

Developer Guide

Security Starter Kit with STM32MP1 and OPTIGA™ TPM 2.0

Date: March 03, 2021 | Version 1.9



The Solutions People



Contents

1	INTRODUCTION.....	5
1.1	Purpose of the Document	5
1.2	Intended Audience	5
1.3	Prerequisites	5
1.4	Scope of Detailed Design.....	5
2	ENVIRONMENT SETUP	6
3	HARDWARE SETUP	7
3.1	Hardware setup - Security Starter Kit with STM32MP1 and OPTIGA™ TPM 2.0	7
3.1.1	STM32MP1 board S3 Dip Switch settings for booting from SD Card.....	7
3.1.2	Hardware connection between Avenger96 and OPTIGA™ TPM2.0	7
3.1.3	Powering up the Board	8
3.1.4	Open board's terminal - console (Minicom) on Linux PC.....	8
4	SOFTWARE SETUP	10
4.1	SSK- STM32MP1 and OPTIGA™ TPM 2.0 Yocto Environment Setup.....	10
4.1.1	Pre-requisite	10
4.1.2	Steps to build the BSP for the SSK- STM32MP1 and OPTIGA™ TPM 2.0 through Yocto	10
4.2	Keys and certificates information.	12
4.3	OPTIGA™ TPM2.0 Setup Script.....	12
4.4	TLS Mutual Authentication & Session Establishment Using H/w Security	15
4.4.1	Linux Environment: Generate the required keys and certificate	16
4.4.1.1	AWS Custom Gateway CA Creation	17
4.4.1.2	Registering Your CA Certificate	17
4.4.2	Linux Environment: Secure Device Certificate and Private Key Gen	20
4.5	AWS Greengrass Group Creation.....	24
4.6	AWS Console and Board: Setup AWS IoT for the Demo	27
4.6.1	Register the Device Certificate to the AWS IoT for the “thing”	28
4.6.2	AWS Console: Create Publish/Subscribe Policy	32
4.6.3	Linux Environment: Configure the AWS Example Application that Connects to AWS	33
4.6.4	Lambda Functions on AWS IoT Greengrass	34
4.6.5	On Board: Execute the AWS Example Application	43
5	STM32MP15 SECURE BOOT	46
5.1	Secure boot implementation	46
5.1.1	Overview	46
5.1.2	Key Generation	47
5.1.2.1	Install STM32MP Key Generator	47
5.1.2.2	STM32MP Key Generator command line interface	47
5.1.2.3	Extending Public key Hash to bootfs in SD-card.....	48
5.1.2.4	Verify Public Key Hash	49
5.1.3	Key Registration	49
5.1.3.1	Register hash public key.....	49
5.1.4	Signing the FSBL and SSBL	50
5.1.4.1	SSBL signing.....	50
5.1.4.2	FSBL signing.....	51
5.1.5	Flash the Signed Image	51
5.1.6	Verify Authentication.....	52
5.1.6.1	Bootrom Authentication	52
5.1.6.2	TF-A authentication	52
5.1.7	Close the device	52
5.2	Measured Boot with OPTIGA™ TPM2.0.....	53

5.2.1	Measured boot Step to verify the platform Integrity	53
6	APPENDIX.....	55
6.1	Avenger – 96 Boards	55
6.2	AWS Greengrass	56
6.3	Tresor Mezzanine OPTIGA™ TPM 2.0.....	58
6.4	Trusted Platform Module – TPM	59
6.5	Boot chains Environment overview	61
6.5.1	Generic boot sequence	61
6.5.2	STM32MP15 boot chain.....	61
6.5.2.1	Overview.....	61
6.5.2.2	ROM Code.....	62
6.5.2.3	First Stage Boot Loader (FSBL)	62
6.5.2.4	Second Stage Boot Loader (SSBL)	62
6.5.2.5	Linux.....	63
6.5.2.6	Secure OS / Secure Monitor	63
6.6	Building a Secure Signed Image.....	63
6.7	Measured boot Principles	65
7	REFERENCES.....	66

FIGURES

Figure 1: Security Starter Kit Architecture	5
Figure 2: Hardware Connection Setup on Avenger96 board	8
Figure 3: Secure boot process flow	47
Figure 4: Measured Boot flow between TPM and Avenger Board.....	53
Figure 5: AWS Greengrass Group	57
Figure 6: Tresor Mezzanine OPTIGA™ TPM 2.0.....	58
Figure 7: Root of Trust	59
Figure 8: Measured/Trusted Boot Process	60
Figure 9: Generic Boot Sequence	61
Figure 10: STM32MP15 boot chain	62
Figure 11: STM32 Image Header Description	63
Figure 12: STM32 Image Header Detailed Description	64

TABLES

Table 1: Preloaded Keys and Certificates	16
--	----

DEFINITION, ACRONYMS AND ABBREVIATIONS

Definition/Acronym/Abbreviation	Description
AV96	Avenger96 Board (with STM32MP157CAC MPU installed)
CSR	Certificate Signing Request
DDR	Double Data Rate Synchronous Dynamic
FSBL	First Stage Boot Loader
PCR	Platform Configuration Register
SSBL	Second Stage Boot Loader
SSK	Security Starter Kit
TF-A	Trusted Firmware-A
TFTP	Trivial File Transfer Protocol
TPM	Trusted Platform module

1 INTRODUCTION

1.1 Purpose of the Document

This guide describes - how to setup the OPTIGA™ TPM 2.0 on Arrow Avenger96 based Yocto platform with integrated TPM driver and Amazon Greengrass support. This is a hardware layer security for Avenger96 communication with cloud.

1.2 Intended Audience

This document is for end-user who wants to use OPTIGA™ TPM 2.0, Avenger96 with STM32MP157CAC and AWS services, enabled with the hardware layer security.

1.3 Prerequisites

Below is the list of Hardware and Software needed to enable demonstration of the AWS GG and OPTIGA™ TPM 2.0 security,

- Security Starter Kit Setup will require following:
 - Avenger96 board (with the STM32MP157CAC MPU installed)
 - Tresor Mezzanine board (with the OPTIGA™ TPM 2.0 installed)
 - SD-card
 - MicroUSB debug cable
 - Power Supply
- Linux PC (Minicom for serial console)
- Internet connectivity (Wi-Fi/Ethernet) of Board and Linux HOST PC should be on same Network

1.4 Scope of Detailed Design

Integration of AWS IoT Greengrass with OPTIGA™ TPM 2.0 to provide secure, hardware-based gateway/edge compute device. This integration ensures the use of private key to establish device identity, which is securely stored in tamper-proof hardware devices, and which prevents the device from being compromised, impersonated, or other malicious activities.

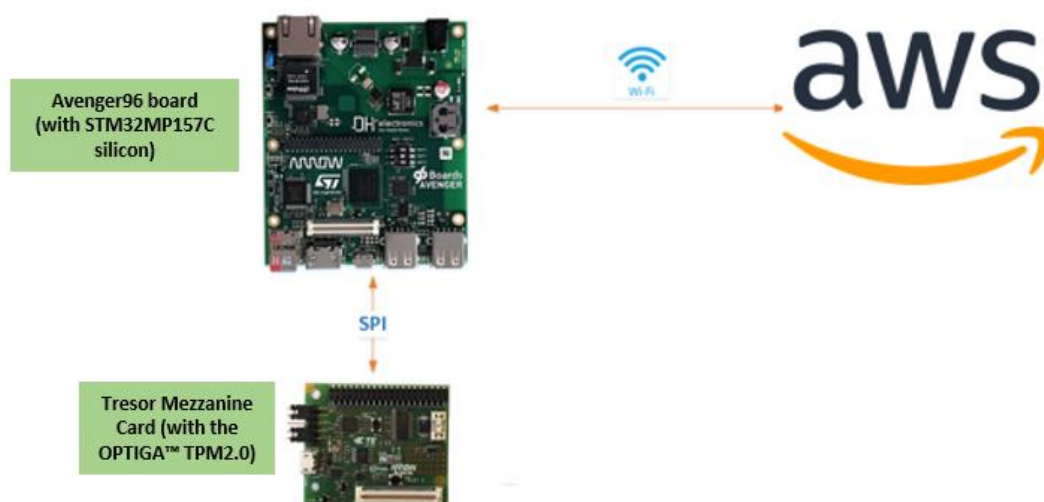


Figure 1: Security Starter Kit Architecture.

2 ENVIRONMENT SETUP

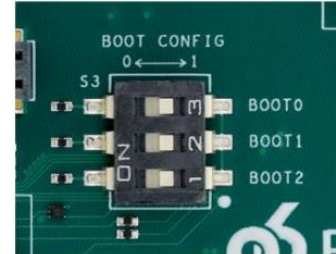
1. **Cloud Services** – Amazon Web Services (User must have an AWS Account Credentials before using this Guide)
2. **Gateway device** - Avenger96 – based on STM32MP157A ([96 boards](#)) (**Must have the STM32MP157CAC processor for Secure Boot Enablement**)
3. **Hardware security device** - [Tresor Mezzanine Infineon OPTIGA™ TPM 2.0](#). (TPM device swapped with the Infineon OPTIGA™ SLB9670 or SLM9670 TPM2.0)
4. **Power Supply** –12V-2A 24W AC/DC Power Supply
5. **Debug Cable** - MicroUSB debug cable
6. **HOST PC** – Linux Operating System (Ubuntu 16.04 preferred) or Windows 10

3 HARDWARE SETUP

3.1 Hardware setup - Security Starter Kit with STM32MP1 and OPTIGA™ TPM 2.0

3.1.1 STM32MP1 board S3 Dip Switch settings for booting from SD Card

The Avenger96 supports multiple boot options which are selected by the DIP-switch S3. To select a logical "1" a switch needs to be pushed to the right. Therefore a logical "0" is set by pushing the switch to the left. The numeration of these pins is printed next to the switch on the circuit board.



BOOT Mode	Comments	Switch 1 - BOOT 2	Switch 2 - BOOT 1	Switch 3 - BOOT 0
UART and USB	USB high-speed Device	0	0	0
NOR-Flash	On Quad SPI	0	0	1
eMMC	On SDMMC2	0	1	0
NAND-Flash (Not available)	SLC NAND Flash	0	1	1
Reserved (NoBoot)	Get boot access without boot from Flash memory	1	0	0
SD-Card (Standard)	On SDMMC1	1	0	1
UART and USB	USB OTG	1	1	0
Serial NAND Flash (Not available)	NAND flash on Quad SPI	1	1	1

3.1.2 Hardware connection between Avenger96 and OPTIGA™ TPM2.0

The mezzanine will be mounted on top of the Avenger96 board as shown in Figure 2. When the Avenger96 board is powered-up, the Power LED on the OPTIGA™ TPM2.0 board turns on, indicating that the board is correctly connected.

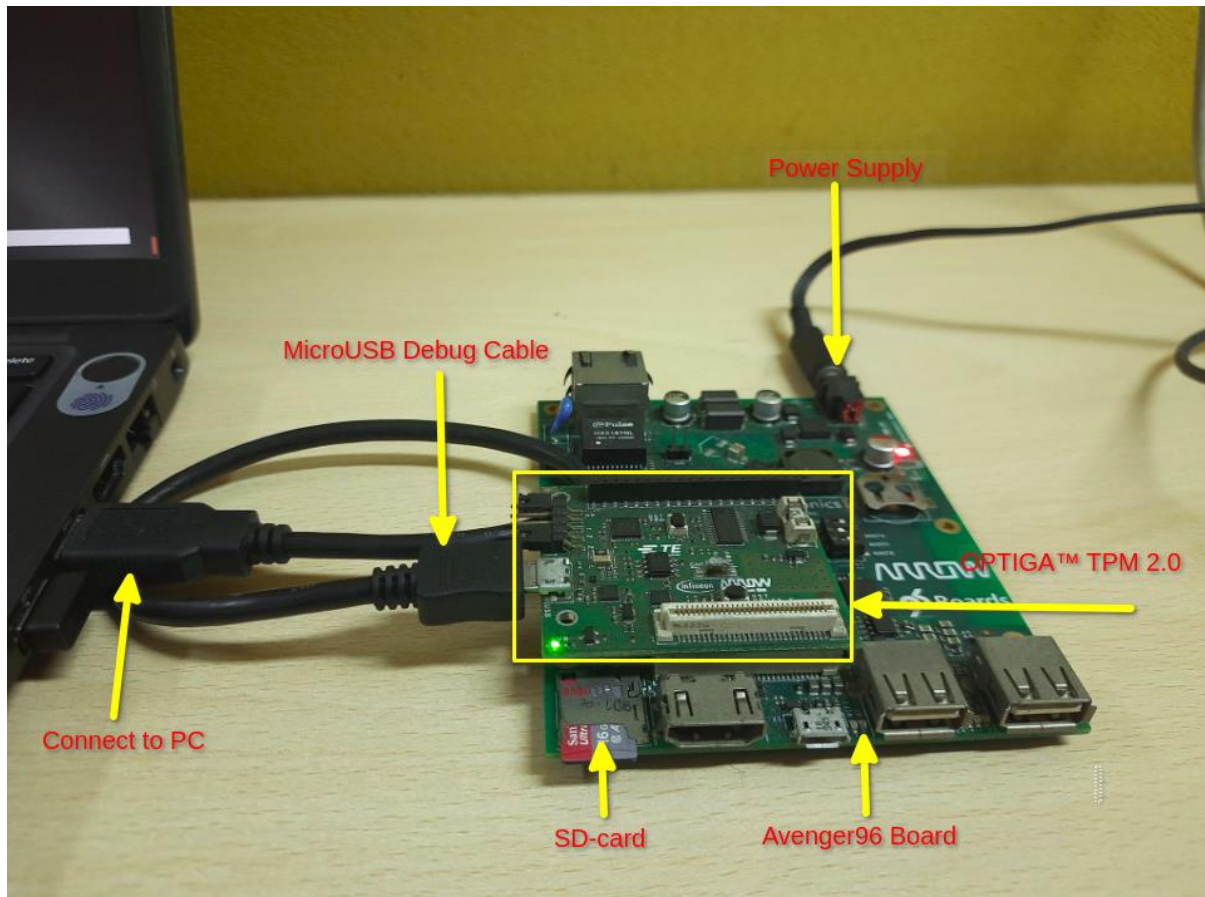


Figure 2: Hardware Connection Setup on Avenger96 board

3.1.3 Powering up the Board

1. Take Avenger96 Board, Insert the provided SD-card (Ensure SD-card is flashed with binaries first time)
2. Connect Micro USB Debug Cable on OPTIGA™ Tresor as shown in above Figure 2.
3. Connect Power-Adapter to Avenger96 and the board is ready to use.
4. Open Serial terminal utility viz. Minicom on HOST PC to serially connect with the board

3.1.4 Open board's terminal - console (Minicom) on Linux PC

1. Before starting with this step, the SD-card must be flashed with binary image and serial cable is plugged into board as mentioned in hardware setup 3.1.1.
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
```

4. Set baud rate and other settings as follows
 - a. Baud rate 115200
 - b. Parity none
 - c. Hardware flow control/software flow control none
 - d. Serial device /dev/ttyUSB0
 - e. **save setup as dfl**

5. After the Avenger96 board boots up, it will display login console on minicom terminal on Linux PC as shown below.
6. Username for board is “root” without any password

```
Avenger96 v3.3 - ST OpenSTLinux - Weston - (A Yocto Project Based Distro) 2.6-snapshot stm32mp1-av96 ttySTM0
stm32mp1-av96 login: root (automatic login)
root@stm32mp1-av96:~#
```

4 SOFTWARE SETUP

4.1 SSK- STM32MP1 and OPTIGA™ TPM 2.0 Yocto Environment Setup

4.1.1 Pre-requisite

- Linux HOST PC (x86) with Ubuntu 16.04 LTS installed (to build Yocto image)
- Basic understanding of Linux commands
- [Required steps for building a BSP for ST's development boards can be found here](#)

4.1.2 Steps to build the BSP for the SSK- STM32MP1 and OPTIGA™ TPM 2.0 through Yocto

[**Note:** The image pre-installed on the SD card in the box should be ready to go. If for some reason it is corrupted, or it becomes corrupted, the user should follow below steps to re-build the image.]

1. Download the SSK Avenger96 release package from Arrow GitHub [Security-Starter-Kits-STM32MP157-SSK.zip](#) on Linux HOST PC
2. Extract the [Security-Starter-Kits-STM32MP157-SSK.zip](#)

```
Linux-PC $ unzip Security-Starter-Kits-STM32MP157-SSK.zip
```

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

- [STM32MP157-SSK Developers Guide.pdf](#) that has detailed description of all the components, examples, and how to enable all features of the Kit.
- [STM32MP157-SSK Quick Start Guide.pdf](#)
- [STM32MP1 SSK Pkg Rel](#)
 - Firmware_Image
 - SSK_AWS_Demo
 - SSK_Cert_And_Config
 - SSK_Suit_Configuration
 - Stm32mp1_Yocto_Build

3. Run the build script `build_script_avg.sh` to complete Yocto environment setup.

[**Note:** Please refer [PC prerequisites](#) before running the below script.]

