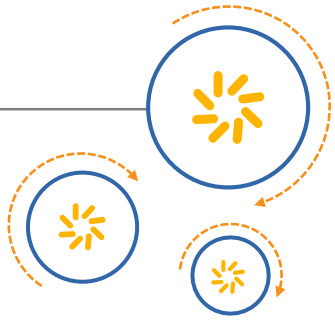
Qualcomm Technologies, Inc.



Enabling Secure Boot

80-NL239-45 D

September 7, 2015

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

Revision history

Revision	Date	Description
A	June 2014	Initial release
B	February 2015	Added ▪ Information related to MSM8936, MSM8939 and MSM8909 chipsets ▪ Sections 2.7, 9.6.2, and 9.6.3 Updated ▪ Sections 2.3.1.1 and 2.3.1.2 for 64 bit elf format ▪ Section 3.4 for OEM_HW_ID ▪ Sections 2.6.1, 9.1 and 9.5.1
C	June 2015	Added ▪ Information related to APQ8016,APQ8039, MSM8929, and MSM8952 chipsets ▪ Section 2.8 for MSM8952 secure boot flow ▪ Chapter 10 ▪ Updated Section 2.6.1
D	September 2015	Updated the following to include MSM8956/MSM8976 information: ▪ Chapters 4, 9, and 10 ▪ Sections 1.1, 2.6.2, 2.8, and 5.4 ▪ Appendix D.1

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document describes the process to enable secure boot in MSM8916, MSM8936, MSM8939 MSM8909, APQ8016, APQ8039, MSM8929, MSM8952, and MSM8956/MSM8976 chipsets.

1.2 Scope

This document is intended for developers to understand the detailed steps regarding eFuse blowing and image signing procedures.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Code variables appear in angle brackets, e.g., `<number>`.

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

Button and key names appear in bold font, e.g., 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.

1.4 Technical assistance

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

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

2 Secure boot overview

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 only authorized software can be 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 boot up process. This process asserts the authenticity of all secure software images that are executed by the device. 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; 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 of the raw code that they want to use, and adds this to the device alongside the software binary. The device contains the corresponding public key of the vendor, which can be used to verify that the binary has not been modified and that it was provided by the trusted vendor in question.

Images (the format in which the code is packaged) can be signed using QTI's CSMS (refer Code Signing Management System User Guide (80-V8000-1)), or using the licensee's own code signing system. Signed images include the code signature and the certificate chain. The certificates carry the public keys used to decrypt the certificate and image signatures. Secure boot supports 2048 bit exponent 3 or 65537 public RSA keys for the certificate and image signatures. The certificate signatures support the PKCS v1.2 standard format for SHA1 or SHA256, while the code signature follows a proprietary algorithm that is described in this document.

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 boot up 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 (e.g., Primary Boot Loader (PBL) → Secondary Boot Loader (SBL) → ARM® TrustZone). The root of trust (the most trusted entity that kicks off this process) is the PBL, which is firmware and therefore already trusted. Before the next image in the boot up sequence is executed, that image is first authenticated to ensure that it contains authorized software. For example, control passes to the SBL only after it has been successfully authenticated by the PBL. Since the SBL is now trusted, it can be 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 boot up software and secure applications is validated.

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

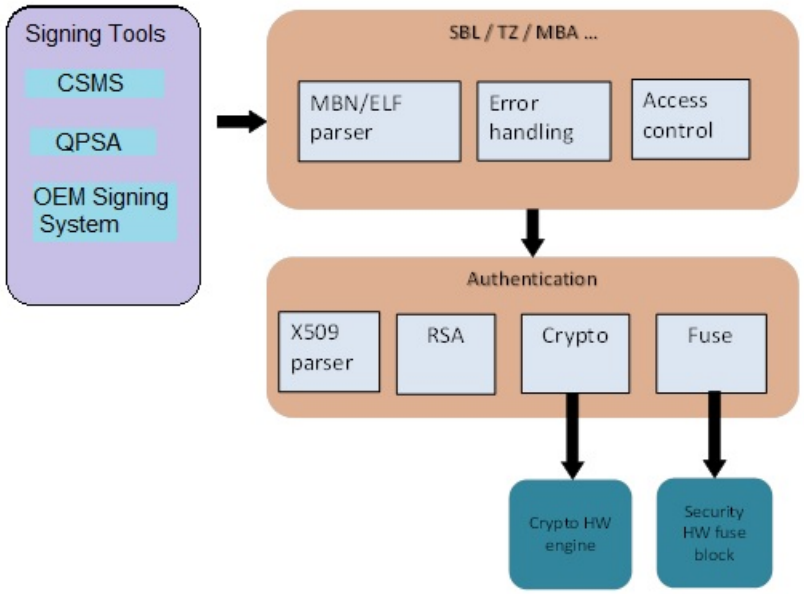


Figure 2-1 Secure boot architecture diagram

NOTE: CSMS is not available to customers automatically upon the set up of a Salesforce or Docs and Downloads account. Additional agreements and authorizations may be required to access CSMS. Contact your company's sales account representative for more information.

2.1 Certificate and signature format

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

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

2.2 Certificate profiles

A *chained* approach is used to authenticate an image. First, the certificate chain of the image is validated. for example, the SBL authenticates the Applications Boot Loader (APPSBL) image using the APPSBL's certificate chain. 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 QTI's key table residing in a trusted area.

Once 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).

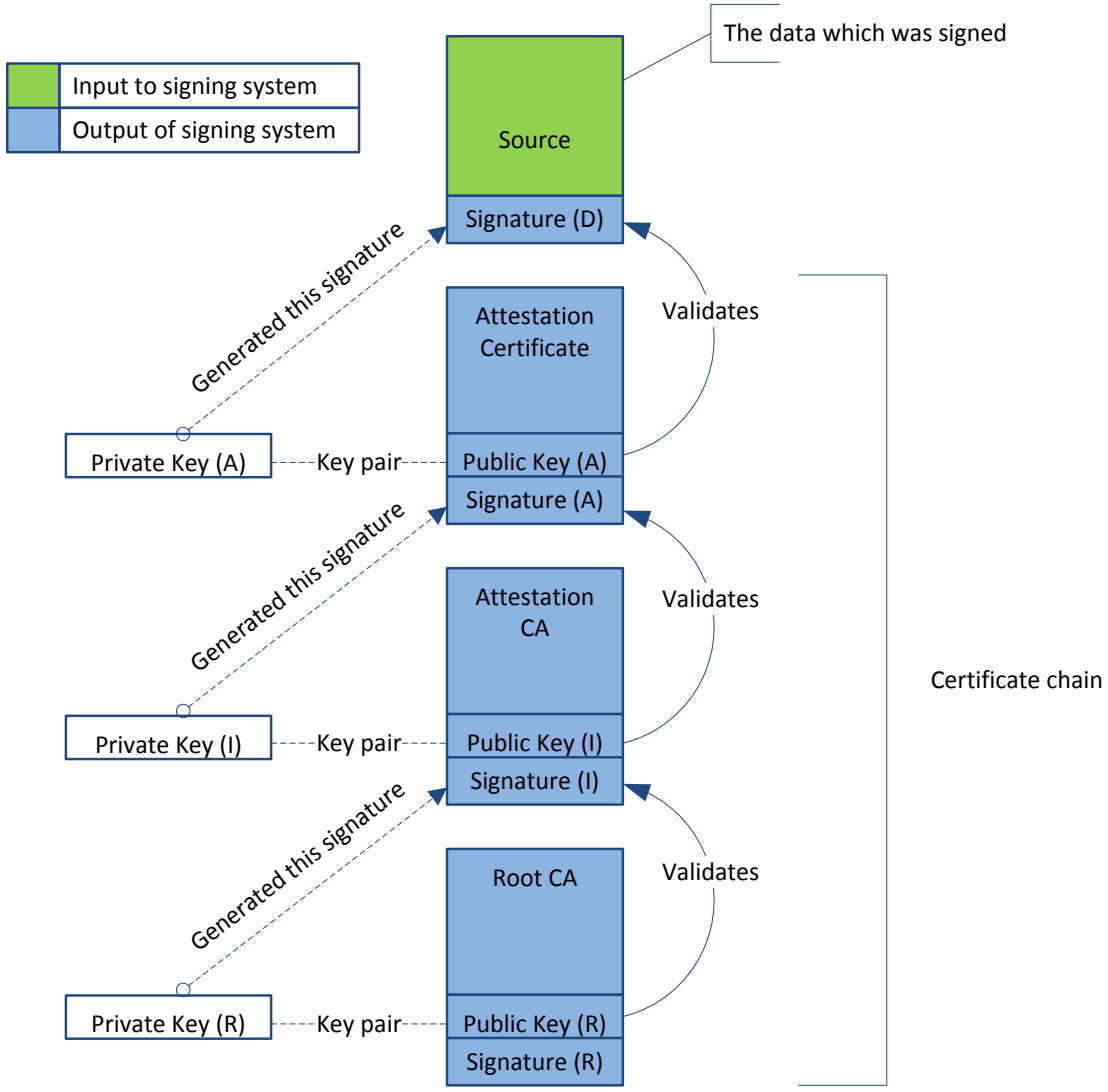


Figure 2-2 Certificate chain and signatures

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.

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 2-1 shows the generic profile root certificate from QTI's CSMS signing service. OEMs can follow similar profile for the OEM root certificate.

Table 2-1 Root certificate profile

Field	Settings
Subject DN Common name Organization Org Unit Locality State Country	 QCT Root CA 1 Qualcomm CDMA Technologies San Diego CA US
Version	V3
Serial number	VeriSign specified
Signature algorithm	SHA1 RSA
Issuer	Same as Subject DN
Valid from	Date/time issued
Valid to	20 years from issued date
Public key size	2048, with 3 as exponent
Basic constraints	Subject type=CA Path length= None
Key usage	Certificate signing Off-line 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 2-2 shows the generic profile attestation CA certificate from QTI's CSMS signing service. OEMs can follow similar profile for the OEM attestation CA certificate.

Table 2-2 Attestation CA certificate profile

Field	Settings
Subject DN Common name Organization Org Unit Locality State Country	 Qualcomm Attestation CA Qualcomm CDMA Technologies San Diego CA US
Version	V3
Serial number	VeriSign specified
Signature algorithm	SHA1 RSA
Issuer	(Root Subject DN)
Valid from	Date/time issued
Valid to	20 years from issued date
Public key size	2048, with 3 as exponent
Basic constraints	Subject type=CA Path length=0
Key usage	Certificate signing Off-line 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 2-3 shows the generic profile attestation certificate from QTI's CSMS signing service. OEMs can follow similar profile for the OEM attestation certificates.

Table 2-3 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 Locality State Country	 Qualcomm Developer S.M Company Name 01 <SW ID 16 hex> SW_ID 02 <MSM SW ID 16 hex> HW_ID 03 <debug 16 hex> DEBUG 04 <OEM ID 4 hex> OEM_ID 05 <image size 8 hex> SW_SIZE 06 <model ID 4 hex> MODEL_ID 07 <algo 4 hex> SHA1 or SHA256 08 <APP ID 16 hex> APP_ID 09 <CRASH DUMP 16 hex> CRASH_DUMP San Diego CA US
Version	V3
Serial number	VeriSign specified
Signature algorithm	SHA1 RSA
Issuer	Same as Subject DN
Valid from	Date/time issued
Valid to	20 years from issued date
Public key size	2048, with 3 as exponent
Basic constraints	Subject type=End Entity Path length=0
Key usage	Certificate signing Off-line 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 0000000000000002 DEBUG
OU = 04 0001 OEM_ID
OU = 05 00002000 SW_SIZE
OU = 06 0002 MODEL_ID
OU = 07 0001 SHA256
OU = 08 8916AAAA00000001 APP_ID
OU = 09 0000000000000001 CRASH_DUMP
```

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 has not been 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	32 bits
Software version	Software type

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

If the software version in the certificate is older than the version in the anti-rollback fuses, authentication fails. This ensures that there is no rollback to an older buggy image and the fuses blown to contain the fixed version number. For example:

1. An OEM signs TrustZone with version 0.
2. Anti-rollback fuses for TrustZone are blown to 0.
3. The OEM signs the fixed TrustZone image with version 1, and blows the anti-rollback TrustZone fuses to 1.

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

The software type specifies the signed image (SBL, TrustZone, etc.) and is used to ensure that the boot-up sequence cannot be changed.

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

2.2.1.4.2 HW_ID field

This field contains the 64-bit MSM_HW_ID value used in the HMAC to sign the image (see Section 5.1). It comprises one of two cases, A or B.

Case A:

32 bits	16 bits	16 bits
MSM ID (JTAG ID)	OEM hardware ID CSMS account ID for the OEM, or 0 if the OEM does not use CSMS and does not want to use an identifier for signing/authentication.	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:

32 bits	32 bits
MSM ID (JTAG ID)	SERIAL NUM

In both cases, 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 (i.e., the MSM8916 chipset) along with the variant information, and is blown (provisioned) by QTI.

In case A, the lower 32 bits comprise the OEM identifier (the value that identifies the OEM and the OEM's phone model). In case B, the lower 32 bits comprise the chip-specific unique serial number.

This 64-bit HW_ID effectively binds the image to the specified hardware so it can only pass authentication (and execute) on the specified hardware.

2.2.1.4.3 DEBUG field

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

This field is only acted upon as follows:

- By the Applications PBL (APPS PBL) based on the Debug OU field of the SBL image's attestation certificate
- By the Modem PBL based on the Debug OU field of the Modem Boot Authentication (MBA) attestation certificate

The field comprises:

32 bits	32 bits
SERIAL NUM	DEBUG SETTINGS

The lower 32 bits denote the action to be taken by the PBL in writing to the one-time writable OVERRIDE_2 and 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 settings value:

- 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 chip's serial number 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 will fail.

2.2.1.4.4 OEM_ID field

This field is for information purposes only and denotes the OEM_HW_ID value in the MSM_ID OU field (see Table 5-1).

NOTE: The code itself can be signed with an OEM ID (part of the MSM ID value used in the HMAC of the code signature) and the code signature can be valid. But if the debug certificate is not meant for that specific chip, authentication fails.

2.2.1.4.5 SW_SIZE field

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

2.2.1.4.6 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 Table 5-1).

2.2.1.4.7 SHA1/SHA256 field

This field denotes the hash algorithm (SHA-1 or SHA-256; QTI strongly recommends SHA-256) used to generate the code signature.

2.2.1.4.8 APP_ID field

This field is required for a TrustZone application, e.g., when SW_ID is TZ_EXEC_HASH_TABLE = 0xC.

NOTE: The value must be 16 hex.

This field is used when anti-rollback fuses for TrustZone are blown to 1. The TrustZone secure application anti-rollback tracks the TrustZone application's software version by APP_ID. All zeroes is an invalid input when the TrustZone anti-rollback fuse is blown, and loading the TrustZone application will fail.

This field is also used by the TrustZone SFS to generate a unique secret key for that TrustZone application's SFS encryption. NULL or zero value APP_ID TrustZone applications share the same SFS secret key for SFS encryption.

It is the signer's (OEM) responsibility to keep the APP_ID list and ensure that different TrustZone applications do not share the same APP_ID.

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 software to software. For MSM8916, QTI software supports ELF binaries only.

2.3.1 ELF format

ELF format binaries contain multiple segments. Various types of information (i.e., source address, destination address, type, etc.) about each segment are kept as part of the program headers.

In QTI software components, sbl.mbn, rpm.mbn, tz.mbn,hyp.mbn, wcnss.mbn and, qdsp6sw.mbn fall under this category. 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.

NOTE: The mba.mbn image is a special case; for more detail, see Section 7.1.

2.3.1.1 ELF header

Each ELF file begins with a header defined as follows:

```
typedef struct {
unsigned char e_ident[ELFINFO_MAGIC_SIZE]; /* 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;

/* 64bit ELF header */
typedef struct
{
  unsigned char e_ident[ELFINFO_MAGIC_SIZE]; /* Magic number and other info
*/
  uint16 e_type;                    /* Object file type             */
  uint16 e_machine;                 /* Architecture                 */
  uint32 e_version;                 /* Object file version          */
  uint64 e_entry;                   /* Entry point virtual address   */
  uint64 e_phoff;                   /* Program header table file offset*/
  uint64 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 */
} Elf64_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 must be 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

Each ELF segment is defined by a structure.

```
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;

/* 64 bit Program segment header.  */
typedef struct
{
  uint32 p_type;                /* Segment type */
  uint32 p_flags;               /* Segment flags */
  uint64 p_offset;              /* Segment file offset */
  uint64 p_vaddr;               /* Segment virtual address */
  uint64 p_paddr;               /* Segment physical address */
  uint64 p_filesz;              /* Segment size in file */
  uint64 p_memsz;               * Segment size in memory */
  uint64 p_align;               /* Segment alignment */
} Elf64_Phdr;
```

Following are some of the values for the p_type element:

- PT_NULL – The array element is unused; other members 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 segment's memory size (p_memsz) is larger than the file size (p_filesz), the extra bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may 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 must be loaded into memory.

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

Table 2-4 provides information on what the bits in the p_flags field correspond to (as described in the QTI miprogressive.h header file).

Table 2-4 ELF p_flags with Qualcomm extension

Bits	Description
0:19	Standard segment flag Execute (0x1), Write (0x2), Read (0x4), Read Execute (0x5), etc.
20	Segment mode
21-23	Access type: ▪ Non-paged segment (0x0) that must be loaded to RAM before booting the respective processor. These segments are hashed and must be validated. ▪ Paged segment (0x1). Demand paged segments; obsoleted by the MSM8916 chipset.
24-26	Segment type L4 segment (0x0), AMSS segment (0x1), Hash segment (0x2), Boot segment (0x3), etc.
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 segments must be 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 will execute 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 must be validated before boot up. Also, before booting up a processor, all MI_PBT_NON_PAGED_SEGMENT segments must be 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 SHA1 digest (D_n) for each loadable section, so the result is n ($n \in [1, m]$)

SHA1 digests, where each SHA1 digest is 20 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 Qualcomm 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 above.

ELFs that are executed via TrustZone Peripheral Image Loading (PIL) contain the hash table segment whether the secure boot feature is enabled or not. This allows non-secure boot builds to have a basic validation check of each segment (i.e., TrustZone PIL hashes a segment and compares it with the hash contained in the hash table segment to see if it matches.

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

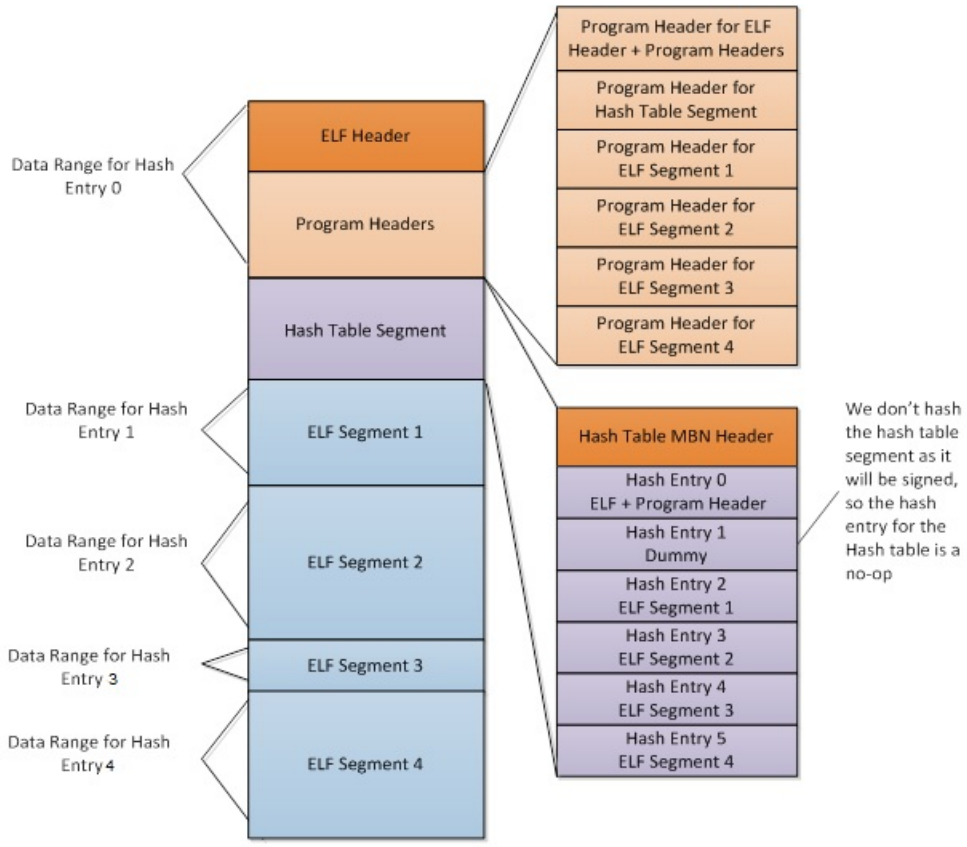


Figure 2-3 Overview of unsigned ELF image

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 illustrates a signed ELF image.

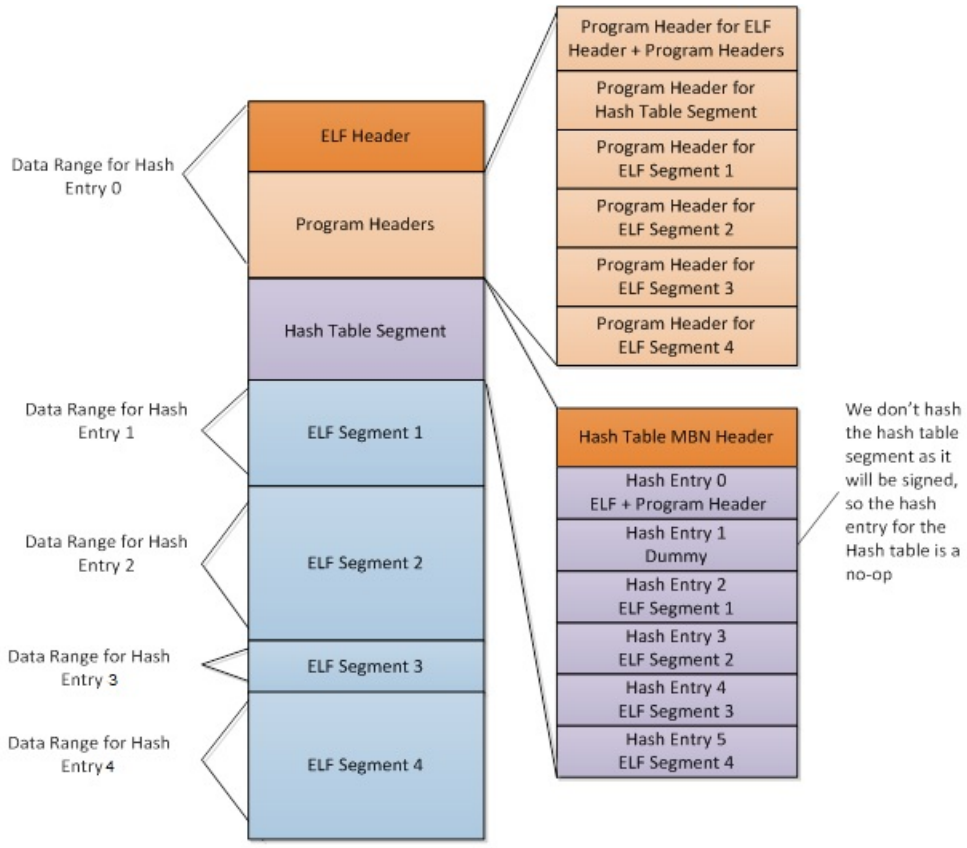


Figure 2-4 Signed multiple segments binary format

When verifying the signed ELF, secure boot verifies:

- Whether the certificate chain and signature (in the hash segment) is valid.
- Whether $D_1 \ldots D_n$ matches the SHA1 digest of each loadable ELF section (excluding the hash segment itself).

2.3.2 Image signature

This signature is calculated over the hash table segment. The hash table segment contains an entry for the hash of the ELF header plus the program header. The signature validation of the hash table segment is performed, and then the hash of the ELF header plus program header is calculated and authenticated through the entry in the hash table segment.

The ELF header contains information about the location of the program header table, the number of program header entries, and where each entry corresponds to a segment in the image. The program header entries contain security critical information (such as paged vs. non-paged), and it is important to protect them from manipulation. The hash table segment contains the hashes of each segment, so it is security-critical to protect the hash segment from manipulation.

NOTE: Legacy chipsets used a similar ELF format for modem images, with one difference: the ELF header and program header were not authenticated. The hash table segment was signed, and the hash header provided pointers to the authentication information, the signature, and the certificate.

However, with the new format, the ELF header plus program header is authenticated because the hash is stored in the hash table segment.

The address of the signature and the certificate for image authentication are provided by the hash table segment. The signature is a fixed length of 256 bytes, and the certificate chain is 6144 bytes (with 0xFF padding).

2.4 Signing hash and verification

The generated signature is decrypted with a public key stored in the boot ROM or QFPROM, to be compared with the local calculated hash digest. If they are the same, the image authentication is considered to be passed; otherwise, it is considered to be a failure.

Figure 2-5 shows the image signature generation.

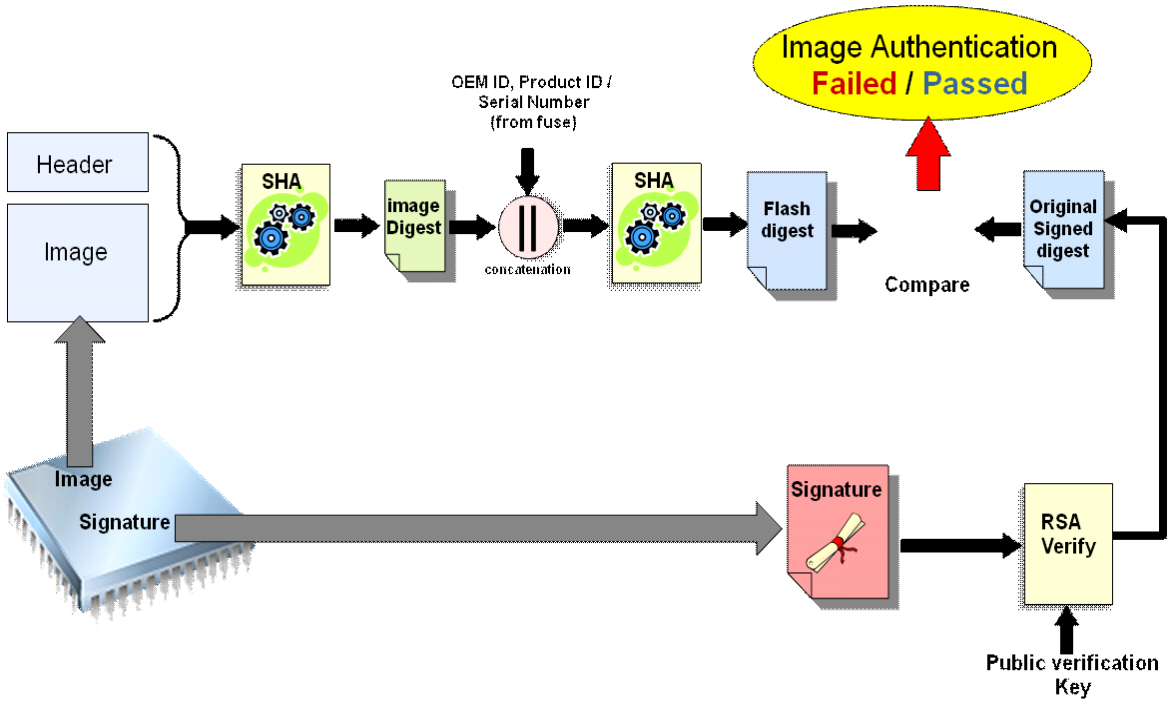


Figure 2-5 Image signature generation

For QTI digital signature hashing, slightly more than just a hash of the image is encrypted. The QTI digital signature involves HMAC.

