Qualcomm Technologies, Inc.

Enable Secure Boot on SDM636/SDM660/ SDM630/SDM670 Chipsets

80-P8754-68 C

February 15, 2018

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	April 2017	Initial release
B	December 2017	▪ Added the SDM636 chipset in the title and relevant sections ▪ Updated new product name for QSEE in Chapter 2 ▪ Updated References in Section 3.7 and 14.1
C	February 2018	▪ Updated title and added SDM670 chipset ▪ Numerous changes were made to chapters 2, 3, 5, and 10, and appendices A and B; it should be read in its entirety. ▪ Updated References

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document describes the process to enable secure boot on the SDM636/SDM660/SDM630/SDM670 chipsets by blowing the eFuses and signing the images.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

If you are viewing this document using a color monitor, or if you print this document to a color printer, **red boldface** indicates code that is to be **added**, and ~~blue strikethrough~~ indicates code that is to be replaced or removed.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Secure boot overview

NOTE: Numerous changes were made in this chapter.

Secure boot refers to the bootup sequence that establishes a trusted platform for secure applications. It starts as an immutable sequence that validates the origin of the code using cryptographic authentication so that, only authorized software is executed. The bootup sequence places the device in a known security state and protects against binary manipulation of software and reflashing attacks.

A secure boot system adds cryptographic checks to each stage of the bootup process. This process asserts the authenticity of all secure software images that the device executes. This additional check prevents any unauthorized or maliciously modified software from running on the device. Secure boot is enabled through a set of hardware fuses. For the code to be executed, it must be signed by the trusted entity identified in the hardware fuses.

NOTE: Secure boot is not guaranteed without blowing an eFuse using Qualcomm fuse programmable read only memory (QFPROM); licensees must blow certain eFuses, which are described in this document.

To sign the images, a trusted vendor uses their private key to generate a signature on the raw code that they want to use, and adds this signature to the device alongside the software binary. The device contains the corresponding public key of the vendor, which is used to verify that the binary was not modified and that the trusted vendor in question provided it.

Images (the format in which the code is packaged) can be signed using the code signing management system (CSMS, refer to *Transitioning to Code Signing Management System* 2.0 (80-V8000-4)) by QTI, or by using the code signing system of the licensee. Signed images include the code signature and the certificate chain. The certificates carry the public keys used to decrypt the certificate and image signatures.

The certificate chain can have three certificates: attestation certificate → attestation certificate authority (CA) certificate → root certificate; or two certificates: attestation certificate → self-signed root certificate. For the two-certificate chain model, the root of the certificate chain must be self-signed, which need not be the case for the three-certificate chain model. Version 1, 2, or 3 X.509 certificate format is supported.

The bootup of a device comprises a multiple-stage process. Each image in the stage performs a specific function, and each image is verified by the previous image (for example, primary boot loader (PBL) → eXtensible boot loader (XBL) SEC → Qualcomm® Trusted Execution Environment (QTEE, formerly known as Qualcomm Secure Execution Environment (QSEE)). The root of trust (the most trusted entity that kicks off this process) is the PBL, which is on-chip code, and therefore already trusted and immutable. Before the next image in the bootup sequence is executed, the image is first authenticated to ensure that it contains authorized software. For example, control passes to the XBL only after the PBL successfully authenticates it. Since the

XBL is now trusted, it is trusted to authenticate the image. The images further establish the security of the device through their functionality.

With these pieces of cryptographic components and architecture in place, the authenticity of the bootup software and secure applications is validated.

Figure 2-1 is an overview of the secure boot architecture.

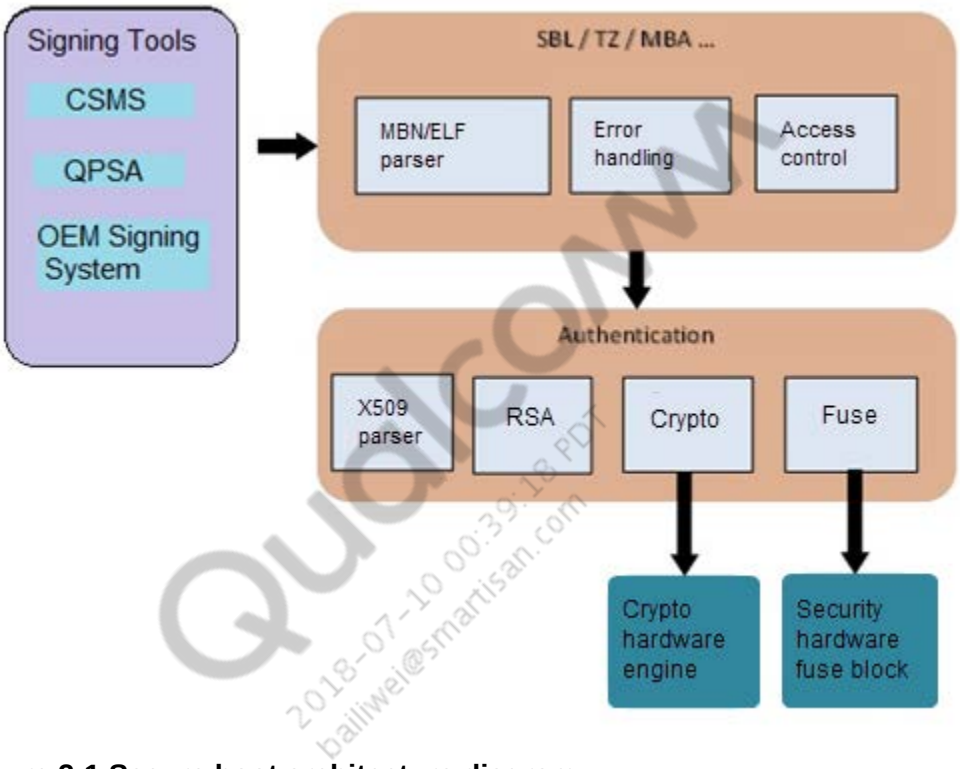


Figure 2-1 Secure boot architecture diagram

NOTE: CSMS is not available to customers automatically upon the setup of a Salesforce or CreatePoint account. More agreements and authorizations are required to access CSMS. Contact the sales account representative of your company for more information.

NOTE: CSMS 1.0 is being upgraded to CSMS2.0. Refer to *Transitioning to Code Signing Management System* 2.0 (80-V8000-4) for details on how to prepare for the transition.

2.1 SDM636/SDM660/SDM630/SDM670 secure boot enhancements

The following improvements over secure boot are implemented in the SDM636/SDM660/SDM630/SDM670 chipset:

- All the images are in standard ELF format, including modem boot authentication (MBA) and XBL. See Section 2.3.1 for ELF image format.
- Double image signing. See Section 2.6 for double image signing.
- PKCS#1 v2.1 RSASSA-PSS signature scheme. Refer to Section 2.3.2.1.

- Signing binary image for multiple families of chipset. Refer to Section 5.1 .

Note: Root of trust (ROT) transfer feature is not supported.

The following table lists the recommended RSA key size and signing algorithm for enabling secure boot for SDM636/SDM660/SDM630/SDM670

Table 2-1 Recommended RSA key size and signing algorithm

	To be removed and not accepted for SDM660/ SDM630/SDM636/SDM670	Recommended for SDM636/SDM660/SDM630/ SDM670	Recommended setting in Sectool
Code Signature -- Padding scheme.	Modified version of RSASSA-PKCS#1_v1.5. <rsa_padding> not specified default to RSASSA-PKCS#1_v1.5 padding.	RASSA-PSS	<rsa_padding>pss</rsa_padding>
Code Signature -- Digest Algorithm	HMAC (based on SHA1) <hmac>true</hmac> <hash_algorithm>sha1</hash_algorithm>	SHA256 <hmac>false</hmac> <hash_algorithm>sha256</hash_algorithm>	<hmac>false</hmac> <hash_algorithm>sha256</hash_algorithm>
Certificate Signature -- padding scheme.	RSASSA-PKCS#1_v1.5	RSASSA-PSS	
Certificate Signature -- Digest Algorithm	SHA1	SHA256	
Code and Certificate RSA Key modulus size	Modulus < = 2048 bits <key_size>2048</key_size>	Modulus >= 2048 <key_size>2048</key_size>	<key_size>2048</key_size>
Code and Certificate RSA public key exponent	F0 (e = 3) <exponent>3</exponent>	F4 (e=65,537) <exponent>65537</exponent>	<exponent>65537</exponent>

Recommended setting in <sectools>\config\sdm660\sdm660_secimage.xml are as following:

```
<key_size>2048</key_size>
<exponent>65537</exponent>
<hmac>false</hmac>
<rsa_padding>pss</rsa_padding>
```

2.2 Certificate and signature format

This section describes the expected format of the certificates and the code signature mandated by the authentication algorithms during bootup.

Following this scheme, licensees can create QTI-compatible code signing tools to sign the images.

2.2.1 Certificate profiles

A chained approach is used to authenticate an image. First, the certificate chain of the image is validated. For example, the XBL authenticates the applications boot loader (APPSBL) image using the certificate chain of APPSBL. The attestation certificate in the chain is authenticated against the intermediate certificate (attestation CA, which in turn is authenticated against the root CA (see Figure 2-2). The root certificate itself is authenticated against the hash provisioned in an immutable area – QFPROM or the key table of QTI residing in a trusted area.

When the certificate chain is validated, the code signature (signature D) is validated by first decrypting it using the public key in the attestation certificate (public key A).

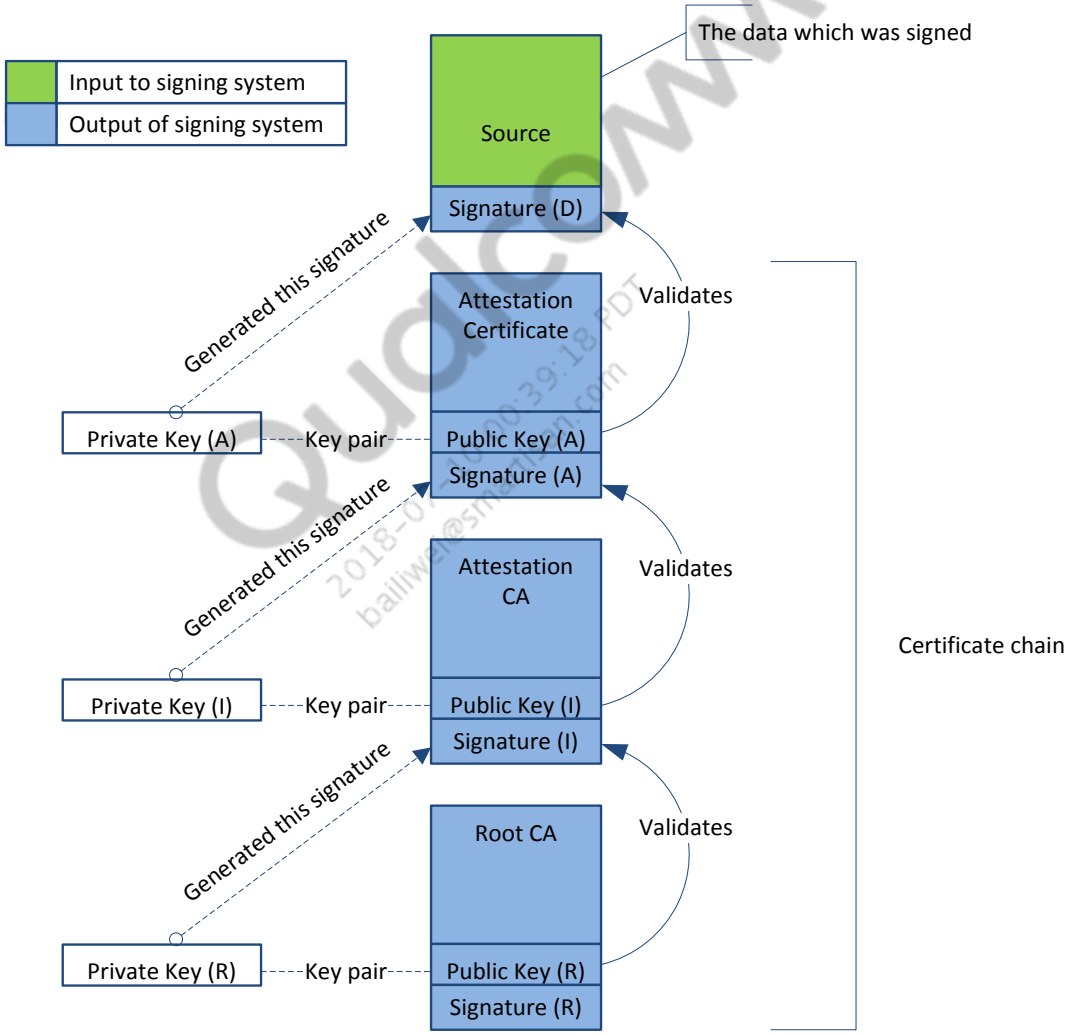


Figure 2-2 Certificate chain and signatures

All three certificates follow the ITU-T X.509 v3 format.

Three signatures are used for image authentication: one computed over the image source, one computed over the attestation certificate, and one computed over the attestation CA. The root signature itself is validated automatically when the entire root certificate is validated against the provisioned root certificate hash stored in the `OEM_PK_HASH` fuse or in the `BOOT ROM Hash` index.

NOTE: A two-certificate chain can also be used; it contains the attestation certificate followed by self-signed root certificate.

2.2.1.1 Root certificate

Table 2-2 shows the generic profile root certificate from QTI CSMS signing service. OEMs can follow similar profile for the OEM root certificate.

Table 2-2 Root certificate profile

Field	Settings
Subject DN Common name Organization Org Unit Locality State Country	 QTI Root CA 1 Qualcomm CDMA Technologies San Diego CA US
Version	V3
Serial number	Verisign specified
Signature algorithm	SHA256 RSA
Issuer	Same as subject DN
Valid from	Date/time issued
Valid to	20 years from issued date
Public key size	2048, with 65537 as exponent
Basic constraints	Subject type=CA Path length=None
Key usage	Certificate signing Offline certificate revocation list (CRL) signing CRL signing (06)
Authority key identifier	N/A
Subject key identifier	No
Certificate policies	Do not use
CRL distribution points	No
SubjectAltName	No
Extended key usage	No
Private extensions	No

2.2.1.2 Attestation CA certificate

Table 2-3 shows the generic profile attestation CA certificate from QTI CSMS signing service. OEMs can follow similar profile for the OEM attestation CA certificate.

Table 2-3 Attestation CA certificate profile

Field	Settings
Subject DN Common name Organization Org Unit Locality State Country	 QTI Attestation CA Qualcomm CDMA Technologies San Diego CA US
Version	V3
Serial number	Verisign specified
Signature algorithm	SHA256 RSA
Issuer	(Root subject DN)
Valid from	Date/time issued
Valid to	20 years from issued date
Public key size	2048, with 65537 as exponent
Basic constraints	Subject type=CA Path length=0
Key usage	Certificate signing Offline certificate revocation list (CRL) signing CRL signing (06)
Authority key identifier	Yes, KeyID=KeyIdentifier
Subject key identifier	Yes, the subjectKeyIdentifier extension is computed as a hash of part of the data in the certificate
Certificate policies	Do not use
CRL distribution points	No
SubjectAltName	No
Extended key usage	No
Private extensions	No

2.2.1.3 Attestation certificate

Table 2-4 shows the generic profile attestation certificate from QTI CSMS signing service. OEMs can follow similar profile for the OEM attestation certificates.

Table 2-4 Attestation certificate profile

Field	Settings
Subject DN Common name Organization Org Unit 1 Org Unit 2 Org Unit 3 Org Unit 4 Org Unit 5 Org Unit 6 Org Unit 7 Org Unit 8 Org Unit 9 Org Unit 10 Org Unit 11 Org Unit 12 Org Unit 13 Org Unit 14 Locality State Country	 QTI developer Company name SW_ID (mandatory, varies per image) HW_ID (mandatory) OEM_ID_INDEPENDENT (Reserved for QTI use) OEM_ID (mandatory, default 0x0000) SW_SIZE (mandatory) MODEL_ID (mandatory, default 0x0000) SHA256 (0000 for sha1, 0001 for sha256) APP_ID (optional, for TrustZone apps images, varies per image) CRASH_DUMP (optional) ROT_EN (Not applicable) SOC_HW_VERSION (optional, for SDM660/630 and later) MASK_SOC_HW_VERSION (deprecated) IN_USE_SOC_HW_VERSION (optional, 0 or 1, use with SOC_HW_VERSION) USE_SERIAL_NUMBER_IN_SIGNING (optional, 0 or 1) San Diego CA US *Note: Sectools always uses SHA256 unless a customized certificate is passed to Sectools.
Version	V3
Serial number	Verisign specified
Signature algorithm	SHA256 RSA
Issuer	Same as subject DN
Valid from	Date/time issued
Valid to	20 years from issued date
Public key size	2048, with 65537 as exponent
Basic constraints	Subject type=End entity Path length=0
Key usage	Certificate signing Offline certificate revocation list (CRL) signing CRL signing (06)
Authority key identifier	Yes, KeyID=KeyIdentifier
Subject key identifier	No
Certificate policies	No
CRL distribution points	http://crl.geotrust.com/crls/qctattest.crl
SubjectAltName	No
Extended key usage	No
Private extensions	None

2.2.1.4 OU fields

The attestation certificate has Organizational Unit (OU) fields in the Subject DN field. The OU fields contain information on the image that was signed and what values were used to generate the signature.

Sample Subject DN field

```
OU = 01 0000000200000009 SW_ID
OU = 02 003100E100010002 HW_ID
OU = 03 0000 OEM_ID_INDEPENDENT (OPTIONAL – Required for Double signing)
OU = 04 0000000000000002 DEBUG
OU = 05 0001 OEM_ID
OU = 06 00002000 SW_SIZE
OU = 07 0002 MODEL_ID
OU = 08 0001 SHA256
OU = 09 8996AAAA00000001 APP_ID
OU = 10 0000000000000000 CRASH_DUMP (OPTIONAL)
OU = 11 0000000000000000 ROT_EN (Not applicable)
OU = 12 0000000300000000 SOC_HW_VERSION (OPTIONAL)
OU = 13 0000000000000001 MASK_SOC_HW_VERSION (DEPRECATED)
OU = 14 0000000000000000 IN_USE_SOC_HW_VERSION (OPTIONAL)
OU = 15 0000000000000000 USE_SERIAL_NUMBER_IN_SIGNING (OPTIONAL)
```

Some of these OU fields are mandatory because their values are compared against the QFPROM fuses, while others are present for informational purposes only. The validation of the certificate signature automatically ensures that the information present in the certificate was not modified.

2.2.1.4.1 SW_ID field

This OU field contains the `SW_ID` value used in the HMAC to sign the image (see Section 5.2).

32 bits [63:32]	32 bits [31:0]
Software version	Software type

The software version in the certificate is checked against the minimum supported version specified in the antirollback fuses for the image.

If the software version in the certificate is older than the version in the antirollback fuses, authentication fails. This ensures that there is no rollback to an older defective image after a software error is fixed and the fuses blown to contain the updated fixed version number. For example:

1. An OEM signs TrustZone (TZ) with version 0.
2. Antirollback fuses for TZ are blown to 0.
3. A software error is discovered and fixed in TZ.
4. The OEM signs the fixed TZ image with version 1 and blows the antirollback TZ fuses to 1.

   Now if anyone tries to use the older version 0 signed TZ image, it fails authentication.

The software type specifies the signed image (XBL, TZ, and so on) and is used to ensure that the boot-up sequence cannot be changed.

The Sample Subject DN field (SW_ID) specifies the image as the APPSBL (0x9), which is signed with a version of 0x2 (see Section 5.2).

2.2.1.4.2 HW_ID field

This OU field contains the 64-bit MSM_HW_ID value used in the HMAC to sign the image (see Section 5.1). The MSM_HW_ID effectively binds the image to the specified hardware so that it can only pass authentication (and execute) on the specified hardware.

The MSM_HW_ID value is composed as shown in either case A or case B.

To indicate whether the Case A or Case B MSM_HW_ID value is used, the USE_SERIAL_NUMBER_IN_SIGNING field (see OU 14 in Table 2-4) is set to either 0 or 1. For example, if USE_SERIAL_NUMBER_IN_SIGNING is 1, the serial number is used, as shown in Case B.

Case A (USE_SERIAL_NUMBER_IN_SIGNING = 0)

32 bits [63:32]	16 bits [31:16]	16 bits [15:0]

SOC_HW_VERSION check-in SOC_VERS else
{
JTAG_ID (IN_USE_SOC_HW_VERSION=0)
Or
SOC_HW_VERSION (IN_USE_SOC_HW_VERSION=1)
}
	OEM_HW_ID
	CSMS account ID for the OEM. Starting MSM8998 chipset QTI requires licensees to use unique OEM_ID obtained from QT.
		OEM_MODEL_ID
		OEM-specified phone model identifier or 0 if the OEM wants to use a phone model identifier for signing/authentication.

Case B (USE_SERIAL_NUMBER_IN_SIGNING = 1)

32 bits [63:32]	32 bits [31:0]
JTAG ID	SERIAL NUM

Upper 32 bits of HW_ID

When JTAG_ID is used, the upper 32 bits denote the JTAG ID (MSM™ identifier) QFPROM fuse value with the upper 4 bits (bits 31 to 28) containing the Die Revision/Version masked out. The JTAG ID describes the hardware (that is, the SDM636/SDM660/SDM630/SDM670 chipset) along with the variant information, and is blown (provisioned) by QTI.

The SOC_HW_VERSION field is used in SDM636/SDM660/SDM630/SDM670 chipsets and later to tie the signed binary image to a family of chipsets. The IN_USE_SOC_HW_VERSION field is either 0 or 1 to indicate whether SOC_HW_VERSION is used (that is, if it is 0, JTAG_ID is used).

The SOC_VERS OU field is list of space-delimited SOC_HW_REVISION for supported chipsets by a signed image. The SOC_VERS OU field can be used in SDM636/SDM660/SDM630 chipset to tie the signed image to chipset from different family, that is, chipsets for which SOC_HW_REVISION are different. If SOC_HW_VERSION from hardware is found in SOC_VERS OU field of the signed image, then JTAG_ID or SOC_HW_VERSION from HW_ID OU field is ignored or not used for authentication of image that is, SOC_HW_VERSION check-in SOC_VERS supersedes JTAG_ID or SOC_HW_VERSION check from HW_ID OU field.

If SOC_VERS OU field is not found in signed image or SOC_HW_VERSION from hardware is not found in SOC_VERS of signed image, then only JTAG_ID or SOC_HW_VERSION from HW_ID OU field is check for authentication of the image. SOC_VERS OU field essentially enable signed binary image to be secure boot compatible for SOC_HW_VERSION captured in this OU field.

NOTE: Note SEC_VERS OU field is only supported if image is signed with PSS padding that is, <rsa_padding>pss</rsa_padding> is used for signing of image.

Lower 32 bits of HW_ID

In Case A, the lower 32 bits comprise the OEM identifier (the value that identifies the OEM and their phone model).

