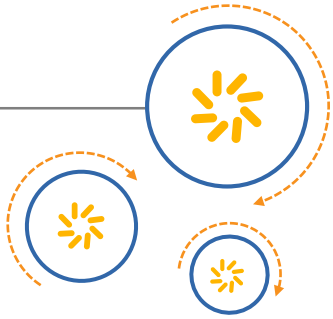
Qualcomm Technologies, Inc.



Sectools: SecImage Tool

User Guide

80-NM248-1 K

July 20, 2015

Questions or comments: https://support.cdmatech.com

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

Revision history

Revision	Date	Description
A	January 2014	Initial release
B	March 2014	Changed document title; updated Chapter 4.
C	April 2014	Added Section 4.7 and Appendix C; updated Chapter 4.
D	May 2014	Updated Chapters 2, 3, and 4; added Sections 4.2.1, 4.2.4.2.4., and 4.3.
E	July 2014	Updated Sections 4.2.4, 4.8.2, and 4.10, and Appendix C.
F	August 2014	Added Section 4.7.2; updated Appendixes A and B.
G	September 2014	Updated Sections 1.1, 2.1, 2.3, 4.2, 4.5–4.8, 4.10, 4.11, Chapter 3, and Appendix A.
H	January 2015	Added Section 4.13 and Appendix D.
J	May 2015	Updated Section 4.7.2 with new CASS capability names.
K	July 2015	Added CASS Linux reference, removed (former) Section 4.7.2, updated Section 4.9.

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document provides detailed instructions on how to sign binary images using the SecImage tool. A complete image signing solution, including test and production signing, re-signing with pre-existing certificates and private keys, unified encryption for ELF images, and validating signed secure images, is provided through the SecImage tool.

This document describes SecImage tool version 3.x. For 2.x usage, refer to previous versions.

1.2 Scope

This document is intended for developers who need to understand the detailed steps for signing images.

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>`.

1.4 References

Reference documents are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies, Inc.		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q2	*Code Signing Management System User Guide*	80-V8000-1
Q3	*MSM8274/MSM8274AB, MSM8674/MSM8674AB, MSM8974/MSM8974AB Device Specification*	80-NA437-1
Q4	*Application Note: Enable Secure Boot on MSM8974, MSM8X26, MSM8X10, and MSM8X12 Chipsets*	80-NA157-20
Q5	*Secure Software Download on Windows 8 Platform User Guide*	80-NE685-1
Q6	*Installing the CASS Credential Software 8.3 User Guide*	80-N7185-3
Q7	*Application Note: Enable Secure Boot on MSM8992 and*	80-NM328-88

1.5 Technical assistance

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

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

1.6 Acronyms

For definitions of terms and abbreviations, refer to [Q1]. Table 1-2 lists terms that are specific to this document.

Table 1-2 Acronyms

Acronym	Definition
ADSP	Audio Digital Signal Processor
CASS	Code Authorization Signing Services
CSMS	Code Signing Management System
MBA	Modem Boot Authentication
MRC	Multiple Root Certs
MSA	Modem Self Authentication
OTA	Over-the-Air
PBL	primary boot loader
PKI	public key infrastructure
QFPROM	Qualcomm® Fuse-Programmable Read-Only Memory
QPSA	Qualcomm Platform Signing Application
RPM	Resource Power Manager
SBL	secondary boot loader
SSD	secure software download
TZBSP	TrustZone Board Support Package
TZExec	TrustZone Executive
WCNSS	Wireless Connect Subsystem

2 Introduction to SecImage Tool

2.1 Key features

The SecImage tool is a standalone signing tool that is developed in Python. It provides the following functionality:

- Sign or re-sign images: MBN/ELF/EWM
- Unified encryption for ELF image
- SSD encryption for ELF/MBN image
- Validation for image hash and signature
- Support for MRC feature, 4k key
- Support CSMS signing
- Support CASS2 signing

The SecImage tool has the following features:

- Independent of the build system
- Takes one or multiple image binaries to sign
- Supports SBL1, ARM TrustZone Board Support Package (TZBSP), TrustZone Executive (TZExec), RPM, WCNSS, ADSP, and Venus images
- Dynamically detects 32-bit and 64-bit images and signs accordingly
- Provides consistent commands to initiate image operations (i.e., signing, encryption, validation)

The main config file has the following sections:

- Metadata: chipset and config file version
- General_properties: Basic configuration of signing attributes
- Parsegen: Image configuration of image format.
- Signing
  - Signer attributes
    - CASS: CASS signer configurations
- Postprocess: Consists of external tools to be run
- Data_provisiong: set base path for data provisioning assets

- Images_list: Consists of a number of supported images with properties. It contains sign_id, image name, image type, and cert_config to be used for signing, postprocess commands, general properties to override

To support general secured boot signing, the SecImage tool is released in: <Meta build>/common/tools/sectools (see Section 1.1).

2.2 System diagrams

Figure 2-1 illustrates using the SecImage tool to sign and re-sign images.

