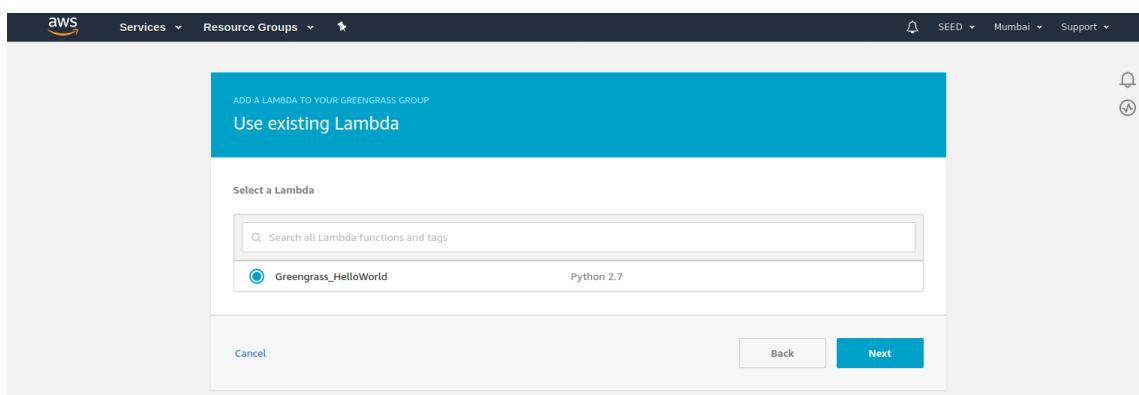
[Create new Lambda](#)

Use an existing Lambda function

You will choose from a list of existing Lambda functions.

[Use existing Lambda](#)

- Search for the name of the Lambda you created in the previous step (Greengrass_HelloWorld, not the alias name), select it, and then choose Next:



- For the version, choose Alias: GG_HelloWorld, and then choose Finish. You should see the Greengrass_HelloWorld Lambda function in your group, using the GG_HelloWorld alias.
- Choose the ellipsis (...), and then choose Edit Configuration.

The screenshot shows the AWS Greengrass Groups console for the 'MyFirstGroup' group. On the left, there's a sidebar with 'Deployments', 'Subscriptions', 'Cores', 'Devices', 'Lambdas' (which is selected), and 'Resources'. The main area shows the group details: 'GREENGRASS GROUP' and 'MyFirstGroup' (status: 'Not deployed'). In the 'Lambdas' section, there's a list with one item: 'Greengrass_HelloWorld' (LAMBDA FUNCTION). To the right of this list is a 'USING' section. A context menu is open over the 'Greengrass_HelloWorld' entry, with options: 'Edit configuration' (which is highlighted with a red box), 'Remove function', and three dots (...).

- On the Group-specific Lambda configuration page, edit the following:

- Set **Timeout** to 25 seconds. This Lambda function sleeps for 20 seconds before each invocation.
- For Lambda lifecycle, choose **Make this function long-lived and keep it running indefinitely**.

Memory limit

16	MB
----	----

Timeout

25	Second
----	--------

Lambda lifecycle

On-demand function

Make this function long-lived and keep it running indefinitely

- Keep the default values for all other fields, such as Run as, Containerization, Input payload data type, and choose Update to save your changes.

Timeout

25	Second
----	--------

Lambda lifecycle

On-demand function

Make this function long-lived and keep it running indefinitely

Read access to /sys directory

Disable

Enable

Input payload data type

JSON

Binary

Environment variables

You can define environment variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code.

Key	Value
e.g. color	e.g. blue

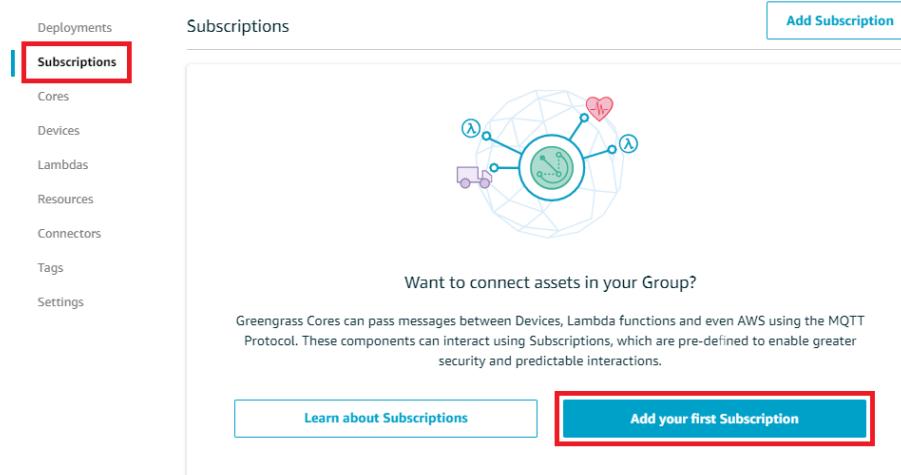
Add another version of this Lambda function

You can add individual historical versions of a Lambda function to your Group, each with their own set of Group-specific configuration, and your Group will treat them as individual and distinct Lambda functions.

[Add another version](#)

[Cancel](#) [Update](#)

- On the **group configuration** page, choose **Subscriptions**, and then choose **Add your first Subscription**.



- In Select a source, choose Select. Then, on the Lambdas tab, choose **Greengrass_HelloWorld** as the source.
9. To Select a the Service then choose target, choose Select. Then, on tab choose **IoT Cloud**, and next.

- For Topic filter, enter `hello/world`, and then choose Next.

Source
Greengrass_HelloWorld LAMBDA

Topic filter
hello/world [How do I enter a topic filter?](#)

Target
IoT Cloud SERVICE

[Back](#) **Next**

- Click on **Finish**

CREATE A SUBSCRIPTION

Confirm and save your Subscription

Your Subscription is complete and your objects are connected in this Group. You can now save, and then deploy your new Group definition to have this change take effect.

Greengrass_HelloWorld LAMBDA

hello/world

IoT Cloud SERVICE

[Back](#) **Finish**

10. Configure the group's logging settings. User can configure AWS IoT Greengrass system components and user-defined Lambda functions to write logs to the file system of the core device.
 - On the group configuration page, choose **Settings**.
 - For **Local Logs configuration**, choose **Edit**.
 - On the **Configure Group Logging** page, choose **Add another log type**.
 - For event source, choose **User Lambdas and Greengrass system**, and then choose **Update**.
 - Keep the default values for logging level and disk space limit, and then choose **Save**.
 - **Disable** the Stream Manager Status.

4.6.5 On Board: Execute the AWS Example Application

To check whether the daemon is running:

```
root @ mx8qxpai nh:~# ps aux | grep -E 'Greengrass.*daemon'
```

If the output contains a root entry for /Greengrass/ggc/packages/1.10.0/bin/daemon, then the daemon is running.