In Case B, the lower 32 bits comprise the chip-specific unique serial number.

NOTE: There is no serial number field in secimage.xml; the user must update the OEM_ID and MODEL_ID field accordingly.

2.2.1.4.3 OEM_ID_INDEPENDENT

If enabled, Sectools sets OEM_ID to 0001 and MODEL_ID to 0000 in the attestation certificate.

- Setting this field to 1 means enabled. Setting 1 is used for QTI signing of images, which needs double signing.
- Setting this field to 0 means disabled. If the field is not present, it is disabled.

The setting is reserved for QTI use. The recommended customer setting is 0 (disabled).

2.2.1.4.4 DEBUG field

This field contains information on whether the OEM debug disable fuse settings must be preserved or overridden (that is, debugging is to be re-enabled) for a chip with the specified serial number.

This field is only acted upon as follows:

- By the application PBL (APPS PBL) based on the debug OU field of the attestation certificate of the XBL image.
- By the modem PBL based on the debug OU field of the modem boot authentication (MBA) attestation certificate.

The field comprises:

32 bits [63:32]	32 bits [31:0]
SERIAL NUM	DEBUG SETTINGS

The lower 32 bits denote the action that must be taken by the PBL in writing to the one-time writable OVERRIDE_2, OVERRIDE_3 (APPS PBL), and OVERRIDE_4 (Modem PBL) registers. These registers allow override of the OEM debug disable fuses:

- Setting them to 1 maintains the OEM debug disable fuse values.
- Setting them to 0 overrides the OEM debug disable fuse values with the QTI debug disable fuse values.

Since QTI does *not* blow the debug disable fuses, writing 1 to the one-time writable registers essentially means re-enabling debugging.

OEMs can specify what the PBL is to do using the following debug value settings:

- 0x2 indicates that 0 is to be written to the one-time debug override registers. This preserves the OEM debug disable fuse settings. No image post-PBL can change these settings using the one-time debug override registers.
- 0x3 indicates that 1 is to be written to the one-time debug override registers *only* if the serial number of the chip matches the serial number in the upper 32 bits of this field. This causes debug to be re-enabled.

For example, the value of 0x1234567800000003 denotes a debug certificate for a chip with serial number 0x12345678. If this certificate is used on a chip with a different serial number, authentication fails.

If this OU field is not found in the certificate, the default value of 0x0000000000000000 is used, that is, no operation is performed in the PBL with the one-time override registers.

2.2.1.4.5 OEM_ID field

QTI assigns each OEM an identifiable ID for secure boot and CSMS license management. The `OEM_ID` must be burned with the assigned ID so the OEM can use this capability to support either Qualcomm secure boot solutions (CSMS) or DSP dynamic feature authentication of QTI or third-party feature enhancements. The DSP enabling feature authentication ensures that the OEM approves the DSP dynamic updates. So even if an OEM does not use CSMS for signing, it is recommended they blow the `OEM_ID` to their assigned value.

2.2.1.4.6 SW_SIZE field

This field is used for information purposes only; it denotes the number of bytes that were signed. During the signing process, signing tools update this field to the correct size.

```
SW_SIZE (in bytes)= (hash size in bytes * no.of segments) + 40 bytes header
of the Hash table.
```

2.2.1.4.7 MODEL_ID field

This field is used for information purposes only; it denotes the `OEM_MODEL_ID` value in the `MSM_ID` OU field (see Section 2.2.1.4.2). Blow with nonzero values. Different models based on the same MSM device must have a unique `Model_ID` to prevent misuse of images across various models.

2.2.1.4.8 SHA1/SHA256 field

This field denotes the hash algorithm (SHA1 or SHA256); QTI strongly recommends using SHA256 to generate the code signature**.**

2.2.1.4.9 APP_ID field

This field is required for a TZ application, for example, when `SW_ID` is `TZ_EXEC_HASH_TABLE = 0xC`. The value must be in hexadecimal.

This field is used when antirollback fuses for TZ are blown to 1. The TZ secure application antirollback tracks the software version of the TZ application by APP_ID. All zeroes is an invalid input when the TrustZone antirollback fuse is blown, and loading the TrustZone application fails.

This field is also used by the TZ SFS to generate a unique secret key for the SFS encryption of that TZ application. NULL or zero value `APP_ID` TZ applications share the SFS secret key for the SFS encryption.

It is the responsibility of the signer (OEM) to keep the `APP_ID` list and ensure that different TZ applications do not share `APP_ID`.

2.2.1.4.10 CRASH_DUMP field

This field supports the retail unlock feature. It enables logging for a subsystem, such as UEFI, nonsecure registers, HLOS, and encrypted modem EFS. It is an optional signing attribute that TZ reads. For MSM8916 chipsets and later, the XBL image needs to sign with this OU field, and PBL reads it.

The field comprises:

32 bits [63:32]	32 bits [31:0]
SERIAL NUM	ENABLE

The lower 32 bits controls enabling of the feature:

- Setting them to 1 means the feature is enabled.
- Setting them to 0 means the feature is disabled.

The higher 32 bits specify the serial number of the target device.

The behavior of setting `CRASH_DUMP` with DEBUG settings is explained in the following table.

DEBUG setting	CRASH_DUMP setting	Behavior
X	0	`CRASH_DUMP` is not enabled. DEBUG field controls the debug behavior (see Section2.2.1.4.4).
0	1	Logs for security subsystems are disabled. Logs for subsystems such as UEFI, nonsecure registers, HLOS, and encrypted modem EFS are enabled.
2	1	Logs for security subsystems are disabled. Logs for subsystems such as UEFI, nonsecure registers, HLOS, and encrypted modem EFS are enabled.
3	1	This enables dummy keys. As RPMB has provisioned with production key, using this setting blocks all reads/writes from RPMB, as the dummy key is now used for all crypto operations. On Windows, as secure variables and other boot critical information is stored in RPMB, the device fails to boot. On Android, all TZ applications, which rely on RPMB for their version control might fail to load. The device should boot up to HLOS without any issues. Logs for all security subsystems are enabled.

For example, the value of 0x1234567800000001 denotes that the retail unlock feature is enabled for a chip with serial number, 0x12345678. If this certificate is used on a chip with a different serial number, authentication fail and the device fails to boot.

2.2.1.4.11 ROT_EN_FIELD

This field supports the ROT (root of trust) feature but ROT is not applicable for SDM636/SDM660/SDM630/SDM670 chipsets.

2.3 Image formats

Image signing requires calculating the hash over the code segment and header segment of the raw binaries.

The number of code segments varies from one software image to the other. In the SDM636/ SDM660/SDM630/SDM670 chipsets all the images, including MBA and XBL images, are in the standard ELF format.

2.3.1 ELF format

ELF format binaries contain multiple segments. Various types of information (that is. Source address, destination address, type, and so forth) about each segment are kept as part of the program headers.

In QTI software components, all binaries are in ELF format and contain multiple segments. For these kinds of images, authentication is performed for the hash segment, and later the individual value of hash for the individual segment is compared against the authenticated hash value.

The mba.mbn image is a special case; for more details, see Section 6.2.

2.3.1.1 ELF header

Each ELF file begins with a header defined as follows:

```
typedef struct {
  unsigned char e_ident[16]; /* Magic number and other info */
  uint16 e_type;             /* Object file type */
  uint16 e_machine;          /* Architecture */
  uint32 e_version;          /* Object file version */
  uint32 e_entry;            /* Entry point virtual address */
  uint32 e_phoff;            /* Program header table file offset */
  uint32 e_shoff;            /* Section header table file offset */
  uint32 e_flags;            /* Processor-specific flags */
  uint16 e_ehsize;           /* ELF header size in bytes */
  uint16 e_phentsize;        /* Program header table entry size */
  uint16 e_phnum;            /* Program header table entry count */
  uint16 e_shentsize;        /* Section header table entry size */
  uint16 e_shnum;            /* Section header table entry count */
  uint16 e_shstrndx; /* Section header string table index */
} Elf32_Ehdr;
```

This header identifies the file as *.elf and defines the overall structure of the file. An ELF file is subdivided into segments. Each segment has a size, a location in the file, and an address at which the segment is loaded in memory.

The `Elf32_Ehdr.e_phoff` element identifies the location of a program table that contains a definition of each segment.

The `Elf32_Ehdr.e_phnum` element identifies the number of segments.

2.3.1.2 ELF segment

A structure defines each ELF segment.

```
typedef struct {
  uint32 p_type;            /* Segment type */
  uint32 p_offset;          /* Segment file offset */
  uint32 p_vaddr;           /* Segment virtual address */
  uint32 p_paddr;           /* Segment physical address */
  uint32 p_filesz;          /* Segment size in file */
  uint32 p_memsz;           /* Segment size in memory */
  uint32 p_flags;           /* Segment flags */
  uint32 p_align;           /* Segment alignment */
} Elf32_Phdr;
```

Following are some values for the p_type element:

- `PT_NULL` – The array element is unused; other member values are undefined. This type lets the program header table have ignored entries.
- `PT_LOAD` – The array element specifies a loadable segment, described by `p_filesz` and `p_memsz`. The bytes from the file are mapped to the beginning of the memory segment. If the memory size of the segment (p_memsz) is larger than the file size (p_filesz), the extra bytes are defined to hold the value 0 and to follow the initialized area of the segment. The file size might not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the p_vaddr member.

The `Elf32_Phdr.p_offset` field defines the offset in the file at which the segment is located.

The `Elf32_Phdr.p_vaddr.Elf32_Phdr.p_vaddr` field defines the address where the segment is to be loaded.

The `Elf32_Phdr.p_filesz` field defines the segment size in the file, which is the number of bytes that is loaded into memory.

Bits 20 to 27 of the `p_flags` field are the `PF_MASKOS` bits reserved for OS-specific semantics that are used for QTI implementation.

Table 2-5 provides information on what the bits in the p_flags field correspond to (as described in the QTI miprogressive.h header file).

Table 2-5 ELF p_flags with QTI extension

Bits	Description
0:19	Standard segment flag. Execute (0x1), Write (0x2), Read (0x4), Read Execute (0x5), and so forth
20	Segment mode.
21:23	Access type. ▪ Nonpaged segment (0x0) that is loaded to RAM before booting the respective processor. These segments are hashed and are validated. ▪ Paged segment (0x1). Demand paged segments; obsoleted by the MSM8974 chipset.
24:26	Segment type. L4 segment (0x0), AMSS segment (0x1), Hash segment (0x2), Boot segment (0x3), and so forth
27	Pool index.

For further understanding, part of the program header from an existing ELF file is shown here:

```
Sample Program Header table (readelf --l xxx.elf)
  Type  Offset   VirtAddr   PhysAddr   FileSiz  MemSiz   Flg  Align
  NULL  0x000000 0x00000000 0x00000000 0x00414  0x00000       0
  LOAD  0x001000 0x8d944000 0x8d944000 0x00294  0x01000       0x1000
  LOAD  0x002000 0x8d400000 0x8d400000 0x008f4  0x008f4  R E  0x1000
```

The readelf tool does not describe bits 20 onwards of the p_flags column because these bits are OS-specific. The p_flags field for the three segments contains the following values (from looking at the actual p_flags bytes in the ELF header).

```
0x07000000: Access Type 0, Segment Type 7. Non-Paged Segment.
0x02200000: Access Type 1, Segment Type 2. Paged segment. Type Hash Table
            Segmentsh Table
0x00000005: Access Type 0, Segment Type 0, Non-Paged segment. Type L4.
            Read & Execute Segment.
```

2.3.1.3 Hash table segment

The hash segment table contains a hash header, a hash entry for the ELF header + program header, and hashes of all the individual segments.

The ELF image consists of n segments of mode type, defined by the p_flags entry in the respective program header entries. The ELF image supports one of the following modes:

- Segment mode type flag in p_flags = MI_PBT_NON_PAGED_SEGMENT – These segmentsare loaded to RAM before booting the respective processor.
- Segment mode type flag in p_flags = MI_PBT_PAGED_SEGMENT – Given the overhead of demand with paging 4000 pages on some architectures like the MDM9200 device, this mode is not used in MSM8960, MSM8974, and later devices.

To validate the code that executes on the respective processor, an SHA1 hash of the code must be stored immutably in the image. These hashes are stored in a hash table segment, which can be looked up by checking `p_flag = MI_PBT_HASH_SEGMENT`.

The segments that are required to be hashed are all the segments with an access type flag in `p_flags = MI_PBT_NON_PAGED_SEGMENTS`. They should be validated before bootup. Also, before booting up a processor, all `MI_PBT_NON_PAGED_SEGMENT` segments are loaded in their final destination RAM addresses.

The segments with p_flags `MI_PBT_NOTUSED_SEGMENT`, `MI_PBT_SHARED_SEGMENT`, and `MI_PBT_HASH_SEGMENT` are not hashed.

2.3.1.4 Signing with ELF format

An ELF image `file_to_sign.elf` must have m sections, where $m \geq 1$. The signing tool must compute (with `PT_LOAD flag`) an SHA256 digest (D_n) for each loadable section, so the result is n ($n \in [1, m]$)

SHA256 digests, where each SHA256 digest is 32 bytes. A new section is created with the contents:

$D = D_1 + D_2 + \ldots + D_n$

hash_segment = header(40b) + D + signature(256b) + cert_chain(6144b)

This header is the 40-byte header listed in the mi_boot_image_header_type structure. The new section also has `PT_LOAD`, as well as the QTI extension set that indicates it is a hash segment.

NOTE: Even if secure boot is not enabled, the ELF binary for secure boot also needs the hash segment as described.

ELFs that are executed via TZ peripheral image loading (PIL) contain the hash table segment whether the secure boot feature is enabled or not. This segment allows nonsecure boot builds to have a basic validation check of each segment. That is, TZ PIL hashes a segment and compares it with the hash contained in the hash table segment to see if it matches.

Figure 2-3 shows an unsigned ELF image with a hash segment.

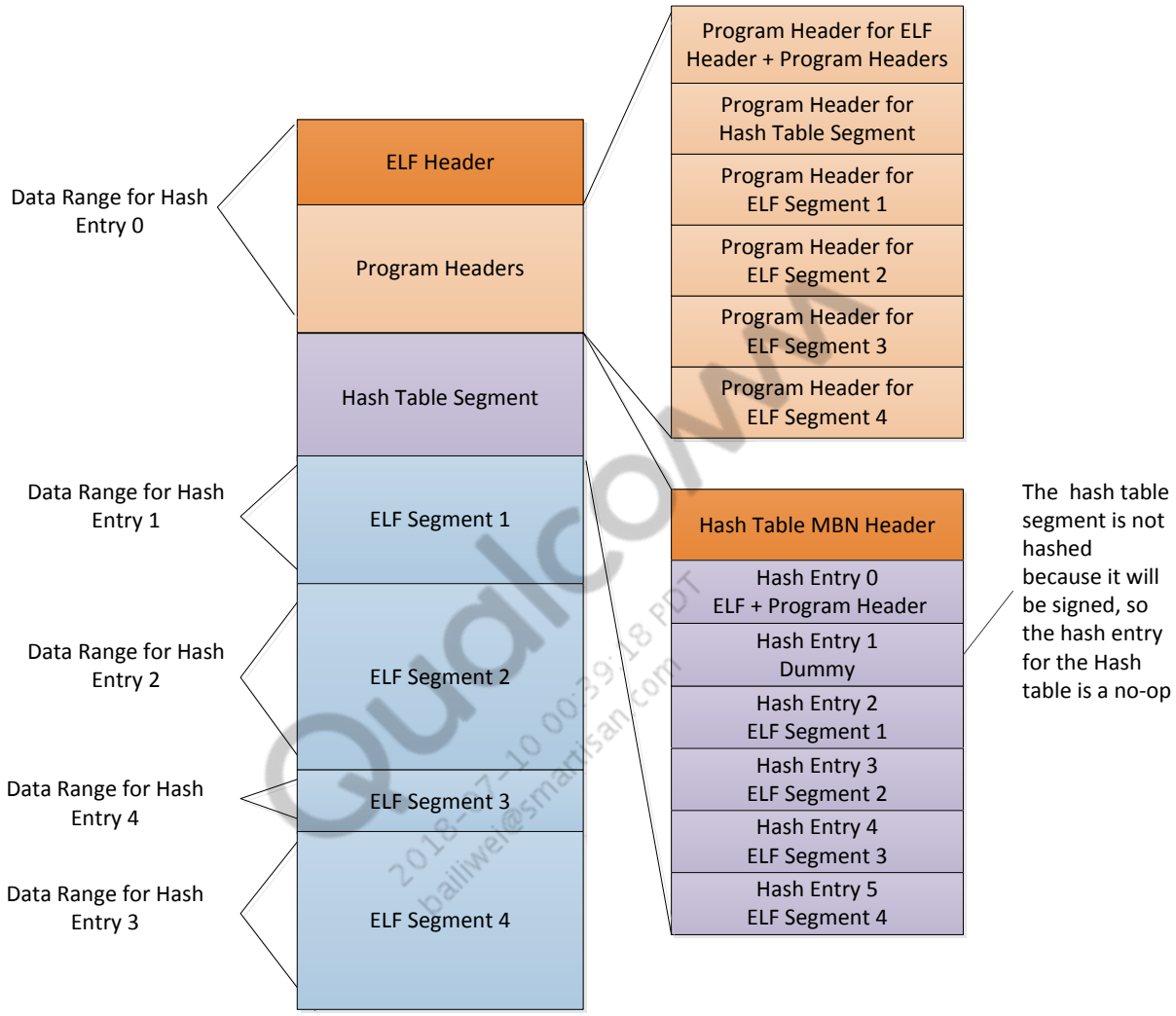


Figure 2-3 Overview of unsigned ELF image

After the hash segment, signing the ELF is straightforward: sign the hash contents D, and update signature, `cert_chain`, and the hash segment header after signing. Figure 2-4 shows a signed ELF image.

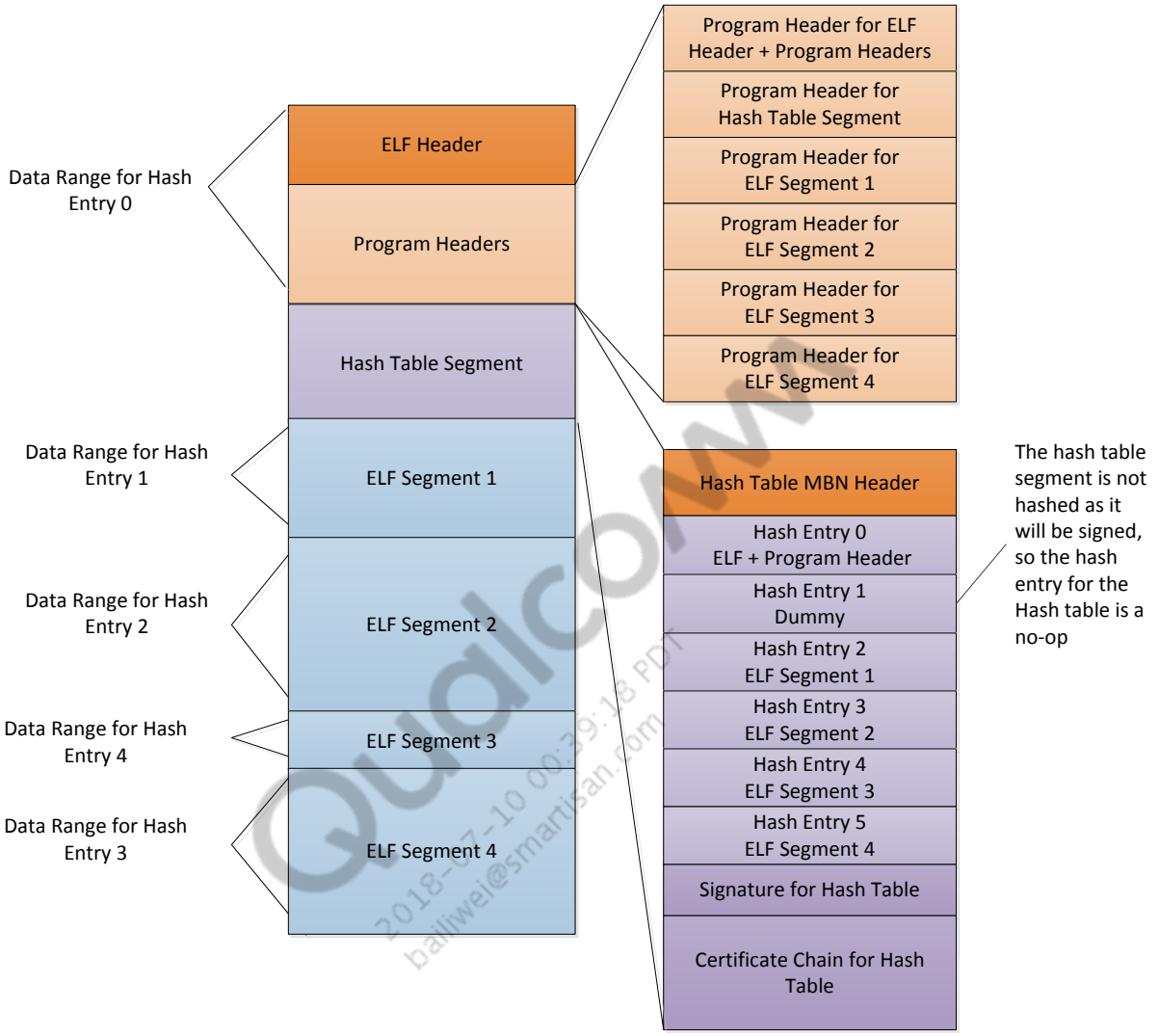


Figure 2-4 Signed multiple segments binary format

When verifying the signed ELF, secure boot verifies:

- Whether the certificate chain and signature (in the hash segment) are valid.
- Whether D1. . . Dn matches the SHA1 digest of each loadable ELF section (excluding the hash segment itself).

2.3.2 Signing hash table segment

In asymmetric key cryptography (RSA), signature generation is typically done in three steps:

1. Apply one-way hash function to the message to produce a digest (mHash)
2. Apply padding to digest that is, transform the digest value (mHash) into an encoded message EM.
3. Encryption of encoded message (EM) with RSA private key, which is $(EM)^d \bmod(n)$ where {d,n} is RSA private key, to produce a signature S.

The output of the preceding steps is a cryptographic signature, which can be later used to prove integrity and authenticity of messages. This step of signature generation is performed in a secured location, ensuring that the RSA private key {d,n} is not leaked when executing the preceding process.

