Qualcomm Technologies, Inc.

Qualcomm Linux Wireless Edge Services Guide

80-70018-11B AA

April 4, 2025

Contents

1 Qualcomm wireless edge services overview

Qualcomm® wireless edge services (WES) are a suite of trusted services embedded in hardware, designed to securely connect and manage devices. This guide details the Qualcomm WES software and its API references.

WES provides the following services:

- Feature licensing
- Device attestation
- Secure provisioning
- Trusted report

Note:

- See Hardware SoCs that are supported on Qualcomm Linux.
- For information about upgrading the feature packs, see Qualcomm Linux Security Guide $\rightarrow InstallorupgradeSoftSKUfeaturepacks$.

---

2 Feature licensing

Feature licensing enables one or more device features identified by a feature ID, which is typically aliased by a feature name. These device features are enabled by installing the corresponding feature licenses or certificates.

Feature licensing APIs support the following functions:

- Installing a feature license or certificate
- Verifying one or more feature IDs against the installed feature licenses or certificates
- Retrieving a list of all installed feature licenses or certificates
- Obtaining information about an installed feature license or certificate

A feature license or certificate can be restricted to one or more device attributes. For more information, see Feature license/certificate restrictions.

A license primarily contains a list of features to be enabled. Each feature is identified by a feature ID that's relevant to the technology it pertains to. A license can be further bound to one or more attributes of a device.

- OEM identifier (OEM ID) bound, platform license, or blanket licenses are valid on a device with matching OEM ID and public key (PK) hash values. For more information see:
  - OEM identifier
  - Public key hash
- Device ID-bound or per-device licenses are valid on the device with the matching device ID. Typically, a device ID-bound license is also OEM ID-bound. Each device has a unique device ID. For more information, see Feature licensing API reference.

2.1 Feature licensing models

If a specific feature requires a feature license or certificate, it adheres to one of the following feature licensing models. Qualcomm will communicate the licensing model used for that feature.

Feature licenses are used for:

- Evaluation feature license/certificate: Issued for evaluating features that typically carry a license expiry date and are used during the evaluation phase of the device.

- Production feature license/certificate: Issued to commercially enable a feature. These feature licenses/certificates apply to the production phase of the device.

OEM platform or blanket feature license/certificate

An OEM platform or blanket feature license/certificate enables one or more features on all devices with the same chip product and OEM.

Evaluation phase

In the evaluation phase for an OEM platform or blanket feature license/certificate, you must:

1. Blow the OEM ID fuse.

   For more information, see Blow OEM ID fuse.

2. Request an OEM evaluation feature license/certificate.

   For more information, see Request OEM evaluation feature license.

3. Provision feature license/certificate.

Production phase

In the production phase for an OEM platform or blanket feature license/certificate, you must:

1. Enable secure boot.

   For more information, see Enable secure boot .

2. Blow the OEM ID fuse.

   For more information, see Blow OEM ID fuse.

3. Request an OEM production feature license/certificate.

   For more information, see Request OEM production feature license.

4. Provisioning production license/certificate.

ISV platform or blanket feature license/certificate

The independent software vendor (ISV) platform or blanket feature license/certificate enables one or more features on all devices with the same chip product and for one or more application clients signed with unique signing keys.

Evaluation phase

In the evaluation phase for an ISV platform or blanket feature license/certificate, you must request an ISV evaluation feature license/certificate.

For more information, see Request ISV evaluation feature license.

Production phase

In the production phase for an ISV platform or blanket feature license/certificate, you must:

Request an ISV production feature license/certificate.

See Request ISV production feature license for details.

Provision feature license/certificate

To provision a feature license/certificate for an ISV platform or blanket feature license/certificate, you must:

- Package feature license/certificate in application.

  The feature license/certificate can be added as an asset in the application so that it can be installed after the application is installed and run.

- Install feature license/certificate.

  To install the license, call the `InstallLicense` Qualcomm WES API. See Qualcomm WES APIs for more information.

OEM per-device feature license/certificate using Qualcomm WES sync

A per-device feature license/certificate is bound to the unique identifier of the device. This feature licensing model provides a mechanism for Qualcomm software running on the OS to download automatically and install per-device feature licenses/certificates meant for that device.

This licensing model allows you to use the OS without the need to provide any software running on it.

Qualcomm reserves a few bits (typically 32-bits) for Qualcomm WES in the Qualcomm fuse-programmable read-only memory (QFPROM) region. You must define the values and associate each value to a feature or list of features. Contact Qualcomm and inform the association between value and feature list on a given chip product. Qualcomm WES Cloud issues licenses based on policy configured on the Qualcomm WES Cloud side and the value indicated in the license request.

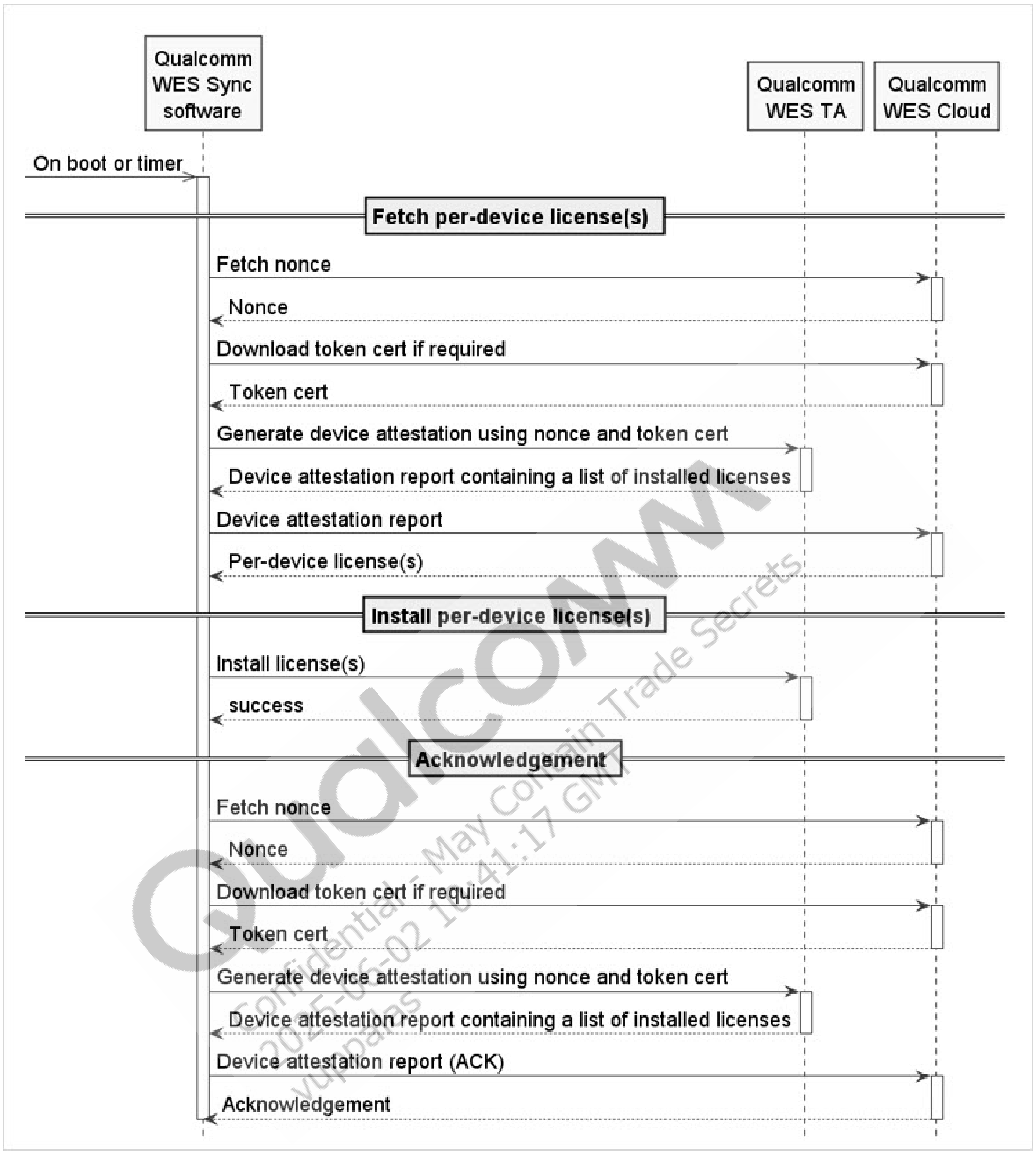

Qualcomm WES Sync software
Qualcomm WES TA
Qualcomm WES Cloud
On boot or timer
Fetch per-device license(s)
Fetch nonce
Nonce
Download token cert if required
Token cert
Generate device attestation using nonce and token cert
Device attestation report containing a list of installed licenses
Device attestation report
Per-device license(s)
Install per-device license(s)
Install license(s)
success
Acknowledgement
Fetch nonce
Nonce
Download token cert if required
Token cert
Generate device attestation using nonce and token cert
Device attestation report containing a list of installed licenses
Device attestation report (ACK)
Acknowledgement

Figure : Qualcomm WES sync process

For this feature license/certificate that uses the Qualcomm WES sync, do the following:

1. Enable secure boot.

   See Enable secure boot for details.

2. Get assistance from Qualcomm to set up a Qualcomm WES cloud account that delivers a per-device license and communicate the fuse value along with an associated feature list.

3. Configure the Qualcomm delivered sync software running on the OS. Specifically, enable sync by setting `initial_sync=1` in `qwesd.rc`.

See Configure feature license/certificate sync for details.

4. Configure fuse bits in QFPROM as per feature requirement.

   See Blow feature configuration QFPROM fuses for details.

Per-device feature license/certificate using customer application and cloud

A per-device feature license/certificate is bound to the unique identifier of the device. This licensing model requires you to have your own cloud and software running on the device to communicate with the Qualcomm WES cloud. Additionally, you must generate a device attestation report by invoking the Qualcomm WES device attestation APIs.

This licensing model applies to OEMs and ISVs.

See Device attestation API reference for details.

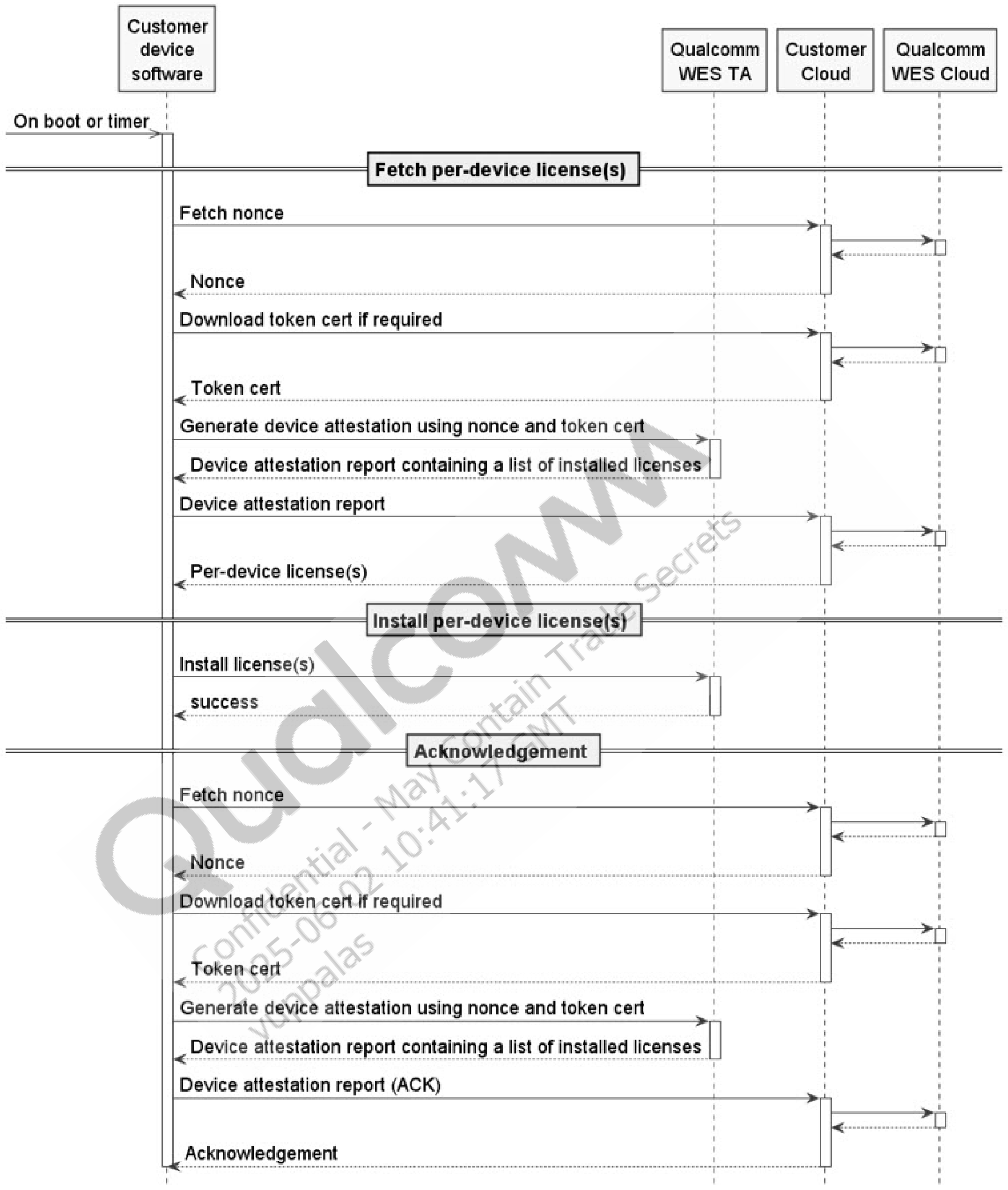

Customer device software
Qualcomm WES TA
Customer Cloud
Qualcomm WES Cloud
On boot or timer
Fetch per-device license(s)
Fetch nonce
Nonce
Download token cert if required
Token cert
Generate device attestation using nonce and token cert
Device attestation report containing a list of installed licenses
Device attestation report
Per-device license(s)
Install per-device license(s)
Install license(s)
success
Acknowledgement
Fetch nonce
Nonce
Download token cert if required
Token cert
Generate device attestation using nonce and token cert
Device attestation report containing a list of installed licenses
Device attestation report (ACK)
Acknowledgement

Figure : Customer feature license/certificate delivery process

For this feature license/certificate that uses the customer application and cloud, do the following:

1. Get assistance from Qualcomm to set up a Qualcomm WES cloud account that will deliver per-device licenses.

   Request one or more feature names to include in the per-device feature license/certificate at this point. See Feature licensing definitions Feature identifier..

2. Work with Qualcomm to acquire a feature license/certificate for calling into the device attestation APIs.

   This feature license/certificate isn't the same as the per-device feature license/certificate. The per-device feature license/certificate enables features in the software product (platform) while this feature license/certificate permits the device software to use the device attestation APIs.

   For the device attestation feature license/certificate, follow the procedure as described in ISV platform or blanket feature license/certificate.

3. Write software responsible for the following:

   1. Communicating with the customer's cloud. The customer cloud in turn communicates with the Qualcomm WES cloud.

   2. Invoke the Qualcomm WES device attestation APIs to generate the device attestation.

      See Device attestation API reference for more information.

4. Decide how to provision the software on your device:

   1. You may choose to write your software as privileged by making it a part of the software image.

   2. ISVs must deliver software through an application store or work with you to run your software.

2.2 Definitions

The following is a list of commonly used terms and their definitions.

Feature license/certificate restrictions

A feature license/certificate that enables a feature may be bound by one or more device attributes.

Each such bound attribute restricts the feature license/certificate to a subset of devices on which they're applicable. For example, if a feature license/certificate has an OEM ID restriction then it enables the feature on the devices with that OEM ID. The following license restrictions are supported.

Feature identifier

A feature identifier (feature ID) is a number that identifies a feature.

A feature license/certificate file can contain one or more feature IDs. Each feature ID enables some device feature. Each feature ID or a group of feature IDs (called feature bundle) is assigned a feature name that's more indicative of the feature to enable. Using a feature name also makes it easy to communicate with Qualcomm.

OEM identifier

An OEM identifier (OEM ID) is a unique number to identify an OEM.

The OEM ID is blown into fuses making feature licenses/certificates valid on devices with that OEM ID. The OEM identifier is a number agreed upon by the OEM and Qualcomm.

Expiration date

An expiration date counted as the number of milliseconds from the UNIX epoch (January 1, 1970 GMT) is the time for which the feature license/certificate is valid.

If the device's time is more than the feature licenses/certificates expiration date for that feature, then the feature stops working. Expiration is used with evaluation feature licenses/certificates that are meant to evaluate a feature.

Hardware version

The hardware version is a unique number associated with the processor's family.

All supported chipsets have the same hardware version. Feature licenses/certificates restricted by one or more hardware versions enable features on the corresponding processor families.

For example, use the processor's name QCS6490/QCS5430 for communication.

Public key hash

A public key hash (PK Hash) is the SHA-256 hash of the public key corresponding to the OEM's private key used to sign OS images. The public key hash is blown into fuses as part of secure-boot enablement.

Feature licenses/certificates with one or more public key hashes enables the features on devices with those hashes.

OEM product identifier

The OEM product identifier (product ID) is a number that identifies an OEM product or variant.

For example, an OEM can have variants of the same phone (all other device attributes are the same) for different markets. Feature licenses/certificates with an OEM product identifier will enable features on devices with that OEM product ID.

Extra data

An extra data is any sequence of bytes encoded in feature licenses/certificates.

It can contain context information about the feature for which the feature license/certificate is issued. This data can be a maximum of 512-bytes. Extra data never acts as a license restriction however a specific feature could choose to restrict valid feature licenses/certificates based on extra data.

2.3 Feature licensing processes

This section describes the different processes related with feature licensing.

Request OEM evaluation feature license

To obtain an OEM evaluation feature license/certificate, contact Qualcomm with the following information:

- OEM product identifier
- Public key hash - Indicate if the evaluation feature licenses/certificates are required without any public key hash.
- Expiration date
- Chipsets on which the features must be enabled (hardware version).
- Feature identifier - Name of features requested.
- Indicate if the evaluation feature licenses/certificates are required to work on devices without enabling secure boot.
- Optional:
  - OEM product identifier
  - Extra data

Request OEM production feature license

To obtain an OEM production feature license/certificate, contact Qualcomm with the following information:

- OEM identifier
- Public key hash
- Chipsets on which the features must be enabled (hardware version).
- Feature identifier – Name of features requested.
- Optional:
  - OEM product identifier
  - Extra data

Request ISV evaluation feature license

To obtain an ISV evaluation feature license/certificate, contact Qualcomm with the following information:

- OEM identifier
- Expiration date
- Chipsets on which the features must be enabled (hardware version).
- Feature identifier – Name of features requested.
- Indicate if the evaluation feature licenses/certificates are required to work on devices without enabling secure boot (development devices).
- Optional:
  - OEM product identifier
  - Extra data

Request ISV production feature license

To obtain an ISV production feature license/certificate, contact Qualcomm with the following information:

- OEM identifier
- Public key hash
- Chipsets on which the features must be enabled (hardware version).
- Feature identifier – Name of features requested.
- Optional:
  - OEM identifier
  - Extra data

Blow OEM ID fuse

You can blow the OEM ID fuse without enabling secure boot.

To blow the OEM ID, do the following:

1. Obtain the unique OEM ID from Qualcomm.

   Contact the Qualcomm customer engineering team if:

   - You don't have an OEM ID or
   - Don't know if your organization has been assigned the OEM ID In this example, `OEM_ID = 0x0001`:

   ```
   <meta>/common/sectoolsv2/ext/<platform>/sectools fuse-blower --
   security-profile <meta>/common/sectoolsv2/<chipset>_security_
   profile.xml --generate --sign --fuse-oem-hw-id=0x0001 --signing-
   mode TEST --outfile basic_sec.elf
   ```

2. To flash the signed images and `sec.elf` using, see the flash procedure Qualcomm Linux Build Guide → *Flashimages*.
3. To check if the `OEM_ID` is blown, read `HWIO_QFPROM_RAW_OEM_CONFIG_ROW1_LSB_IN` or `HWIO_QFPROM_CORR_OEM_CONFIG_ROW1_LSB_IN` fuse address from within the Qualcomm TEE environment.
4. To read QFPROM from your TrustZone application, use the `qsee_fuse_read()` APIs.

Enable secure boot

Enabling secure boot and blowing the `OEM_ID` eFuse on the device is mandatory for using production feature licenses/certificates.

Contact Qualcomm for instructions on enabling secure-boot.

Each OEM production feature license/certificate is restricted to a unique OEM identifier. The production feature license/certificate allows the feature to operate on devices for which these eFuses have been blown.

To enable secure boot, do the following:

1. Send `sec.elf` to Qualcomm for verification after the `sec.dat` (fuse configuration used to blow fuses) file is generated.
2. Use `sec.elf` that's verified by Qualcomm.
3. After verification, continue the remaining steps as mentioned in Qualcomm Linux Security Guide → *Enablesecureboot*.

Note: If the software image on a secure boot-enabled device is updated, restart the device only after all the signed images have been fastboot flashed to the device. Otherwise, the device won't be able to boot up subsequently because the secure chain verification will fail.

Configure feature license/certificate sync

This process is a part of the feature licensing model as described in OEM per-device feature license/certificate using Qualcomm WES sync.

Feature license/certificate sync is configured through the `qwesd.rc` file. The syntax of configuration parameters is similar to a `.ini` file, where each line contains a key and value, separated by an equal sign.

The file is on the device at `/etc/`.

The following table lists the configuration parameters that are supported. The default value column provides the value of a parameter if `qwesd.rc` is absent or the parameter isn't specified.

