Qualcomm Technologies, Inc.

Qualcomm® Mobile Security Hardware Token Integration Guide

80-P9301-147 A

January 22, 2018

Revision history

Revision	Date	Description
A	January 2018	Initial release

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document is for software engineers and system integrators needing detailed information to integrate the Qualcomm Mobile Security (QMS, formerly known as Qualcomm Haven) hardware token (formerly known as Haven token) device attestation into their application. It addresses both the client device side and the server side.

This document uses the term Independent Software Vendor (ISV) to refer to the author of the app that calls the QMS hardware token API, and that implements the server that decodes hardware tokens.

1.2 Conventions

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

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

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

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

1.3 Technical assistance

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

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

2 Token generation system overview

The general flow of the device attestation process begins with an Android application calling the QMS hardware token Java APIs to create a hardware token. The call will return a binary blob of encrypted data that is typically around 1KB in size, but will increase if the caller chooses to add their own data.

The Android app then sends this data to the server that it talks to, using a protocol such as HTTP, or including a hardware token in an HTTP header. At the server, the token must be decrypted, and then its signature verified. The QMS token verification tool provided by QTI decrypts and verifies the signature, making use of OpenSSL libraries. The tool includes the QTI public root certificate and public attestation certificates needed for signature verification.

Once decrypted and the signature verified, the token is stored as a concise binary object representation (CBOR) formatted binary file, or can be converted to JSON if desired (See Appendix A for more information on hardware token format).

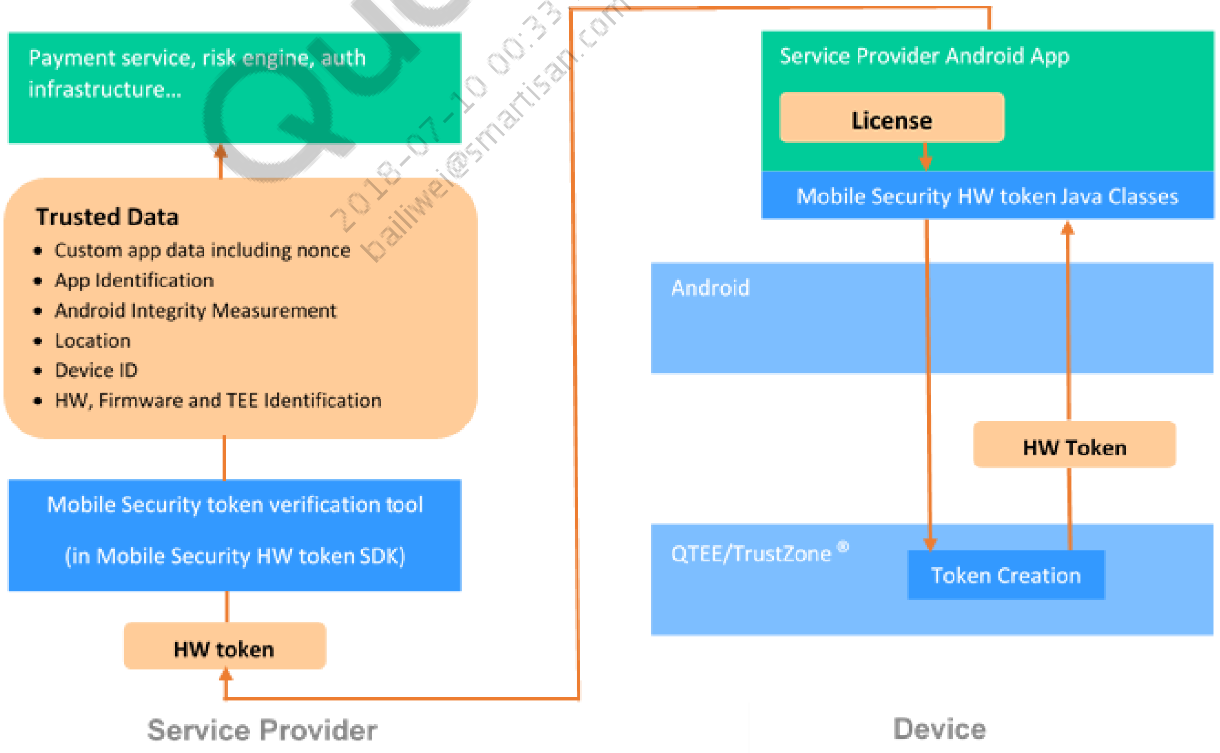
Payment service, risk engine, auth infrastructure...
Trusted Data
Custom app data including nonce
App Identification
Android Integrity Measurement
Location
Device ID
HW, Firmware and TEE Identification
Mobile Security token verification tool
(in Mobile Security HW token SDK)
HW token
Service Provider
Service Provider Android App
License
Mobile Security HW token Java Classes
Android
HW Token
QTEE/TrustZone ®
Token Creation
Device

Figure 2-1 Token generation system overview

This document is oriented around an Android app, but there are QMS hardware token implementations for other operating systems. In those cases, the type of app calling the API depends on what the operating system supports. It is also possible to call the API from a trusted application (TA) running in the Qualcomm® Trusted Execution Environment (QTEE).

3 QMS keys and certificates

QMS hardware token makes heavy use of public key cryptography. This appears in the form of X.509 certificates containing public keys, and their corresponding private keys. It is common to use the OpenSSL command line utility to create these keys and certificates.

The three public key hierarchies involved are:

1. Standard Android application signing
2. Cryptographic activation certificate (license certificates) and corresponding decryption keys
3. Attestation certificates and keys

3.1 Description of key uses

3.1.1 Android application signing keys

All Android APKs that call the QMS hardware token APIs must be signed in the same way as any standard Android APK. The only difference is the additional step of supplying QTI with a hash of the ISV's X.509 app signing certificate to be included in the issued license certificate. Android signing certificates do not chain up to any root certificate.

3.1.1.1 License certificates

To access the QMS hardware token API, the ISV must obtain a license certificate from QTI. This license certificate is an X.509 certificate signed by QTI with a few special extensions. It must be passed to the API to unlock it for use. The ISV owns the corresponding RSA private key for the license certificate (as well as the private key for signing Android apps).

Instructions on obtaining the license certificate is provided in detail in the following section, Obtaining a License Certificate from QTI.

These license certificates are always subordinate to the QTI-owned license attestation root.

3.1.2 Device attestation certificates

Device attestation certificates are used by the QMS token verification tool to verify the hardware token signatures. Generally, the ISV will not have to handle these certificates. Instead, they are used internally when a token is verified. The certificates are currently bundled with the SDK for the SDM630, SDM660 and SDM845 chipsets. For future devices, it may be looked up by a network connection to QTI, or included with the token.

Device attestation certificates are subordinate to a QTI-owned attestation root. ECDSA cryptography is used as opposed to the RSA cryptography used for licensing.