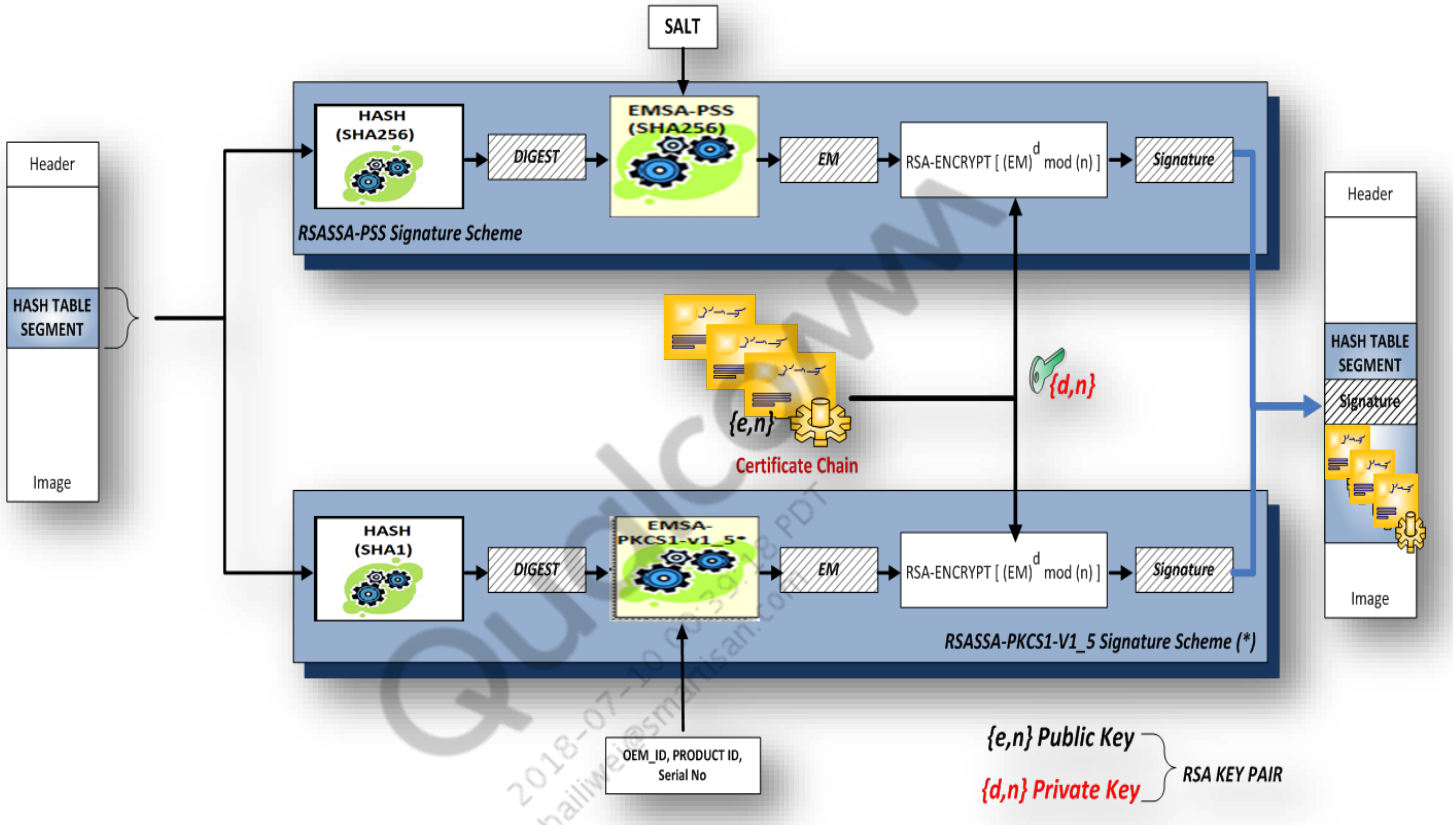


Figure 2-5 QTI hash table segment signature generation process

Figure 2-5 shows the hash table segment signature generation process and SDM636/SDM660/SDM630/SDM670 supports only one signature generation schemes:

Signature based on RSASSA-PSS scheme − SDM636/SDM660/SDM630/SDM670 platform supports more modern and robust signature scheme option based on EMSA-PSS encoding/padding as specified in PKCS#1 v2.1 and implemented as per standard with no modifications.
RSASSA-PSS signature scheme offers more security benefits over other signature schemes and is widely accepted as the industry standard. Refer to the next section for details on this signature scheme.

NOTE: The signature based on RSASSA-PSS (SHA256) is the recommended algorithm because of security and performance benefits with RSASSA-PSS (SHA256) as compared to RSASSA-PKCS-V1_5 (SHA1). QTI Sectool continues to support both signature generations to support legacy targets, but new targets discontinue support for `RSASSA-PKCS-V1_5` signature scheme, so licensees should use RSASSA-PSS (SHA256) for the SDM636/SDM660/SDM630/SDM670 platform.

2.3.2.1 Signature based on RSASSA-PSS

Signature based on RSASSA-PSS is a new addition to the SDM636/SDM660/SDM630/SDM670 platform and is a modern signature generation scheme, which offers greater security assurances over other signature generation schemes. The RSASSA-PSS signature scheme is based on EMSA-PSS encoding/padding as specified in PKCS#1 v2.1, which incorporates a pseudorandom number for padding generation, called salt. Because the salt changes for every signature generation, signing the same message twice using the same private key results in different signatures, offering an added security benefit.

Figure 2-6 shows EMSA-PSS encoding based on following parameters

1. Hash Function = SHA256 and Hash Length (hLen) = 256 bits.
2. Mask Generation Function (MGF) = MGF1.
3. Salt Length (sLen) = 256 bits.
4. RSA Key Length (modulus n) = 2048 bits.

EMSA-PSS encoding depends on the preceding parameters, refer to PKCS#1 v2.1 or later for more information on EMSA-PSS encoding dependencies on the preceding parameters.

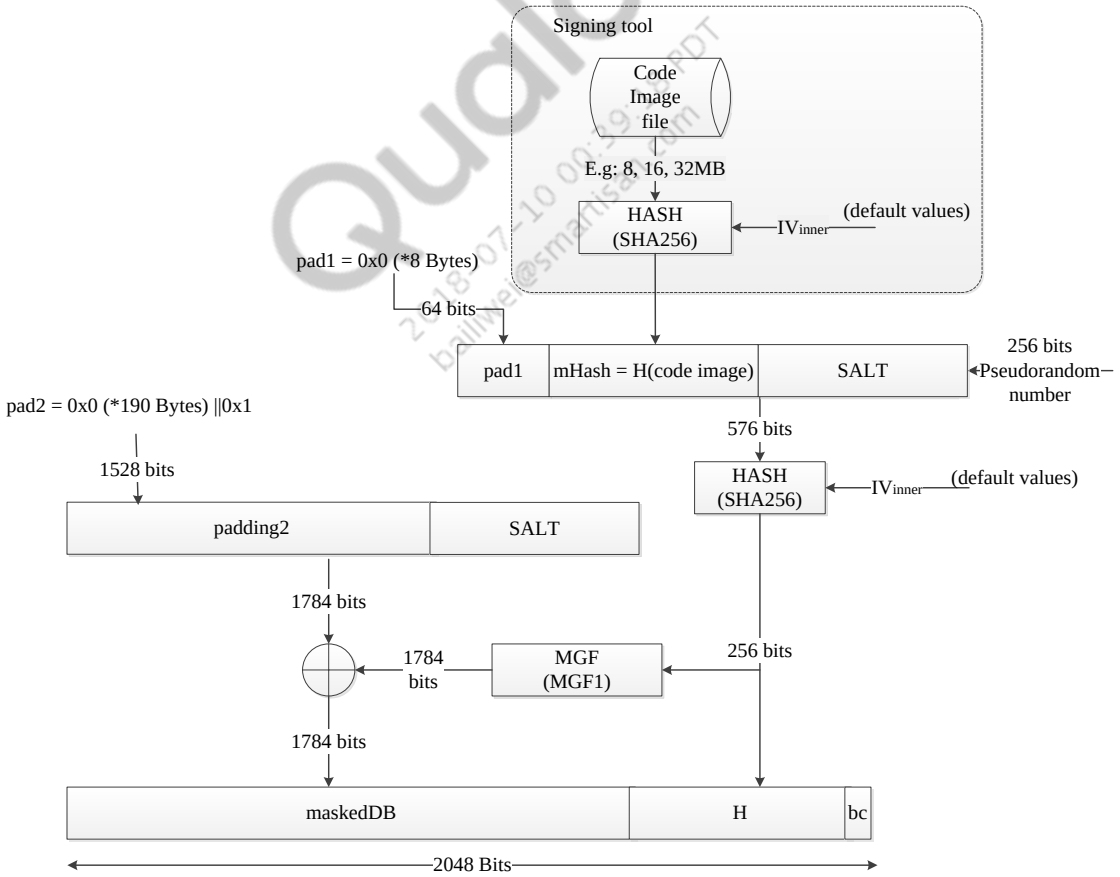


Figure 2-6 EMSA-PSS encoding

2.3.3 Hash table segment signature verification

On the device side, hash table segment signature validation is done by decrypting the signature with a public key {e,n} stored in the boot ROM or QFPROM (public key from attestation certificate chaining to root certificate in boot ROM or certificate hash blown in QFPROM eFuses), to be compared with local calculated hash digest on hash table segment.

The following figures show the hash table segment signature verification process and SDM636/SDM660/SDM630/SDM670 supports only one signature verification scheme, Verification based on RSASSA-PSS signature scheme.

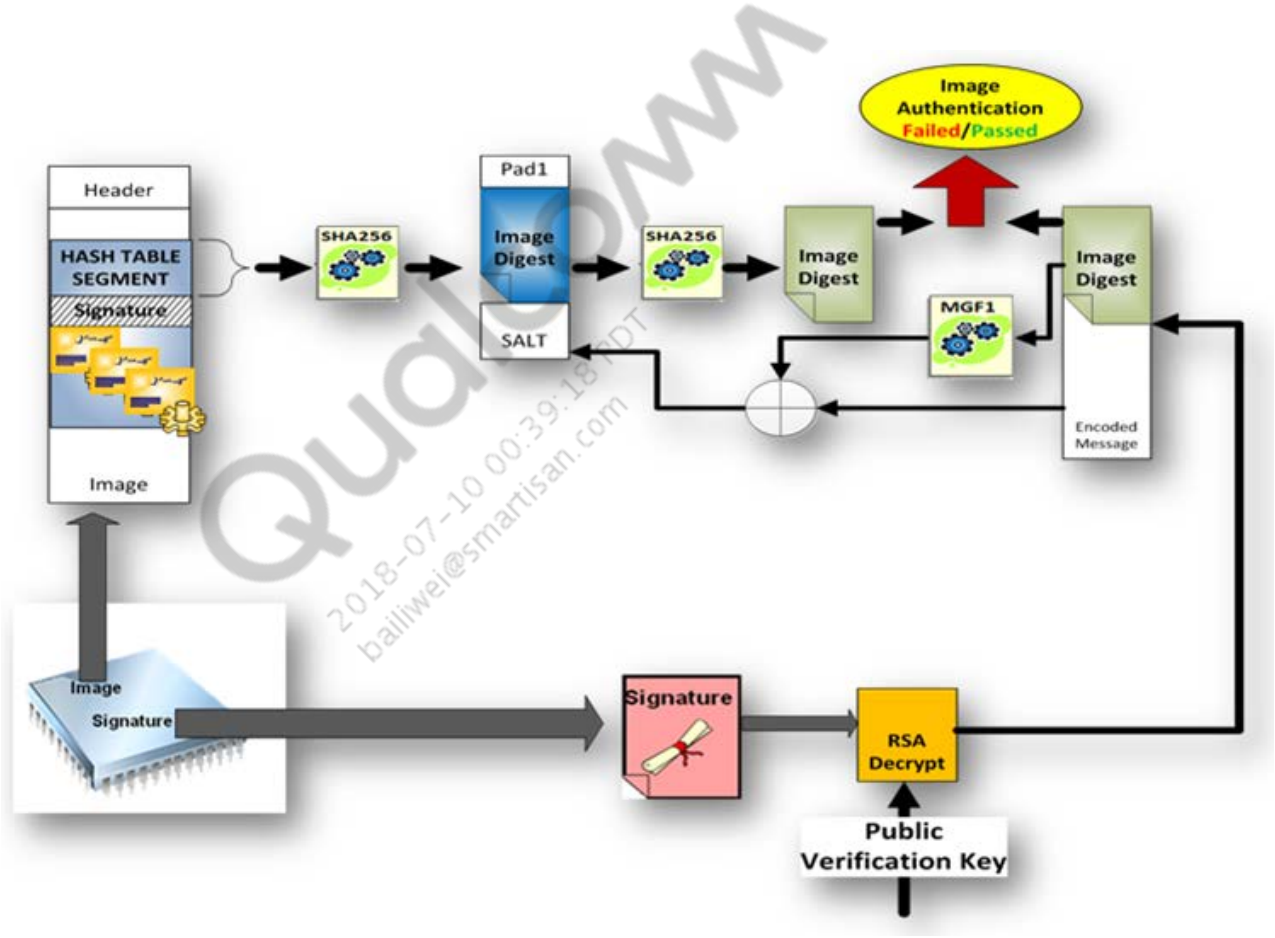


Figure 2-7 Hash table signature verification

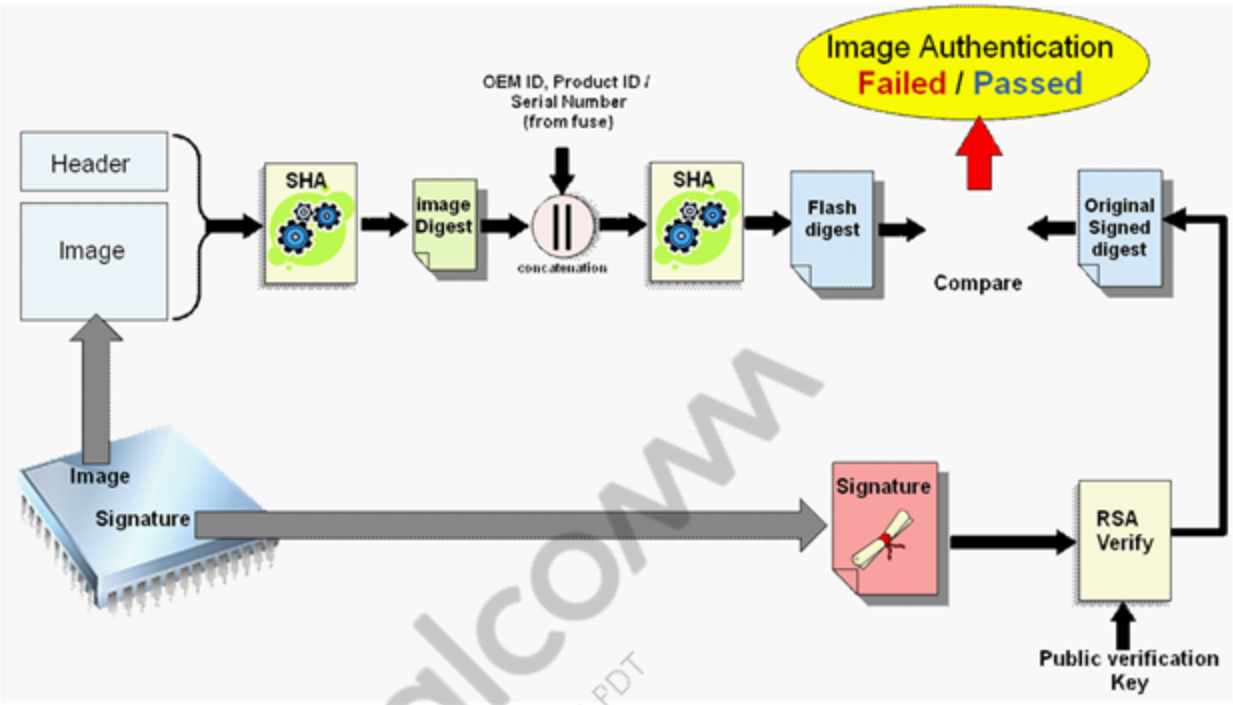


Figure 2-8 Hash table signature verification

2.4 Image authentication components

In the SDM636/SDM660/SDM630/SDM670 chipset, secure information related to secure boot is contained in the QFPROM, boot ROM, and SDRAM. Figure 2-8 shows the image authentication components.

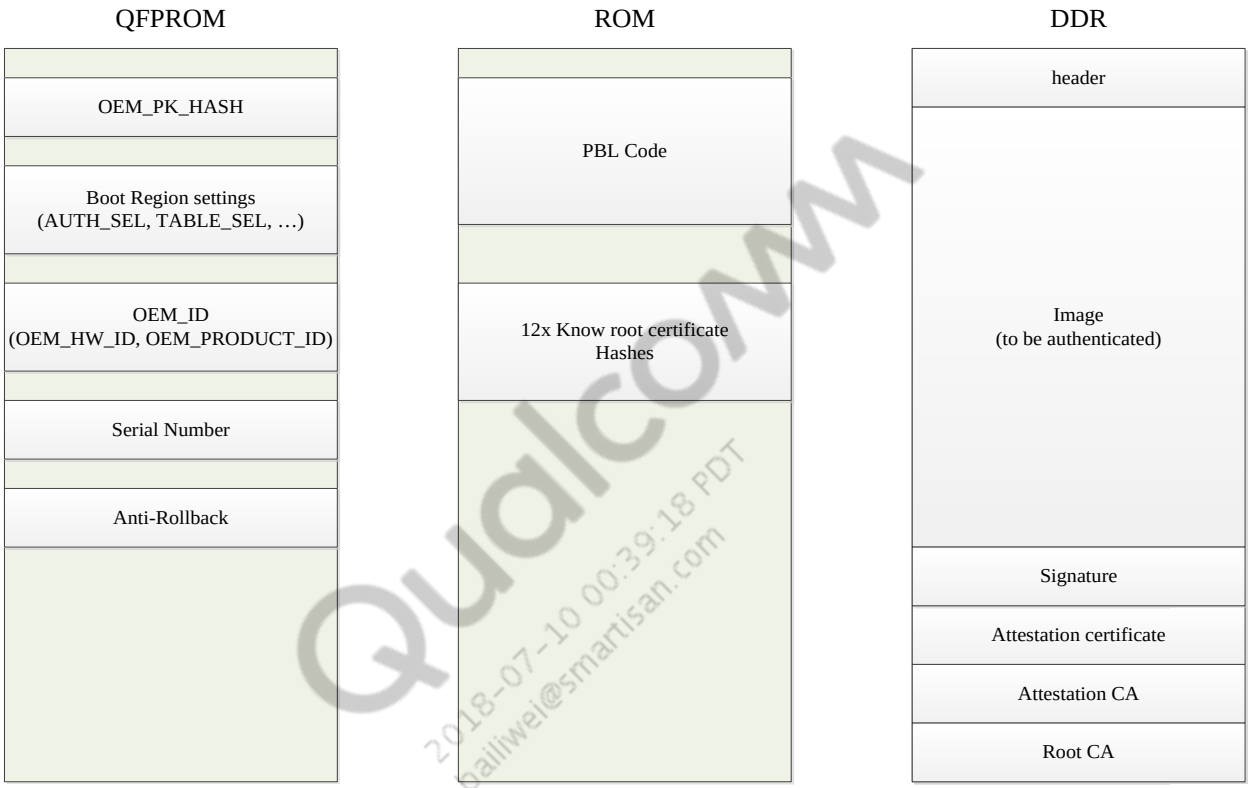


Figure 2-9 Image authentication components

2.5 Secure boot flowcharts

2.5.1 SDM636/SDM660/SDM630 Secure boot flowchart

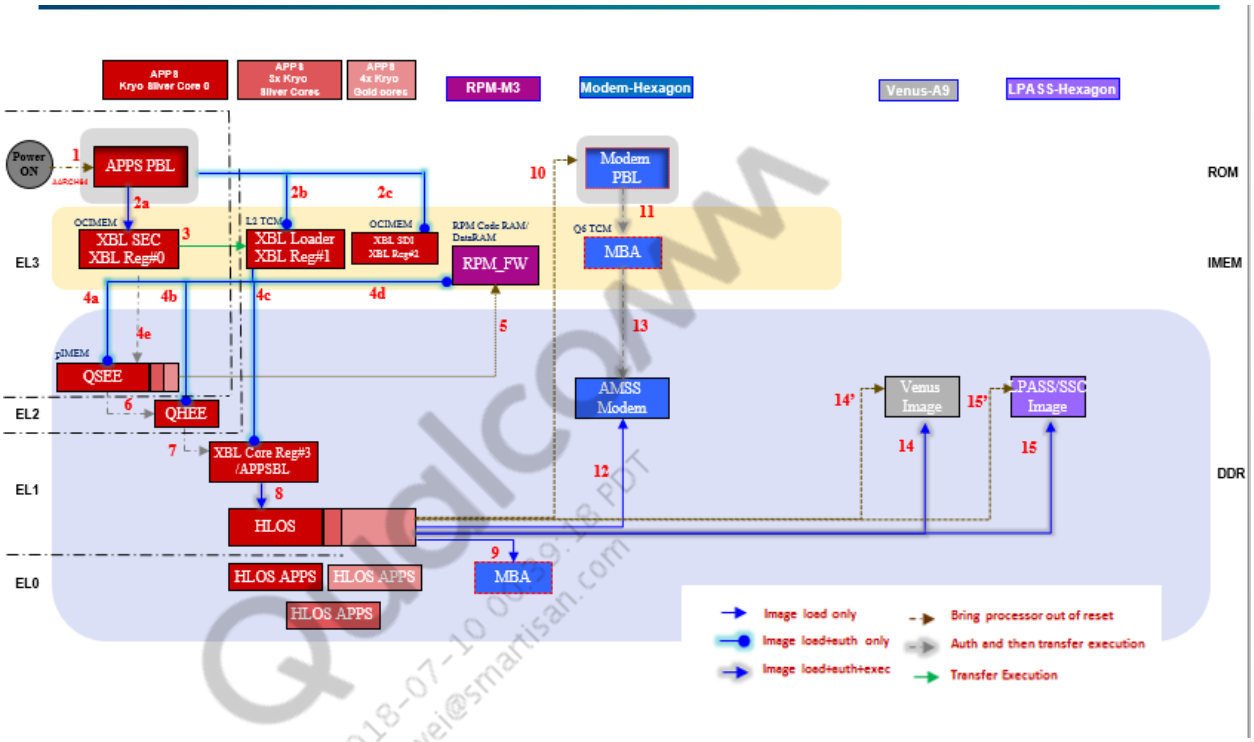


Figure 2-10 SDM636/SDM660/SDM630 secure boot flowchart

As shown in Figure 2-9. the secure boot flow is as follows:

1. The Qualcomm® Kryo™ CPU or Cortex A53 application CPU is moved out of the Reset mode.
2. In Kryo or Cortex A53, the APPS PBL does the following:
    a. Loads and authenticates XBL SEC (region #0) from the storage device to OCIMEM
    b. Loads and authenticates XBL (region #1) from the storage device to L2TCM
    c. Loads and authenticates XBL debug (region #2) to OCIMEM, then jumps to XBL region #1.
3. XBL SEC runs the security configuration in EL3 mode, and then executes the XBL in EL1 mode.
4. XBL loads other firmware in the following order:
    a. Loads and authenticates the QSEE image from the boot device to pIMEM
    b. Loads and authenticates the QHEE image from the boot device to DDR
    c. Loads and authenticates the XBL Core (region #3) and ABL image from the boot device to DDR
    d. Loads and authenticates the RPM firmware image from the boot device to RPM code RAM

e. XBL SEC transfers execution to QSEE.