- Image hash with SW_ID
- Image hash with MSM_HW_ID (see Section 5.1) or SERIAL_NUM
- Image hash is encrypted with a private key in a code signing system

Figure 2-6 illustrates the QTI SHA1 hash generation.

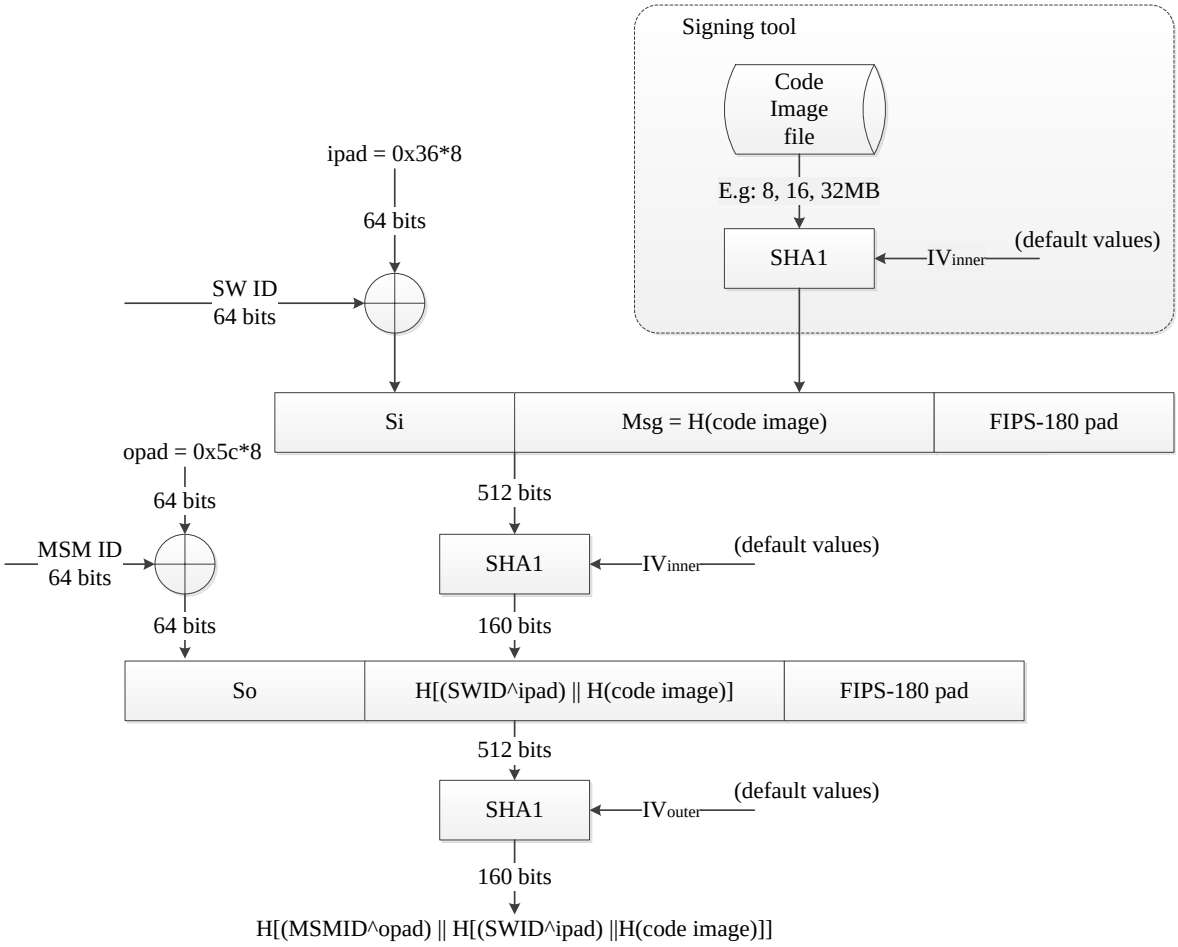


Figure 2-6 QTI hash generation SHA1

NOTE: SHA256 is the recommended algorithm because the performance is higher in comparison to SHA1. The output is 160/256 bits for SHA256.

2.5 Image authentication components

In the MSM8916 chipset, secure information related to secure boot is contained in the QFPROM, boot ROM, and SDRAM. Figure 2-7 illustrates the image authentication components

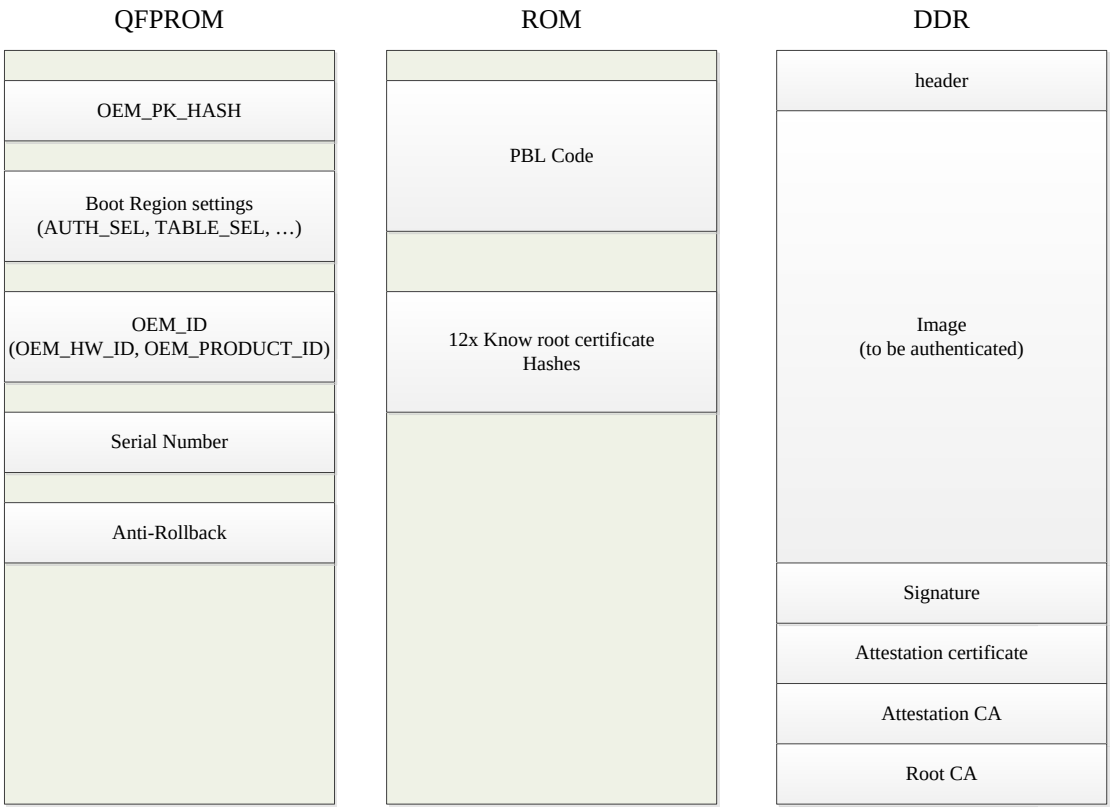


Figure 2-7 Image authentication components

2.6 MSM8916/MSM8936/MSM8939/MSM8929/APQ8016/ APQ8039 secure boot

The MSM8916/MSM8936/MSM8939/MSM8929/APQ8016/APQ8039 chipset has the following changes compared to MSM8974, MSM8x26, and MSM8x10 chipsets.

Some of the major changes.

- APPS processor is the primary boot processor since RPM PBL is removed. RPM PBL is removed because FLCB is not required after USB Engineering Change Notice (ECN), which allows 500 mA up to 2 min of USB connect and later 100 mA.
- PBL supports ELF type image load and authentication, which allows SBL1 and HEXProgrammer to retain ELF format like other system images.
- SDI functionality is implemented within SBL and TZ, unlike in MSM8974/MSM8x26, where it was a different image loaded by SBL.
- AArch64-bit mode is supported in APPS processor.

There are different processors in the MSM8916 chipset. Table 2-5 lists the processor types and boot addresses.

Table 2-5 Boot address for processors

Subsystem	Processor	Boot address	
APPS	Cortex-A53	0xFC010000*	
RPM	Cortex-M3	0x00200000(Subsystem view)	0x0(System view)
Modem	MSS_QDSP6	Configurable*	
Pronto	ARM9™	0x0 or 0xFFFF0000 or hardware remap*	

* No change in boot address in system and subsystem views.

2.6.1 MSM8916/MSM8936/MSM8939/MSM8929/APQ8016/APQ8039 secure boot flowchart

Secure boot in MSM8916/MSM8936/MSM8939/MSM8929/APQ8016/APQ8039 chipsets is different from the secure boot in MSM8974, MSM8x26, and MSM8x10 chipsets. There are two boot ROMs compared to the previous generation APPS PBL and Modem PBL.

Figure 2-8 illustrates the secure boot flowchart for the MSM8916/MSM8936/MSM8939 chipset.

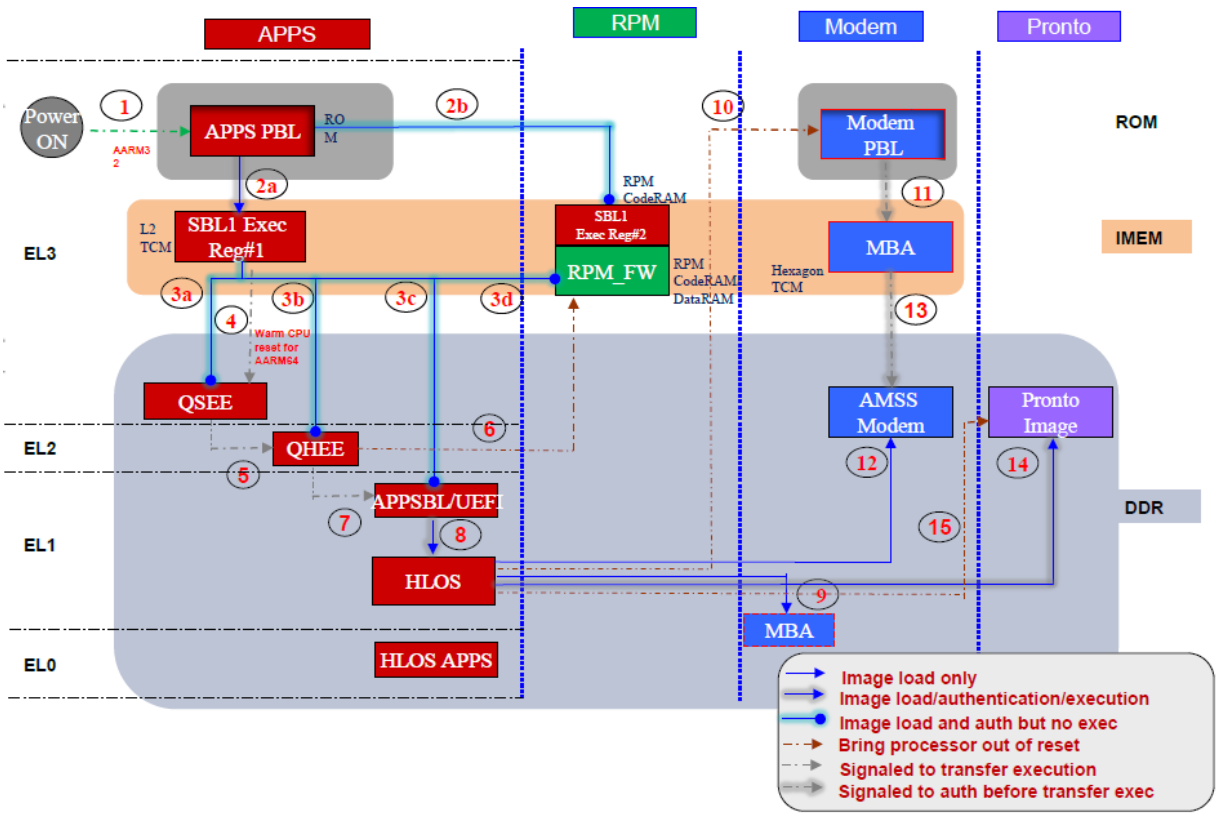


Figure 2-8 MSM8916/MSM8936/MSM8939/MSM8929/APQ8016/APQ8039 secure boot flowchart

NOTE: APQ8016, APQ8039 supports only GPS functionality.

Refer to Figure 2-8:

1. The system powers on and takes MSM8916 AP CPU out of reset.
2. In Cortex-A53 APPS PBL executes the following:
   a. Loads and authenticates the SBL1 segment #1 from the boot device to L2 (as TCM)
   b. Loads and authenticates SBL1 segment #2 (SDI equivalent) to RPM code RAM, and then jumps to SBL1
3. SBL1#1 initializes DDR

a. Loads and authenticates the QSEE/TZ image from the boot device to DDR

b. Loads and authenticates the QHEE image from the boot device to DDR

c. Loads and authenticates the RPM firmware image from the boot device to RPM code RAM

d. Loads and authenticates the HLOS APPSBL image from the boot device to DDR

4. SBL1 transfers execution to QSEE/TZ. QSEE/TZ sets up a secure environment, configures xPU, and supports the fuse driver.
5. SBL1 runs in AArch32 mode. If AArch64 mode switch is required, then SBL1 sets boot remapper for QSEE entry and writes to RMR register and then triggers warm-reset. Now, QSEE starts in AArch64 mode.
6. QSEE transfers execution to QHEE. QHEE* is responsible for VMM setup, SMMU configurations, and xPU access control.
7. QHEE notifies RPM to start the RPM firmware execution.
8. QHEE transfers execution to HLOS APPSBL to initialize the system.

   a. The Linux APPS boot loader (HLOS APPSBL) starts the execution in AArch32 mode-only.

   b. This is done by QHEE by looking at the ELF header for HLOS APPSBL, which indicates that it uses 32-bit instruction set architecture. QHEE changes to 32-bit mode and starts Linux APPS boot loader (HLOS APPSBL) execution in 32-bit mode.