4 Obtaining a license certificate from QTI

4.1 Android software developers

ISVs calling the QMS Java APIs (e.g. for application security or hardware token) must provide two things for QTI to create a license certificate for them. The first is a certificate signing request (CSR) for an RSA key pair that they have generated and own. This will become their license certificate, used to unlock the QMS hardware token APIs. The private key will be used to decrypt hardware tokens. Second, the ISV must provide the fingerprint(s) of the app signing certificate(s) associated with the private key(s) used to sign their Android APKs. The license certificate will be bound to apps that are signed with the ISV's app signing key.

To be clear, the app developer needs two separate RSA key pairs, one that is a standard Android application signing key and another that licenses QMS features. Developer's using QMS application security will need three key pairs as described in the section, Application Security Developers, below.

4.1.1 License certificate and key

The key pair should be 2048-bit RSA with exponent f4 (65537). The following text describes how to generate this with OpenSSL, but any software capable of generating the key pair and CSR may be used. If using OpenSSL, it is preferred to use version 1.0 or higher. The following command can be used to check the version:

```
openssl version
```

The ISV should protect their private key (or keys) in a reasonable way. One way to do that is to password protect the file holding the private key, as supported by OpenSSL's key generation method.

Generate the key and CSR with this OpenSSL command:

```
openssl req –newkey rsa:2048 -keyout <isv_key_name>.pem  \
-out <isv_csr_name>.pem
```

There will be a prompt for a passphrase, which should be a secure passphrase. The file will be encrypted and the passphrase must be given every time the key is used. To protect the private key by other means, use the –nodes option.

There will also be prompts for the country, state, company name, and other information. These can be left blank, except for the Organizational Name. Ensure that a clear and correct name is provided here.

The `<isv_csr_name>.pem` file should be sent to the ISV's contact at QTI via email. Often certificates will be stripped or blocked as attachments to emails because they are considered dangerous. However, this can be avoided by zipping it. QTI will confirm the hash of this file through a voice call, and then return the license certificate when it is ready. The OpenSSL command to generate the hash to be verified is as follows (The hash can be over the CSR in either PEM or DER format):

```
openssl dgst –sha256 –c <isv_csr_name>.pem
```

`<isv_key_name>.pem` is the file with the ISV's private key that should be preserved and protected.

4.1.2 Fingerprint of app signing certificate

For QTI to issue a license certificate, the SHA-256 fingerprint of the app signing cert is required, which is the SHA-256 hash of the DER-formatted X.509 app signing certificate. This can be obtained using keytool with the –list –v options as follows:

```
keytool -exportcert -list -v -alias <isv-key-name> \
-keystore <path-to-keystore>
```

`<path-to-keystore>` is the file path name where the keystore containing the certificate is. For debug keys, the default is `~/.android/debug.keystore`. For release keys, the location is usually specified by the user.

`<isv-key-name>` is the alias name of the key in the keystore that was specified during key generation. The default for debug keys is `androiddebugkey`.

The part of the output that must be sent to QTI looks like this:

```
SHA256:37:B7:E3:7D:AC:D7:99:7D:4B:C4:D6:C2:8B:6F:8F:31:21:5E: \
9F:D1:AA:70:4F:BC:64:4E:EA:4F:FF:D4:93:19
```

However, the hex digits of the output will be different.

Note that this is almost identical to what is required to sign in to Google Play services via OAuth2. The main difference is that QMS uses the SHA-256 hash instead of the SHA-1 hash. Google describes the process here:

https://developers.google.com/android/guides/client-auth

More information about how Android signing works, how to use the tools, how the key aliases and keystore paths work etc. can be found here:

https://developer.android.com/studio/publish/app-signing.html

More information about keytool is provided here:

https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html.

If the app signing certificate is not in keytool , the following OpenSSL command can be used to get the fingerprint:

```
openssl x509 –in <isv_cert_file> -fingerprint –sha256
```

The Keychain access app on MacOS also displays the SHA-256 fingerprint.

If several keys are used for signing Android apps, then several fingerprints can be sent to QTI and bound to the license. However, it is best to avoid providing the fingerprints of debug certificates with unsecured keys, as that will allow the license to be used with any debug signed app.

4.2 Application security developers

Application security has a training mode that provides access to features that should not be enabled on commercial devices. Developers are issued a separate license for this mode. Therefore, an application security developer will use the following three RSA key pairs: one for the QMS commercial license certificate, one for the application security license certificate, and one for their Android app signing key.

They should send the following to QTI:

1. CSR for commercial license certificate
2. CSR for application security training license certificate
3. SHA-256 fingerprint of the Android app signing key.

4.3 Device OEM SW developers

Device OEMs wishing to use QMS features must also have a license certificate and key. The process for obtaining the license is the same as described in the section for Android SW developers.

OEMs writing TAs that run in TrustZone, or system apps running in Android do not need to send a fingerprint of the application to QTI. Instead, they need to ensure that the correct OEM ID is configured into the device fuses. More information can be found in the section on the OEM ID.

If the OEM is writing non-system Android apps that are signed like typical Android apps, then they should follow the entire procedure detailed in the Android SW developer section.

5 QMS SDK

This section lists the major parts of the QMS SDK and what they are used for.

5.1 Java API classes

The hardware token Java API classes, as part of the QMS Java API, get packaged and included with the developer’s application. The token APIs are found in the `tokenapi` directory. Note that the APIs found in the `appapi` and `csapi` directories are not related to hardware token.

5.2 Demo application

The demo application, `qms_demo`.apk, is a runnable binary application signed by QTI. The source code for this is in the app directory. To compile and modify this application, the developer must use their own Android application signing key. The developer must also have their own license certificate. The license that is used by the QTI signed application will not work.

5.3 QMS token verification tool and certificates

The QMS token verification tool, `qms_tvt`, is located under the `tools/token_server` directory. This is python code for use on a server to decrypt and verify tokens, as well as translate them to JSON, if desired.

6 Calling the QMS hardware token API

Hardware tokens can be created using a Java API for Android apps, or a C API for TAs running in QTEE. A native C API for Linux Android is not yet supported. The Java classes are supplied in the QMS SDK. The C headers for calling from a TA, however, are supplied with QTEE as part of the chipset SW; they are not included in the SDK. The C headers are only available to OEMs and Advanced Mobile Subscriber Station (AMSS) licensees.

The general process for token creation is to first call the `startInstance()`method, then optionally add application data to the token through various methods, and finally call the `finish()` method to return the encrypted hardware token. The rest of this section describes each step in more detail.

6.1 Starting token creation

The `startInstance()`method sets up a context for creating a token. This context will exist until `finish()`is called, the Java instance is destroyed, or the app in which the Java instance exists is destroyed. Only a limited number of contexts are supported.

The first argument of the `startInstance()` method is the Android context, obtained by calling `getApplicationContext()`, or `getContext()`.

The second argument selects the attestation key used to sign the token. It is one of the `HAVENT_KEYTYPE_XXX` integers. The type and strength of the attestation key varies from device to device, and a device typically only supports one or two key types. The type that is supported depends on the mobile processor the device uses.

All devices support `HAVENT_KEYTYPE_DEMO`. It is available on all chips that support hardware token. It is available whether or not secure boot is enabled. It is a global key that is constant and always the same. They key is not secured. The primary purpose of this key is for demonstration and testing as discussed in the Demo Signing Key section.