5. QSEE brings RPM out of the Reset mode to execute the RPM firmware.
6. QSEE sets up a secure environment, and then proceeds to execute the QHEE image.
7. QHEE proceeds to execute the XBL core (or XBL region #3) and XBL core mount, and then runs the UEFI apps (abl.elf).
8. Linux loader application (part of ABL FV) loads and authenticates the HLOS kernel with verify boot.
9. HLOS kernel loads the MBA to DDR through the peripheral image loader (PIL).
10. HLOS kernel brings the Hexagon DSP modem out of the Reset mode.
11. Modem PBL copies the MBA from DDR to the modem TCM, authenticates MBA, and then jumps to the MBA image.
12. HLOS loads the AMSS modem image to DDR through PIL.
13. MBA authenticates the modem image, and then jumps to the modem.
14. HLOS loads the Venus, LPASS, and SSC images to DDR through PIL.
15. HLOS brings Venus, LPASS, and SSC out of the Reset mode, and then starts the execution through a secure SMC call.

2.5.2 SDM670 Secure boot flow chart

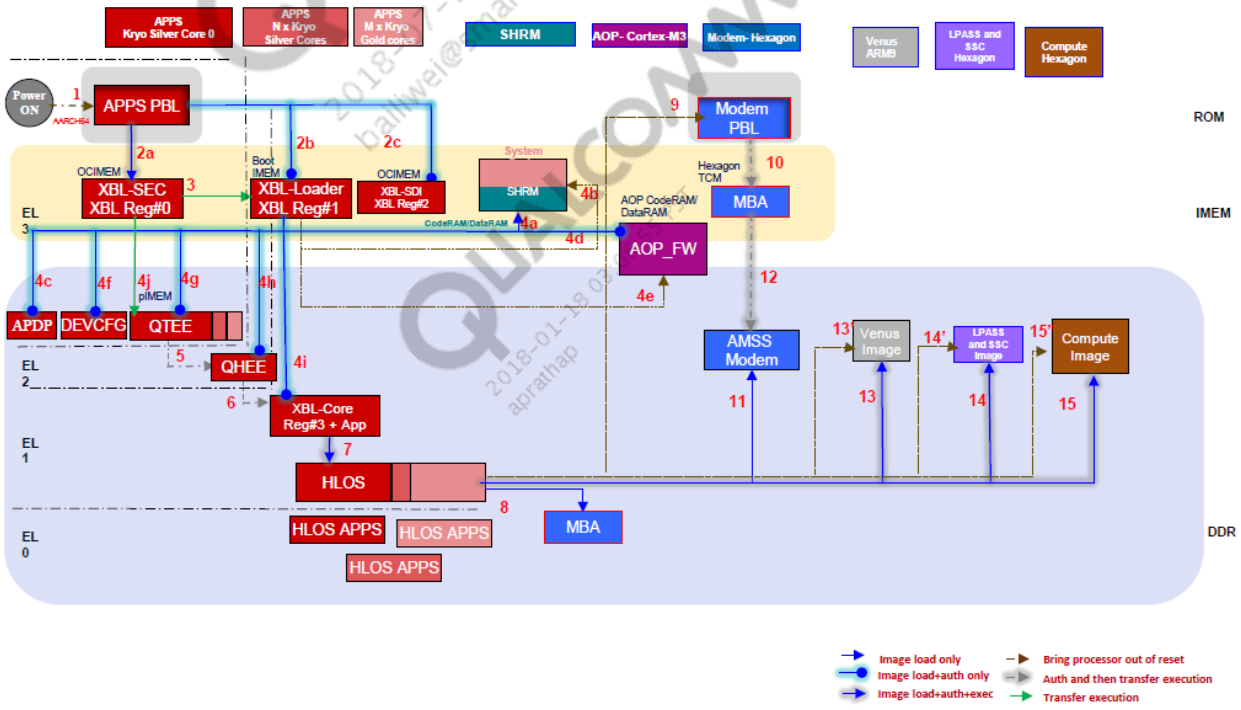


Figure 2-10 SDM670 secure boot flowchart

As shown in Figure 2-10 the secure boot flow is as:

1. After the MSM™ reset, the Kryo Silver core 0 comes out of reset and executes PBL
2. The APPS PBL does the following:
   - Initializes hardware (clocks and so on)
   - Initializes CPU caches and MMU
   - Detects the boot device as per the boot option configuration
     a. Default boot option is UFS □SD □USB
     b. Default boot option can be overridden by EDL cookie or Force USB GPIO

   2a. Loads and authenticates XBL-SEC (Reg#0) from the bootdeviceto OCIMEM

   2b. Loads and authenticates XBL-Loader (Reg#1) from the bootdeviceto Boot IMEM

   2c. Loads and authenticates XBL-SDI (Reg#2) from the boot device to OCIMEM

   Jumps to XBL-SEC
3. XBL-SEC runs the security configuration in the EL3mode, and then executes the XBL-Loader in the EL1mode
4. XBL-Loader performs the following actions on the hardware and firmware images: Initializes CPU caches and MMU
5. Initializes the boot device
6. Initializes XBL config
7. Initializes PMIC driver
8. Initializes DDR and performs the DDR training, if applicable
9. Executes an SCM call to XBL-SEC to initialize pIMEM
10. Initializes clocks and configures the clock frequencies as per the clock plan
11. Initializes the shared memory (SMEM); updates the platform ID and RAM partition table
12. 4a. Loads and authenticates the SHRM image from the boot device
13. 4b. Brings SHRM out of reset
14. 4c. Loads and authenticates the APPS debug policy (APDP) image from the boot device. If the DLOAD cookie is set, loads and authenticates XBLRAMDUMP, and then jumps to XBL RAM dump to collect the crash dump
15. 4d. Loads and authenticates the AOP image from the boot device
16. 4e. Brings the AOP processor out of reset
17. 4f. Loads and authenticates the TZ device configuration (DEVCFG) image from the boot device
18. 4g. Loads and authenticates the QTEE image from the boot device; loads the SEC.dat (fuse blowing data) image from the boot storage if the image exists
19. 4h. Loads and authenticates the QHEE image from the boot device
20. 4i. Loads and authenticates the XBL-Core image from the boot device
21. 4j. Executes an SCM call to XBL-SEC to jump to the QTEE cold boot

2.5.3 Boot ROMs

The SDM636/SDM660/SDM630/SDM670 chipset has two boot ROMs – APPS PBL and Modem PBL. Boot ROMs are read-only, so it is not required to authenticate them.

2.5.4 PBL error handling

This information will be provided in next revision of this document.

2.6 Double signing for SDM636/SDM660/SDM630/SDM670

To enhance the security of the platform, QTI has introduced a new concept of image signing, called double signing. Double signing images are dual signed by QTI and Licensees and double signed binary will only be allowed to execute if both signings are authenticated on the device.

Currently, double signing is enforced only for boot-critical secure world software components, which execute at the highest privilege level and, sets up a secure execution environment on the device. Double signing ensures that boot code, which setups secure execution environment, is QTI reviewed, released, and approved code and not modified by licensee in any manner.

QTI releases production signed images for images identified as candidates for double signing and licensees must add their signature on top of QTI signed images, leading to two signatures on the image.

Dual authentication is enforced by device for such images and such image will be authenticated twice, once for QTI root of trust and once for the root-of-trust configured to device of the licencee.

Double signing is fully supported in Sectools version 4.15 or greater.

The images identified as a candidate for double signing in SDM636/SDM660/SDM630/SDM670 boot sequence are indicated in red in Figure 2-10 and are as following.

1. XBL_SEC.ELF/Flash_Programmer.ELF
2. QSEE.ELF

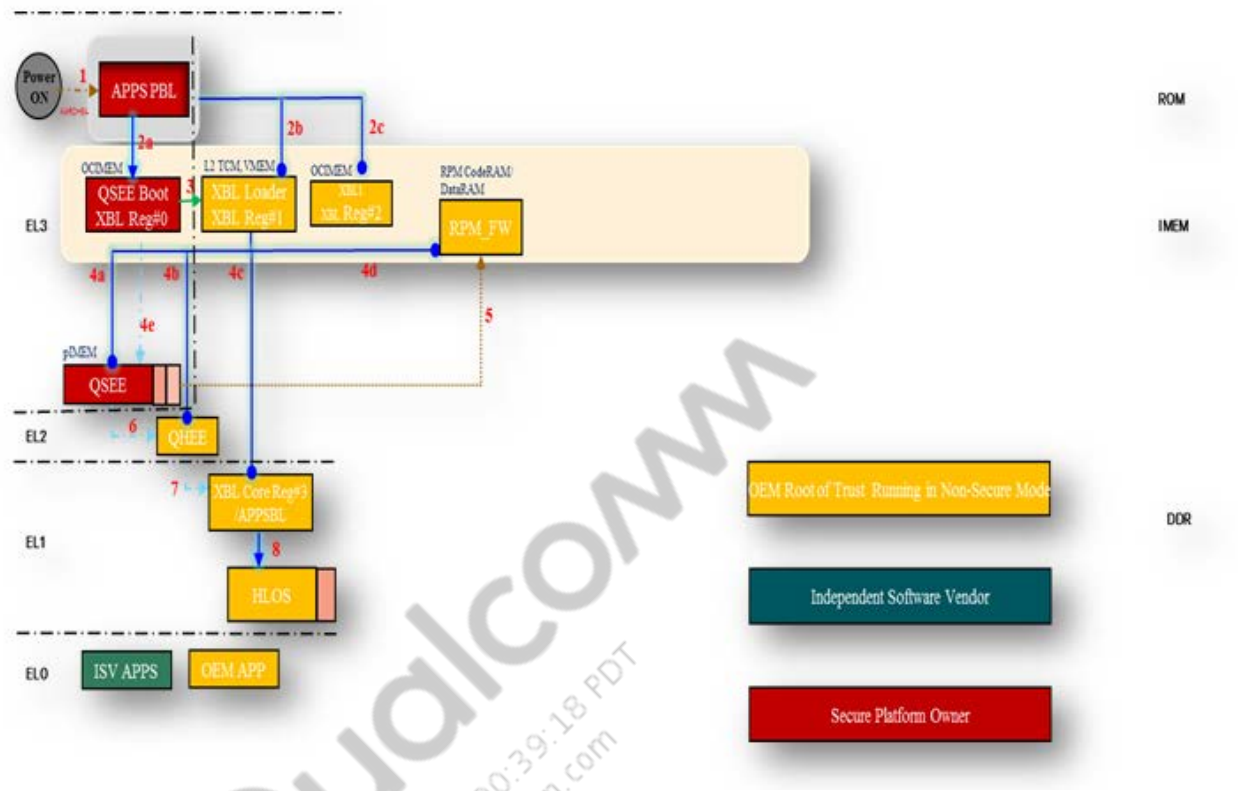


Figure 2-11 Double signed images in SDM636/SDM660/SDM630/SDM670 boot sequence

The double signing can be extended to more images in the future.

There are two different image formats for double signing in terms of how code signature is packed into final signed binary image. The two formats are as follows:

1. Double signing for XBL_SEC.ELF/Flash_Programmer.ELF
2. Double signing for QSEE.ELF.

2.6.1.1 Double signing for XBL_SEC.ELF

Signature packing format for XBL_SEC.ELF and Flash_Programmer.ELF is as follows and is shown in the following Figure 2-12.

- Hash table segment of XBL_SEC.ELF is signed with QTI ROT.
- Signed XBL_SEC.ELF is inserted as one of the segments into the XBL.ELF
- Hash Table Segment of XBL.ELF is updated with the hash value of the inserted XBL_SEC segment.
- Hash segment of XBL.ELF is signed with the ROT of the OEM.

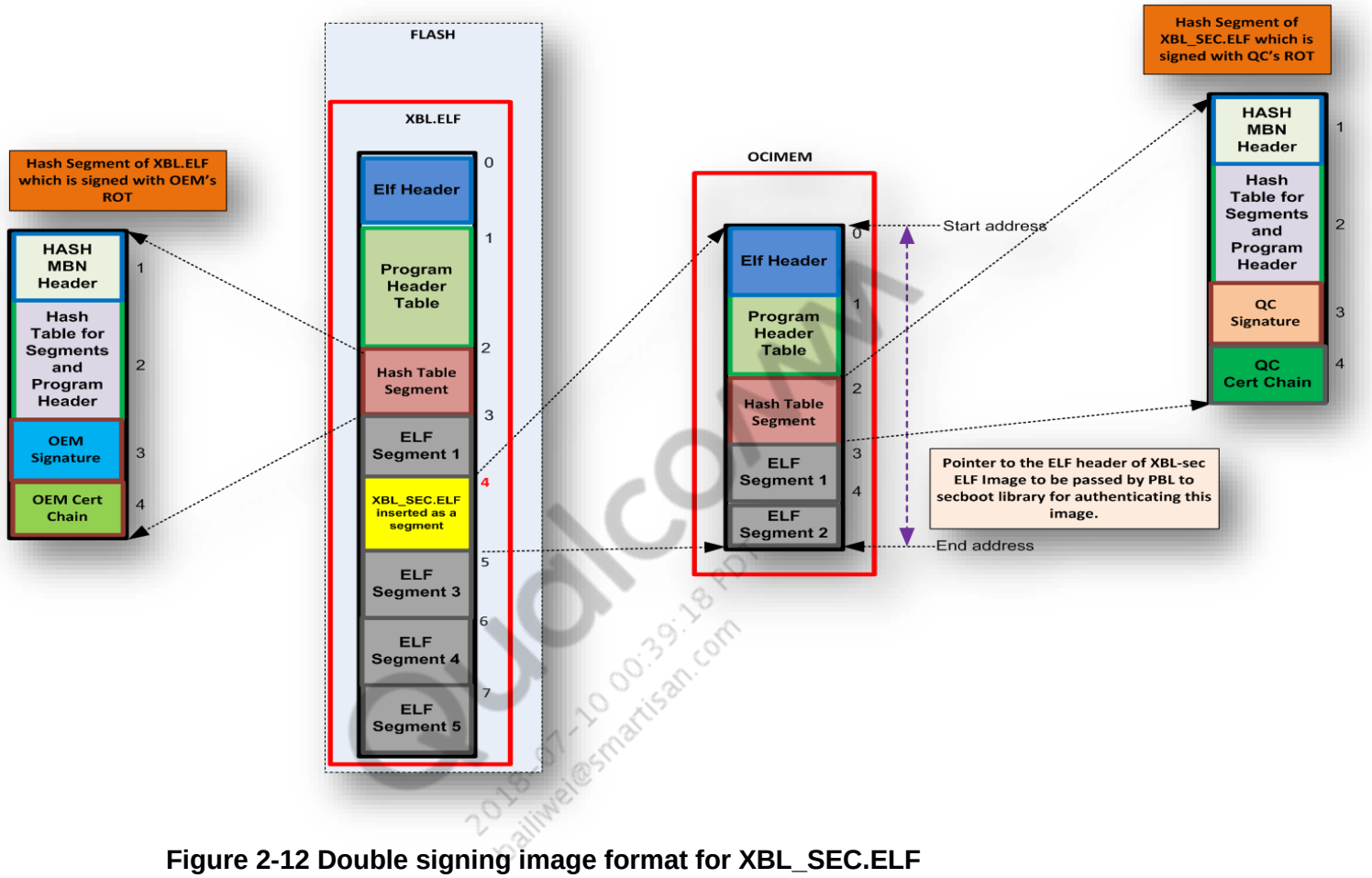


Figure 2-12 Double signing image format for XBL_SEC.ELF

2.6.1.2 Double signing for QSEE.ELF

Signature packing format for QSEE.ELF is as follows:

- Hash table segment of QSEE.ELF is signed with QC ROT. QC ROT signature is inserted in hash segment and space is reserved for OEM ROT signature in hash segment.
- Hash table segment of QSEE.ELF is signed with OEM ROT. OEM ROT signature is inserted in space reserved for OEM ROT signature in hash segment.
- This results in hash table segment with hash segment header followed by hash table, followed by OEM ROT signature and certificate chain, followed by QC ROT signature and certificate chain, followed by any UIE data (if any).

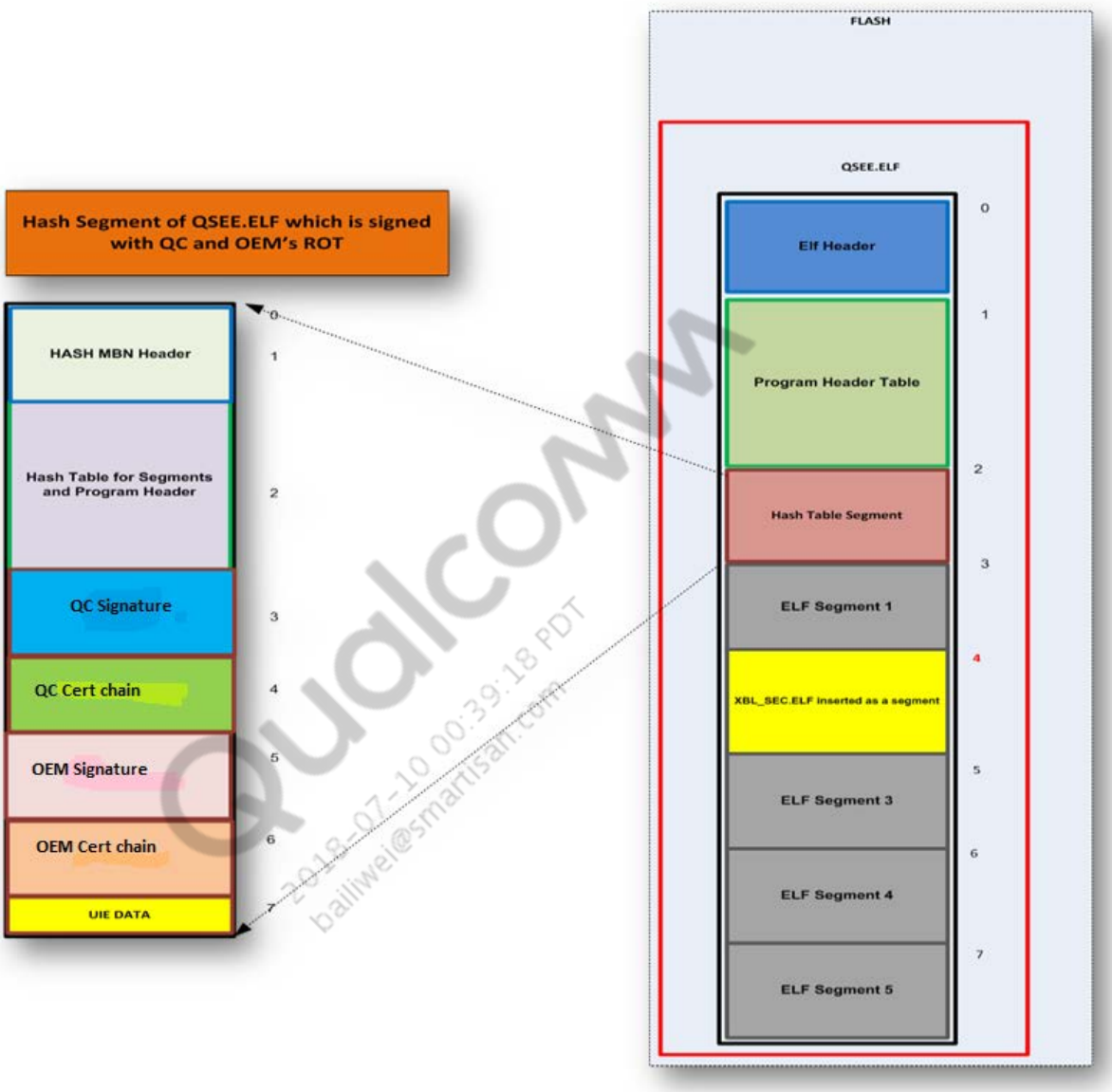


Figure 2-13 Double signing image format for QSEE.ELF

2.6.1.3 Double signing for QUPV3FW.ELF – Applicable for SDM670 chipsets

Signature packing format for QUPV3FW.ELF is as follows.

- Hash table segment of QUPV3FW.ELF is signed with QC ROT. QC ROT. Signature is inserted in hash segment and space is reserved for OEM ROT signature in hash segment.
- Hash table segment of QUPV3FW.ELF is signed with OEM ROT. OEM ROT signature is inserted in space reserved for OEM ROT signature in hash segment.
- This results in hash table segment with hash segment header followed by hash table, followed by OEM ROT signature and certificate chain, followed by QTI ROT signature and certificate chain, followed by any UIE data (if any).

2.6.1.4 List of images, which needs double signing

1. Tz.mbn (xbl sec and qsee are packed into tz.mbn)
2. qupv3fw.elf (applicable for SDM670 chipsets)

NOTE: As per Section 2.2.1.4.3, oem_id_independent should be 0 for OEM use.

2.6.2 Multiple root certificate support for SDM670 chipsets

The multiple root certificate (MRC) feature allows an OEM to select one from a maximum of our root certificates to be the trust anchor of the certificate chain that is used to sign and verify the image. The root hash of a concatenation of the N number of root certificates must be blown to the OEM PK HASH eFuse, where N varies from 1 to 4. One of the N root certificates can then be selected to sign of the images. Later, an OEM can re-sign the images with a different root certificate selected if necessary. All of the root certificates that were concatenated and used in the calculation of the root hash should be packaged in the signed image, so that a comparison can be made with the OEM PK HASH blown in eFuse.

OEMs can select a specific root by using the eFuse ROOT_CERT_ACTIVATION_LIST and revoke a root by using eFuse ROOT_CERT_REVOCATION_LIST. The total number of root certificates should also be blown into eFuse ROOT_CERT_TOTAL_NUM.

3 QFPROM configuration

NOTE: Numerous changes were made in this chapter.

QFPROM is one-time programmable memory. The QFPROM configuration enables the security environment that is required for secure boot. QFPROM writing is irreversible, so exercise the highest level of caution during the fuse blowing process.

3.1 QFPROM RAW address and corrected address

QFPROM accesses the RAW regions that are intended for fuse blowing and checking only.

It also accesses the corrected address space that is intended for use by the software. This address space is read-only.

3.2 QFPROM code segment

QFPROM, like OEM_SEC_BOOT, has many configurations for different code segments. You can find SEC_BOOTn, where n ∈ [1, 7] from OEM_SEC_BOOT. This scheme allows for different SEC_BOOT configurations even on a single device; it also implies that multiple SROTs are possible. The code segment for the SDM636/SDM660/SDM630/SDM670 chipset is defined as follows.

```
/**
 Identifies the secure boot fuses that represent the authentication information for
the code.
 */
#define SECBOOT_HW_APPS_CODE_SEGMENT 1 /**< Code segment in SECURE_BOOTn register */
                                       /**< representing authentication information */
                                       /**< for the application processors images */

#define SECBOOT_HW_MBA_CODE_SEGMENT 2  /**< Code segment in SECURE_BOOTn register */
                                       /**< representing authentication information */
                                       /**< for the Modem Boot Authentication (MBA) */
                                       /**< image */

#define SECBOOT_HW_MSS_CODE_SEGMENT 3  /**< Code segment in SECURE_BOOTn register */
                                       /**< representing authentication information */
                                       /**< for the modem image */
```

3.3 Blowing secure boot fuses

The following fuses must be blown to enable secure boot (authentication of images in the bootup sequence).