```
Linux-PC $ cd Stm32mp1_Yocto_Build/
Linux-PC $ ./build_script_avg.sh
```

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

4. Once Yocto build is complete user will be able to see the below screenshot. Create SD-card image.

```
NOTE: Tasks Summary: Attempted 8962 tasks of which 5770 didn't need to be rerun and all succeeded.
NOTE: Writing buildhistory
```

- Navigate to the directory “Firmware_Image “ in order to find the built SD-card image.

```
Linux-PC $ cd STM32MP1_SSK_Pkg_Rel/Firmware_Image /
```

5. Flash the SD-card using the following command

```
Linux-PC $ sudo dd if=flashlayout_av96-weston_FlashLayout_sdcard_stm32mp157a-av96-trusted.raw of=/dev/sdb bs=1M conv=fsync ;sync
```

6. Unmount and eject the SD-card from the Linux PC.
7. Re-inserting the SD-card to Linux PC, it will mount below partitions, verify using below command

```
Linux-PC $ sudo lsblk
```

```
└─mmcblk0p4 179:4  0  64M 0 part /media/<username>/bootfs
└─mmcblk0p5 179:5  0  16M 0 part /media/<username>/vendorfs
└─mmcblk0p6 179:6  0   2G 0 part /media/<username>/rootfs
└─mmcblk0p7 179:7  0  1.9G 0 part /media/<username>/userfs
```

8. Copy below listed files to SD card file system i.e SSK_Suit_package

- Linux-PC \$ sudo cd STM32MP1_SSK_Pkg_Rel
- Linux-PC \$ sudo cp -r SSK_AWS_Demo SSK_Suit_Configuration /media/<username>/rootfs/home/root/
- Linux-PC \$ sudo cp -r SSK_Cert_And_Config/AWS_Config/openssl.cnf /media/<username>/rootfs/etc/ssl/
- Linux-PC \$ sudo cp -r SSK_Cert_And_Config/AWS_Config/config.json /media/<username>/rootfs/greengrass/config/
- Linux-PC \$ sudo cp -r SSK_Cert_And_Config/AWS_ROOTCA/rootCA.key /media/<username>/rootfs/greengrass/certs/
- Linux-PC \$ sudo cp -r SSK_Cert_And_Config/AWS_ROOTCA/rootCA.pem /media/<username>/rootfs/greengrass/certs/
- Linux-PC \$ sudo cp -r SSK_Cert_And_Config/AWS_ROOTCA/root.ca.pem /media/<username>/rootfs/greengrass/certs/
- Linux-PC \$ sudo cp -r SSK_Suit_Configuration/Tpm_Measured_Boot/Measured_boot.sh /media/<username>/rootfs/etc/init.d/
- Linux-PC \$ sudo cp -r SSK_Suit_Configuration/Tpm_Measured_Boot/rc-local.service /media/<username>/rootfs/etc/systemd/system/
- Linux-PC \$ sudo cp -r SSK_Suit_Configuration/Tpm_Measured_Boot/rc.local /media/<username>/rootfs/etc/
- Linux-PC \$ sync

9. The SD-card is now ready for use on the Avenger96 board
10. Plug the SD-card in the Avenger96 board

11. Power up the board. With the board powering up, launch the Minicom terminal (or Putty on Windows). The terminal should display the boot-up logs and ask for a login once the boot process completes.

4.2 Keys and certificates information.

- Preloaded one root certificate (associated private key provided as a file) - **rootCA.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 HSM**
- 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 OPTIGA™ TPM2.0 Setup Script

After successful boot-up of the Avenger96 board, the user needs to setup the hardware security chip OPTIGA™ TPM2.0. This Setup script will setup install prerequisite packages and TPM configuration (i.e. create Keypair, device certificate, and store them in the OPTIGA™ TPM2.0).

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

```
root@stm32mp1-av96:~# cd /greengrass/certs
root@stm32mp1-av96:~# openssl genrsa -out rootCA.key 2048
root@stm32mp1-av96:~# 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@stm32mp1-av96:~# cd ~/SSK_Suit_Configuration
root@stm32mp1-av96:~# ./SSK_Suit_Configuration.sh
```

```
root@stm32mp1-av96:~# cd SSK_Suit_Configuration/
root@stm32mp1-av96:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh
*****
Welcome to SECURITY-STARTER-KIT (SSK) Auto Flashing Tool
*****

-----Prerequisite-----

---> Please Enable Your Mobile Or Router Hotspot for internet connectivity <---

---> Have you enable the hostpot? y/n <---
y
```

```

+++++
Wifi Connection Provisioning Board
+++++

---> [WIFI] List of available Wifi devices in Range... <---
SSID: Leica-Argos
SSID: ei-SecureWifi
SSID: ei-GuestWifi
SSID: ei-SecureWifi
SSID: ei-GuestWifi
SSID: Rahul
SSID: Sai Financial
SSID: ei-GuestWifi
SSID: ei-SecureWifi
SSID: Test
SSID: ei-SecureWifi
SSID: ei-GuestWifi
SSID:
    * SSID List
SSID: ORBI70
    * SSID List
SSID: Chetan Soni\X20
SSID: KIFS
SSID: ei-GuestWifi

---> Can you see your wifi devices:SSID? y/n <---
y
---> Please Enter the Name of your Wifi-Device SSID <---
Test
---> Can you please Provide the Password of your Wifi-Device <---
12345678
Successfully initialized wpa_supplicant

```

- Now, it will start installing the package. Note: for the first time install 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 <---

```

- 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 <---

```

- Verify the Steps using below commands [Note: if asked the PIN is: 1234]

```
root@stm32mp1-av96:~# ./SSK_Suit_Configuration.sh setup_result
```

```

root@stm32mp1-av96:~/SSK_Suit_Configuration# ./SSK_Suit_Configuration.sh setup_result

---> Running TPM Self-Test <---

---> Checking TPM Self-Test Result <---
status: success
data: 001fe18b00000001bc7

---> 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: c439b7c86d161077cf35c1ff0e3e4d6cc8c9ef3d3ff66fe0f4a8127e529362779b92eeb9817a95baf446c5f5d6f8dee31f5574d5597c0665df95b817ac982b3c37305ba4bfd6281

```

```

---> Verifying Token Module Created with Provide Lable 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: 13.11
  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=%35%34%63%64%64%35%38%64%31%37%39%32%65%35%30%63;oe
Token 'greengrass' with URL 'pkcs11:model=SLM9670;manufacturer=Infineon;serial=0000000000000000;token=greengrass' requires user PIN
Enter PIN:
  Type: Private key (RSA-2048)
  Label: greenkey
  Flags: CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
  ID: 35:34:63:64:64:35:38:64:31:37:39:32:65:35:30:63

---> 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]

6. TPM Clear command

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

```
root@stm32mp1-av96:~# ./SSK_Suit_Configuration.sh tpm_clear
```

```

root@stm32mp1-av96:~/SEED_Suit_Configuration_Rel_01# ./SEED_Suit_Configuration.sh tpm_clear
---> TPM2.0 Cleared with Tokens and Labels <---
0x1500016:
---> Clearing Done Exiting <---
root@stm32mp1-av96:~/SEED_Suit_Configuration_Rel_01#

```

7. If TPM is cleared, then user will get below logs. Once cleared, the user will need to execute the above steps from 1 to 5 again as described in section 4.3.

```

---> Running TPM Self-Test <---

---> Checking TPM Self-Test Result <---
status: success
data: 0001f9db000000000000

---> Verifying Token Handle Created or Not <---

---> Verifying Token Module Created with Provide Lable 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 <---

```

4.4 TLS Mutual Authentication & Session Establishment Using H/w Security

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 better fits 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 Avenger96 Gateway with TPM2.0 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 TPM 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; this is because 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 TPM2.0.

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, for this example, users 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 TPM and AV96	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 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 and 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 in **Section 4.3 - OPTIGA™ TPM2.0 Setup Script** then move to step 4.4.1.2

4.4.1 Linux Environment: Generate the required keys and certificate

To use your own X.509 device certificates, **you must 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 who need to use their own device certificates signed by their preferred **Certificate Authority (CA)**. This is, in addition, to support the 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 lines lack 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. However, in reality, you will have the signing certificates issued by your CA vendor in place of this sample CA. This sample CA certificate is used later in the walkthrough to sign a device certificate that will be registered with AWS IoT:

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

To create a CA certificate

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

```
root@stm32mp1-av96:~# openssl genrsa -out rootCA.key 2048
```

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

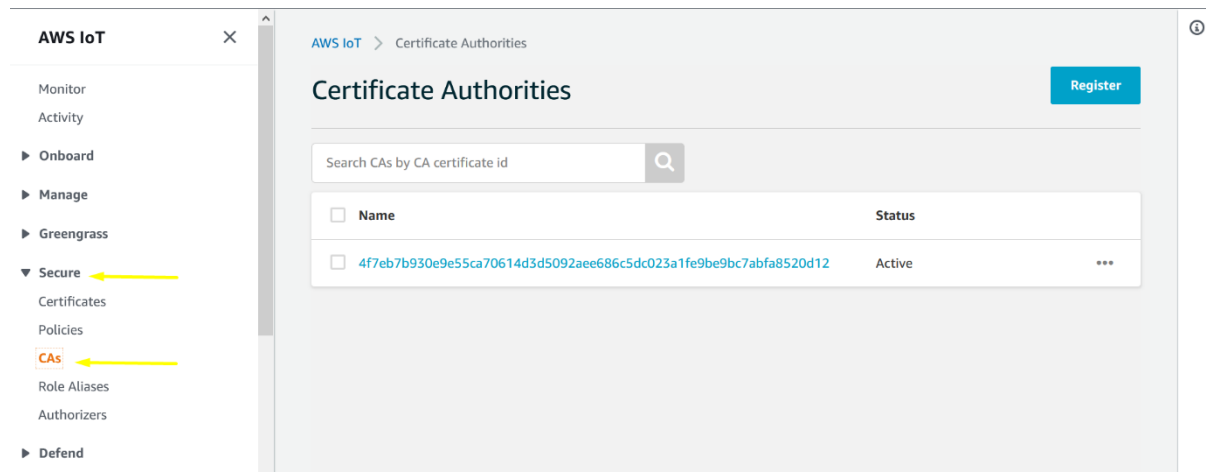
```
root@stm32mp1-av96:~# openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.pem
```

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 certificate, 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

```
1c88ceefdaf69a90f579e3abe85784b1c6d5b8e40c10743cfc59fd28e432e56
```

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

```
root@stm32mp1-av96:~# openssl genrsa -out verificationCert.key 2048
```

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

```
root@stm32mp1-av96:~# openssl req -new -key verificationCert.key -out verificationCert.csr
```

User need to update some information, including the Common Name for the certificate.

Country Name (2-letter code) [AU]:
 State or Province Name (full name) []:
 Locality Name (for example, city) []:
 Organization Name (for example, company) []:
 Organizational Unit Name (for example, section) []:
 Common Name (e.g. server FQDN or YOUR name)
 []: XXXXXXXXXXXXXMYREGISTRATIONCODEXXXXXX
 Email Address []:

- Use the **CSR** to create a private key verification certificate:

```
root@stm32mp1-av96:~# openssl x509 -req -in verificationCert.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out verificationCert.crt -days 500 -sha256
```

- 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:

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
```

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.crt
```

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 be registered with AWS IoT before use

1. We have made changes in Avenger96 meta-security build image
2. Check if any persistent handle is already present with OPTIGA™ TPM2.0.

```
root@stm32mp1-av96:~# tpm2_listpersistent
- handle: 0x81000000
  name-alg:
    value: sha256
    raw: 0xb
  attributes:
    value: fixedtpm|fixedparent|sensitivedataorigin|userwithauth|restricted|decrypt
    raw: 0x30072
.
.
.
```

3. If you want to clear the previous token and handle, inside TPM2.0 Chipset, then, use the below command

```
root@stm32mp1-av96:~# tpm2_evictcontrol -a o -c 0x81000000 -p 0x81000000
```

4. Install python packages,

```
root@stm32mp1-av96:~# 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
```

5. Clone the script file to board

```
root@stm32mp1-av96:~# git clone https://github.com/tpm2-software/tpm2-pkcs11
root@stm32mp1-av96:~# cd tpm2-pkcs11/
root@stm32mp1-av96:~# git checkout a82d0709c97c88cc2e457ba111b6f51f21c22260
```

6. Run the script to generate keys and token inside TPM2.0 inside given directory.

```
root@stm32mp1-av96:~# cd ~/tpm2-pkcs11/tools

root@stm32mp1-av96:~# ./tpm2_ptool.py init --pobj-pin=1234 --path=/opt/tpm2-pkcs11/

-Created a primary object of id: 1

root@stm32mp1-av96:~# ./tpm2_ptool.py addtoken --pid=1 --pobj-pin=1234 --sopin=1234 --
userpin=1234 --label=greengrass --path=/opt/tpm2-pkcs11/

-Created token label: greengrass

root@stm32mp1-av96:~# ./tpm2_ptool.py addkey --algorithm=rsa2048 --label=greengrass --
userpin=1234 --key-label=greenkey --path=/opt/tpm2-pkcs11/
```

```
-Added key as label: "greenkey"
```

7. Soft link the resource manager libraries for listing token from TPM and providing to board console.

```
root@stm32mp1-av96:~#cd /usr/lib/
root@stm32mp1-av96:~# ln -s libtss2-tcti-tabrmd.so.0 libtss2-tcti-tabrmd.so
```

8. Now for checking the URL's of token generated, p11tool and p11-kit and other packages are needed. Use p11-kit list-modules: command to list the HSI modules available with tokens.

```
root@stm32mp1-av96:~# 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: SLB 9670
  serial-number: 0000000000000000
  hardware-version: 1.16
  firmware-version: 7.40
  flags:
    rng
    login-required
    user-pin-initialized
    token-initialized
```

9. Use command “p11tool –list-tokens” to see token with its URL.

```
root@stm32mp1-av96:~# p11tool --list-tokens
Token 0:
  URL:
pkcs11:model=SLB9670;manufacturer=Infineon;serial=0000000000000000;token=greengrass
  Label: greengrass
  Type: Hardware token
  Flags: RNG, Requires login
  Manufacturer: Infineon
  Model: SLB9670
  Serial: 0000000000000000
  Module: libtpm2_pkcs11.so
```

10. Use command “p11tool --list-privkeys pkcs11:manufacturer=Infineon” to see PKCS listing OPTIGA™ TPM2.0 private and public keys

Note: Provide PIN: 1234 if asked

```
root@stm32mp1-av96:~# p11tool --list-privkeys pkcs11:manufacturer=Infineon
Object 0:
URL:
pkcs11:model=SLB9670;manufacturer=Infineon;serial=0000000000000000;token=greengrass;id=%36
%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=Infineon;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@stm32mp1-av96:~# cd /usr/lib/engines-1.1/
root@stm32mp1-av96:~# /usr/lib/engines-1.1# ln -s pkcs11.so libpkcs11.so
root@stm32mp1-av96:~# export PKCS11_MODULE_PATH=/usr/lib/libtpm2_pkcs11.so
```