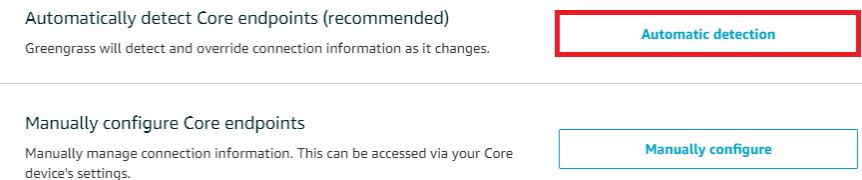
- To start the daemon:

```
root @ mx8qxpai nh:~# cd/greengrass/ggc/core/
root @ mx8qxpai nh:~# ./greengrassd start
```

- In the AWS IoT console, on the [group configuration page](#), from Actions, choose Deploy.



- Choose [Automatic detection](#) to allow devices discover your core page. This enables devices to automatically acquire connectivity information for the core, such as IP address, DNS, and port number. Automatic detection is recommended, but AWS IoT Greengrass also supports manually specified endpoints. You are only prompted for the discovery method for the first time when the group is deployed.



- The first deployment might take a few minutes. When the deployment is complete, you should see successfully completed in the status column on the Deployments page.

The screenshot shows the AWS Greengrass Group 'MyFirstGroup_TPM' interface. At the top, it says 'Successfully completed'. Below that is a table titled 'Group history overview' with columns 'Deployed', 'Version', and 'Status'. The 'Status' column is highlighted with a red box. The table lists several deployment entries, each with a status indicator (green for successful, red for failed) and a 'More details' link.

Deployed	Version	Status
Apr 20, 2020 6:59:45 PM +0530	a6a22fc8-f518-4a48-a147-00be9b1157ee	Successfully complet...
Apr 20, 2020 3:26:48 PM +0530	a6a22fc8-f518-4a48-a147-00be9b1157ee	Successfully complet...
Apr 20, 2020 3:18:33 PM +0530	baf99c60-b526-4650-8e8f-df447ddbb2fc	Failed
Apr 20, 2020 3:10:02 PM +0530	1f2a8916-3a07-4c4a-b7ad-1a270f8d96b0	Successfully complet...
Apr 20, 2020 2:46:19 PM +0530	2e1a9f5e-c7e4-487b-b451-9243d9630596	Successfully complet...
Apr 20, 2020 12:51:46 PM +0530	2e1a9f5e-c7e4-487b-b451-9243d9630596	Successfully complet...
Apr 20, 2020 12:29:03 PM +0530	2e1a9f5e-c7e4-487b-b451-9243d9630596	Failed
Apr 20, 2020 12:02:25 PM +0530	a9c0543f-cb37-47a6-bd1e-87a360ff6df0	Failed

5. Verify that the Lambda Function is running on the Core Device with HW Security enabled with OPTIGA™ TPM 0 security keys.
6. From the navigation pane of the AWS IoT console, choose **Test**.



Monitor

Onboard

Manage

Greengrass

Secure

Defend

Act

Test

7. Choose **Subscribe to topic**, and configure the following fields:
 - For **Subscription topic**, enter **hello/world** (Do not choose Subscribe to topic yet.)
 - For **Quality of Service**, choose **0**
 - For MQTT payload display, choose **Display payloads as strings**.
 - Click on "Choose Subscribe to topic"

Subscriptions

Subscribe to a topic

Publish to a topic

Subscription topic hello/world **Subscribe to topic**

Max message capture 100

Quality of Service 0 - This client will not acknowledge to the Device Gateway that messages are received
1 - This client will acknowledge to the Device Gateway that messages are received

MQTT payload display Auto-format JSON payloads (improves readability)
Display payloads as strings (more accurate)
Display raw payloads (in hexadecimal)

[Note: Assuming the Lambda function is running on your device, it publishes messages similar to the hello/world topic]

8. Display MQTT messages on the screen like "Message from AI_ML"

AWS IoT

MQTT client

Connected as iotconsole-1587374243150-0

Subscriptions hello/world **Export** **Clear** **Pause**

Subscribe to a topic

Publish to topic

hello/world **Publish to topic**

hello/world Apr 20, 2020 2:52:12 PM +0530 **Export** **Hide**

```
{
  "request": {
    "message": "Hello from AWS IoT console"
  },
  "id": "req_123"
}
```

hello/world Apr 20, 2020 2:52:07 PM +0530 **Export** **Hide**

```
{
  "request": {
    "message": "Message from IMX8-AI_ML"
  },
  "id": "req_123"
}
```


5 | MX8X SECURE BOOT

5.1 Preparing the environment to build a secure boot image

```
Li nux-PC $ cd /lib/dkms/and-alin/tmp/deploy/images/mx8qxplatform/
```

```
Li nux-PC $ cp mx8qx-ahab-container.img Secureboot_AL_M/.
```

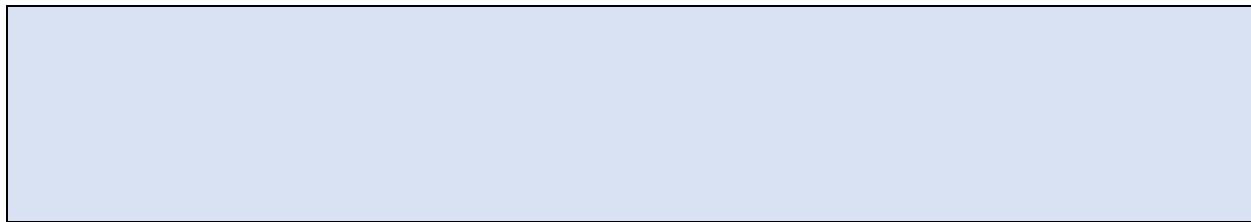
```
Li nux-PC $ cd Secureboot_AL_M/.
```