It is best to use the strongest signing key that the device supports. On the SDM845 chipsets and newer devices, the `HAVENT_KEYTYPE_BEST_AVAILABLE` option will select the most secure key type available on the device. If a key type that is not supported by the device is selected, a Java exception with code 11 (`ERROR_SIGNING_KEY_UNAVAIL`) will occur.

The third argument adds option flags. The following table describes the available options. This argument can be zero if no options need to be set.

Table 6-1 Available options for token creation

Option name	Description
`HAVENT_OPT_SIMDATA`	This option causes the token to consist of simulated data that never changes and is the same across devices. More details can be found in the Simulated Data section
`HAVENT_OPT_APPCERT`	This causes the full Android application signing certificate to be included in the token. The certificate is usually 1KB or more, so this will double the size of the typical token. Note that the fingerprint (the SHA-256 hash) of this certificate is always included in tokens and can often be used instead of the certificate to identify the signer of the app.
`HAVENT_OPT_LOCATION`	This causes the GNSS location to be included in the token. The location included is a secure location that cannot be spoofed or changed by Android. See the Trusted Location section for more details
`HAVENT_OPT_RTIC_CURRENT`	This will include the latest cached rootkit detection report in the token. It does not trigger a rootkit attestation check. See the Real-Time Rootkit Detection section for more details.
`HAVENT_OPT_LONGRTIC`	This will include the rootkit report for an attestation check that is run at token creation time. This option slows down token creation.
`HAVENT_OPT_CIPHER_SUITE_1`	The cipher suite controls the encryption format. Currently, the only supported format is Cryptographic Message Syntax (CMS) described in RFC 5652. The session key is always AES 256 CBC and the asymmetric key is RSA 2048 bit. This option does not need to be provided because it will be selected automatically.

6.2 Adding user data

This section will discuss adding user data to the hardware token. This step is completely optional, and can be skipped if not needed.

The caller can add their own data to the token that will be signed and encrypted just like the rest of the data in the token. The API includes methods to add simple data types (e.g. longs, strings, bytes) under a specific label in the map-like CBOR formatted token, as well as methods to include structured data (e.g. maps, arrays) and more complex data types (e.g. dates, floats) within the CBOR data type.

Please note that the CBOR standard refers to its map structure as *key*/value pairs, whereas this document will refer to it as *label*/value pairs to avoid confusion with cryptographic keys.

6.3 Adding simple data

When adding very simple data, use the methods for adding integers, strings, or binary data. In the CBOR standard, text strings, and byte strings are expected to be US-ASCII or UTF-8. It is recommended that the labels be short ASCII strings, although UTF-8 is also acceptable.

This following example in Java creates a token with a string whose value is "this is a string" and whose label is "string-label", and an integer whose value is 4839 and whose label is "integer-label".

```
Token token = Token.startInstance(ctxt, Token.HAVENT_KEYTYPE_DEMO,
                                0, sLicenseCert);
  token.addString(”string-label”, “this is a string”);
  token.addLong(”integer-label”, 4839);
  byte[] tokenBytes = token.finish();
```

If the token is translated to JSON, the translation tool will determine what is done with binary data items. Often, they are base-64 encoded.

6.4 Adding structured data

To structure data with maps or arrays, use the `startMap()` / `endMap()` and `startArray()` / `endArray()`methods to create the CBOR structure.

Maps consist of label / value pairs. When adding integers, text strings, or byte strings to maps, a label string must be provided. An array is an ordered collection of values with no labels. When adding to an array, NULL must be provided as the label.

Here is an example or creating an array and a map. The array is a list of three names for an account. The map describes a transaction.

```
Token token = Token.startInstance(ctxt, Token.HAVENT_KEYTYPE_DEMO,
                                0, sLicenseCert);
  token.addString(”AccountHolder”, “Joseph Modeliste”);
  token.startArray(“AccountNames”);
  token.addString(NULL, “slush fund”);
  token.addstring(NULL, “rainy day”);
  token.addString(NULL, “extra”);
  token.endArray();
  token.StartMap(“Transaction”);
  token.addLong(“Amount”, 123);
  token.addString(“Vendor”, “Acme Services”);
  token.addString(“Date”, “2017-04-23T18:25:43.511Z”);
  token.EndMap();
  byte[] tokenBytes = token.finish();
```

It is possible to nest maps within maps, maps within arrays, arrays within maps, and arrays within arrays. There is a nesting limit of ten, and going beyond this will produce an error. It is important to nest the calls correctly, otherwise the CBOR formatting will be incorrect. Consistency checking of this is not supported and so an error may not be produced, however the token will not be parse-able on the server side.

6.5 Adding complex structured CBOR data

If the data is complicated or contains CBOR types that are not supported by the simple methods described so far, a CBOR encoder, such as one of the open source Java CBOR encoders, can be used to create pre-encoded data that can be added to the token.

CBOR supports a variety of types beyond simple integers, text strings and byte strings. These include date formats, floats, Boolean values and others. To add data of these types, use a full CBOR encoder to generate raw CBOR, then add it to the token with `addCBORMap()` or `addRawCBOR()`. The same can be done to add more complicated CBOR structures that might be too cumbersome to create using QMS hardware token's simple interface.

The `addCBORMap()` method is the safest. It assumes that the CBOR map being added is well formed, and adds it to the token under the specified label. The `addRawCBOR()` method concatenates its input directly in the token, meaning the caller will have to carefully think about map nesting and labeling, to create a valid CBOR map.

There is no parsing or validation performed on the inputs of these methods. An error may not be returned when creating the token if a mistake is made, and the CBOR will not be parse-able on the server side. Note that incorrect CBOR will not crash the device, make QMS hardware token malfunction, or create any security holes, so it is safe to experiment.

6.6 Adding non-CBOR data

Non-CBOR data can be added by putting the data in either a text string or a byte string. The data will be carried in the CBOR formatted token, but it will not be changed in any way. The server side code will have to know how to identify and handle this data correctly. For example, a binary DER-encoded X.509 certificate can be put into a byte string, or encoded JSON can be put into a text string. CBOR also has tagging facilities for data like this, however, they are not directly supported in the QMS hardware token API. To use them the caller will have to encode the raw CBOR data and add it using methods described previously.

6.7 Size limit and encoding process

The amount of user data that can be added is limited to 64KB. If exceeded, the error `ERROR_TOO_MUCH_APPDATA` is returned. Labels and CBOR encoding overhead for user data are counted in this 64KB, but the CBOR encoding overhead is generally small (about 3 bytes or less per item). As an example, adding a 1KB blob of data with a label of 10 bytes will count for 1,014 bytes against the 64KB limit. Data items like trusted location and rootkit detection reports that are not user-added do not count against the 64KB.

As each data item is added with the methods described in this section, they are copied into an internal linked. When `finish()` is called, the data in the linked list is formatted into CBOR and the linked list is destroyed.