12. Edit openssl.conf for enabling PKCS11 interface for TPM (Edit only highlighted points)
Open /etc/ssl/openssl.conf

```
#
# OpenSSL example configuration file.
# This is mostly being used for generation of certificate requests.
#
# Note that you can include other files from the main configuration
# file using the .include directive.
#.include filename
# This definition stops the following lines choking if HOME isn't
# defined.
HOME          = .
openssl_conf = openssl_init
```

13. Edit below contents at the end of openssl.conf

```
[openssl_init]
engines=engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
dynamic_path = /usr/lib/engines-1.1/libpkcs11.so
MODULE_PATH = /usr/lib/pkcs11/libtpm2_pkcs11.so
init = 0
```

14. Generate certificate Signing Request with openssl

```
root@stm32mp1-av96:~# openssl req -engine pkcs11 -new -key
"pkcs11:model=SLB9670;manufacturer=Infineon;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]:GUJARAT
Locality Name (eg, city) []:AHM
Organization Name (eg, company) [Internet Widgits Pty Ltd]:EIC
Organizational Unit Name (eg, section) []:KB
Common Name (e.g. server FQDN or YOUR name) []:SSK
Email Address []:

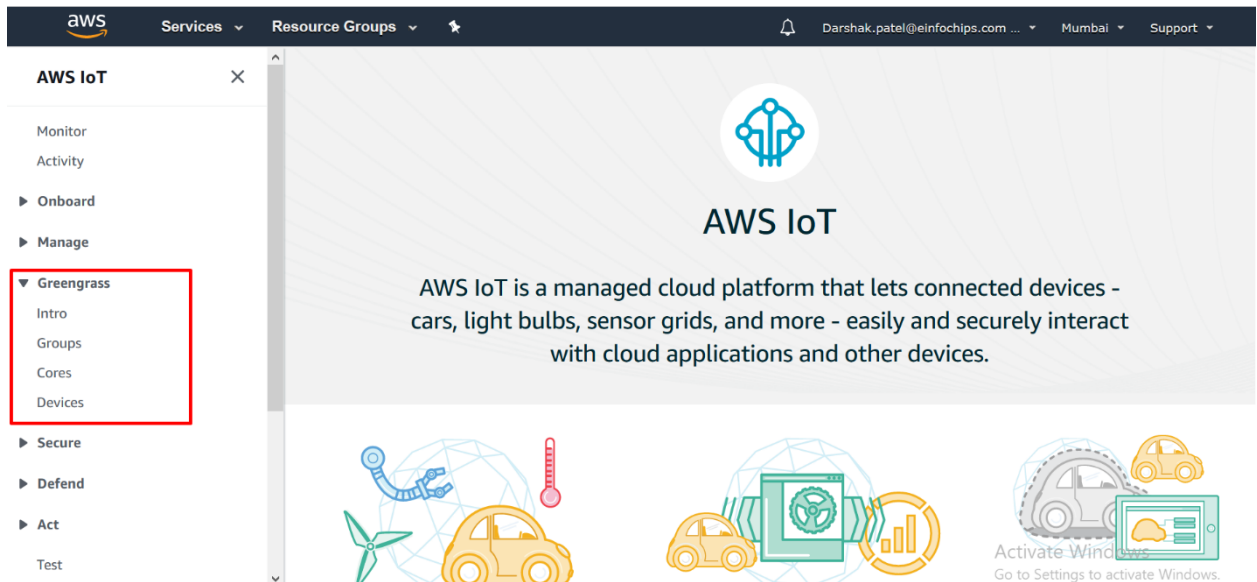
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:1234
An optional company name []:ARROW
```

15. Registering Device Certificates Manually

```
root@stm32mp1-av96:~# cd /greengrass/certs/
root@stm32mp1-av96:~# openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key
-CAcreateserial -out aws_device_cert.pem -days 500 -sha256
```

4.5 AWS Greengrass Group Creation

1. Create a Greengrass group after log-in to your AWS account.
2. Sign-in to the AWS Management Console and open the AWS IoT console. Choose Get started if you are opening this console for the first time
 - 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>.
 - If prompted by Greengrass, asking for your permission to access other services via dialog box (see below), 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:

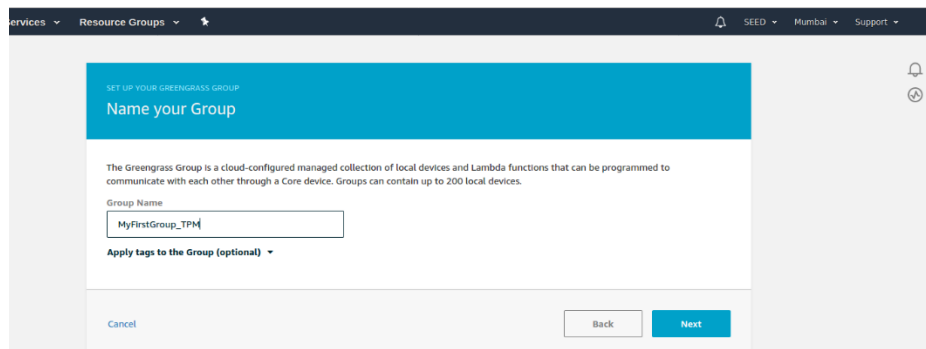
- Creates a service role named Greengrass_ServiceRole, if one doesn't exist, and attaches the [AWSGreengrassResourceAccessRolePolicy](#) managed policy to the role.
- Attaches the service role to your AWS account in the AWS Region that's currently selected in the console.

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

Cancel

Grant permission

4. When setting up your Greengrass group, choose “Customize”
5. Enter a name for your group (for example, MyFirstGroup_TPM), and then choose Next



SET UP YOUR GREENGRASS GROUP

Name your Group

The Greengrass Group is a cloud-configured managed collection of local devices and Lambda functions that can be programmed to communicate with each other through a Core device. Groups can contain up to 200 local devices.

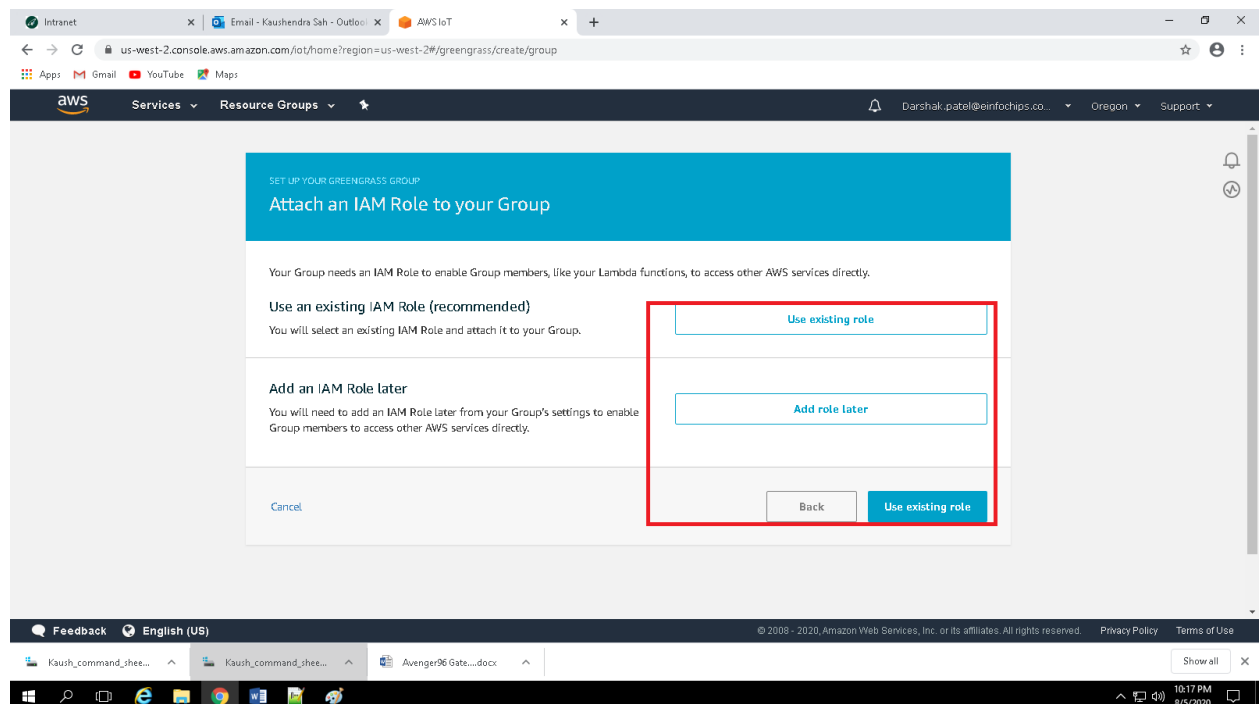
Group Name

MyFirstGroup_TPM

Apply tags to the Group (optional) ▼

Cancel Back Next

6. Add IAM Role to the Greengrass Group. Select ‘Use existing role’ and then select ‘Greengrass_ServiceRole’ from the list of roles, and select ‘Next’



SET UP YOUR GREENGRASS GROUP

Attach an IAM Role to your Group

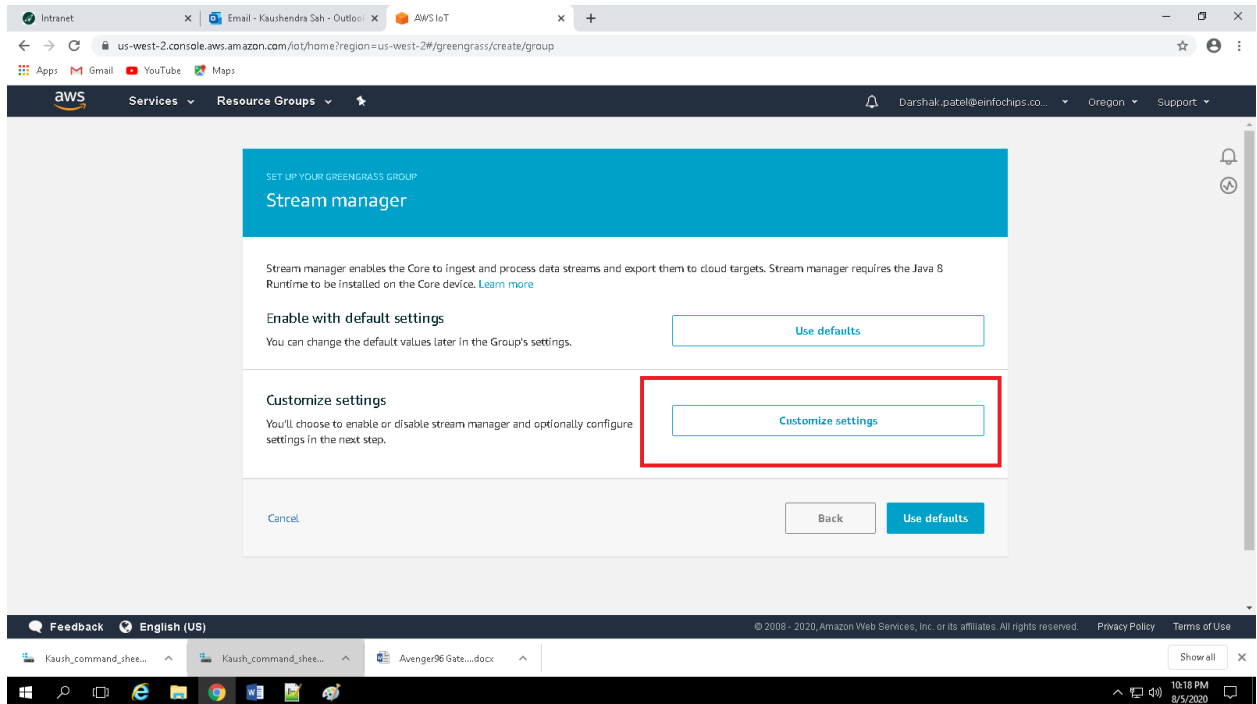
Your Group needs an IAM Role to enable Group members, like your Lambda functions, to access other AWS services directly.

Use an existing IAM Role (recommended)
You will select an existing IAM Role and attach it to your Group.

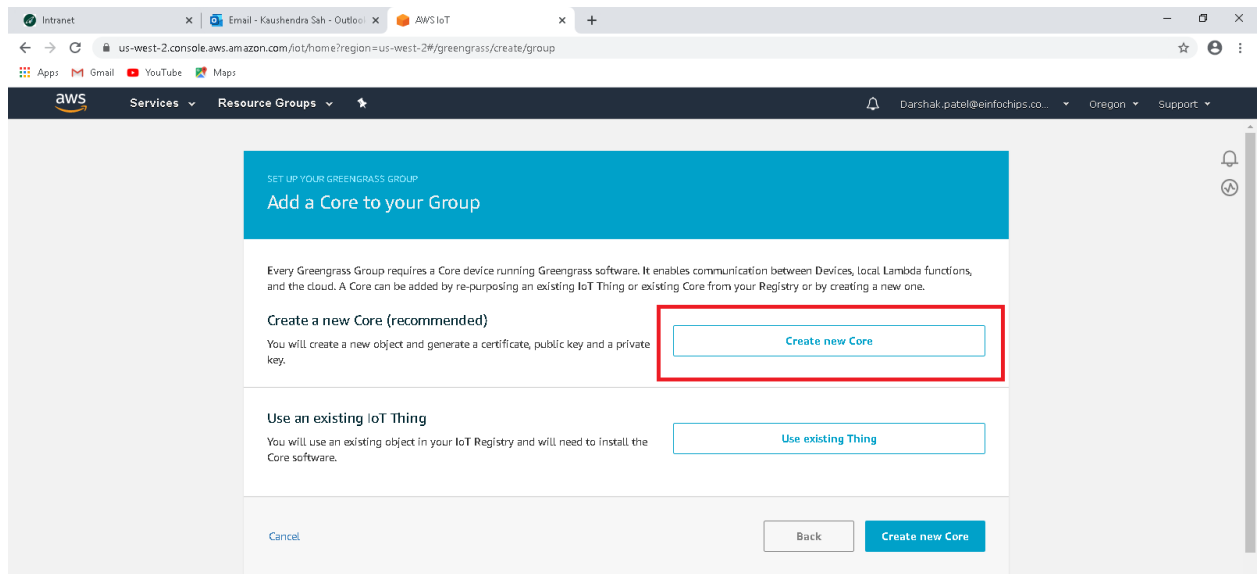
Add an IAM Role later
You will need to add an IAM Role later from your Group's settings to enable Group members to access other AWS services directly.