Table 3-1 Secure boot fuses

QFPROM_RAW_OEM_SEC_BOOT_ROWn_LSB				
Bits	**Name**	**Description**		**Used for**
23:16	SEC_BOOT3	Bits	Name	For authentication information of MPSS (modem) image
		7	RESERVED	
		6	USE_SERIAL_NUM	
		5	AUTH_EN	
		4	PK_HASH_IN_FUSE	
		3:0	ROM_PK_HASH_INDEX	
15:8	SEC_BOOT2	Bits	Name	For authentication information of MBA (modem boot authenticator) image (Not applicable for APQ8084)
		7	RESERVED	
		6	USE_SERIAL_NUM	
		5	AUTH_EN	
		4	PK_HASH_IN_FUSE	
		3:0	ROM_PK_HASH_INDEX	
7:0	SEC_BOOT1	Bits	Name	For authentication information of all other images: ▪ XBL ▪ RPM firmware ▪ TZ kernel ▪ Hypervisor ▪ APPSBL ▪ TZ Application images ▪ WCNSS (WLAN/RIVA) ▪ LPASS ▪ Video (Venus) ▪ Emergency downloader ▪ Debug watchdog
		7	RESERVED	
		6	USE_SERIAL_NUM	
		5	AUTH_EN	
		4	PK_HASH_IN_FUSE	
		3:0	ROM_PK_HASH_INDEX	

AUTH_EN

Blow to mandate authentication of the associated images.

PK_HASH_IN_FUSE

Blow when using the own code signing system of the licensee (that is, the licensee has their own signing keys), and the licensee has provisioned the hash of their trusted root certificate in the OEM_PK_HASH fuses.

The authentication logic uses the hash of the trusted root certificate to validate that the certificate chain is trusted (that is, the public keys within the certificates comes from the authorized source of the licensee).

NOTE: If the licensee uses QTI CSMS with the SHA256 option in "Signing Root Certificate" then this fuse must be blown with the hash of the trusted root certificate.

ROM_PK_HASH_INDEX

Blow if the root certificate hash of the licensee already resides in the QTI key table in the boot ROM. This key table contains hashes of 12 known root certificates. The QTI CSMS root certificate resides at index 0.

These fuses must be blown to the index in the key table that contains the hash of the root certificate of the licensee.

If the licensee uses CSMS for signing, this fuse must not be blown (that is, it remains 0) because the CSMS root certificate is at index 0 in the key table.

USE_SERIAL_NUM

Blow when the licensee is signing with the unique chip serial number value contained in the QFPROM_CORR_SERIAL_NUM_LSB fuses. This indicates to the authentication logic that it is to use the serial number value to authenticate the image.

If this fuse is not blown, the authentication logic uses the value in the OEM_ID hardware register that contains the OEM_HW_ID values from the QFPROM_RAW_OEM_CONFIG_ROW1_MSB and OEM_MODEL_ID values in the QFPROM_RAW_OEM_CONFIG_ROW1_LSB fuses for authentication.

3.4 Blow OEM identifier fuses

The following fuses must be blown if the licensee uses CSMS or wants to use a nonzero value for the lower 32 bits of MSM_HW_ID (see Section 5.1).

Table 3-2 OEM identifier fuses

QFPROM_RAW_OEM_CONFIG_ROW1_MSB		
Bits	**Name**	**Description**
15:0	OEM_HW_ID	QTI assigns the OEM an ID for secure boot and license management. This OEM_ID must be burned with the assigned ID of the OEM, so that the OEM can use this capability to support either QTI secure boot solutions (CSMS) or DSP dynamic feature authentication of QTI or third-party feature enhancements. The DSP enables feature authentication and ensures that the OEM approves DSP dynamic updates. So even if an OEM does not use CSMS for signing, blow the OEM_ID with the assigned ID.
31:16	OEM_PRODUCT_ID	Blow this eFuse with nonzero values. Different models based on the same MSM devices must have a unique Model_ID to prevent misuse of images across various models.

3.5 Disable debug ports

When secure boot is enabled, OEMs must blow the debug disable fuses to prevent access to the processors of the device through the JTAG ports. This action is essential to ensure that secure boot cannot be bypassed.

OEMs might also want to disable the various tracing capabilities.

3.6 Use debug overrides

The one-time writable debug override registers can be used to either keep the debug-disable fuse settings or to re-enable JTAG (override) the debug-disable fuses settings. These override registers are written to only once during each power cycle, and subsequent writes to them do not take effect.

The APPS PBL writes the bit fields of OVERRIDE_2/OVERRIDE_3 based on the debug settings in the XBL image attestation certificates (see Section 2.2.1.3).

Bit fields of OVERRIDE_4 are written to by the Modem PBL based on the debug settings in the MBA image attestation certificates (see Section 2.2.1.3).

3.6.1 Re-enable JTAG access

Typically, field devices are locked from JTAG access to enable certain hardware security features. But by writing to the override registers described in Section 3.6, JTAG access can be re-enabled. To permit access and unlock a device, the OEM must sign the XBL and MBA image with a specific value of the DEBUG OU option.

If the OEM wants to disable the ability to re-enable JTAG access through possible use of override registers, all the images (including XBL and MBA) must be signed with DEBUG OU = 0x0000000000000002. This signing results in APPS_PBL and MODEM_PBL writing 0x0 to the override registers. Since the override registers are only one-time writable in a device reset cycle. The hardware itself prohibits further access to enable JTAG.

If the OEM wants to re-enable JTAG access using the override registers, the XBL and MBA images must be signed with DEBUG OU = (SERIAL_NUM « 32) | 0x00000003. Other software images need not be resigned and can retain the old default value of DEBUG OU = 0x0000000000000002. This also allows the MBA and TZ to enable the RAM dump feature for the various subsystems. The DEBUG OU field is 64-bit, because it must change to (SERIAL_NUM « 32) | 0x00000003. Re-enabling JTAG on the SDM636/SDM660/SDM630/SDM670 devices is done in a per-device manner; SERIAL_NUM can be read from 0x740138, QFPROM_CORR_PTE_ROW1_LSB for SDM636/SDM660/ SDM630 chipsets and from 0x786134, QFPROM_CORR_PTE_ROW1_LSB for SDM670 chipset.

Also, to avoid any misuse of the debug-enabled signed images, it is important to tie the debug image only to a specific unit. Before enabling the JTAG access, the MODEM_PBL and APPS_PBL match the upper 32 bits of the DEBUG OU field against the serial number of the device read from the QFPROM_CORR_SERIAL_NUM_LSB register. All devices carry a unique value for QFPROM_CORR_SERIAL_NUM_LSB.

3.7 Enable SFS

To enable SFS:

1. Licensee should enable secure boot.
2. Secondary key dervation key should be blown (See Appendix B, FAQ 31 for SKDK).
3. JTAG disable fuses should be blown. For more details on JTAG disable fuses see the OEM_CONFIG sheet of QFPROM Programming Reference Guide (See D.1)

3.8 RPMB provisioning

RPMB key provisioning is a one-time operation for the lifetime of eMMC. Once provisioned, it cannot be altered or deleted. The RPMB key type is based on the security fuses state. The OEM must be aware of the security fuses state when picking up the RPMB key provision type. A key mismatch between eMMC and the host will lead to dysfunctional RPMB storage.

Some scenarios that can lead to authentication key mismatch between eMMC and host:

- Blowing the security fuse or JTAG disable fuse after the RPMB key is provisioned
- Enabling debug policy in a commercial device i.e JTAG enable flag in debug policy.

Hence it is advised to not enable secure boot on a device in which RPMB is provisioned.

4 TZ board support package (BSP)

For more details on the TZ BSP for the SDM636/SDM660/SDM630/SDM670, see *QSEE Version 4.0 User Guide* (80-NH537-2).

5 Security components

NOTE: Numerous changes were made in this chapter.

5.1 MSM_HW_ID

MSM_HW_ID is a 64-bit variable and the value of this variable is calculated at runtime based on device fuse/registers settings and OU fields present for the signed image that is being authenticated.

MSM_HW_ID variable ensures that the image authentication is successful only on hardware, which is specified in signed Image. A signed image specifies a supported hardware platform by embedding the identity of the supported hardware platform in attestation certificate in form of OU fields.

MSM_HW_ID check, which is done at load and authenticate stage for each image, consists of comparing hardware identity derived from hardware and from signed image. Hardware identities are compared by the two following methods:

1. Hardware identity derived from device fuse/registers settings and OU fields from signed image, this hardware identity is called as MSM_HW_ID.
2. Hardware identity derived from signed image OU = 02 XXXXXXXXXXXXXXXX HW_ID field, this hardware identity is called as HW_ID.

If the MSM_HW_ID from the device and the HW_ID from the signed image are equal, then Image authentication proceeds to the next step of authentication, otherwise Image authentication fails.

Figure 5-1 shows key elements and algorithm used to derive MSM_HW_ID and HW_ID on device from fuses and signed image OU fields.

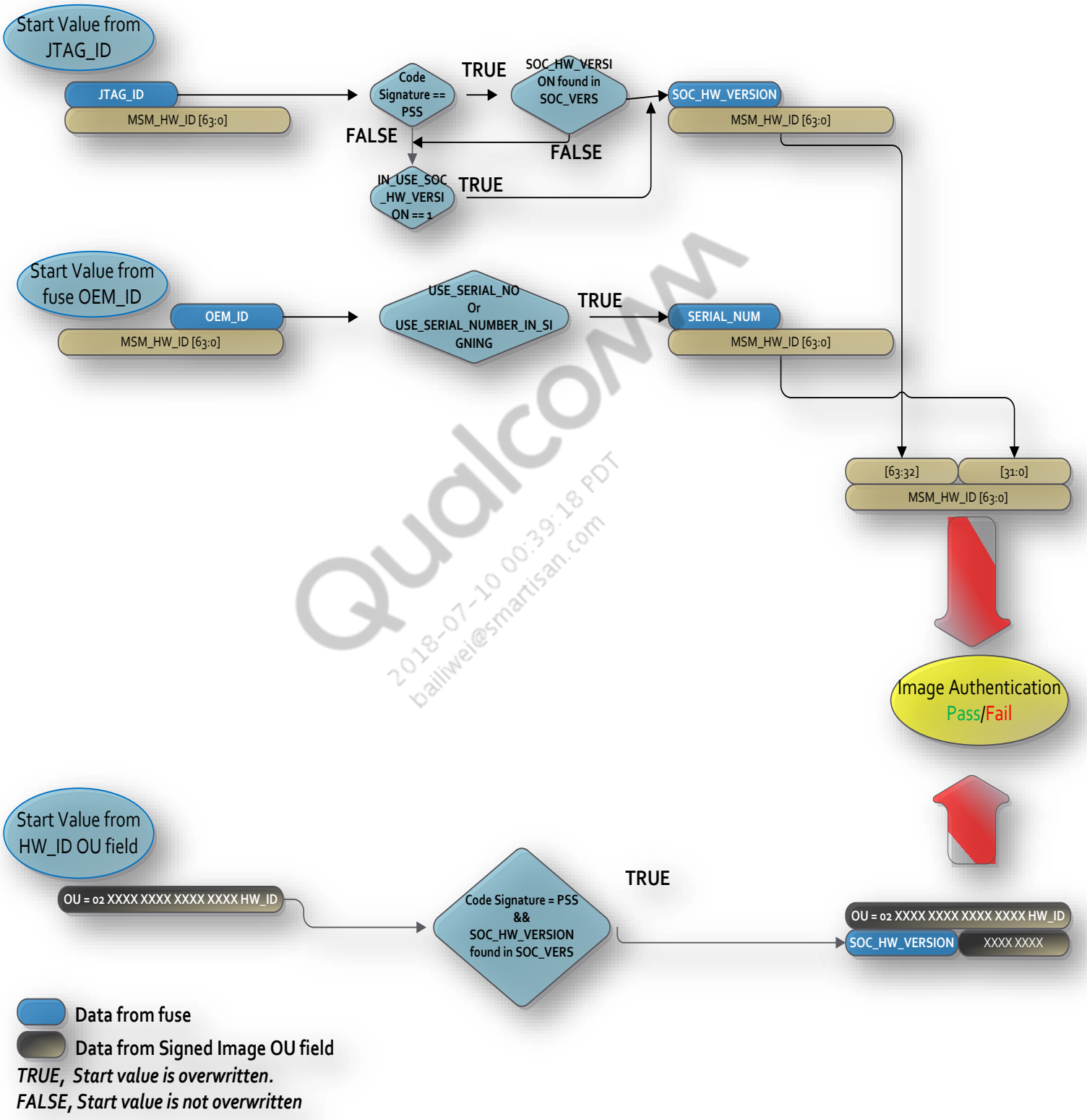


Figure 5-1 MSM_HW_ID check

The fuses and registers, which are used to determine hardware platform identity on device, are as following:

1. OEM_ID: OEM_ID register holds the OEM ID and OEM Product ID values as 16-bit values configured by OEM.OEMs configure OEM_ID and OEM_PRODUCT_ID by blowing 16 bit value to fuse row QFPROM_RAW_OEM_CONFIG_ROW1_MSB.When QFPROM_RAW_OEM_CONFIG_ROW1_MSB is blown with the OEM_ID and OEM_PRODUCT_ID, the Security Control Core (SCR) OEM_ID register reflects the blown values.

Bits	Name	Description
31:16	OEM_ID	The OEM identifier
15:0	OEM_PRODUCT_ID	The OEM product identifier

2. JTAG_ID: JTAG_ID register holds the JTAG ID of the device. The JTAG ID fuses controls the upper 20 bits of this register. The lower 12 bits are a constant value of 0x0E1 as this is QTI identifier.

Bits	Name	Description
31:0	JTAG_ID	The chip JTAG identifier. Bits 31-28: Die Revision/Version Bits 27-12: Product/Device ID Bits 11-0: Manufacturer ID (Fixed for QTI @ 0x0E1)

The top 4 bits of JTAG_ID contain the chip revision, which is not taken into account when computing the MSM_HW_ID value. This omission ensures that the signed software works for any version of a specific MSM device.

Within the same variant of a device, such as MSM8996, MSM8096 etc., JTAG_ID has different values. It can be cumbersome to manage different images for variants even though all variants can work with the same binary. To overcome this issue, there is the option to write the OU14 field in the attestation certificate as 0x1. If OEMs use this option, all the images can be signed with the single value of SOC_HW_VERSION for all the variants of the device.

3. SOC_HW_VERSION:SOC_HW_VERSION is a 16-bit value, which consists of FAMILY_NUMBER and DEVICE_NUMBER from TCSR_SOC_HW_VERSION register. TCSR_SOC_HW_VERSION register provide version information of the chip and details are as following:

Bits	Name	Description
31:28	FAMILY_NUMBER	bit[31:28]: Unique number per family
27:16	DEVICE_NUMBER	bit[27:16]: Unique number per device within family
15:8	MAJOR_VERSION	bit[15:8]: Major number: Tracks all-layer revs
7:0	MINOR_VERSION	bit[7:0] Minor number: Track metal revs

4. USE_SERIAL_NUM fuse bit in OEM_SEC_BOOT_ROW fuse region. The USE_SERIAL_NUM fuse bit determines whether the OEM_ID or SERIAL_NUMBER is used for authentication. The OEMs can also choose to personalize the phone by signing the images for each phone with the chip serial number instead of the OEM_ID, by blowing the authentication use serial number bit (bit 6) in bit fields SECBOOTn of the QFPROM_RAW_OEM_SEC_BOOT_ROW register. If this bit is blown, the 32-bit SERIAL_NUM is used instead of the OEM_ID in the authentication.
5. SERIAL_NUM: This register holds the unique 32-bit Serial Number of the device.

Bits	Name	Description
31:0	SERIAL_NUM	32-bit Serial Number value

The OU fields, which play pivotal role in determining supported hardware platform by signed image are as following:

```
1. OU = 02 XXXXXXXXXXXXXXXX HW_ID (MANDATORY)
2. OU = 14 0000000000000000 IN_USE_SOC_HW_VERSION (OPTIONAL)
3. OU = 12 0000000300000000 SOC_HW_VERSION (OPTIONAL)
4. OU = 12 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXXX XXXX SOC_VERS
   (OPTIONAL)
5. OU = 14 0000000000000000 USE_SERIAL_NUMBER_IN_SIGNING (OPTIONAL)
6. OU 17 = SN (optional, six serial numbers separated by space; there can
   be up to 34 OU 17 to support 200 serial numbers)
```

The following is descriptions of above OU fields:

1. **OU = 02 XXXXXXXXXXXXXXXX HW_ID (MANDATORY)**: HW_ID OU field in Image attestation certificate holds the hardware identity on which this signed image is allowed to execute. Licensee can populate HW_ID OU field to bind image to
   a. Hardware platforms with same JTAG_ID.
   b. Hardware platforms with same SOC_HW_VERS.
   c. Multiple hardware platforms by specifying multiple SOC_HW_VERS.
   d. Hardware platform with specific OEM_ID.
   e. Specific device by specifying serial number that is, image loads only on one particular device specified by Serial no, which is unique to each MSM chip.

In Sectools, HW_ID OU field is generated based on following XML tags defined in <sectools>\config\sdm660\sdm660_secimage.xml:

```
   <msm_part>0x000XXXXX</msm_part>
<soc_hw_version>0xXXXXXXXX</soc_hw_version>
<in_use_soc_hw_version>X</in_use_soc_hw_version>
<oem_id>0xXXXX</oem_id>
<model_id>0xXXXX</model_id>
<use_serial_number_in_signing>X</use_serial_number_in_signing>
<serial_number>0xXXXXXXXX</serial_number>
<soc_vers>0xXXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX </soc_vers>
```

Based on the XML tags HW_ID OU fields is generated as following:

Table 5-1 HW_ID OU field

<table>
<tr><th colspan="5">HW_ID[63:0]</th></tr>
<tr><td colspan="3">[63:32]</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>in_use_soc_hw_version = 0</td><td>in_use_soc_hw_version = 1 & SOC_HW_VERSION</td><td>in_use_soc_hw_version = 1
& SOC_VERS</td></tr>
<tr><td rowspan="3">MSM_PART masked with 0x0FFF_FFFF</td><td rowspan="3">SOC_HW_VERSION masked with 0xFFFF</td><td rowspan="3">SOC_HW_VERSION masked with 0xFFFF</td><td>[31:16]</td><td>[15:0]</td></tr>
<tr><td>OEM_HW_ID</td><td>OEM_MODEL_ID</td></tr>
<tr><td colspan="2">SERIAL_NUMBER
(use_serial_number_in_signing =1)</td></tr>
</table>

If <in_use_soc_hw_version>, <soc_hw_version> and <soc_vers> are not used then upper 32-bits of HW_ID OU field is populated by <msm_part> MSM_PART is JTAG_ID of a chipset).

SOC_VERS is newly added OU field for SDM636/SDM660/SDM630/SDM670 platform and applicable only if code signature is generated with PSS padding that is, <rsa_padding>pss</rsa_padding> is used in Sectools when signing the images.

1. OU = 14 0000000000000000 IN_USE_SOC_HW_VERSION and OU = 12 0000000300000000 SOC_HW_VERSION: Licensee can choose to tie the signed binary image to a family of chipset that is, tie to SOC_HW_VERSION instead of tying to JTAG_ID.The following tags can be used to specify SOC_HW_VERSION
   ```
   <in_use_soc_hw_version>1</in_use_soc_hw_version>
   <soc_hw_version>0xXXXXXXXX</soc_hw_version>
   ```
2. OU = 12 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXXX XXXX SOC_VERS: The licensee can choose to tie the signed binary image to multiple families of chipset that is, tie to multiple SOC_HW_VERSION instead of tying to JTAG_ID or single SOC_HW_VERSION.The following tags can be used to specify SOC_HW_VERSION
   ```
   <in_use_soc_hw_version>1</in_use_soc_hw_version>
   <soc_vers>0xXXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
   </soc_vers>
   ```
3. OU = 15 0000000000000000 USE_SERIAL_NUMBER_IN_SIGNING: Licensee can choose to sign binary image for a particular chip by specifying chip serial number.The following tags can be used to specify SERIAL NUMBER:
   ```
   <use_serial_number_in_signing>1</use_serial_number_in_signing>
   <serial_number>0xXXXXXXXX</serial_number>
   ```

5.2 Software ID

The software ID has changed in the secure boot architecture to support more images. The bits in the SW_ID OU field in the signed image must match the following code.

For XBL and eHOSTDL, which the PBLverifies, the lowest 32 bits in the SW_ID OU field must match either 0x0 (XBL) or 0x3 (eHOSTDL).

NOTE: The PBL can match a maximum of two software IDs at the same time.

The highest 32 bits must match the antireplay fuse settings, as described in Chapter 5.

SW_ID definitions

Table 5-2 SW_ID definitions

SWID	SDM636/SDM660/SDM630/SDM670 images	SWID	SDM636/SDM660/SDM630/SDM670 images
0x0	XBL	0XE	Video hash table
0x1	MBA	0xF	Reserved
0x2	AMSS_HASH_TABLE	0x12	Reserved
0x3	EHOSTDL	0x13	Reserved
0x4	DSP_HASH_TABLE	0x14	GPU microcode
0x5	Device configuration image	0x15	QHEE
0x6	Reserved	0x16	PMIC
0x7	QSEE	0x17	CDSP
0x8	Reserved	0x18	SLPI
0x9	APPSBL	0x19	EOS firmware image
0xA	RPM firmware	0x1A	Validated image programmer (VIP)
0xB	Reserved	0x1B	EFS TAR
0xC	TZ exec hash table (Apps)	0x1C	ABLs
0xD	Reserved	0x200	Debug policy (DP)

In the SDM636/SDM660/SDM630/SDM670 chipset, the HLOS resets the modem, but the modem has its own PBL, so the modem PBL actually authenticates the MBA image. See Chapter 6 for more details. Therefore, the MBA image ID also has a different image ID: BOOT_SW_MBA_TYPE, which is 0x1. To sign the MBA image, use 0x1 instead of 0x13 (19).

5.3 Code signing

In SDM636/SDM660/SDM630/SDM670 chipset, the PBL only allows images that are signed with software type 0 (XBL) and 3 (device programmer). This restriction disables other images that are presented to the PBL for authentication. The software version is used for rollback prevention and ensures that legacy software cannot be used on the phone after a software update that fixes critical bugs is applied. Software versioning is enforced by signing the image with the new software version and blowing the software version fuses to prevent a rollback.