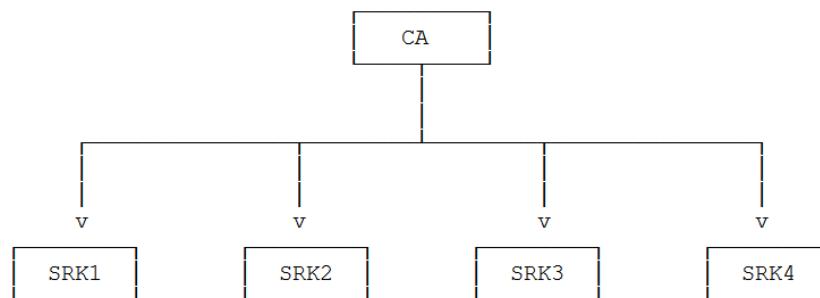
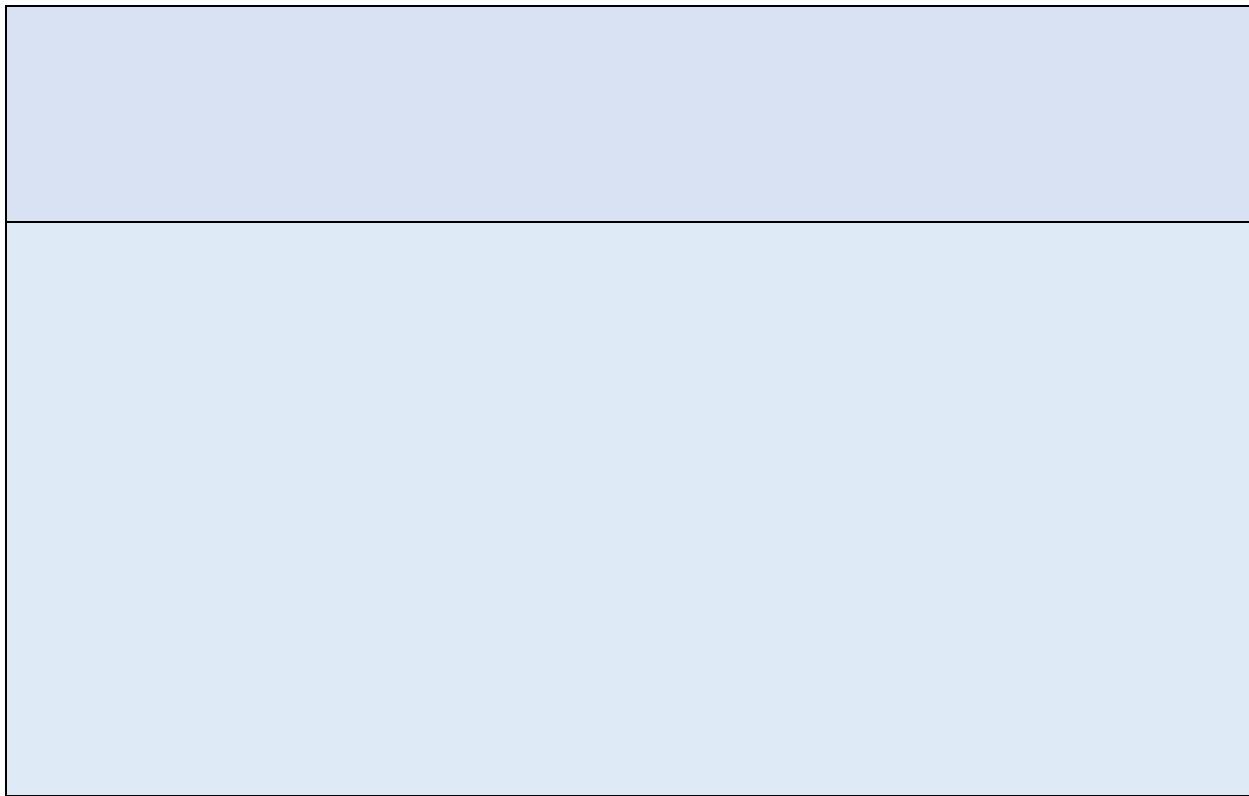
```
Li nux-PC $ mv mx8qx-ahab-container.img ahab-container.img
```



5.2 Building a Secure Signed Image

5.2.1 Programming SRK Hash

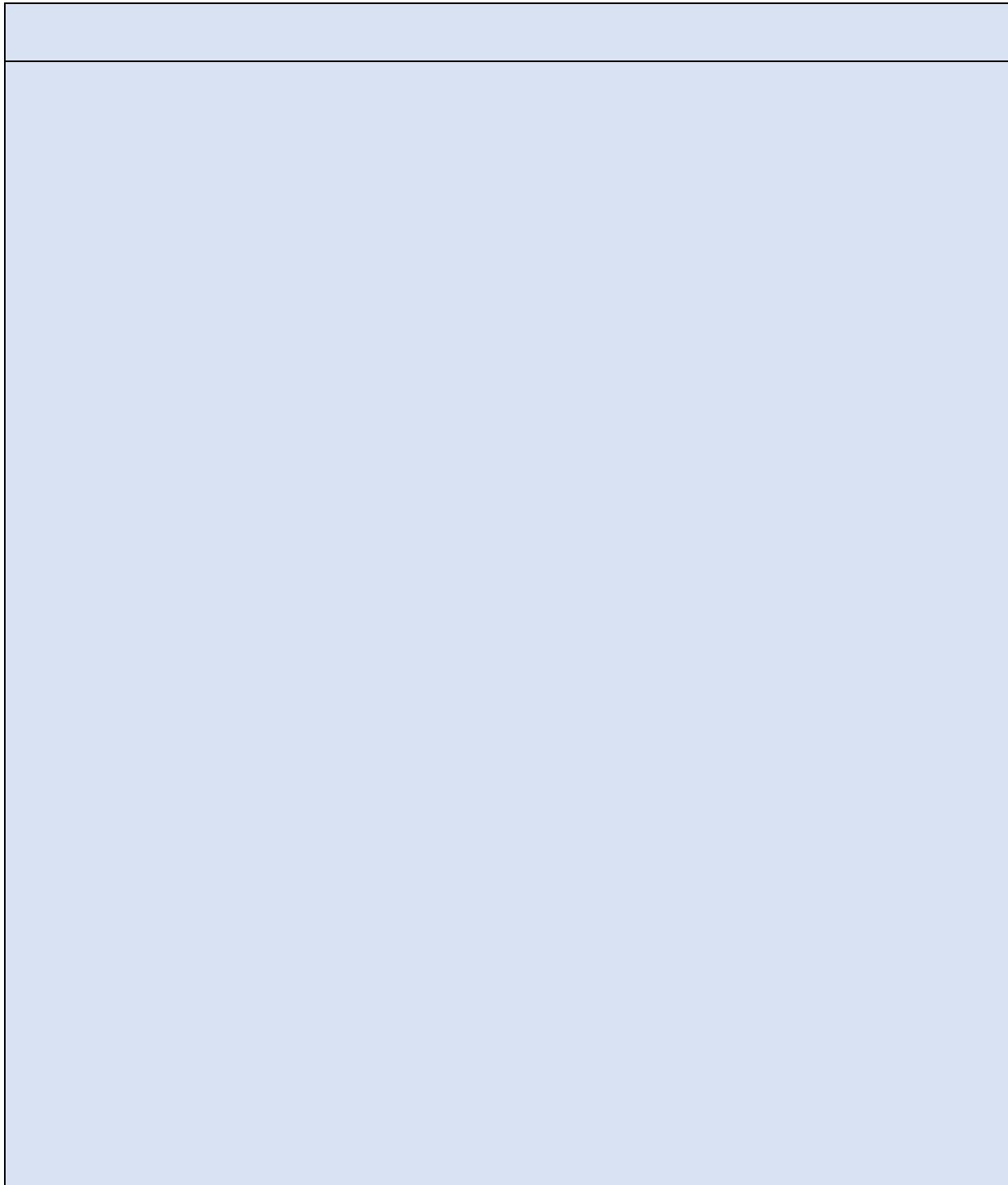






5.2.2 Prepare the boot image layout





```
+ Offsets = 0x400      0x590
```

5.2.3 Signing the boot image

Now you use the CST to generate the signed boot image from the previously created csf_boot_image.txt commands sequence file

```
Li nux-PC $ cd Secureboot_AM/  
Li nux-PC $ ./rel ease/li nux64/bin/cst -i csf_boot_i mage.txt -o flash signed.bin  
Li nux-PC $ scp flash signed.bin <user name> @<IP Addr>:/
```

5.2.4 Flash the signed image

5.2.5 Prepare the OS container image

```
Li nux-PC $ sudo mount /dev/sdX1 partition
```

SCFWConsde

SECO Event[1] = 0x0087EE00 [The container image was signed with wrong key which are not matching the OTP SRK hashes]

5.2.6 Dumping SRK Hash fuse values on HOST machine

The SRK Hash fuse values are generated by the srktod and should be programmed in the SoC SRK_HASH [511:0] fuses.