Cancel Back Use existing role Add role later

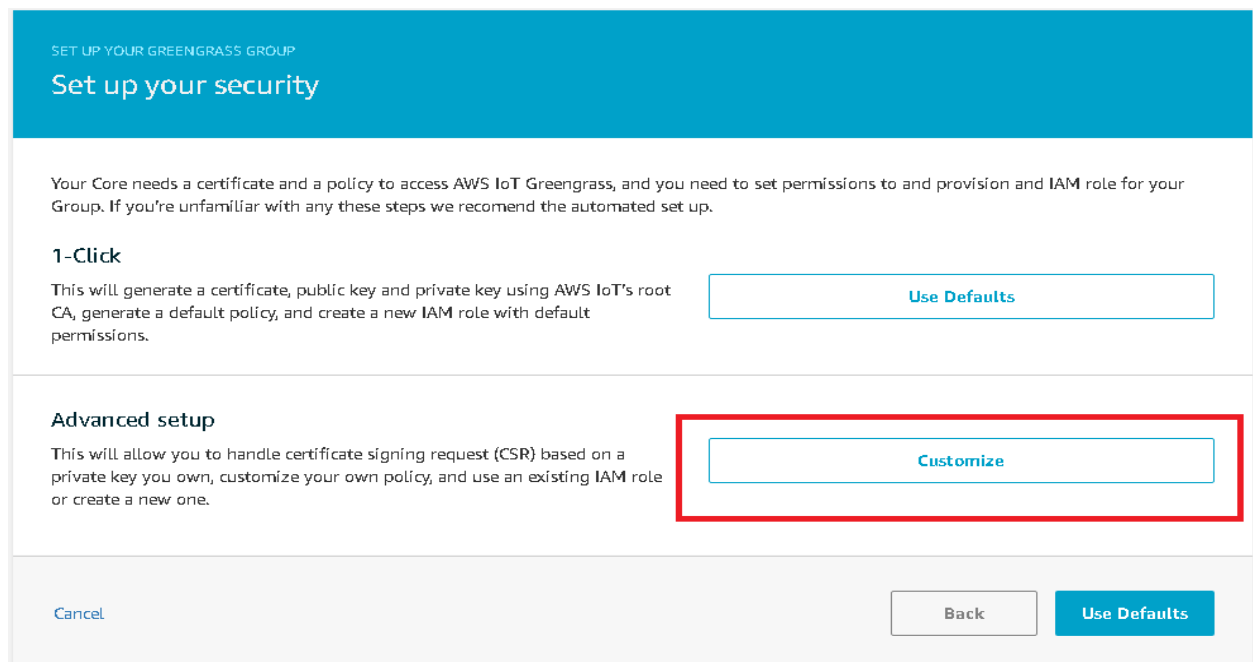
7. On the **Stream Manager** page, select the option 'Customize settings' and then select 'Disable'



8. Select the Greengrass Core name to reflect the Gateway Device running with it.
E.g. **MyFirstGroup_TPM_Core**



9. On the Security set up page, select 'Customize' to upload the Gateway Device certificate Generated by OPTIGA™ TPM2.0 and signed with Registered RootCA with AWS depicted in Section 4.4.1



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

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

Create certificate

Create with CSR

Get started

2. As shown in the screenshot below, the user will have to click and select the CA that was just registered and then click on the bottom blue button “Register certificates”.

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

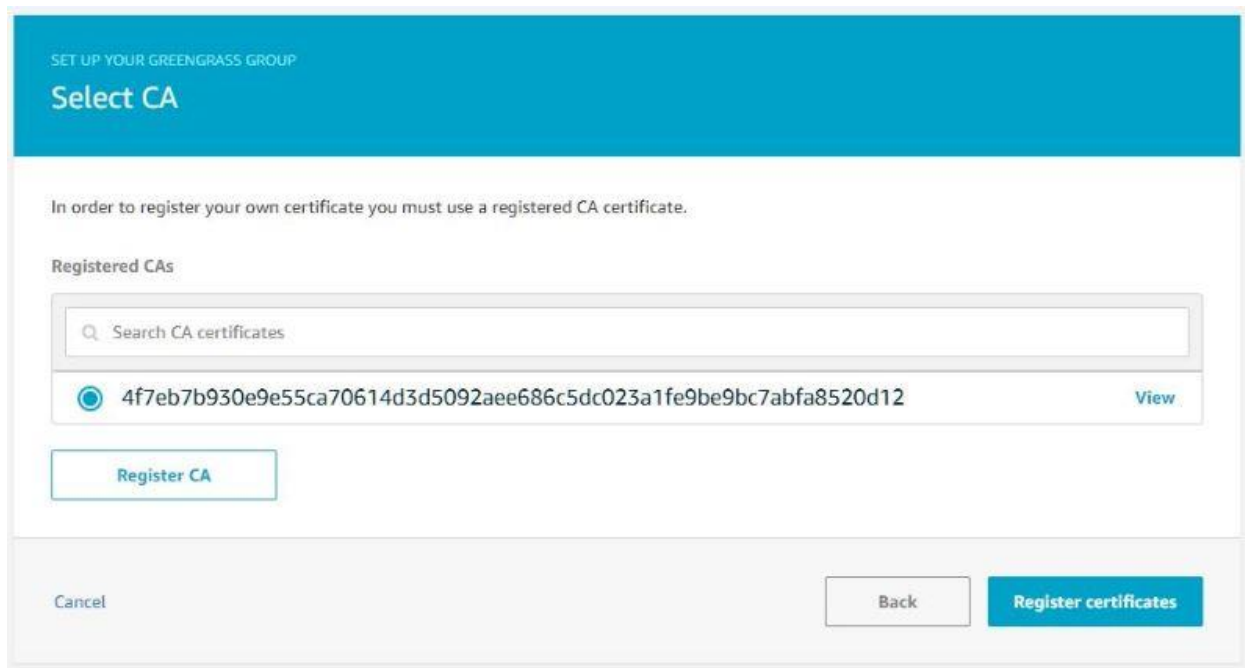
Search CA certificate

3d8d56d4997296baa360a2196baef2261fb932fbfb1741eb45598ab57146f656 [View](#)

Register CA

Cancel

Next



SET UP YOUR GREENGRASS GROUP

Select CA

In order to register your own certificate you must use a registered CA certificate.

Registered CAs

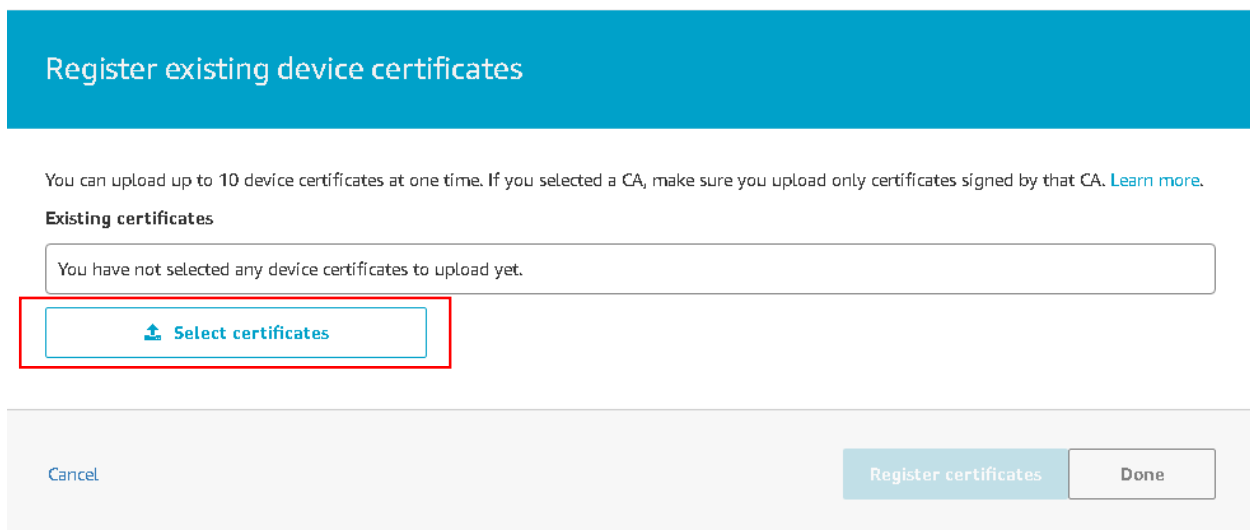
Search CA certificates

☒ 4f7eb7b930e9e55ca70614d3d5092aee686c5dc023a1fe9be9bc7abfa8520d12 [View](#)

Register CA

Cancel Back Register certificates

- As shown 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

- 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	<input checked="" type="radio"/>	<input type="radio"/>	Remove

[Select certificates](#)

[Done](#) [Register certificates](#)

- 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**.

SET UP YOUR GREENGRASS GROUP

Attach policies to your certificate

You are attaching policies to the following certificate:

46e134cb0fb658f7f10196a548aac43ac24914d557a337ec202c8dcd67bc9849

☒ SSK_Test_Policy [View](#)

[Create new policy](#)

1 policy selected [Back](#) [Create Group and Core](#)

Connect your Core device

The final steps are to load the Greengrass software and then connect your Core device to the cloud. You can defer connecting your device at this time, but **you must download your public and private keys now as these cannot be retrieved later.**

You also need to download a root CA for AWS IoT:

[Choose a root CA](#) ↗

Download the current Greengrass Core software

By downloading this software you agree to the [Greengrass Core Software License Agreement](#). To install Greengrass on your Core download the package and follow the [Getting Started Guide](#).

[Choose your platform](#) ↗

Finish

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 as shown in [the example policy](#) in the below image:

Policy document

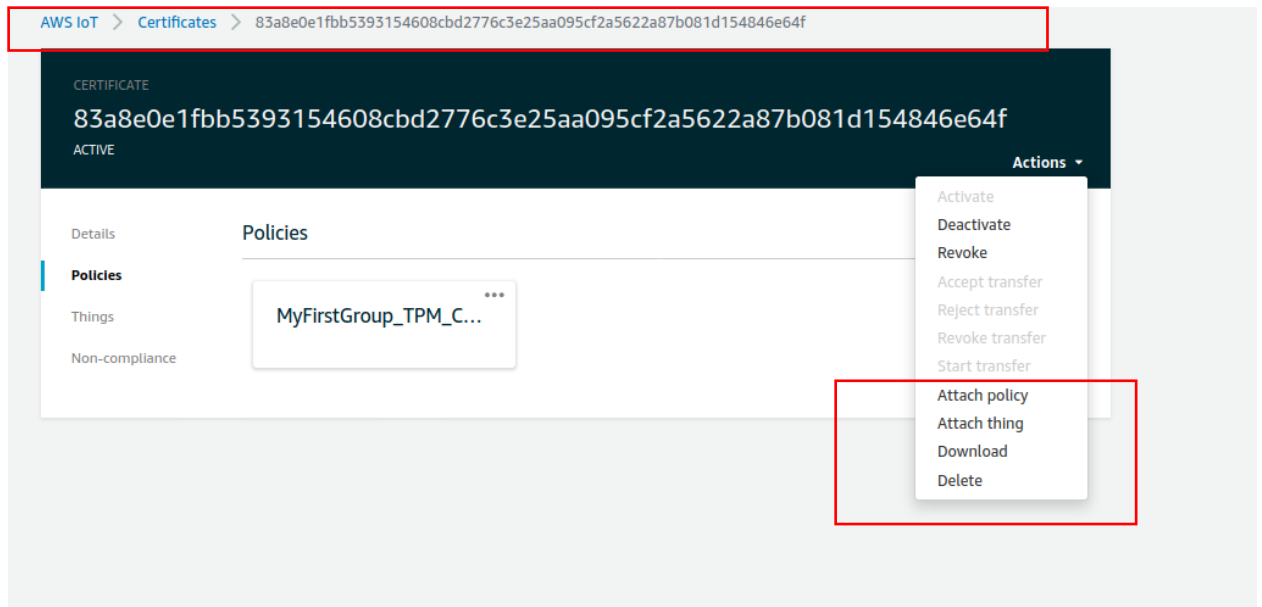
The policy document defines the privileges of the request. [Learn more](#)

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 TPM 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 in the preceding sections resembles the following:

```
root@stm32mp1-av96: vi /greengrass/config/config.json
{
  "coreThing": {
    "thingArn"
: "arn:aws:iot:<AWS_REGION>:<AWS_ACCOUNT_NUMBER>:thing/<GG_Thing_Name>",
    "iotHost": "XXXXXXXXXXXXXXXX-ats.iot.us-east-1.amazonaws.com",
    "ggHost": "greengrass-ats.iot.ap-south-1.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {
      "useSystemd": "yes"
    }
  },
  "managedRespawn": false,
  "crypto": {
    "PKCS11": {
      "OpenSSLEngine": "/usr/lib/engines-1.1/pkcs11.so",
      "P11Provider": "/usr/lib/pkcs11/libtpm2_pkcs11.so",

```

```

    "SlotLabel": "greengrass",
    "SlotUserPin": "1234"
  },
  "principals" : {
    "IoTCertificate" : {
      "privateKeyPath" :
"pkcs11:model=SLB9670;manufacturer=Infineon;token=greengrass;object=greenkey;type=private;
pin-value=1234",
      "certificatePath" : "file:///greengrass/certs/aws_device_cert.pem"
    }
  },
  "caPath" : "file:///greengrass/certs/root.ca.pem"
}
}

```

The screenshot displays the AWS IoT Greengrass console. The left sidebar shows the navigation menu with 'Greengrass' expanded, highlighting 'Groups'. The main content area shows the configuration for a specific group, 'Test_SSK_Core'. The 'Details' tab is active, displaying the 'Thing ARN' as 'arn:aws:iot:us-west-2:565720404376:thing/Test_SSK_Core' and the 'Type' as 'No type'. Below this, the 'Settings' section is visible, showing the 'Custom endpoint' as 'a18a51yripct3j-ats.iot.us-west-2.amazonaws.com' and a status of 'ENABLED'.

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. Download the AWS IoT Greengrass Core SDK for Python to your computer.

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

Create function Info

Choose one of the following options to create your function.

Author from scratch Info
 Start with a simple Hello World example.

Use a blueprint
 Build a Lambda application from sample code and configuration presets for common use cases.

Browse serverless app repository
 Deploy a sample Lambda application from the AWS Serverless Application Repository.

Basic information

Function name Info
 Enter a name that describes the purpose of your function.

 Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
 Choose the language to use to write your function.

Permissions Info
 Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.
 ▶ Choose or create an execution role

Cancel **Create function**

- 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.function_handler`

Greengrass_HelloWorld Throttle Qualifiers Actions Select a test event Test Save

Function code Info

Code entry type

Runtime

Handler Info

Function package
 hello_world_python_lambda.zip (30.9 kB)
 For files larger than 10 MB, consider uploading using Amazon S3.

Environment variables (0) Edit

Key	Value
No environment variables	
No environment variables associated with this function.	

- 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.)

aws

Services

Resource Groups

SEEDMumbaiSupport

Greengrass_HelloWorld

ThrottleQualifiersActionsSelect a test eventTestSave

Function code

Code entry type

Upload a .zip file

Runtime

Python 2.7

Handler

greengrassHelloWorld.function_handler

Function package

Uploadhello_world_python_lambda.zip (30.9 kB)

For files larger than 10 MB, consider uploading using Amazon S3.