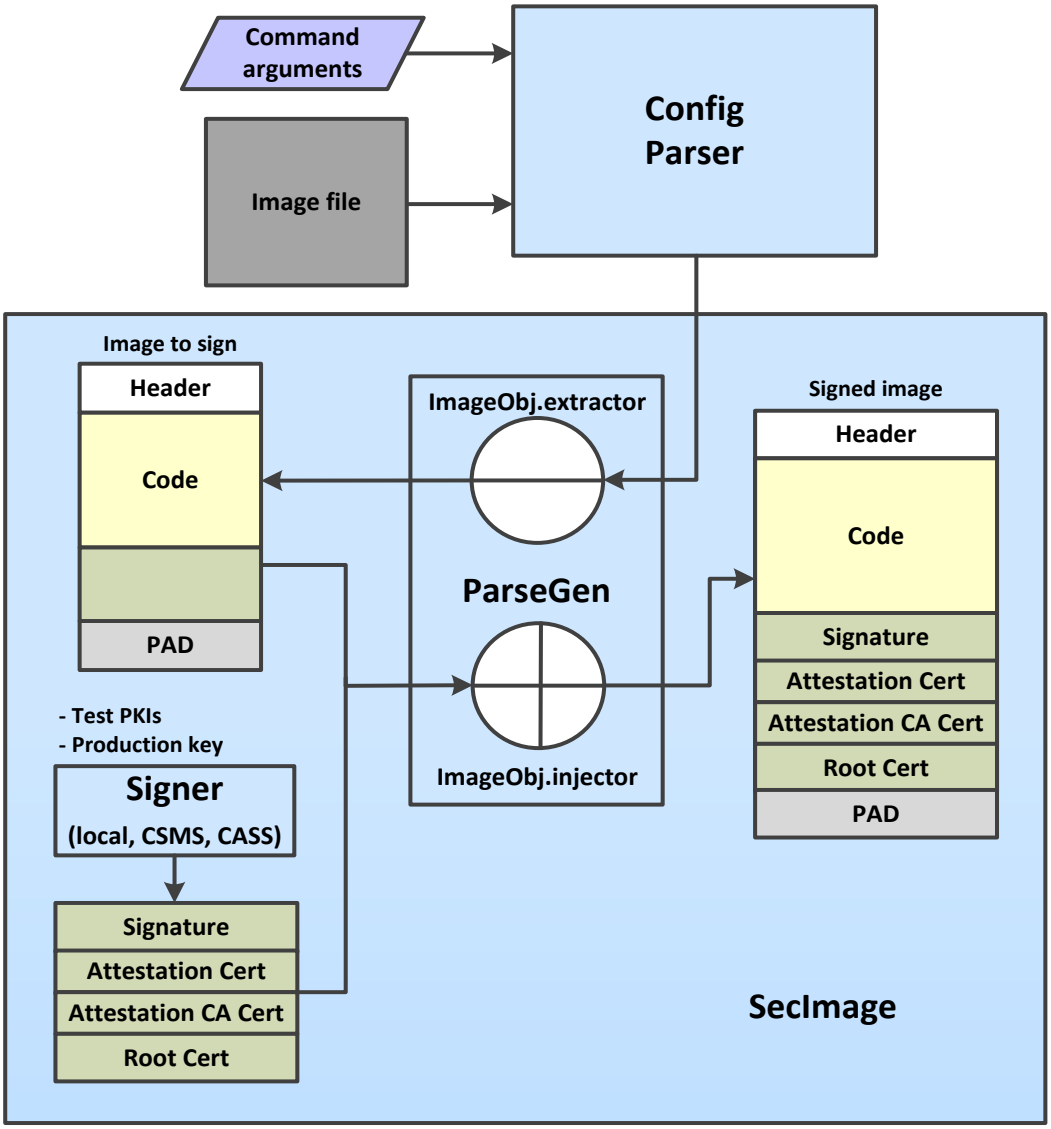


Figure 2-1 Image signing with SecImage

Figure 2-2 illustrates using the SecImage for unified encryption.

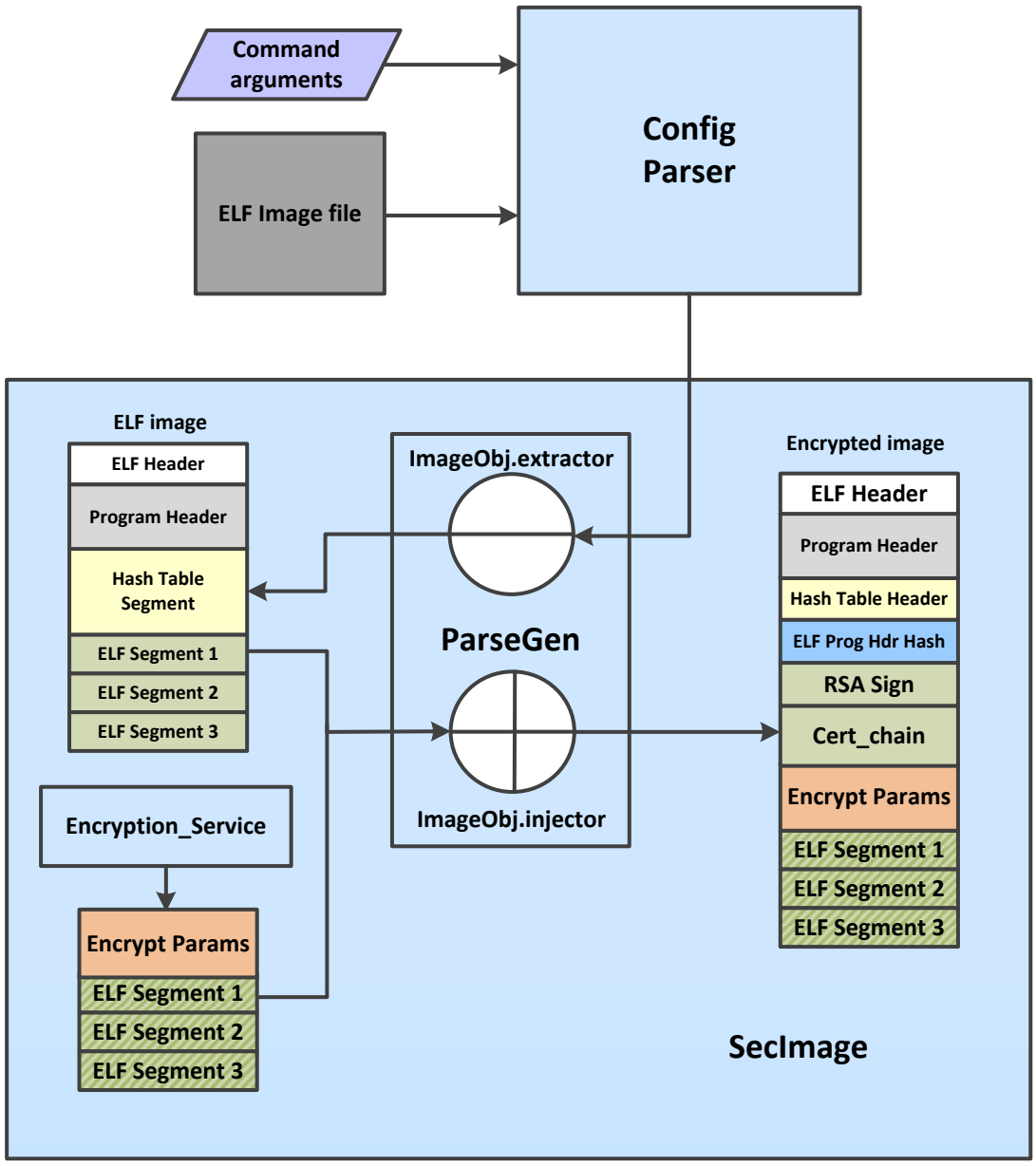


Figure 2-2 Unified encryption with SecImage

Figure 2-3 illustrates using the SecImage for SSD encryption.