Key	Description	Default value
`initial_sync`	To trigger sync on boot up. After you push the file to */etc* and reboot the device, it starts to sync the license from the cloud. Only if it is not done already.	0

Key	Description	Default value
`log_level`	To enable logs until `log_level` level. Reference levels are based on the Linux ranges (from 0 to 7).	0
`disable_net_up_down`	This key depends on the Wi-Fi to download the license and does not try to bring up the network but expects the network to be up by the time `qwesd` tries to sync the license.	0
`Server`	This parameter is the URL prefix for the Qualcomm WES server, including protocol, host name, and partial path.	https://api.qwes.qualcomm.com/api/d2c/1.0

Sample `qwesd.rc` content:

```
initial_sync=1
server=https://api.qwes-tst.qualcomm.com/api/d2c
```

Blow feature configuration QFPROM fuses

This process is a part of the feature licensing model as described in OEM per-device feature license/certificate using Qualcomm WES sync.

Few QFPROM fuses (usually 32) from the OEM configuration region are reserved for feature configuration. Each QFPROM fuse corresponds to 1 bit. The QFPROM bits are combined to form a feature bitmap value, which is then used by the Qualcomm WES cloud to generate a per-device feature license/certificate with the right feature set.

Note: The feature configuration bits may not be contiguous. They can be spread across different rows in the QFPROM region. The address for these bits varies based on the chip product.

See Secure Boot Enablement User Guide for blowing QFPROM fuses.

Table : Feature configuration QFPROM bit address for supported chip products

Region	Address	Bit range (inclusive)
`QFPROM_RAW_OEM_CONFIG_ROW0_MSB`	0x7801C4	Bit 16 – Bit 31
`QFPROM_RAW_OEM_CONFIG_ROW2_MSB`	0x7801D4	Bit 16 – Bit 31

In this case, 32-bits are spread across two QFPROM rows, `OEM_CONFIG_ROW0` and `OEM_CONFIG_ROW2`.

For example, In QCM6490 the bitmap value of 1 (Bit 16 at address 0x7801C4 blown) is mapped to a higher SKU feature pack. Hence, blowing bit-16 at the address indicated allows the device to get a feature license/certificate for a higher SKU feature pack. Contact Qualcomm team for more information about the higher SKU feature pack.

2.4 Feature licensing API reference

See Feature licensing.

IPFM

Interface to the platform feature manager (IPFM) provides access to a managed collection of feature licenses.

PFM also refers to an ASCII-encoded license format based on PEM. A complete description of feature licenses is beyond the scope of this guide. From a high level, a feature license is an X.509 certificate with extensions that define what features are enabled and under what conditions. The extensions provide a blacklist. If a value specified by any extension doesn't match either the query or the state of the device, the license validation fails. The validation will indicate an error code where the first extension mismatch was detected. Technically, a feature ID or FID is an integer that identifies a feature. However, the terms feature, feature ID, and FID are used interchangeably in the rest of this guide.

CBOR

Many of the IPFM methods use CBOR to encode inputs or outputs.

A full description of CBOR is beyond the scope of this guide. One reason CBOR is used in the API is to allow responses to include additional data in the future without requiring changes to client code or the `.idl`. Therefore, the client code shouldn't depend on the exact order of items in a CBOR map in a response, or on the exact number of items in a CBOR structure. Clients should skip unexpected items in arrays and maps in responses, and should only fail if any expected fields aren't present.

CBOR inputs and outputs are described using both CDDL and a CBOR diagnostic format, which is a JSON-like text version of the CBOR data. Square braces indicate a CBOR array, which may be heterogeneous. Curly braces indicate a CBOR map. CBOR maps used by IPFM always use text strings as map key values.

Distinguished names

The primary purpose of an X.509 certificate is to assert that a public key is owned by a subject.

The issuer and subject are identified by distinguished names. A distinguished name (DN) is a series of well-known attribute types (keys) and values. Most key and value pairs are optional. Several IPFM methods return the issuer and subject DNs in a CBOR map. The values in the CBOR map have been converted from the raw DER encoding into a CBOR map. Both the keys and values in the map are TSTR values.

The following key values are supported at a minimum:

- O - Organization
- OU - Organizational Unit
- CN - Common name

Common validation error codes

A number of IPFM methods return errors that indicate which restriction failed to match.

These errors are a general problem with the X.509 certificate structure. These error codes include `_CERT` in their names and are listed as follows:

- Restriction mismatch errors:
  - `ERROR_CERT_PKHASH`
  - `ERROR_CERT_FEATUREID`
  - `ERROR_CERT_EXPIRED`
  - `ERROR_CERT_OEM`
  - `ERROR_CERT_HWVERSION`
  - `ERROR_CERT_LICENSEE_HASH`
  - `ERROR_CERT_DEVICEID`
  - `ERROR_CERT_NOTYETVALID`
  - `ERROR_CERT_PRODUCTID`
- Errors for corrupted or incorrectly generated licenses:
  - `ERROR_INVALID_CERT`
  - `ERROR_CERT_NOT_TRUSTED`
  - `ERROR_CERT_GENERAL_ERR`
  - `ERROR_CERT_LEAF_IS_CA`

- Currently unused errors:
  - `ERROR_CERT_FIELD_TOO_BIG`
  - `ERROR_CERT_RESERVE2`

Other common error codes

The following list describes the common error codes that each method can return:

- `Object_ERROR_SIZE_OUT`: Many IPFM methods return variable-length data in CBOR format. If the output buffer provided by the client is not large enough to hold the data, `Object_ERROR_SIZE_OUT` is returned. Typical client code is written for a specific feature, with well-known limits on the license extensions that will be used for their licenses, especially ExtraData.
- `Object_ERROR`: This error code can be returned from most methods. This indicates an internal error of some kind and may indicate a system misconfiguration. For example, it can be returned if PFM cannot find the persistent storage for licenses. The circumstances under which the error occurred and the `qsee_log` must be reported to the maintainers of PFM.
- Other error codes: While every effort is made to list the errors that each method can return, the list shouldn't be considered exhaustive. A method may return error codes in addition to the one listed, either now or in the future.

Methods

This section lists the different methods for feature licensing API reference.

CheckFeatureIds

This method checks for licensing of multiple features.

```
method CheckFeatureIds(
    in buffer RequestCBOR,
    out buffer ResponseCBOR)
```

Given a CBOR list of feature IDs/licensee hash pairs, the method returns a CBOR list of structures of license information. For each entry in the request list, there will be a corresponding entry at the same position in the response list.

CDDL for RequestCBOR

```
RequestCBOR = {
    blobVersion : uint32,
    FeatureIDs  : [
        *[uint32, BSTR]
    ]
}
```

RequestCBOR example

```
RequestCBOR = {
    “blobVersion” : 1,
    “FeatureIDs” : [
        [1, 12fc331f2c],
        [2, cadfae1900],
        [3, 1502005678]
    ]
}
```

ResponseCBOR format for version 1

```
ResponseCBOR = [
    *[uint32, BSTR, BSTR, uint32, BSTR, uint64_t]
]
ResponseCBOR = [
    *[FeatureID, LicenseeHash, SerialNumber, ResponseCode, ExtraData,
Flag]
]
```

CBOR value descriptions

- `FeatureID`: Feature ID from the request
- `LicenseeHash`: Licensee hash from the request
- `SerialNumber`: The serial number of a license enabling the feature for the licensee. Zero-length BSTR, if no matching license was found.
- `ResponseCode`: License restrictions. Some combination of `LICENSE_HAS_XXX` values. Zero, if no matching license was found.
- `ExtraData`: Version 1 only. Extra data extension from the license. Zero-length BSTR, if no matching license was found.
- `Flag`: Version 1 only. `OPT_NO_TIME_CHECK`, if license validation ignored expiration. 0 otherwise.

ResponseCBOR example for version 1

```
ResponseCBOR = [
    [800, “”, 22df4e681bc159922749d591219bc3ec, 32, “”, 0],
    [900, “”, “”, 0, “”, 0]
]
```

Note: The `ResponseCBOR` format for version 0 is the same, except that `ExtraData` and `Flag` aren't present.

Parameters

in	RequestCBOR	CBOR formatted list of FeatureIDs and `LicenseeHashes`
out	–	Gives out all the serial numbers in CBOR format and the error codes corresponding to the feature ID.

Returns

- `Object_OK` on success
- `Object_ERROR_SIZE_OUT`
- `ERROR_CBOR_ENCODE_ERR`
- `ERROR_CBOR_DECODE_ERR`
- `ERROR_CBOR_DECODE_DATATYPE_ERR`

CheckInstalledLicense

```
method CheckInstalledLicense(
    in uint32 FeatureID,
    in buffer LicenseeHash,
    out buffer LicenseSerialNumber)
```

Find a license that currently supports the given feature ID for the given licensee. If one or more managed licenses exist for the given licensee and feature ID, but validation of the licenses fails, `ERROR_FILE_NOT_FOUND` is returned and the reasons for validation failure is logged.

Parameters

in	FeatureID	`Feature ID` to check.
in	LicenseeHash	ISV hash of the client requesting the feature. This is only needed when the feature is in a license with a licensee hash extension.
out	LicenseSerialNumber	Serial number of the license on success.

Returns

- `Object_OK` on success
- `Object_ERROR_SIZE_OUT`

- `ERROR_FILE_NOT_FOUND`

CheckLicenseBuffer

```
method CheckLicenseBuffer(
    in buffer PFMLicense,
    in uint32 FeatureID,
    in buffer LicenseeHash,
    out buffer LicenseBufferCBOR)
```

Check whether the given license provides the given feature to the given licensee.

CDDL format for LicenseInfoCBOR

```
LicenseInfoCBOR = {
    “SerialNumber” : BSTR
    “Issuer” : CBOR Map
    “SubjectKeyID” : BSTR
    “SubjectDN” : CBOR Map
    “Expiration” : BSTR
    “LicenseeHashes” : BSTR
    “FIDs” : [*uint32]
    “FIDRanges” : [*[uint32, uint32]]
    “LicenseRestrictions” : uint64
    “ExtraData” : BSTR
    “Flag” : uint64_t
}
```

LicenseInfoCBOR example

```
LicenseInfoCBOR = {
    “SerialNumber” : 22df4e681bc159922749d591219bc3ec
    “Issuer” : { “OU”: “Issuer Org Unit”, “O” : “Issuer Organization”
}
    “SubjectKeyID” : 878e26b6d2b292e10ecd0cc085454118682c734f
    “SubjectDN” : { “OU”: “Subject Org Unit”, “CN”: “Subject Common
Name” }
    “Expiration” : 010000001234567833445566
    “LicenseeHashes” : “”
    “FIDs” : [1,2,3]
    “FIDRanges” : [[5,8], [10,15]]
    “LicenseRestrictions” : 1
    “Flag” : 8
}
```

CBOR value descriptions

- `SerialNumber`: Must match `LicenseSerialNumber`.
- `Issuer`: Distinguished name of the issuer. Refer to the description of distinguished names listed.
- `SubjectKeyID`: Subject-specific public key identifier. Normally 20 bytes.
- `SubjectDN`: Distinguished name of the subject. Refer to the description of distinguished names listed.
- `Expiration`: Valid time range of the license. 12 bytes for time-bound licenses.
- `LicenseeHashes`: Accepted client ISV hashes for this license. 32-bit count followed by concatenated hashes.
- `FIDs`: FIDs provided by the license.
- `FIDRanges`: FID ranges provided by the license.
- `LicenseRestrictions`: Bitmask of license restrictions. Some combination of `LICENSE_HAS_XXX` values.
- `ExtraData`: Extra data extension from the license, or zero-length BSTR if the license doesn't have the extension.
- `Flag`: `OPT_NO_TIME_CHECK` if license validation ignored expiration. 0 otherwise.

Parameters

in	PFMLicense	Feature license in PFM or DER format.
in	FeatureID	The feature ID to check.
in	LicenseeHash	ISV hash of the client requesting the feature. This is only needed when the license has a licensee hash extension.
out	LicenseBufferCBOR	Information about the license as CBOR bytes.

Returns

- `Object_OK` on success
- `Object_ERROR_SIZE_OUT`

Common validation error codes

`ERROR_LICENSE_TOO_BIG`

GetAllInstalledFeatureIDs

```
method GetAllInstalledFeatureIDs(out buffer FeatureIDsCBOR)
```

Get a list of installed feature IDs.

This API examines and verifies every managed license, and returns an aggregate list of currently supported feature IDs.

CDDL for FeatureIDsCBOR

```
FeatureIDsCBOR = {
    FIDs: [*uint32],
    FIDRanges: [*[uint32, uint32]]
}
FeatureIDsCBOR Example
FeatureIDsCBOR = {
    "FIDs" : [1,2,3],
    "FIDRanges" : [[10,20],[30,40]]
}
```

CBOR value descriptions

- `FIDs`: `FIDs` provided by currently valid licenses
- `FIDRanges`: `FID` ranges provided by currently valid licenses

Parameters

out	FeatureIDsCBOR	CBOR formatted data out for `FIDs` and `FIDRanges`.

Returns

- `Object_OK` on success
- `Object_ERROR_SIZE_OUT`

GetAllInstalledSerialNumbers

```
method GetAllInstalledSerialNumbers(out buffer SerialNumbersCBOR)
```

Get the serial numbers for all installed licenses.

This method iterates over all the licenses in the license store and gives the caller a list of serial numbers, one for each license. The client can use this information to report the list of licenses present to get the details of the licenses, and to know which licenses to remove.

CDDL for SerialNumbersCBOR

```
SerialNumbersCBOR = [
    *BSTR
]
SerialNumbersCBOR Example
SerialNumbersCBOR = [
    1234561,
    3af4e0d8,
    567456
]
```

Parameters

out	SerialNumbersCBOR	List of installed serial numbers.

Returns

`Object_OK` on success else error out.

CheckFIDAndGetAllSerialNums

```
method CheckFIDAndGetAllSerialNums(
    in buffer RequestCBOR,
    out buffer ResponseCBOR)
```

Get all license information for a list of features.

For reach FID/licensee pair in the request, the response contains an entry that reports information for all of the licenses that provide the given feature for the given licensee.

CDDL for RequestCBOR

```
RequestCBOR = {
    blobVersion : uint32,
    FeatureIDs  : [
        *[uint32, BSTR]
        ]
}
RequestCBOR Example
RequestCBOR = {
    "blobVersion" : 1,
    "FeatureIDs" : [
        [1,12fc331f2c],
        [2,cadfae1900],
        [3,1502005678]
    ]
}
```

CDDL for ResponseCBOR for version 1

```
ResponseCBOR = [
    *{
        "FID": uint32
        "LicenseeHash": BSTR
        "Result": [
          * {
                "SerialNum": BSTR
                "LicenseRestrictions": uint64
                "ExtraData": BSTR
            }
        ]
        "Response": uint32
        "Flag": uint64
    }
]
ResponseCBOR = [
    {
        "FID" : 100
        "LicenseeHash" : ""
        "Result" : [
            {
                "SerialNum" : 23456,
                "LicenseRestrictions" : 1F,
                "ExtraData" : ""
            },
            {
                "SerialNum" : 234534,
                "LicenseRestrictions" : 0C,
                "ExtraData" : "Hello"
            }
        ]
        "Response" : 0
        "Flag" : 8
    },

    {
        "FID" : 200
        "LicenseeHash" : 234568
        "Result" : [
            {
                "SerialNum" : 23454,
                "LicenseRestrictions" : 03,
                "ExtraData" : ""
            },
            {
                "SerialNum" : 234123
                "LicenseRestrictions" : 02,
                "ExtraData" : ""
            }
        ]
        "Response" : 0
        "Flag" : 8
    }
]
```

Note: Version 0 is the same, except ExtraData and flag aren't present.

CBOR value descriptions

- `FID`: Feature ID from the request
- `LicenseeHash`: Licensee hash from the request
- `Result`: CBOR array of license information, one entry per matching license
- `SerialNum`: Serial number of a license enabling the feature for the licensee
- `LicenseRestrictions`: Bitmask of license restrictions. Some combination of `LICENSE_HAS_XXX` values
- `Response`: 0 for allowed, 1 for not allowed, 3 for processing error. Normally, this value is 0 if any licenses are listed in the results, and 1 if the result is empty. If a failure occurs while processing a potentially matching license, the response will be 3.
- `ExtraData`: Version 1 only. Extra data extension from the license, which may be empty.
- `Flag`: `OPT_NO_TIME_CHECK` if license validation ignored expiration. 0 otherwise.

Parameters

`in`	`RequestCBOR`	CBOR list of FeatureID and LicenseeHash pairs.
`out`	`ResponseCBOR`	CBOR structure as described.

Returns

`Object_OK` on success

GetInstalledLicenseInfo

```
method GetInstalledLicenseInfo(
    in buffer LicenseSerialNumber,
    out buffer LicenseInfoCBOR)
```

Fetch license information for the given license serial number. The information is encoded in the following CBOR structure described.

CDDL format for LicenseInfoCBOR

```
LicenseInfoCBOR = {
    "FileName" : string
    "SerialNumber" : BSTR
    "Issuer" : CBOR Map
    "SubjectKeyID" : BSTR
    "SubjectDN" : CBOR Map
    "Expiration" : BSTR
    "LicenseeHashes" : BSTR
    "FIDs" : [*uint32]
    "LicenseRestrictions" : uint64
    "ExtraData" : BSTR
    "Flag" : uint64_t
}
```

LicenseInfoCBOR example

```
LicenseInfoCBOR = {
    "FileName" : ""
    "SerialNumber" : 22df4e681bc159922749d591219bc3ec
    "Issuer" : { "OU": "Issuer Org Unit", "O" : "Issuer Organization"
}
    "SubjectKeyID" : 878e26b6d2b292e10ecd0cc085454118682c734f
    "SubjectDN" : { "OU": "Subject Org Unit", "CN": "Subject Common
Name" }
    "Expiration" : 010000001234567833445566
    "LicenseeHashes" : ""
    "FIDs" : [1,2,3]
    "FIDRanges" : [[5,8], [10,15]]
    "LicenseRestrictions" : 1
    "ExtraData" : 81828384
    "Flag" : 8
}
```

CBOR value descriptions

- FileName: Empty string that's no longer used.
- SerialNumber: Must match the license serial number.
- Issuer: Distinguished name of the issuer. Refer to the description of distinguished names list.
- SubjectKeyID: Subject-specific public key identifier. Normally 20 bytes.
- SubjectDN: Distinguished name of the subject. Refer to the description of distinguished names listed.
- Expiration: Valid time range of the license. 12 bytes for time-bound licenses.
- LicenseeHashes: Accepted client ISV hashes for this license. 32-bit count followed by concatenated hashes.
- FIDs: FIDs provided by the license.
- FIDRanges: FID ranges provided by the license.
- LicenseRestrictions: Bitmask of license restrictions. Some combination of `LICENSE_HAS_XXX` values.
- ExtraData: Extra data extension from the license, or zero-length BSTR if the license doesn't have the extension.
- Flag: `OPT_NO_TIME_CHECK` if license validation ignored expiration. 0 otherwise.

Parameters

`in`	`LicenseSerialNumber`	Serial number to search for.
`out`	`LicenseInfoCBOR`	CBOR structure as described.

Returns

- `Object_OK` on success
- `Object_ERROR_SIZE_OUT`

Common validation error codes

- `ERROR_FILE_NOT_FOUND`
- `ERROR_PFMFILER_GETFILECONTENTS_FAILED`

GetLicenseCertPFM

```
method GetLicenseCertPFM(
    in buffer LicenseSerialNumber,
    out buffer PFMLicense)
```

Return an installed license in PFM format.

Parameters

in	LicenseSerialNumber	Serial number of the license.
out	PFMLicense	License information in PFM format.

Returns

- `Object_OK` on success
- `ERROR_FILE_NOT_FOUND`
- `ERROR_PFMFILER_GETFILECONTENTS_FAILED`

GetNextExpiration

```
method GetNextExpiration(out uint64 expiration)
```

Get the expiration time for the next expiring licenses.

Parameters

out	expiration	Epoch time when the next licenses will expire.

Returns

- `Object_OK` on success
- `ERROR_INVALID_CURRENT_TIME`

InstallLicense

```
Method InstallLicense   (
    in buffer PFMLicense,
    out buffer LicenseSerialNumber,
    out buffer FeatureIDsCBOR,
    out uint64 LicenseRestrictions)
```

Install a new license

Installation will fail if device-based license restrictions don’t match the device. Installation ignores licensee ID, feature ID, and expiration restrictions. Output buffers will only be set on success.

CDDL for FeatureIDsCBOR

```
FeatureIDsCBOR = {
    FIDs: [*uint32],
    FIDRanges: [*[uint32, uint32]]
}
FeatureIDsCBOR Format
FeatureIDsCBOR = {
“FIDs” : [1,2,3],
    “FIDRanges” : [[10,20],[30,40]]
}
```

CBOR value descriptions

- FIDs - FIDs provided by the license
- FIDRanges - FID ranges provided by the license

Parameters

in	PFMLicense	Feature license in PFM or DER format.
out	LicenseSerialNumber	Serial number of the license.
out	FeatureIDsCBOR	Features provided by the license.
out	LicenseRestrictions	Bitmask of license restrictions. Some combination of `LICENSE_HAS_XXX` values.

Returns

`Object_OK` on success

Common validation error codes

- `ERROR_CBOR_ENCODE_ERR`
- `ERROR_LICENSE_TOO_BIG`

- `ERROR_LICENSE_STORE_FULL`
- `ERROR_DUPLICATE`
- `Object_ERROR_SIZE_OUT`

RemoveLicense

```
method RemoveLicense(in buffer LicenseSerialNumber)
```

Remove an installed license.

Parameters

`in`	`LicenseSerialNumber`	Serial number of the license to remove.

Returns

- `Object_OK` on success
- `ERROR_FILE_NOT_FOUND`
- `ERROR_PFMFILER_GETFILECONTENTS_FAILED`

RemoveLicenseExpired

```
method RemoveLicenseExpired()
```

Remove expired licenses.