Environment variables (0)

Edit

Key	Value
No environment variables	
No environment variables associated with this function.	
Manage environment variables	

GreenGrass_HelloWorld

Throttle Qualifiers Actions Select a test event Test

Configuration Permissions Monitoring

▼ Designer

GreenGrass_HelloWorld

Layers {0}

+ Add trigger + Add destination

Function code info Actions Deploy

File Edit Find View Go Tools Window Test Deploy

Environment

- GreenGrass_HelloWorld
 - greengrasssdk
 - shim_manager
 - data
 - _init_.py
 - _init_.py
 - exceptions.py
 - shimmanagerdir
 - util.py
 - utilmanager.py
 - util
 - _init_.py
 - logging.py
 - _init_.py
 - client.py
 - iotdatautils.py
 - lambda_function.py
 - SecurityManager.py
 - greengrassHelloWorld.py

greengrassHelloWorld lambda_function

```

1 1
2 # Copyright 2018-2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
3 #
4
5 import logging
6 import re
7
8 from io import BytesIO
9
10 from greengrass.common.function import FunctionMetadata
11 from greengrass.spc.python_sdk_ipc_client import IPCClient, IPCException
12 from greengrassdk.util.testing import mock
13
14 # Log messages in the SDK are part of customer's log because they're helpful for debugging
15 # customer's lambda. Since we configured the root logger to log to customer's log and set the
16 # propagate flag of this logger to True, the log messages submitted from this logger will be
17 # sent to the customer's local Cloudwatch handler.
18 customer_logger = logging.getLogger(__name__)
19 customer_logger.propagate = True
20
21 valid_base64_regex = '^([A-Za-z0-9+/]{4}){0,1}([A-Za-z0-9+/]{4}){0,1}([A-Za-z0-9+/]{4}){0,1}([A-Za-z0-9+/]{2}){0,1}$'
22
23
24 class InvocationException(Exception):
25     pass
26
27
28 class Client:
29     def __init__(self, endpoint='localhost', port=None):
30         ...
31         :param endpoint: Endpoint used to connect to IPC.
32         :type endpoint: str
33

```

1:1 Python Spaces: 4

Runtime settings info Edit

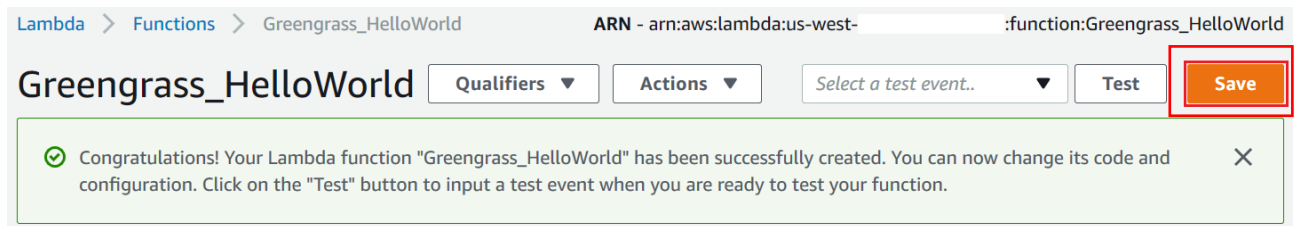
Runtime: Python 2.7

Handler info greengrassHelloWorld.function_handler

Environment variables (0) Edit

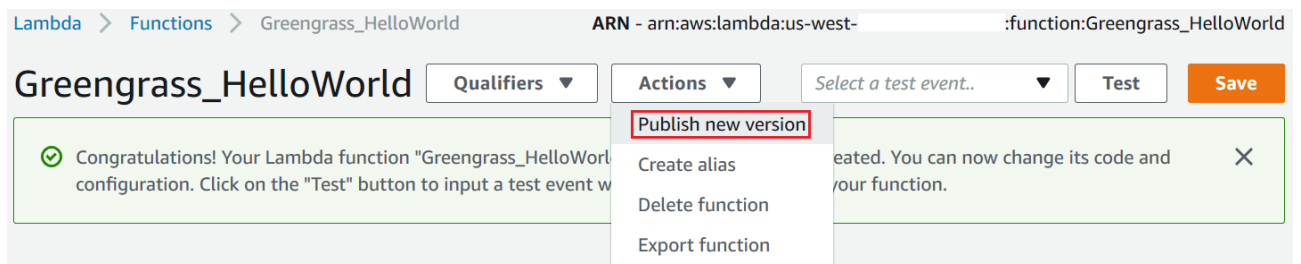
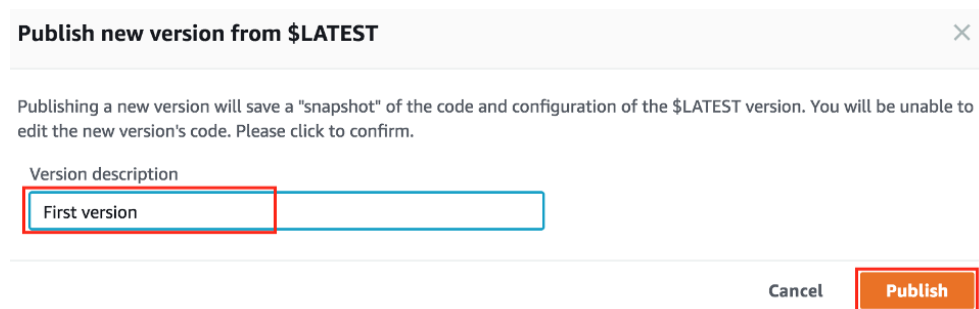
Key	Value
No environment variables	
No environment variables associated with this function.	

Edit

2. Choose **Save**

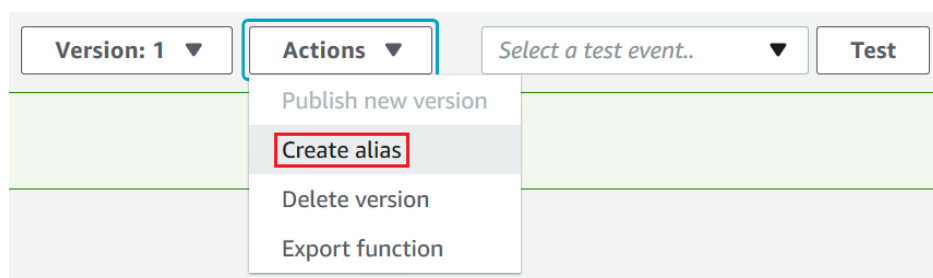
3. Publish the Lambda function:

- 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*

Description

Version*

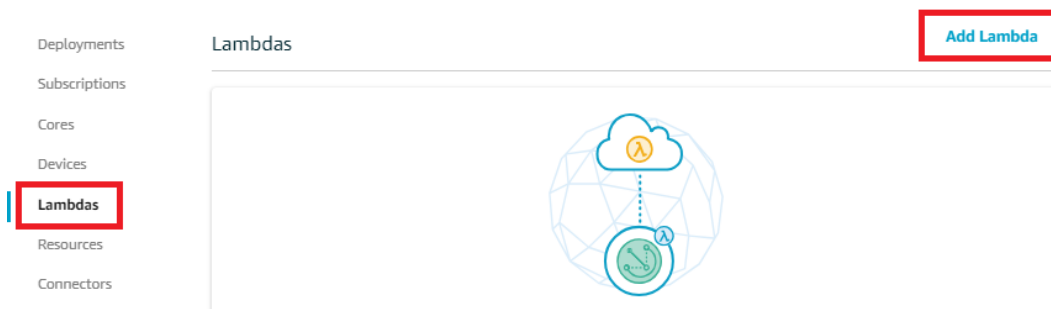
You can shift traffic between two versions, based on weights (%) that you assign. Click [here](#) to learn more.

Additional version

Cancel

Create

6. In the AWS IoT console, under Greengrass, choose Groups, and then choose the group that you created in above Steps.
 - 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.

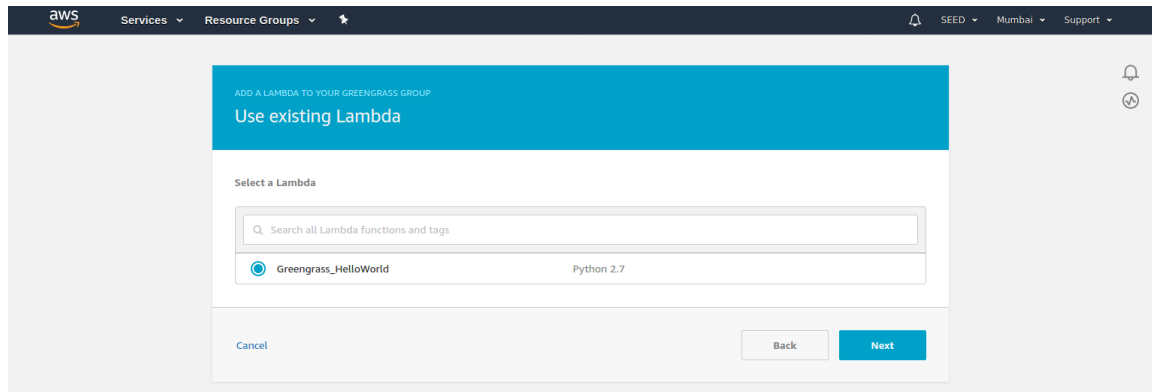
[Create new Lambda](#)

Use an existing Lambda function

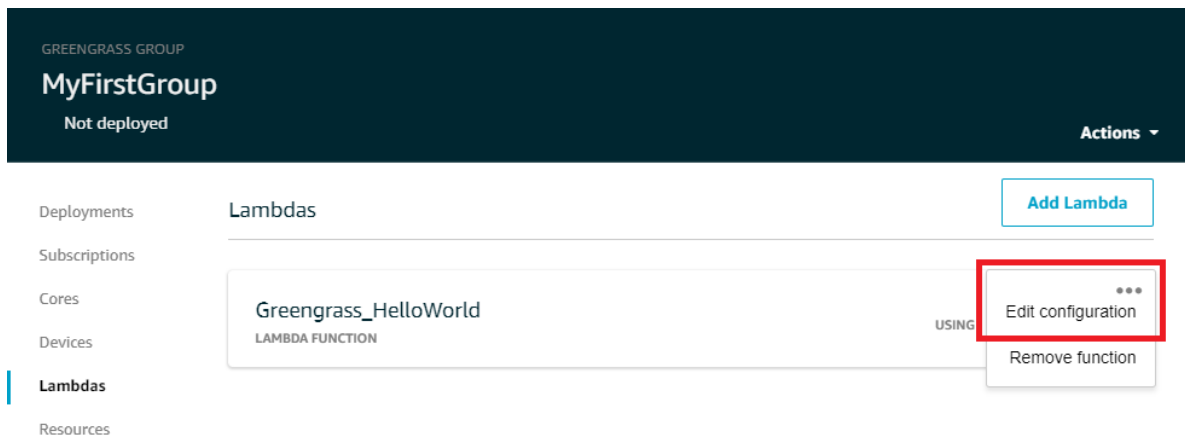
You will choose from a list of existing Lambda functions.

[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:



- On the Group-specific Lambda configuration page, make the following changes:
 - 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

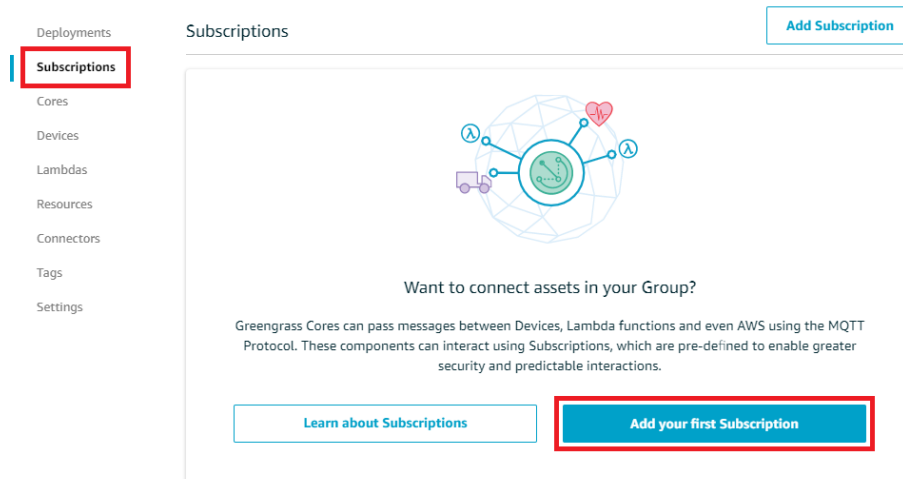
25 Second

Lambda lifecycle

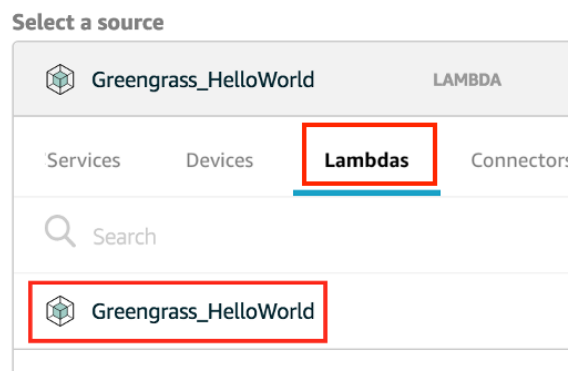
☐ On-demand function

☒ Make this function long-lived and keep it running indefinitely

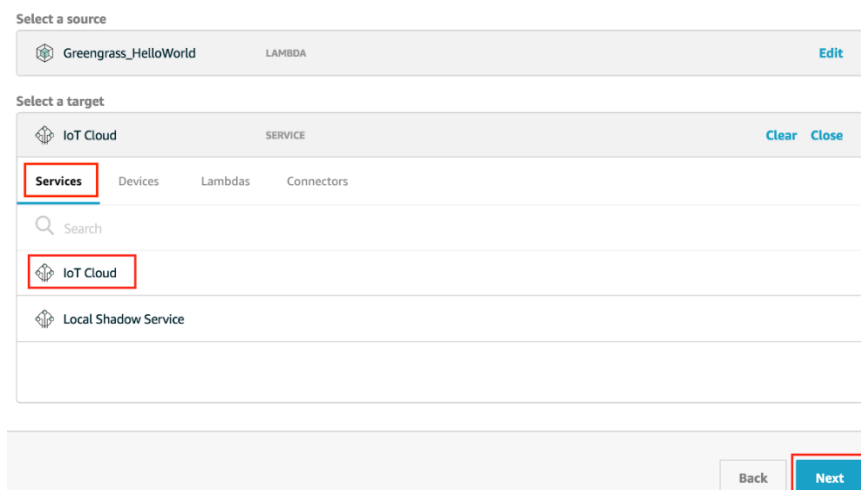
- Keep the default values for all other fields such as 'Run As', Containerization, and Input payload data type. Lastly, choose Update to save your changes.
8. On the **group configuration page**, choose **Subscriptions**, and then choose **Add your first Subscription**.



- When asked to Select a source, choose Select. Then on the **Lambdas** tab, choose **Greengrass_HelloWorld** as the source.



9. To Select a target, choose Select. Then on the Service tab, choose **IoT Cloud**, and then choose **next**.



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

Source

Greengrass_HelloWorld LAMBDA

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

hello/world

Target

IoT Cloud SERVICE

Back Next

- Choose **Finish**

10. Configure the group's logging settings. Users 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 execute the following command:

```
root@stm32mp1-av96:~# 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.

```
root@stm32mp1-av96:~# ps aux | grep -E "greengrass.*daemon"
root  4471  7.5  2.8 954812 25228 ttySTM0  Sl   11:30   0:07 /greengrass/ggc/packages/1.10.0/bin/daemon -core-dir /greengrass/ggc/packages/1.10.0 7
root  4819  0.0  0.0  1776   352 ttySTM0  S+   11:32   0:00 grep -E greengrass.*daemon
root@stm32mp1-av96:~#
```

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyUSB0

- To start the daemon, use these commands:

```
root@stm32mp1-av96:~# cd /greengrass/ggc/core/
root@stm32mp1-av96:~# ./greengrassd start
```

- In the **AWS IoT console**, on the **group configuration page**, from Actions, choose **Deploy**.



- On the page “Configure how devices discover your core”, choose Automatic detection. 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 prompted for the discovery method for first time when group is deployed.

Automatically detect Core endpoints (recommended)

Greengrass will detect and override connection information as it changes.

Automatic detection

Manually configure Core endpoints

Manually manage connection information. This can be accessed via your Core device's settings.

Manually configure

- The first deployment might take a few minutes. When the deployment is complete, one can see ‘Successfully completed’ in the Status column on the Deployments page:

The screenshot shows the AWS IoT Greengrass console interface. At the top, there's a navigation bar with 'Services', 'Resource Groups', and a search icon. Below this, the main header shows 'GREENGRASS GROUP' and 'MyFirstGroup_TPM' with a status indicator 'Successfully completed' and an 'Actions' dropdown. The left sidebar contains a navigation menu with options like Subscriptions, Cores, Devices, Lambdas, Resources, Connectors, Tags, and Settings. The main content area is titled 'Group history overview' and features a table of deployments. The table has columns for 'Deployed', 'Version', and 'Status'. The 'Status' column is highlighted with a red box, showing 'Successfully completed' for several deployments. The table also includes a 'By deployment' dropdown and three dots for each row.

Deployed	Version	Status
Apr 20, 2020 6:59:45 PM +0530	a6a22fc8-f518-4a48-a147-00be9b1157ee	Successfully completed...
Apr 20, 2020 3:26:48 PM +0530	a6a22fc8-f518-4a48-a147-00be9b1157ee	Successfully completed...
Apr 20, 2020 3:18:33 PM +0530	baf99c68-b526-4650-0e0f-df447ddb2fc	Failed
Apr 20, 2020 3:10:02 PM +0530	1f2a0916-3a07-4c4a-b7ad-1a270fd96b0	Successfully completed...
Apr 20, 2020 2:46:19 PM +0530	2e1a9f5e-c7e4-487b-b451-9243d9630596	Successfully completed...
Apr 20, 2020 12:51:46 PM +0530	2e1a9f5e-c7e4-487b-b451-9243d9630596	Successfully completed...
Apr 20, 2020 12:29:03 PM +0530	2e1a9f5e-c7e4-487b-b451-9243d9630596	Failed
Apr 20, 2020 12:02:25 PM +0530	a9c8543f-cb37-47a6-bd1e-87a360ff6dfo	Failed

- Verify the Lambda Function Is Running on the Core Device with hardware Security enabled with TPM 2.0 security keys.
- 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

Subscribe
Devices publish MQTT messages on topics. You can use this client to subscribe to a topic and receive these messages.

Subscription topic
 Subscribe to topic

Max message capture ?

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 following to the hello/world topic]

- You should see MQTT messages on the screen like “Message from Avenger96”

MQTT client ? Connected as iotconsole-1587574243130-0

Subscriptions hello/world Export Clear Pause

Subscribe to a topic
Publish to a topic

hello/world ×

Publish
Specify a topic and a message to publish with a QoS of 0.
 Publish to topic