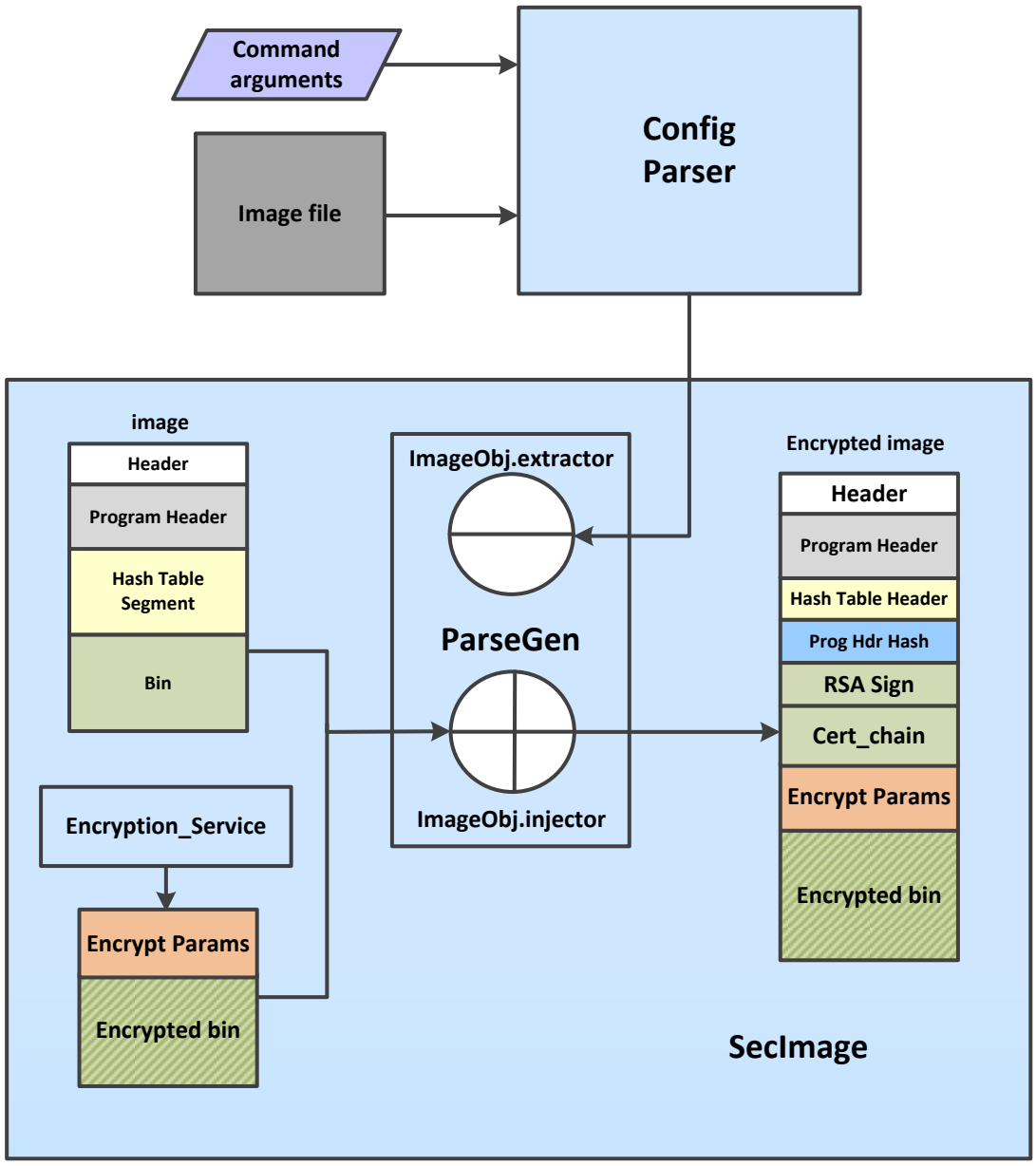


Figure 2-3 SSD encryption with SecImage

Figure 2-4 illustrates using the SecImage for image validation.

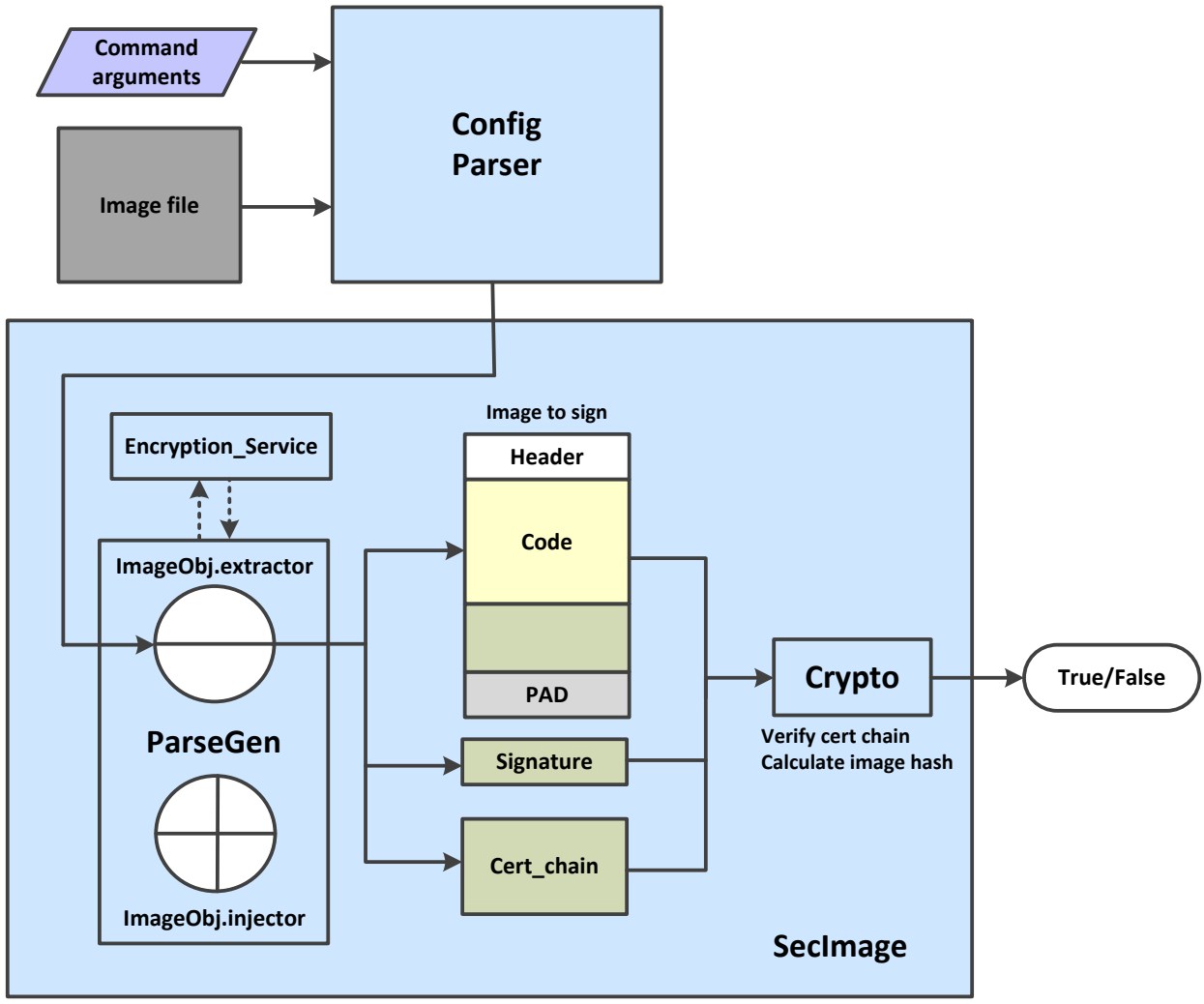


Figure 2-4 Image validation with SecImage

2.3 Tool components

The SecImage tool is part of the Sectools package and includes the following components/folders that are used for image signing:

```
<sectools>/
   | sectools.py               (main tool launcher command interface)
   |
   | -- bin/WIN (Windows binary to perform cryptographic operations)
   | -- bin/LIN (Linux binary to perform cryptographic operations)
   |
   | -- config/                (chipset-specific config files directory)
   | -- config/<chipset>/      (preconfigured templates directory)
   | -- config/xsd             (xsd for config xml)
   |
   | -- sectools/features/isc/secimage.py  (main Secimage python script)
   | -- sectools/features/isc/             (main Secimage core code)
   |
   | -- resources/data_prov_assets (assets for signing and encryption)
   |
   | -- sectools/common/core              (infrastrure)
   | -- sectools/common/crypto            (crypto services)
   | -- sectools/common/data_provisiong   (data provision)
   | -- sectools/common/parsegen          (image utilities)
   | -- sectools/common/utils             (core utilities)
```

3 SecImage Tool Usage

The supported command line options for the SecImage tool are:

```
Sectools.py secimage --meta_build=<meta_build_path> --chipset=<chipset>
--image_file=<file_path> --sign_id=<sign_id> --mini_build=<mini_build_dir>
--output_dir=<output_dir> --config=<config_file_path> --<operation>
```

or

```
sectools.py secimage -m <meta_build_path> -i <file_path> -g <sign_id> -p
<chipset> -n <mini_build_dir> -o <output_dir> -c <config_file_path> -
<operation_short>
```

Where:

- `<meta_build>` is the meta build location
- `<image_file>` is the individual image file path
- `<sign_id>` identifies the image format and signing attributes if the image name is the same, but the image types are different for different build variants (i.e., same `sign_id` can be used across platforms, but is unique for given chipset)
- `<chipset>` is optional. If it is given, SecImage will automatically use the corresponding chipset-specific configuration xml
- `<mini_build>` is the minimized build directory. SecImage will process the images and output to minimized build directory. Each image path is defined in meta build contents.xml
- `<output_dir>` is the output directory to store the signed images. The output directory would be <output_dir>\<chipset>\<sign_id>. If not given, the default location would be <Secimage_dir>\secimage_output.
- `<config_file_path>` is the chipset-specific SecImage configuration xml path
- `<operation>` can be one of the following
  - Integrity check (t) – add hash table segment
  - Sign (s) – Process the image for signing
  - Encryption (e) – Process the image for encryption
  - Validate (a) – Validate signed image
  - verify_input (l) – Verify arguments and config file

Starting from Sectools 3.x, override parameters from command line arguments is supported.