Be careful when programming these values, as this data is the basis for the root of trust. An error in SRK Hash will result in parts that will not boot.

The U-Boot fusetod can be used for programming eFuses on i.MX SoCs.

- Dump SRK Hash fuses values on HOST machine:

```
Linux-PC $ od -t x4 SRK_1_2_3_4_fuse.bin  
imx8 @V1.43: crt$ od -t x4 SRK_1_2_3_4_fuse.bin  
00000000 e59aaa4e 97253333 f783083f 7c10bbb3  
00000020 8febb2d1 4bf36109 95815423 7c6599af  
00000040 ff32c663 4c7b8c5d bb127029 d101fe1d
```

```
0000060 be36af12 37472c1b 7ad6d58c 21253de3  
0000100
```

- Switch to uboot console and Program SRK_HASH[511:0] fuses:

```
[U-Boot] Normal Boot  
[U-Boot] Hit any key to stop autoboot: 0  
=>  
=>fuse prog 0 730 0xe59aaa4e  
=>fuse prog 0 731 0x97253333  
=>fuse prog 0 732 0xf783083f  
=>fuse prog 0 733 0x7c10bbb3  
=>fuse prog 0 734 0x8febb2d1  
=>fuse prog 0 735 0x4bf36109  
=>fuse prog 0 736 0x95815423  
=>fuse prog 0 737 0x7c6599af  
=>fuse prog 0 738 0xff32c663  
=>fuse prog 0 739 0x4c7b8c5d  
=>fuse prog 0 740 0xbb127029  
=>fuse prog 0 741 0xd101fe1d  
=>fuse prog 0 742 0xbe36af12  
=>fuse prog 0 743 0x37472c1b  
=>fuse prog 0 744 0x7ad6d58c
```

5.2.7 Verify SECO events

5.2.8 Close the device



5.3 Measured Boot with OPTIGA™ TPM2.0

5.3.1 Measured boot Step to verify Platform Integrity

5.3.2 Measuring Kernel Image Hash

Once the Board is booted and Rootfs is mounted on it, a measure of kernel Image will be taken

```
root@mx8qxpai-nh:~# $ sha256sum /boot/uimage | cut -d ' ' -f1 >> kernel_hash
```

5.3.3 Extending Measured Hash to PCR

Calculated Kernel Image hash will be extended to PCR for Measurement storage.

```
root@mx8qxpai-nh:~# $ tpm2_pcr_extend 16:sha256=$kernel_hash
```

5.3.4 Measuring Content from Specified PCR

Taking the stored value from PCR in order to store in NV-area

```
root@mx8qxpai-nh:~# $ tpm2_pcrlist -L sha256:16 -o pcr_kernel_origin.bin
```

5.3.5 Generating TPM Measured Boot Policy

Generating a TPM based policy in order to store PCR based architecture value into NV-area

```
root@mx8qxpai-nh:~# $ tpm2_createpolicy --policy-pcr -L sha256:16 -F pcr_kernel_origin.bin -o policy_pcr_kernel_origin.out
```

5.3.6 Define NV-Area in TPM for PCR Storage

Create an NV-area in order to store the PCR value and specify the policy to it

```
root @ mx8qxpai nh:~# $ tpm2_nvdefine -x 0x1500016 -a 0x40000001 -s 32 -L
policy_pcr_kernel_origin.out -b
"policyread|policywrite|authread|authwrite|ownerwrite|ownerread"
```

5.3.7 Extending PCR value to Secure NV RAM

Storing the PCR value, the Specified NV-index is defined

```
root @ mx8qxpai nh:~# $ tpm2_nvwrite -x 0x1500016 -a 0x1500016 -P pcr:sha256:16
pcr_kernel_origin.bn
```

5.3.8 Verifying Platform Integrity Check

Again, reboot the board and perform steps

```
root @ mx8qxpai nh:~# $ sha256sum/boot/image | cut -d ' -f1 >> Measure_kernel_hash
```

```
root @ mx8qxpai nh:~# $ tpm2_pcrextend 16:sha256=$kernel_hash
```

```
root @ mx8qxpai nh:~# $ tpm2_pcrlist -L sha256:16 -o pcr_kernel_measured.bn
```

```
root @ mx8qxpai nh:~# $ tpm2_nvread -x 0x1500016 -a 0x1500016 -s 32 >> pcr_kernel_origin.bn
```

```
root @ mx8qxpai nh:~# $ cmp pcr_kernel_measure.bn pcr_kernel_origin.bn
```

If values are the same, this verifies the Platform is secure and no tamper has occurred

6 APPENDIX

6.1 A_ML-96 Boards

This A_ML Board is MK8QuadXPi based on 96Boards™ specification. It features the NXP® i.MX 8QXP processor with advanced implementation of the Quad Arm Cortex®-A35+ Arm Cortex®-M4 core, operating at speeds up to 1.2 GHz. Each processor provides a 32-bit DDR3L/LPDDR4 memory interface and other interfaces for connecting peripherals, such as WLAN, Bluetooth™, GPS and camera sensors.

96Boards (<http://www.96boards.org>) is a 32-bit and 64-bit ARM® Open Platform hosted by Linaro™ with the intention to serve the software maker and embedded OEM communities.

Processor

- NXP i.MX 8X Processor
- Quad-core ARM® Cortex® A35 at up to 1.2 GHz per core 64-Bit capable
- 4x Vec4 shaders with 16 execution units optimized for higher performance
- Supports OpenGL 3.0_21; OpenGL ES 3.1, 3.0, 2.0, and 1.1; OpenCL 1.2 Full Profile and 1.1; OpenVG 1.1;
- Vulkan High performance 2D Bit Engine

Memory/Storage

- 2GB LPDDR4 1600 MHz
- SD 3.0 (UHS-I)

I/O Interfaces

- Two USB 3.0 Type A connector and one USB 2.0 Micro B
- One 40-pin Low Speed (LS) expansion connector (UART, SPI, I2S, I2C x2, GPIOx12, DC power)
- One 60-pin High Speed (HS) expansion connector (4L-MIPI-DSI, USB, I2C x2, 4L-MIPI-CSI, 1-SPI)
- The board can be made compatible as an add-on mezzanine board

Connectivity

- Bluetooth 4.2 (Bluetooth Low Energy)
- High performance 2.4 GHz and 5 GHz WLAN
- Ethernet support 10/100/1000 Mbps speed

Camera Support

- One 4-lane MIPI-CSI

Video

- H.265 decode (4Kp30)
- H.264 decode (1080p60)
- VP6/VP8 decode (1080p60)
- MPEG-2 decode (1080p60)
- MPEG4, H263, Sorenson Spark decode (1080p)
- Real Video decode (1080p)
- JPEG dec (64Kx64Ki mage size)

- H.264 encode (1080p30)
- HDMI

Power, Mechanical and Environmental

- Power: +8.0V to +18V
- Dimensions: 85 mm x 100 mm
- 96Boards™ Consumer Edition standard dimensions specifications
- Operating Temp: 10°C to +70°C
- RoHS compliant

Software

Yocto based Linux distribution 4.14GA release

6.2 AWS Greengrass

AWS IoT Greengrass software extends cloud capabilities to local devices.