9. HLOS APPSBL loads and authenticates the HLOS kernel.
10. The Linux APPS boot loader (HLOS APPSBL) will indicate about the HLOS kernel AArch64 mode by making an SCM call to secure the monitor before exiting. LK does not jump into kernel directly as it used to do previously.
11. HLOS kernel loads the Modem Boot Authenticator (MBA) to DDR via PIL.
12. HLOS kernel brings Modem DSP Hexagon out of reset.
13. Modem PBL then continues its boot process.
14. HLOS kernel loads the AMSS modem image to DDR via PIL
15. Modem PBL authenticates MBA and then jumps to it.
16. HLOS loads the Pronto image to DDR via PIL.
17. HLOS brings the Pronto image out of reset so that the Pronto image can starts execution.

2.6.2 Boot ROMs

The MSM8916/MSM8936/MSM8939/MSM8929/APQ8016/APQ8039/MSM8909/MSM8952/MSM8956/MSM8976 chipsets have two boot ROMs – APPS PBL, and Modem PBL. Boot ROMs are read-only, so there is no authentication for them.

2.6.3 PBL error handling

APPS PBL – Saves the error information in RPM code RAM @ 0x00205400.

- When a PBL error occurs, it goes to the error handler.
- APPSPBL logs the necessary log information into the RPM code RAM.
- APPSPBL goes into Emergency Download mode if it is not disabled by the fuse.
- APPSPBL tries SDC Port-2 recovery first
- If the recovery fails, the PBL goes into HS-USB Emergency Download mode.
- In Emergency Download mode, the PBL enters the Sahara protocol to receive and authenticate the Flash programmer from the host.
- When the flash programmer is loaded and authentications are passed, the system jumps to the flash programmer's start address.
- The Flash programmer executes and downloads the necessary boot images from the host side.

Modem PBL – Saves the error information in Modem TCM memory @ 0x044034e0 and it also Log error code and reason in RMB registers. Table 2-7 lists the RMB registers.

In OEM_CONFIG0 → PBL_LOG_DISABLE is used to save the error information on both APPS and Modem side.

2.7 MSM8909 secure boot

There is a major boot difference between MSM8909 and MSM8916/MSM8936/MSM8939 chipsets

The Boot_config [0..4] GPIOs or BOOT_CONFIG fuses can be used to select the following boot options. Once the fuse is blown, the GPIOs used for the boot option are free to be used as common GPIOs.

BOOT_CONFIG	MSM8909	Comments
0x00	BOOT_DEFAULT_OPTION	BOOT_DEFAULT_OPTION
0x01	BOOT_SDC_PORT2_THEN_SDC_ PORT1_OPTION	BOOT_SDC_PORT2_THEN_SDC_PORT1_ OPTION
0x02	BOOT_eMMC_OPTION	BOOT_eMMC_OPTION
0x03	BOOT_USB_OPTION	BOOT_USB_OPTION
0x04	BOOT_NAND_OPTION	BOOT_NAND_OPTION

OEMs must select default boot device as eMMC boot or NAND boot based on FAST BOOT fuses blown. The eMMC GPIO interface is MUXed with NAND GPIOs for bootable device (both are mutual exclusive).

0x00 is always the default boot option for all earlier MSM™ chipsets, but in MSM8909 chipset eMMC and NAND boot are mutually exclusive.

For details on MSM8909 boot, refer to *MSM8909 Boot Architecture Overview* (80-NR964-3).

2.7.1 MSM8909 secure boot flowchart

Figure 2-9 shows the secure boot flowchart for the MSM8909 chipset.

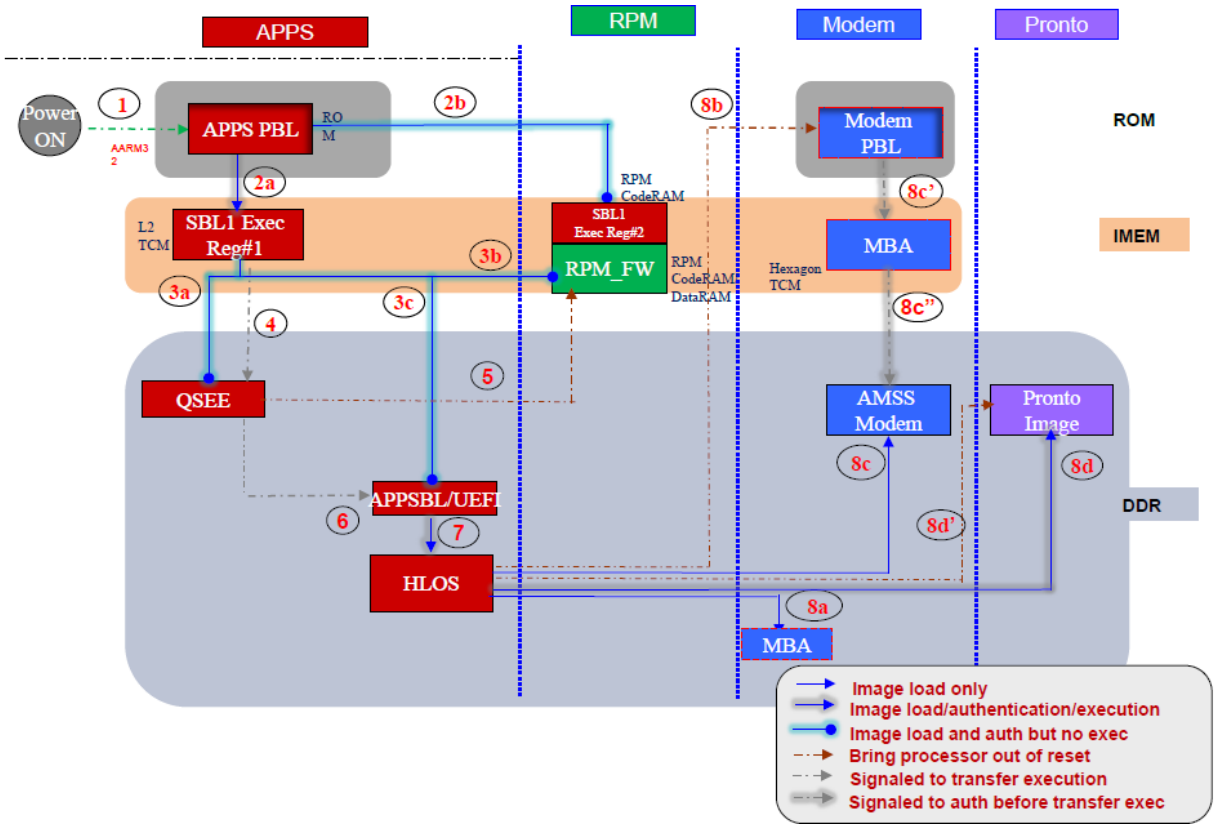


Figure 2-9 MSM8909 secure boot flowchart

Refer to Figure 2-9:

1. The system powers on and takes MSM8909 AP CPU out of reset.
2. Cortex-A7 APPS PBL:
   a. Loads, executes, and authenticates the SBL1 segment #1 from the boot device to L2 (as TCM).
   b. Loads, executes, and authenticates SBL1 segment #2* (DDR/SDI equivalent) to RPM code RAM, then jumps to SBL1.
3. SBL1#1
   a. Loads and authenticates the QSEE/TZ image from the boot device to DDR.
   b. Loads and authenticates the RPM firmware image from the boot device to RPM code RAM.
   c. Loads and authenticates the HLOS APPSBL image from the boot device to DDR.
4. SBL1 #1 transfers execution to QSEE/TZ.

5. QSEE/TZ set up secure environment and bring RPM out of RESET to start execution of RPM firmware.
6. QSEE/TZ jumps to HLOS APPSBL to start execution.
   - *SBL1 segment#2 is equal to DDR driver + SDI equivalent copied to RPM code RAM.
   - DDR is initialized by SBL1 segment#2 and part of the SDI functionality included in SBL1 segment#2. For details, refer to *MSM8909 Boot Architecture Overview* (80-NR964-3).
7. HLOS APPSBL loads and authenticates the HLOS kernel.
8. HLOS kernel:
   a. Loads the Modem Boot Authenticator (MBA) to DDR via PIL.
   b. Brings modem DSP Q6 out of reset.
   c. Loads the AMSS modem image to DDR via PIL.
   c'. Modem PBL copies the MBA from DDR to modem TCM and authenticates MBA and jump to MBA image.
   c". MBA authenticates modem image and then jumps to modem.
   d. HLOS loads the Pronto image to DDR via PIL.
   d'. HLOS brings Pronto out of reset and Pronto image starts execution.

2.8 MSM8952/MSM8956/MSM8976 secure boot flowchart

NOTE: Numerous changes were made in this section.

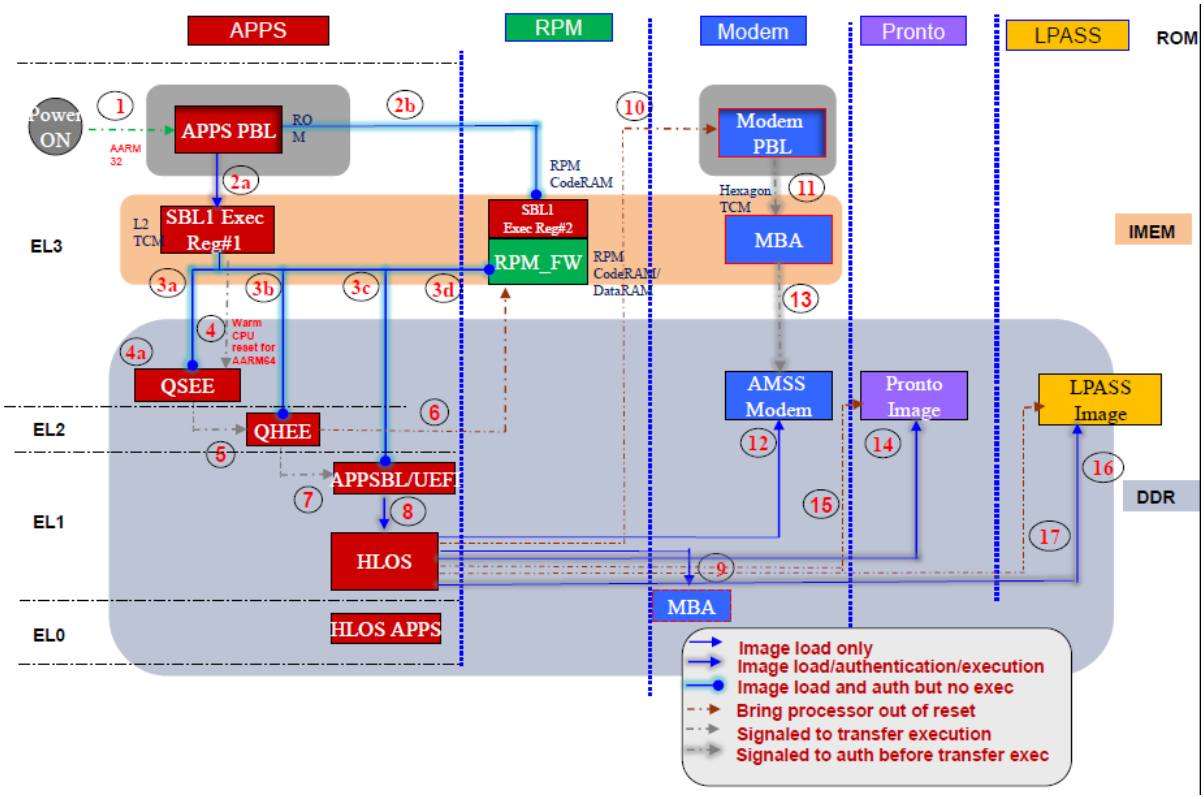


Figure 2-10 MSM8952/MSM8956/MSM8976 secure boot flowchart

Refer to Figure 2-10:

1. The system powers on and takes MSM8952/MSM8956/MSM8976 AP CPU out of reset.
2. In Cortex-A53, APPS PBL executes the following:
   a. Loads and authenticates the SBL1 segment #1 from the boot device to L2 (as TCM)
   b. Loads and authenticates SBL1 segment #2 (SDI equivalent) to RPM code RAM, and then jumps to SBL1
3. SBL1#1 initializes DDR:
   a. Loads and authenticates the QSEE/TZ image from the boot device to DDR
   b. Loads and authenticates the QHEE image from the boot device to DDR
   c. Loads and authenticates the RPM firmware image from the boot device to RPM code RAM
   d. Loads and authenticates the HLOS APPSBL image from the boot device to DDR
4. SBL1 transfers execution to QSEE/TZ. QSEE/TZ sets up a secure environment, configures xPU, and supports the fuse driver.

   a. SBL1 runs in AArch32 mode. QSEE/TZ runs in AArc64 mode. For AArch64 mode switch SBL1 sets boot re-mapper for QSEE entry and writes to RMR register and then triggers warm-reset. Now, QSEE starts in AArch64 mode.

5. QSEE transfers execution to QHEE. QHEE* is responsible for VMM setup, SMMU configurations, and xPU access control.
6. QHEE notifies RPM to start the RPM firmware execution.
7. QHEE transfers execution to HLOS APPSBL to initialize the system.
   a. The Linux APPS boot loader (HLOS APPSBL) starts the execution in AArch32 mode only.
   b. QHEE performs the execution by looking at the ELF header for HLOS APPSBL, which indicates that it uses 32-bit instruction set architecture. QHEE changes to 32-bit mode and starts Linux APPS boot loader execution in 32-bit mode.
8. HLOS APPSBL loads and authenticates the HLOS kernel. The Linux APPS boot loader (HLOS APPSBL) indicates about the HLOS kernel AArch64 mode by making an SCM call to secure the monitor before exiting. LK does not jump into kernel directly as it used to do previously.
9. HLOS kernel loads the Modem Boot Authenticator (MBA) to DDR via PIL.
10. HLOS kernel brings Modem DSP Hexagon out of reset.
11. Modem PBL then continues its boot process.
12. HLOS kernel loads the AMSS modem image to DDR via PIL
13. Modem PBL authenticates MBA and then jumps to it.
14. HLOS loads the Pronto image to DDR via PIL.
15. HLOS brings the Pronto image out of reset so that the Pronto image starts executing.
16. HLOS loads the LPASS image to DDR via PIL.
17. HLOS brings the LPASS image out of reset so that the LPASS image starts executing.