This API removes any installed licenses that have an expiration date before the current date. Licenses will only be removed if the trust level of the current date is MEDIUM.

Returns

- `Object_OK` on success
- `ERROR_INVALID_CURRENT_TIME`

GetLicenseCertPFM

```
method GetLicenseCertPFM(
    in buffer LicenseSerialNumber,
    out buffer PFMLicense);
```

Return an installed license in PFM format.

Parameters

in	LicenseSerialNumber	Serial number of the installed license.
out	PFMLicense	License bytes in PFM format.

Returns

- `Object_OK` on success.
- `ERROR_FILE_NOT_FOUND` if a license with the given serial number isn't found.
- `ERROR_NO_LICENSES` if the license store is empty.
- `ERROR_PFMFILER_GETFILECONTENTS_FAILED` if an internal error occurs when searching of installed licenses.

GetFeatureConfig

```
method GetFeatureConfig(out uint64 fuses)
```

Get the status of fuses used to request licenses. On success, the fuses are set to a bitmask indicating which sync-request fuses have been blown. The fuse bitmask is also included in the device attestation when the IAttestationBuilder object is created using the IPFM's GeneratePFMReport method.

The fuse bitmask is used by the Qualcomm WES cloud to issue a per-device license when combined with a platform license already present on the device.

Parameters

in	fuses	Bitmask that indicates the fuse status. Zero if no fuses blown.

Returns

- `Object_OK` on success.
- `ERROR_NOMEM` if an internal allocation fails.

Error codes

The following is the list of all error codes and the license restriction codes:

- `error ERROR_BLOB_DECAP_FAILED`

  Decryption of the encrypted request failed.

- `error ERROR_BLOB_ENCAP_FAILED`

  Generation of encrypted response failed.

- `error ERROR_CBOR_DECODE_DATATYPE_ERR`

  The data type is incorrect in the client CBOR input parameter.

- `error ERROR_CBOR_DECODE_ERR`

  Client CBOR input parameter is invalid.

- `error ERROR_CERT_DEVICEID`

  The device's ID doesn't match the license.

- `error ERROR_CERT_EXPIRED`

  The license certificate has expired.

- `error ERROR_CERT_FEATUREID`

  The requested feature ID doesn't match the license.

- `error ERROR_CERT_GENERAL_ERR`

  Error while verifying the cert chain.

- `error ERROR_CERT_HWVERSION`

  The device's chip ID (chip family) doesn't match the license.

- `error ERROR_CERT_LEAF_IS_CA`

  Leaf certificate in the license is marked as a CA.

- `error ERROR_CERT_LICENSEE_HASH`

  The requested licensee hash doesn't match the license.

- `error ERROR_CERT_NOT_TRUSTED`

  The license is not signed with a recognized root CA.

- `error ERROR_CERT_NOTYETVALID`

  The license certificate isn't yet valid.

- `error ERROR_CERT_OEM`

The device's OEM ID doesn't match the license.

- `error ERROR_CERT_PKHASH`

  The device's PKHash doesn't match the license.

- `error ERROR_CERT_PRODUCTID`

  The device's product/model ID doesn't match the license.

- `error ERROR_DUPLICATE`

  The license can't be installed because it has the same serial number as a managed license.

- `error ERROR_FILE_NOT_FOUND`

  No managed license matches the client request.

- `error ERROR_HASH_GENERATION`

  The signing hash used by the license has the wrong length or the hash operation failed.

- `error ERROR_INVALID_CERT`

  The license can't be parsed or is corrupt.

- `error ERROR_INVALID_CURRENT_TIME`

  Time-based operation can't be completed because the current time is unknown, or the trust level is too low.

- `error ERROR_LICENSE_STORE_FULL`

  No space is available to install a license.

- `error ERROR_LICENSE_TOO_BIG`

  License data is too large.

- `error ERROR_NOMEM`

  Memory allocation failed. Retry may succeed.

- `error ERROR_OPTS_NOT_SUPPORTED`

  Set options aren't allowed for the current client.

- `error ERROR_OVERFLOW`

  Request would cause an overflow in an internal calculation.

- `error ERROR_PFMFILER_GETFILECONTENTS_FAILED`

  A managed license exists but can't be read.

- `error ERROR_NO_LICENSES`

  Get or check failed because no licenses are available.

License restriction codes

- `const uint64 LICENSE_HAS_DEVICEID = 0x00000004`

  Device ID must match the value in the license
- `const uint64 LICENSE_HAS_HWVERSION = 0x00000002`

  The hardware version must match the value in the license
- `const uint64 LICENSE_HAS_LICENSEEHASH = 0x00000010`

  Client ISV hash must match the value in the license
- `const uint64 LICENSE_HAS_OEMID = 0x00000001`

  OEM ID must match the value in the license
- `const uint64 LICENSE_HAS_PKHASH = 0x00000008`

  Device PKHash must match the value in the license
- `const uint64 LICENSE_HAS_PRODUCTID = 0x00000040`

  OEM product ID must match the value in the license
- `const uint64 LICENSE_HAS_VALIDTIME = 0x00000020`

  The current time must match the value specified in the license.

3 Device attestation

Device attestation feature provides cryptographically signed and encrypted data items describing the security state of the device and its software, which are useful for risk engines and other similar applications.

Device attestation is used in conjunction with Qualcomm WES business-to-business (B2B) cloud APIs for verification and retrieval of the data items in attestation reports. See Qualcomm Wireless Edge Services (WES) Business-to-Business (B2B) API Reference for more information.

The contents of an attestation report are:

- Device identifier
- Client-provided application data
- Real time integrity check (RTIC)
- Trusted time (TTIME)
- Trusted location (TLOC)
- Device information
- Device security state
- Connection health report

See Attestation report data format for more information.

3.1 Device attestation API reference

The device attestation service generates an attestation report, which consists of the state information of the device. The attestation report is signed and encrypted and can be transmitted over unsecure networks.

To get information that is present in the attestation report, Qualcomm WES provides B2B cloud APIs. The application that generates a device attestation report sends the report to a cloud component, which then uses the attestation report.

An application can add user-data in the form of key-value pairs while generating an attestation report.

See Qualcomm WES APIs for examples code on how to generate a device attestation report.

The device attestation `IDeviceAttestation.idl` file exposes the following interfaces:

- `IDeviceAttestation`: Starts creation of a new device attestation report.
- `IAttestationBuilder`: Adds client application-specific data and builds the device attestation report.
- `IAttestationReport`: Fetches the bytes of the built attestation reports.
- `IDAError`: Error constants used by all methods in the IDL.

Device attestation feature license/certificate

The device attestation service requires two feature license/certificates.

First, a valid feature license/certificate is required to open the device attestation service object. The feature license/certificate is passed to the `createServiceById` method of the `IQWESTAServices` interface. See Qualcomm WES trusted application services API reference for more information.

Second, a special feature license/certificate is required to encrypt the generated attestation report. This feature license/certificate is also referred to as a token certificate.

For more information on how to fetch the token certificates, see Qualcomm Wireless Edge Services (WES) Business-to-Business (B2B) API Reference.

For testing, the token certificate can be downloaded from:
https://api.qwes.qualcomm.com/api/d2c/1.0/token-certificates/RSA.

The token certificate has an expiration date. In the production code, the applications must have a mechanism to refresh the token certificate before it expires.

It's also possible for the token certificate to be revoked if the private key is exposed (rarely). In this case invoking the B2B API may result in a "`Token decryption failed`" message. The application must then refresh the token certificate and retry.

Sequence diagrams

The device attestation API sequence is as follows:

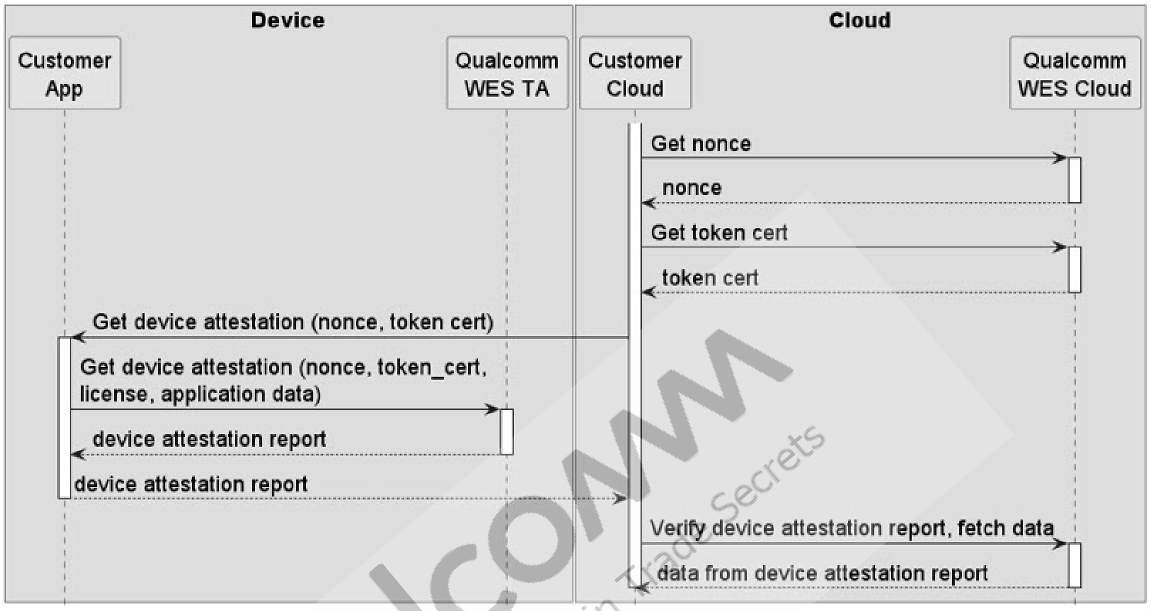

Device
Cloud
Customer App
Qualcomm WES TA
Customer Cloud
Qualcomm WES Cloud
Get nonce
nonce
Get token cert
token cert
Get device attestation (nonce, token cert)
Get device attestation (nonce, token_cert, license, application data)
device attestation report
device attestation report
Verify device attestation report, fetch data
data from device attestation report

Figure : Recommended sequence to use device attestation

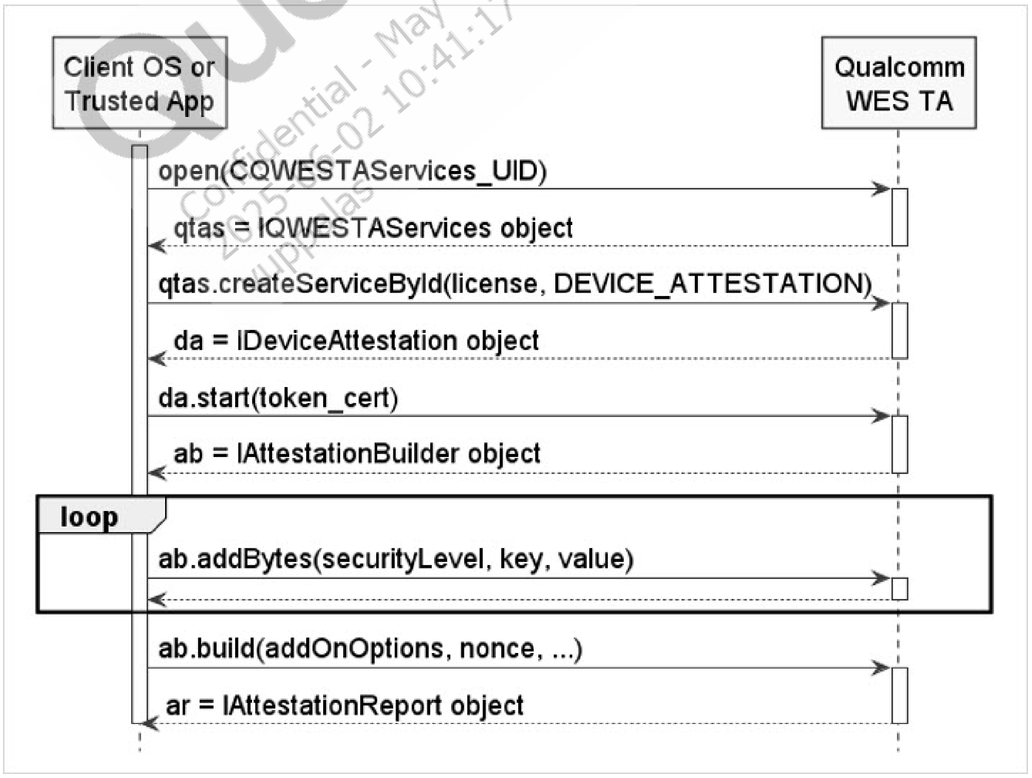

Client OS or Trusted App
Qualcomm WES TA
open(CQWESTAServices_UID)
qtas = IQWESTAServices object
qtas.createServiceById(license, DEVICE_ATTESTATION)
da = IDeviceAttestation object
da.start(token_cert)
ab = IAttestationBuilder object
loop
ab.addBytes(securityLevel, key, value)
ab.build(addOnOptions, nonce, ...)
ar = IAttestationReport object

Figure : Device attestation API sequence [1 of 2]

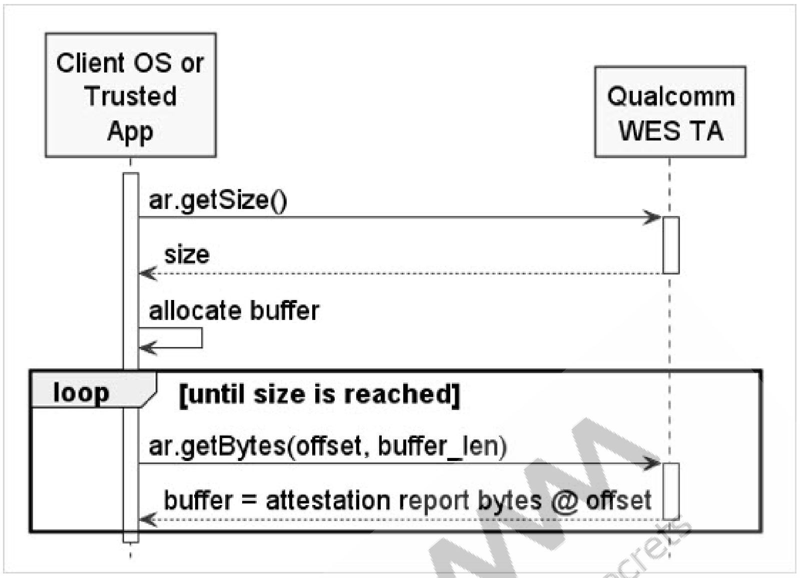


Figure : Device attestation API sequence [2 of 2]

IDeviceAttestation methods

An object of the type IDeviceAttestation is returned when the device attestation service is opened.

This interface contains a method to start creating an attestation service.

start

```
method start(
    in buffer licenseCert,
    out IAttestationBuilder attestationBuilder)
```

Starts creating an attestation report.

Parameters

in	licenseCert	Buffer that contains a license certificate.
out	attestationBuilder	AttestationBuilder object to construct an attestation.

Returns

- Object_OK on success
- IDAError error codes otherwise

getWarmUpStatus

```
method getWarmUpStatus(out buffer warmUpStatus)
```

Returns a CBOR structure containing the timestamp of the cached report corresponding to one or more of the following add-on options.

- `OPT_ADDON_LOCATION`
- `OPT_ADDON_RTIC`
- `OPT_ADDON_TRUSTEDTIME`
- `OPT_ADDON_CONNSEC_CELLULAR`

Note: `OPT_ADDON_RTIC` isn't supported.

The timestamp is the seconds elapsed since the UNIX epoch (January 1, 1970) when the cache was refreshed.

See Concise binary object representation for more information.

CDDL for warmUpStatus

```
warmup_status = {
    "RTIC TA": {
        rtic => timestamp / "UNAVAILABLE",
        connsec_cellular => timestamp / "UNAVAILABLE"
    },
    "TLOC": {
        "Trusted Location": timestamp / "UNAVAILABLE",
    },
    "TTIME": {
        "Trusted Time": timestamp / "UNAVAILABLE",
    }
}

rtic = -77200
connsec_cellular = -77280

timestamp = uint
```

Example CDN for warmUpStatus

```
{
    "RTIC TA": {
        -77200: 1627945558,
        -77280: 1627945558
    },
      "TLOC": {
        "Trusted Location": 1627943682
    },
    "TTIME": {
        "Trusted Time": 1627945544
    }
}
```

Parameters

out	warmUpStatus	Warm up status as CBOR bytes.

Returns

- `Object_OK` on success
- `WARM_UP_FAILURE` on any error while fetching the warm-up status
- `IDAError` error codes otherwise

warmUp

```
method warmUp(in uint64 options, in uint64 timeout, in interface
callback)
```

Triggers an update to the reports corresponding to one or more of the following add-on options:

- `OPT_ADDON_LOCATION`
- `OPT_ADDON_RTIC`
- `OPT_ADDON_TRUSTEDTIME`
- `OPT_ADDON_CONNSEC_CELLULAR`

Note: `OPT_ADDON_RTIC` isn't supported.

This method triggers a warm-up and returns immediately. A warm-up may take up to 15 s to complete. To know if the warm-up was completed successfully, wait for a few seconds then call the `getWarmUpStatus`. If the timestamp is greater than an earlier call to `getWarmUpStatus`, then the corresponding option was refreshed successfully.

Parameters

in	options	OPT_ADDON_* flags
in	timeout	Not used: set to 0
in	callback	Not used: set to Object_NULL

Returns

- Object_OK on success
- WARM_UP_FAILURE on any error while warming up data
- IDAError error codes otherwise

IAttestationBuilder

IAttestationBuilder provides an interface to build the attestation using IDeviceAttestation.

Constants

Security levels

Security levels are tagged to various pieces of information carried in the device attestation report as well as to the user-specified data added using addBytes.

For more information on how to query from a verified device attestation report, see Qualcomm Wireless Edge Services (WES) Business-to-Business (B2B) API Reference.

- const uint32 SECURITY_LEVEL_UNRESTRICTED = 1

  Tagged to data that originates or added by non-secure entities. All Linux data fall under this category. This included data is added by Linux applications using addBytes.

- const uint32 SECURITY_LEVEL_SECURERESTRICTED = 3

  Tagged to data that originates from software running on Qualcomm TEE, which includes OEM authored TA.

Add-ons

Provide options to select add-on features. Add-on features are optional information or data that can be included in the generated attestation report.

- `const uint64 OPT_ADDON_NONE = 0x0000000000000000`

  No additional information is included, which is the default value.

- `const uint64 OPT_ADDON_LOCATION = 0x0000000000000001`

  Include cached trusted location information.

  – If the build method of the IAttestationBuilder interface is called by an Android application with this add-on option enabled then the Linux application must have the `ACCESS_FINE_LOCATION` permission.

  – If the build method is called by another TA, then the location permission isn't checked. The TA must ensure that its client has the appropriate permissions.

  – For all other callers, the location permission isn't checked; it's the responsibility of the calling entity to ensure that the location access is granted.

- `const uint64 OPT_ADDON_RTIC_LEGACY = 0x0000000000000002`

  Deprecated

- `const uint64 OPT_ADDON_TRUSTEDTIME = 0x0000000000000004`

  Include trusted time information sourced from modem, GPS, or NTP servers along with an applicable trust level.

- `const uint64 OPT_ADDON_CONNSEC_CSR = 0x0000000000000008`

  Include connection security CSR that covers information about threats in the phone network.

- `const uint64 OPT_ADDON_TMEHW = 0x0000000000000010`

  On chipsets that support the trust management engine (TME) hardware, include a report generated by the TME.

- `const uint64 OPT_ADDON_QRKS_WITH_BASIC_INTEGRITY = 0x0000000000000020`

  Include Qualcomm runtime kernel security (QRKS) basic integrity report.

- `const uint64 OPT_ADDON_QRKS_WITH_KP_INCIDENT = 0x0000000000000040`

  Include QRKS kernel integrity report with kernel protect (KP) incidents.

- `const uint64 OPT_ADDON_PERIPHERAL = 0x0000000000000080`

  Include a peripheral status report.

- `const uint64 OPT_ADDON_CONNSEC_CDR = 0x0000000000000100`

  Include connection security cellular detailed report (CDR) that covers detailed information about threats in the phone network.

Attestation context

One of these constants is passed as the attestationContext argument of the IAttestationBuilder's build method.

- `const uint32 ATTESTATION_CONTEXT_GENERIC = 1`

  The generic attestation context must be set when this `IAttestationBuilder` object is created using the IDeviceAttestation's start method.

- `const uint32 ATTESTATION_CONTEXT_REGISTRATION = 2`

  Not used.

- `const uint32 ATTESTATION\_CONTEXT\_PROVISIONING = 3`

  The provisioning attestation context must be set when this `IAttestationBuilder` object is created using the IProvisioning's `generateAttestation` method.

- `const uint32 ATTESTATION_CONTEXT_CERT_ISSUANCE = 4`

  Not used.

- `const uint32 ATTESTATION_CONTEXT_PROOF_OF_POSSESSION = 5`

  Not used.

- `const uint32 ATTESTATION_CONTEXT_LICENSING = 6`

  Not used.

Methods

addBytes

```
method addBytes(
    in uint32 securityLevel,
    in int8[] label,
    in buffer bytes
)
```

Adds client/user-specific data.

Data is any byte array. To add, provide a unique string label to the data and call this method. To add more data, call this method more than once with unique string labels, which creates multiple entries. If this method is called from a TA, then securityLevel can be either set to `SECURITY_LEVEL_UNRESTRICTED` or `SECURITY_LEVEL_SECURERESTRICTED`. The TA must add data sourced from within the TA as `'secure restricted'`. Any data that the TA receives from the OS (Linux Android, Linux Embedded, or ThreadX) must be added as `'unrestricted'`. If this method is called from a Linux Android, Linux Embedded, or ThreadX application then securityLevel can only be set to `SECURITY_LEVEL_UNRESTRICTED`.