Cloud-based management of application logic

- to collect and analyze data
- react automatically to local events
- communicate securely on local networks.
- AWS Lambda functions and pre-built connectors to create serverless applications that are deployed to devices for local execution
- provides a local pub/sub message manager that can intelligently buffer messages to preserve inbound and outbound messages to the cloud in case there is no connectivity to cloud

The following diagram shows the basic architecture of AWS IoT Greengrass.



Figure 3: AWS Greengrass Group

AWS IoT Greengrass core software provides the following functionality:

- Deployment and local execution of connectors and Lambda functions.
- Process data streams locally with automatic exports to the AWS Cloud
- MQTT messaging over the local network between devices, connectors, and Lambda functions using managed subscriptions.
- MQTT messaging between AWS IoT and devices, connectors, and Lambda functions using managed subscriptions.
- Secure connections between devices and the AWS Cloud using device authentication and authorization.
- Local shadow synchronization of devices. Shadows can be configured to sync with the AWS Cloud
- Secure, encrypted storage of local secrets and controlled access by connectors and Lambda functions.
- Automatic IP address detection that enables devices to discover the Greengrass core device

6.3 Tresor Mezzanine OPTIGA™ TPM 2.0

The Tresor Mezzanine Board provides state-of-the-art secure elements to 96Boards host board. The board is shown in Figure 2.

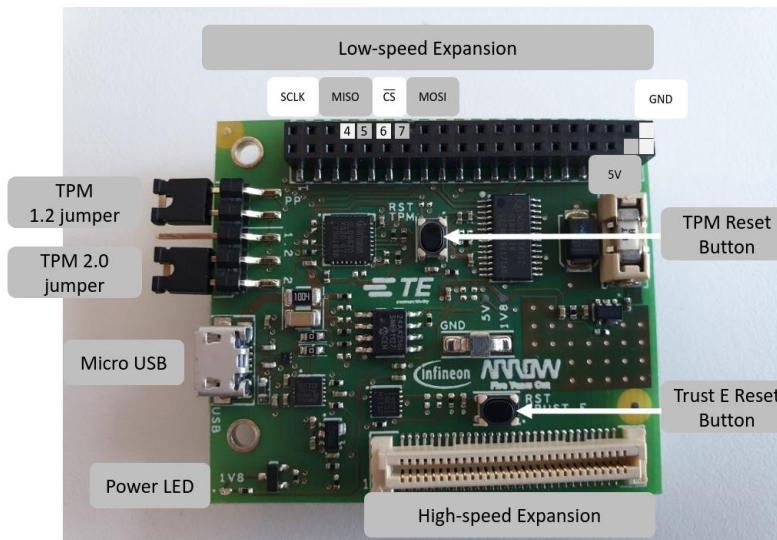


Figure 4: Tresor Mezzanine OPTIGA™ TPM M2.0

The board is equipped with three separate chips that can provide security features

- The SLB9670x provides OPTIGA™ Trusted Platform Module (TPM) 2.0 functionality through SPI communication on the standard 96Boards Low speed expansion connector.
- The SLB9645x TPM 1.2 chip communicates via I2C on the standard 96Boards Low speed expansion connector.
- The SLS32AIA020A TRUST-E authentication chip shares the same I2C bus with the TPM 1.2 module.

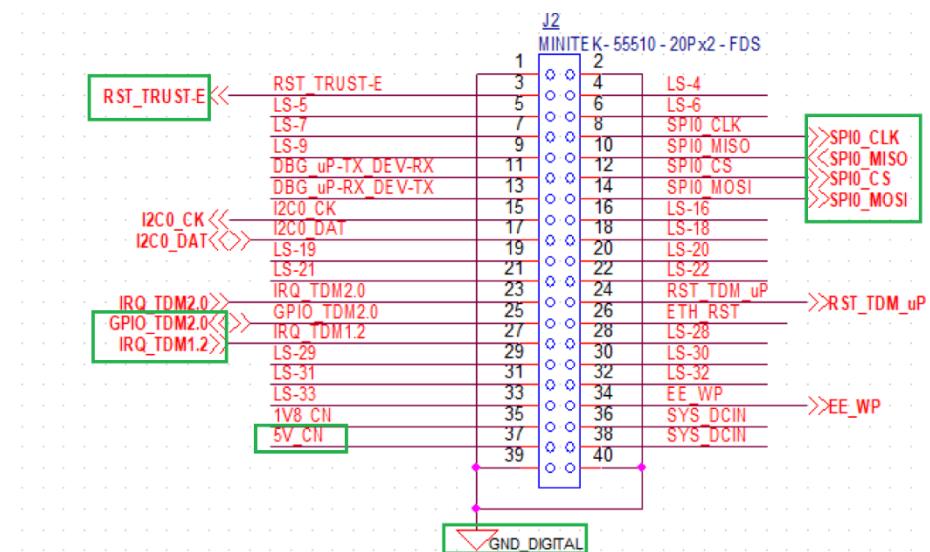


Figure 5: Tresor Mezzanine connector OPTIGA™ TPM 2.0

6.4 Trusted Platform Module – TPM

A cryptographic processor is present on most commercial PCs and servers.

It will typically have three key cryptographic capabilities

- Establishing a root of trust
- Secure boot
- Device identification

Establishing a root of trust

A TPM can prevent a bootkits attack by providing a trusted sequence of boot operations.

The following questions often arise in a running system

- Is the operating system that is running appropriately secure?
- Is the firmware booting the OS appropriately secure?
- Is the underlying hardware appropriately secure?

Each layer has to trust the layer below as illustrated in the following diagram.

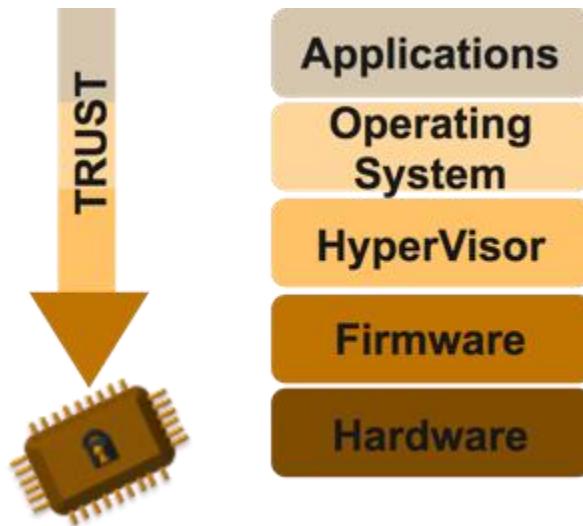


Figure 6: Root of Trust

At the root of this chain is the hardware, which must be inherently trusted. It forms the base on which the chain of trust has been established.