To get parameters supported for overrides, user can run "`--overrides -h`," described in Section 4.2

4 Signing, Encryption, and Validation

The SecImage tool can be used to sign and encrypt images, and verify if the given signed/encrypted images are valid.

The following sections provide instructions on how to perform signing, encryption, and validation.

4.1 Prerequisites

Prerequisites for SecImage tool operation are:

- OpenSSL 1.0.1g for Linux (or later version); OpenSSL 1.0.1g for Windows is included in the package
- Python 2.6 (or later version)
- The SecImage tool makes use of the system temporary folder as scratch space to create intermediate output. Ensure that the tool has permission to write to that directory.
  - Windows location: `%temp%` (This is an environment variable.)
  - Linux location: */tmp*

4.2 SecImage configuration file

SecImage config file provides configurations to sign, postprocess, and validate secure images. It consists of a number of configurations for signing attributes and image properties.

The main config file has the following sections:

```
<secimage.xml>
   | -- metadata (chipset and config file version)
   | -- general_properties (basic configuration of signing attributes)
   | -- parsegen (image configuration of image formate)
   | -- signing (signing attributes)
      | -- signer_attributes
         | -- cass_signer_attributes (CASS server configuration and user
              identity)
   | -- postprocess (external tools to be run post image process)
   | -- data_provisioning (base path to data provisiong assets)
   | -- image_list (a list of supported images with image format and
        signing attributes)
```

More details about the configurations are provided below.

4.2.1 Metadata

As SecImage configuration xml is chipset specific, the metadata section specifies which chipset this xml is for and what config version it is. For Sectools v2.6 and beyond, config version check is enforced. This prevents the user from mistakenly using an old config file template.

```
<metadata>
   <chipset>8994</chipset>
   <version>2.0</version>
</metadata>
```

4.2.2 General properties

The basic configuration includes all the fields the user can modify for generating secure images.

```
<general_properties>
    <selected_signer>local</selected_signer>
    <selected_encryptor>unified_encryption_2_0</selected_encryptor>
    <selected_cert_config>qc_presigned_certs</selected_cert_config>
    <cass_capability>secboot_sha2_root</cass_capability>

    <key_size>2048</key_size>
    <exponent>3</exponent>

    <mrc_index>0</mrc_index>
    <num_root_certs>1</num_root_certs>

    <msm_part>0x009400E1</msm_part>
    <oem_id>0x0000</oem_id>
    <model_id>0x0000</model_id>
    <debug>0x0000000000000002</debug>

    <max_cert_size>2048</max_cert_size>
    <num_certs_in_certchain>3</num_certs_in_certchain>
</general_properties>
```

4.2.3 Parsegen

This section defines image type and format (i.e., header size, if has hash table, etc.). SecImage supports MBN (header 40 bytes or 80 bytes), ELF (with or without hash table, with preamble), padding for OTA upgrade, and MBA (ELF wrapped mbn).

The image format supported is provided below.

4.2.3.1 SBL1 signing

An SBL1 image can be signed with MBN or ELF format. With MBN format, by default, preamble, magic number, and ota_enabled feature are set to “no” for flashless builds. To support OTA upgrade for 9x25 and 9x35, nand based sbl1 has preamble and magic number. The size of the SBL1 image will be padded to align with the nand page/block size for optimal access. The following signing attributes should be set accordingly for nand sbl1 image.

```
<image_type id='mbn_80b_pbl_ota'>
   <file_type>mbn_pmbl</file_type>
   <mbn_properties>
      <header_size>80</header_size>
   </mbn_properties>
   <pmbl_properties>
      <preamble_size>10</preamble_size>
      <has_magic_num>true</has_magic_num>
      <ota_enabled>false</ota_enabled>
      <page_size>0</page_size>
      <num_of_pages>0</num_of_pages>
      <min_size_with_pad>256</min_size_with_pad>
   </pmbl_properties>
</image_type>
```

In MDM9x45, SBL1 image is ELF format. It is standard ELF for flashless build, but ELF with preamble for nand build. The following signing attributes should be set accordingly for nand sbl1 image. This new format is supported in Sectools v2.14 and beyond.

```
<image_type id='elf_preamble'>
   <file_type>elf_pmbl</file_type>
   <elf_properties>
      <has_hash_table>true</has_hash_table>
      <image_type>0</image_type>
   </elf_properties>
   <pmbl_properties>
      <preamble_size>10</preamble_size>
      <has_magic_num>true</has_magic_num>
      <ota_enabled>false</ota_enabled>
      <page_size>0</page_size>
      <num_of_pages>0</num_of_pages>
      <min_size_with_pad>256</min_size_with_pad>
   </pmbl_properties>
</image_type>
```

4.2.3.2 MBA signing

An MBA image can be signed in one of the following two formats:

- ELF wrapped MBN: An MBA image is signed as MBN format and wrapped into ELF format by calling mba_elf_builder.py.
- ELF: An MBA image is signed as standard ELF format.

For signing MBA in ELF wrapped MBN format, the following basic configuration should be used:

```
<image_type id='elf_wrapped_mbn'>
    <file_type>ewm</file_type>
    <ewm_properties>
        <image_entry>0x0C700000</image_entry>
        <relocatable>true</relocatable>
    </ewm_properties>
    <elf_properties>
        <has_hash_table>false</has_hash_table>
        <image_type>0</image_type>
    </elf_properties>
    <mbn_properties>
        <header_size>80</header_size>
    </mbn_properties>
</image_type>
```

For signing MBA in ELF format, the following basic configuration should be used:

```
<image_type id='elf_has_ht'>
   <file_type>elf</file_type>
   <elf_properties>
      <has_hash_table>true</has_hash_table>
      <image_type>0</image_type>
   </elf_properties>
</image_type>
```

4.2.4 Signing attributes configuration

4.2.4.1 Signer selection

SecImage supports a local signer, which uses QPSA test PKIs for image signing; a CSMS signer, which uses certificate and signatures from the CSMS server to sign images; and a CASS signer (from SecImage version 2.4), which uses CASS services to sign images. The default signer is the local signer.

4.2.4.2 Default attributes

The following signing attributes are specified in the configuration files for all images and can be overridden for individual image in the image list section. By default, the SecImage tool uses SHA-256 to generate the code signature.

- MSM_PART – Specify the MSM™ ID (or JTAG ID) described in Section 4.2.4.2.1.
- SOC_HW_VERSION – Specify the SOC_HW_VERSION described in Section 4.2.4.2.1.
- IN_USE_SOC_HW_VERSION – Specify whether MSM_PART or SOC_HW_VERSION should be used for HW_ID described in Section 4.2.4.2.1.
- OEM_ID – Specify the OEM HW ID described in Section 4.2.4.2.1.
- MODEL_ID – Specify the OEM MODEL ID described in Section 4.2.4.2.1.
- USE_SERIAL_NUMUBER_IN_SIGNING – Specify if serial number is used for HW_ID described in Section 4.2.4.2.1.
- DEBUG – Specify the DEBUG field described in Section 4.2.4.2.2.
- EXPONENT – Set the exponent used in attestation key generation. The supported values are 3 and 65537. The default value is 3.
- SW_ID – Specify the SW_ID described in Section 4.2.4.2.3.
- APP_ID – Specify the APP_ID described in Section 4.2.4.2.4. This field is optional. Use it only to sign TZExec.
- CRASH_DUMP – Specify the CRASH_DUMP described in Section 4.2.4.2.5. This field is optional.
- ROT_EN – Specify the ROT_EN described in Section 4.2.4.2.5. This field is optional.