3 QFPROM configuration

QFPROM configuration enables the security environment that is necessary for secure boot.

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 is a read-only address space.

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 allows different SEC_BOOT configurations even on a single device; this also implies that multiple SROTs are possible. The code segment for the MSM8916 chipset is defined as follows.

```
/**
 Identifies the secure boot fuses which 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 boot-up 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
		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: ▪ SBL ▪ RPM firmware ▪ TrustZone kernel ▪ APPSBL ▪ TrustZone Application images ▪ WCNSS (WLAN/RIVA) ▪ LPASS ▪ Sensors (DSPS) ▪ 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 the licensee's own code signing system is being used (i.e., 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 hash of the trusted root certificate is used by the authentication logic to validate that the certificate chain is trusted (i.e., the public keys within the certificates comes from the licensee's authorized source).

NOTE: If the licensee is using QTI's CSMS along with QC SHA256 option in "Signing Root Certificate", then this fuse must be blown with hash of the trusted root certificate.

NOTE: For SHA1 this fuse must not be blown because the hash of the trusted root certificate is specified within the boot ROM code itself. Refer to *Code Signing Management System User Guide* (80-V8000-1) for details.

ROM_PK_HASH_INDEX

Blow if the licensee's root certificate hash 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 should be blown to the index in the key table that contains the hash of licensee's root certificate.

If the licensee uses CSMS for signing, this fuse should not be blown (i.e., it remains 0) because the CSMS root certificate is located 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 SECURITY_CONTROL_CORE_QFPROM_CORR_SERIAL_NUM 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 and OEM_MODEL_ID values from the SECURITY_CONTROL_CORE_OEM_CONFIG2 for authentication.

3.4 Blowing 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 in the HMAC of the code signature (see Section 5.1).

Table 3-2 OEM identifier fuses

SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_CONFIG_ROW1_LSB		
Bits	**Name**	**Description**
31:16	OEM_PRODUCT_ID	OEM-assigned product identifier. This can be left at 0 (unblown) if the OEM does not want to sign images for a particular licensee phone model.
15:0	OEM_HW_ID	Qualcomm assigns the OEM an ID for secure boot and license management. This OEM_ID must be burned with the ID value that was assigned to the OEM so that the OEM can use this capability to support either Qualcomm secure boot solutions (CSMS) or DSP dynamic feature authentication of Qualcomm or third party feature enhancements. The DSP enables feature authentication and ensures that the OEM approves DSP dynamic updates. So even if the OEM does not use CSMS for signing, it is recommended to blow the OEM_ID with their assigned ID.

NOTE: Refer to *OEM_HW_ID Provisioning Recommendation* (80-VT310-20) for more details on OEM_HW_ID use cases. Consider ADSP dynamic modules in case of LA.

3.5 Disabling debug ports

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

OEMs may also want to disable the various tracing capabilities as well.

3.6 Using 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 can be written to only once during each power cycle, and subsequent writes to them do not take effect.

Bit fields of OVERRIDE_2/OVERRIDE_3 are written by the APPS PBL based on the debug settings in the SBL 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-enabling JTAG access

Typically, field devices are locked from JTAG access to enable certain hardware security features. By writing 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 SBL1 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 SBL1 and MBA) must be signed with DEBUG OU = 0x0000000000000002. This 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, further access to enable JTAG is prohibited by the hardware itself.

If the OEM wants to re-enable JTAG access using the override registers, the SBL1 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 TrustZone 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 MSM8916 device is done in a per-device manner; SERIAL_NUM can be read from 0x0005C008, SECURITY_CONTROL_CORE_QFPROM_CORR_SERIAL_NUM.

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 SECURITY_CONTROL_CORE_QFPROM_CORR_SERIAL_NUM register. All devices carry a unique value for SECURITY_CONTROL_CORE_QFPROM_CORR_SERIAL_NUM.

4 TrustZone BSP

For details on TrustZone BSP, refer to *Security TrustZone QSEE Overview* (80-NL239-5) for MSM8916, MSM8936, MSM8939, MSM8909, MSM8952, and MSM8956/MSM8976 devices.

However, the PIL and authentication by TrustZone is not changed when compared to MSM8974/MSM8x26 chipsets. Refer to *Application Note: Enable Secure Boot on APQ8084, MSM8974, MSM8x26, MSM8x10, and MSM8x12 Chipsets* (80-NA157-20) for MSM8974/MSM8x26 chipsets.

5 Security components

This chapter explains about secure components used while authenticate the image.

5.1 MSM_HW_ID

The OEM_ID or SERIAL_NUMBER with the MSM hardware revision number (carried in the JTAG_ID register) makes up the 64-bit MSM hardware ID that is used in the hash calculation. Table 5-1 lists how the MSM_HW_ID is composed.

Table 5-1 MSM_HW_ID value

<table>
<tr><th colspan="3">MSM_HW_ID[63:0]</th></tr>
<tr><td>[63:32]</td><td colspan="2">[31:0]</td></tr>
<tr><td rowspan="3">JTAG_ID masked with 0x0FFF_FFFF</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</td></tr>
</table>

The blown eFuse combination determines whether the OEM_ID or SERIAL_NUMBER is used. Based on the fuse configuration, the value of the 64-bit MSM hardware ID can be used for the hash calculation.

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 is to ensure that the signed software works for any version of a specific MSM device.

Once the SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_CONFIG_ROW1_LSB is blown, the value is reflected in the OEM_ID register.

The licensee 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 SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_SEC_BOOT_ROW0_LSB register. If this bit is blown, the 32-bit SERIAL_NUM is used instead of the OEM_ID in the authentication.

NOTE: Instead of reading QFPROM_RAW_SERIAL_NUM register, the software should read SECURITY_CONTROL_CORE_QFPROM_CORR_SERIAL_NUM for SERIAL_NUM.

Compared to previous platforms, all 32 bits of OEM_ID can be used by licensees. The MSM_HW_IDs are checked by the software, which compares the values in the software with the values blown by eFuse.

OEMs can also use SERIAL_NUM instead of OEM_ID, as described in OU Fields HW_ID (Section 2.2.1.4.2).

When signing the build, the OEM_ID on the signing website must be the same as the QFPROM OEM_ID, which is allocated for each licensee by QTI. The model on the signing website must be the same as the OEM_PRODUCT_ID, which is allocated for each product by the licensee.

5.2 Software ID

There is no change in software ID compared to MSM8974 secure boot architecture except Hypervisor image software ID added.

The bits in the SW_ID OU field in the signed image must match the following code.

For SBL1 and eHOSTDL, which are verified by the PBL, the lowest 32 bits in the SW_ID OU field must match either 0x0 (SBL1) 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 anti-replay fuse settings, as described in Chapter 6.

SW_ID definitions