A root of trust is all of the following:

- Set of functions in a trusted computing module that is always trusted by the firmware/OS
- Prerequisite for secure boot process
- Component that helps in detection of boot kits

Secure boot

A secure boot builds on the underlying notion of a root of trust to protect the boot process from being compromised on the device.

In case a chain of trust is broken, the boot process is aborted and the device attempts to go back to its last known good state. An extension to secured boot process is a measured boot – where the device does not halt the boot process. Instead, it records the identity of each component that participates in the boot process so that the component identities can be verified against a list of approved component identities for that device. This is called a measured boot.

These two processes are illustrated in the following diagram

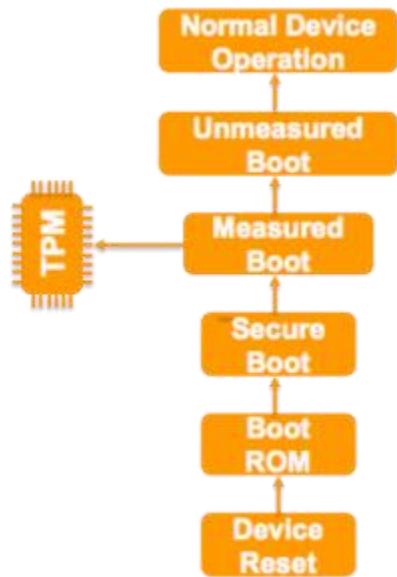


Figure 7: Measured/Trusted Boot Process

A typical sequence of a measured boot is as follows:

- The boot ROM acts as the Root of Trust.
- Upon a device reset, each image that forms part of the boot sequence is validated (measured) before execution.
- The measurements are stored in a TPM.
- Each measurement serves as the proxy for the Root of Trust for the subsequent step in the boot sequence.
- Normally, only critical and security-sensitive process and configuration files are considered for the measurement.
- After the security-sensitive processes are completed, the device enters the unmeasured boot stage before entering normal system operation state.

Device identification

- Check the identity of the device that is communicating with the messaging gateway.
- Generate key pairs for the devices, which are then used to authenticate and encrypt the traffic.
- TPM stores the keys in a tamper-resistant hardware.
- The keys are generated using TPM itself and are thereby protected from being retrieved by external programs.

The rest of this post focuses on how to integrate and use features of TPM to protect the edge gateways running AWS IoT Greengrass. This integration uses the PKCS#11 protocol as the interface to the TPM.

6.5 i.MX8X Secure Boot Overview

6.5.1 Secure AHAB boot architecture

The AHAB secure boot feature relies on digital signatures to prevent unauthorized software execution during the device boot sequence. In case a malware takes control of the boot sequence, sensitive data, services and network can be impacted.

The AHAB authentication is based on public key cryptography in which image data is signed offline using one or more private keys. The resulting signed image data is then verified on the i.MX processor using the corresponding public keys. The public keys are included in the final binary and the SRK Hash is programmed in the SoC fuses for establishing the root of trust.

On i.MX 8 and i.MX 8X families, the SCU is responsible to interface with the boot media, managing the process of loading the firmware and software images in different partitions of the SoC. The SECO is responsible to authenticate the images, authorizing the execution of them.

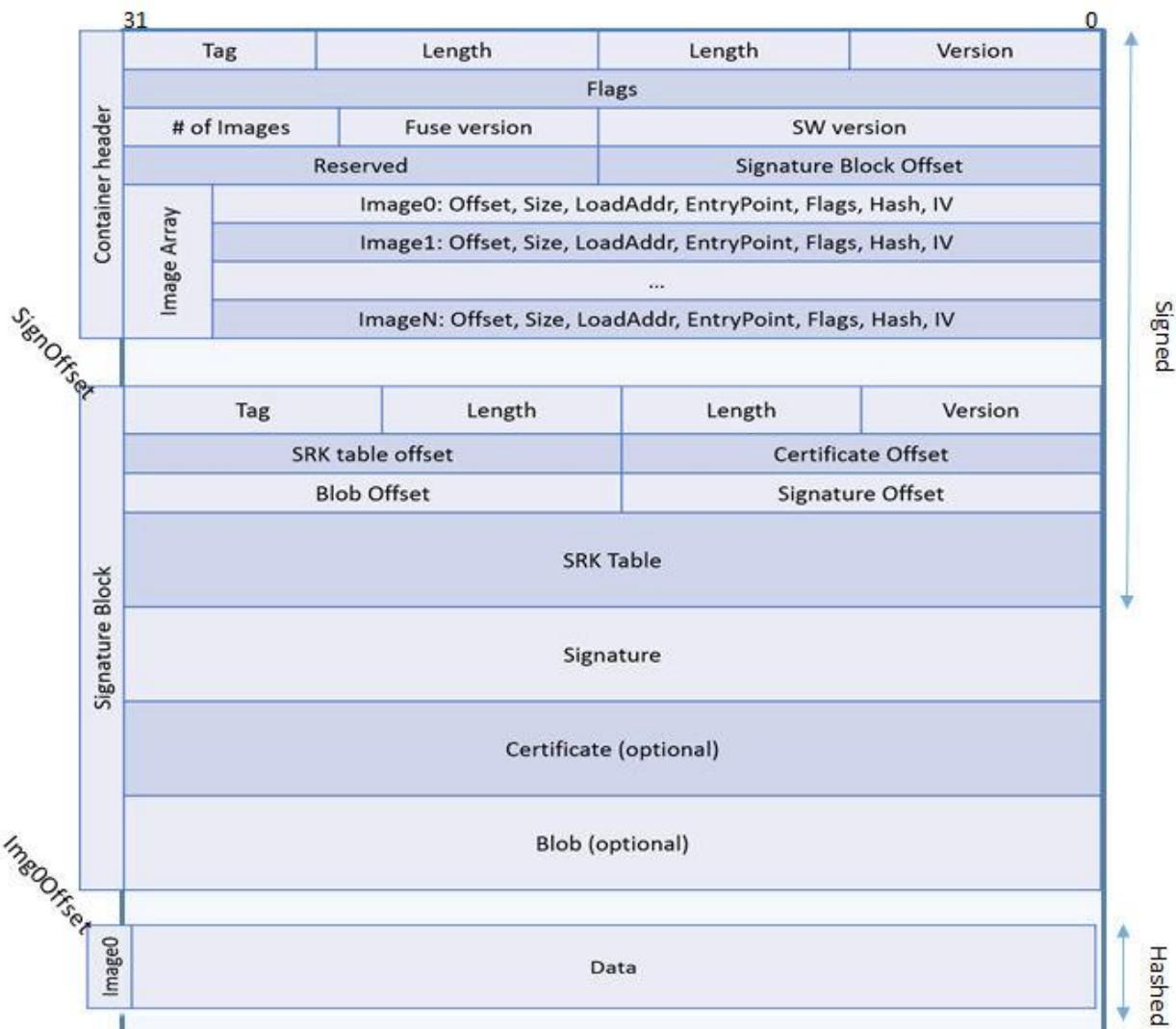


Figure 8: Container layout

The SRK Table is generated with the SRK Tool, provided with the Code Signing Tool (CST).