5.4 Crypto engine usage

For the SDM636/SDM660/SDM630/SDM670 chipset, crypto engine usage is:

- Modem CE – Accessible by MBA and MPSS
- Apps CE – Accessible by XBL, TZ, LK, and HLOS

The modem EFS uses PHK, while SHK is used for SFS.

Table 5-3 Hardware key allocation for image

Key	Modem	TZ	MBA	XBLs	HLOS
MSS Crypto5 secure/non-secure QTI key	✓	-	-	-	-
Secure/non-secure OEM key	-	✓	-	-	-
Crypto5_0 secure/non-secure QTI key	-	✓	-	-	✓

5.5 SHA1 to SHA256 transition

To improve security and performance (15% faster hash speed), SHA256 widely replaces SHA1 in secure boot for these chipsets. Because of concerns about the security of SHA1, use SHA256 for both ELF hashes and certificates. Details of how to make the transition are described in Section 5.5.2.

5.5.1 ELF section hash algorithm

Previously, the ELF/PIL image used SHA1 to create an SHA1 hash for each loadable ELF section; however, in SDM636/SDM660/SDM630/SDM670 the hash (not signature) algorithm was changed to SHA256.

5.5.2 Signature in certificate chain

SHA256 is the recommended signature algorithm in the certificate chain. This change only affects the signature algorithm when the root CA signs the attestation CA, and the attestation CA signs the attestation certificate. It does not affect how the attestation certificate signs the target image, which is already SHA256, subject to the OU field in the attestation certificate (OU = 07 0001 SHA256). To adapt that, the OEM must recreate the entire certificate chain; see Appendix 0. Argument 1 must be SHA256. SHA1 as the signature algorithm for the certificate chain might become obsolete in the future for security concerns.

For CSMS, note that because the certificates chain is recreated, PBL no longer has the root certificate hash in PBL ROM. The OEM must blow PK_HASH_IN_USE/OEM_PK_HASH with the new root CA hashed properly.

6 Rollback prevention

Antirollback is used to determine if the current software version is at least greater than the one provided in the fuses. The software version is in the upper 32 bits of the SW_ID OU field attestation certificate. There is no FEC for the antirollback fuses.

The software only reads the numbers of bits in the rollback fuse. For a 16-bit value, the maximum value is 15; ensure to prevent a 16-version rollback.

The software only reads corrected QFPROMs, which are read-only. To program the QFPROM, program the RAW QFPROMs. For details, see Section 3.1.

6.1 Enable antirollback

The rollback feature is independently activated for the various images based on following bits of QFPROM QFPROM_CORR_OEM_CONFIG_ROW2_MSB[3:0].

Table 6-1 QFPROM antirollback enable register

Bit position	Name	Subsystem
0	BOOT_ANTI_ROLLBACK_EN	XBL, RPM, TZ, APPSBL, HYP, Apps DP Policy
1	TZAPPS_ANTI_ROLLBACK_EN	TZ secure applications
2	PIL_SUBSYS_ANTI_ROLLBACK_EN	PIL images (loaded by TZ)
3	MSA_ANTI_ROLLBACK_EN	MSA, Modem, DP

The secure boot feature is mandatory to perform rollback prevention.

Find the correct versions of the images in the device by counting the number of bits set by QFPROM, as shown in Table 6-2.

Table 6-2 Bits set by QFPROM

Image type	Maximum version of secure upgrade supported	HWIO	Bitmask
XBL/DeviceProgrammer (SECBOOT_SBL_SW_TYPE) APPSBL (SECBOOT_APPSBL_SW_TYPE) PMIC (SECBOOT_PMIC_SW_TYPE)	64	QFPROM_CORR/RAW_ANTI_ROLLBACK_1_LSB[31:0] QFPROM_CORR/RAW_ANTI_ROLLBACK_1_MSB[31:0]	0xffffffff 0xffffffff
TZ (SECBOOT_QSEE_SW_TYPE)	17	QFPROM_CORR/RAW_ANTI_ROLLBACK_2_MSB[16:0]	0x1ffff
Hypervisor (SECBOOT_QHEE_SW_TYPE)	12	QFPROM_CORR/RAW_ANTI_ROLLBACK_3_MSB[11:0]	0xfff
RPM (SECBOOT_RPM_FW_SW_TYPE)	8	QFPROM_CORR/RAW_ANTI_ROLLBACK_2_MSB[24:17]	0x1fe0000
TZ Devcfg (SECBOOT_QSEE_DEVCFG_SW_TYPE)	11	QFPROM_CORR/RAW_ANTI_ROLLBACK_3_MSB[27:17]	0xffe0000
Apps DP (SECBOOT_APDP_SW_TYPE)	5	QFPROM_CORR/RAW_ANTI_ROLLBACK_3_MSB[16:12]	0x1f000
PILs (except Modem)	48	QFPROM_CORR/RAW_ANTI_ROLLBACK_2_LSB[31:0] QFPROM_CORR/RAW_ANTI_ROLLBACK_3_LSB[23:8]	0xffffffff 0xffff00
MBA	16	QFPROM_CORR/RAW_ANTI_ROLLBACK_4_LSB[15:0]	0xffff
Modem PIL (SEC_BOOT_MSS_SW_TYPE) MSA DP (DBG_POLICY_ELF_IMAGE_ID)	16	QFPROM_CORR/RAW_ANTI_ROLLBACK_4_LSB[31:16]	0xffff0000
TZ secure applications	0xFFFFFFFF (using RPMB)	NA	

The software version is determined from the SW_ID OU[63:32] field contents in the attestation certificate. The lowest versioned number of a specific image that can be run is equal to or greater than the value in fuse.

The image type is determined from the SW_ID OU[31:0] field contents in the attestation certificate.

6.2 MBA antirollback

The Modem PBL verifies that QFPROM_RAW_ANTI_ROLLBACK_4_LSB [15:0] is less than or equal to the SW_ID OU field in the MBA attestation certificate.

The MBA verifies that QFPROM_RAW_ANTI_ROLLBACK_4_LSB [31:16] is less than or equal to the SW_ID OU field in the PMI attestation certificate.

If either condition fails, the RMB error registers are set.

6.3 TZ antirollback

QFPROM_RAW_ANTI_ROLLBACK_2_MSB [16:0] specifies TZ antirollback.

6.4 XBL antirollback

The APPS PBL verifies that QFPROM_RAW_ANTI_ROLLBACK_1_MSB[31:0], QFPROM_RAW_ANTI_ROLLBACK_1_LSB [31:0] is less than or equal to the SW_ID OU field in the XBL attestation certificate.

6.5 Software version rollback prevention examples

SW_ID is inserted into the attestation certificate when the image is signed; a persistent memory is reserved to save the oldest software version that can be loaded onto the device.

When bit 0 of both BOOT_ANTI_ROLLBACK_EN and PIL_SUBSYS_ANTI_ROLLBACK_EN is set to 1, the boot loader and TZ are configured to do a software version rollback check, stored in SW_ID, against the oldest software version, which is stored in persistent memory.

The software version is for rollback prevention purposes only. A licensee must increment the version only when a previous version of the boot loader package is no longer to be loaded, that is, to close a security hole.

The software version value saved in the attestation certificate is interpreted as a hexadecimal value. The licensees set the software version value when they sign an image.

To get the SW_ID to sign a TZ build with version 0x2:

- The hexadecimal value of SECBOOT_TZ_KERNEL_SW_TYPE is 0x7.
- SW_ID configuration
  - Bits 0 to 31 – Used to store the image type
  - Bits 32 to 63 – Bits set in QFPROM for antirollback functionality
  - SW_ID = 0x0000000200000007

The software version value saved in QFPROM_RAW_AP_ANTI_ROLLBACK_ROW0_ LSB[31:16] is interpreted by the number of bits set as 1. The secure boot loader updates the QFPROM software version value:

- 0000000000000000000 = 0
- 0000000000000000001 = 1
- 0000000000000000111 = 3
- 1111111111111111111 = 20

The maximum eFuse version value is 16 for the TZ image.

6.6 Software version rollback prevention use case

A phone starts with software version 0, and a licensee then decides to upgrade the TZ software version to 1. The licensee then must sign TZ images with the new SW_ID.

TZ with SW_ID = 0x0000000100000007

If the TZ image is signed with a lower software version, the phone cannot boot up. For example, if the TZ image is signed with SW_ID = 0x0000000000000007 (a lower software version), then the device fails to authenticate the TZ image and fails to boot.

Similarly, the boot loader does a software version rollback check for APPSBL/TZ/RPM during phone bootup. If a newer software version image is loaded, the boot loader blows a fuse with the latest higher software version to roll back the fuse accordingly.

6.7 Software version rollback limitation

The rollback version prevention has a limitation on a software version that exceeds the maximum eFuse version. For a specified image, if the software version exceeds the maximum eFuse version, the image bypasses the version check.

The next example shows that the licensee can track versions only up to the maximum version number/number of bits allotted for various images. If the number of versions exceeds the maximum, only the maximum allowed bits, 20, are blown. For all later bootups, builds with versions less than the maximum of 19 are not allowed to proceed; only the maximum versions and up are allowed.

A phone can successfully boot up with the TZ image signed with SW_ID = 0x0000001700000007, even if the software version is 0x17 (decimal 23), which exceeds the maximum eFuse version (16). After that, the following occurs:

- If the phone loads an old TZ image signed with SW_ID = 0x0000001000000007, software version = 0x10 = 16, the phone can boot up. Basically, any software version ≥ 16 can be loaded.
- If the phone loads an old TZ image signed with SW_ID = 0x000000F000000007, software version = 0xF = 15, the phone cannot boot up. Any software version < 16 cannot be loaded.

6.8 TZ secure application antirollback

RPMB is a signed access to a replay protected memory block. This function provides a means for the system to store data to the specific memory area in an authenticated and replay-protected manner. This is provided by first programming authentication key information to the eMMC memory (shared secret).

The QFPROM software version value (Section 6.5) does not apply to TZ secure application antirollback. TZ saves a secure application APP_ID (Section 2.2.1.4.9) and software version (Section 2.2.1.4.1) to the TZ RPMB partition. Up to 40 antirollback version records of the TZ secure applications can be saved to the TZ RPMB partition.

The maximum TZ secure application version value is 0xFFFFFFFF.

7 MBA

NOTE: This chapter does not apply to the APQ8084 chipset.

The MBA is used to authenticate the primary modem image (PMI). In the past, the HLOS loaded and authenticated modem subsystem (MSS) images. With the SDM636/SDM660/SDM630/ SDM670 chipsets, the HLOS still loads the modem image, but the MBA performs the authentication. The modem now has its own secure root of trust (SRoT). This scheme offloads modem proprietary code from APPS and TZ, and because APPS and TZ do not have to load authenticated MSS images, APPS can run more seamlessly.

7.1 Load MBA image

As discussed in Section 2.5, the APPS resets the MSS. When the MSS comes out of reset, the Modem PBL is running and authenticates the MBA image.

The PIL only loads the MBA image into DDR; the Modem PBL then loads the MBA image from DDR into TCM.

On the HLOS/PIL side, the HLOS still authenticates the MBA image. However, the HLOS does not perform authentication itself, but polls the MBA RMB status register set by MBA. The modem PBL itself does the real authentication of MBA. See Figure 7-1.

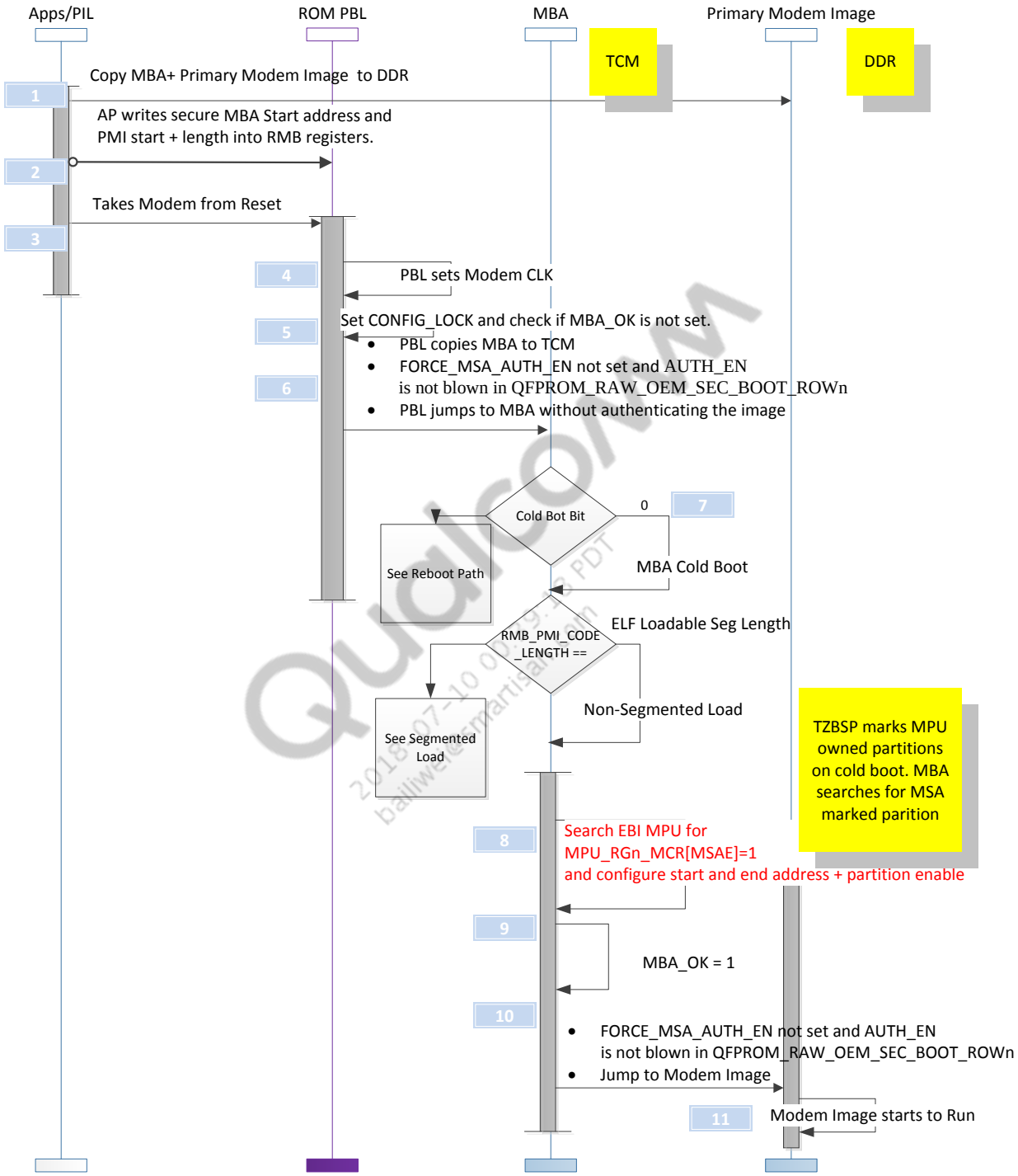


Figure 7-1 MBA authentication procedure

7.2 MSA QFPROM

MSA_ENA controls the MSA hardware block; QTI preblows the QFPROM .
See Figure 7-1 for details.

2. Secure boot library reads FORCE_MSA_AUTH_EN | AUTH_EN for the segment that is being authenticated (MBA or PMI).
1. Secure boot library reads PK_HASH_IN_FUSE.

   If SECURE_BOOTn is not blown, the default is 0 (use ROM).
2. Secure boot library reads ROM_PK_HASH_INDEX.

   If SECURE_BOOTn is not blown, the default is index 0, which correlates to the QTI root CA.
3. PBL/MBA authenticates the image.

   FORCE_MSA_AUTH_EN is allowed even if the OEM did not enable secure boot. MSS authentication can be performed, which is also one major benefit using MSA.

7.2.1 MSA JTAG debug

The OEM_CONFIG0 controls the OEM-related QFPROM:

```
OEM_CONFIG0_MSS_DBGEN_DISABLE
       OEM_CONFIG0_MSS_NIDEN_DISABLE
```

The OEM_CONFIG0 controls the overall debug disabling:

```
OEM_CONFIG0_ALL_DEBUG_DISABLE
```

If any of the preceding is blown, MBA JTAG is disabled.

NOTE: By default, the DEBUG OU fields do not override the JTAG debug settings.

8 Generate secure images

To sign the images for secure devices, QTI has introduced a standalone tool called Sectools. See Appendix A.6 for step by step guidance for signing images. See *Sectools: SecImage Tool User Guide* (80-NM248-1) for details and usage.

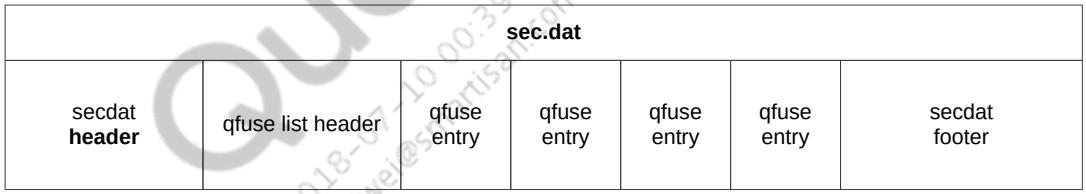
9 QFROM programming

QFPROM programming has an important role while enabling security features in the QTI MSM and MDM devices. With an increasing number of security features, such as antirollback, secure boot, multiple root certificates (MRC), secure file system, and JTAG disabling, the variety of fuse bits must be handled in a correct manner. To reduce the complexity of fuse programming, the fuse programming process has been fine-tuned for the APQ8084 chipset.

9.1 Fuse blowing process

The general process for blowing fuses is as follows:

1. The OEM uses Fuse_blower.py to generate a customized set of fuse values. Refer to *Sectools: FuseBlower Tool User Guide* (80-NM248-3) for more details on this tool. This tool generates a sec.dat binary file with the following format:



WARNING: Because the sec.dat partition is always used in the raw format, avoid using the sec.dat partition to blow fuses for sensitive data, for example, Customer Private Keys (CPKs). Use this method for general-purpose fuse blowing.

2. The generated sec.dat file is programmed to the “sec” partition in an EMMC/NAND device. The following command can be used to flash sec.dat into the “sec” partition:

   ```
   Fastboot flash sec <Path of sec.dat>
   ```

3. In the next reboot after flashing, XBL reads the “sec” partition from the device and loads it into the TZ DDR region.
4. At the end of XBL, TZ determines if fuse blowing has already occurred. This check is done by reading whether the OEM_SEC_BOOT bit of QFPROM_RAW_WR_PERM_LSB is blown.
5. If it is not blown, TZ parses the loaded sec.dat file from DDR, and programs the list of fuses one by one.
6. To enforce the programmed value, TZ reboots the device.
7. After reboot, all the intended fuse lists are programmed.

Figure 9-1 shows an overview of the fuse blowing process.

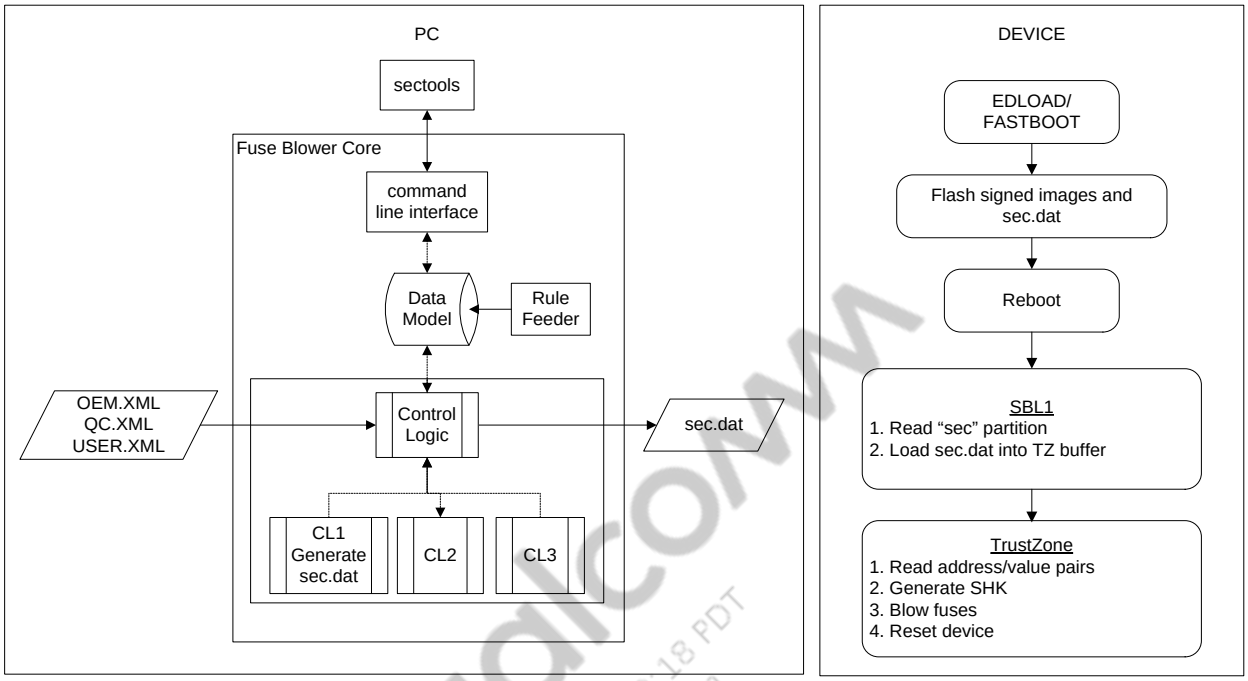


Figure 9-1 Fuse blowing process

10 Debug policy (DP)

NOTE: Numerous changes were made in this chapter.

To control the debug capability of the secure device, DP image is introduced in SDM636/SDM660/SDM630/SDM670. This feature allows control of the debug capability for a specific device. For further details, refer to *Sectools: Debug Policy Tool User Guide* (80-NM248-6) and *Debug Policy User Guide for MSM8996, MSM8976, MSM8956* (80-NV396-72).

The DP image enables debug capabilities such as JTAG re-enable, RAM dump, and TZ logging on commercial secure devices. For security reasons, the DP is tightly controlled by using the serial number of that device.

However, due to requests from multiple OEMs to skip the SN with DP image QTI introduced two stage sec.dat procedure along with XBL, DEVCFG and DP image changes. Enable OEMs to skip the serial number (SN) binding with DP images, the security implications, and OEM responsibilities/liabilities related to this change.

The fuse blow procedure must be divided into two stages for sec.dat.

10.1 Stage 1 sec.dat (without SNs binding)

The following are the changes to be added for stage1 sec.dat. ARB fuses and OEM_SEC_BOOT, ARB WR permission fuses entry to be removed.