Up to a total of 50 KiB of label, bytes, and overhead across all entries (all calls to `addBytes`) can be added using this method. Overhead is up to 10 bytes per entry. Each entry has an additional limit of 20 KiB due to the underlying IPC mechanism. Depending on the chip product, this limit may be lower.

Parameters

`in`	`securityLevel`	One of `SECURITY_LEVEL_*`.
`in`	`label`	String to tag the item. The string does not have to be null-terminated.
`in`	bytes	Data bytes to be added.

Returns

- `Object_OK` on success
- `IDAError` error codes otherwise

build

```
method build(
    in uint32 attestationContext,
    in uint64 ADDONOptions,
    in uint32 formatType,
    in uint32 keyType,
    in buffer nonce,
    out uint64 reportStatus
    out IAttestationReport attestationReport
)
```

Builds the attestation report and returns an object of type IAttestationReport that represents the generated attestation report.

This method takes a 32-byte nonce. Set to a value based on the Qualcomm WES Cloud B2B customer policy, which can either be a value valid only for a limited time or statically assigned. If the value is valid only for a limited time, then the application must query the cloud to get a fresh nonce before calling this method.

Parameters

in	attestationContext	One of the ATTESTATION_CONTEXT_* values. See Constants Attestation context for more information.
in	ADDONOptions	OPT_ADDON_* flags. See Constants Add-ons for more information.
in	formatType	Set to FORMAT_EAT (1).
in	keyType	Set to KEY_QDAK (2)
in	nonce	A buffer that contains a 32-byte nonce.
out	reportStatus	Bitmap that shows which addon options were successfully fetched.
out	attestationReport	AttestationReport object that represents an attestation report.

Returns

- Object_OK on success
- IDAError error codes otherwise

clearBytes

```
method clearBytes()
```

Clears all data added by addBytes

Returns

Object_OK on success

IAttestationReport methods

The IAttestationReport method provides an interface to get the attestation bytes using IAttestationBuilder.

getBytes

```
method getBytes(in uint64 offset, out buffer attestation)
```

Gets the bytes in the attestation.

Parameters

in	offset	Attestation offset
out	attestation	Attestation buffer

Typical use is to get the size and then call this size in a loop with a 4 kB buffer incrementing the offset by the number of bytes returned until all the bytes have been fetched.

The caller should track the bytes fetched to know when they have got them all. There is no “end of file” return code.

A buffer larger than 4 kB may be used with some implementations. If the buffer is too large for the underlying IPC mechanism, then an error is returned.

An error is returned if the offset isn’t in the range of 0 and the size of the attestation.

To fetch the data present in the attestation report, the application must upload the report to a cloud component, which in turn uses Qualcomm WES Cloud B2B APIs.

Returns

- `Object_OK` on success
- `IDAError` error codes otherwise

getSize

```
method getSize(out uint64 attestationReportSize)
```

Gets the full size of the signed and encrypted attestation.

Parameters

out	attestationReportSize	Size of the attestation report.

Returns

- `Object_OK` on success
- `IDAError` error codes otherwise

IDAError constants

This section lists the IDAError constants.

This interface contains error codes returned by `IDeviceAttestation`, `IAttestationBuilder`, and `IAttestationReport` methods.

- `error NO_MEMORY`

  Failure during memory allocation.

- `error INVALID_BUFFER`

  Null or zero-length buffer is passed.

- `error INVALID_CERTIFICATE`

  The passed certificate is invalid or it doesn’t contain the correct feature ID.

- `error MAX_APP_DATA_LIMIT_REACHED`

  A maximum of 50 kB cumulative app data can be added to generate a token.

- `error INVALID_SECURITY_LEVEL`

  Passed security level isn’t within the expected range or not allowed.

- `error INVALID_ATTESTATION_CONTEXT`

  Passed attestation context isn’t within the expected range.

- `error INVALID_SIGNING_KEY`

  Passed key type isn’t allowed, or fetching the key to sign the attestation report based on the key type failed.

- `error INVALID_NONCE`

  An invalid nonce buffer or a nonce with an invalid length is passed.

- `error INVALID_REPORT_OFFSET`

  Offset exceeds the attestation report size.

- `error ATTESTATION_REPORT_FAILURE`

  Generic error returned by any step while building the attestation report.

- `error WARM_UP_FAILURE`

  Generic error returned by any step while warming up any submod and retrieving its status.

- `error INVALID_REPORT_OFFSET`

  Offset exceeds the attestation report size.

- `error ATTESTATION_REPORT_FAILURE`

Generic error returned by any step while building the attestation report.

4 Secure provisioning

The Qualcomm WES secure provisioning service enables the generation and use of cryptographic keys that are secured using device unique keys.

These keys are used for:

- Securely provisioning data to the device post-sale and over-the-air (OTA).
- Signing data on the device.

The secure provisioning service also provides a persistent and secure key storage feature, for the keys generated by the secure provisioning APIs.

4.1 Secure provisioning API reference

The secure provisioning service is exposed by the `IProvisioning`interface and the key store service by the `IQWESKeyStore` interface.

A valid feature license/certificate is required to use the secure provisioning service. For more information about how to open `IProvisioning` and `IQWESKeyStore` interfaces, see Qualcomm WES trusted application services API reference.

Use cases

Secure provisioning APIs on the device provide methods for key generation and crypto operations.

The APIs, currently, enable three use cases.

The typical use of the secure provisioning service requires a customer-authored client application running on a Qualcomm WES-enabled device and a corresponding customer cloud. The following table lists the three use cases supported by the secure provisioning service.

Table : Secure provisioning use cases

	Use case	What does the customer cloud receive from the Qualcomm WES cloud?	What does the customer-authored client application use the secure provisioning APIs for?
1	Signature with RSASSA or ECDSA	A Qualcomm WES cloud-signed X509 certificate with the public key generated on the device	Signing the data is intended for the customer cloud to verify.
2	Decryption with RSAES		Decrypting the data sent by the customer cloud.
3	ECDH key derivation with AES decryption		Deriving a content encryption key (CEK) using ECDH, and then using that CEK to decrypt the data sent by the customer cloud.

In all the three use cases, the customer-authored client application is responsible to fetch the encrypted data from, or send the signed data to, the customer cloud.

Signature using RSASSA or ECDSA

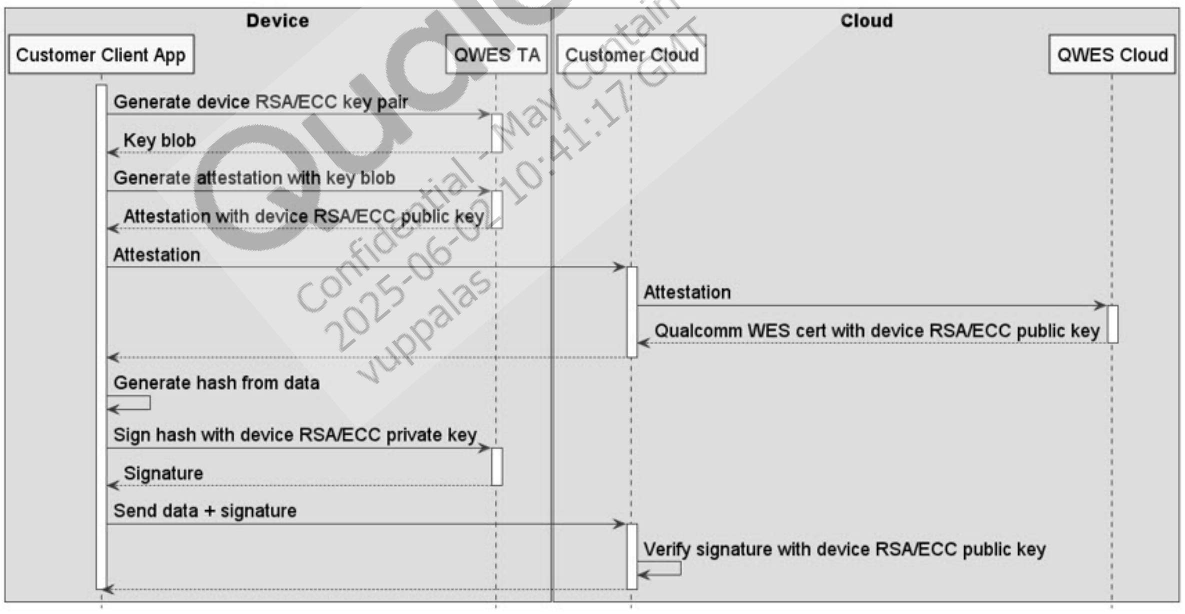

Device
Cloud
Customer Client App
QWES TA
Customer Cloud
QWES Cloud
Generate device RSA/ECC key pair
Key blob
Generate attestation with key blob
Attestation with device RSA/ECC public key
Attestation
Attestation
Qualcomm WES cert with device RSA/ECC public key
Generate hash from data
Sign hash with device RSA/ECC private key
Signature
Send data + signature
Verify signature with device RSA/ECC public key

Figure : Signature using RSASSA or ECDSA sequence diagram

Note: RSASSA can be used for signing data with the same sizes as the accepted digest (hash) algorithms are supported. ECDSA can be used for signing data smaller than or with the same size as the key.

Verify ECDSA signature

To verify the signature generated using the ECDSA algorithm, use the following OpenSSL command:

```
openssl pkeyutl -inkey verify_key.pkcs8.der -keyform
DER -sigfile signature.der -in tbs.bin -verify
```

Where,

- `verify_key.pkcs8.der` is the PKCS8 formatted, DER encoded file containing ECC public key of the device.
- `signature.der` is the file containing the ASN.1 formatted signature bytes.
- `tbs.bin` is the file containing a to-be-signed byte.

Verify RSASSA signature

To verify the signature generated using the RSASSA algorithm, use the following OpenSSL command:

```
openssl pkeyutl -inkey verify_key.pkcs8.der -keyform DER \
    -in tbs.bin -sigfile signature.bin -verify \
    -pkeyopt rsa_padding_mode:pss \
    -pkeyopt rsa_pss_saltlen:32 \
    -pkeyopt rsa_mgf1_md:sha256 \
    -pkeyopt digest:sha256
```

Where,

- `verify_key.pkcs8.der` is the PKCS8 formatted, DER encoded file containing the RSASSA public key of the device.
- `signature.der` is the file containing the ASN.1 formatted signature bytes.
- `tbs.bin` is the file containing a to-be-signed byte. This file must be 32-byte long in this case as the secure provisioning API and OpenSSL expects the to-be-signed data to be already hashed.

Decryption using RSAES (asymmetric decryption)

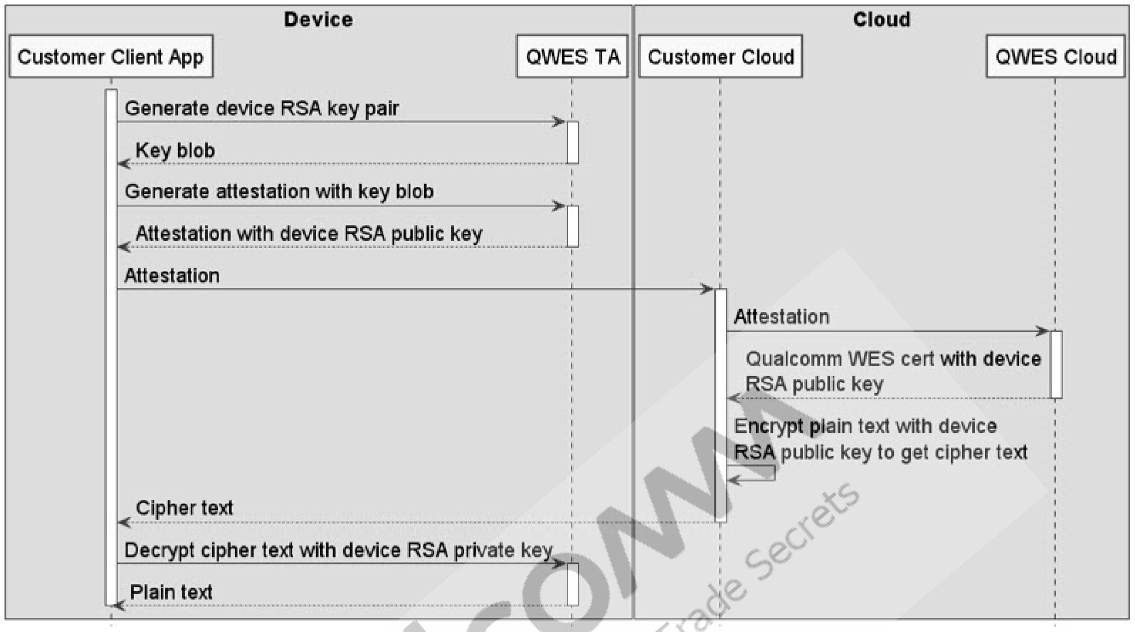

Device
Cloud
Customer Client App
QWES TA
Customer Cloud
QWES Cloud
Generate device RSA key pair
Key blob
Generate attestation with key blob
Attestation with device RSA public key
Attestation
Attestation
Qualcomm WES cert with device RSA public key
Encrypt plain text with device RSA public key to get cipher text
Cipher text
Decrypt cipher text with device RSA private key
Plain text

Figure : Decryption using RSAES sequence diagram

The maximum size of the plain text that can be encrypted using RSAES depends on the selected key size and hashing algorithm. For example, if the key size is 4096 bits (512 bytes) and the hashing algorithm is 256 bits (32 bytes) then the plain text can be a maximum of 446 bytes. If the size of the plain text is larger, see ECDH key derivation with AES decryption instead.

To test this use case with OpenSSL, see Test decryption using RSAES (asymmetric decryption).

ECDH key derivation with AES decryption

Customer cloud-side CEK derivation

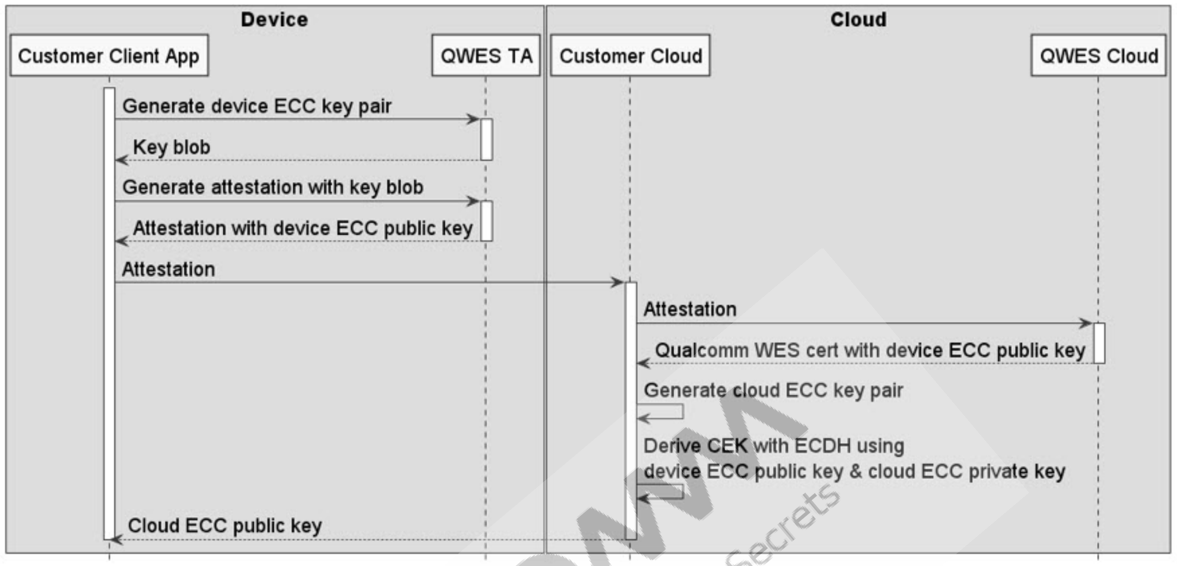

Device
Cloud
Customer Client App
QWES TA
Customer Cloud
QWES Cloud
Generate device ECC key pair
Key blob
Generate attestation with key blob
Attestation with device ECC public key
Attestation
Attestation
Qualcomm WES cert with device ECC public key
Generate cloud ECC key pair
Derive CEK with ECDH using device ECC public key & cloud ECC private key
Cloud ECC public key

Figure : ECDH key derivation with AES decryption sequence diagram [1/2]

Device-side CEK derivation and decryption

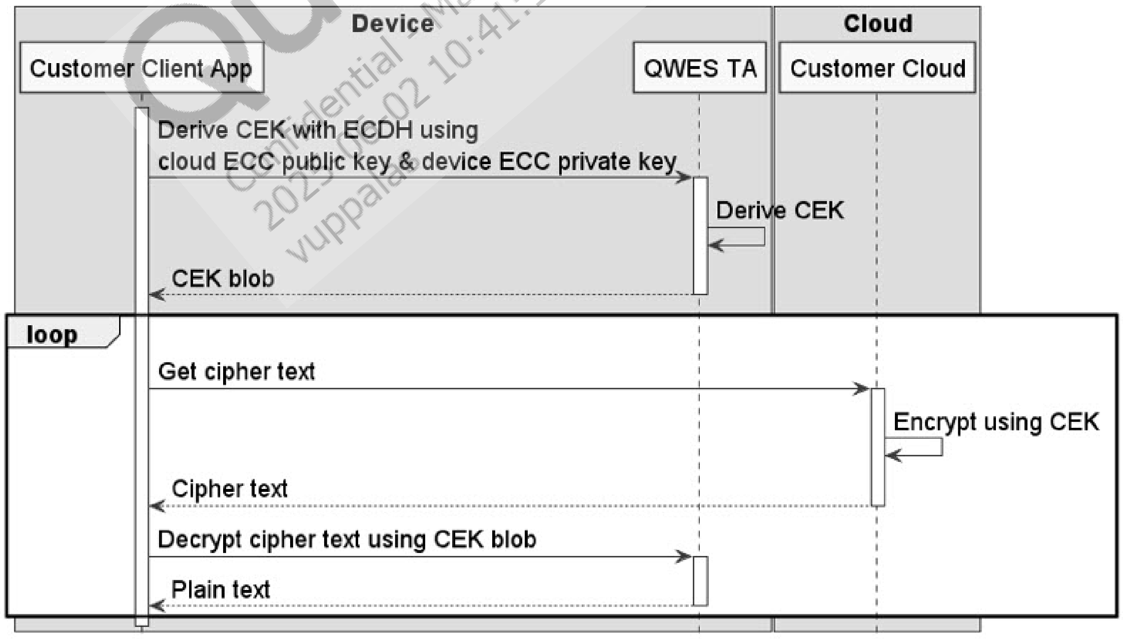


Figure : ECDH key derivation with AES decryption sequence diagram [2/2]

Sequence diagrams

The following sequence digrams enable you to generate a key and attestation, sign the data, decrypt the data, and derive a symmetric key.