6.5.1.1 The System Control Unit (SCU)

The System Control Unit SCU is a subsystem equipped with a programmable MCU core, which is responsible to handle the resource allocation, power, docking, configuration and muxing.

The SCU is also responsible to interface between the rest of the system. In the secure boot flow the SCU interfaces with the Security Controller (SEC0), requesting the image authentication.

The System Control Unit FW(SCF FW) is responsible to control all the functionalities of the SCU. This firmware is distributed in a porting kit form. Instructions to download the SCF FW Porting Kit are available in the Linux BSP Release Notes.

Details about SCU can be found in the processors Reference Manual (RM).

6.5.1.2 The Security Controller (SECO)

The SECO is a M0+ core dedicated to handle the SoC security subsystem. The controller communicates with SCU domain through a dedicated message unit (MU).

The SECO has a dedicated ROM which is responsible to initialize low level security features and to authenticate the SECO firmware previously loaded by the SCU ROM.

The SECO firmware provides security services at runtime to different domains of the SoC, one of these being the capability to authenticate images.

The SECO firmware is signed and distributed by NXP and is always authenticated in OEM open and closed configuration, instructions to download the SECO FW are available in the Linux BSP Release Notes.

Details about SECO can be found in the processors Security Reference Manual (SRM).

6.5.1.3 The Image Container

Due to the new architecture, multiple firmwares and softwares are required to boot i. MK8 and i. MK8x family devices.

In order to store all the images in a single binary the container image structure is used.

At least two containers are needed for the boot process, the first container must include only the SECO FW (provided by NXP). Additional containers can contain one or multiple images, depending on the user's specific application.

The final binary is generated by the mx-nki mage tool. The tool can generate additional containers and combine all containers in a single binary.

6.5.1.4 Secure boot flow

Due to the multicore architecture, the i. MK8 boot sequence involves SCU ROM, SCU Firmware, SECO ROM and SECO FW.

Figure 9 below illustrates the secure boot flow overview.

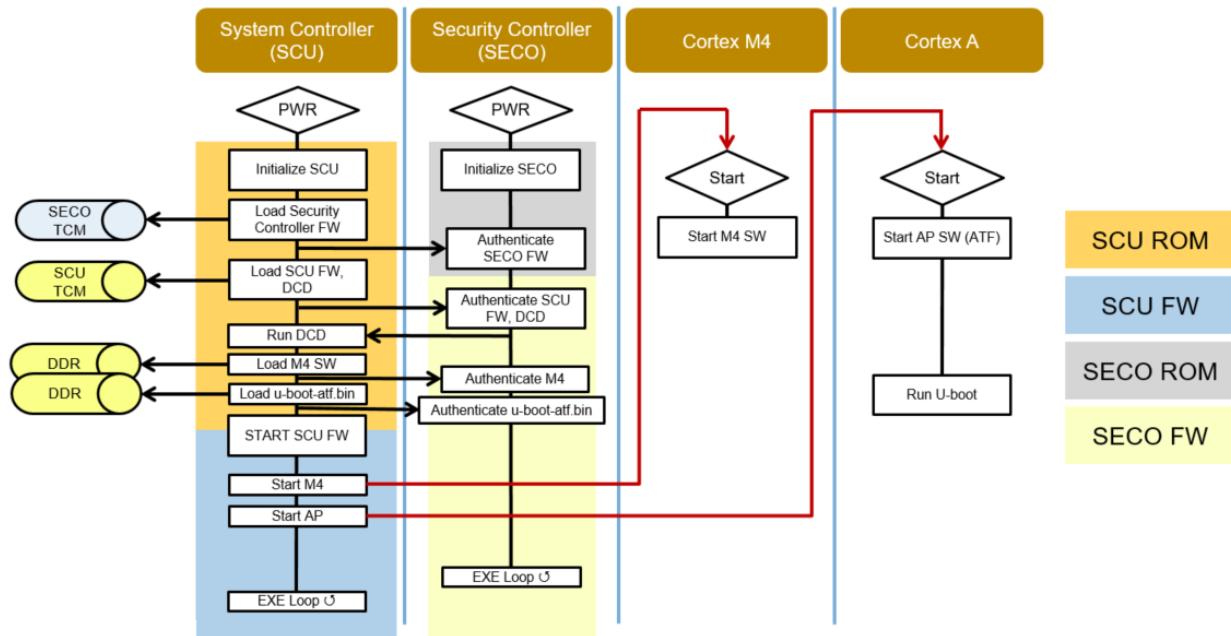


Figure 9. Secure boot flow overview

The i. MK8 and i. MK8x boot flows as follows.

1. At reset, the SCU ROM and SECO ROM both start execution.
2. The SCU ROM reads the boot configuration and loads the SECO FW (first container) from the boot media to the SECO TCM.
3. A message is sent by the SCU ROM via MJ requesting the SECO ROM to authenticate the SECO FW which is signed using NXP key.
4. The SCU ROM loads the second container from the boot media, this container must contain at least the SCFW which is signed using the OEM keys.
5. The SCU ROM loads the SCFW to the SCU TCM, a message is sent via MJ requesting the SECO FW to authenticate the SCU FW and DCD table.
6. The SCU ROM configures the DDR and loads the M4 and AP images to their respective load addresses.
7. The SCU ROM requests the SECO FW to authenticate the M4 image.
8. The SCU ROM requests the SECO FW to authenticate the AP image.
9. The SCU FW is initialized and starts the Arm® - Mand Cortex-A cores.

After each authentication, SECO FW returns a success or failure status to SCU.

If SCU receives a fail response from SECO FW authentication while attempting to boot from the primary boot source, the SCU will attempt to boot from the secondary boot source (if any).

If SCU receives a fail response from SECO FW authentication while attempting to boot from the secondary boot source, the SCU will go into recovery mode.

If the SCU receives a fail response for the second container, the SCU will enter the recovery mode.

6.5.1.5 AHAB Secure Boot Process Overview

The boot image is composed of different layers, as shown in Figure 3 below



Figure 10: Secure boot image layout

The boot image contains two containers, one for the SECO firmware (AHAB), and one for the SCFW with the ATF, U-Boot and M4 image. They are preceded by their headers. The first one, containing the SECO firmware image, is padded to 0x1000 to fix the start address of the second one, which can contain one or multiple images.

NOTE

The only required images for the device are the SECO FW and the SCFW. The Cortex-A or Cortex-M images may or may not be part of it depending on the OEM system design.

In contrast with the secure boot process used in the HABv4 architecture, there is no need for CSF in this architecture. The CST is responsible to handle the signature block, as shown in Figure 4 below.