6.8 Security levels

All the data items within the token have an associated security level, discussed further in the Working with the Data Items section. The HLOS Java API does not include a way of setting the security level for user added data. Instead, all data items added from the HLOS are classified at the lowest security level since they are subject to compromise if the HLOS is compromised.

When adding data with the C API in a TA, however, it is possible to specify the security level of the added data with the integer security level parameter. The most secure level that can be specified in the TA is that of a TEE user (`HAVENT_SIG_SEC_LEVEL_TEEU`). The TA can also add data of a lower security level in the event that it is getting the data from a less secure source. For example, if the TA is adding data that it received from the HLOS, it should specify the Rich OS user security level (`HAVENT_SIG_SEC_LEVEL_RICHOSU`) for that data.

6.9 Completing the token

The final step is to call the `finish()` method to obtain the completed token. Upon making the call, all non-user added data items are included, and the token is signed and encrypted. All cryptographic operations for signing and encrypting occur in the TEE during the `finish()` call.

It is important that the token is not decrypted on the device as this would require the ISV to store their license key on the device. Doing so puts the key at risk of capture and reduces the security level of the data to that of the HLOS, greatly devaluing the token.

7 Data item specific information

This section provides information on data items within the token that require special attention.

7.1 Real-time rootkit detection

QMS real-time rootkit detection is a scheme where the integrity of the Linux kernel is checked to determine whether it has been modified by a rootkit or malware.

The rootkit report in the token includes the following fields:

Table 7-1 Rootkit report

integrity	`True` means the check passed, `false` that it didn't pass, and `null` that it was unable to measure the integrity
flags	Flags passed when report was generated (e.g. `HAVENT_OPT_RTIC_LONG)` as a 64-bit integer
age	Time delta between report generation and system time in seconds
diag	A diagnostic information field used by QTI for rootkit detection debugging

The real-time rootkit detection is not run by default since the checks it performs on the Linux kernel can delay the creation of a hardware token too much for some use cases. However, every time it is run, the generated report is cached and timestamped. Only the report from the latest run is ever saved. Whenever a token is created, the latest cached report is always included, which does not impact the time it takes to create a token. Note that it is possible for rootkit detection to not have been run at all, in which case no rootkit report will be included in the token.

On the SDM845 chipsets, the caller can pass the `HAVENT_OPT_RTIC_LONG` flag to `initialize()` when creating a token. This will cause rootkit detection to run during the `initialize()` call for the token being created. It will delay initialization up to 1 second, though usually much less. The report included with the token will be up to date, and will be saved for use in later tokens that do not include the `HAVENT_OPT_RTIC_LONG` flag.

On the SDM670 chipsets, as an alternative to passing the `HAVENT_OPT_RTIC_LONG` flag, the ISV can call `rootkit_report_gen()` some time before calling the token creation APIs to create a relatively fresh report. This allows token creation to be both fast and have an up-to-date rootkit report. It will take up to 1 second to run `rootkit_report_gen()`, but it can be called at a time that will not require the end user to wait for its completion.

Also, on the SDM670 chipsets, a rootkit check is always performed when the device boots, thus there will always be a rootkit report in every hardware token. Since devices are not rebooted frequently, using the `HAVENT_OPT_RTIC_LONG` or `rootkit_report_gen()` is recommended for up-to-date reports.

7.2 Trusted location

The token can include location information that is obtained directly from the location engine on the chip, and is never exposed to Android. This location information cannot be manipulated even if the device is rooted, and so it is trusted.

Trusted location is not supported for Android app developers on the SDM660 and SDM630 chipsets, but it is supported for TA developers. On the SDM845 and later chipsets, it is supported for both Android and TA developers.

Location privacy for trusted location is handled in the same manner as it is for normal Android location APIs. Location information is only included in a token if the `HAVENT_OPT_LOCATION` is set. If set, then the app must have the `ACCESS_FINE_LOCATION` Android permission, or else token creation will fail with `ERROR_PRIVILEGE_ERR`. The app creating tokens should obtain permission from the user through standard Android APIs, which will prompt the user for location permission.

Trusted Location is based on the Global Navigation Satellite System (GNSS) engine, not Wi- Fi or cellular triangulation. The token creation process does not wait for a GNSS position fix, and instead includes cached location information from the GPS engine. It is therefore possible that the location is unknown or is very old. Each value in the location information has a Boolean that indicates whether the data is valid or not. For example, if the location has not been fetched, then the latitude and longitude validity data items will be false. The location information in the token is also timestamped to indicate its age.

A small daemon wakes up every thirty minutes, and attempts to get a fix for 20 seconds. This means that there is likely to always be some trusted location information, although it may be old. Any use of GNSS by the end user will also generate a fix that is cached.

To fetch an up to date location, the standard Android location API should be used. The following example shows how to do this in an Activity class:

```
LocationManager manager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);

manager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
locationListener);
```

The first parameter of the above example is the type of location provider (GPS in this case). The second parameter, minimum time interval between notifications, and the third parameter, minimum distance between notifications, control the frequency at which the listener (provided as the fourth parameter) receives updates. Setting both parameters to zero requests location notifications as frequently as possible.

Although the Android API requires a listener, the caller does not need to handle this location information. The location will be cached internally in the GPS engine and picked up during token creation. Keep in mind that it generally takes tens of seconds, sometimes even minutes to get a fix.

When creating tokens from the C API in a TA, no permission, Android or otherwise, is required. However, it is the caller’s responsibility to obtain the user’s permission. If this is not done, privacy regulations and the Android CDD will be violated. Generally, TAs have an associated Android app. That Android app is the logical place to handle this permission request.

8 Hardware token API error codes

This section describes different error codes and the reason for their occurrence. In the Java interface, these error codes show up as text in the error string. In the C interface, they are integers returned by the various functions, defined in `IHavenTokenApp.h`. Note that in the C API, the error code names have "IHavenTokenApp_" prefixed to them (e.g. `IHavenTokenApp_ERROR_TOO_MUCH_APPDATA` instead of `ERROR_TOO_MUCH_APPDATA`)

Table 8-1 Token creation errors

Error	Comment
ERROR_SECURITY_LEVEL_NOT_SUPPORTED	This error relates to the `nSecurityLevel` parameter in the C API. It means that the context in which data is being added is not at the right security level. Data can only be added at a security level that is less secure than the current level. For example, Rich OS security level data can be added from a TA, but TEE user security level data cannot be added from an HLOS app.
ERROR_TOO_MUCH_APPDATA	Attempting to add too much data with the `AddData` calls. The token is limited to 64KB of user added data.
ERROR_NO_DATA	In the C API, this error may occur when calling `HavenT_GetBytes()` before `HaventT_GetSize()`
ERROR_OFFSET_TOO_LARGE	This error occurs in the C API when the `uOffset` parameter to `HavenT_GetBytes()` is too large. This should never happen with the Java API.
ERROR_OUT_OF_ORDER	This error occurs in the C API when there is an attempt to add data to a token that has already been generated. For example, calling `HavenT_AddBytes()` after calling `HavenT_GetSize()`or calling `addLong()` after calling `finish()` in the Java API.
ERROR_PRIVILEGE_ERR	This error occurs if the caller includes the `HAVENT_OPT_LOCATION` option, but has not been granted the Android `ACCESS_FINE_LOCATION` permission.
ERROR_NOT_ALLOWED	One of the options selected to generate the token is not allowed. Debugging with simulation data can help to resolve this error.