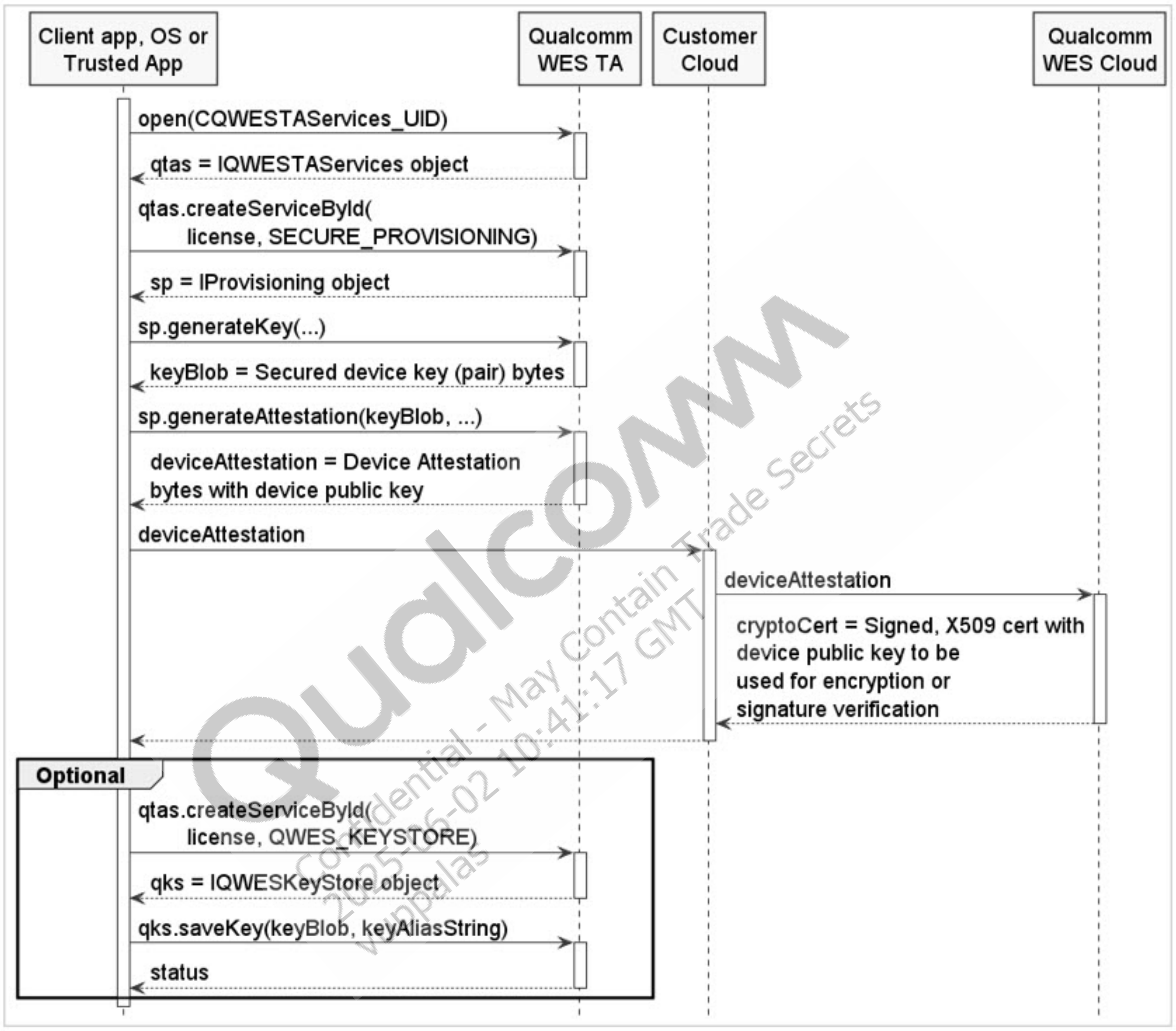

Client app, OS or Trusted App
Qualcomm WES TA
Customer Cloud
Qualcomm WES Cloud
open(CQWESTAServices_UID)
qtas = IQWESTAServices object
qtas.createServiceById(license, SECURE_PROVISIONING)
sp = IProvisioning object
sp.generateKey(...)
keyBlob = Secured device key (pair) bytes
sp.generateAttestation(keyBlob, ...)
deviceAttestation = Device Attestation bytes with device public key
deviceAttestation
deviceAttestation
cryptoCert = Signed, X509 cert with device public key to be used for encryption or signature verification
Optional
qtas.createServiceById(license, QWES_KEYSTORE)
qks = IQWESKeyStore object
qks.saveKey(keyBlob, keyAliasString)
status

Figure : Generate a key (pair), generate attestation and optionally save the key

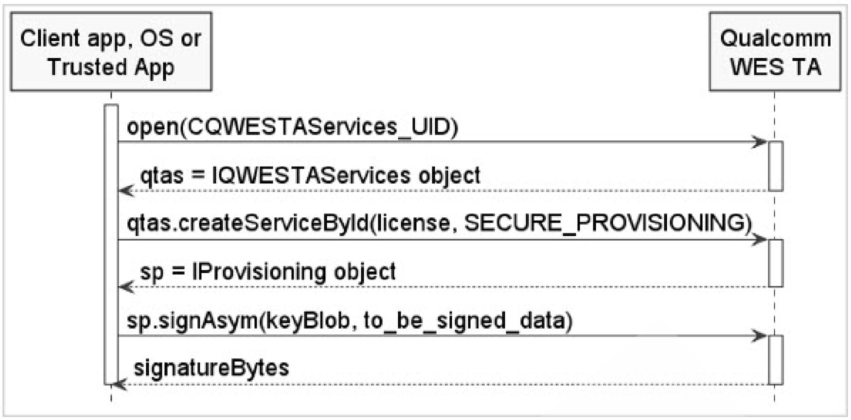

Client app, OS or Trusted App
Qualcomm WES TA
open(CQWESTAServices_UID)
qtas = IQWESTAServices object
qtas.createServiceById(license, SECURE_PROVISIONING)
sp = IProvisioning object
sp.signAsym(keyBlob, to_be_signed_data)
signatureBytes

Figure : Sign data using an asymmetric key

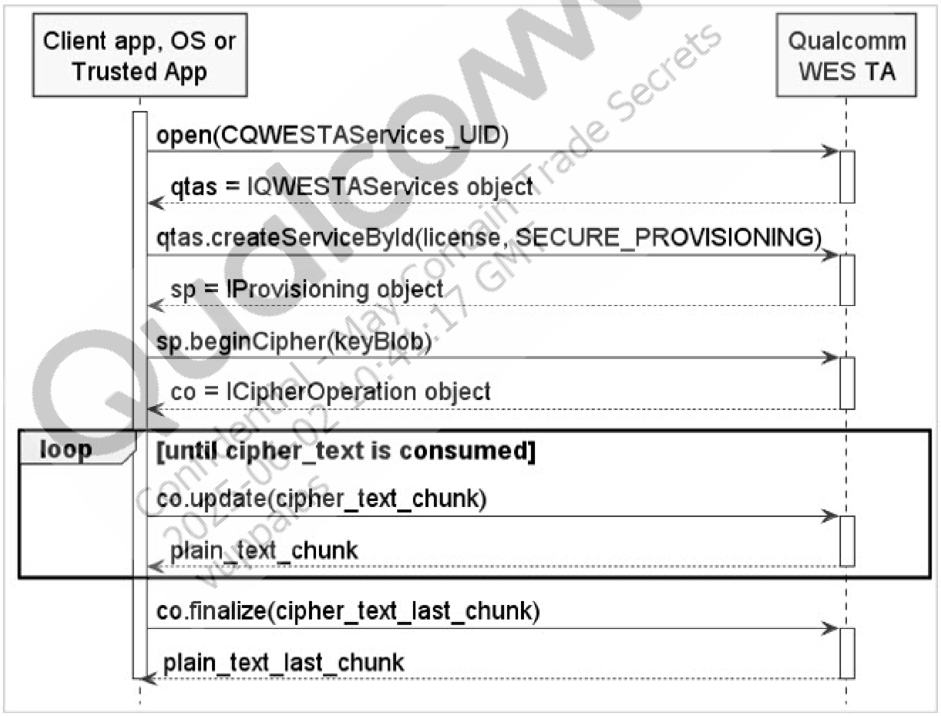

Client app, OS or Trusted App
Qualcomm WES TA
open(CQWESTAServices_UID)
qtas = IQWESTAServices object
qtas.createServiceById(license, SECURE_PROVISIONING)
sp = IProvisioning object
sp.beginCipher(keyBlob)
co = ICipherOperation object
loop
[until cipher_text is consumed]
co.update(cipher_text_chunk)
plain_text_chunk
co.finalize(cipher_text_last_chunk)
plain_text_last_chunk

Figure : Decrypt data using an asymmetric key

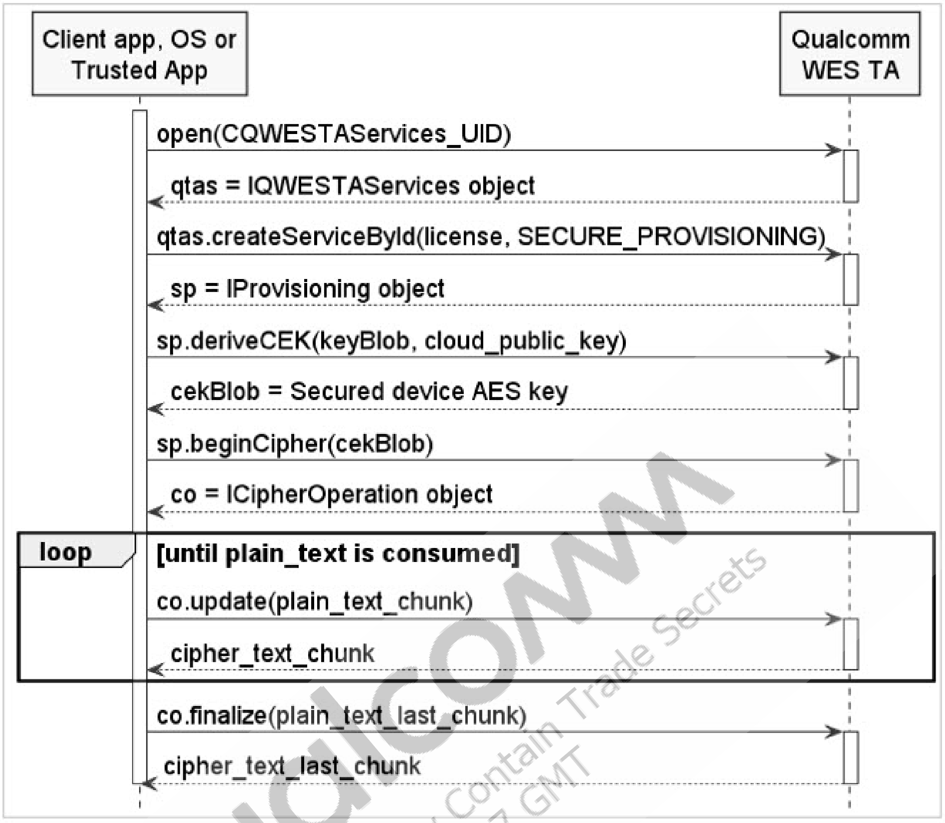

Client app, OS or Trusted App
Qualcomm WES TA
open(CQWESTAServices_UID)
qtas = IQWESTAServices object
qtas.createServiceById(license, SECURE_PROVISIONING)
sp = IProvisioning object
sp.deriveCEK(keyBlob, cloud_public_key)
cekBlob = Secured device AES key
sp.beginCipher(cekBlob)
co = ICipherOperation object
loop
[until plain_text is consumed]
co.update(plain_text_chunk)
cipher_text_chunk
co.finalize(plain_text_last_chunk)
cipher_text_last_chunk

Figure : Derive a symmetric key and use it to decrypt data

Crypto support

The secure provisioning service supports the following cryptographic operations and properties:

Figure : Crypto operations and properties supported by secure provisioning

Algorithm	Purpose	Properties
RSASSA	Signature	4096-bit key, PSS 32-byte input and MGF1
RSAES	Decryption	4096-bit key, OAEP 32-byte input and MGF1
ECDSA	Signature	521-NIST named curve Less than or equal to 64-byte input
ECDH	Derivation	521-NIST named curve 64-byte secret derivation AES 256-bit CEK using HKDF
AES	Decryption	256-bit CEK, GCM 128-bit authentication tag 12-byte caller-provided IV

Key store

The keys generated by the secure provisioning service can be optionally stored in a persistent key storage, which is identified by the client-provided aliases.

This functionality is provided by the `IQWESKeyStore` interface.

Note: Only RPMB is supported as the underlying persistent storage for the key store service. This service is supported on platforms that have the RPMB storage.

Securing generated keys

Generated RSA, ECC, and AES keys are encrypted using device unique keys.

Thus, the keys are bound to the security state and hardware of the device that they were generated on. In addition, the keys are also bound to the credentials of the caller, such as the hash of an Android Java application, or a DID of a trusted application.

An encrypted key blob is returned to the caller, which contains the encrypted key material. The key blob can be safely stored on an unsecure FS. The key blobs can be optionally stored in the provided key store as well.

IProvisioning

The `IProvisioning` interface contains methods to generate a key, generate an attestation that contains a generated key, sign data, and start a crypto session.

The key and algorithm parameters, such as the key size and padding are passed to these methods using CBOR-formatted tag-value pairs.

Constants

Key type

The type of key to generate. This constant is given as the `keyType` argument of the `generateKey` method.

- `const uint32 KEY_TYPE_RSA = 0`

  Generate an RSA key.

- `const uint32 KEY_TYPE_ECC = 1`

  Generate an ECC key.

CEK type

The type of content encryption key (CEK) to generate. The constant is given as the `cekType` argument of the `deriveCEK` method.

- `const uint32 CEK_TYPE_AES = 0`

  Generate an AES CEK.

Key purpose

The cryptographic purpose for which the generated key is used. These constants are applicable as values of `TAG_KEY_PURPOSE`.

- `const uint32 KEY_PURPOSE_DECRYPT = 1`

  The generated key can be used for decryption operation.
- `const uint32 KEY_PURPOSE_SIGN = 2`

  The generated key can be used for signature operation.
- `const uint32 KEY_PURPOSE_DERIVE_KEY = 4`

  The generated key can be used for deriving a CEK.

RSA key size

The size of the RSA key to generate. These constants are applicable as values of `TAG_KEY_SIZE`.

- `const uint32 RSA4096_KEY_SIZE = 4096`

  A 4096 (512-byte) RSA key is generated.

RSA public exponent

The public exponent of the RSA key to generate. These constants are applicable as values of `TAG_RSA_PUBLIC_EXPONENT`.

- `const uint32 RSA65537_PUB_EXP = 65537`

  65537 will be used as the RSA public exponent.

RSA padding

The type of RSA padding to associate the generated RSA key. Only one of the corresponding padding types can be used while decrypting or signing data using the generated key. These constants are applicable as values of `TAG_PADDING`.

- `const uint32 PAD_RSA_OAEP = 1`

  Associate the generated RSA key with OAEP padding.

- `const uint32 PAD_RSA_PSS = 2`

  Associate the generated RSA key with PSS padding.

Digest algorithm

The MGF1 digest algorithm to associate the generated RSA key. Only one of the corresponding digest algorithms can be used while decrypting or signing data using the generated key. These constants are applicable as values of `TAG_RSA_MGF1_DIGEST`.

- `const uint32 DIGEST_SHA_2_256 = 4`

  Associate the generated RSA key with the SHA-256 digest algorithm.

- `const uint32 DIGEST_SHA_2_384 = 5`

  Associate the generated RSA key with the SHA-384 digest algorithm.

- `const uint32 DIGEST_SHA_2_512 = 6`

  Associate the generated RSA key with the SHA-512 digest algorithm.

EC curve

The type of the named curve of the EC key to generate. These constants are applicable as values of `TAG_EC_CURVE`.

- `const uint32 EC_CURVE_P_224 = 0`

  Generate an EC NIST P-224 key.

- `const uint32 EC_CURVE_P_256 = 1`

  Generate an EC NIST P-256 key.

- `const uint32 EC_CURVE_P_384 = 2`

  Generate an EC NIST P-384 key.

- `const uint32 EC_CURVE_P_521 = 3`

  Generate an EC NIST P-521 key.

AES key size

The size of the AES CEK to be derived. These constants are applicable as values of `TAG_KEY_SIZE`.

- `const uint32 AES256_KEY_SIZE = 256`

  Derive a 256-bit (32-byte) AES CEK.

AES block mode

The block mode to associate the derived AES CEK. Only one of the corresponding modes can be used when decrypting data using the derived AES CEK. These constants are applicable as values of `TAG_BLOCK_MODE`.

- `const uint32 BLOCK_MODE_GCM = 32`

  Associate the derived AES CEK with GCM block mode.

Tag

Tags are used as labels in the label-value pairs used to construct CBOR-formatted key and algorithm parameters. For example, `TAG_KEY_SIZE` is the label used for specifying the size of the key to be generated.

- `const uint32 TAG_KEY_PURPOSE = 1`

  The purpose of the generated key. See Key purpose for the supported values.

- `const uint32 TAG_KEY_SIZE = 2`. The size of the generated key or derived CEK. The possible values that can be used for RSA keys are listed in RSA key size, and the ones for AES CEKs are listed in AES key sizes.

- `const uint32 TAG_PADDING = 3`

  The padding to be associated with the generated key. This is only applicable to RSA keys. See RSA padding for the supported values.

- `const uint32 TAG_RSA_PUBLIC_EXPONENT = 5`

  The public exponent of the generated RSA key. See RSA public exponent for the supported values.

- `const uint32 TAG_BLOCK_MODE = 7`

  The block mode to associate with the derived CEK. This is only applicable to AES keys. See AES block mode for the supported values.

- `const uint32 TAG_EC_CURVE = 8`

The type of the named curve of the generated EC key. See EC curve for the supported values.

- `const uint32 TAG_NONCE = 9`

  The caller-provided initialization vector (IV) or nonce. This is only applicable to AES keys when `TAG_BLOCK_MODE` is set to `BLOCK_MODE_GCM`. The nonce value gets passed in the inParams argument to the beginCipher method.

- `const uint32 TAG_RSA_MGF1_DIGEST = 12`

  The MGF1 digest algorithm to associate with the generated RSA key. This is used as the digest algorithm for MGF1 when decrypting or signing data using the generated key. See Digest algorithm for the supported values.

Methods

generateKey

```
method generateKey(
    in uint32 keyType,
    in buffer keyParams,
    out buffer keyBlob);
```

Generates a key pair based on the input algorithm, and key parameters.

`keyParams` is a CBOR-formatted data consisting of tag-value pairs. The following is the list of the supported key parameters. The constants are defined in Constants.

- CDN of RSA key parameters for the signature use case (RSASSA):

```
rsa_sign_key_params = {
    TAG_KEY_PURPOSE => [ KEY_PURPOSE_SIGN ],
    TAG_KEY_SIZE => RSA4096_KEY_SIZE,
    TAG_PADDING => [ PAD_RSA_PSS ],
    TAG_RSA_PUBLIC_EXPONENT => RSA65537_PUB_EXP,
    TAG_RSA_MGF1_DIGEST => [ DIGEST_SHA_2_256 ]
}
```

- CDN of RSA key parameters for the decryption use case (RSAES):

```
rsa_decrypt_key_params = {
    TAG_KEY_PURPOSE => [ KEY_PURPOSE_DECRYPT ],
    TAG_KEY_SIZE => RSA4096_KEY_SIZE,
    TAG_PADDING => [ PAD_RSA_OAEP ],
    TAG_RSA_PUBLIC_EXPONENT => RSA65537_PUB_EXP,
    TAG_RSA_MGF1_DIGEST => [ DIGEST_SHA_2_256 ]
}
```

- CDN of EC key parameters for the signature use case (ECDSA):

```
ec_sign_key_params = {
    TAG_KEY_PURPOSE => [ KEY_PURPOSE_SIGN ],
    TAG_EC_CURVE => EC_CURVE_P_521
}
```

- CDN of EC key parameters for the derivation use case CDDL (ECDH):

```
ec_derive_key_params = {
    TAG_KEY_PURPOSE => [ KEY_PURPOSE_DERIVE_KEY ],
    TAG_EC_CURVE => EC_CURVE_P_521
}
```

Note: The values for the tags `TAG_KEY_PURPOSE`, `TAG_PADDING`, and `TAG_RSA_MGF1_DIGEST` are arrays with only one element.

Parameters

`in`	`keyType`	Either `KEY_TYPE_RSA` or `KEY_TYPE_ECC`.
`In`	`keyParams`	CBOR-formatted key parameters, such as the key type, key purpose, key size, padding.
`out`	`keyBlob`	The buffer that contains the encrypted key material, which is bound to the caller information, the security state and hardware of the device.

Returns

- `Object_OK` on success
- `Object_ERROR` on general failure
- `IProvError` error codes otherwise

generateAttestation

```
method generateAttestation(
    in buffer licenseCert,
    in buffer keyBlob,
    in buffer serviceCtx,
    out IAttestationBuilder builder);
```

Generates an attestation report for provisioning a key.

The generated attestation report contains the public key, and key parameters present in `keyBlob`. Qualcomm WES cloud verifies this report, and generates a Qualcomm-signed certificate with the public key and key parameters of the device. This signed certificate can be used by the customer cloud for the signature verification, data encryption, or key derivation purpose with only one corresponding device as the endpoint, which is the sole owner of the `keyBlob`.

Parameters

in	licenseCert	Buffer containing the license to use the device attestation feature.
in	keyBlob	Buffer that contains the key material exactly as returned by `generateKey`.
In	serviceCtx	Bytes that make sense to the customer cloud. This information gets passed along by the Qualcomm WES cloud to the customer cloud.
out	builder	IAttestationBuilder object to add further user data, and build and get the bytes of the device attestation report. See Device attestation API reference for more information.

Returns

- `Object_OK` on success
- `Object_ERROR` on general failure
- `IProvError` error codes otherwise

deriveCEK

```
method deriveCEK(
    in uint32 cekType,
    in buffer keyBlob,
    in buffer inParams,
    in buffer cloudPublicKey,
    out buffer outParams,
    out buffer cekBlob);
```

`inParams` is a CBOR-formatted data consisting of the tag-value pairs for the key parameters that apply to the derived CEK (`cekBlob`). Currently, only the key derivation for the AES256-GCM algorithm is supported. Constants are defined in Constants.

The key parameters for deriving an AES CEK CDDL (AES256-GCM):

```
aes_cek_key_params = {
    TAG_KEY_SIZE => AES256_KEY_SIZE,
    TAG_BLOCK_MODE => BLOCK_MODE_GCM
}
```

Parameters

in	cekType	CEK algorithm type. Always CEK_TYPE_AES.
in	keyBlob	Buffer that contains the encrypted key material (private key of the device) exactly as returned by generateKey.
in	inParams	CBOR-formatted parameters, such as the AES mode, and key size.
in	cloudPublicKey	Buffer that contains the public key of the customer cloud.
out	outParams	Reserved for future use—set it to NULL.
out	cekBlob	Output buffer containing the encrypted derived CEK material.

Returns

- Object_OK on success
- Object_ERROR on general failure
- IProvError error codes otherwise

signAsym

```
method signAsym(in buffer keyBlob, in buffer tbs, out buffer
signature);
```

This method signs the given input using the key identified by keyBlob.

For RSA, the size of the to-be-signed data (tbs) must be equal to the size of the digest corresponding to the algorithm specified with TAG_RSA_MGF1_DIGEST when generateKey is called. This method doesn't hash the input, but assumes that the input is already hashed using the same algorithm as the one specified with TAG_RSA_MGF1_DIGEST.

Parameters

in	keyBlob	Buffer that contains the encrypted key material exactly as returned by generateKey.
in	tbs	To-be-signed data.
out	signature	Output buffer containing the signature.

Returns

- Object_OK on success
- Object_ERROR on general failure
- IProvError error codes otherwise

beginCipher

```
method beginCipher(
    in buffer keyBlob,
    in buffer inParams,
    out buffer outParams,
    out ICipherOperation cipherOperation);
```

This method creates an `ICipherOperation` object for decrypting data using the key identified by `keyBlob`.

`inParams` is a CBOR-formatted data consisting of the tag-value pairs for the algorithm parameters applicable to the cipher operation. For the AES256-GCM algorithm, a 12-byte IV/nonce is provided by the caller. Constants are defined in Constants.

CDDL of the algorithm parameters for decrypting data using AES (AES256-GCM):

```
aes_decrypt_params = {
    TAG_NONCE => bstr.size 12 }
```

Parameters

in	keyBlob	Buffer containing the encrypted key material as returned by `generateKey`.
in	inParams	CBOR-formatted algorithm parameters for the cipher operation.
in	outParams	Reserved for future use—set it to NULL.
out	cipherOperation	`ICipherOperation` object to decrypt data.