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_fuseblower_QTI.xml
- <fuse ignore="false" n="2">
+ <fuse ignore="true" n="2">
<address>0x00780198</address>
<operation>BLOW</operation>
<field id="OEM_CONFIG_BIT_32-35">
<description></description>
<owner>QC</owner>
<value>0x0F</value>
<bits>35:32</bits>
</field>
.....................
<fuse_region id="QFPROM_RAW_WR_PERM">
<description></description>
<fuse ignore="false" n="0">
<address>0x00780158</address>
.............
- <field id="OEM_CONFIG">
- <description></description>
- <owner>QTI</owner>
- <value>0x1</value>
```

```
- <bits>10</bits>
-</field>
.................
- <field id="OEM_SEC_BOOT">
- <description></description>
- <owner>QTI</owner>
- <value>0x1</value>
- <bits>20</bits>
-</field>
...............
```

Then regenerate sec.dat and download.

The devices flashed with stage 1 sec.dat are termed as precommercial devices.

10.1.1 Debug setting for full dump (secure dump + nonsecure dump)

2. DP image:

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_debugpolicy.xml
<!--DP_ENABLE_NONSECURE_CRASH_DUMPS set to 1-->
<!--DP_ENABLE_ONLINE_CRASH_DUMPS set to 1-->
<!--DP_ENABLE_OFFLINE_CRASH_DUMPS set to 1-->
<!--DP_ENABLE_LOGS set to 1-->
<!--DP_ENABLE_JTAG set to 0 -->
```

3. b. OEM_enablecrashdumps_before_arb set to 1 in devcfg

```
trustzone_images/securemsm/trustzone/qsee/mink/oem/config/sdm660/oem_co
nfig.xml
    <!-- Mark third party signed secure applications sandboxed by
default-->
    <props name="OEM_enablecrashdumps_before_arb"
type=DALPROP_ATTR_TYPE_UINT32>
      0 1
  </props>
```

4. c. xbl.elf image to be signed with DEBUG_OU filed set to 0.

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_secimage.xml
   <debug>0x0000000000000000</debug>
```

5. d. Define SKIP_SERIAL_NUMBER_CHECK in XBL

```
boot_images\QcomPkg\Library\SecDbgLib\oem\secDbg_oem.c
      /* Serial Number Flag */
#define SKIP_SERIAL_NUMBER_CHECK 1
```

NOTE: For SDM630 use `<Meta build>\common\tools\sectools\config\sdm630\` CONFIG files.

For nonsecure dump `OEM_enablecrashdumps_before_arb` set to be 0.

10.2 Stage 2 sec.dat

Blow the antirollback fuse when OEMs finished all the test on the devices and prepare to ship the devices to market. Following is the changes

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_fuseblower_QTI.xml
- <fuse ignore="true" n="2">
+ <fuse ignore="false" n="2">
<address>0x00780198</address>
<operation>BLOW</operation>
<field id="OEM_CONFIG_BIT_32-35">
<description></description>
<owner>QC</owner>
<value>0x0F</value>
<bits>35:32</bits>
</field>
.....................
<fuse_region id="QFPROM_RAW_WR_PERM">
 <description></description>
<fuse ignore="false" n="0">
<address>0x00780158</address>
.............
+ <field id="OEM_CONFIG">
+ <description></description>
+ <owner>QTI</owner>
+ <value>0x1</value>
+ <bits>10</bits>
 +</field>
.........................
+<field id="OEM_SEC_BOOT">
+ <description></description>
+ <owner>QTI</owner>
+ <value>0x1</value>
+ <bits>20</bits> +</field>

.........................
```

Then regenerate sec.dat and download

The devices which blow the fuse with stage2 sec.dat are termed as Secure boot enabled commercial devices. In other words that is the devices, which enabled secure boot and antirollback fuse, is blown.

10.2.1 Debug setting for full dump (secure dump + nonsecure dump)

1. DP image:

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_debugpolicy.xml
<!--DP_ENABLE_NONSECURE_CRASH_DUMPS set to 1-->
<!--DP_ENABLE_ONLINE_CRASH_DUMPS set to 1-->
<!--DP_ENABLE_OFFLINE_CRASH_DUMPS set to 1-->
```

```
<!--DP_ENABLE_LOGS set to 1-->
<!--DP_ENABLE_JTAG set to 0 -->
```

2. OEM_enablecrashdumps_before_arb set to 1 in devcfg

```
trustzone_images/securemsm/trustzone/qsee/mink/oem/config/sdm660/oem_co
nfig.xml
    <!-- Mark third party signed secure applications sandboxed by default--
>
<props name="OEM_enablecrashdumps_before_arb"
type=DALPROP_ATTR_TYPE_UINT32>
      0 1
  </props>
```

3. xbl.elf image to be signed with DEBUG_OU filed set to 0.

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_secimage.xml
   <debug>0x0000000000000000</debug>
```

NOTE: For secure dump, add Serial number in DP image. For nonsecure dump, Serial number is not required in DP image.

The following is an example to add SN in DP image

```
<Meta build>\common\tools\sectools\config\sdm660\sdm660_debugpolicy.xml:
         <serial_num_list> <!--Range: 1 to 200-->
            <serial_num>00000000 9d925867</serial_num>
        </serial_num_list>
```

To add multiple serial numbers in debug policy, see FAQ.

A Guidance to enable secure boot

NOTE: Numerous changes were made in this appendix.

This chapter provides step-by-step instructions for enabling platform secure boot by blowing eFuses and instructions for signing images (boot loader and peripheral binaries) using Sectools and local signer, which excepts signing certificates and keys to be present on local signing machine. OEMs can extend these instructions based on their specific secure boot requirements.

This chapter provides step-by-step instructions only for following scenarios:

1. Minimum required eFuse configuration that can be used to enable platform secure boot, which includes enabling Image authentication, disabling JTAG access and blowing read-write permission fuses for SDM636/SDM660/SDM630/SDM670 chipset.
2. Local Signer that is, OEM signing key accessible from local signing machine. The instructions require signing certificates and keys to be accessible by machine on which Sectool is running and thus keys are not secured and are exposed during and after the signing on local machine on which Sectool is running. These instructions do not provide any guidance on securing the keys and it is out of scope of this chapter to provide any such guidance to be followed by an OEM. It is responsibility of the OEM to follow best security industry practices by obtaining keys/certificates from Commercial Authority (CA) and protect keys in HSM.
3. Self-generated keys and certificate using OpenSSL tool. These certificates do not chain to any certificate authority.
4. Refer to elsewhere in this document for advanced security configuration (like rollback protection, Image encryption, etc.) other than just minimum platform secure boot configuration.

The following figure shows the scenario covered in this chapter.

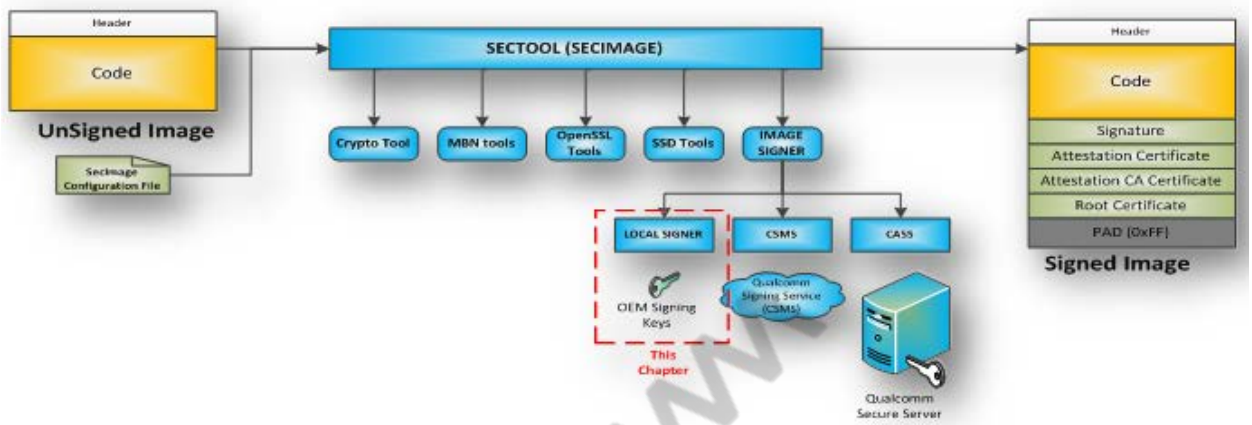


Figure A-1 Process to enable secure boot

Enabling secure boot for SDM636/SDM660/SDM630/SDM670 requires following steps at high level and covered in details in next sections:

1. Prerequisite for enabling secure boot.
2. Generate secure boot keys
3. Configuring ROT to OFPROM eFuses.
4. Signing of images for ROT configured to device.

A.1 Prerequisite to enable secure boot

Prerequisites for enabling secure boot are:

1. OpenSSL 0.9.80 Jun 2010 (or later version) – This tool is available at http://www.slproweb.com/products/Win32OpenSSL.html.
2. Obtain OEM_ID. OEMs to obtain unique OEM_ID from QTI, either by signing for CASS license or by opening Salesforce case and requesting OEM_ID.
3. Choose product ID.OEMs can select any 16-bit value as their product ID. Alternatively can also choose to keep it 0.
4. Make a new folder, for example, <sectools>\resources\data_prov_assets\Signing\Local\OEM-KEYS. Next section will capture instruction to generate secure boot keys and certificate in this folder.

A.2 Generate secure boot keys

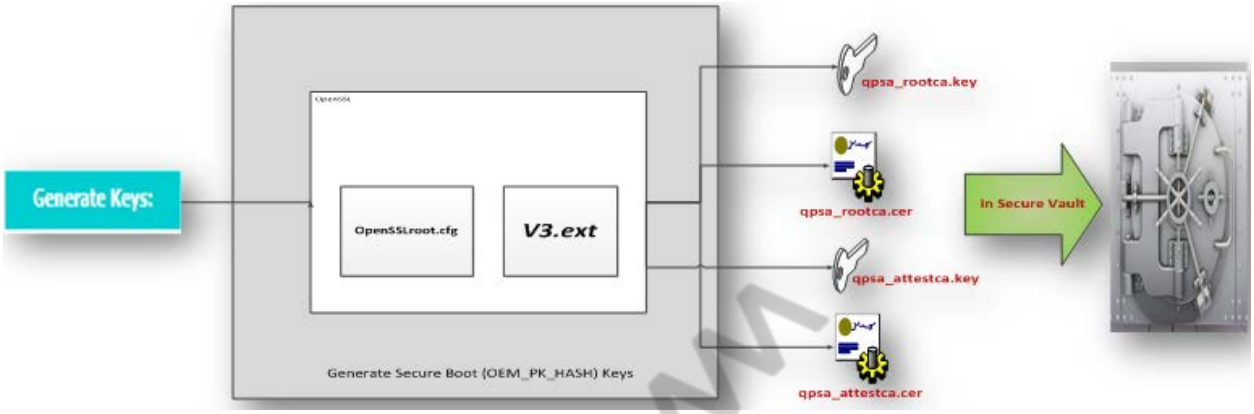


Figure A-2 Process to generate self-signed secure boot keys

Figure A-2 shows the process to generate self-signed secure boot keys and certificates, step-by-step instructions are as follows:

1. Open a new elevated command prompt and <sectools>\resources\data_prov_assets\Signing\Local\OEM-KEYS\.
2. Copy <sectools>\resources\data_prov_assets\General_Assets\Signing\openssl\opensslroot.cfg to <sectools>\resources\data_prov_assets\Signing\Local\OEM-KEYS\opensslroot.cfg.
3. Copy <sectools>\resources\data_prov_assets\General_Assets\Signing\openssl\v3.ext to <sectools>\resources\data_prov_assets\Signing\Local\OEM-KEYS\v3.ext.
4. Copy <sectools>\resources\data_prov_assets\General_Assets\Signing\openssl\v3_attest.ext to <sectools>\resources\data_prov_assets\Signing\Local\OEM-KEYS\v3_attest.ext.
5. Create <sectools>\resources\data_prov_assets\Signing\Local\OEM-KEYS\config.xml with following contents :

```
<METACONFIG>
  <is_mrc>False</is_mrc>

  <root_pre>True</root_pre>
  <attest_ca_pre>True</attest_ca_pre>
  <attest_pre>False</attest_pre>

  <root_cert>qpsa_rootca.cer</root_cert>
  <root_private_key>qpsa_rootca.key</root_private_key>

  <attest_ca_cert>qpsa_attestca.cer</attest_ca_cert>
  <attest_ca_private_key>qpsa_attestca.key</attest_ca_private_key>

  <attest_cert_params>
  C=US
  ST=California
  L=San Diego
  O=SecTools
  CN=SecTools Test User
  </attest_cert_params>
</METACONFIG>
```

A.2.1 Root key pair and certificate

Issue the following commands to generate the root public-private key pair and certificate:

- `OpenSSL genrsa -out qpsa_rootca.key 2048`
- `OpenSSL req -new -sha256 -key qpsa_rootca.key -x509 -out rootca_pem.crt -subj /C=US/ST=California/L="San Diego"/OU="General Use Test Key (for testing 13 only)"/OU="CDMA Technologies"/O=QUALCOMM/CN="QCT Root CA 1" -days 7300 -set_serial 1 -config opensslroot.cfg -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -sigopt digest:sha256`
- `OpenSSL x509 -in rootca_pem.crt -inform PEM -out qpsa_rootca.cer -outform DER`

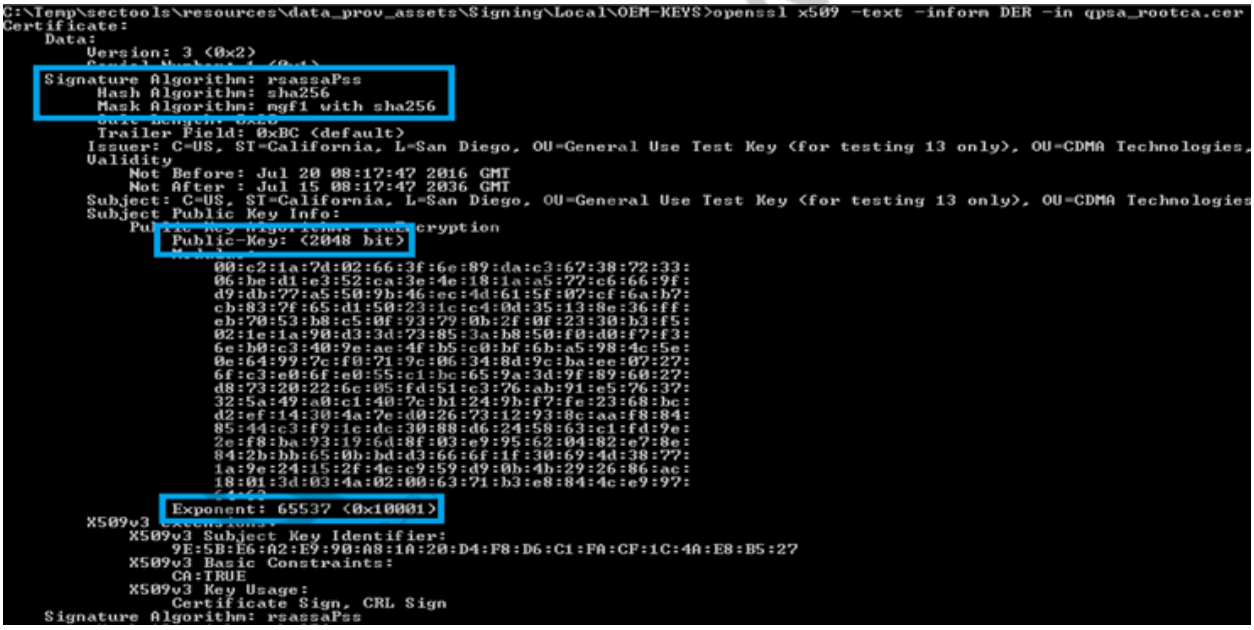


Figure A-3 Root key pair and certificate

NOTE: Ensure that the highlighted are set correctly.

A.2.2 Attestation CA key pair and certificate

Issue the following commands to generate Attestation CA key and certificate signed by the preceding generated root certificate:

- `OpenSSL genrsa -out qpsa_attestca.key 2048`
- `OpenSSL req -new -key qpsa_attestca.key -out attestca.csr -subj /C=US/ST=CA/L="San Diego"/OU="CDMA Technologies"/O=QUALCOMM/CN="QUALCOMM Attestation CA" -days 7300 -config opensslroot.cfg`

- `OpenSSL x509 -req -in attestca.csr -CA rootca_pem.crt -CAkey qpsa_rootca.key -out attestca_pem.crt -set_serial 5 -days 7300 -extfile v3.ext –sha256 -sigopt rsa_padding_mode:pss -sigopt rsa_pss_saltlen:-1 -sigopt digest:sha256`
- `OpenSSL x509 -inform PEM -in attestca_pem.crt -outform DER -out qpsa_attestca.cer`

A.3 Configure ROT to OFPROM eFuses

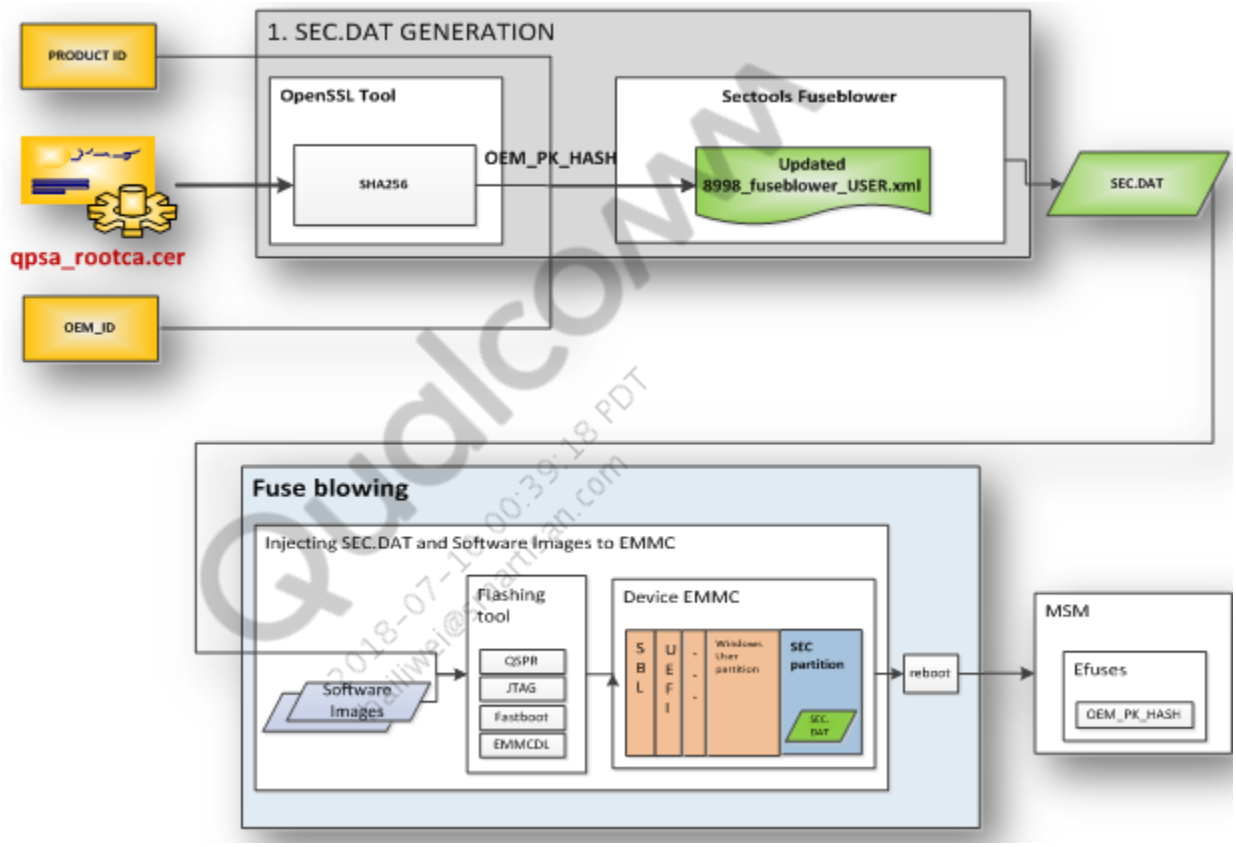


Figure A-4 Process to generate SEC.DAT

Figure A-4 shows the process to generate SEC.DAT and process to configure root-of-trust to QFPROM eFuses. Step-By-Step instructions are as follows:

1. Issue the following command to generate the SHA256 of root certificate:

   `OpenSSL dgst -sha256 qpsa_rootca.cer >sha256rootcert.txt`

2. Update <sectools>\config\sdm660\sdm660_fuseblower_USER.xml XML tags for following highlighted texts:

NOTE: For configuring fuseblower and secimage for SDM636, use configuration xmls of sdm660/sdm630. SDM636 has different SOC_HW_VERSION and JTAG ID. See FAQ 21 for the command to read JTAG ID of the device.

3. Rest of tags in file <sectools>\config\sdm660\sdm660_fuseblower_USER.xml default value requires no update.

4. Run Sectool with following command to generate SEC.DAT, which can be used to blow eFuses to device:

   python sectools.py fuseblower -p SDM660 -g –d

   Here,

   -d is for debug information that generates file with fuse information in file <sectools>\common_output\v2\debug\ secdat_repr.txt

   -p is platform or chipset and config files for platform are in <sectools>\config\

   -g to generate SEC.dat file

5. SEC.DAT generates on following path

   <sectools>\fuseblower_output\v2\sec.dat

6. Flash SEC.DAT to device along with other software images and reboot the device.

   Fastboot flash sec <sectools>\fuseblower_output\v2\sec.dat

7. On reboot, eFuses will be blown and root-of-trust is configured to device and device reboots again. After the last reboot secure boot is enabled on device and device can run only signed and trusted software.

A.4 OEM_PK_HASH

Root cert hash can be configured at two places. Perform either of the two configurations.