8.1 License certificate errors

These errors all relate to the license certificate passed in to the `startInstance()` method. The license certificate must have the following characteristics:

- Must be an X.509 Certificate in DER format for the SDM630 and SDM660 chipsets, or PEM or DER format for the SDM845 and later chipsets (See License Certificate Formats section)
- It must have been issued by QTI (see section on Obtaining a License Certificate from QTI)
- It must be valid for the current time or have no time expiration

Table 8-2 License certificate errors

Error	Comment
ERROR_INVALID_CERT	The certificate is syntactically invalid or corrupted. Ensure that an accepted certificate format is being used.
ERROR_CERT_FEATUREID	A QMS feature is not allowed to be used with the license. Some of the HAVENT_OPT_ flags require a special license, however these flags are generally not useful for developers. While it is not usually the case, a license certificate may allow a feature like secure camera and not hardware token, in which case this error would be returned.
ERROR_CERT_EXPIRED_OR_NOTYETVALID	The license is expired or not yet valid.
ERROR_CERT_OEM	The license is restricted to a specific device manufacturer that does not match the manufacturer of the device being used.
ERROR_CERT_HWVERSION	The license is restricted to a specific chip that does not match the chip of the device being used.
ERROR_CERT_ISVCERTHASH	The license is bound to a specific Android app signing key and certificate, which is sent to QTI when obtaining the certificate. This error occurs when the calling app is signed by a key that does not correspond to the certificate.
ERROR_CERT_PKHASH	The license certificate is not valid for the device PKHash. The PKHash identifies an OEM, so this error is equivalent to ERROR_CERT_OEM.
ERROR_CERT_NOT_TRUSTED	The license certificate was not issued by QTI.
ERROR_CERT_GENERAL_ERR	Either an internal error with the license certificate processing, or a corrupt or syntactically invalid certificate.
ERROR_LICENSE_TOO_BIG	The license certificate is above the allowed size. QTI will never issue licenses that are too large, so this means that the license file is corrupt or the incorrect file.

8.2 Signing key related errors

These errors relate to the selection of the attestation key.

Table 8-3 Signing key related errors

Error	Comment
ERROR_SIGNING_KEY_UNAVAIL	Issue with the `keySelect` argument to `startInstance()`. The key being requested is not supported on this device. See the Starting Token Creation section.
ERROR_DEVICE_NOT_SECURE	The attestation key selected cannot be used unless secure boot is enabled for the device. It is the OEM that enables secure boot.

8.3 System errors

System errors usually indicate something internally wrong with the QMS hardware token implementation. Usually there is little that can be done by the caller of the APIs.

Table 8-4 System errors

Error	Comment
ERROR_NOMEM	Internally out of memory. In some cases, reducing the amount of added data may help.
ERROR_DATA_MARSHAL	Error in data transfer to the TEE. The data formatted by the QMS SDK libraries is not as expected by the TEE.
ERROR_HASH_GENERATION	Internal implementation error invoking a hash function.
ERROR_RPMB_ERR	Error relating to the RPMB. See section below on RPMB.
ERROR_CBOR_ENCODE_ERR	Error creating the CBOR encoding. A potential cause is adding malformed raw CBOR data.
ERROR_SIGNING_ERR	Error during computation of the signature.
ERROR_ENCRYPTION	Internal error while performing CMS encryption.
ERROR_GROUPKEY_ERR	Error while using group key.
TRANSPORT_ERROR_xxx	These errors relate to transfer of data into the TEE from the HLOS.
ERROR_DATA_TOO_BIG	Internal buffer size error.

9 Simulated data, development and test

9.1 Simulated data

When the `HAVENT_OPT_SIMDATA` option is provided, the token will include fixed hard-coded data that simulates real data. The simulated data is the same across all devices. For example, the device ID will always be `0x00000200` no matter what chip is in use, and the location will always be in the Laurentian Abyss in the middle of the Atlantic Ocean, regardless of the actual location of the device.

Although the data is simulated when this option is selected, all other token features work in the same manner that they would with real data. The `HAVENT_OPT_LOCATION` option will still control whether location information is returned or not, the caller can still add user data, and the token will still be signed and encrypted as usual.

This mode is useful for implementing and automating system end-to-end test cases. Since the data values will never change, the developer can know exactly what to expect after the data has been decrypted, verified, translated, and otherwise processed.

9.2 Demo signing key

In addition to simulated data, the demo signing key, selected with the `HAVENT_KEYTYPE_DEMO` key type, is very useful for testing since it is available on every device and is always the same.

Note, however, that despite the simulated data and demo signing key remaining constant, the token signature will change each time. This is because a random number is used in computing each ECDSA signature, which is typical of the ECDSA signature algorithm. The encrypted data of each token will also be different because a random AES session key is used every time.

10 Decrypting, verifying, and decoding a hardware token

To process a hardware token on the server, it must first be decrypted. Then the signature must be verified to ensure that the data in the token is valid, not modified, and not a forgery or a fake. After these two steps occur, the result is CBOR formatted data (both user-added and standard data items) from the device.

For most applications, it is helpful to convert the CBOR formatted data to JSON. Doing this is very straight-forward and can be done with standard tools.

10.1 Using the QMS token verification tool

The QMS token verification tool is a python script that verifies the signature

of a hardware token using an attestation certificate. The tool takes a CMS encrypted token or an unencrypted CBOR encoded token and verifies its signature using configured attestation certificates.

The tool is supplied in the QMS SDK and runs on Linux, Windows, and in Amazon Web Services. See the README.md file, also provided in the SDK, for the installation requirements and instructions.

An example command to decrypt, verify and decode a token is:

```
python qms_tvt.py -k <decrypt_file>
-j <json_output> -x <cert_out> <token_file>
```

`<token_file>` is the encrypted and signed token received from the device. This is a binary file. If it was base-64 encoded for transport, the base-64 encoding must be removed before providing it to the tool.

`<decrypt_file>` is the decryption key in PEM or DER format. This is the private key associated with the ISV’s license certificate. It will be needed for decrypting the token.

`<json_output>` is the file into which the data items from the token are written in JSON format.

`<cert_out>` is the file into which the attestation certificate will be written.

The subject of the attestation certificate will be written to stdout, relaying the type and strength of the attestation key used to sign the token.

10.1.1 Unsecured demo attestation signing keys

Some of the attestation keys used are demo / test keys that are not secured. The certificates associated with these keys have `UNSECURED - TEST ONLY` in the OU of the Subject DN of the certificate. Normally the tool will error out and return no data when a demo attestation key is used. To proceed with these attestation keys for demo and test work, then provide the `-u` option.

10.1.2 Output format

The example above shows how to receive the output as JSON formatted data. To work with CBOR formatted output instead of JSON, replace `-j <json_output>` with `-c <cbor_output>`.