Returns

- `Object_OK` on success
- `Object_ERROR` on general failure
- `IProvError` error codes otherwise

ICipherOperation methods

The `ICipherOperation` interface is used to decrypt data using `keyBlob` specified in the `beginCipher` method.

To decrypt data larger than the key, call the `update` method multiple times with the right offset. When all the data is consumed, call the `finish` method. To decrypt small data (typically less than 8K), call the `finish` method without calling `update`.

Note: Decryption is performed in the Qualcomm TEE; therefore, the decryption operations must be limited to small data. Decrypting large amounts of data can lead to system slow-downs or partial failures of other subsystems that depend on Qualcomm TEE.

update

```
method update(
    in buffer input,
    in buffer inParams,
    out buffer output,
    out uint32 inputConsumed);
```

Decrypts a part of a large data.

Parameters

in	input	Buffer that contains the partial input data to decrypt.
in	inParams	Reserved for future use—set it to NULL.
out	output	Output buffer containing the decrypted data.
out	inputConsumed	The amount of `input` data consumed for producing `output`

Returns

- `Object_OK` on success.
- `Object_ERROR` on general failure.
- `IProvError` error codes otherwise.

finish

```
method finish(in buffer input, in buffer inParams, out buffer
output);
```

Performs the final decryption.

For RSA, the data to be encrypted (`input`) should be equal to the size of the digest corresponding to the algorithm specified with `TAG_RSA_MGF1_DIGEST` when calling `generateKey`.

Parameters

`in`	`input`	Buffer that contains the final input data to decrypt.
`in`	`inParams`	Reserved for future use—set it to NULL.
`out`	`output`	Output buffer containing the final decrypted data.

Returns

- `Object_OK` on success.
- `Object_ERROR` on general failure.
- `IProvError` error codes otherwise.

IProvError constants

The error codes defined in the `IProvError` interface are used by both the `IProvisioning` and `ICipherOperation` interfaces.

The error codes are:

- `error NO_MEMORY`: Failure during memory allocation.
- `error INVALID_DA_CERTIFICATE`: Invalid attestation certificate.
- `error INVALID_CLOUD_CERTIFICATE`: Invalid cloud certificate.
- `error INVALID_ARGUMENT`: Invalid arguments.
- `error INVALID_PARAMS_CBOR`: Invalid CBOR parameters.

IQWESKeyStore

The Qualcomm WES key store service provides a secure and persistent storage feature for the keys generated by the secure provisioning service.

Methods

saveKey

```
method saveKey(in buffer keyBlob, in buffer keyAlias);
```

Saves a key generated by the `generateKey` method of the `IProvisioning` interface in the secure storage (SFS).

This function fails if:

- `keyAlias` exists
- Caller doesn't have the permission to use the key identified by `keyBlob`
- Caller limit of the number of keys to store has reached

Parameters

in	keyBlob	Buffer that contains the encrypted key material exactly as generated by `generateKey`.
in	keyAlias	Per-application and caller-provided alias for the key being saved.

Returns

- `Object_OK` on success
- `Object_ERROR` on general failure
- `IQWESKeyStore` error codes otherwise

loadKey

```
method loadKey(in buffer keyAlias, out buffer keyValue);
```

Loads the key identified by `keyAlias` from the SFS.

This function fails if:

- `keyAlias` doesn't exist
- The caller doesn't have the permission to use the key identified by keyAlias

Parameters

in	keyAlias	Alias for the queried key
out	keyValue	Output buffer containing the loaded key material

Returns

- `Object_OK` on success
- `Object_ERROR` on general failure
- `IQWESKeyStore` error codes otherwise

removeKey

```
method removeKey(in buffer keyAlias);
```

Removes the key identified by `keyAlias`, and any certificate associated with it.

This function fails if:

- `keyAlias` doesn't exist.
- The caller doesn't have the permission to use the key identified by `keyAlias`.

Parameters

Returns

- `Object_OK` on success
- `Object_ERROR` on general failure
- `IQWESKeyStore` error codes otherwise

saveValue

```
method saveValue(in buffer keyAlias, in buffer valueBytes);
```

This method isn't implemented.

loadValue

```
method loadValue(in buffer keyAlias, out buffer valueBytes);
```

This method isn't implemented.

clearValue

```
method clearValue(in buffer keyAlias);
```

This method isn't implemented.

Error codes

- `error ERROR_NO_MEMORY`

  Failure during memory allocation.

- `error ERROR_INVALID_KEY_BLOB`

  Invalid key blob.

- `error ERROR_INVALID_ARGUMENT`

  Invalid arguments.

- `error ERROR_KEY_STORE_FULL`

  Storage is full.

OpenSSL

This section describes the test procedures with OpenSSL.

Test decryption using RSAES (asymmetric decryption)

1. Generate an RSA-4096 key using the `generateKey` method of the `IProvisioning` interface and save it to the `keyblob.bin` file, which contains the private and public key of the device. The CDN of the key parameters to generate this key are as follows:

   ```
   rsa_decrypt_key_params = {
       TAG_KEY_PURPOSE => [ KEY_PURPOSE_DECRYPT ],
       TAG_KEY_SIZE => RSA4096_KEY_SIZE,
       TAG_PADDING => [ PAD_RSA_OAEP ],
       TAG_RSA_PUBLIC_EXPONENT => RSA65537_PUB_EXP,
       TAG_RSA_MGF1_DIGEST => [ DIGEST_SHA_2_256 ]
   }
   ```

2. Generate a device attestation report that embeds the RSA key generated in Step 1 and save it to a file named `da.bin`. Call the `generateAttestation` method of the `IProvisioning` interface to generate the device attestation report.

3. Copy `da.bin` to a Qualcomm Linux machine (Cygwin or WSL on Windows should also work) for further steps.

4. Send the device attestation report to the stage instance of the Qualcomm WES cloud to receive a Qualcomm signed X509 certificate with the public key of the device.

   See Fetch Qualcomm signed X509 certificate for more information.

5. To encrypt test data using OpenSSL, do the following:

   a. Generate test data using the following command:

   ```
   echo Hello, world! > plain_text.bin
   ```

   b. Encrypt it using the public key of the device:

   ```
   openssl pkeyutl -inkey da_cert.pem -certin -in
   plain_text.bin -encrypt -out cipher_text.bin -pkeyopt
   rsa_padding_mode:oaep -pkeyopt rsa_mgf1_md:sha256 -pkeyopt
   rsa_oaep_md:sha256
   ```

   This command encrypts `plain_text.bin` using the public key of the device in `da_cert.pem` into `cipher_text.bin`.

6. Decrypt `cipher_text.bin` using the same `keyblob.bin` saved in Step 1 above on the device. Call the `ICipherOperation` methods to decrypt and get back the same `plain_text.bin`.

Fetch Qualcomm signed X509 certificate

To send a device attestation report to the stage instance of the Qualcomm WES cloud and get a Qualcomm signed X509 certificate with the public key of the device in return, do the following:

Note: The steps assume that the device attestation report is written to the `da.bin` file.

1. Convert `da.bin` to base64 and save it to `da.bin.b64`:

```
base64 da.bin | tee da.bin.b64
```

2. Create `da.json` with the following format:

```
{
    "attestations": [
        "<Add contents of da.bin.b64 here>"
    ]
}
```

The following command can be used to generate `da.json`:

```
awk 'BEGIN{printf "{\"attestations\":[\""}; {printf "%s", $0};
END{printf "\"]}"}' da.bin.b64 | tee da.json
```

3. Get an `id_token` from the Qualcomm WES cloud:

```
curl -v -X POST -u "<user_id>"
https://corpawswsgwx.qualcomm.com/qwes/idtoken | tee id_token.
txt
```

Where `<user_id>` is the Qualcomm WES service account user identifier.

Open `id_token.txt` and copy the id_token's value.

If this command fails, then a Qualcomm WES cloud service account may not be set. To set up the account, contact the Qualcomm customer engineering team.

4. Call the REST API to get the X509 certificate with the device public key on the stage instance of Qualcomm WES Cloud:

```
curl -XPOST -H 'Authorization: Bearer <id_token>'
"https://api.qwes-stg.qualcomm.com/api/b2b/1.0/devices/secure_
provisioning"
-d @da.json -H "Content-type:application/json" -v -o da_cert.
json
```

When this command returns, da_cert.json should contain a certificate with the device public key and one or more parent certificates that form a certificate chain.

If this command fails on a non-secure boot-enabled device, then a Qualcomm WES cloud policy may be required to bypass signature verification for the device on which this test is being conducted. Contact for support.

5. For testing, the certificate with the device public key is sufficient. Open da_cert. json and extract the value of the certificate JSON label. The following command can also be used to extract the certificate:

```
sed -E "s/.*certificate\":\"(.*)\"}].*/\1/;s/\\n/\n/g"
da_cert.json | tee da_cert.pem
```

5 Trusted report

The trusted report service generates a trusted report.

It consists of one or more of the following report types:

- TLOC
- TTIME
- Cellular summary report (CSR)

5.1 Trusted report API reference

The trusted report IDL file, `ITrustedReport.idl`, exposes the following interfaces:

– `IRefreshCallback`: The callback definition that is invoked when the refresh operation completes.

– `ITrustedReport`: Triggers the creation of a trusted report.

A valid feature license/certificate is required to use the trusted report service. For information on how to open an `ITrustedReport` interface, see Qualcomm WES trusted application services API reference.

Sequence diagrams

The following sequence digrams enable you to view the trusted report API sequence.

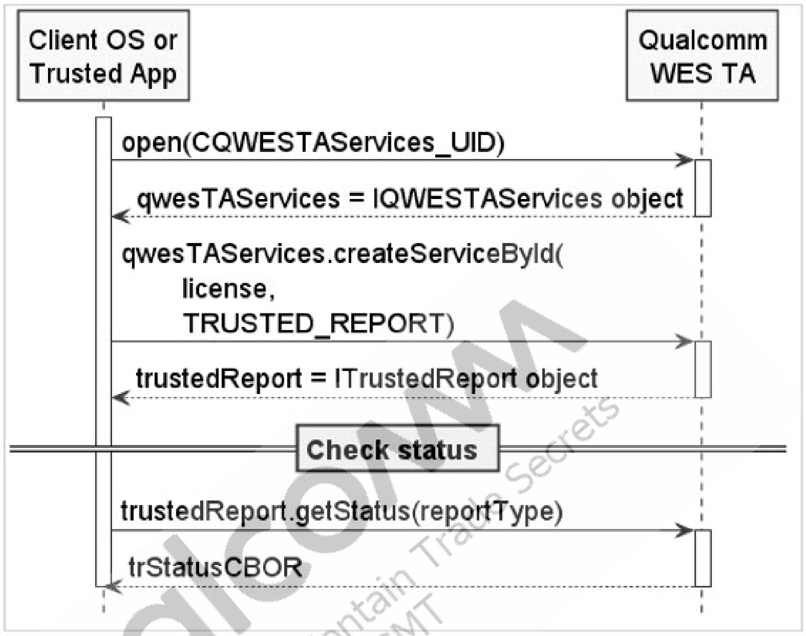

Client OS or Trusted App
Qualcomm WES TA
open(CQWESTAServices_UID)
qwesTAServices = IQWESTAServices object
qwesTAServices.createServiceById(license, TRUSTED_REPORT)
trustedReport = ITrustedReport object
Check status
trustedReport.getStatus(reportType)
trStatusCBOR

Figure : Trusted report API sequence diagram [1 of 2]

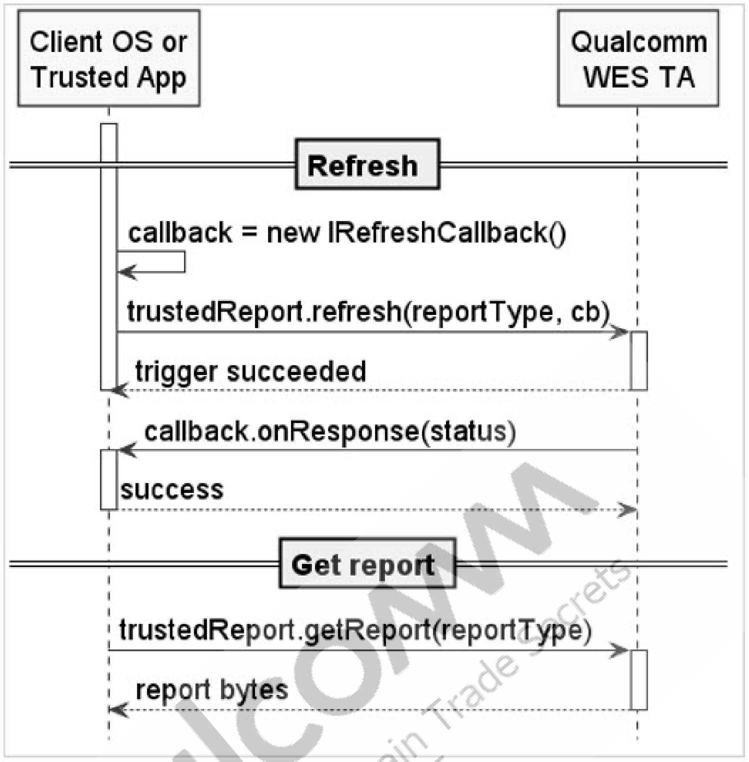

Client OS or Trusted App
Qualcomm WES TA
Refresh
callback = new IRefreshCallback()
trustedReport.refresh(reportType, cb)
trigger succeeded
callback.onResponse(status)
success
Get report
trustedReport.getReport(reportType)
report bytes

Figure : Trusted report API sequence diagram [2 of 2]

ITrustedReport

An object of the type `ITrustedReport` is returned when the `createServicebyId` method is called with the ID of the trusted report service.

Report type constants

These constants correspond to the supported report types to be used with the methods of ITrustedTime interface.

For report contents, see Report specifications.

– `const uint64 TLOC_REPORT = 0x0000000000000001`

  TLOC

– `const uint64 TTIME_REPORT = 0x0000000000000004`

  TTIME

– `const uint64 CSR_REPORT = 0x0000000000000008`

  CSR

- `const uint64 CDR_REPORT = 0x0000000000000010`

  CDR
- `const uint64 SMSR_REPORT = 0x0000000000000020`

  Short messaging service report (SMSR)
- `const uint64 OEM_REPORT = 0x0000000000000040`

  OEM information report
- `const uint64 PLATFORMBOOT_REPORT = 0x0000000000000100`

  Platform boot measurement report
- `const uint64 HLOSBOOT_REPORT = 0x0000000000000200`

  Linux boot report
- `const uint64 SECURITY_REPORT = 0x0000000000000400`

  Device security status report

Methods

getStatus

```
method getStatus(in uint64 reportType, out buffer
statusReport)
```

This method returns a CBOR structure containing the timestamp of the cached report corresponding to one or more of the following report types:

- `TLOC_REPORT`
- `TTIME_REPORT`

The timestamp is the seconds elapsed since the UNIX epoch (January 1, 1970) when the cache was refreshed.

CDDL for statusReport

```
tr_get_status = {
    "Trusted Time": uint / "UNAVAILABLE",
    "Trusted Location": uint / "UNAVAILABLE"
}
```

Example CDN for statusReport when both report types are requested

```
{
    "Trusted Time": 1627943322,
    "Trusted Location": 1627943318
}
```

Parameters

`in`	`reportType`	One or more of `TLOC_REPORT` or `TTIME_REPORT`
`out`	`statusReport`	Status report as CBOR bytes

Returns

– `Object_OK` on success

– `Object_ERROR` on general failures

– `ITrustedReport` error codes otherwise

refresh

```
method refresh(
    in uint64 reportType,
    in uint64 timeout,
    in IRefreshCallback callback)
```

Triggers an update to the reports corresponding to one of the following report types:

– `TLOC_REPORT`

– `TTIME_REPORT`

This method returns immediately. When the refresh operation for the specified `reportType` completes, the specified callback `onResponse` method of the object is called. See IRefreshCallback methods for more information.

Use separate `ITrustedReport` objects for refreshing each report type. This object is required to associate the callback with a specific `reportType` in onResponse.

Parameters

`in`	`reportType`	`Either TLOC_REPORT` or `TTIME_REPORT.`
`in`	`timeout`	Not used – set to 0.
`in`	`callback`	A caller implements a callback object of the type `ITrustedReport` that is invoked when refresh completes, successfully or unsuccessfully.

Returns

- `Object_OK` on success
- `Object_ERROR` on general failures
- `ITrustedReport` error codes otherwise

getReport

```
method getReport(in uint64 reportType, out buffer report)
```

This method returns a CBOR-formatted report corresponding to one of the following report type constants:

- `TLOC_REPORT`
- `TTIME_REPORT`
- `CSR_REPORT`
- `CDR_REPORT`
- `SMSR_REPORT`
- `OEM_REPORT`
- `PLATFORMBOOT_REPORT`
- `HLOSBOOT_REPORT`
- `SECURITY_REPORT`
- Call `refresh` to update the cache.

For report contents, see Report specifications.

Parameters

`in`	`reportType`	One of the `ITrustedReport_*` constants.
`out`	`report`	Trusted report as CBOR bytes.

Returns

- `Object_OK` on success
- `Object_ERROR` on general failures
- `ITrustedReport` error codes otherwise

Error codes

– `ITrustedReport_NO_MEMORY`

Failure during memory allocation.

– `ITrustedReport_INVALID_BUFFER`

Null or zero-length buffer is passed.

– `ITrustedReport_INVALID_CERTIFICATE`

Passed feature license/certificate is invalid.

– `ITrustedReport_INVALID_REPORT_TYPE`

Passed report type is unsupported.

– `ITrustedReport_INVALID_CALLBACK`

Passed callback object is invalid.

– `ITrustedReport_INVALID_TIMEOUT`

Reserved.

IRefreshCallback methods

A caller-implemented mink interface that's provided to the `refresh` method of `ITrustedReport` to serve as a callback when the refresh operation completes.

onResponse

```
method onResponse(in uint32 status);
```

The callback method is invoked by the refresh method of ITrustedReport when the refresh operation for the requested report type completes.

The `status` argument can be set to one of the following constants:

– `const uint32 REFRESH_SUCCESS = 0`

The refresh operation was completed successfully.

– `const uint32 REFRESH_FAILURE = 1`

The refresh operation failed.

– `const uint32 REFRESH_TIMEOUT = 2`

Reserved.

Parameters

in	`status`	Indicates refresh status

Returns

– `Object_OK` on success

– `Object_ERROR` on general failures

– `ITrustedReport` error codes otherwise

Report specifications

The following are the CDDL specification for the available reports. The reports are returned by the `ITrustedTime` interface's `getReport` method.

TLOC

CDDL

```
tr_get_report_tloc = {
     "latitude": float64,
     "longitude": float64,
     "accuracy": float64,
     "altitude": float64,
     "technology": technology_type,
     "bearing": float64,
     "speed": float64,
     "conformity_index": float64,
     "time": uint
}

technology_type =
    technology_unknown /
    GNSS /
    Cell_ID /
    Wi-Fi

technology_unknown = 0
GNSS = 1
Cell_ID = 2
Wi-Fi = 3
```

Example CDN

```
{
    "latitude": 40.7946,
    "longitude": -119.404,
    "accuracy": 518078,
    "altitude": 6,
    "technology": 2,
    "speed": 0,
    "conformity_index": 0.5,
    "time": 1627943318
}
```

TTIME

CDDL

```
tr_get_report_ttime = {
    "best_available_time": {
        "trust_level": &(
            "UNTRUSTED",
            "LOW_TRUST",
            "MEDIUM_TRUST",
            "HIGH_TRUST"),
        "time": uint / "UNAVAILABLE"
    },
    "time_source": {
        "last_refresh_time": uint / "UNAVAILABLE",
        "RTC": {
            "time": uint / "UNAVAILABLE",
            "offset": uint / "UNAVAILABLE"
        },
        "GPS": {
            "time": uint / "UNAVAILABLE",
            "technology": technology_type
        },
        "NETWORK": {
            "time": uint / "UNAVAILABLE"
        },
        "HLOS": {
            "time": uint / "UNAVAILABLE"
        }
    }
}

technology_type =
    technology_unknown /
    GNSS /
    Cell_ID /
    Wi-Fi

technology_unknown = 0
GNSS = 1
Cell_ID = 2
Wi-Fi = 3
```

Example CDN

```
{
    "best_available_time": {
        "trust_level": "HIGH_TRUST",
        "best_time": 1629252850
    },
    "time_source": {
        "last_refresh_time": 1629252840,
        "RTC": {
            "time": 2194,
            "offset": 1629250646
        },
        "GPS": {
            "time": 1629252821,
            "technology": 2
        },
        "NETWORK": {
            "time": 1629252840
        },
        "HLOS": {
            "time": 1629252821
        }
    }
}
```

CSR

CDDL

```
cellular_summary_report = {
    "Timestamp": uint,
    "Timestamp Base": uint,
    "Reports": [
        *{
            "Report Type": &(
                second,
                minute,
                hour,
                day,
                week,
                month),
            "Mailicious": float,
            "Unknown": float,
            "IMSI Leak": float,
            "Imprisoner": float,
            "DOS": float,
            "Downgrade": float,
            "Location Tracker": float,
            "Attractive": float,
            "Threat Prevented": float
        }
    ]
}

; CSR report types
second = 0
minute = 1
hour = 2
day = 3
week = 4
month = 5
```

Example CDN

```
{
    "Timestamp": 877172000,
    "Timestamp Base": 0,
    "Reports": [
        {
            "Report Type": 2,
            "Malicious": 65535,
            "Unknown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.499992,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.499992
        },
        {
            "Report Type": 4,
            "Malicious": 58636,
            "Unknown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.352941
        },
        {
            "Report Type": 8,
            "Malicious": 58636,
            "Unknown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.352941
        },
        {
            "Report Type": 16,
            "Malicious": 58636,
            "Unknown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
                "Threat Prevented": 0.352941
            },
            {
                "Report Type": 32,
                "Malicious": 58636,
                "Unknown": 0,
                "IMSI Leak": 1,
                "Imprisoner": 0,
                "DOS": 0.352941,
                "Downgrade": 0,
                "Location Tracker": 0,
                "Attractive": 0,
                "Threat Prevented": 0.352941
            }
        ]
}
```

The timestamp represents the seconds elapsed since the UNIX epoch (January 1, 1970) when the cache was refreshed.