```

1
2
3
{"message": "hello from AWS IoT console"}

```

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

```

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

```

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

5 STM32MP15 SECURE BOOT

This section is a practical example to illustrate the construction of a secure boot image and to configure the target device to run securely, which is possible because of the Trusted Firmware-A (TF-A) and the [STM32 KeyGen tool](#). This document targets the Secure Boot feature on the following applications processors from the [STM32MP157CAC](#) family.

This application not only demonstrates the secure boot solution on the Avenger96 and [STM32MP157CAC processors](#), but also some trusted features for secure boot. It focuses on:

- STM32MP boot sequence.
- Secure boot implementation
- Authentication processing
- Key generation
- Key registration
- Image signing
- Image programming
- Authentication
- Closing the device

Target audience:

User with knowledge on normal booting process, familiar with signing image tools, and fuse related concepts. User should be familiar with the basics of digital signatures and public key certificates.

[Note: For a step-by-step technical guide, please refer to this [Avenger96 Secure Boot Reference](#)]

[Note: See Appendix **Section 6.5** for more information on secure boot environment]

5.1 Secure boot implementation

5.1.1 Overview

STM32 MPU provides authentication processing with the ECDSA [\[1\]](#) verification algorithm, based on ECC [\[2\]](#). ECDSA offers better results than RSA with a smaller key. STM32 MPU relies on a 256 bits ECDSA key.

Two algorithms are supported for ECDSA calculation:

- P-256 NIST
- Brainpool 256

The algorithm selection is done via the signed binary header, as shown in [STM32 header](#) (subchapter in this same article).

The ECDSA verification follows the process below:

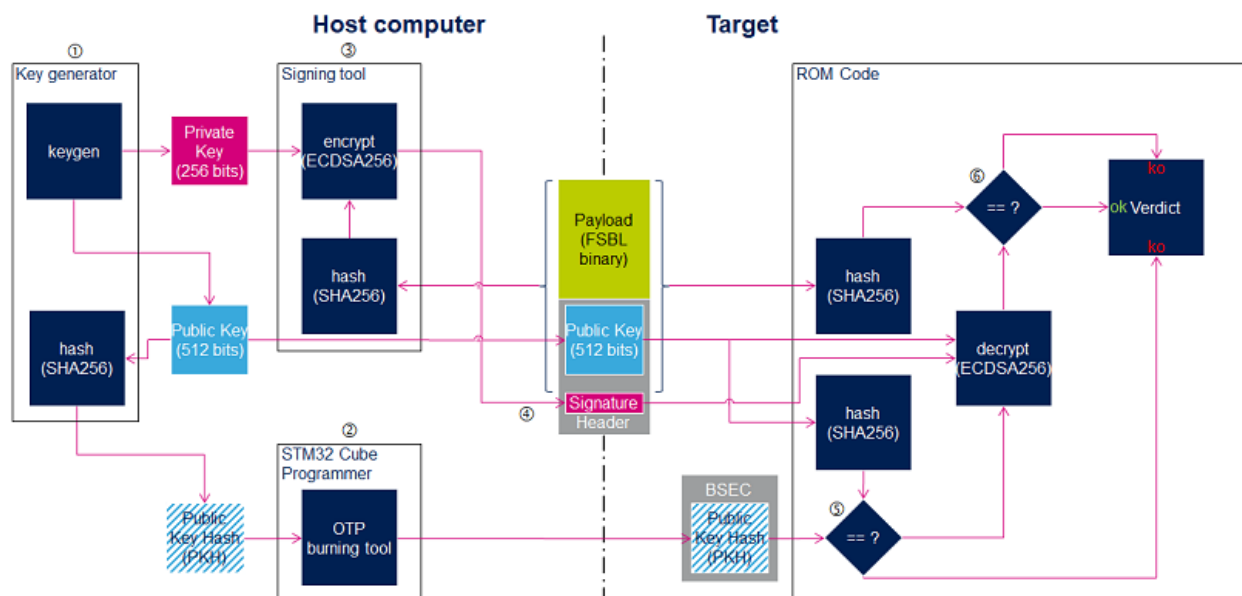


Figure 3: Secure boot process flow

5.1.2 Key Generation

First step is to generate the ECC pair of keys with STM32 KeyGen tool. This is the key pair that will be used to sign the images.

The tool also generates a third file containing the public key hash (PKH) that will be used to authenticate the public key on the target.

5.1.2.1 Install STM32MP Key Generator

The STM32MP Key Generator software is tested on Ubuntu 14.04 and 16.04, both 32-bit and 64-bit, and should work on any distribution.

To run the STM32MP Key Generator tool, you need to first download and install the [STM32 Cube Programmer](#). To execute, launch the `./STM32MP_KeyGen_CLI` script.

5.1.2.2 STM32MP Key Generator command line interface

On the Linux System with STM32 KeyGen Tool installed one can use it to generate your ECC key pair. Go to path mention below and use STM32MP_KeyGen_CLI command tool.

```
Linux-PC $ cd /home/<username>/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin
```

Make a new directory with name "secure_keys" by running the below command.

```
Linux-PC $ mkdir /home/user/secure_keys
```

Generate key pair using the following command

```
Linux-PC $ ./STM32MP_KeyGen_CLI -ecc 2 -pubk /home/user/secure_keys/public.pem -prvk /home/user/secure_keys/private.pem -hash /home/user/secure_keys/pubKeyHash.bin -pwd seed
```

User will see the following on the screen if key pair generated successfully, please note down the key type.

STM32MP Key Generator v1.0.0

Prime256v1 curve is selected.
 AES_256_cbc algorithm is selected for private key encryption
 Generating Prime256v1 keys...
 Private key PEM file created
 Public key PEM file created
 public key hash file created
 Keys generated successfully.

+ public key: /home/<username>/secure_keys/public.pem
 + private key: /home/<username>/secure_keys/private.pem
 + public hash key: /home/<username>/secure_keys/pubKeyHash.bin

User can verify the generated key curve-type to sign the image with the same curve

Linux-PC \$ **sudo openssl ec -in private.pem -text**

Private-Key: (256 bit)

priv:

85:73:e4:29:14:28:ec:e2:80:a8:79:30:0b:2f:80:
 b2:c9:0b:fa:af:4f:71:df:fb:80:a6:81:0c:de:0b:
 c0:5b

pub:

04:1e:19:71:22:4b:15:81:cd:cb:3d:cf:07:69:3b:
 ce:25:62:21:98:ad:f5:7e:52:9e:50:a8:e9:9c:8f:
 1d:83:bf:e2:a1:58:e7:8d:a7:ce:e1:21:3d:66:2c:
 86:17:0a:ab:f8:22:83:62:b2:6a:c4:f9:30:48:d5:
 25:b4:df:1a:e4

ASN1 OID: **prime256v1**

NIST CURVE: **P-256**

writing EC key

-----BEGIN EC PRIVATE KEY-----

MHCaQEEIIVz5CkUKOzigKh5MAsvgLLJC/qvT3Hf+4CmgQzeC8BboAoGCCqGSM49
 AwEHoUQDQgAEHhIxlksVgc3LPc8HaTvOJWIhmK31flKeUKjpnI8dg7/ioVjnjafo
 4SE9ZiyGFwqr+CKDYrJqxPkwSNUltN8a5A==
 -----END EC PRIVATE KEY-----

5.1.2.3 Extending Public key Hash to bootfs in SD-card

Insert Avenger96 Image Flashed SD-card into your Linux PC.
 Verify the nodes created for SD-card in the /dev directory

Linux-PC \$ **ls -l /dev/sd***

Copy the pubKeyHash.bin in to the bootfs SD-card partition and boot the board

Linux-PC \$ **cp /home/<username>/secure_keys/pubKeyHash.bin /media/<username>/bootfs/**

5.1.2.4 Verify Public Key Hash

User should verify the value after reading this pubKeyHash.bin before fusing and locking the Key registration

```
Linux-PC $ xxd -g1 /home/ <username> /secure_keys/pubKeyHash.bin
```

Users should check and validate this output against the file copied to bootfs with a mmc read command in the following steps

```
00000000: eecb48dc c42541ee 3de223ff dd7a9976 ..H..%A.=.#..z.v
00000010: 3eaea651 1b52001b 3a0bc3c6 0c190365 >..Q.R.....e
```

5.1.3 Key Registration

5.1.3.1 Register hash public key

First step to enabling the authentication is to burn the [OTP WORD 24 to 31](#) in [BSEC](#) with the corresponding public key hash (PKH, output file from [STM32 KeyGen](#)). OpenSTLinux embeds a **stm32key** tool that can be called from [U-Boot command line interface](#) to program the PKH into the OTP.

PKH file (pubKeyhash.bin) must be available in a file system partition (like bootfs) on a storage device (like SD-card) before proceeding.

Insert the SD-card into the Avenger96 board, power on the board and pause the U-boot console by pressing any key while U-Boot log is seen on the serial debug console

Load hash file from mmc 0 partition 4 (ext4) in DDR

```
root@stm32mp1-av96:~#
$ STM32MP> ext4load mmc 0:4 0xc0000000 pubKeyhash.bin
32 bytes read in 0 ms
```

Read loaded key from DDR to confirm it is valid (without writing it in OTP)

```
root@stm32mp1-av96:~#
$ STM32MP> stm32key read 0xc0000000

OTP value 24: eecb48dc
OTP value 25: c42541ee
OTP value 26: 3de223ff
OTP value 27: dd7a9976
OTP value 28: 3eaea651
OTP value 29: 1b52001b
OTP value 30: 3a0bc3c6
OTP value 31: c190365
```



If hash key is ok, the key in OTP can be fused

```
root@stm32mp1-av96:~#
$ STM32MP> stm32key fuse -y 0xc0000000
```

5.1.4 Signing the FSBL and SSBL

In the following steps you will be editing the FSBL and SSBL files which are found here:

FSBL: tf-a-stm32mp157a-av96-trusted.stm32

Yocto Path: Avenger96/build-openstlinuxweston-stm32mp1-av96 /tmp-glibc/deploy/images/stm32mp1-av96/tf-a-stm32mp157a-av96-trusted.stm32

SSBL: u-boot-stm32mp157a-av96-trusted.stm32

Yocto Path: Avenger96/build-openstlinuxweston-stm32mp1-av96 /tmp-glibc/deploy/images/stm32mp1-av96/u-boot-stm32mp157a-av96-trusted.stm32

For more info about Avenger96 Image Partition please review the Yocto file named “[README.HOW_TO.txt](#)” found in this directory or path:

Avenger96/layers/meta-st/meta-st-stm32mp/recipes-bsp/trusted-firmware-a/tf-a-stm32mp/

Follow the steps to sign the images:

- copy “tf-a-stm32mp157a-av96-trusted.stm32” binary from Avenger96 Yocto build directory path to /home/user/secure_keys/

```
Linux-PC $ cp Avenger96/build-openstlinuxweston-stm32mp1-av96 /tmp-
glibc/deploy/images/stm32mp1-av96/tf-a-stm32mp157a-av96-trusted.stm3
/home/user/secure_keys/
```

- copy “u-boot-stm32mp157a-av96-trusted.stm32” binary from Avenger96 Yocto build directory path to /home/user/secure_keys/

```
Linux-PC $ cp Avenger96/build-openstlinuxweston-stm32mp1-av96 /tmp-
glibc/deploy/images/stm32mp1-av96/ u-boot-stm32mp157a-av96-trusted.stm32
/home/user/secure_keys/
```

- Go to path /home/<user>/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin

```
Linux-PC $ cd /home/<username>/STMicroelectronics/STM32Cube/STM32CubeProgrammer/bin
```

- Now proceed to sign the Image using STM32MP_SigningTool_CLI tool.

5.1.4.1 SSBL signing

```
Linux-PC $ sudo ./STM32MP_SigningTool_CLI -bin /home/<user>/secure_keys/u-boot-stm32mp157a-
av96-trusted.stm32 -pubk /home/<user>/secure_keys/public.pem -prvk
/home/<user>/secure_keys/private.pem -pwd seed -o /home/<user>/secure_keys/u-boot-
stm32mp157a-av96-trusted-signed.stm32
```

Prime256v1 curve is selected.

Reading Private Key File...

ECDSA signature generated.

signature verification: SUCCESS

The Signed image file generated successfully:

```
/home/<user>/secure_keys /u-boot-stm32mp157a-av96-trusted-signed.stm32
```

5.1.4.2 FSBL signing

```
Linux-PC $ sudo ./STM32MP_SigningTool_CLI -bin /home/<user>/secure_keys/tf-a-stm32mp157a-av96-trusted.stm32 -pubk /home/<user>/secure_keys/public.pem -prvk /home/<user>/secure_keys/private.pem -pwd seed -o /home/<user>/secure_keys/tf-a-stm32mp157a-av96-trusted-signed.stm32
```

Prime256v1 curve is selected.

Reading Private Key File...

ECDSA signature generated.

signature verification: SUCCESS

The Signed image file generated successfully:

/home/<user>/secure_keys/tf-a-stm32mp157a-av96-trusted-signed.bin

Note the curve used to sign the images in these steps matches the curve used to generate the keys in section 5.1.2.2. These curves must match for the signing and locking process to succeed.

5.1.5 Flash the Signed Image

Once the images are signed, they can be flashed to the target board

1. From the mounted SD-card, identify the FSBL and SSBL partition
2. Typically, the partitions will map in the following order

```
bootfs -> ../../mmcb1k0p4
```

```
fsbl1 -> ../../mmcb1k0p1 → FSBL (TF-A)
```

```
fsbl2 -> ../../mmcb1k0p2 → FSBL backup (TF-A backup—same content as FSBL)
```

```
rootfs -> ../../mmcb1k0p5
```

```
ssbl -> ../../mmcb1k0p3 → SSBL (U-Boot)
```

```
userfs -> ../../mmcb1k0p6
```

3. Signed-FSBL will be flashed at /dev/sdx1 and /dev/sdx2
4. Signed-SSBL will be flashed at /dev/sdx3
5. Flash the newly signed images using the following commands

FSBL:

```
sudo dd if=tf-a-stm32mp157a-av96-trusted-signed.stm32 of=/dev/sdb1 bs=1M conv=fdatasync status=progress && sync
```

```
sudo dd if=tf-a-stm32mp157a-av96-trusted-signed.stm32 of=/dev/sdb2 bs=1M conv=fdatasync status=progress && sync
```

SSBL:

```
sudo dd if=u-boot-stm32mp157a-av96-trusted-signed.stm32 of=/dev/sdb3 bs=1M conv=fdatasync status=progress && sync
```

5.1.6 Verify Authentication

5.1.6.1 Bootrom Authentication

Using a **signed** binary, the ROM code authenticates and starts the FSBL.

If the authentication fails, the ROM code enters into a serial boot loop indicated by the blinking Error LED (see reference: [Bootrom common debug and error cases](#))

The [ROM code](#) provides secure services to the FSBL for image authentication with the same ECC pair of keys, so there is no need to support ECDSA algorithm in FSBL.

5.1.6.2 TF-A authentication

TF-A is the FSBL used by the trusted boot chain. It oversees loading and verifying U-boot and (if used) OP-TEE image binaries.

Each time a **signed** binary is used, TF-A will print the following status:



```
INFO: Check signature on Non-Full-Secured platform
```

If the image authentication fails the boot stage traps the CPU and no more trace is displayed.

5.1.7 Close the device

Notice that this last step is not shown in the diagram above, Figure 3.

Without any other modification, the device performs image authentication. However, non-authenticated images can still be used and executed: The device is still 'open'. This is a kind of test mode to check that the PKH is properly set.

As soon as the authentication process is confirmed, the device can be closed, and the user forced to use signed images.

[OTP WORD0](#) bit 6 is the OTP bit that closes the device. Burning this bit will lock authentication processing and force authentication from the Boot ROM permanently. Non signed binaries will not be supported anymore on the target.

To program this bit, the [STM32CubeProgrammer](#) or [U-Boot command line interface](#) can be used. Here is how to proceed with U-Boot:



```
Board $> fuse prog 0 0x0 0x40
```



Once this bit is written the platform is locked

5.2 Measured Boot with OPTIGA™ TPM2.0

5.2.1 Measured boot Step to verify the platform Integrity

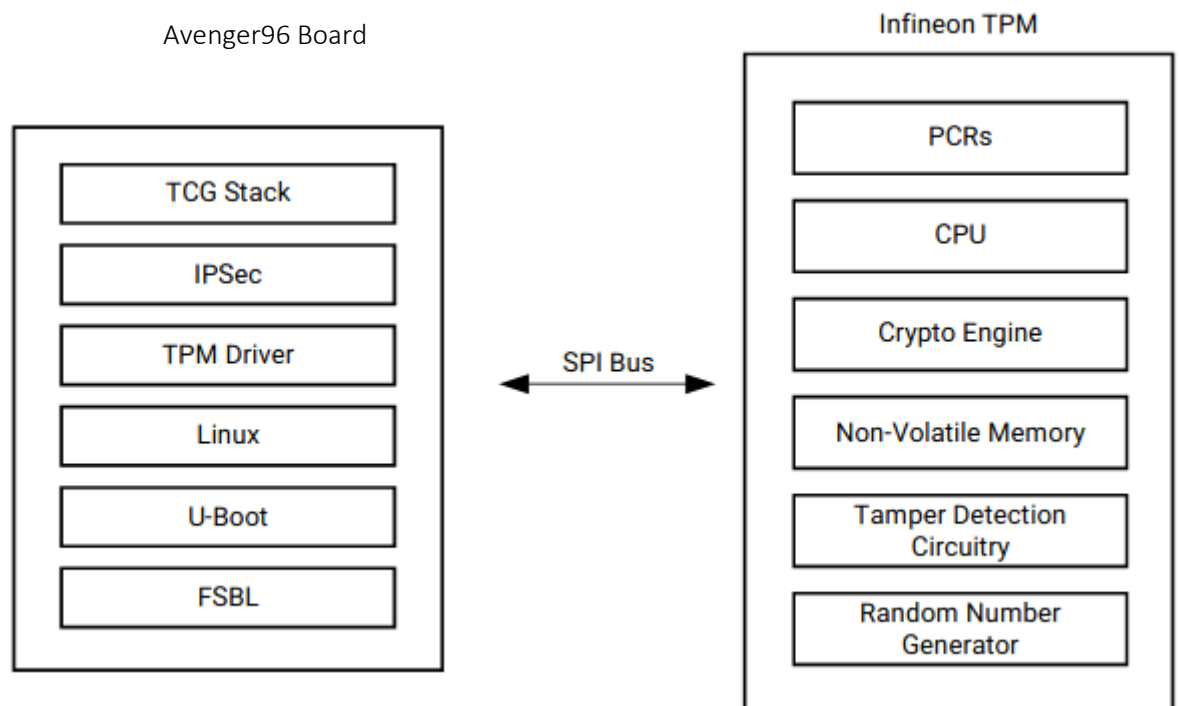


Figure 4: Measured Boot flow between TPM and Avenger Board.

At power-up, the Avenger96 device CSU ROM code loads the FSBL. The FSBL loads U-Boot, and U-Boot loads the Linux kernel, root file system, device tree and Linux application software. In one approach to booting with a chain of trust, the following steps occur:

1. The device hardware measures the Kernel Image.
2. The Kernel Image being authenticated is hashed.
3. The PCR API pushes the Kernel Image Hash measurement to the TPM.
4. The authenticated measure is stored to TPM NVRAM securely.
5. The PCR API pushes measurements to the TPM.
6. The Booting authenticates/measures Kernel Image using the TPM.
7. At every boot time, the device will measure the Linux partitions using the TPM and Verifies the integrity check with original value stored in Tamper Proof TPM NV area.

Measuring Kernel Image Hash

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

```
root@stm32mp1-av96 $ sha256sum /boot/ulimage | cut -d' ' -f1 >> kernel_hash
```

Extending Measured Hash to PCR

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

```
root@stm32mp1-av96 $ tpm2_pcrextend 16:sha256=$kernel_hash
```

Measuring Content from Specified PCR

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

```
root@stm32mp1-av96 $ tpm2_pcrlist -L sha256:16 -o pcr_kernel_original.bin
```

Generating TPM Measured Boot Policy

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

```
root@stm32mp1-av96 $ tpm2_createpolicy --policy-pcr -L sha256:16 -F pcr_kernel_original.bin -o policy_pcr_kernel_original.out
```

Define NV Area in TPM for PCR Storage

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

```
root@stm32mp1-av96 $ tpm2_nvdefine -x 0x1500016 -a 0x40000001 -s 32 -L policy_pcr_kernel_original.out -b "policyread|policywrite|authread|authwrite|ownerwrite|ownerread"
```

Extending PCR value to Secure NV RAM

Storing the PCR value, the Specified NV index defined

```
root@stm32mp1-av96 $ tpm2_nvwrite -x 0x1500016 -a 0x1500016 -P pcr:sha256:16 pcr_kernel_original.bin
```

Verifying Platform Integrity Check

Again, reboot the Board and perform below steps