4.2.4.2.1 HW_ID field

This field contains the 64-bit MSM_HW_ID value used in the HMAC to sign the image (see Section A.1), which comprises either Case A or Case B.

Case A

32 bits	16 bits	16 bits
JTAG ID (IN_USE_SOC_HW_VERSION=0) or SOC_HW_VERSION (IN_USE_SOC_HW_VERSION=1)	OEM HW ID CSMS account ID for the OEM 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
JTAG ID	SERIAL NUM

When JTAG_ID is used, the upper 32 bits denote the JTAG ID (MSM identifier) QFPROM fuse value with the upper 4 bits (bits 31 to 28) containing the Die Revision/Version generaled out. The

JTAG ID describes the hardware (i.e., the MSM8974 chipset) along with the variant information, and is blown (provisioned) by Qualcomm.

The SOC_HW_VERSION field is used in the newer family of processors to tie an image to a family of chipsets. Currently, only MSM8996 supports this feature. The IN_USE_SOC_HW_VERSION field is either 0 or 1 to indicate whether SOC_HW_VERSION is used (i.e., if it is 0, JTAG_ID is used). In MSM8996, even if SOC_HW_VERSION is enabled, XBL and MBA image are always signed with JTAG_ID.

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. USE_SERIAL_NUMUBER_IN_SIGNING field is either 0 or 1 to indicate whether Case A or Case B for HW_ID (i.e. if it is 1, serial number is used). Note that there is no serial number field in secimage.xml, user should update OEM_ID and MODEL_ID field accordingly.

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.

4.2.4.2.2 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 Application 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, 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 Qualcomm debug disable fuse values.

Since Qualcomm 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.

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

4.2.4.2.3 SW_ID field

This OU field contains the SW_ID value used in the HMAC to sign the image (see Section A.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 after a bug has been fixed and the fuses blown to contain the updated fixed version number. For example:

1. An OEM signs TrustZone with version 0.
2. Anti-rollback fuses for TrustZone are blown to 0.
3. A bug was discovered and fixed in TrustZone.
4. 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 will fail 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), which has been signed with a version of 0x2.

4.2.4.2.4 APP_ID field

This field is required for a TrustZone application, e.g., when SW_ID is TZ_EXEC_HASH_TABLE = 0xC. 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's) responsibility to keep the APP_ID list and ensure that different TrustZone applications do not share the same APP_ID.

4.2.4.2.5 CRASH_DUMP field

This field supports retail unlock feature. It enables logging for subsystem, such as UEFI, non-secure registers, HLOS, and encrypted modem EFS. It is an optional signing attribute that is read by TZ. For MSM8916 and later, SBL image needs to sign with this OU field and it will be ready by PBL.

The field comprises:

32 bits	32 bits
SERIAL NUM	ENABLE

The lower 32 bits controls enabling of the feature:

- Setting them to 1 means enabled.
- Setting them to 0 means disabled.

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

The behavior of setting CRASH_DUMP with DEBUG SETTINGS is explained in the table below.

DEBUG settings	CRASH_DUMP	Behavior
x	0	CRASH_DUMP is not enabled. Debug behavior is controlled by DEBUG Field [4.2.4.2.2]
0	1	Logs for security subsystems are disabled. Logs for subsystem, such as UEFI, non-secure registers, HLOS, and encrypted modem EFS are enabled.
2	1	Logs for security subsystems are disabled. Logs for subsystem, such as UEFI, non-secure registers, HLOS, and encrypted modem EFS are enabled.
3	1	This will enable dummy keys. As RPMB would have been provisioned with Production key, using this setting will block all reads/writes from RPMB as dummy key will now be used for all crypto operations. On Windows, as secure variables and other boot critical information is stored in RPMB, the device will fail to boot. On Android, all TZ APPS which rely on RPMB for their version control might fail to load. Device should boot up to HLOS without any issues. Logs for all security subsystems will be enabled.

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

4.2.4.2.6 ROT_EN field

This field supports ROT (Root of Trust) feature in MSM8996.

The field comprises:

32 bits	32 bits
SERIAL NUM	ENABLE

The lower 32 bits controls enabling of the feature:

- Setting them to 1 means enabled.
- Setting them to 0 means disabled.

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

4.2.4.3 Signer attributes

This section defines signer configuration and attributes for each signer. Starting from SecImage 2.4, local signer, CSMS signer, and CASS signer are supported.

OpenSSL instructions to generate SHA256 signed certificates can be found in Appendix B.

4.2.4.4 Postprocess

The postprocess configuration allows users to specify external commands to be run after the images are processed (i.e., signed and/or encrypted). Postprocess configuration is optional. In Android, the default postprocess configurations are used to call pil-splitter.py to split the signed images and facilitate loading into the hardware.

NOTE: Because pil-splitter.py is located within Meta build, the default postprocess operation is supported only when Meta build is provided as input (i.e., it will not work on individual image files).

4.2.4.5 Data provisioning

Data provisioning is introduced in Sectools v2.11, and is used to retrieve unified encryption keys. In `<chipset>_secimage.xml`, user should set the base directory for data provisioning. The default base path is set to use the test unified encryption keys in Sectools package.

```
<data_provisioning>
   <base_path>./../../resources/data_prov_assets/</base_path>
</data_provisioning>
```

4.2.4.6 Image list

The image list consists of all supported images and their configurations from previous configuration sections. Each image has the following attributes.

```
| -- image_list (a list of supported images with following attributes)
   | -- sign_id (unique ID to identify image signing configuration)
   | -- name (image name)
   | -- image_type (image format ID, defined in parsegen section)
   | -- cert_config (cert_config ID, defined in general properties section)
   | -- general_properties_overrides
   | -- pil_split (true/false, optional)
   | -- post_process_commands (optional)
   | -- meta_build_location (optional)
```

The user can modify attributes to override default configurations for each image. By default, <meta_build_location> is the relative image path from meta build; however, the user can update it with the absolute path for local image signing.

4.3 Integrity check

This operation is used to add hash table segment for ELF image.

Example command:

```
Sectools.py secimage –i <image_file> -c <config_file> -o <output_dir> -t
```

4.4 Verify input

After the user made modifications to the SecImage configuration file, it is recommended to verify it before performing an image operation.

To verify input arguments and configuration file:

```
Sectools.py secimage –m <meta_build> -p <chipset> -o <output_dir> -l
Sectools.py secimage –i <image_file> -c <config_file> -o <output_dir> -l
```

4.5 Local signing

NOTE: The local signer and the test PKI are provided by QTI as a test tool to demonstrate the certificate and signature format expected by secure boot. The Test PKI package is the same as the Qualcomm Platform Signing Application (QPSA) tool for backward compatibility. This tool uses OpenSSL to generate test keys and signatures, and is only to be used as a basic test/development tool until licensees develop their own signing tool. The test keys are not secure and are exposed during and after the signing. For production devices, licensees should obtain keys/certificates from the commercial Certificate Authority (CA) and protect keys within the Hardware Security Module (HSM).

