WIDEVINE

Widevine Modular DRM Security Integration Guide for Common Encryption (OEMCrypto)

Version 16.2

December 31, 2019

Revision History

Version	Date	Description	Author
1	4/5/2013	Initial revision Refactored from *Widevine Security Integration Guide for DASH on Android Devices*	Jeff Tinker, Fred Gylys-Colwell, Edwin Wong, Rahul Frias, John Bruce
2	4/9/2013	Update to reflect License Protocol V2.1	Jeff Tinker, Fred Gylys-Colwell
3	4/25/2013	Clarified refresh key parameters	Jeff Tinker, Fred Gylys-Colwell
4	5/9/2013	Clarify signature length in GenerateRSASignature	Fred Gylys-Colwell
5	8/6/2013	Add Out-Of-Resource and Key Expired error codes	Fred Gylys-Colwell
9	2/25/2014	Add Version 9 updates	Fred Gylys-Colwell
10	3/9/2015	Add Version 10 updates	Fred Gylys-Colwell
10.1	3/27/2015	Add LoadTestRSAKey to API version 10, and discuss optional API	Fred Gylys-Colwell
10.2	4/21/2015	Add keybox definitions	Fred Gylys-Colwell
10.3	9/23/2015	Clarify HDCP 2.2 requirements	Fred Gylys-Colwell
11	10/31/2015	Add Version 11 updates	Fred Gylys-Colwell
11.1	4/4/2016	Specify generic encryption buffer size	Fred Gylys-Colwell
11.2	5/16/2016	Update offset values in DecryptCENC	Fred Gylys-Colwell
12	11/28/2016	Add Version 12 updates, includes provisioning 3.0	Fred Gylys-Colwell
13	1/19/2017	Add V13 updates, includes SRM and big usage table	Fred Gylys-Colwell
13.1	1/31/2017	Recent decrypt prevents UsageReport	Fred Gylys-Colwell
13.2	5/16/2017	Update key control block verification field	Fred Gylys-Colwell
14	Dec 2017	Add version 14 updates, includes entitlement license	Fred Gylys-Colwell
14.1	5/16/2018	Correct PST Report structure	Fred Gylys-Colwell
15	10/15/2018	Full Decrypt Path Testing, recoverable errors, resource ratings, sandbox support	Fred Gylys-Colwell
15.1	1/30/2019	modification to Full Decrypt Path Testing API.	Fred Gylys-Colwell
15.2	4/29/2019	Updates to HDCP 2.3 and resource ratings	Fred Gylys-Colwell
16	9/12/2019	ODK Library, Core Messages, Combined Decrypt calls	Fred Gylys-Colwell
16.1	12/9/2019	ODK Library + Core Messages updates	Fred Gylys-Colwell
16.2	12/31/2019	ODK Library Updates (renewal message updates)	Fred Gylys-Colwell

Table of Contents

Terms and Definitions

Common Encryption (CENC) — ISO/IEC 23001-7 standards based scheme for encryption and key management

Content Decryption Module (CDM) — the software that calls the OEMCrypto library and implements CENC.

Digital Content Protection (DCP) — (https://digital-cp.com/) The consortium of companies that specifies HDCP.

Device Id — A short string that uniquely identifies the device. For devices with a keybox, this is the 32 byte string from the keybox. For Provisioning 3.0 devices, this is another stable unique identifier, such as the serial number.

Device Key — 128-bit AES key assigned by Widevine and used to secure licenses. This is part of the keybox, and is used for Provisioning 2.0.

DRM Certificate — A certificate provided to the device from a provisioning server. The DRM certificate is used to identify the device and attest its security level to a license server. The DRM certificate's signing chain includes a Google signature. A device may have multiple DRM certificates corresponding to multiple content providers.

Factory Provisioning — The process of installing a Keybox or OEM Certificate that has been constructed for a specific device. This is done before the device reaches the customer.