```
SBL1               = 0x0  /* SBL1 image */
MBA                = 0x1  /* MBA image */
AMSS_HASH_TABLE    = 0x2, /* Modem image hashtable */
EHOSTD             = 0x3  /* Emergency downloader image */
DSP_HASH_TABLE     = 0x4, /* lpass etc running on ADSP*/
TZ_KERNEL          = 0x7, /* TZBSP image */
APPSBL             = 0x9, /* APPSBL */
RPM_FW             = 0xA, /* RPM firmware */
TZ_EXEC_HASH_TABLE = 0xC, /* TrustZone applications - Playready/TrustZone */
WCNSS_HASH_TABLE   = 0xD  /* Pronto/WCN image */
VIDEO_HASH_TABLE   = 0xE, /* Venus image */
WATCHDOG           = 0x12 /* System debug image */
HYP                = 0x15 /* Hypervisor image */
```

In the MSM8916 chipset, the modem is reset by the HLOS, but the modem has its own PBL, so the MBA image is authenticated by the modem PBL (for more details, see Chapter 0). Thus, 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 the MSM8916 chipset, the PBL only allows images that are signed with software type 0 (APPS SBL) and 3 (eMMCBLD). This 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.

The image can also have multiple licensee root certificates (although the certificate chain goes back to only one of these root certificates). This allows a different root certificate to be chosen later in the lifetime of the phone through a software change, by signing the image with the new root certificate instead of reblowing the OEM_PK_HASH fuses with the hash of the new root certificate. In this case, the OEM_PK_HASH fuses contain the SHA256 hash over all the root certificates.

NOTE: The default use case is to have only one root certificate in the image. Multiple root certificates provide a flexibility that only a few licensees may need.

5.4 Crypto engine usage

For the MSM8916/MSM8939/MSM8929/MSM8909/MSM8952/MSM8956/MSM8976 crypto engine usage:

- CRYPTO0 – Accessible by TZBSP/apps
- MSS-CRYPTO – Accessible from Modem

PHK is used by the modem EFS; while SHK is used for SFS.

MSS-CRYPTO/CRYPTO0 use PHK1/2 respectively when a secure fuse is blown; and CRYPTO0 use SHK when a secure fuse is not blown:

- PHK: CRYPTO0 => KDF A
- PHK: MSS_CRYPTO => KDF C
- SHK: CRYPTO0 => KDF D (non-secure) or SHK (secure)
- SHK: MSS_CRYPTO => NO CONNECT

A different PHK or SHK is used depending on the debug feature; see Table 5-2 for more details.

Table 5-2 PHK/SHK key usage

	Both JTAG disable fuse and secure boot fuse are blown	Both JTAG disable fuse and secure boot fuse are not blown	Both JTAG and secure boot are re-enabled by debug OU field*
MSS_CRYPTO	Secure Qualcomm key C	Non-secure Qualcomm key C	Debug key
CRYPTO0	Secure Qualcomm key A Secure OEM key D	Non-secure Qualcomm key A Non-secure OEM key D	Debug key

*See Section 3.6.1 for details.

Per the information above, the OEM must ensure that SFS is re-created (or created) after SHK is blown, otherwise it may lead to SFS decryption failures.

Compared to MSM8960 and APQ8064 chipsets, the new crypto engine allows a unique key to be used even without blowing any eFuses.

By default, hardware keys are used for cryptography. This is done by:

- Setting the USE_HW_KEY_ENCR bit of the CE_n ($n \in [0, 2]$) configuration register, $CRYPTO_n$_CRYPTO_ENCR_SEG_CFG
- Invoking the crypto engine by writing $CRYPTO_n$_CRYPTO_GOPROC_OEM_KEY

- The OEM can override the hardware keys with their own pseudo software keys by:
  - Calling the qsee_kdf() function
  - Passing a non-NULL key as the first argument

The pseudo software key is copied to CRYPTO_ENCR_KEYn, the USE_HW_KEY_ENCR bit is cleared, and the crypto engine is invoked by writing $CRYPTO_n$_CRYPTO_GOPROC instead.

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.

6 Rollback prevention

Anti-rollback 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 attestation certificate's SW_ID OU field. There is no FEC for the anti-rollback fuses.

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

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 Enabling anti-rollback

The rollback feature is independently activated for the various images based on following bits of QFPROM
SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_CONFIG_ROW0_MSB[19-22]

Table 6-1 QFPROM anti-rollback enable register

Bit position	Name	Subsystem
0	BOOT_ANTI_ROLLBACK_EN	SBL1, RPM, TZ, HYP, APPSBL
1	TZAPPS_ANTI_ROLLBACK_EN	TrustZone secure applications
2	PIL_SUBSYS_ANTI_ROLLBACK_EN	PIL images (loaded by TrustZone)
3	MSA_ANTI_ROLLBACK_EN	MSA

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 follows:

- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_1_LSB [11:1] – Version of SBL1
- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_1_LSB [25:12] – Version of Trustzone.
- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_1_MSB [17:0] + SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_1_LSB [26:31]

  – Version of PIL images
- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_2_MSB[3:0] +

  SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_1_MSB [31:18] +

  SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_2_LSB [31:0]
  – Version of APPSBL image.

- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_2_MSB [11:4] – Version of RPM image.
- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_2_MSB[23:12] – Version of Hypervisor image.
- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_3_LSB [15:0] – Version of MBA image.
- SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_3_LSB [31:16] – Version of Modem image

The software version is determined from the SW_ID OU[63:32] field contents in the attestation certificate.

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

6.2 MBA anti-rollback

The Modem PBL verifies that QFPROM SECURITY_CONTROL_CORE_QFPROM_CORR_ANTI_ROLLBACK_3_LSB [15:0] is less than or equal to the SW_ID OU field in the MBA attestation certificate.

The MBA verifies that QFPROM SECURITY_CONTROL_CORE_QFPROM_CORR_ANTI_ROLLBACK_3_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 TrustZone anti-rollback

TrustZone anti-rollback is specified by

SECURITY_CONTROL_CORE_QFPROM_CORR_ANTI_ROLLBACK_1_LSB [25:12].

6.4 Hypervisor anti-rollback

Hypervisor anti-rollback is specified by

SECURITY_CONTROL_CORE_QFPROM_CORR_ANTI_ROLLBACK_2_MSB[23:12]

6.5 SBL1 anti-rollback

The APPS PBL verifies that

SECURITY_CONTROL_CORE_QFPROM_CORR_ANTI_ROLLBACK_1_LSB[11:1] is less than or equal to the SW_ID OU field in the SBL1 attestation certificate.

6.6 Software version rollback prevention example

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 TrustZone 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, i.e., to close a security hole.

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

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

- The hex 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 – Used to store the software version;

  Concatenate software version | image type
  - SW_ID = 0x0000000200000007

The software version value saved in

SECURITY_CONTROL_CORE_QFPROM_RAW_ANTI_ROLLBACK_1_LSB[25:12]

is interpreted by the number of bits set as 1. The QFPROM software version value is updated by the secure boot loader:

- 00000000000000000000 = 0
- 00000000000000000001 = 1
- 00000000000000000111 = 3
- 11111111111111111111 = 20

The maximum eFuse version value is 14 for the TrustZone image.

6.7 Software version rollback prevention use case

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

TrustZone with SW_ID = 0x0000000100000007

If the TrustZone image is signed with a different software version, the phone cannot boot up. For example, the TrustZone image is signed with SW_ID = 0x0000000000000007 (lower software version), or SW_ID = 0x0000000200000007 (higher software version).

Similarly, the boot loader does a software version rollback check for APPSBL/TrustZone/RPM during phone boot up. 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.

6.8 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 illustrates 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 boot ups, builds with versions less than the maximum of 19 are not allowed to proceed; only the maximum versions and up are allowed.

Table 6-2 Software version rollback limitation

Image type	Maximum version of secure upgrade supported
SBL1	11
TrustZone,	14
Hypervisor	12
RPM	8
APPSBL	50
PILs (except Modem)	24
MBA	16
MODEM PIL	16
TrustZone secure applications	0xFFFFFFFF

A phone can successfully boot up with the TrustZone image signed with SW_ID = 0x0000001400000007, even if software version = 0x14 = 20 exceeds the maximum eFuse version, which is 14. After that:

- If the phone loads an old TrustZone image signed with SW_ID = 0x0000000D00000007, software version = 0xD = 14, the phone can boot up. Basically, any software version ≥ 14 can be loaded.
- If the phone loads an old TrustZone image signed with SW_ID = 0x000000C000000007, software version = 0xC = 13, the phone cannot boot up. Any software version < 14 cannot be loaded.

6.9 TrustZone secure application anti-rollback

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 6.6) does not apply to TrustZone secure application anti-rollback. Trustzone saves a secure application APP_ID (Section 2.2.1.4.8) and software version (Section 2.2.1.4.1) to the TrustZone RPMB partition. Up to 40 anti-rollback version records of the TrustZone secure applications can be saved to the TrustZone RPMB partition.

The maximum TrustZone secure application version value is 0xFFFFFFFF.

7 Modem Boot Authentication (MBA)

MBA is used to authenticate modem subsystem images. In the past, Modem Subsystem (MSS) images were loaded and authenticated by the HLOS. With the MSM8916 chipset, the modem image is still loaded by the HLOS, but authentication is performed by the MBA. The difference now is that the modem has its own SRoT. This off-loads modem proprietary code from the APPS and TrustZone, and also because APPS and TrustZone do not have to load authenticated MSS images, APPS can run more seamlessly.

7.1 MBA image loading

As discussed in Section 2.6.1, the MSS is reset by the APPS. When the MSS comes out of reset, the Modem PBL is running and authenticates the MBA image. In MSM8916, the MBA image is in ELF format compared to MSM8974 and MSM8926, which support the MBN format.

The PIL loads MBA ELF as is in DDR and loads RMB0 with start address; The modem PBL starts loading MBA ELF from DDR to Hexagon L2 TCM. Modem PBL first loads and verifies ELF headers, loads program headers, and hash segments, and then authenticates hash segment. Figure 7-1 provides the details.

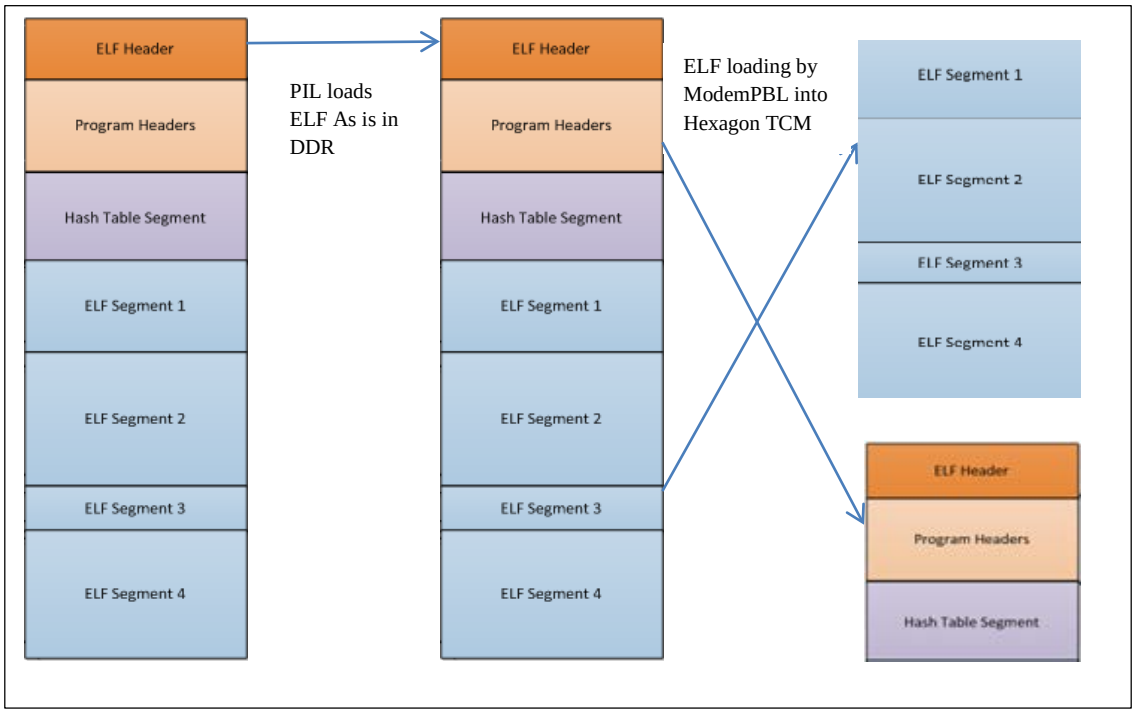


Figure 7-1 MBA Image format and loading

On the HLOS/PIL side, the HLOS still authenticates the MBA image. However, the authentication is not done by the HLOS itself, but by polling the MBA RMB status register set by MBA. Real authentication is done by the modem itself.

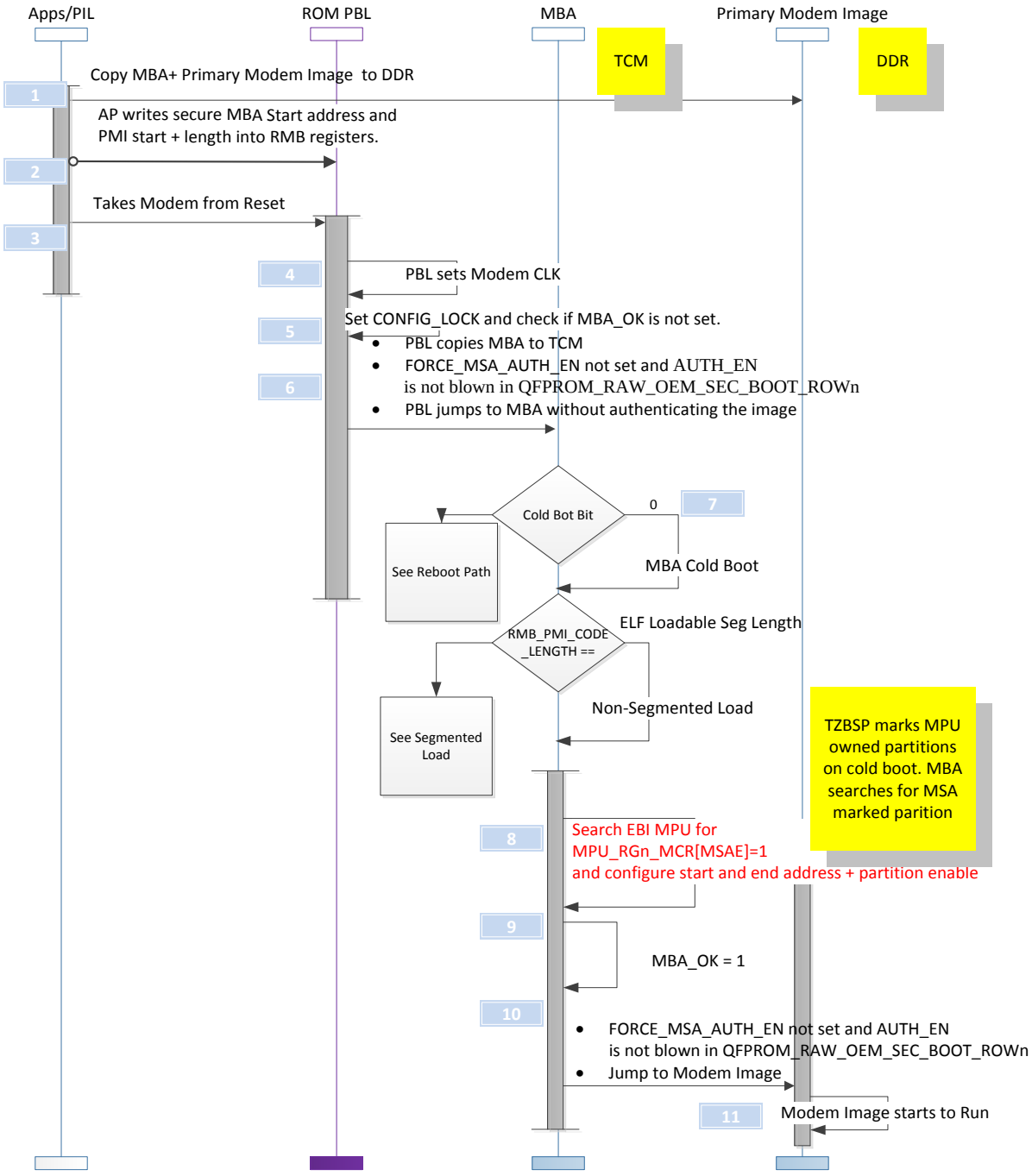


Figure 7-2 MBA authentication procedure

7.2 MSA QFPROM

MSA_ENA controls the MSA hardware block; this QFPROM is pre-blown by QTI. See Figure 7-2 for details.

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

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

   If SECURE_BOOTn is not blown, the default is index 0, which correlates to the QTI root CA.
4. 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

OEM-related QFPROM is controlled by OEM_CONFIG0:

- SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_CONFIG_ROW0_MSB
- OEM_CONFIG0_MSS_DBGEN_DISABLE
- OEM_CONFIG0_MSS_NIDEN_DISABLE

Overall debug disabling is controlled by OEM_CONFIG0:

- OEM_CONFIG0_ALL_DEBUG_DISABLE

If any of the above 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 introduced a standalone tool called SecImage. Refer to *Sectools: SecImage Tool User Guide* (80-NM248-1) for details and usage.

NOTE: It is recommended to use the latest version of SecImage tool. However, SecImage tool (ver. 2.5) was being used when this document was published.

9 QFPROM programming

QFPROM programming has an important role while enabling security features in QTI's MSM and MDM devices. With an increasing number of security features, such as anti-rollback, 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 MSM8916, MSM8936, MSM8939 MSM8909, MSM8929, APQ8016, APQ8039, MSM8952, and MSM8956/MSM8976 chipsets.

9.1 Fuse blowing process

The general process for blowing fuses is:

1. The OEM uses sectools.py to generate a customized set of fuse values. This tool generates a sec.dat binary file with the following format:

   Refer to *Sectools: FuseBlower Tool User Guide* (80-NM248-3) for details and usage of SecImage tool.

sec.dat						
secdat **header**	qfuse list header	qfuse entry	qfuse entry	qfuse entry	qfuse entry	secdat footer

WARNING: Since the sec.dat partition is always used in the raw format, avoid using the sec.dat partition to blow fuses for sensitive data, e.g., secret information. This method must only be used 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, SBL1 reads the "sec" partition from the device and loads it into the TrustZone DDR region.
4. Before jump to Appsbl, TrustZone determines if fuse blowing has already occurred. This is done by reading whether the OEM_SEC_BOOT bit of QFPROM_RAW_RD_WR_PERM_MSB is blown.
5. If it is not blown, TrustZone parses the loaded sec.dat file from DDR, and programs the list of fuses one by one.
6. To enforce the programmed value, TrustZone reboots the device.

7. After reboot, all the intended fuse lists are programmed.

   Figure 9-1 shows a block diagram overview of the process described above.

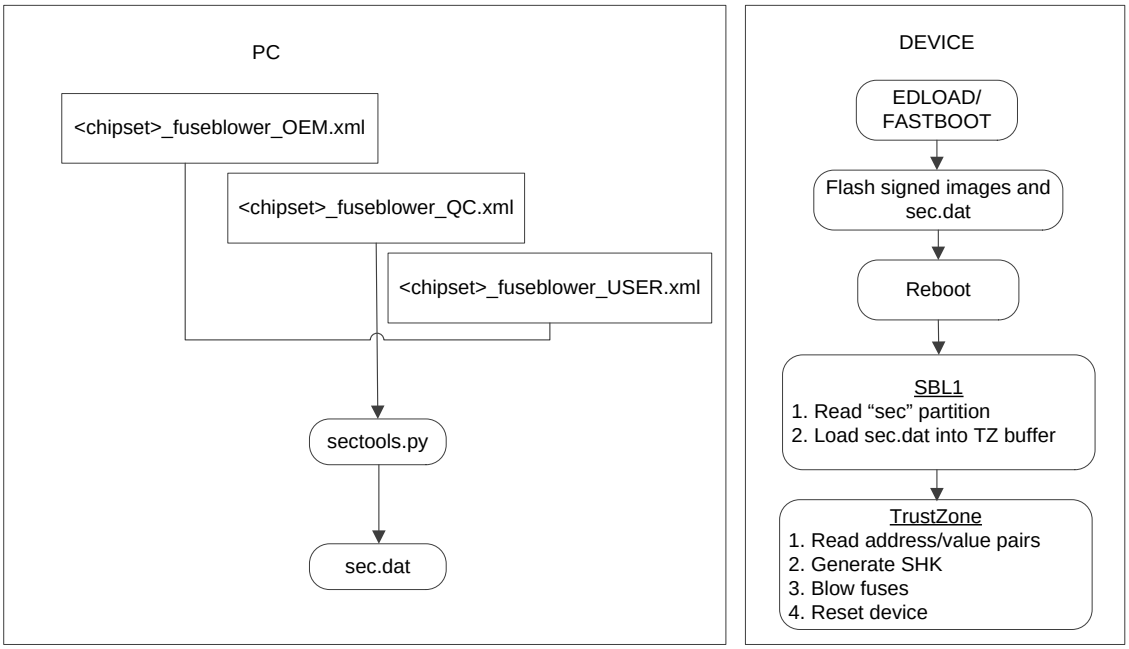


Figure 9-1 Fuse blowing process

9.2 eFuse programming procedure

In the current software, the QFPROM programming driver has been integrated to the TrustZone. And TrustZone is not executed after loaded by SBL1. TrustZone is executed after SBL1 finishes its own functionality. So, QFPROM APIs must be called in TrustZone to program the QFPROM, not SBL1.

NOTE: Licensees are recommended to use sec.dat to blow fuse. Avoid using boot_debug.cmm script method.

9.3 Fuse blown method using boot_debug.cmm script

This is another method to blow the fuses using `boot_debug.cmm` script along with TZ test image.

Licensees need to generate tzbsp_with_test image by issuing TZ build command with tzbsp_with_test parameter.

9.3.1 To generate TZ test image

Update the build command to generate tzbsp_with_test image:

```
trustzone_images/build/ms/build.sh CHIPSET=msm8916 tz hyp tzbsp_with_test
sampleapp tzbsp_no_xpu playready widevine isdbtmm securitytest keymaster
commonlib
```

NOTE: Make sure that below flag is set in

```
\trustzone_images\core\bsp\tzbsp \build\tzbsp_def_with_test.cfg file.
(It must be set in the build provided).
# Turn on test services
tzbsp_with_test_svc=1
```

9.3.2 To use boot_debug.cmm script

1. Ensure `tzbsp_with_test.mbn` is flashed onto target.
2. Execute boot_debug.cmm from boot_images/core/boot/secboot3/scripts
3. Select MSM8916 device at prompt.
4. Select non-rumi device at prompt.
5. Select QFPROM test at prompt.
6. User is prompted with a warning that `tzbsp_with_test.mbn` must be flashed on target and then asked if device has been flashed with the image. Answer YES or NO.
   a. If the answer is YES, the script continues
   b. If the answer is NO, the script terminates
7. Select read fuse, write single fuse, or write multiple fuse.
8. Supply raw fuse value (Ensure that it is the row LSB address).

9. If read fuse was selected, then script displays LSB and MSB value.
10. If write single fuse then script prompts user to enter LSB and MSB values

9.4 Programming SHK

All the keys derived in TrustZone are based off the SHK; so, it is recommended to blow this key with random values rather than using a constant value for all devices. Blowing the SHK is critical to security because various key derivations are based on this key. On MSM8916, SHK is blown in Trustzone using sec.dat.

The following permission bits must be blown for secure hardware keys.

SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_CONFIG_ROW0_MSB (MSS PHK)

- MSS_DBGEN_DISABLE
- MSS_NIDEN_DISABLE

SECURITY_CONTROL_CORE_QFPROM_RAW_OEM_CONFIG_ROW0_MSB (APPS PHK/SHK)

Refer to *MSM8916 QFPROM Programming Reference Spreadsheet* (80-NK807-97) for the list of fuses to be blown for APPS PHK/SHK.

After the hardware keys are programmed, the read/write permission for the Secondary Key Derivation Key must also be blown to protect any further read back and writes. However, blowing the SHK write/read permission must be done with caution and sufficient testing.

The read/write permission can be granted by blowing the relevant bits of READ_PERM and WRITE_PERM in SECURITY_CONTROL_CORE_QFPROM_RAW_RD_WR_PERM_LSB and SECURITY_CONTROL_CORE_QFPROM_RAW_RD_WR_PERM_MSB.

The Primary Key Derivation Key is programmed by QTI to include the read/write permissions as well. Also, further reading and writing to the SHK QFPROM must be blocked.

Figure 9-2 illustrates the programming steps for QFPROM.

The blow settings for DEVICEEN, DBGEN, NIDEN, SPIDEN, and SPNIDEN listed in *MSM8916 QFPROM Programming Reference Spreadsheet* (80-NK807-97) must be followed for an end-to-end secure solution.

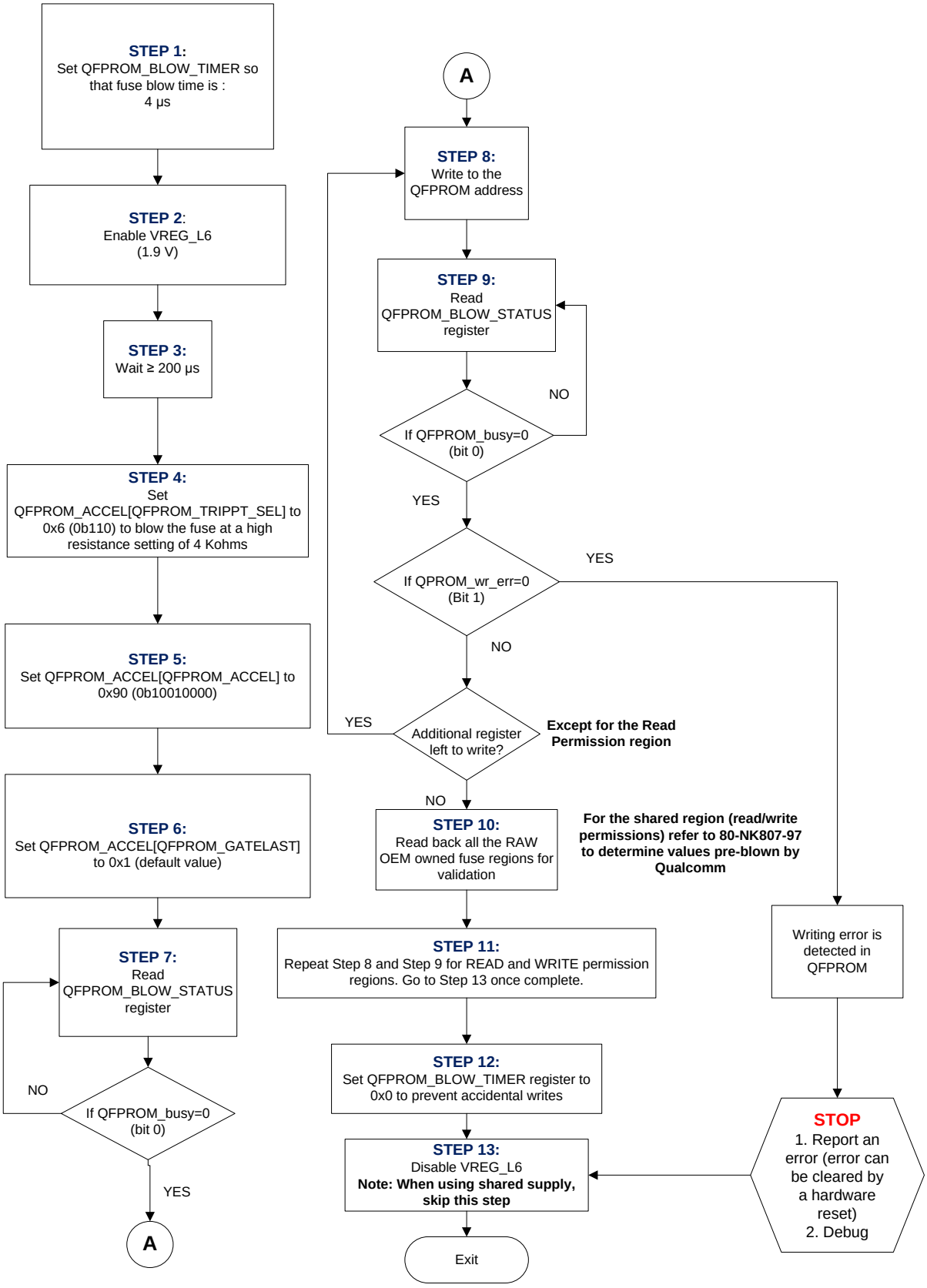


Figure 9-2 QFPROM programming steps

9.5 Sample fuse configuration

9.5.1 OEM_PK_HASH

Example

Program the QTI public key hash into the OEM_PK_HASH. First calculate the SHA256 of the QTI public key. The value is the QTI root certificate hash as follows:

```
8e cf 3e aa 03 f7 72 e2 84 79 fa 2f 0b ba e2 14 1c ca d6 f1 06 b3 84 d1 c4
62 63 ed b5 b0 28 38
```

ARM implemented by QTI is LSB; so, the whole string is reversed:

```
38 28 b0 b5 ed  63 62 c4 d1 84 b3 06 f1 d6 ca 1c 14 e2 ba 0b 2f fa 79 84 e2
72 f7 03 aa 3e cf 8e
```

Table 9-1 and Table 9-2 list the bits, names, and descriptions for OEM_PK_HASH fields.

Table 9-1 OEM_PK_HASH LSB

Bits	Name	Description
31:0	HASH_DATA0	32 bits of the hash data in this row

Table 9-2 OEM_PK_HASH MSB

Bits	Name	Description
31	PK_HASH_ROW_FEC_EN	This fuse is used to enable the Forward Error Correction for this row of PK Hash.
23:0	HASH_DATA1	24 bits of the hash data in this row

In row 0, e.g., the bits after 23rd bit are filled as 0.

```
Hex: 72f703aa3ecf8e
Binary: 11100101111011100000011101010100011111011001111110001110
Set FEC bit to 0x0: 01110010111011100000011
10101010001111101100111110001110
LSB Address 0x0005C0A8 MSB Address 0x0005C0AC
MSB(0x72f703)                          LSB (0xaa3ecf8e)
...
```

The result is:

```
SECURITY_CONTROL_CORE_QFPROM_RAW_PK_HASH_ROWn_LSB = \
  0xaa3ecf8e 0xfa7984e2  0xca1c14e2 0xc4d184b3 0x3828b0b5
SECURITY_CONTROL_CORE_QFPROM_RAW_PK_HASH_ROWn_MSB= \
  0x0072f703 0x00ba0b2f 0x0006f1d6 0x00ed6362 0x0
```

The provision QTI public key hash to OEM_PK_HASH is:

```
SECURITY_CONTROL_CORE_QFPROM_RAW_PK_HASH_ROWn_LSB (0x000580A8+8n)
```

NOTE: After OEM_PK_HASH is blown, the user must also disable further write permission to OEM_PK_HASH by blowing QFPROM_RAW_RD_WR_PERM_MSB[OEM_PK_HASH].

9.6 Multiple root certificates support

Background

OEMs need a way to customize devices for different customers who use different roots. The image has all the roots. OEMs can select a specific root by using the eFuse ROOT_CERT_HASH_INDEX. The root certificate at that index is then used in the authentication of the image.

Impact

- No impact to non-MRC request OEMs

9.6.1 Implementation and fuse settings

OEM signed images contain all the root certificates and are signed with a specific root. The fuses that are blown are:

- PK_HASH_IN_FUSE – eFuse is blown to 1.
- PK_HASH – The hash of all the root certificates is blown into this fuse.
- ANTI_ROLLBACK_ 2_MSB(ROOT_CERT_PK_HASH_INDEX[bit 31:24]) to indicate which root to use for APPS – This is used to indicate which root is used in the image except only MSA.
- ANTI_ROLLBACK_3_MSB(MODEM_ROOT_CERT_PK_HASH_INDEX[bit 7:0] to indicate which root to use for MSA – This is used to indicate which root is used in the image for MSA.
- OEM_CONFIG_ROW0_LSB(ROOT_CERT_TOTAL_NUM [bit 21:18]) to indicate Total number of roots – This is used to indicate how many roots are hashed in the PK_HASH fuse. A value of 0 indicates legacy support where the hash of one root certificate is blown
- ROOT_CERT_TOTAL_NUM(QFPROM_OEM_CONFIG_ROW0_LSB) is used to indicate how many roots are hashed in PK_HASH Fuse. A value of 0 indicates legacy support where the hash of one root certificate is blown.

Table 9-3 ROOT_CERT_HASH_INDEX values and fuses

ROOT_CERT_HASH_INDEX	Value	Fuse
00	0000 0000	Production device, no certificate selected; default to certificate 0
F0	1111 0000	Production device fixed to certificate 0 (Use for exclusivity and makes a different SKU) * Should not set this value if PK_HASH was set based on MRC. It disables boot.
E1	1110 0001	Device fixed to certificate 1
D2	1101 0010	Device fixed to certificate 2
C3	1100 0011	Device fixed to certificate 3
B4	1011 0100	Device fixed to certificate 4
A5	1010 0101	Device fixed to certificate 5
96	1001 0110	Device fixed to certificate 6
87	1000 0111	Device fixed to certificate 7
78	0111 1000	Device fixed to certificate 8
69	0110 1001	Device fixed to certificate 9
5A	0101 1010	Device fixed to certificate 10
4B	0100 1011	Device fixed to certificate 11
3C	0011 1100	Device fixed to certificate 12
2D	0010 1101	Device fixed to certificate 13
1E	0001 1110	Device fixed to certificate 14
0F	0000 1111	Device fixed to certificate 15
Others	Invalid value	Disable boot

9.6.2 MRC index switch in TZ code

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.

OEMs can switch MRC index using below function in TZ.
trustzone_images\core\securemsm\trustzone\qsee\oem\<chipset>\src\tzbsp_oem_secboot.c

```
void tzbsp_oem_fuse_config(void)
{
~~#if 0~~
#if 1
  /* In this example, Apps root will be 16th . MSA root will be 2nd
because the index is 0-based */
secboot_switch_mrc_index(15, 1);
#endif
}
```

NOTE: In this example, the total number of root certificates is 16.

9.6.3 To switch MRC index

1. Flash signed images with 16 root certificates and set root cert index for both apps and modem to 0 (default).
2. Flash sec.dat which blows secboot enable, combined oem_pk_hash of 16 roots, total number of root certificates, etc.
3. Flash new TZ, which blows apps and modem root cert index to <OEM's preferred index>. Refer to Section 9.6.2 for details.
4. Restart the device after flashing new TZ. It boots for the first time as it blow fuses during bootup.
5. Restart the device again. It goes to emergency download mode (9008).
6. Sign images with 16 root certificates. Given images should have certificate chain and signature, which are matched with root cert index.
7. Flash the signed images on to the device and reboot.
8. Device should boot and all subsystems should be up and running.

NOTE: In this example, total number of root certificates is 16.

10 Debug policy

NOTE: Numerous changes were made in this section.

CAUTION: *MSM8994/MSM8992/MSM8952 Debug Policy* (80-NU498-1) is not applicable for MSM8952.LA products.

NOTE: For MSM8956/MSM8976 debug policy, refer to *Debug Policy User Guide for MSM8996, MSM8976, MSM8956* (80-NV396-72). This document is also applicable for LA products.

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.

To get all regions as part of RAMDUMP from secure boot device, sign the sbl1.mbn image with DEBUG ON + CRASHDUMP ON along with serial number as follows:

File name: <sectool>/config/ <chipset>/<chipset>_secimage.xml

Example: SecTools-Build-3.10\config\8952\8952_secimage.xml

Change 1