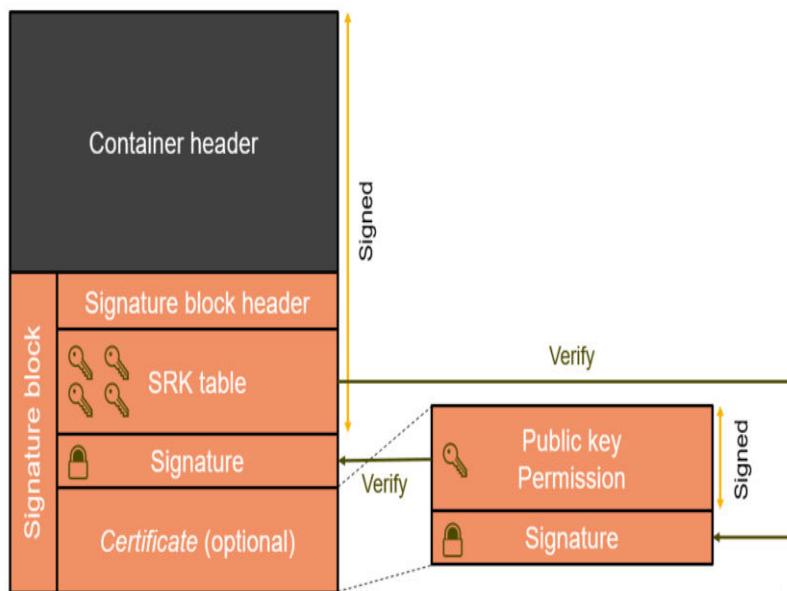
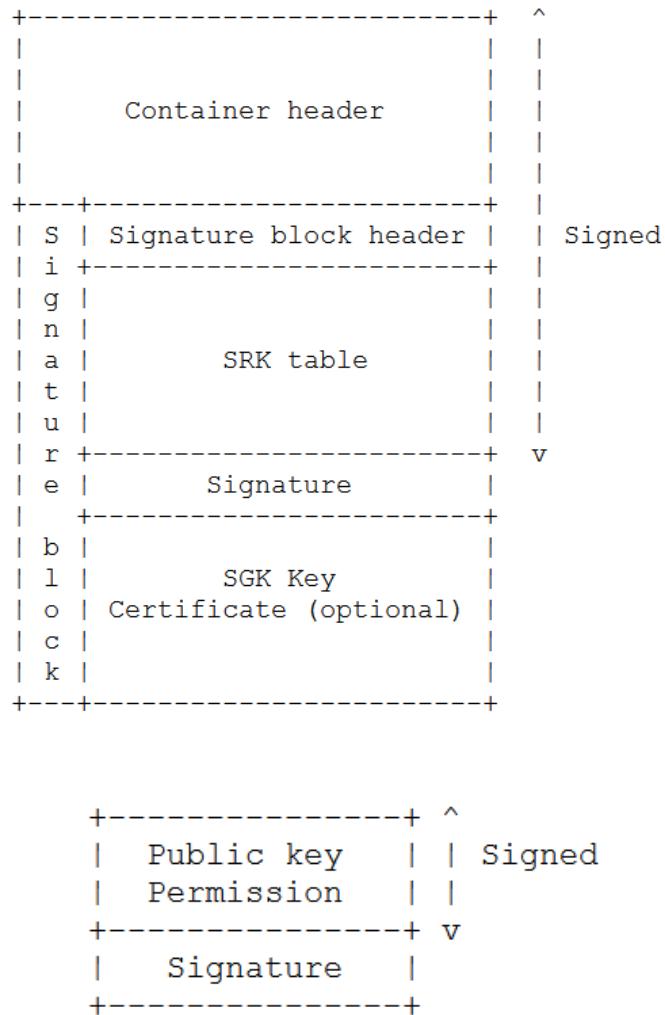


Figure 11: Signature block

The container signature is verified against the SGK key certificate, which is then verified against the SRK table. If the subordinate key is not used, the container signature is directly verified against the SRK keys.

6.5.1.6 Architecture and image supporting secure boot

```
+-----+ <-- *start
| 1st Container header |
| and signature         |
+-----+
| Padding for 1kB alignment |
+-----+ <-- *start + 0x400
| 2nd Container header |
| and signature         |
+-----+
| Padding               |
+-----+
| SECO FW               |
+-----+
| Padding               |
+-----+
| SCU FW with DDR      |
| initialization Image |
| embedded              |
+-----+
| Cortex-M4 Image      |
+-----+
| Cortex-A bootloader   |
+-----+
```



The first block (public key per mission) verify the Signature block preceding (between SRK table and Certificate blocks), while the second block (signature) is verified by the SRK table block.

6.6 Measured boot Principles

Measuring boot is a way to inform the last software stage if someone tampered with the platform. It is impossible to know what has been corrupted exactly, but knowing someone has is already enough to not reveal secrets. Indeed, TPMs offer a small secure locker where users can store keys, passwords, authentication tokens, etc. These secrets are not exposed anywhere (unlike with any standard storage media) and TPMs have the capability to release these secrets only under specific conditions. Here is how it works.

Starting from a *root of trust* (typically the SoC Boot ROM), each software stage during the boot process (BL1, BL2, BL31, BL33/U-Boot, Linux) is supposed to do some measurements and store them in a safe place. A *measure* is just a digest (let's say, a SHA256) of a memory region. Usually **each stage will 'digest' the next one**. Each digest is then sent to the TPM which will *merge* this measurement with the previous ones.

The hardware feature used to store and merge these measurements is called **Platform Configuration Registers (PCR)**. At power-up, a PCR is set to a known value (either 0x00s or 0xFFs, usually). Sending a digest to the TPM is called extending a PCR because the chosen register will extend its value with the one received with the following logic:

$$\text{PCR}[x] := \text{sha256}(\text{PCR}[x] \mid \text{digest})$$

This way, a PCR can only evolve in one direction and never go back unless the platform is reset. In a typical measured boot flow, a TPM can be configured to disclose a secret only under a certain PCR state. Each software stage will be in charge of extending a set of PCRs with digests of the next software stage. Once in Linux, user software may ask the TPM to deliver its secrets but the only way to get them is having all PCRs matching a known pattern. This can only be obtained by extending the PCRs in the right order, with the right digests.

7 REFERENCES

- [1] <https://www.yoctoproject.org/docs/latest/bitbake-user-manual/bitbake-user-manual.html>
- [2] <https://patchwork.kernel.org/patch/10750087/>
- [3] <https://git.yoctoproject.org/cgit/cgit.cgi/meta-security>
- [4] <https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-dg.pdf>
- [5] <https://docs.aws.amazon.com/iot/latest/developerguide/register-CA-cert.html>
- [6] https://www.infinisilicon.com/dgd/lnfisilicon-SLB%209670VQ2.0-Datasheet-v01_04-EN.pdf?fileId=5546d4626fc1ce0b016fc78270350cd6
- [7] <https://github.com/tpm2-software>
- [8] [CST TOC](#)
- [9] [U-Boot Project](#), on the imx_v2018.03_4.14.78GA release branch (initial release)