1. To sign images using local signer with OEM-generated key for non-MRC signing, create folder in

```
sectools/resources/data_prov_assets/Signing/Local/<oem_presigned_certs>.
```

2. The folder name must be one word which is used as “selected_cert_config.”
3. Add pre-generated certificates and private key files in the folder.
4. Create config.xml with the following template and ensure the file names match:

```
<METACONFIG>
    <is_mrc>False</is_mrc>
    <root_pre>True</root_pre>
    <attest_ca_pre>True</attest_ca_pre>
    <attest_pre>True</attest_pre>
    <root_cert>test_rootca.cer</root_cert>
```

```
        <root_private_key>test_rootca.key</root_private_key>
        <attest_ca_cert>test_attestca.cer</attest_ca_cert>
        <attest_ca_private_key>test_attestca.key</attest_ca_private_key>
        <attest_cert>test_attestation.cer</attest_cert>
        <attest_private_key>test_attestation.key</attest_private_key>
    </METACONFIG>
```

5. In <chipset>_secimage.xml, user can update the following field with folder name:

```
<selected_cert_config>oem_presigned_certs</selected_cert_config>
```

Alternatively, user can override the cert config selection by add the following command line argument:

```
--cfg_selected_cert_config=oem_presigned_certs
```

4.6 CSMS signing

Chipset-specific configuration files are available in the config directory. User needs to update the config file to select CSMS signer.

```
<selected_signer>csms</selected_signer>
```

Alternatively, user can override the signer selection by add the following command line argument:

```
--cfg_selected_signer=csms
```

To sign one image using CSMS, follow these steps:

1. Extract the binary to be signed by running the following command in the SecImage directory.

```
Sectools.py secimage –i <image_file_path> -c <config_file_path> -s
```

2. In the output directory, a file named `<image_name>_tosign.mbn` is generated. Upload this file to CSMS to sign. You must use the correct signing attributes (e.g., SW_ID) when using CSMS.
3. If the signing is successful, place the zip file generated from CSMS into the cert folder in the output directory.
4. Inject the signature and the public certificates back to the image by running SecImage again using the same command as in step 1.
5. The signed file is stored in the output directory. The signed image will have the original image name.

4.7 CASS signing

Code Authorization Signing Services (CASS) is a Qualcomm operated signing engine, backed by secure cryptographic operations, made available to Signing Authorities (SA).

CASS is exposed as a web service to SAs, who will connect through strong factor hardware secured and PKI based authentication. Private signing keys are generated, stored, and accessed for operations on Hardware Security Modules (HSMs). Signing Authorities are responsible to determine whether code or content should be authorized to run on devices.

CASS, as an application, is designed to deliver:

- Online code signing
- With strongly authenticated users and applications
- Providing strong protection of both identity and code signing assets (keys)

Common failures and troubleshooting for CASS are described in Appendix C.

NOTE: Use of CASS signer will require additional license agreement and contract. Contact CASS team by raising case in Sales Force with below Problem Area:

BSP/HLOS(PA1)	Security(PA2)	Secure Boot(PA3)
BSP/HLOS(PA1)	Security(PA2)	CSMS(PA3)

4.7.1 Prerequisite

System requirements are:

- Windows 7 (32-bit or 64-bit)
- Linux
- JRE 1.8 or higher
- SafeNet eToken drivers

NOTE: For SafeNet eToken driver and any CASS installation questions, refer to [Q6] [Q8].

4.7.2 CASS configuration using Sectools

Chipset-specific configuration files are available in the config directory. User needs to update the config file to select CASS signer.

```
<selected_signer>cass</selected_signer>
```

Alternatively, user can override the signer selection by add the following command line argument:

```
--cfg_selected_signer=cass
```

In config file, user can provide token password as highlight in red. User will be prompted if the token password is left empty.

```
<cass_signer_attributes>
   <user_identity>
      <keystore_type>PKCS11</keystore_type>
      <token_password></token_password>
```

```
        ...
        </user_identity>
        ...
    </cass_signer_attributes>
```

Capability is mapped to a signing key with a set of restrictions. For signing key, CASS supports both SHA1 and SHA256 roots used by CSMS. Additional or different capabilities may be exposed to the OEMs.

secboot_sha2_root root hash (this should be the same root as CSMS SHA256 root):
7be49b72f9e4337223ccb84d6eccca4e61ce16e3602ac2008cb18b75babe6d09

The **secboot_sha2_root** root hash must be blown to OEM PK HASH to use this root for verification.

secboot_sha1_root root hash (this should be the same root as CSMS SHA1 root):
cc3153a80293939b90d02d3bf8b23e0292e452fef662c74998421adad42a380f

The **secboot_sha1_root** root hash does ***NOT*** need to be blown to OEM PK HASH. The root hash is used by default in the ROM.

```
<cass_capability>secboot_sha2_root</cass_capability>
```

Alternatively, user can override the CASS capability selection by add the following command line argument:

```
--cfg_cass_capability= secboot_sha2_root
```

4.8 Multiple Root Certificate (MRC) support

The Multiple Root Certificate (MRC) feature allows an OEM to select one from a maximum of 16 root certificates to be the trust anchor of the certificate chain that is used in signing and verifying the image. The root hash of a concatenation of the N number of root certificates will need to be blown to the OEM PK HASH e-fuse, where N varies from 1 to 16. One of the N root certificates can then be selected for the signing of the images. Later, an OEM can re-sign the images with a different root certificate selected if necessary. For details of the MRC feature, refer to [Q4].

All of the root certificates that were concatenated and used in the calculation of the root hash should be packaged in the signed image, so that a comparison can be made with the OEM PK HASH blown in e-fuse.

NOTE: CSMS and CASS do not support the MRC feature currently. Users will have to use “local” signer or their own signing tool to use the MRC feature.

4.8.1 Packaging of MRC test PKI

16 signed root and sub-CA certificates can be found under the following directories:

- resources/data_prov_assets/Signing/Local/mrc_sha256cert-key2048_exp3
- resources/data_prov_assets/Signing/Local/mrc_sha256cert-key2048_exp65537
- resources/data_prov_assets/Signing/Local/mrc_sha256cert-key4096_exp3

- resources/data_prov_assets/Signing/Local/mrc_sha256cert-key4096_exp65537
- resources/data_prov_assets/Signing/Local/mrc_certs_manual-key2048_exp3
- resources/data_prov_assets/Signing/Local/sha1_mrc_certs-key2048_exp3

They are named with the convention `qpsa_rootca<index>.cer` and `qpsa_attestca<index>.cer`, where `<index>` varies from 0 to 15.

4.8.2 Configuration parameters

In the general_properties, use `<num_root_certs>` to specify the total number of root certificates included in the image after signing. Supported values are from 1 to 16; the default is 1. User should select "mrc_sha256cert" and use <index> to specify which set of multiroot cert should be used. <index> value is 0 to 15.

```
<general_properties>
...
    <selected_cert_config>mrc_sha256cert</selected_cert_config>
...
    <num_root_certs>16</num_root_certs>
    <mrc_index>0</mrc_index>
...
</general_properties>
```

Another MRC cert configs is "mrc_certs_manual," which can be used to manually pick MRC test PKI.