10.1.3 Decrypting with another tool

Since the CMS encryption format is an IETF standard, decryption can be done by common tools like OpenSSL instead of the QMS token verification tool (see the Decrypting with OpenSSL section). This approach may be necessary if the ISV decryption key is not simply stored in a file, but in a cryptographic HW engine, for example. Decrypting the token produces CBOR formatted signed data. The token verification tool will still be necessary to verify the signature, and can be used to translate the data to JSON format.

The QMS token verification tool automatically detects whether the input token file is encrypted or not. If the input token is not encrypted, there is no need to provide the `-k` option or any other special option.

10.2 Decrypting with OpenSSL

Hardware tokens are encrypted using standard CMS format specified in RFC 5652 section 6. The data is encrypted with a randomly generated 256-bit AES key, in CBC mode, and padded to the AES block size as per RFC 5652. This encryption key is wrapped with the ISV's 2048-bit RSA public key using RSAES_OAEP encryption.

With OpenSSL, the command to decrypt a token is:

```
openssl cms -decrypt -in Token.der -inform der
   -inkey <isv_key_name>.pem
```

Some things to keep in mind:

- The following versions of OpenSSL work: 1.0.2g
- The following versions of OpenSSL do NOT work: 0.9.8zg, 1.0, 1.0.1
- Other versions may or may not work

In the command above `cms –decrypt` instructs OpenSSL to perform CMS format decryption.

`Token.der` is the input file name. It should be the exact binary bytes that were returned by the `finish()` command described in the Completing the Token section. The actual file name and extension are not important.

`<isv_key_name>.pem` is the name of the file with the ISV's private key, generated as described above in the section on Obtaining a License Certificate from QTI.

10.3 Working with the data items

After using the QMS token verification tool, the output will be CBOR and/or JSON formatted data. A separate document provides a detailed description of each data item and how it is useful in an authentication risk engine.

The data is grouped by security levels as follows:

10.3.1 Rich OS user

Data at this level originates in Android. This is the lowest security level, as the data can be corrupted if the HLOS is compromised or rooted.

10.3.2 TEE user

Data at this level originates in a TA or a subsystem on the device that is not affected when Android is rooted. A compromise in the system SW written by the device OEM can lead to the compromise of this data.

10.3.3 TEE kernel

Data at this level originates either in the TEE Kernel or in HW. It will require a compromise of the TEE kernel or a HW flaw for this data to be compromised. A compromise of the OEM’s SW is unlikely to lead to a compromise of this data. Some of the data items at this level are set by the OEM in fuses when they configure the device. The OEM cannot, however, alter these values in the token.

11 Miscellaneous topics

11.1 RPMB

Hardware token requires that the device supports the replay protected memory block (RPMB). RPMB is common in most flash memories, and it is used by QTEE to store data that should be protected from a roll-back attack (This is an attack where a rooted device saves a copy of QTEE data and restores it at a later time).

Occasionally, on brand new non-commercial development devices, such as the devices provided to QTI partners, RPMB may not be initialized. This issue should not be present on commercial handsets where OEMs should have set up RPMB correctly. Initialization is only needed once and does not need to be repeated when devices are re-flashed.

When RPMB is not initialized, an error, `ERROR_RPMB_ERR`, will be produced. As explained in the Hardware Token API Error Codes section , this may be returned as an int in the C API or may be in the string returned from the TokenException Java class. A line like the following may also be produced in the log file:

```
 921   921 E SmcInvoke_IMinkBinder Failed to invoke, err = 28
```

If this error is received, contact the device manufacturer for help initializing RPMB. If the OEM does not know how to do this, they can contact their customer engineering representative at QTI.

11.2 OEM ID

The OEM ID is a 32-bit integer that identifies the device manufacturer, or the OEM. These IDs are assigned by QTI to distinguish one OEM from another. QTI does not currently publish the list of OEM IDs, however individual OEMs can obtain their ID with support from QTI Customer Engineering.

The ID is blown into fuses on the Qualcomm® SnapdragonTM Mobile Processor SoC by the OEM. A lock bit is then blown so that the ID cannot be changed. The OEM ID is only readable by QTEE, meaning that it can be trusted even if the HLOS is rooted. During token creation, the OEM ID is used for license checks and is inserted into the hardware token by QTEE.

Sometimes OEMs do not set their ID in the device fuses. If that is the case the OEM ID will be 0, which is an invalid OEM ID. An OEM ID value of 1 is also invalid for commercial devices.

11.3 License certificate formats

The native binary format of an X.509 certificate is DER. A DER format certificate is a binary file that usually starts with the bytes 0x30 or 0x82. The file extension can be `.der`, `.crt`, or `.cer`.

There is also a text encoding of certificates called PEM. A PEM format certificate is a text file that usually starts with a separator like "`-----BEGIN CERTIFICATE----`". It may also have a textual description of the certificate or the license before the separator. The file extension is usually `.pem` or `.crt`.

The formats can be converted without any loss using the OpenSSL conversion commands below:

```
openssl x509 –inform der –in <isv_cert>.der \
-outform pem –out <isv_cert>.pem

openssl x509 –inform pem –in <isv_cert>.pem \
–outform der –out <isv_cert>.der
```

The license formats supported in QMS are as follows:

Table 11-1 Supported license format

Chipsets	Format	Comments
SDM835	PEM Only	Used only with QMS application security
SDM660 and SDM630	DER Only	
SDM845 and beyond	PEM or DER	Format is auto-detected. An intermediate certificate is also supported.

On the SDM845 chipsets and beyond, both DER and PEM formats are supported. The license processor examines the data to determine which format it is in, and so the file extension does not matter. There are no option flags to select the format of the certificate.

The SDM660 and SDM630 chipsets only support DER formatted license certificates, and the SDM835 chipsets only supports the PEM format for licensing the application security feature.

The license certificates for the SDM835, SDM630, and SDM660 chipsets are simple, single X.509 certificates, which can easily be converted from one format to another. For SDM845, the license may include multiple X.509 certificates, and should not be converted between formats. However, this should not be a problem as both formats are accepted on the SDM845 chipsets.

A Token data format

A.1 Basic Format

The token format varies depending on the chip. This should not concern the end user as the tools provided by QTI will hide most of the variations from one chip to the next. Regardless, the token is always formed in three nested layers.