Keybox — Widevine structure containing keys and other information used to establish a root of trust on a device. The keybox is either installed during manufacturing or in the field. Factory provisioned devices have a higher level of security and may be approved for access to higher quality content. Used in Provisioning 2.0.

License - Authenticated data object / message which contains cryptographic keys needed to decrypt media content, as well as policy information about the usage of those keys and their security robustness requirements. A License is intended for a single device, and is non-transferrable.

OEM Certificate — A certificate provided to the device by the OEM. The OEM certificate is used to identify the device and attest its security level to the provisioning server.

OEMCrypto — the low level library implemented by the OEM to provide key and content protection, usually in a separate secure memory or process space. This term refers to both the API described in this document and the library implementing the API.

Over-the-Air (OTA) Provisioning - Provisioning of DRM credentials after the device reaches the user.

Provider Session Token (PST) — Token which can be assigned by a content provider to the

session, and which is used for enabling collection of usage information for the session..

Private Key — DRM and OEM certificates will have an RSA public key embedded in them. The corresponding RSA private key will be stored on the device and must be either encrypted or protected from user space memory.

Provisioning — Install a certificate or keybox on the device. See the section below for details.

Provisioning 2.0 — Provisioning protocol which uses a Keybox to request a DRM certificate from a provisioning server. A device should use either Provisioning 2.0 or 3.0.

Provisioning 3.0 — Provisioning Protocol which uses an OEM Certificate to request a DRM certificate from a provisioning server. A device should use either Provisioning 2.0 or 3.0.

System Renewability Message (SRM) — Data object created by the DCP organization, which is used to revoke HDCP keys.

Trusted Execution Environment (TEE) — Opaque runtime environment in which none of the resources (memory, registers, etc) are directly available to other runtime environments such as kernel or user space. TEE's only execute trusted code such as OEMCrypto.

References

DASH - 23001-7 3rd Edition ISO -BMFF Common Encryption Specification

DASH - 14496-12 ISO BMFF-Amendment

W3C Encrypted Media Extensions (EME)

Widevine Modular DRM Security Integration Guide for Common Encryption (CENC) : Android Supplement

Draft International Standard ISO/IEC DIS 23001-7

Widevine Level-1 Provisioning Models

Audience

This document is intended for SOC and OEM device manufacturers to integrate with Widevine content protection using Common Encryption (CENC) on consumer devices.

Purpose

This document describes the low-level security APIs used in Widevine content protection for playing content using MPEG Common Encryption (CENC). Examples of such media, but not

limited to, are MPEG-DASH, and Apple HLS.

This document defines the Widevine Modular DRM functionality common across device integrations that use the OEMCrypto integration API. There are supplementary documents describing the integration details for each supported platform, as listed in the References section.

Overview of Widevine Content Protection System

The Widevine Content Protection System uses a hierarchical system of trust. The root of trust is based on an OEM Certificate or keybox which is typically installed at the factory. It is OEMCrypto's job to prevent the private keys of the keybox or OEM Certificate from being visible to the user.