To run MRC signing without modifying the config xml, user can also pass the override parameters. The following command line argument is an example:

```
--cfg_selected_cert_config=mrc_sha256cert --cfg_num_root_certs=16 --
cfg_mrc_index=0
```

4.8.3 Configuring root hash

To support the multiple root certificates feature, users need to blow the hash of the concatenated root certificates used to OEM PK HASH e-fuse. For example, if the number of root certificates used is 16, users can find out the hash using the following commands in the Windows command prompt below:

Step 1: Concatenate 16 root certificates into qpsa_16_roots.bin.

```
copy /B
qpsa_rootca0.cer+qpsa_rootca1.cer+qpsa_rootca2.cer+qpsa_rootca3.cer+qpsa_ro
otca4.cer+qpsa_rootca5.cer+qpsa_rootca6.cer+qpsa_rootca7.cer+qpsa_rootca8.c
er+qpsa_rootca9.cer+qpsa_rootca10.cer+qpsa_rootca11.cer+qpsa_rootca12.cer+q
psa_rootca13.cer+qpsa_rootca14.cer+qpsa_rootca15.cer qpsa_16_roots.bin
```

Step 2: Generate SHA256 hash from the concatenated roots (qpsa_16_roots.bin).

```
openssl dgst -sha256 qpsa_16_roots.bin >sha256_16roots_hash.txt
```

The hashes for one root, four roots, and 16 roots are pre-calculated and included in the following files: resources/data_prov_assets/Signing/Local/mrc_certs_manual-key2048_exp3.

```
sha256_root0_hash.txt (hash for qpsa_rootca0.cer)
sha256_4roots_hash.txt (hash for root 0 to root 3 concatenated)
sha256_16roots_hash.txt (hash for root 0 to root 15 concatenated)
```

4.9 Image validation

Image validation can be used to check if an image is signed and if signed, whether the signed image is valid. User needs to provide image files and corresponding configuration files.

- For an unsigned image, SecImage will return the image as unsigned and print the image type and its default image structure from the configuration file.
- For a signed image, SecImage will extract signature and certificate chain to verify if the image hash match and certificate chain is valid.
- From Sectools v3.15 and later, if root cert hash value is given, SecImage can validate if a signed image was signed with that root cert.

4.10 Image encryption

To support SSD and secure PIL, ELF image unified encryption is supported in SecImage tool 2.4. The image decryption on the device will be done by Boot code during target startup.

The SecImage tool is compliant with the unified image encryption and SSD encryption.

NOTE: SSD encryption can be applied on unsigned images to support OTA upgrade on non-secure devices. Unified encryption shall only apply to signed images. However, encryptions only on signed image is not supported as after adding encryption parameters, signature of the signed image does not match anymore (i.e., in order to encrypt a signed image, re-sign and encryption should be done together, which is "-se" operation for SecImage).

4.10.1 SSD encryption

SSD encryption is supported for MSM8916, MSM8939, MSM8909 and MDM9x45 as legacy encryption for OTA.

4.10.1.1 Select SSD encryption

In `<chipset>_secimage.xml`, edit encryptor to use SSD encryption.

```
<selected_encryptor>ssd_encryption</selected_signer>
```

Alternatively, user can override the encryptor selection by add the following command line argument:

```
--cfg_selected_encryptor=ssd_encryption
```

4.10.1.2 Configuration

Step 1: Generate the OEM encryption keys: Run the following command under

```
/sectools/features/isc/encryption_service/ssd/ssd:
gen_keys.py --key_dir=keys
```

This generates the following folders that contain the encryption keys generated for the corresponding algorithm.

/sectools/features/isc/encryption_service/ssd/keys/dvc_aes

/sectools/features/isc/encryption_service/ssd/keys/dvc_rsa

/sectools/features/isc/encryption_service/ssd/keys/oem_rsa

The production encryption keys do not overwrite the test encryption keys, which are named test_dvc_aes, test_dvc_rsa, and test_oem_rsa, respectively.

Step 2: Modify sectools/features/isc/encryption_service/ssd/ssd/key_config.xml.

Update the <path> and <id_path> of each key to point to the path of the generated key. Update the key type to "prod" instead of "test" depending on whether the secure boot fuse is blown. The "test" keys are used when the secure boot fuse is not blown, while the "prod" keys are used when the secure boot fuse is blown. The script also checks if the keys are the same as the test keys and generates an error if they are the same. These restrictions are a security feature to avoid using test keys in a production environment and/or leaking production keys accidentally. These safeguards cannot be relaxed without compromising one of the above mitigations.

An example of the modified key name is shown below in red:

```
<key name="rsa device private" type="prod"
  <path>keys/dvc_rsa/rsa_pkcs8_pr_key.der</path>
  <id_path>keys/dvc_rsa/key_id.dat</id_path>
</key>
```

Step 3: Updating keystore with OEM generated encryption keys.

Once OEM encryption keys are generated and the `key_config.xml` is updated, users can use the following command under `/sectools/features/isc/encryption_service/ssd/ssd` to update the keystore.

```
gen_keystore.py
```

The keystore.dat can be found under

`/sectools/features/isc/encryption_service/ssd/ssd/` keys directory.

Step 4: Selecting the key encryption algorithm.

Two key encryption algorithms are supported by SSD: RSA 2048-bit and AES 128-bit. The algorithm can be selected by modifying

`sectools/features/isc/encryption_service/ssd/ssd/ssd_bin.cfg`.

The default configuration is the RSA 2048-bit algorithm as shown below:

```
IEK_ENC_ALGO:RSA-2048
IEK_ENC_PADDING_TYPE:PKCS#1-V1.5
```

If the AES 128-bit algorithm is preferred, change the above configuration to the following:

```
IEK_ENC_ALGO: AES-128
IEK_ENC_PADDING_TYPE:NULL
```

4.10.2 Unified encryption

Unified encryption is introduced in MSM8994 and will be used for future targets image encryption.

4.10.2.1 Select unified encryption

In `<chipset>_secimage.xml`, edit encryptor to use unified encryption.

```
<selected_encryptor>unified_encryption_2_0</selected_signer>
```

Alternatively, user can override the encryptor selection by add the following command line argument:

```
--cfg_selected_encryptor= unified_encryption_2_0
```

4.10.2.2 Configuration

If user would like to use OEM-generated encryption keys, add the encryption keys in resources/data_prov_assets/Encryption/Unified/<chipset>.

In the `config.xml`, user shall update the key files name in red (i.e., define key file mapping).

```
<METACONFIG>
   <l1_file_name>l1_key.bin</l1_file_name>
   <l2_file_name>l2_key.bin</l2_file_name>
   <l3_file_name>l3_key.bin</l3_file_name>
</METACONFIG>
```

Unified encryption key should be 16 bytes binary file. Test key files can be found in `/resources/ data_prov_assets/Encryption/Unified/<chipset>/.`

4.11 Example commands

- To find all supported chipsets:

```
secools.py secimage -p LIST -h
```

- To find all sign_id for a given chipset:

```
secools.py secimage -p <chipset> -g LIST -h
```

- To find all parameters for override:

```
secools.py secimage --overrides -h
```

- To sign all images from meta build:

```
secools.py secimage –m <meta_build> -p <chipset> -o <output_dir> -s
```

- To sign all images from meta build and validate the signed image:

```
secools.py secimage –m <meta_build> -p <chipset> -o <output_dir> -sa
```

- To sign and validate an individual image:

```
secools.py secimage –i <image_file> -c <config_file> -o <output_dir> -sa
```

- To sign and encrypt all images from meta build:

```
secools.py secimage –m <meta_build> -c <config_file> -o <output_dir> -se
```

- To sign and encrypt all images from meta build and validate the encrypted image:

```
secools.py secimage –m <meta_build> -c <config_file> -o <output_dir> -sea
```

- To sign/encrypt/validate an individual image from meta build:

User can use sign_id to process a single image from meta build (i.e., if the user does not need to know the image path, Sectools will automatically use sign_id to find the image path).

```
secools.py secimage –m <meta_build> -p <chipset> -g <sign_id> -sea
```

- To validate an individual image (no operation on the images):

```
secools.py secimage –i <image_file> -p <chipset> -a
```

- To validate an individual image with root cert hash (no operation on the images):

```
secools.py secimage –i <image_file> -p <chipset> --rch=<hash> -a
```

4.12 How to debug

User can use "-v" to increase the verbosity level of the log file:

```
secools.py secimage –m <meta_build> -p <chipset> -o <output_dir> -sa –d –v
```

With "-d," SecImage will also save intermediate image files and certificates for debugging. It can be found in `<output_dir>\<hardware>\<software>\debug`.