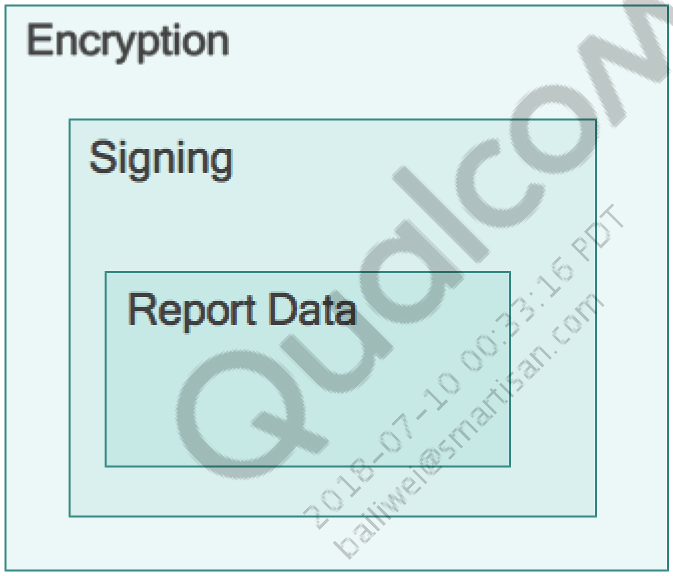

Figure A-1 Token data basic format

The inner layer is a report that is made up of a series of data items. These include things like the device ID, trusted location, and user added data. Hardware tokens generally segment the report data by security level, using a separate map for each security level.

The report is always in CBOR format. CBOR was selected as an upcoming IoT-oriented IETF standard for representing data in a very efficient and compact way. Due to its compactness, it can easily be included in many protocols with little impact. It is a binary format and not readable by humans, but it can easily be converted to and from JSON. CBOR, however, is more compact than JSON, and like JSON, supports array and map types.

The signing layer guarantees the authenticity of the origin of the data, and that the data was not modified in transit. There is one signature for all the data in the report.

The encryption layer is all that's visible for an encrypted binary token. It is composed of encryption type information, an encrypted session key, and the encrypted payload, all as binary formatted data. The encrypted payload contains the signed data.

A.2 Format on the SDM845 and later chipsets

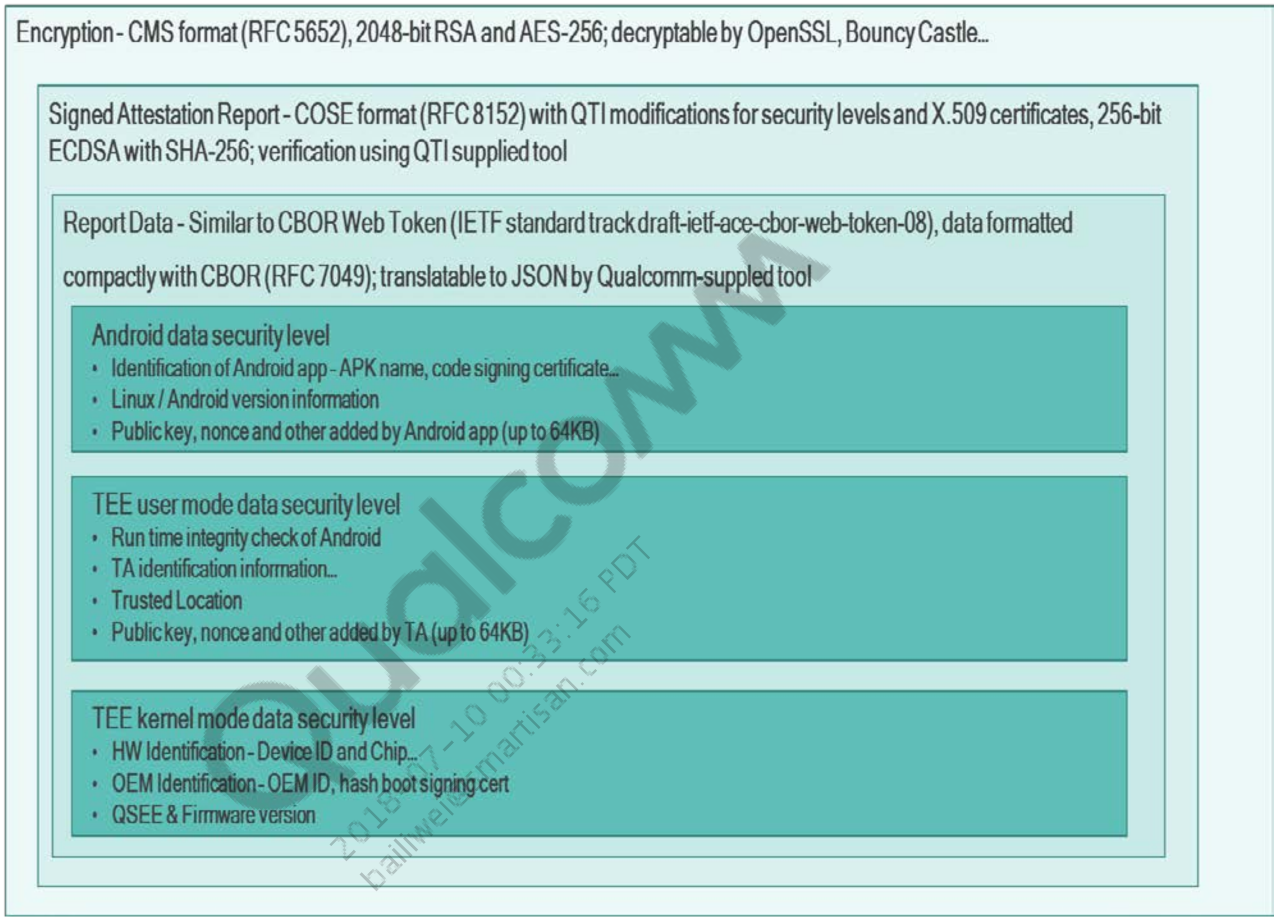
Encryption - CMS format (RFC 5652), 2048-bit RSA and AES-256; decryptable by OpenSSL, Bouncy Castle...
Signed Attestation Report - COSE format (RFC 8152) with QTI modifications for security levels and X.509 certificates, 256-bit ECDSA with SHA-256; verification using QTI supplied tool
Report Data - Similar to CBOR Web Token (IETF standard track draft-ietf-ace-cbor-web-token-08), data formatted compactly with CBOR (RFC 7049); translatable to JSON by Qualcomm-suppled tool
Android data security level
Identification of Android app - APK name, code signing certificate...
Linux / Android version information
Public key, nonce and other added by Android app (up to 64KB)
TEE user mode data security level
Run time integrity check of Android
TA identification information...
Trusted Location
Public key, nonce and other added by TA (up to 64KB)
TEE kernel mode data security level
HW Identification - Device ID and Chip...
OEM Identification - OEM ID, hash boot signing cert
QSEE & Firmware version

Figure A-2 Format on SDM845 chispets and later

The SDM845 chipsets and later use a modified COSE format for the signature. COSE is an up-and-coming IETF standard for signing data. It is a CBOR version of the JOSE signing format.

QTI modified the COSE format to handle multiple data security levels. COSE is defined to have only one payload, but the modification allows for multiple payloads (one for each security level).

Although COSE supports an encryption layer, the encryption format on the SDM845 chipsets is standard IETF CMS format using RSA public keys. The reason for this is that CMS supports RSA keys and X.509 for managing certificates, whereas COSE does not. There is also support for CMS in standard tools like OpenSSL. It is possible that a switch to COSE encryption will be made in the future. If that is done, the QTI tools will be updated, and the CMS format will be supported as an option, selectable by the caller.