CDR

CDDL

```
cellular_detailed_report = {
    "Timestamp": uint,
    "Timestamp Base": uint,
    "Reports": [
        *{
            "Report Type": &(
                second,
                minute,
                hour,
                day,
                week,
                month),
            "Cell ID": uint,
            "ARFCN": uint,
            "Physical Cell ID": uint,
            "MCC": uint,
            "MNC": uint,
            "Radio": uint,
            "Score": int,
            "Category": uint,
            "Timestamp": uint,
            "asid": uint,
        }
    ]
}

; CDR report types
second = 0
minute = 1
hour = 2
day = 3
week = 4
month = 5
```

Example CDN

```
{
    "Timestamp": 214000,
    "Timestamp Base": 0,
    "Reports": [
        {
            "Report Type": 0,
            "Cell ID": 0,
            "ARFCN": 0,
            "Physical Cell ID": 0,
            "MCC": 0,
            "MNC": 0,
            "Radio": 0,
            "Score": 0,
            "Category": 0,
            "Timestamp": 214000,
            "asid": 0
        },
        {
            "Report Type": 0,
            "Cell ID": 0,
            "ARFCN": 0,
            "Physical Cell ID": 0,
            "MCC": 0,
            "MNC": 0,
            "Radio": 0,
            "Score": 0,
            "Category": 0,
            "Timestamp": 214000,
            "asid": 0
        },
    ]
}
```

SMSR

CDDL

```
cellular_sms_report = {
    "Timestamp": uint,
    "Timestamp Base": uint,
    "Reports": [
        *{
            "Physical Cell ID": uint,
            "Radio": uint,
            "Score": int,
            "Category": uint,
            "Timestamp": uint,
            "asid": uint,
            "Transaction ID": uint,
            "Is Broadcast": uint
        }
    ]
}
```

Example CDN

```
{
    "Timestamp": 1361925076400,
    "Timestamp Base": 1,
    "Reports": [
        {
            "Physical Cell ID": 169,
            "Radio": 4,
            "Score": 0,
            "Category": 64,
            "Timestamp": 1362023809714,
            "asid": 0,
            "Transaction ID": 5,
            "Is Broadcast": 0
        },
        {
            "Physical Cell ID": 169,
            "Radio": 4,
            "Score": 0,
            "Category": 64,
            "Timestamp": 1362023809714,
            "asid": 0,
            "Transaction ID": 6,
            "Is Broadcast": 0
        },
        {
            "Physical Cell ID": 169,
            "Radio": 4,
            "Score": 0,
            "Category": 64,
            "Timestamp": 1362023810972,
            "asid": 0,
            "Transaction ID": 2,
            "Is Broadcast": 0
        },
        {
            "Physical Cell ID": 169,
            "Radio": 4,
            "Score": 0,
            "Category": 64,
            "Timestamp": 1362023810973,
            "asid": 0,
            "Transaction ID": 3,
            "Is Broadcast": 0
        },
        {
            "Physical Cell ID": 169,
            "Radio": 4,
            "Score": 0,
            "Category": 64,
            "Timestamp": 1362023809713,
            "asid": 0,
            "Transaction ID": 4,
            "Is Broadcast": 0
        }
    ]
}
```

OEM information report

Note: TME hardware isn't present.

CDDL

```
device_trust_get_reports_qtee = {
    "OEM Info" => {
        oemid_qtee => uint,
        oem_pk_hash_qtee => bstr,
        oem_prod_id_qtee => bstr
    }
}

oemid_qtee = -77121
oem_pk_hash_qtee = -77122
oem_prod_id_qtee = -77126
```

Example CDN

```
{
    "OEM Info": {
        -77121: 1,
        -77122: h'abcdef...',
        -77126: h'0100'
    }
}
```

Device security status report

Note: TME hardware isn't present.

CDDL

```
device_trust_get_reports_qtee = {
    "Security State" => {
        secboot_enabled_qtee => bool,
        debug_status_qtee => bool,
        hw_version_qtee => uint,
        antirollback_qtee => bool,
        debug_disable => bool,
        tz_debug => bool,
        mss_debug => bool,
        cp_debug => bool,
        nonsecure_debug => bool,
        hyp_debug_disable => bool,
        sw_version => tstr,
        fw_version => tstr,
        jtag_id => uint,
        sec_hwkey_programmed => bool,
        fuse_config => bool,
        rpmb_provisioned => bool,
        debug_image => bool,
    }
}

secboot_enabled_qtee = -77151
debug_status_qtee = -77153
hw_version_qtee = -77123
antirollback_qtee = -77154
debug_disable = -77153
tz_debug = -77159
mss_debug = -77160
cp_debug = -77161
nonsecure_debug = -77162
hyp_debug_disable = -77163
sw_version = -77124
fw_version = -77125
jtag_id = -77127
sec_hwkey_programmed = -77152
fuse_config = -77155
rpmb_provisioned = -77156
debug_image = -77157
```

6 Wirless edge services APIs

This section provides the detail of APIs available for wireless edge service application development in the Linux user space and in TEE. The details include setting up the application development environment and writing a client application.

6.1 Qualcomm WES APIs

The Qualcomm WES APIs are remote procedure calls (RPC) interacting with a trusted application (TA) running under Qualcomm TEE.

These APIs are available as interfaces defined in interface definition language (IDL) files.

Qualcomm WES APIs are grouped by the following Qualcomm WES services:

Table : Qualcomm WES services

Service name	Mink interface	Mink IDL file
Factory	`IQWESTAServices`	`IQWESTAServices.idl`
Feature licensing	`IPFM`	`IPFM.idl`
Device attestation	`IDeviceAttestation`, `IAttestationBuilder`, `IAttestationReport`, and `IDAError`	`IDeviceAttestation.idl`
Secure provisioning	`IProvisioning`, `IQWESKeyStore`, and `IProvError`	`IProvisioning.idl` and `IQWESKeyStore.idl`
Trusted report (VASC)	`ITrustedReport`	`ITrustedReport.idl`

The `CQWESTAServices.idl`, `CPFM.idl`, and `CDeviceAttestation.idl` files contain the unique ID (UID) of the factory interfaces, `PFM` and `DeviceAttestation` respectively. This number is used to open the mink interface. Secure provisioning and trusted report are accessed through the `IQWESTAServices` interface.

If the client is a Linux-Embedded application, then additionally an opener interface exposed by `Iopener.idl` is also used to open the feature licensing (`IPFM`) or device attestation (`IdeviceAttestation`) service.

The IDL files are compiled into C headers, C++ headers, or Java class files using the `minkidl` IDL compiler. The files generated from IDLfiles are also called proxy files. These proxy files work with a transport library that handles the RPC into the TA. The client application, which calls the APIs, can be an LE native application or another TA.

For more information, see Qualcomm Trusted Execution Environment 5.0 Security Guidelines for Trusted Applications User Guide.

Concise binary object representation

Several Qualcomm WES APIs take arguments or return data in CBOR formatted bytes.

CBOR can be best thought of as binary JSON and is more flexible. For more information on CBOR RFC, see: https://tools.ietf.org/html/rfc8610.

For a CBOR encoder and decoder software libraries, see https://cbor.io/impls.html.

For an online CBOR encoder and decoder tool, see http://cbor.me/.

CDDL

A language to define the structure of the CBOR data.

For Qualcomm WES API arguments that accept or return CBOR data, the definition in CDDL is provided in the API reference. For more information about the CDDL RFC, see Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures.

CBOR diagnostic notation

It's a JSON-type notation to represent CBOR and make it human readable.

For Qualcomm WES API arguments that accept or return CBOR data, an example in CBOR diagnostic notation (CDN) is given in the API reference. For online decoder and encoder that use CDN, see http://cbor.me/.

Right client application type

This information helps you to select the right client application type.

- For OEMs: If you are an OEM, use C or C++ for TA,
- ISVs: Since ISVs can't independently develop trusted applications for Linux, they must work with OEMs for these platforms.

Write client application

To write a client application, you need the Qualcomm WES Product Kit, IDL files, and Qualcomm TEE SDK.

Get Qualcomm WES Product Kit v2

Download the latest version of the Qualcomm_WES_Platform.ZIP.1.0 Files WES2.

Note: The link may point to an older version of the ZIP file. Qualcomm recommends you to download the latest version.

If the link doesn't work, do the following:

1. Log in to https://createpoint.qti.qualcomm.com.
2. Search for Qualcomm WES.
3. Go to the **Tools** tab.
4. Click on **Qualcomm_WES_PLATFORM.ZIP.1.0 FILES WES2**.
5. Click on **Download**.

The downloaded zip contains the`Qualcomm_WES_SDK_v2` directory required for writing client applications.

The rest of the guide uses `<QWES_PK_v2_ROOT>` to refer to the`Qualcomm_WES_SDK_v2/v2` directory extracted from Qualcomm_WES_PLATFORM.ZIP.1.0 FILES WES2.

Get IDL files

The IDL files are available at location: `<QWES_PK_v2_ROOT>/IDL/mink>`.

Get the following IDL files based on which of the following services the client application uses:

- Feature licensing
  - `IPFM.idl`
  - `CPFM.idl`
- Device attestation
  - `IDeviceAttestation.idl`
  - `CDeviceAttestation.idl`
- If the client is a Linux-Embedded application (not a trusted application)
  - `IOpener.idl`

For Linux

Get minkidl compiler

The `minkidl` compiler, which compiles IDL files into C header file (.h) and C++ header files (.hpp) is available at `<QWES_PK_v2_ROOT>/host_tools/<host_os>`.

Run minkidl commands to generate proxy files

1. Generate `IPFM.h`, `CPFM.h`, `IPFM.hpp`, and `CPFM.hpp`.
2. Generate `IDeviceAttestation.h`, `IDeviceAttestation.hpp`, and `CDeviceAttestation.h and CDeviceAttestation.hpp`.

   `IDeviceAttestation.idl` has three interfaces, therefore running the IDL through `minkidl` generates three interfaces:

   (`IDeviceAttestation`,

   `IAttestationReport`, and

   `IAttestationBuilder`) within `IDeviceAttestation.h`/`IDeviceAttestation.hpp`.

   ```
   ./minkidl -o IdeviceAttestation.h IdeviceAttestation.idl
   ./minkidl -o CdeviceAttestation.h CdeviceAttestation.idl
   ./minkidl -cpp -o IdeviceAttestation.hpp IdeviceAttestation.idl
   ./minkidl -cpp -o CdeviceAttestation.hpp CdeviceAttestation.idl
   ```

3. Generate `IClientEnv.h` and `IClientEnv.hpp`.

   Skip this step if the client application is a TA.

   ```
   ./minkidl -o IclientEnv.h IclientEnv.idl
   ./minkidl -cpp -o IclientEnv.hpp IclientEnv.idl
   IDL to Java arguments data-type mapping
   ```

To display the available help options, run the following command:

```
./minkidl --help
```

This command provides detailed information about the usage, options, and functionalities of `minkidl`. It typically includes descriptions of various command-line options and examples of how to use `minkidl`.

Open and close IPFM or IDeviceAttestation

Get IClientEnv object

```
Object clientEnvMaster = Object_NULL;
int32_t  nReturn = Object_OK;
nReturn = TZCom_getClientEnvObject(&clientEnvMaster);
if (nReturn != Object_OK)
{
  // handle error
}
```

If `TZCom_getClientEnvObject` is successful, then `clientEnvMaster` is used as an environment object to be used to get handle to `IPFM` or `IdeviceAttestation` objects as described in the following sections.

Note: The function `TZCom_getClientEnvObject(...)` is defined as part of the `InvokeLib.lib` library.

Get IPFM object

```
Object pfmObj = Object_NULL;
nReturn = IClientEnv_open(clientEnvMaster, CPFM_UID, pfmObj);
if (nReturn != Object_OK)
{
  IClientEnv_release(clientEnvMaster);
  clientEnvMaster = Object_NULL;
}
```

Get IDeviceAttestation object

```
ProxyBase da_controller{
        GetAppController(QWES_TA, NULL, 0, false)};
    if (da_controller.isNull()) {
        CLI_PRINT("Failed to load %s; continuing anyway\n", QWES_TA);
    }

    ProxyBase client_env{GetClientEnv(NULL, 0, false)};
    if (client_env.isNull()) {
        CLI_PRINT("Unable to get client env for pfm\n");
        return;
    }

    IDeviceAttestation da{};
    int32_t err = Object_ERROR;
    IQWESTAServices qts{
        GetObjectById(client_env.get(), CQWESTAServices_UID)};
    if (!qts.isNull()) {
        err = qts.createServiceById(
            license.ptr, license.len, IQWESTAServices::DEVICE_
ATTESTATION, da);
        if (Object_isERROR(err)) {
            CLI_PRINT("Unable to get IDA thru IQTAS (%d); \n", err);
            da.extract();   // ensure null
            // error handling and return
        }
    }
```

Release IPFM

```
if(!Object_isNull(pfmObj)) {
        IPFM_release(pfmObj);
}
```

Release IClientEnv object

```
if(!Object_isNull(clientEnvMaster) ) {
        IClientEnv_release(clientEnvMaster);
}
```

Check license buffer

This method is the simplest method to check if a feature ID is allowed. A feature ID is allowed if the license passed as argument is valid for the feature ID. Use the following code:

```
uint32_t FeatureID_val = 55;
void *license_ptr;
size_t license_len;
int32_t  nReturn = Object_OK;
Object pfmObj = Object_NULL;
uint8_t hashVal[32]; //size of SHA256 output hence 32
size_t hashLen;
uint8_t out_buff[1024];
size_t out_buff_len;
/*
```

To check the license buffer, follow these steps:

1. Obtain `pfmObj` as described in Open and close IPFM or IDeviceAttestation.
2. Copy license certificate into `license_ptr`.
3. Update `license_len` with the length of the license certificate copied into `license_ptr`.
4. Compute the SHA-256 hash of the signing certificate hash of the application.
5. Get the licensee hash from the application.
6. Update `hashVal` with the SHA-256 hash output and `hashLen` with the length of the hash:

```
*/
nReturn = IPFM_CheckLicenseBuffer(pfmObj, license_ptr, license_
len, FeatureID_val, hashVal, hashLen, out_buff, sizeof(out_
buff), &out_buff_len);
if (nReturn != Object_OK)
{
  // handle error
}
```

Generate attestation report with client application data

Obtain `daObj` as described in Open and close IPFM or IDeviceAttestation, and use the following code:

```
IDeviceAttestation da{ daObj };
if (da.isNull()) {
  // handle error
}
IAttestationBuilder ab{};
int32_t err = da.start(daLicense, daLicense_len, ab);
if (Object_isERROR(err)) {
  // handle error
}
const uint8_t label[10] = {"label1"};
const uint8_t data[10] = {"data1"};
err = ab.addBytes(IAttestationBuilder::SECURITY_LEVEL_UNRESTRICTED,
                  label,
                  sizeof(label),
                  data,
                  sizeof(data));
if (Object_isERROR(err)) {
  // handle error
}
IAttestationReport ar{};
```

```
uint64_t report_status = 0;
uint8_t nonce[32] = {};  // 32-byte array (see API documentation)
err = ab.build(IAttestationBuilder::ATTESTATION_CONTEXT_GENERIC,
               0,
               IAttestationBuilder::FORMAT_EAT,
               IAttestationBuilder::KEY_QDAK,
               nonce,
               sizeof(nonce),
              &report_status,
               ar);
if (Object_isERROR(err)) {
  // handle error
}
uint64_t report_bytes_len = 0;
err = ar.getSize(&report_bytes_len);
if (Object_isERROR(err)) {
  // handle error
}



uint8_t* report_bytes; //Allocate memory as per report_bytes_len
uint64_t report_bytes_lenout = 0;
err = ar.getBytes(0,
                  report_bytes,
                  report_bytes_len,
                  (size_t *)&report_bytes_lenout);
if (Object_isERROR(err)) {
  // handle error
}
```

Handle Object_ERROR_BUSY

Calling into Qualcomm WES APIs can result in returning `Object_ERROR_BUSY` with the `-99` value. This error indicates that Qualcomm TEE is busy and unable to handle the API request currently. The client application other than TA must handle this error and retry. The retry interval can be as low as 100 ms with 10 retries.

For trusted applications

You will need the Qualcomm TEE SDK to write a trusted application.

For more information, see Qualcomm Trusted Execution Environment 5.0 Security Guidelines for Trusted Applications User Guide.

The Qualcomm TEE SDK contains the following C header files specific to feature licensing and device attestation:

- `IPFM.h` for feature licensing APIs
- `CPFM.h` for the UID needed to open an object of type IPFM
- `IDeviceAttestation.h` for device attestation APIs
- `CDeviceAttestation.h` for UID is needed to open an object of the type IDeviceAttestation.

Open and close IPFM or IDeviceAttestation

Include required header file

```
#include <qsee_env.h>
```

Open an object of type IPFM

```
#include "IPFM.h"
#include "CPFM.h"

Object pfmObj = Object_NULL;
int32_t ret = qsee_open(CPFM_UID, &pfmObj);
```

Close the object

```
Object_RELEASE_IF(&pfmObj);
```

Open an object of type IDeviceAttestation

```
#include "IDeviceAttestation.h"
#include "CDeviceAttestation.h"

Object daObj = Object_NULL;
int32_t ret = qsee_open(CDeviceAttestation_UID, &daObj);
```

Close the object

```
Object_RELEASE_IF(&daObj);
```

Generate attestation report with client application data

```
int32_t GenerateAttestationReport(
    UsefulBuf token_cert,
    UsefulBuf nonce,
    UsefulBuf client_data,
    uint64_t addon_opts,
    UsefulBuf* report_bytes)
{
    const uint32_t attestation_context =
        IAttestationBuilder_ATTESTATION_CONTEXT_GENERIC;
     const uint32_t format_type = IAttestationBuilder_FORMAT_EAT;
    const uint32_t key_type = IAttestationBuilder_KEY_QDAK;
    const char* client_data_key = "my_data";

    int32_t err = Object_ERROR;
    uint64_t report_status = 0;
    Object daObj = Object_NULL;  // IDeviceAttestation
    Object abObj = Object_NULL;  // IAttestationBuilder
    Object arObj = Object_NULL;  // IAttestationReport

    UsefulBuf_Set(report_bytes, 0);

    err = qsee_open(CDeviceAttestation_UID, &daObj);
    if (Object_isERROR(err)) {
        goto done;
    }

    err = IDeviceAttestation_start(daObj, token_cert.ptr, token_cert.
len, &abObj);
    if (Object_isERROR(err)) {
        goto done;
    }

    // This method can be called many times with different keys
    err = IAttestationBuilder_addBytes(
        abObj,
        IAttestationBuilder_SECURITY_LEVEL_UNRESTRICTED,
        client_data_key,
        strlen(client_data_key),
        client_data.ptr,
        client_data.len);
    if (Object_isERROR(err)) {
        goto done;
    }

    err = IAttestationBuilder_build(
        abObj,
        attestation_context,
        addon_opts,
        format_type,
        key_type,
        nonce.ptr,
        nonce.len,
        &report_status,
        &arObj);
    if (Object_isERROR(err)) {
        goto done;
    }

    err = IAttestationReport_getSize(arObj, &report_bytes->len);
    if (Object_isERROR(err)) {
        goto done;
    }

    report_bytes->ptr = (uint8_t*)qsee_zalloc(report_bytes->len);
    if (!report_bytes->ptr) {
        goto done;
    }

    err = IAttestationReport_getBytes(
        arObj, 0,
        report_bytes->ptr, report_bytes->len, &report_bytes->len);
    if (Object_isERROR(err)) {
        qsee_free(report_bytes->ptr);
        report_bytes->len = 0;
        goto done;
    }

    err = Object_OK;

done:
    Object_RELEASE_IF(arObj);
    Object_RELEASE_IF(abObj);
    Object_RELEASE_IF(daObj);

    return err;
}
```

Qualcomm WES trusted application services API reference

The Qualcomm WES trusted application service API is used as a factory to create objects.

The Qualcomm WES services that use this API are:

- Secure provisioning
- Qualcomm WES key store
- Trusted report

The Qualcomm WES services that don't use this API are:

- Platform feature manager
- Device attestation

This functionality is exposed by the `IQWESTAServices` interface and every service requires a feature license/certificate for the requested service.

The `IQWESTAServices` interface has the following constants and methods.

IQWESTAServices constants

Service Ids

- `const uint32 PLATFORM_FEATURE_MANAGER = 1;`

  Reserved

- `const uint32 DEVICE_ATTESTATION = 2;`

  Reserved

- `const uint32 TRUSTED_REPORT = 3;`

  Returns an object of type `ITrustedReport`

- `const uint32 SECURE_PROVISIONING = 4;`

  Returns an object of type `IProvisioning`

- `const uint32 QWES_KEYSTORE = 5;`

  Returns an object of type `IQWESKeyStore`

IQWESTAServices methods

createServiceById

```
method createServiceById(
    in buffer license,
    in uint32 serviceId,
    out interface service);
```

Returns an object corresponding to the service id.

Parameters

in	license	Feature license/certificate to use the requested service.
in	serviceId	Service id See IQWESTAServices constants Service IDs.
out	service	Receives the object of the requested service type of the IDs. See IQWESTAServices constants Service IDs.