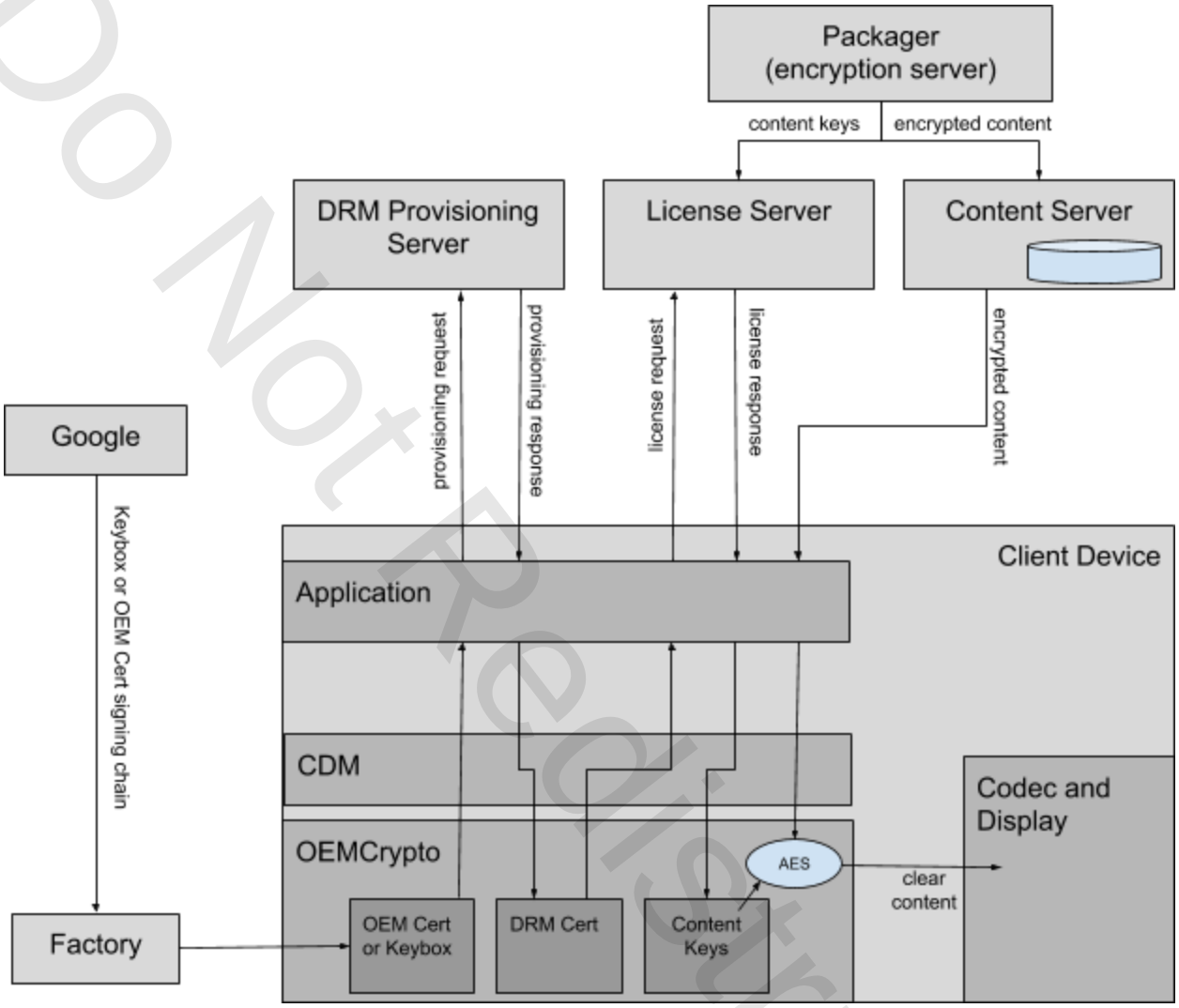
A DRM Certificate is provisioned over the air (OTA) onto the device in the field. A provisioning request is sent from the device with either keybox information or an OEM Certificate. The provisioning server sends a DRM Certificate and an encrypted private key to the device. It is OEMCrypto's job to re-encrypt the private key and prevent the private key from being visible to the user.

The keys needed to decrypt the content come from a license server in a Content License. A set of keys and policy restrictions on the keys' use is encrypted and sent to the device. It is OEMCrypto's job to verify that the license has not been tampered with, decrypt the content keys, prevent the content keys from being visible to the user, and enforce all license restrictions. The license request message will use the DRM Certificate to identify the device to the license server and attest to the device's security features.