A.3 Format on the SDM660 and SDM630 chipsets

The encryption format used on the SDM660 and SDM630 chipsets is the same as on the SDM845 chipsets.

The signature format used, however, is a QTI-proprietary format as COSE was not yet mature when these chips were released.

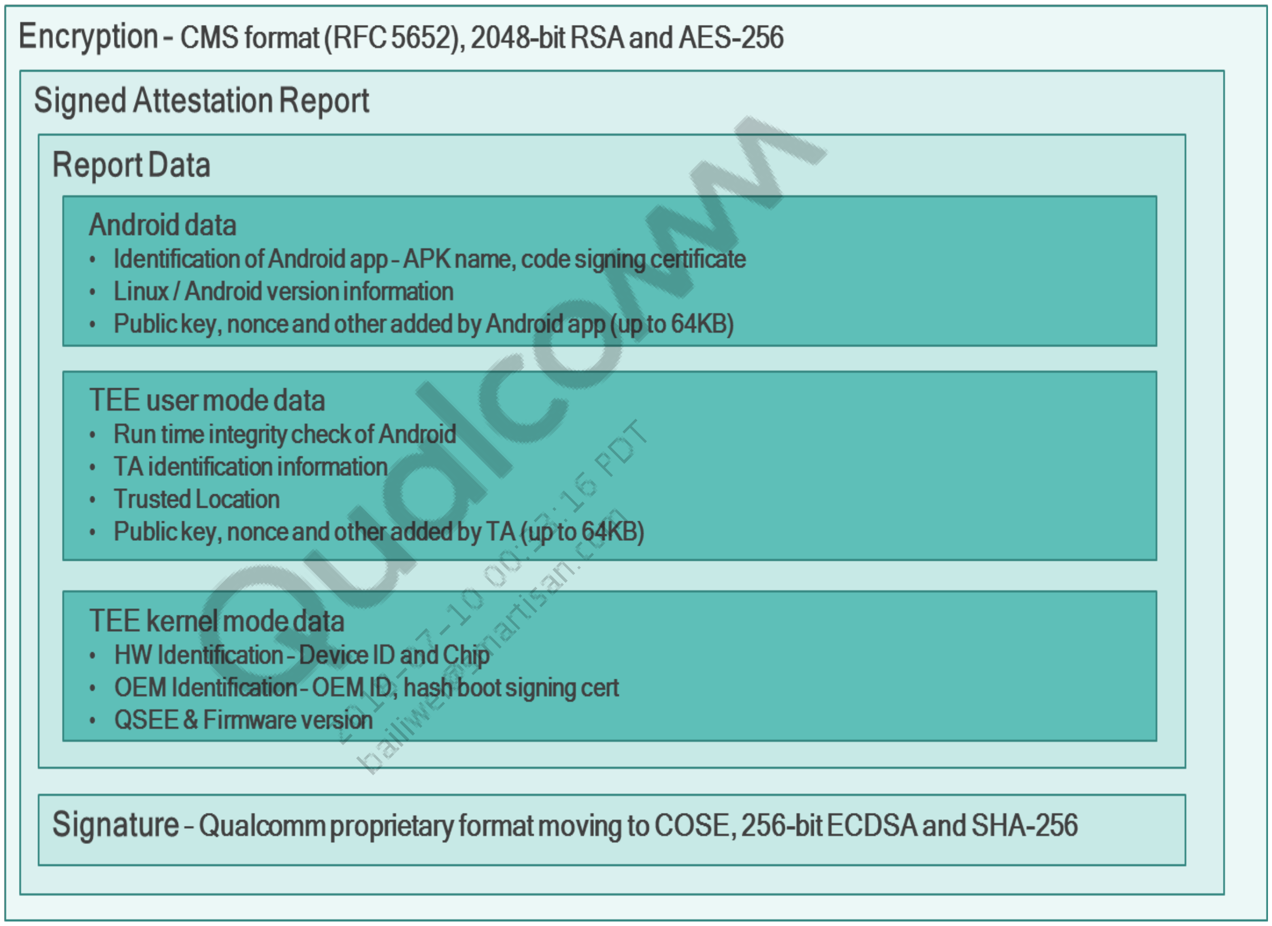
Encryption - CMS format (RFC 5652), 2048-bit RSA and AES-256
Signed Attestation Report
Report Data
Android data
• Identification of Android app - APK name, code signing certificate
• Linux / Android version information
• Public key, nonce and other added by Android app (up to 64KB)
TEE user mode data
• Run time integrity check of Android
• TA identification information
• Trusted Location
• Public key, nonce and other added by TA (up to 64KB)
TEE kernel mode data
• HW Identification - Device ID and Chip
• OEM Identification - OEM ID, hash boot signing cert
• QSEE & Firmware version
Signature - Qualcomm proprietary format moving to COSE, 256-bit ECDSA and SHA-256

Figure A-3 Format on the SDM660 and SDM630 chipsets

B References

B.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Haven Security Suite Token Overview	80-P8754-77
Standards	
Cryptographic Message Syntax (CMS)	RFC 5652 (September 2009)
Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile	RFC 5280 (May 2008)
Concise Binary Object Representation (CBOR)	RFC 7049 (October 2013)
CBOR Object Signing and Encryption (COSE)	RFC 8152 (July 2017)
Resources	
CBOR Website (Includes open source, tools, and more information)	http://cbor.io
FIDO Alliance Website (More information on the FIDO protocol)	http://fidoalliance.org

B.2 Acronyms and terms

Acronyms or terms	Definition
AES	Advanced Encryption Standard symmetric-key algorithm
AMSS	Advanced Mobile Subscriber Station
API	Application Programming Interface
APK	Android Package Kit
ASCII	American Standard Code for Information Interchange
CBC	Cipher Block Chaining
CBOR	Concise Binary Object Representation
CDD	Compatibility Definition Document
CMS	Cryptographic Message Syntax
COSE	CBOR Object Signing and Encryption
CSR	Certificate Signing Request
DER	Distinguished Encoding Rule
ECDSA	Elliptic-Curve Digital Signature Algorithm
FIDO	Fast Identity Online
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HLOS	High Level Operating System
HTTP	Hypertext Transfer Protocol
HW	Hardware

Acronyms or terms	Definition
IETF	Internet Engineering Task Force
ISV	Independent Software Vendor
JOSE	JavaScript Object Signing and Encryption
JSON	JavaScript Object Notation
MSM	Mobile Station Modem
OAuth	Open Authorization
OEM	Original Equipment Manufacturer
PEM	Privacy-Enhanced Electronic Mail
QTEE	Qualcomm Trusted Execution Environment (formerly known as Qualcomm secure execution environment (QSEE))
QTI	Qualcomm Technologies, Inc.
RFC	Request for Comments
RPMB	Replay Protection Memory Block
RSA	Rivest-Shamir-Adleman public-key cryptography
SDK	Software Development Kit
SDM	Snapdragon MSM
SHA	Secure Hash Algorithm
SW	Software
TA	Trusted Application
TEE	Trusted Execution Environment
TZ	TrustZone
UTF	Unicode Transformation Format