Returns

- `Object_OK` on success.
- `Object_ERROR_INVALID` if the service identified by `serviceId` isn't found or if any of the arguments are invalid.
- `Object_ERROR` if the service fails to open.

Privacy preserving ID API reference

The `PrivacyPreservingID` ClassID isn't included in the default privilege set. Add it to the privileges section in the metadata, which allows the trusted application to use the `CPrivacyPreservingID` class.

Constants

Service Ids

```
const uint32 CPrivacyPreservingID_UID = 455;
```

Methods - getDeviceID

Parameters

in	salt	Salt provided by the client.
out	id	32-byte client-specific device ID.

Returns

- `Object_OK` on success.
- `ERROR_FEATURE_NOT_LICENSED` - This feature isn't enabled in the PFM license.
- `ERROR_LICENSE_UNAVAILABLE` - The PFM license can't be accessed.

Methods - getClientDeviceID

Parameters

in	credential	Credentials for the client using this API.
in	salt	Salt provided by the client.
out	id	32-byte client-specific device ID.

Returns

`Object_OK` on success

7 Qualcomm WES examples

This section provides the sample format for attestation report generated by the wireless edge service.

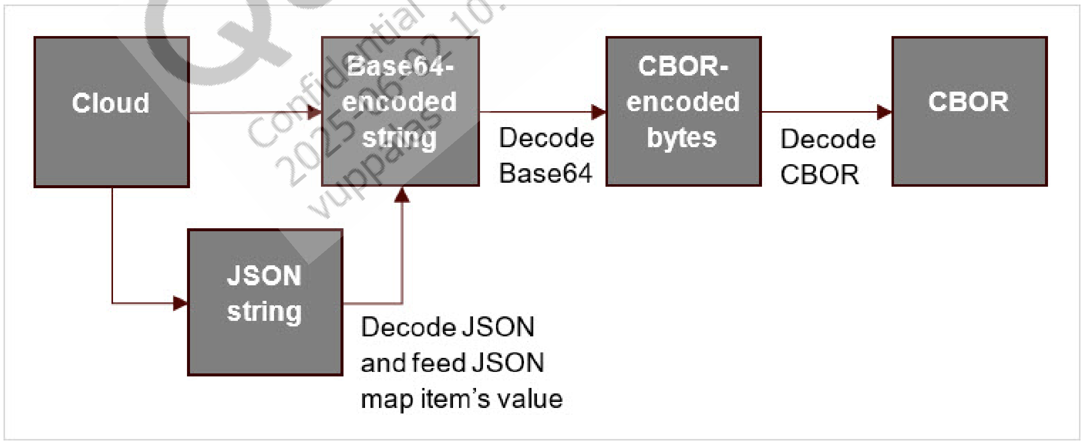
7.1 Attestation report data format

This section provides some examples of the attestation report data format.

Contents of the attestation report can be retrieved using Qualcomm WES cloud B2B APIs. Here is a list of keys and data formats returned as part of the following Qualcomm WES cloud B2B endpoint response: `<server>/api/b2b/2.0/attestations/generic`.

The cloud endpoint either returns a JSON map or a Base64-encoded string. For a JSON map, the item value of the map is a Base64-encoded string unless stated otherwise. The Base64-encoded string in turn is generated using CBOR-encoded bytes.

In this section, the CBOR-encoded bytes are represented in CDDL. See RFC 8610 for details.



An example is also given in CDN that's useful to visualize CBOR data and is based on JSON. Within CDN, everything between a '/' are comments. See RFC 7049 for details.

device_security_state

CDDL

```
device_security_state = {
    secboot_enabled => bool,
    sec_hwkey_programmed => bool,
    debug_disable => bool,
    anti_rollback => bool,
    fuse_config => bool,
    rpmb_provisioned => bool,
    debug_image => bool,
    misc_debug => bool,
    tz_debug => bool,
    mss_debug => bool,
    cp_debug => bool,
    nonsecure_debug => bool
}

secboot_enabled = -77151
sec_hwkey_programmed = -77152
debug_disable = -77153
anti_rollback = -77154
fuse_config = -77155
rpmb_provisioned = -77156
debug_image = -77157
misc_debug = -77158
tz_debug = -77159
mss_debug = -77160
cp_debug = -77161
nonsecure_debug = -77162
```

Example CDN

```
{
    -77151: true,      / secboot_enabled /
    -77152: true,      / sec_hwkey_programmed /
    -77153: true,      / debug_disable /
    -77154: true,      / anti_rollback /
    -77155: false,     / fuse_config /
    -77156: true,      / rpmb_provisioned /
    -77157: false,     / debug_image /
    -77159: true,      / tz_debug /
    -77160: true,      / mss_debug /
    -77161: true,      / cp_debug /
    -77162: true        / nonsecure_debug /
}
```

application_data - unrestricted or secure_restricted

If an application or service in Qualcomm TEE generates the device attestation report, then both `secure_restricted` and `unrestricted` can be present. Otherwise only `unrestricted` is present.

The values of both `unrestricted` and `secure_restricted` are CBOR encoded bytes that themselves are Base64 encoded.

Upon CBOR decode, the data format is:

CDDL

```
application_data = {
    + (tstr: bstr)
}
```

Example CDN

```
{
    "hello": h'abcd',
    "world": h'0123'
}
```

application_credentials

unrestricted

When the device attestation report is generated, the data format of the credentials is as follows:

CDDL

```
creds_HLOS_native_type = {
    TAG_UID => uint,
    TAG_SYSTEM_TIME => uint,      ; number of milliseconds elapsed
                                  ; since unix epoch
}

TAG_UID = 1
TAG_SYSTEM_TIME = 6
```

Example CDN

```
{
    1: 2345,                        / TAG_UID /
    6: 1234235234                   / TAG_SYSTEM_TIME /
}
```

secure-restricted

If a trusted application generates the device attestation report, the data format of the credentials is as follows:

CDDL

```
TA_credentials = {
     TA_name => tstr,
     TA_did => bstr,
     TA_uuid => bstr,
     TA_rot => {
          authroot_oem => bool,
          authroot_alt => bool,
          authroot_qti => bool
      },
     TA_rot_hash => bstr
}

authroot_oem = -77012
authroot_alt = -77013
authroot_qti = -77014
TA_name = -77400
TA_did = -77002
TA_uuid = -77003
TA_rot = -77004
TA_rot_hash = -77015
```

Example CDN

```
{
    -77400: "ta_name",      / TA_name /
    -77002: h'abcdef',      / TA_did /
    -77003: h'abcd',        / TA_uuid /
    -77004: {               / TA_rot /
        -77012: true,       / authroot_oem /
        -77013: false,      / authroot_alt /
        -77014: true        / authroot_qti /
    },
    -77015: h'abcdabcd'   / TA_rot_hash (32 or 48 bytes) /
}
```

creds_info

The credential information provides more information about the entity that generated the device attestation (caller).

The following table maps the caller to the CDDL type defined later in this section:

CDDL type	Caller
`creds_info_TEE_type`	The caller is a trusted application running in the Qualcomm TEE.
`creds_info_Primary_VM_native_type`	The caller is a native application residing in the primary virtual machine.
`creds_info_Secondary_VM_type`	The caller resides in the secondary virtual machine. Currently such a caller can only be a native binary.

CDDL

```
creds_info =
    creds_info_TEE_type /
    creds_info_Primary_VM_APK_type /
    creds_info_Primary_VM_native_type /
    creds_info_Secondary_VM_type /
    creds_info_SS_modem_type

creds_info_TEE_type = {
    caller_environment => caller_environment_type .eq QTEE
}

caller_environment_type = &(
    QTEE,
    primary_vm,
    secondary_vm,
    subsystem)

QTEE = 1
primary_vm = 2
secondary_vm = 3
subsystem = 4

    caller_environment => caller_environment_type .eq primary_vm,
    hlos => hlos_type .eq android,
    vm_id => vm_id_type .eq vm_id_hlos,
    caller_client_creds => caller_client_creds_type .eq client_creds_
APK
}

hlos_type = &(
    android,
    linux_embedded,
    windows,
    threadx)

android = 1
linux_embedded = 2
windows = 3
threadx = 4

vm_id_type = &(
    vm_id_hlos,
    vm_id_tvm)

vm_id_hlos = h
'33636434376232306631363635666532393532313032656564666334333353061'
vm_id_tvm = h
'35393830383564616335313635623235613963313932376130323831393730'

caller_client_creds_type = &(
    client_creds_APK,
    client_creds_native)

client_creds_APK = 1
client_creds_native = 2

creds_info_Primary_VM_native_type = {
    caller_environment => caller_environment_type .eq primary_vm,
    vm_id => vm_id_type .eq vm_id_hlos,
    caller_client_creds => caller_client_creds_type .eq client_creds_
native
}

creds_info_Secondary_VM_type = {
    caller_environment => caller_environment_type .eq secondary_vm,
    hlos => hlos_type .eq linux_embedded,  ; Only for TUI-VM
    vm_id => vm_id_type .eq vm_id_tvm,
    caller_client_creds => caller_client_creds_type .eq client_creds_
native
}

creds_info_SS_modem_type = {
    caller_environment => caller_environment_type .eq subsystem,
    ss_id => ss_id_type .eq TZ_SC_MPSS,
    caller_client_creds => caller_client_creds_type .eq client_creds_
native
}

ss_id_type = &(
    TZ_SC_MPSS)

TZ_SC_MPSS = h'00000001'   ; Modem subsystem.

hlos = -77017
vm_id = -77460
ss_id = -77480
caller_environment = -77491
caller_client_creds = -77492
```

Example CDN - Caller is a TA

```
{-77491: 1}
}
```

Example CDN - Caller is in the modem subsystem

```
{-77491: 4, -77480: h'00000001', -77492: 2}
```

rtic_data > data

CDDL

```
rtic_data = {
    integrity => bool,
    is_checkable_kernel => bool,
    age => uint,
    kp_incident => uint,
    kv => tstr,
    ? errors => [ + tstr ]
}
```

```
integrity = -77201
is_checkable_kernel = -77202
age = -77203
kp_incident = -77205
kv = -77206
errors = -77204
```

Example CDN

```
{
    -77201: false,      / integrity /
    -77202: false,      / is_checkable_kernel /
    -77203: 0,          / age /
    -77205: 0,          / kp_incident /
    -77206: "",         / kv /
    -77204: [           / errors /
        "UNLOCKED",
        "INTERNAL",
        "INTERNAL"
    ]
}
```

See Qualcomm Runtime Kernel Security User Guide for more information.

device_info > secure_restricted

CDDL

```
device_info = {
    oemid => uint,
    pkhash => bstr,
    hw_version => uint,
    sw_version => tstr,
    fw_version => tstr,
    oem_product_id => uint,
    jtag_id => uint
}

oemid = -77121
pkhash = -77122
hw_version = -77123
sw_version = -77124
fw_version = -77125
oem_product_id = -77126
```

```
jtag_id = -77127
```

Example CDN

```
{
    -77121: 0,                    / oem_id /
    -77122: h'abcdef',            / pk_hash /
    -77123: 1611596288,           / hw_version /
    -77124: "<s_version>",        / sw_version /
    -77125: "<fw_version>",       / fw_version /
    -77126: 0,                    / oem_product_id /
    -77127: 0                     / jtag_id /
}
```

trusted_time > data

CDDL

```
trusted_time = trusted_time_value_type / trusted_time_errors_type

trusted_time_value_type = {
    "best_available_time": {
        "trust_level": &(
            "UNTRUSTED",
            "LOW_TRUST",
            "MEDIUM_TRUST",
            "HIGH_TRUST"),
        "time": uint / "UNAVAILABLE"
    },
    "time_source": {
        "last_refresh_time": uint / "UNAVAILABLE",
        "RTC": {
            "time": uint / "UNAVAILABLE",
            "offset": uint / "UNAVAILABLE"
        },
        "GPS": {
            "time": uint / "UNAVAILABLE",
            "technology": technology_type
        },
        "NETWORK": {
            "time": uint / "UNAVAILABLE"
        },
        "HLOS": {
            "time": uint / "UNAVAILABLE"
        }
    }
}

trusted_time_errors_type = [+ tstr]

technology_type =
    technology_unknown /
    GNSS /
    Cell_ID /
    Wi-Fi

technology_unknown = 0
GNSS = 1
Cell_ID = 2
Wi-Fi = 3
```

Example CDN

```
{
    "best_available_time": {
        "trust_level": "HIGH_TRUST",
        "best_time": 1629252850
    },
    "time_source": {
        "last_refresh_time": 1629252840,
        "RTC": {
            "time": 2194,
            "offset": 1629250646
        },
        "GPS": {
            "time": 1629252821,
            "technology": 2
        },
        "NETWORK": {
            "time": 1629252840
        },
        "HLOS": {
            "time": 1629252821
        }
    }
}
```

See Qualcomm Runtime Kernel Security User Guide for more information.

trusted_location > data

CDDL

```
trusted_location = trusted_location_value_type / trusted_location_
errors_type

trusted_location_value_type = {
      "latitude": float64,
      "longitude": float64,
      "accuracy": float64,
      "altitude": float64,
      "technology": technology_type,
      "bearing": float64,
      "speed": float64,
      "conformity_index": float64,
      "time": uint
}

trusted_location_errors_type = [+ tstr]

technology_type =
     technology_unknown /
     GNSS /
     Cell_ID /
     Wi-Fi

technology_unknown = 0
GNSS = 1
Cell_ID = 2
Wi-Fi = 3
```

Example CDN

```
{
    "latitude": 40.7946,
    "longitude": -119.404,
    "accuracy": 518078,
    "altitude": 6,
    "technology": 2,
    "speed": 0,
    "conformity_index": 0.5,
    "time": 1626946496
}
```

See Qualcomm Runtime Kernel Security User Guide for more information.

cellular_health_data > data

CDDL

```
cellular_summary_report = {
    "Timestamp": uint,
    "Timestamp Base": uint,
    "Reports": [
        *{
            "Report Type": &(
                second,
                minute,
                hour,
                day,
                week,
                month),
            "Mailicious": float,
            "Unknown": float,
            "IMSI Leak": float,
            "Imprisoner": float,
            "DOS": float,
            "Downgrade": float,
            "Location Tracker": float,
            "Attractive": float,
            "Threat Prevented": float
        }
    ]
}

; CSR report types
second = 0
minute = 1
hour = 2
day = 3
week = 4
month = 5
```

Example CDN

```
{
    "Timestamp": 877172000,
    "Timestamp Base": 0,
    "Reports": [
        {
            "Report Type": 2,
            "Malicious": 65535,
            "Unkown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.499992,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.499992
        },
        {
            "Report Type": 4,
            "Malicious": 58636,
            "Unkown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.352941
        },
        {
            "Report Type": 8,
            "Malicious": 58636,
            "Unkown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.352941
        },
        {
            "Report Type": 16,
            "Malicious": 58636,
            "Unkown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.352941
        },
        {
            "Report Type": 32,
            "Malicious": 58636,
            "Unkown": 0,
            "IMSI Leak": 1,
            "Imprisoner": 0,
            "DOS": 0.352941,
            "Downgrade": 0,
            "Location Tracker": 0,
            "Attractive": 0,
            "Threat Prevented": 0.352941
        }
    ]
}
```

See Qualcomm Runtime Kernel Security User Guide for more information.

8 Bring up Qualcomm WES

You can verify the status of all security features. In particular, ensure to verify the status of Qualcomm WES.

To do this, collect the Qualcomm TEE logs. These logs provide the necessary information to confirm the status of Qualcomm WES. For more information, see Qualcomm Linux Security Guide → *DebugQualcommTEEandsecuredevices*.

```
[015c832b7]<2>qwes: tz_app_init[88]: QWES TA Init
[015c83330]<2>qwes: KeymasterLite_init[148]: KeymasterLite
init
[015c8335e]<2>qwes: tz_app_init[95]: QWES TA Start 1.0
[015c85169]<2>qwes: tz_module_open[51]: QWES Service - uid:
78
```

[015d399a0]<2>qwes: MDTable_init[510]: Initializing storage for License Store (fat ToC), version 1

For additional bring up procedures, see Qualcomm Linux Security Guide → *Bringupsecurityfeatures*.

9 Debug Qualcomm WES

This section provides procedures to obtain the debug logs and use the Qualcomm WES command-line interface (QWES_CLI).

9.1 Debug logs

Debug logs contain detailed information of what an application does during execution. You can use these logs to diagnose issues as they provide detailed insights into the application execution flow, variable state, and error occurrences.

Note: `log_level` = 7, enables logs from 0 to 7 level. Reference levels are based on Linux, therefore, the levels are from 0 to 7. `log_level` = 7, enables all the logs.

Ensure to run all the SSH commands in the SELinux Permissive mode. The Enforcing mode will be supported in the future.

To effectively debug the Qualcomm WES-related issues, get the following logs.

1. To get logcat logs, do the following:
    1. Connect to the device using SSH. For instructions, see Qualcomm Linux Build Guide $\rightarrow Howto \rightarrow SigninusingSSH$.
    2. To enable Qualcomm WES logs, use the following command:

```
mount -o rw,remount /
journalctl -f
echo "log_level=7" > /etc/qwesd.rc
reboot
```

    3. To collect the logs, use the following command:

```
journalctl -f
```

2. To get the TA logs:
    1. Connect to the device using SSH. For instructions, see Qualcomm Linux Build Guide

$\rightarrow Howto \rightarrow SigninusingSSH$.

2. To collect the logs, use the following command:

```
mount -o rw,remount /
cat /proc/tzdbg/qsee_log
```

3. To get Qualcomm TEE kernel logs:

   1. Connect to the device using SSH. For instructions, see Qualcomm Linux Build Guide $\rightarrow Howto \rightarrow SigninusingSSH$.

   2. To collect the logs, use the following command:

```
mount -o rw,remount /
cat /proc/tzdbg/log
```

Note: Qualcomm TEE TA and kernel logs are disabled when secure boot is enabled. It's possible to enable these logs by flashing the debug policy.

Secure boot enabled devices

Secure boot disables the trusted applications and Qualcomm TEE kernel logs. To enable the trusted applications and Qualcomm TEE kernel logs, flash debug-policy with `DP_ENABLE_LOGS` set to 1.

In production devices, the tzdbg driver that's responsible to generate the trusted applications and Qualcomm TEE kernel logs may be absent. In this case, a debug policy change doesn't work.

For more information, see Enable secure boot.

9.2 QWES_CLI

QWES_CLI is a command-line application that makes it easy to run various Qualcomm WES service scenarios on a development device.

Since this is a development tool, to use `qwes_cli`, switch to the root user and disable SE policy enforcement.

Feature licensing

InstallLicense

```
qwes_cli -f license-file install
```

Options

`-f license-file`

Path and name of the feature license/certificate on the device.

CheckFeatureIds

```
qwes_cli -i feature-ids check
```

Options

`-i feature-ids`

List of feature ids to check.

Format

`feature-id1,feature-id2,feature-idn`

GetAllInstalledSerialNumbers

```
qwes_cli list
```

GetInstalledLicenseInfo

```
qwes_cli -s license-serial-number info
```

Options

`-s license-serial-number`

The serial number as the hexadecimal string of the installed feature license/certificate.

RemoveLicense

```
qwes_cli -s license-serial-number remove
```

Options

`-s license-serial-number`

The serial number as the hexadecimal string of the installed feature license/certificate.

Device attestation

Usage

```
qwes_cli -T <license> -t token-cert-file -c nonce [-u key-value-
pairs] [-x addon-options] -o out-file da
```

Options

- `-c nonce`

  A 32-byte nonce to pass to the build API. *nonce* is either a hexadecimal-string or path of a text file that contains a hexadecimal string representing the binary nonce.

  If not specified, the following 32-byte default nonce is used:

  `00000000000000000000000000000000000000000000000000000000deadbeef`.

- `-u key-value-pairs`

  Key value pairs specified in the following format:

  `key_0=value_0,key_1=value_1,key_n=value_n`

  Each key and value are hexadecimal strings.

  Example:

  `dead=beef,c001=d00d`

  `key-value-pair` can also point to a file with hexadecimal strings with one key-value pair per line.

  Example:

  `dead=beef, c001=d00d`

  All data is added at the unrestricted security level.

- `-x addon-options`

  A comma-separated string of any combination of rtic, ttime, tloc, or ccell for real-time integrity check, trusted time, trusted location, or cellular security respectively.

Example:

```
ttime,tloc
```

This sets add-on options for trusted time and trusted location.

If not specified, no add-on option is set.

- `-o out-file`

  Path and name of the output file to save the device attestation report.

10 References

10.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Linux Security Guide	80-70018-11
Qualcomm Linux Security Guide - Addendum Available to licensed developers with authorized access.	80-70018-11A
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Wireless Edge Services (WES) Business-to-Business (B2B) API Reference	80-PL230-1
Qualcomm Trusted Execution Environment 5.0 Security Guidelines for Trusted Applications User Guide	80-NM090-2

Title	Number
Qualcomm Runtime Kernel Security User Guide	80-PF777-42

10.2 Acronyms and terms

Acronym or term	Definition
CBOR	Concise binary object representation
CDDL	Concise data definition language
CDN	CBOR diagnostic notation
CDR	Cellular detailed report
CE	Customer engineering
CSR	Cellular summary report
DN	Distinguished name
Feature ID	Feature identifier
IDL	Interface definition language
ISV	Independent software vendor
LE	Linux Enabled
OEM ID	OEM identifier
OTA	Over-the-air
PFM	Platform feature manager
PK hash	Public key hash
QFPROM	Qualcomm fuse-programmable read-only memory
Qualcomm TEE	Qualcomm Trusted Execution Environment
Qualcomm WES	Qualcomm wireless edge services
RPC	Remote procedure calls
RTIC	Real time integrity check
SMSR	Short messaging service report
TA	Trusted application
TLOC	Trusted location
TTIME	Trusted time