The keys needed to decrypt the content come from a license server in a Content License. A set of keys and policy restrictions on the keys' use is encrypted and sent to the device. It is OEMCrypto's job to verify that the license has not been tampered with, decrypt the content keys, prevent the content keys from being visible to the user, and enforce all license restrictions. The license request message will use the DRM Certificate to identify the device to the license server and attest to the device's security features.

Encrypted content is prepared and stored in a content library. The content is encrypted using a unified standard to produce one set of files that play on all compatible devices. The encrypted streaming content is delivered from the content library to the client devices via the public Internet. OEMCrypto's job is to decrypt the content, and to enforce any license restrictions such as time limits or output protection.Encrypted content is prepared and stored in a content library. The content is encrypted using a unified standard to produce one set of files that play on all compatible devices. The encrypted streaming content is delivered from the content library to

the client devices via the public Internet. OEMCrypto's job is to decrypt the content, and to enforce any license restrictions such as time limits or output protection.



Security Robustness Levels

Content protection is dependent upon the security capabilities of the device platform. Ideally, security is provided by a combination of hardware security functions and a hardware-protected video path; however, some devices lack the infrastructure to support this security.

Widevine security levels are based on the hardware capabilities of the device and embedded platform integration.

Security Level	Secure Boot Loader	Widevine Key Provisioning	Security Hardware or Trusted Execution Environment	Widevine Keybox and Video Key Processing	Hardware Video Path
Level 1	Yes	Factory	Yes	Keys never exposed in	Hardware Protected Video

				clear to host CPU	Path
Level 2	Yes	Factory	Yes	Keys never exposed in clear to host CPU	Software Protected Video Path
Level 3	No	Field	No	Clear keys exposed to host CPU	Software Protected Video Path

An OEM-provided OEMCrypto library is required for implementation of Widevine security Level 1 or 2. This OEMCrypto library shall run in a Trusted Execution Environment, which shall have a signed, secure boot loader.

General Recommendations

Although OEMCrypto usually runs within a Trusted Execution Environment, we should still avoid giving unnecessary information about how the system works to an attacker.

- Widevine requires that production versions of the OEMCrypto library be stripped of debugging symbols and libraries should be linked with as few external symbols as possible -- i.e. with hidden visibility.
- Widevine requires that error logging messages be obfuscated. If possible, debugging logs should be removed from production systems.
- If available, the code should be built with Stack Smash Protectors (SSP) enabled. Other security tools provided by the environment should also be enabled.
- If possible, the Trusted Application (TA) should be encrypted as well as signed so that attackers cannot perform static analysis on OEMCrypto.
- The Root of Trust should be heavily protected. For example, it should not be left unencrypted in memory when not in use. This will make it harder for an attacker who has access to runtime memory through another attack.
- Code should be reviewed by an engineer who is not the author, and other “Industry best practices” shall be followed.
- The TEE should have hardware Anti-Rollback turned on for production devices.

Unless otherwise stated, buffers that are passed into OEMCrypto should not be writeable by the REE while OEMCrypto is processing them. For example, it should be impossible for the REE to modify data between the time OEMCrypto verifies a signature and then processes the data. There are exceptions to this rule: for example, buffers that hold encrypted content may be kept in memory that is shared between the REE and the TEE. In the functions below, these buffers are marked as pointers to OEMCrypto_SharedMemory. The type OEMCrypto_SharedMemory is typedef to a uint8_t.

Provisioning

Provisioning refers to the process of installing a key or set of keys that can be used to authenticate the device to a server. Each device will have a unique keybox or OEM certificate provisioned, usually at the factory. In the field, a device will use these provisioned tokens to request a DRM certificate from a DRM provisioning server. This might happen multiple times, and a device may use different DRM certificates with different content providers.

Factory versus Field (OTA) Provisioning

Factory provisioning refers to the initial installation of a keybox or OEM certificate by the manufacturer. Field provisioning for a keybox refers to a device generating its own keybox. This is not allowed for Level 1 or Level 2 devices.