4.13 Error and warning

- `"WARNING: File not found in meta build: xxx"`

  Image is not found in meta build contents.xml. Some images are exclusive for LA or WP, but <chipset>_secimage.xml can be shared for multiple PLs. It is expected if user is running an LA build and received warning that WP image is not found.

- `"WARNING: OEM ID is set to 0 for sign_id "xxx""`

  Security tools prompt warning message for zero OEM ID. In the future (depending on secure boot requirement), security tools may change this warning behavior to report error and exit (i.e., stop signing and/or encryption).

- `"WARNING: Following overlapping segments were found: xxx"`

  It is required by Secure BOOT team that security tools should check and prompt warning for overlapped ELF segments.

- `"ERROR: The input plaintext is less than the minimum."`

  One or more ELF segments is <16 bytes. This is encryption limitation. User may pad zero to the ELF segment to solve the issue.

- `"ERROR: Cannot encrypt a signed unencrypted image without resigning as the sign no longer matches the format change."`

  To encrypt a signed image, image must be re-signed and encrypted at the same time, run with "-sea" (i.e., otherwise signature will be invalid).

- `"ERROR: No module named meta_lib"`

  `"-m"` can only be used with meta build; for any other build, specify the image path with `"-i <image_file_path>"`.

- `"ERROR: Following validations failed for the image"`

  `"Following signing attributes do not match"`

  Above error message may be shown with image validation operation. A table may be printed for mismatch signing attributes in config xml vs. certficiate chain.

- `"Root certificate from image does not match the given root cert hash value"`

  Above error message may be shown with image validation operation. This message means image was not signed with the given root cert.

A MSM_HW_ID and SW_ID

A.1 MSM_HW_ID

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

Table A-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 0xFFFF_FFFF
or
SOC_HW_VERSION masked with 0xFFFF_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 particular MSM.

Once the 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 QFPROM_RAW_OEM_SEC_BOOT_ROW register. If this bit is blown, the 32-bit SERIAL_NUM will be used instead of the OEM_ID in the authentication.

NOTE: In the QFPROM_RAW_SERIAL_NUM register, the software should read QFPROM_CORR_SERIAL_NUM_LSB, instead of SERIAL_NUM.

Compared to the previous platforms, all 32 bits of the 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.

NOTE: To view appropriate values for MSM_REVISION_ID, refer to [Q3].

OEMs can also use SERIAL_NUM, instead of OEM_ID, as described in Section 4.2.4.2.1.

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 Qualcomm. The model on the signing website must be the same as the OEM_PRODUCT_ID, which is allocated for each product by the licensee.

A.2 Software ID

The software ID has changed in the MSM8974 secure boot architecture to support more images. The bits in the SW_ID OU field in the signed image must match the following code.

For 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 [Q7].

```
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 */
PMIC                = 0x16 /* PMIC image */
SLPI                = 0x18 /* Sensor Low Power Island */
EOS                 = 0x19 /* EOS firmware image */
VIP                 = 0x1A /* Validated Image Programmer */
```

In the MSM8974 chipset, the modem is reset by the HLOS, but the modem has its own PBL, so the MBA image is actually authenticated by the modem PBL (for more details, refer to [Q7]). 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).

NOTE: SW ID from 0xFFFF0000 to 0xFFFFFFFF is reserved for customers.

B Generating SHA256 Signed Certificates

Users can place the keys and certificates in a file directory and use the configuration to select the OEM-generated PKI. Detailed instructions can be found in Section 4.2.4.3. Also, based on the generated root certificate for this process, you should be blowing the SHA256 value of your 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. Refer to [Q4] for further details.

Licensees can use the following OpenSSL commands to generate their own private/public key and certificates. However, it is not recommended to use these keys and certificates for production signing. The `/resources/openssl` has the sample opensslroot.cfg and v3.ext files.

To 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 –sha256
```

NOTE: *-sha256* should be removed at the end of the command when generating a SHA1 signed root certificate.

```
openssl x509 -in oem_rootca.crt -inform PEM -out qpsa_rootca.cer -outform
DER
```

To generate an attestation CA private key:

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

To 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
```

To 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 –sha256
```

NOTE: *-sha256* should be removed at the end of the command when generating a SHA1 signed sub-CA certificate.

```
openssl x509 -in oem_attestca.crt -inform PEM -out qpsa_attestca.cer -
outform DER
```

C Troubleshooting CASS Error

CASS failures are classified by type. The following table details the error types and suggests possible error sources.

Failure message	Potential failure sources
FAILURE: ACCESS NOT AUTHORIZED	▪ The Signing Authority presented to the server is not authorized for the selected signing capability. ▪ The capability requested is incorrectly configured.
FAILURE: SIGNATURE PACKAGE INVALID OR NOT FOUND	Signing attributes are configured incorrectly.
FAILURE: UNABLE TO ACCESS KEYSTORE	▪ The hardware token is unreachable. ▪ The token password is wrong. ▪ The keystore type is configured incorrectly.
FAILURE: UNABLE TO COMMUNICATE WITH SERVER	▪ Network connectivity failures ▪ If using the internal Qualcomm Wi-Fi network, connect through VPN. This problem is due to internal routing issue.
ERROR: Exception in thread "main" java.lang.NoClassDefFoundError: sun/security/pkcs11/SunPKCS11	▪ Run “java -version” and check that JRE 32 bit 1.6 or above version is installed and in the path environment variable. ▪ Run “which java” to make sure the path points to the correct version. If it points to c:\Windows\system32\java, remove the java binary from c:\Windows\system32 and add the explicit path, such as C:\Program Files (x86)\Java\jdk1.6.0_31\bin to the path environment variable.

Occasionally, a failure in execution while the application is conducting a transaction (an eToken) will result in the application failing to access the eToken in subsequent attempts.

Potential resolutions include:

1. Physically remove the eToken USB HSM from the USB port, and then re-connect the HSM to the USB port.
2. If access to the eToken still hangs in a Windows environment, install the Smart Card patch for Windows from: http://support.microsoft.com/kb/2427997.

D OU Fields in Certificate

The following is a list of organizational unit (OU) fields defined for the attestation certificate. The details of each field is described in Section 4.2.4.2.

OU 01 = SW_ID (mandatory, varies per image)

OU 02 = HW_ID (mandatory)

OU 03 = DEBUG (mandatory, default 0x0000000000000002)

OU 04 = OEM_ID (mandatory, default 0x0000)

OU 05 = SW_SIZE (mandatory)

OU 06 = MODEL_ID (mandatory, default 0x0000)

OU 07 = SHA256 (0000 for sha1, 0001 for sha256)

OU 08 = APP_ID (optional, for TZapps images, varies per image)

OU 09 = CRASH_DUMP (optional)

OU 10 = ROT_EN (optional, 0 or 1)

OU 11 = SOC_HW_VERSION (optional, for MSM8996 and later)

OU 12 = MASK_SOC_HW_VERSION (deprecated)

OU 13 = IN_USE_SOC_HW_VERSION (optional, 0 or 1, use with SOC_HW_VERSION)

OU 14 = USE_SERIAL_NUMUBER_IN_SIGNING (optional, 0 or 1)