```
root@stm32mp1-av96 $ sha256sum /boot/ulmage | cut -d' ' -f1 >> Measure_kernel_hash
root@stm32mp1-av96 $ tpm2_pcrextend 16:sha256=$kernel_hash
root@stm32mp1-av96 $ tpm2_pcrlist -L sha256:16 -o pcr_kernel_measured.bin
root@stm32mp1-av96 $ tpm2_nvread -x 0x1500016 -a 0x1500016 -s 32 >> pcr_kernel_original.bin
root@stm32mp1-av96 $ cmp pcr_kernel_measure.bin and pcr_kernel_original.bin
```

If values are similar, the Platform is secure and no tamper has occurred.

6 APPENDIX

6.1 Avenger – 96 Boards

The STM32MP157A is a highly integrated multi-market system-on-chip designed to enable secure and space constraint applications within the Internet of Things. Avenger96 board features dual Arm Cortex-A7 cores and an Arm Cortex-M4 core. In addition, an extensive set of interfaces and connectivity peripherals are included to interface to cameras, touchscreen displays and MMC/SD-cards. It also fully supports wireless communication, including WLAN and BLE.

Arrow's Avenger96 module integrates the high-end STM32MP157 module, which offers dual 650MHz Cortex-A7 cores and a 209MHz Cortex-M4 chip with an FPU, MPU, and DSP instructions. The STM32MP157 model includes the optional 533MHz Vivante 3D GPU with support for OpenGL ES 2.0 and 24-bit parallel RGB displays at up to WXGA (1280×800) at 60fps. This is also the only model with MIPI-DSI support.

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

Processor

- STM32MP157AAC to be replaced with Chipset **STM32MP157CAC** in order to enable secure boot feature
- 2x ARM®Cortex-A7 up to 650 MHz
- 1x ARM®Cortex-M4 up to 200 MHz
- 1x 3D GPU Vivante® @ 533 MHz -OpenGL® ES 2.0

Memory/Storage

- eMMC v4.51: 8 Gbyte SD 3.0 (UHS-I)
- QSPI: 2Mbyte
- EEPROM: 128 byte
- microSD Socket: UHS-1 v3.01
- RAM: 1024 Mbyte @ 533MHz

I/O Interfaces

- Host: 2x type A 2.0 high-speed and OTG, 1x type micro-B 2.0 high-speed
- One 40-pin Low Speed (LS) expansion connector (UART, SPI, I2S, I2C x2, GPIO x12, DC power)
- One 60-pin High Speed (HS) expansion connector (4L-MIPI DSI, USB, I2C x2, 4LMIPI 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

Video

- HDMI: WXGA (1366x768)@ 60 fps, HDMI 1.4

Power, Mechanical and Environmental

- Power: +8.0V to +18V
- Dimensions: 85mm x 100mm
- 96Boards™ Consumer Edition standard dimensions specifications

- Operating Temp: 0 - 40 °C

Software

- U-Boot version: U-Boot 2018.09-stm32mp-r2
- Linux version: Linux stm32mp1-av96 4.14.48

Linux Distribution: ST OpenSTLinux Weston (A Yocto Project Based Distro)

6.2 AWS Greengrass

[AWS IoT Greengrass](#) software extends cloud capabilities to local devices. This allows the cloud-based management of application logic which can be used for any of the following and more:

- to collect and analyze data
- react autonomously to local events
- Communicate securely on local networks
- AWS Lambda functions and pre-built connectors to create server less 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 the cloud

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

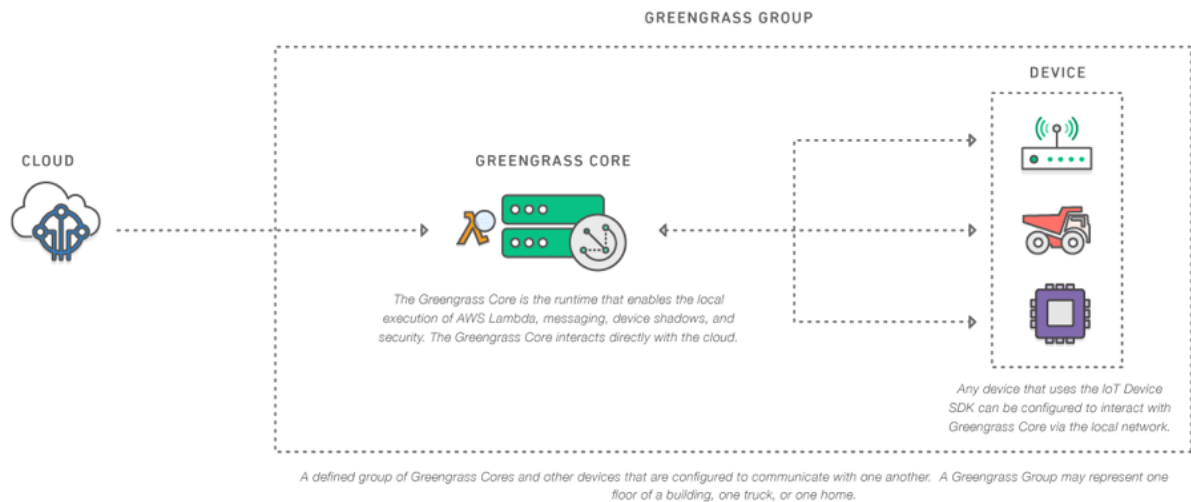


Figure 5: AWS Greengrass Group

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 board. The board is as shown in Figure 6.

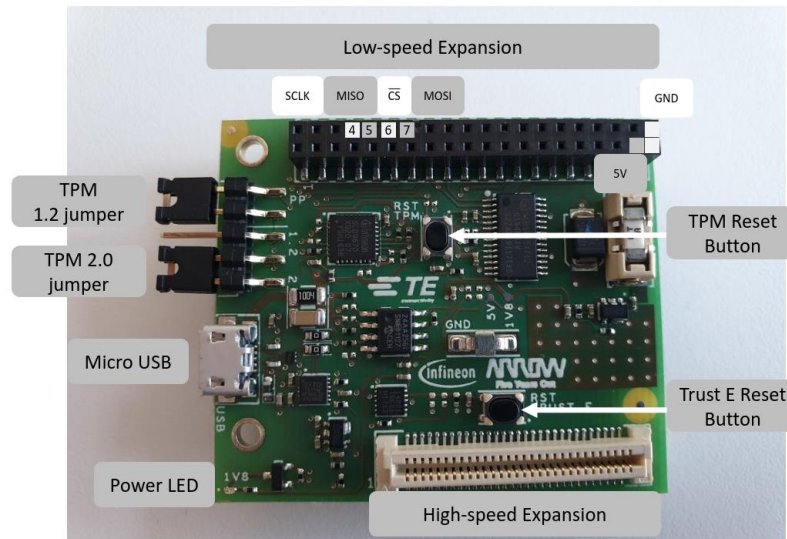


Figure 6: Tresor Mezzanine OPTIGA™ TPM 2.0

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

- The SLB9670x provides Trusted Platform Module (TPM) 2.0 functionality through SPI communication on the standard 96Boards LS 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.

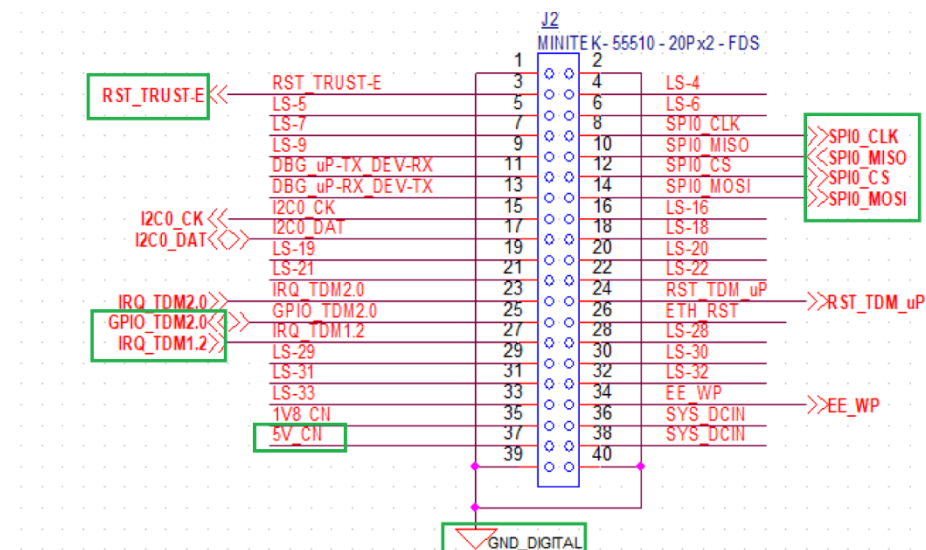


Figure 10: Tresor Mezzanine connector OPTIGA™ TPM 2.0

6.4 Trusted Platform Module – TPM

A cryptographic processor is present on most commercial PCs and servers. A typical crypto processor has three key cryptographic capabilities

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

Establishing a root of trust

A TPM can prevent a bootkit attack by providing a trusted sequence of boot operation.

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 must trust the layer below, as illustrated in the following diagram.

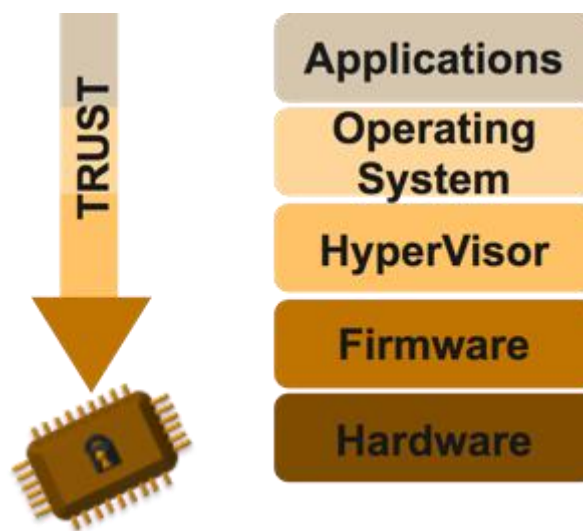


Figure 7: Root of Trust

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

A root of trust can be defined as any or 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 secured 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 these component identities can be verified later 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.

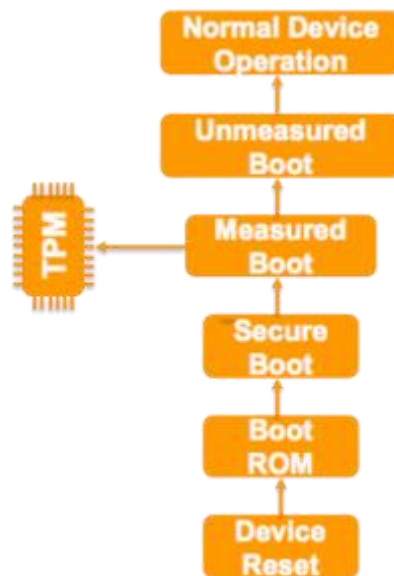


Figure 8: Measured/Trusted Boot Process

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

Device identification steps are comprised as follows:

- 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 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 TPMs to protect the edge gateways running AWS IoT Greengrass. This integration uses the PKCS#11 protocol as the interface to the TPM.

6.5 Boot chains Environment overview

6.5.1 Generic boot sequence

Starting Linux® on a processor is done in several steps that progressively initialize the platform peripherals and memories. These steps are explained in the following paragraphs and illustrated by the diagrams, which also gives typical memory sizes for each stage.

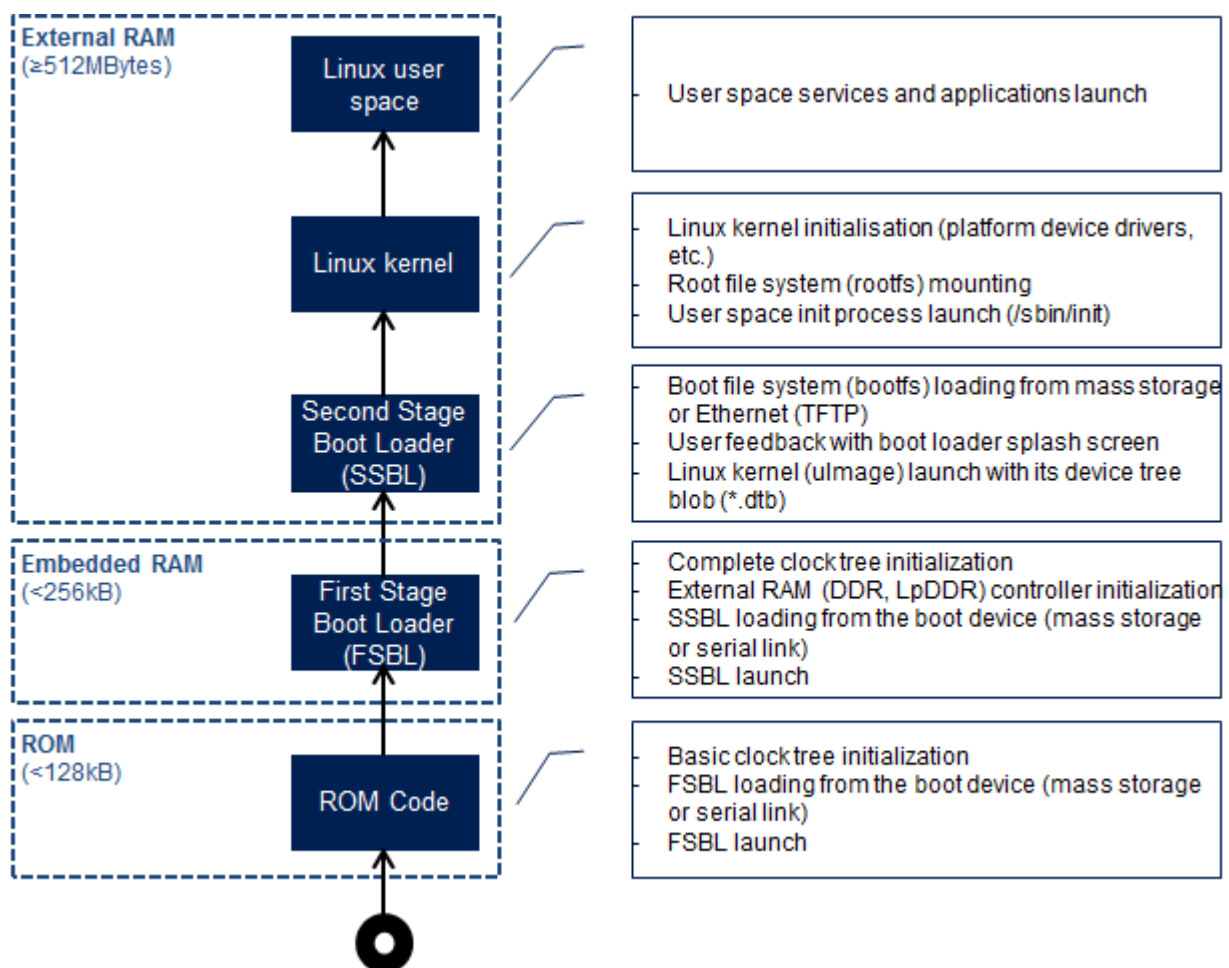


Figure 9: Generic Boot Sequence

6.5.2 STM32MP15 boot chain

6.5.2.1 Overview

STM32MP15 boot chain uses [Trusted Firmware-A \(TF-A\)](#) as the FSBL in order to fulfill all the requirements for security-sensitive customers, and it uses [U-Boot](#) as the SSBL. Note that the authentication is optional with this boot chain, so it can run on any STM32MP15 device [security](#) variant (that is, with or without the Secure boot).

Refer to the [security overview](#) for an introduction of the secure features available on STM32MP15, from the secure boot up to trusted applications execution.

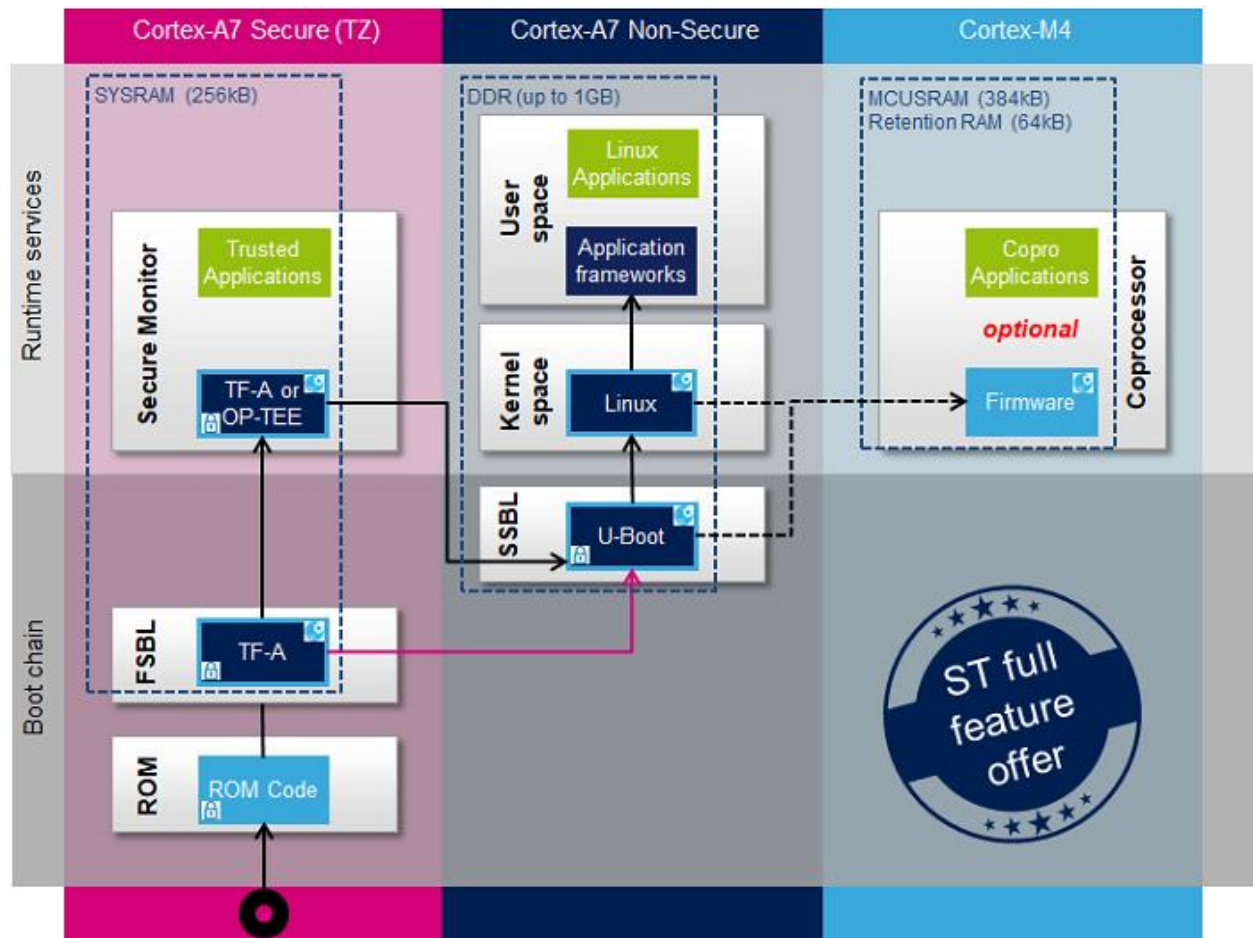


Figure 10: STM32MP15 boot chain

Note:

The STM32MP15 coprocessor can be started at the SSBL level by the [U-Boot early boot](#) feature, or later by the [Linux remoteproc framework](#), depending on the application startup time-targets.

6.5.2.2 ROM Code

The [ROM code](#) starts the processor in secure mode. It supports the FSBL authentication and offers authentication services to the FSBL.

6.5.2.3 First Stage Boot Loader (FSBL)

The FSBL is executed from the [SYSRAM](#).

Among other things, this boot loader initializes (part of) the clock tree and the [DDR controller](#). Finally, the FSBL loads the second-stage boot loader (SSBL) into the DDR external RAM and jumps to it.

[Trusted Firmware-A \(TF-A\)](#) is the FSBL used on the STM32MP15.

6.5.2.4 Second Stage Boot Loader (SSBL)

[U-Boot](#) is commonly used as a bootloader in embedded software and it is the one used on STM32MP15.

6.5.2.5 Linux

Linux® OS is loaded in DDR by U-Boot and executed in the non-secure context.

6.5.2.6 Secure OS / Secure Monitor

The Cortex-A7 secure world can implement a minimal secure monitor (from [TF-A](#) or [U-Boot](#)) or a real secure OS, such as [OP-TEE](#).

6.6 Building a Secure Signed Image

In the second step, FSBL and SSBL binaries must be signed. [STM32 Signing tool](#) allows the user to fill the STM32 binary header that is parsed by the embedded software to authenticate each binary.

- **STM32 Header**
Each binary image (signed or not) loaded by [ROM code](#) and by [TF-A](#) need to include a specific STM32 header added on top of the binary data. The header includes the authentication information.

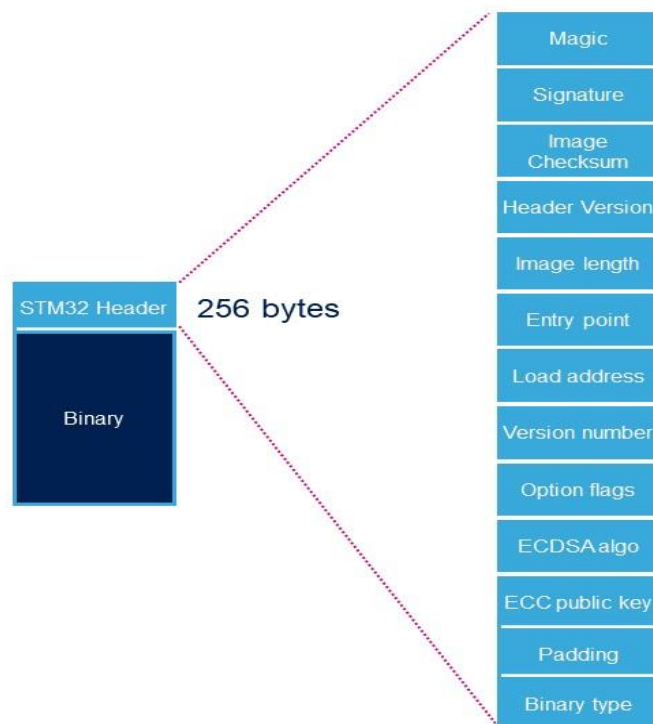


Figure 11: STM32 Image Header Description

Name	Length	Byte Offset	Description
Magic number	32 bits	0	4 bytes in big endian: 'S', 'T', 'M', 0x32 = 0x53544D32
Image signature	512 bits	4	ECDSA signature for image authentication ^[Note 1]
Image checksum	32 bits	68	Checksum of the payload ^[Note 2]
Header version	32 bits	72	Header version v1.0 = 0x00010000 Byte0: reserved Byte1: major version = 0x01 Byte2: minor version = 0x00 Byte3: reserved
Image length	32 bits	76	Length of image in bytes ^[Note 3]
Image entry Point	32 bits	80	Entry point of image
Reserved1	32 bits	84	Reserved
Load address	32 bits	88	Load address of image ^[Note 4]
Reserved2	32 bits	92	Reserved
Version number	32 bits	96	Image Version (monotonic number) ^[Note 5]
Option flags	32 bits	100	b0=1: no signature verification ^[Note 6]
ECDSA algorithm	32 bits	104	1: P-256 NIST ; 2: brainpool 256
ECDSA public key	512 bits	108	ECDSA public key to be used to verify the signature. ^[Note 7]
Padding	83 Bytes	172	Reserved padding bytes ^[Note 8] . Must all be set to 0
Binary type	1 Byte	255	Used to check the binary type 0x00: U-Boot 0x10-0x1F: TF-A 0x20-0x2F: OPTEE 0x30: Copro

Figure 12: STM32 Image Header Detailed Description

- Signature is calculated from first byte of header version field to last byte of image given by image length field.
- 32-bit sum of all payload bytes accessed as 8-bit unsigned numbers, discarding any overflow bits. Used to check the downloaded image integrity when signature is not used (if b0=1 in Option flags) as shown in above Figure 12.
- Length is the length of the built image; it does not include the length of the STM32 header, this field is not used by ROM code as shown in above Figure 12.
- Image **version number** is an anti-rollback monotonic counter. The ROM code checks that it is higher or equal to the monotonic counter stored in OTP as shown in above Figure 12.
- Enabling signature verification is mandatory on secure closed chips. field is an extract of PEM public key file that only kept the ECC Point coordinates x and y in a raw binary format ([RFC 5480](#)). This field will be hashed with SHA-256 and compared to the **Hash of pubKey** that is stored in OTP.
- This padding forces STM32 header size to 256 bytes (0x100) as shown in above Figure 12.

6.7 Measured boot Principles

Measuring boot is a way to inform the last software stage in case someone has tampered with the platform. It is impossible to know what exactly has been corrupted, but knowing simply knowing that tampering has occurred, 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://wiki.st.com/stm32mpu/index.php/STM32MP1_Distribution_Package
- [3] <https://git.yoctoproject.org/cgit/cgit.cgi/meta-security>
- [4] <https://github.com/dh-electronics/meta-av96/tree/master/meta-av96.thud>
- [5] <https://github.com/STMicroelectronics/meta-predmnt>
- [6] https://wiki.st.com/stm32mpu/wiki/How_to_integrate_AWS_IoT_Greengrass
- [7] <https://github.com/dh-electronics/manifest-av96>
- [8] <https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-dg.pdf>
- [9] <https://docs.aws.amazon.com/iot/latest/developerguide/register-CA-cert.html>
- [10] https://www.infineon.com/dgdl/Infineon-SLB%209670VQ2.0-DataSheet-v01_04-EN.pdf?fileId=5546d4626fc1ce0b016fc78270350cd6
- [11] <https://github.com/tpm2-software>
- [12] <https://www.96boards.org/product/avenger96/>
- [13] <https://www.96boards.org/product/tresor/>
- [14] https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
- [15] https://en.wikipedia.org/wiki/Elliptic-curve_cryptography
- [16] https://wiki.st.com/stm32mpu/wiki/STM32MP15_secure_boot#Purpose