Field provisioning also refers to a device sending a provisioning request to a DRM server and then installing the associated keys. This is done by Level 1 and Level 3 devices.

DRM Certificate

A cryptographic token which a device uses to authenticate itself with a license server. A DRM certificate may have a short lifespan, or it may only be valid for a single content provider. For these reasons, a device may need to request multiple DRM certificates and may need to have different DRM certificates loaded in different sessions.

DRM Provisioning 2.0 -- With a Keybox

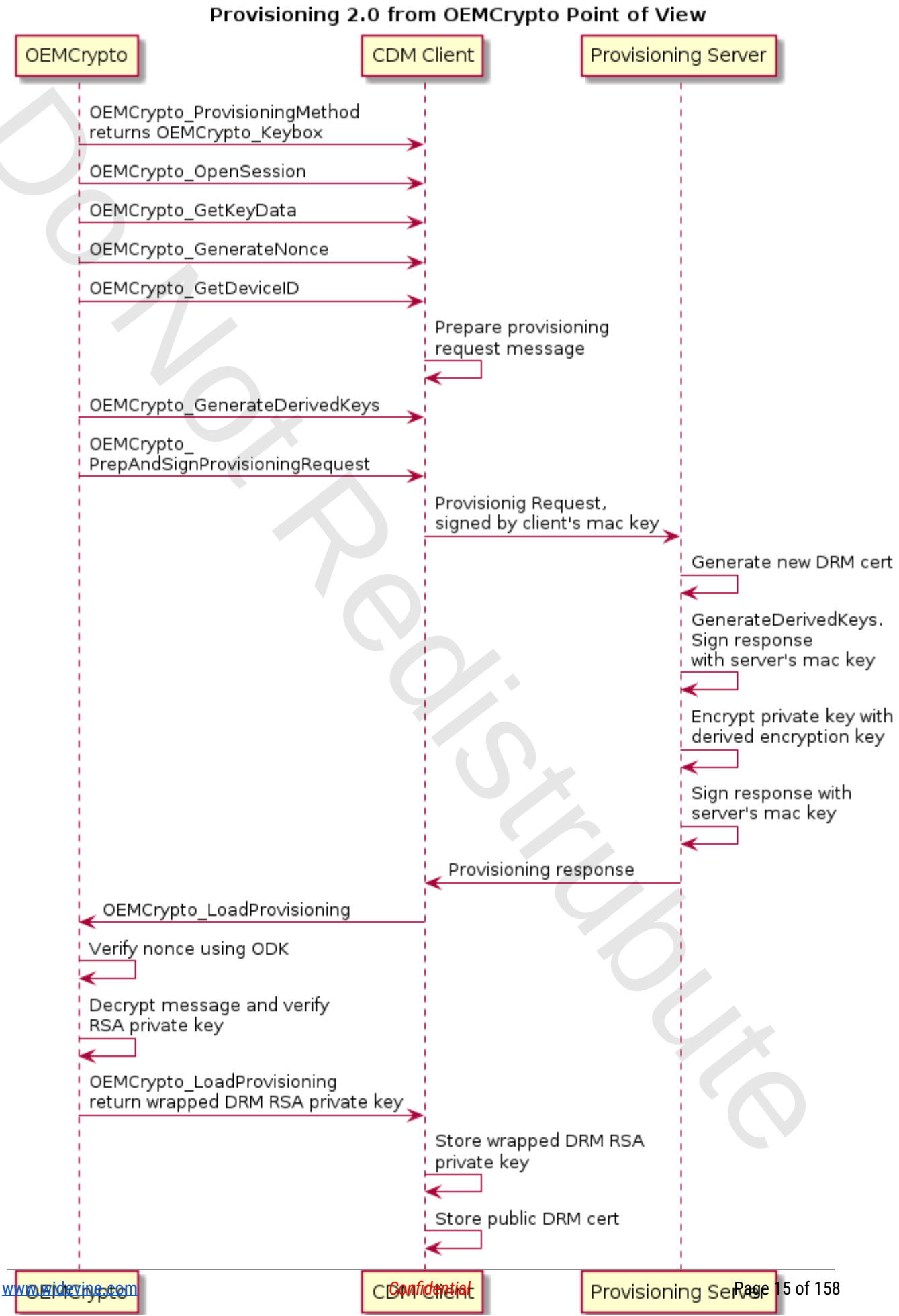
Traditionally, a Widevine keybox is installed on a device to establish a root of trust, which is used to secure content on the device. The device's security hardware, where applicable, is used to protect the contents of the keybox when it is stored. The device key in the keybox is used in the process of decrypting the media content played by the device. Google will support this provisioning method for the foreseeable future, but OEMs creating new devices are encouraged to use Provisioning 3.0 described below.