6. Copy the root cert hash generated in the file sha256rootcert.txt into the entry <value> </value>

   An example is given in the following code, this configuration is used when the OEM has its own mechanism/tools for generating root certificates without using sectools.

```
<module id="SECURITY_CONTROL_CORE">
        <entry ignore="false">
            <description>contains the OEM public key hash as set by
OEM</description>
            <name>root_cert_hash</name>

<value>02cce42866743c69a54dd4862f395036cdf45e2c752546d9f1d6de4a71199ad6<
/value>
        </entry>
        <entry ignore="true">
            <description>SHA256 signed root cert to generate root
hash</description>
            <name>root_cert_file</name>
            <value>./../../resources/testpki/qpsa_rootca.cer</value>
        </entry>
```

OR

7. Provide the root certificate file as an input (instead of the root cert hash) and the sectools will automatically generate the OEM PK HASH as highlighted:

```
<module id="SECURITY_CONTROL_CORE">
        <entry ignore="true">
            <description>contains the OEM public key hash as set by
OEM</description>
            <name>root_cert_hash</name>

<value>02cce42866743c69a54dd4862f395036cdf45e2c752546d9f1d6de4a71199a
d6</value>
        </entry>
        <entry ignore="false">
            <description>SHA256 signed root cert to generate root
hash</description>
            <name>root_cert_file</name>

<value>>./../../resources/data_prov_assets/Signing/Local/OEM-
KEYS/qpsa_rootca.cer </value>
        </entry>
```

A.5 PK HASH IN FUSE

A.5.1 For SEC_BOOT1

```
<entry ignore="false">
   <description>PK Hash is in Fuse for SEC_BOOT1: Apps</description>
   <name>SEC_BOOT1_PK_Hash_in_Fuse</name>
  -<value>false</value>
  +<value>true</value>
</entry>
```

A.5.2 For SEC_BOOT2

```
<entry ignore="false">
   <description>PK Hash is in Fuse for SEC_BOOT2: MBA</description>
   <name>SEC_BOOT2_PK_Hash_in_Fuse</name>
  -<value>false</value>
  +<value>true</value>
</entry>
```

A.5.3 For SEC_BOOT3

```
<entry ignore="false">
   <description>PK Hash is in Fuse for SEC_BOOT3: MPSS</description>
   <name>SEC_BOOT3_PK_Hash_in_Fuse</name>
  -<value>false</value>
  +<value>true</value>
</entry>
```

A.5.4 OEM HW ID

```
<entry ignore="false">
    <description>The OEM hardware ID</description>
    <name>oem_hw_id</name>
    <value>0x0000</value> →to be replaced with OEM ID value
 </entry>
```

A.5.5 OEM Product ID

```
<entry ignore="false">
    <description>The OEM product ID</description>
    <name>oem_product_id</name>
    <value>0x0000</value> →to be replaced with OEM Product ID value
 </entry>
```

Rest of tags in file <sectools>\config\<Chipset>\<Chipset>_fuseblower_USER.xml, no update is required for default values

A.6 Signing of images for ROT on device

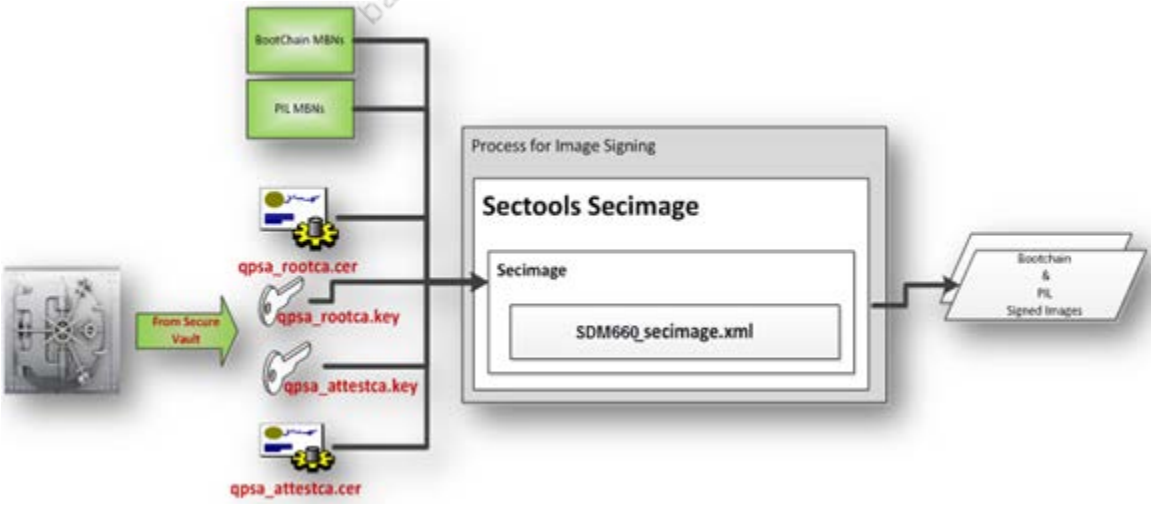


Figure A-5 Signing of images for ROT

Figure A-5 shows the process for signing images for root-of-trust configured to device. The following step-by-step instructions show how to sign one image and OEMs can follow similar steps for rest of images.

1. Configuration of <chipset>_secimage.xml
   a. MSM_PART or MSM_HW_ID or JTAG ID is a chipset specific ID, see to FAQ 21 to read the JTAG ID from the device. Note that SDM660, SDA660, and SDM636 have different JTAG IDs, but all SDM660 devices have unique JTAG ID

      <msm_part>0x000910E1</msm_part>

   b. SOC_HW_VERSION, see FAQ 21 to read SOC_HW_VERSION from the device. Note that SDM660, SDA660, and SDM636 have different SOC_HW_VERSIONs but all SDM660 devices have unique SOC_HW_VERSION

      <soc_hw_version>0x60040100</soc_hw_version>

   c. IN_USE_SOC_HW_VERSION. Keep this 1 to use SOC_HW_VERSION for signing, Keep this 0 to use MSM_PART for signing
      <in_use_soc_hw_version>1</in_use_soc_hw_version>

   d. Use Serial number in signing. Serial number is unique to the device. This setting can be used to create different signed images for different secure boot enabled devices. Image signed with one serial number works only on the device with same serial number.

      The Serial number will be used for signature verification instead of OEM ID and MODEL ID when this is set to 1

      <use_serial_number_in_signing>0</use_serial_number_in_signing>

   e. OEM_ID_INDEPENDENT: Reserved for QTI signing. Set to 0 for customer signing
      <oem_id_independent>0</oem_id_independent>

   f. OEM_ID or OEM HW ID: Refer to 11.3.2.4 If a non zero OEM ID is blown to fuses, configure the following oem_id with same value. Refer to FAQ to request unique OEM ID from Qualcomm.

      <oem_id>0x0000</oem_id>

   g. MODEL_ID or OEM Product ID: Refer to 11.3.2.5, if a non zero model id is blown to fuses, configure the following model ID with same value. MODEL ID is OEM specific.
      <model_id>0x0000</model_id>

   h. DEBUG OU: 0x02 for Secure boot Debug Disable

      <debug>0x0000000000000002</debug>

   i. SERIAL NUMBER: This is used when use_serial_number_in_signing is set to 1, populate with chip serial number: ex: 0x12345678.
      <serial_number>0x12345678</serial_number>

2. List all images that need signing for SDM636/SDM660/SDM630/SDM670 platform by using following command:

   ```
   python sectools.py secimage -h --chipset SDM660  --sign_id LIST
   ```

   ```
   Sign-ids available for chipset "SDM660":
   1. xbl
   2. pmic
   3. abl
   4. prog_emmc_ufs_lite
   5. prog_emmc_ufs_ddr
   6. vip
   7. uefi
   8. mba
   9. modem
   10. TZ
   11. hyp
   12. devcfg
   13. rpm
   14. aDSP
   15. STI
   16. venus
   17. efs_tar
   18. sampleapp32
   19. sampleapp64
   20. isdbtmm
   21. Widevine
   22. cppf
   23. PlayReady
   24. cmnlib
   25. cmnlib64
   26. keymaster
   27. hdcp1
   28. hdcp2p2
   29. dxhdcp2
   30. lksecapp
   31. macchiato_sample
   32. gfx_microcode
   33. mcfg_hw
   34. mcfg_sw
   35. tz_tee
   36. cdsp
   ```

3. Copy tz.mbn to folder as <sectools>\input\tz.mbn.
4. Sign tz.mbn using following command

   ```
   python sectools.py secimage -s -i "input\tz.mbn" -g TZ -p SDM660 --
   cfg_selected_cert_config=OEM-KEYS
   ```

Where,

-s: sign the image.

-i: path to the image file to sign.

-g: sign id corresponding to the image_file provided.

-p: id of the chipset corresponding to the image_file.

--cfg_selected_cert_config: cert config selection file.

Signed tz.mbn will be generated at following path:

```
<sectools>\secimage_output\sdm660\tz\tz.mbn
```

5. OEMs can follow the similar steps for signing other images individually.
6. OEM can use below command to sign the all the images at once using Meta build as input

```
python sectools.py secimage -m “rootpath_till_contents.xml in meta” -s -
c config\sdm660\sdm660_secimage.xml -o .\ --cfg_selected_cert_config=
OEM-KEYS
```

B FAQs

NOTE: Numerous changes were made in this appendix.

QUESTION 1 Can I use nonsecure software on a secure board?

ANSWER No. Always use signed images to boot up. Otherwise, you enter download mode, as PBL fails to boot up because of an authentication failure.

QUESTION 2 Does secure boot test require blowing JTAG disable fuses?

ANSWER Secure boot test can be done without disabling JTAG as part of testing Secure boot in several stages. But for a device to be called fully secure, JTAG should be disabled. But Hence end to end secure boot validation requires blowing JTAG disable fuses also.

QUESTION 3 How do I get CASS access?

ANSWER For any CASS licensees, there is a QTI administrator. Raise a case and it will be assigned to the CASS administrator for further processing. To get a license for the first time might require 30 to 40 days because of the required legal and document processing.

QUESTION 4 Do I need to care about software IDs while generating certificates?

ANSWER Yes. Each software ID is associated with a specific set of images. Any mismatch leads to an authentication failure on fused boards.

QUESTION 5 Can I use a signed image on a nonsecure board?

ANSWER Yes. You can use the signed software images on a nonsecure board without any expected problems.

QUESTION 6 Does the secure boot feature increase the bootup time?

ANSWER Yes. There is a tradeoff between security and bootup time. The extra authentication process takes longer than a nonsecure boot.

QUESTION 7 What is the size difference between signed and unsigned images?

ANSWER Signed and unsigned images differ by 6 KB to 7 KB. Double signing adds another 6KB to 7KB but it is applicable to only few images like tz.mbn

QUESTION 8 Should I resign the software every time I compile?

ANSWER It is always advisable to resign the software component when you recompile it.

QUESTION 9 Are the three certificates for different software images the same?

ANSWER The Root and Attestation CA Certificates for different software are the same. It is the Attestation Certificate that always varies for different software. The Attestation Certificate contains information such as SW_ID, HW_ID, OEM_ID, SW_SIZE, and SHA types.

QUESTION 10 Do I use SHA1 or SHA256 certificates?

ANSWER SHA256 is recommended for SDM636/SDM660/SDM630/SDM670

QUESTION 11 Do I check the validity of certificates against wall clock date and time?

ANSWER No. The validity dates in the certificates are required fields but are not checked against the real time as there is no possibility of tracking the calendar day and time during the bootup process. So there are no expiry dates for your certificates.

QUESTION 12 Is it mandatory to use CASS for code signing?

ANSWER It is advisable, but not mandatory, to use CASS for code signing. It is as secure as any other code signing provider available on the market. QTI also provides support for any technical issues and getting access to it.

QUESTION 13 I do not see the target that I need to use for signing.

ANSWER It might be because you are not licensed for that specific MSM device. Contact the CASS administrator for help. If you have a license for the MSM device, the administrator provides you with the access for that specific MSM device for code signing.

QUESTION 14 Can I do secure boot test without blowing any fuses?

ANSWER It is recommended blowing fuses for secure boot as described in the appropriate QFPROM Programming Reference Spreadsheet for your chipset (see Appendix 0).

QUESTION 15 When do I blow the FEC bit while blowing the QFPROM?

ANSWER The FEC bit must only be enabled after all configuration bits are blown. After the FEC bit is blown for a specific row, no other bit of that row can be changed.

QUESTION 16 Can we use other chipset certificates and keys (Ex: MSM8976/8953) for SDM636/SDM660/SDM630/SDM670?

ANSWER No. The signature algorithm for SDM636/SDM660/SDM630/SDM670 is different compare to 8976 other chipsets. The authentication fails if other chipset certificates and keys are used.

QUESTION 17 List of all images to be signed for secure boot.

ANSWER Refer sdm660_secimage.xml file for the required images to be signed.

Run python sectools.py secimage -h --chipset SDM660 --sign_id LIST command.

Sign xbl.elf, pmic.elf, abl.elf, rpm.mbn, devcfg.mbn, hyp.mbn, tz.mbn, mba.mbn, sqdsp6sw.mbn, adsp.mbn, cmnlib.mbn, cmnlib64.mbn, keymaster.mbn images for basic bootup.

QUESTION 18 Why all the listed images in content.xml file are not signed as given in sdm660_secimage.xml?

ANSWER The content.xml file contains all the image information – HLOS and NON HLOS.

The QTI secure boot feature support NON HLOS images and in NON HLOS images all the images are not required to sign for secure boot feature.

QUESTION 19 We find common/sectools/config/sdm630/sdm630_fuseblower_USER.xml & common/sectools/config/sdm660/sdm660_fuseblower_USER.xml are only different on <chipset>. Is the same sec.dat workable on both SDM660/SDM630?

ANSWER Only if OEM HW ID and hardware Product ID are same for both, same sec.dat can be used. However same secimage.xml cannot be used for signing as SDM660 and SDM630 differ in JTAG ID and SOC_HW version.

QUESTION 20 What SPARE fuses available for OEM usage?

ANSWER Refer to feature selection tab – Customer SPARE fuses in the Document 'QFPROM Programming Reference Guide' for respective chipset. Refer to section "Related Documents " for the DCN.

QUESTION 21 How to dump SERIAL NUMBER, JTAG ID , SOC_HW_VERSION, and OEM ID?

ANSWER adb commands for register dumps.

	SDM660	**SDM630**	**SDM636**	**SDM670**
SERIAL NUMBER	adb shell /system/bin/r 0x00784138	adb shell /system/bin/r 0x00784138	adb shell /system/bin/r 0x00784138	adb shell /system/bin/r 0x00786134
JTAG ID	adb shell /system/bin/r 0x00786130	adb shell /system/bin/r 0x00786130	adb shell /system/bin/r 0x00786130	adb shell /system/bin/r 0x00786130
SOC_HW_VERSION	adb shell /system/bin/r 0x01FC8000	adb shell /system/bin/r 0x01FC8000	adb shell /system/bin/r 0x01FC8000	adb shell /system/bin/r 0x01FC8000
OEM ID	adb shell /system/bin/r x00786138	adb shell /system/bin/r 0x00786138	adb shell /system/bin/r 0x00786138	adb shell /system/bin/r 0x00786138
OEM_SEC_BOOT	adb shell /system/bin/r 0x00780350	adb shell /system/bin/r 0x00780350	adb shell /system/bin/r 0x00780350	adb shell /system/bin/r 0x00780350

QUESTION 22 How to add multiple serial numbers in debug policy?

ANSWER Maximum of 200 serial numbers can be added in DP.
<Metabuild>\common\tools\sectools\config\sdm660\sdm660_debugpolicy.xml:

```
<serial_num_list> <!--Range: 1 to 200-->
   <serial_num>9d925867</serial_num>
          <serial_num>8cc614d8</serial_num>
          <serial_num>bd34fb8a</serial_num>
          <serial_num>f85c0531</serial_num>
   ------------
   </serial_num_list>
```

QUESTION 23 How to solve emergency download fail using QFIL on secure boot enabled devices?

ANSWER Make following modification, rebuilt prog_emmc_firehose_xxxx_lite.mbn, and signed it, then use it for download. It could download success.

File path:Bootimages/QcomPkg/Library/DeviceProgrammerCommonLib/firehose/deviceprogrammer_initialize.c

Modification:

```
static void deviceprogrammer_init_hw()
{
<snip>
     fh.validation_enabled = FALSE;
#ifndef SKIP_SECBOOT_CHECK_NOT_RECOMMENDED_BY_QUALCOMM
    // This check the following to ensure that only VIP programmer is
run on secure boot devices
    // ISigning the non-VIP programmer is highly not recommended
    if( isSecureBootEnabled()==TRUE )
    {
        // To be here means that the secure boot fuses are blown,
therefore must use VIP
        fh.validation_enabled = TRUE;
    }
#endif
+     fh.validation_enabled = FALSE;
<snip>
}
```

QUESTION 24 How to know if secure boot is enabled on the device at run time?

ANSWER Read the register QFPROM_RAW_OEM_SEC_BOOT_ROW0_LSB using Adb command adb shell /system/bin/r 0x00780350. It should contain the value 0x303030.

QUESTION 25 What is OEM ID and how to request it?

ANSWER OEM ID is a unique ID provided to OEM by QTI. OEMs to obtain unique OEM_ID from QTI, either by signing for CASS license or by opening Salesforce case and requesting OEM_ID.

QUESTION 26 How to make the device enter EDL mode (9008 mode)?

ANSWER 1. Erase SBL partition to make the device enter EDL mode

2. Pull up the FORCE_USB_BOOT GPIO_[57] high to make the device fall into EDL mode. Once flashing is done, the GPIO should pulled low.

QUESTION 27 How to sign the non-HLOS.bin?

ANSWER Perform signing of all the split binary images and replace the split binaries with the signed binaries in the meta path common\build\bin\asic\pil_split_bins. Re-generate non-HLOS.bin by running build.py in meta folder path common\build.

QUESTION 28 What is the difference between QFPROM_RAW_ and QFPROM_CORR_ mappings of a register?

ANSWER QFPROM_RAW_* and QFPROM_CORR_* are just separate address mappings for the same QFPROM space.

QFPROM_RAW_* must be used for writing to the QFPROM (blowing fuses)
QFPROM_RAW_* can be read. It will return the fuse value that is in the QFPROM.

QFPROM_CORR_* can not be written. It is read only.
The "CORR" in QFPROM_CORR* stands for "corrected".
QFPROM_CORR_* is the same as the QFPROM_RAW_* value except that FEC (Forward Error Correction) can be applied in order to correct any fuses failures (to handle a situation when the fuses go bad after time).

QUESTION 29 Is there a way to bind serial number to a image for signing?

ANSWER Yes serial number can be bound for image signing using the following setting:

```
<use_serial_number_in_signing>1</use_serial_number_in_signing>
<serial_number>0x12345678</serial_number>
```

Serial number is part of HW_ID instead of OEM_ID and MODEL_ID as follows:

```
| HW_ID                          | 0x3006000012345678
```

QUESTION 30 What RSA Key lengths are supported for signing images of SDM660/630/670/636 chipsets?

ANSWER RSA2048, RSA4096 are supported.

QUESTION 30 How are SKDK fuses blown?

ANSWER SKDK are blown automatically while enabling secure boot. Once a valid sec.dat (with SKDK fuse entry) is flashed into the device, TZ blows SKDK fuses with a random number and disables further read and write of SKDK fuses. Irrespective of the SKDK fuse values configured in sec.dat, the SKDK is blown to a random number by TZ.

C Compute OEM_PK_HASH

other_src/compute_oem_pk_hash.c

```
#include <stdio.h>
#include <inttypes.h>
struct OEM_PK_HASH_VALUE {
uint32_t fec_value:8;
uint32_t hash_data1: 24;
uint32_t hash_data0;
};
int compute_oem_pk_hash(struct OEM_PK_HASH_VALUE* out, uint8_t* bytestream,
size_t size)
{
int i, j;
uint64_t t;
for (i = 0; i < size / 7; i++) {
t = 0;
for (j = 0; j < 7; j ++) {
t |= (uint64_t)((bytestream[i*7 + j])&0xff) << ((j) * 8);
}
out[i].hash_data1 = t >> 32;
out[i].hash_data0 = t & 0xFFFFFFFF;
}
t = 0;
i = size / 7;
if (i * 7 < size) {
for (j = 0; j < 4; j++) {
t |= (uint64_t)((bytestream[i*7 + j]) & 0xFF) << (j * 8);
}
out[i].hash_data1 = t >> 32;
out[i].hash_data0 = t & 0xFFFFFFFF;
++i;
}
return i;
}
```

D References

D.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Sahara Protocol Specification	80-N1008-1
SDM660 QFPROM Programming Reference Guide	80-P7747-97
SDM630 QFPROM Programming Reference Guide	80-P7865-97
SDM636 QFPROM Programming Reference Guide	80-PD860-97
SDM660 Security Overview	80-P8754-22
Sectools: SecImage Tool User Guide	80-NM248-1
Sectools: FuseBlower Tool User Guide	80-NM248-3
Sectools: KeyProvision Tool User Guide	80-NM248-5
Sectools: Debug Policy Tool User Guide	80-NM248-6
QSEE Version 4.0 User Guide	80-NH537-2
Firmware Over The Air (FOTA) Workflow	80-NL409-1
Debug Policy User Guide for MSM8996, MSM8976, MSM8956	80-NV396-72
Transitioning to Code Signing Management System 2.0	80-V8000-4
Installing the CASS Credential Software 8.3 User Guide	80-N7185-3
Installing the Linux CASS Credential Software 8.3 User Guide	80-N7185-4
SDM670 QFPROM Programming Reference Guide	80-PB873-97
Standards	
ITU-T standard for public key infrastructure (PKI) and Privilege Management Infrastructure (PMI)	ITU-T X.509 v3
Resources	
OpenSSL Cryptography and SSL/TLS Toolkit, req(1)	http://openssl.org/docs/apps/req.html
OpenSSL Cryptography and SSL/TLS Toolkit, x509(1)	http://openssl.org/docs/apps/x509.html

D.2 Acronyms and terms

Term	Definition
AArch64	ARM architecture, 64-bit
Aarch32	ARM architecture, 32-bit
APPS	Applications processor
APPS PBL	Applications primary boot loader
APPSBL	Applications boot loader

Term	Definition
CA	Certificate authority
CSMS	Code signing management system
DDR	Double data rate
dm-verity	Device-mapper-verity
DN	Distinguished name
ELF	Executable and linking format
EL	(for example, EL3)
eMMC	Embedded multimedia card
FEC	Forward error correction
FLCB	Fast low current boot
HLOS	High-level operating system
HYP	Hypervisor
HMAC	Hashed message authentication code (RFC 2104)
HS-USB	High-speed universal serial bus
JTAG	Joint test action group
KDF	Key derivation function
LK	Little kernel-based Android boot loader
MBA	Modem boot authentication
MPSS	Modem peripheral subsystem software
MRC	Multiple root certificates
MSA	Modem self-authentication
MSS	Modem subsystem
OCIMEM	On-chip internal memory
OU	Organizational unit name
PBL	Primary boot loader
PHK	Primary hardware key
PIL	Peripheral image loading
PMI	Primary modem image
QHCK	Qualcomm hardware certification kit
QFPROM	Qualcomm fuse programmable read-only memory
QGIC	Qualcomm generic interrupt controller
QPSA	Qualcomm platform signing application (QPSA)
QTEE	Qualcomm Trusted Execution Environment (formerly known as Qualcomm Secure Execution Environment)
RMB	Relay message buffer
RMR	Reset management register
ROT	Root of trust
RPM	Resource power manager
RPMB	Replay protected memory block
RPMFW	Resource power manager firmware
RSASSA-PSS	RSA signature scheme with appendix - probabilistic signature scheme
SBL	Secondary boot loader

Term	Definition
SCM	Secure channel manager
SDCC	Secure digital card controller
SDI	System debug image
SDRAM	Synchronous dynamic random access memory
SFS	Secure file system
SHK	Secondary hardware key
SMC	Secure monitor call
SPS	Smart peripheral subsystem
SRoT	Secure root of trust
TCM	Tightly coupled memory
TZBSP	TZ board support package
TZExec	TrustZone executive
UEFI	Unified extensible firmware interface
VMIDMT	Virtual machine ID mapping table
WCN	Wireless communication network
XBL	eXtensible boot loader