```
--------
<image sign_id="sbl1" name="sbl1.mbn" image_type="elf_as_bin">
<general_properties_overrides>
<sw_id>0x0000000000000000</sw_id>
<crash_dump>0x002341A900000001</crash_dump>
```

The crash_dump option field contains 32-bit chipset serial number (e.g., 0x002341A9) + 32 bit crash dump ON option.

The serial number is unique to each chipset. To dump serial number from device, use the ADB command:

```
>>adb root
>> adb shell /system/bin/r 0x00058008 or 0x0005C008
```

Change 2

```
-----------
msm_part>0x009600E1</msm_part>
<oem_id>0x0000</oem_id>
<model_id>0x0000</model_id>
<debug>0x0000000000000002</debug>
<debug>0x002341A900000003</debug> /*Add this line with chipset serial
number*/
```

The debug option field contains 32-bit chipset serial number (e.g., 0x002341A9) + 32 bit DEBUG option.

NOTE: In CSMS tool, during sign-up, select ON for “Debug” and “Retail Crash Dump” options.

11 Generate components of QPSA

Replace the private/public key in the QDST.zip file if you are using QDST for image signing. Similarly, replace the private/public key in the qpsa.zip file if you are using QPSA for image signing. QPSA uses QDST.py as a baseline and adds the APP_ID support to sign TZ apps. Also, based on the generated root certificate for this process, blow the SHA256 value of the root certificate to OEM_PK_HASH QFPROM along with other indicative bits to allow the PBL to read OEM_PK_HASH instead of the pre-blown hash present in the boot ROM.

Licensees can use the following OpenSSL command to generate their own private/public key and certificates. Both QDST.zip and qpsa.zip have the sample opensslroot.cfg.

Generate a root CA private key

```
openssl genrsa -out oem_rootca.key -3 2048
openssl req -new -key oem_rootca.key -x509 -out oem_rootca.crt -subj
   /C="US"/ST="CA"/L="SANDIEGO"/O="OEM"/OU="General OEM rootca"/CN="OEM
   ROOT CA" -days 7300 -set_serial 1 -config opensslroot.cfg
```

Generate an attestation CA private key

```
openssl genrsa -out oem_attestca.key -3 2048
```

Make a CSR

```
openssl req -new -key oem_attestca.key -out oem_attestca.csr -subj
   C="US"/ST="CA"/L="SANDIEGO"/O="OEM"/OU="General OEM attestation
   CA"/CN="OEM attestation CA" -days 7300 -config opensslroot.cfg
```

Sign the CSR

```
openssl x509 -req -in oem_attestca.csr -CA oem_rootca.crt -CAkey
   oem_rootca.key -out oem_attestca.crt -set_serial 5 -days 7300 -extfile
   v3.ext
openssl genrsa -out oem_attest.key -3 2048
openssl req -new -key oem_attest.key -out oem_attest_cert.csr -subj
   /C="US"/ST="CA"/L="SANDIEGO"/O="OEM"/OU="01 0000000000000000
   SW_ID"/OU="02 0000000000000000 HW_ID"/OU="03 000000000000000F
   DEBUG"/OU="04 0000 OEM_ID"/CN="OEM attestation certificate"/OU="05
   00020000 SW_SIZE"/OU="06 0000 MODEL_ID"/OU="07 0001 SHA256" -days 7300
   -config opensslroot.cfg
openssl x509 -req -in oem_attest_cert.csr -CA oem_attestca.crt -CAkey
   oem_attestca.key -out oem_attest_cert.crt -days 7300 -set_serial 7
   -extfile v3_attest.ext
```

NOTE: QTI uses different OpenSSL extension files (v3.ext, v3_attest.ext) when generating the attestation CA and attestation certificate. Misuse of OpenSSL extension files may lead to secure boot failure due to an invalid certificate chain. For details, refer to Appendix B.

A FAQs

QUESTION 1 Will I be able to use non-secure software on a secure board?

ANSWER No. You always have to use signed images to boot up. Otherwise, you end up going to Download mode, as PBL fails to boot up because of an authentication failure.

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

ANSWER The secure boot process and JTAG disabling fuses are not related in any way. It is always advisable to test secure boot as a standalone without mixing it with any other feature testing. Disabling JTAG is only required for enabling SFS, which is a different concept compared to secure boot.

QUESTION 3 How do I get CSMS access?

ANSWER For any CSMS license, a designated administrator is available at QTI. You need to raise a case and it will be assigned to the CSMS administrator for further processing. Getting a license for the first time may require 30 to 40 days because of the required legal/document processing.

QUESTION 4 Do we need to care about software ID while generating certificates?

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

QUESTION 5 Can we use a signed image on a non-secure board?

ANSWER Yes. You should be able to use the signed software images on a non-secure board without any expected problems..

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

ANSWER Yes. There is a trade-off between security and time. The extra authentication process adds additional time compared to a non-secure boot.

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

ANSWER Signed and unsigned images differ by 6-7 KB if it is non-MRC.

QUESTION 8 How do I read the JTAG_ID register?

ANSWER You can run the command below on an APP window:

```
d.in 0x0005E07C/long
```

QUESTION 9 Should I resign the software every time I do the compilation?

ANSWER It is always advisable to resign the software component if you are recompiling it.

QUESTION 10 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 11 Which option do I need to select when generating SHA1 or SHA256 certificates?

ANSWER Our software supports both configurations, but SHA256 is advisable because of its performance and reliability.

QUESTION 12 Do we check for the validity of certificates against wall clock date and time?

ANSWER No. Validity dates in certificates are required fields but are not checked against the realtime as there is no possibility of tracking the calendar day/time during the boot-up process. There is no expiry date for your certificates.

QUESTION 13 Is it mandatory to use CSMS for code signing?

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

QUESTION 14 I do not see the chipset that I need to use for signing.

ANSWER It might be because you are not licensed for that specific MSM device. Feel free to contact the CSMS administrator to ask for help. If you have a license for the MSM device, the administrator will provide you with the access for that specific MSM device for code signing.

QUESTION 15 Can I do secure boot testing without blowing any fuses?

ANSWER Yes, it is possible. However, it is easy and reliable to do secure boot testing with blown fuse hardware.

QUESTION 16 When should we blow the FEC bit while blowing the QFPROM?

ANSWER The FEC bit should be enabled only after all configuration bits are blown. Once the FEC bit is blown for a particular row, no other bit of that row can be changed any time later.

QUESTION 17 Where can I find the spare fuse for OME usage?

ANSWER Spare regions 15, 16, 17, 18, and 19 are available for OEM use. Refer to relevant QFPROM Programming Reference Guide for the required chipset in Appendix D.

For example, MSM8952 does not have “Spare regions 15 and 16” but has “Spare Regions 20 and 21”.

B QPSA-related scripts

B.1 Regenerate QPSA certificates

other_src/cert.sh

```
#! /bin/sh

DIGEST=""

if [ x$1 == x"sha256" ]; then
        DIGEST="-SHA256"
        echo "Using SHA256 to sign certificates.."
else
        echo "Using SHA1 to sign certificates.."
        DIGEST="-SHA1"
fi

openssl genrsa -out qdst_rootca.key -3 2048

openssl req -new -key qdst_rootca.key -x509 -out qdst_rootca.crt ${DIGEST}
   -subj /C=US/ST=CA/L="San Diego"/OU="General Use Test Key (for testing
   only)"/OU="CDMA Technologies"/O=QUALCOMM/CN="QCT Root CA 1" -days 7300
   -set_serial 1 -config opensslroot.cfg

openssl genrsa -out qdst_attestca.key -3 2048

openssl req -new -key qdst_attestca.key -out qdst_attestca.csr ${DIGEST}
   -subj /C=US/ST=CA/L="San Diego"/OU="CDMA Technologies"/
   O=QUALCOMM/CN="QCT Attestation CA" -days 7300 -config opensslroot.cfg

openssl x509 -req -in qdst_attestca.csr -CA qdst_rootca.crt -CAkey
   qdst_rootca.key -out qdst_attestca.crt ${DIGEST} -set_serial 5
   -days 7300 -extfile v3.ext

openssl genrsa -out qdst_attest.key -3 2048
```

```
openssl req -new -key qdst_attest.key -out qdst_attest.csr ${DIGEST} -subj
   /C=US/ST=CA/L="San Diego"/emailAddress=corebsp.mav@qualcomm.com/
   OU="07 0001 DIGEST"/OU="06 0001 MODEL_ID"/OU="05 00002000
   SW_SIZE"/OU="04 0001 OEM_ID"/OU="03 000000000000000F DEBUG"/
   OU="02 007180E100010001 HW_ID"/OU="01 0000000000000000
   SW_ID"/O=QUALCOMM/CN="QCT Attestation Cert" -days 7300 -config
   opensslroot.cfg

openssl x509 -req -in qdst_attest.csr -CA qdst_attestca.crt -CAkey
   qdst_attestca.key -out fsb_attest.crt ${DIGEST} -set_serial 7 -days 7300
   -extfile v3_attest.ext

openssl x509 -in qdst_rootca.crt -inform PEM -out qdst_rootca.cer -outform
   DER

openssl x509 -in qdst_attestca.crt -inform PEM -out qdst_attestca.cer
   -outform DER

echo "test case 1"
echo
"================================================================================"
cat qdst_rootca.crt qdst_attestca.crt > my.crt            # add for testing
only
openssl verify -CApath . -CAfile my.crt fsb_attest.crt    # add for testing
only
```

B.2 OpenSSL extension files

other_src/v3.ext

```
authorityKeyIdentifier=keyid,issuer
subjectKeyIdentifier=hash
basicConstraints = CA:true,pathlen:0
keyUsage = cRLSign, keyCertSign
```

other_src/v3_attest.ext

```
crlDistributionPoints=URI:http://crl.qdst.com/crls/qctdevattest.crl
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE,pathlen:0
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
```

B.3 qpsa.zip

Create a qpsa.zip file that has the following files:

- opensslroot.cfg
- qpsa_attestca.cer
- qpsa_attestca.key
- qpsa_rootca.cer
- qpsa_rootca.key
- v3_attest.ext v3.ext

C Compute OEM_PK_HASH

Code snippet from other_src/compute_oem_pk_hash.c file

```
#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

Document	
Qualcomm Technologies, Inc.	
Code Signing Management System User Guide	80-V8000-1
Sahara Protocol Specification	80-N1008-1
Sectools: SecImage Tool User Guide	80-NM248-1
Sectools: FuseBlower Tool User Guide	80-NM248-3
Sectools: KeyProvision Tool	80-NM248-5
Sectools: DebugPolicy Tool	80-NM248-6
MSM8916 QFPROM Programming Reference Spreadsheet	80-NK807-97
MSM8x36 and MSM8x39 QFPROM Programming Reference Spreadsheet	80-NM683-97
MSM8909 and MSM8208 QFPROM Programming Reference Spreadsheet	80-NP408-97
MSM8916 Security TrustZone QSEE Overview	80-NL239-5
Application Note: Enable Secure Boot on APQ8084, MSM8974, MSM8x26, MSM8x10, and MSM8x12 Chipsets	80-NA157-20
MSM8909 Boot Architecture Overview	80-NR964-3
OEM_HW_ID Provisioning Recommendation	80-VT310-20
Presentation: MSM8952 Boot Architecture Overview	80-NV610-3
MSM8952 QFPROM Programming Reference Spreadsheet	80-NT665-97
MSM8994/MSM8992/MSM8952 Debug Policy	80-NU498-1
Presentation: MSM8956/MSM8976 Boot Architecture Overview	80-NU154-8
MSM8956, MSM8976, APQ8056 and APQ8076 QFPROM Programming Guide	80-NT667-97
Debug Policy User Guide for MSM8996, MSM8976, MSM8956	80-NV396-72
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

Acronym	Definition
APPS	Applications Processor
APPS PBL	Applications Primary Boot Loader
APPSBL	Applications Boot Loader
CA	Certificate Authority
CSMS	Code Signing Management System
DDR	Double Data Rate
DN	Distinguished Name
ELF	Executable And Linking Format
eMMC	Embedded Multimedia Card
eMMCBLD	Qualcomm image used for downloading eMMC
FEC	Forward Error Correction
FLCB	Fast Low Current Boot
HLOS	High-Level Operating System
HMAC	Hashed Message Authentication Code (RFC 2104)
JTAG	Joint Test Action Group
KDF	Key Derivation Function
MBA	Modem Boot Authentication
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
QFPROM	Qualcomm Fuse-Programmable Read-Only Memory
QGIC	Qualcomm Generic Interrupt Controller
QPSA	Qualcomm Platform Signing Application (QPSA)
RMB	Relay Message Buffer
RPM	Resource Power Manager
RPMB	Replay Protected Memory Block
SBL	Secondary Boot Loader
SCM	Secure Channel Manager
SDI	System Debug Image
SDRAM	Synchronous Dynamic Random Access Memory
SFS	Secure File System
SHK	Secondary Hardware Key
SMC	Secure Monitor Call
SPS	Qualcomm Smart Peripheral Subsystem
SRoT	Secure Root Of Trust

Acronym	Definition
TCM	Tightly Coupled Memory
TZBSP	Trustzone Board Support Package
TZExec	Trustzone Executive
VMIDMT	Virtual Machine Id Mapping Table
WCN	Wireless Communication Network