Each Widevine keybox is associated with a device ID. Every device should have a unique ID. For factory-provisioned devices, the manufacturer will assign the ID when requesting keyboxes.

In addition to the device ID, there is a Widevine-assigned per-model system ID in the keybox that ensures keyboxes are unique across manufacturers and device models. Two device models may use the same device ID since they will have different system IDs. Widevine assigns system IDs based on the Manufacturer/Brand, device type, and model year in the keybox request. The Manufacturer/Brand field in the keybox request is not case sensitive.

When an application first requests a license for content, the CDM layer will look for an appropriate DRM certificate. If one is not found, it will return an error to the application. The application will then initiate a provisioning request. The sequence diagram for a provisioning request for devices with a keybox is below.

Sequence Diagram for Provisioning 2.0



Keybox Definition

The following fields are stored in the keybox:

Field	Description	Size (bytes)
Device ID	C character string identifying the device. It is padded with NULL characters, '\0', to make it 32 bytes long. If it is 32 bytes long, it is not NULL terminated.	32
Device Key	128 bit AES key assigned to device, generated by Widevine.	16
Key Data	Encrypted data	72
Magic	Constant used to recognize a valid keybox: "kbox" (0x6b626f78)	4
CRC	CRC-32-IEEE 802.3 validates integrity of the keybox - computed over whole keybox excluding CRC field.	4
	Total Size	***128***

DRM Provisioning 3.0 -- With an OEM Certificate

Provisioning 3.0 is a way for OEMs to provision their devices using an X.509 certificate generated by the OEM, instead of using a keybox generated by Google. For a description of keybox provisioning, see the section above. This PKI-based approach allows for other third parties to provision devices for bootstrapping DRM or other services. Provisioning 3.0 is the preferred provisioning method going forward. Keyboxes will still be supported by the CDM layer and by Google DRM certificate provisioning servers. There is no plan to deprecate keyboxes at this time, but they will be gradually phased out.

The OEM certificate will have a signing chain that is signed by Google and the OEM. Similar to a keybox, this root of trust can be used with a Google DRM provisioning server. It can also be used with an application specific DRM provisioning server to obtain a DRM certificate that is valid only for specific applications. This allows applications to work in environments where Google servers are not accessible.

OEMs who wish to use Provisioning 3.0 certificates should return OEMCrypto_OEMCertificate from a call to OEMCrypto_GetProvisioningMethod(). They should implement OEMCrypto_GetOEMPublicCertificate(), OEMCrypto_LoadOEMPrivateKey(), and make sure OEMCrypto_PrepAndSignProvisioningRequest works with the OEM certificate, as described below.

Implementations which have not yet been updated to Provisioning 3.0 should return OEMCrypto_UsesKeybox from a call to OEMCrypto_GetProvisioningMethod(). They should return OEMCrypto_ERROR_NOT_IMPLEMENTED from calls to OEMCrypto_GetOEMPublicCertificate() and OEMCrypto_LoadOEMPrivateKey(). They can ignore the rest of this section.

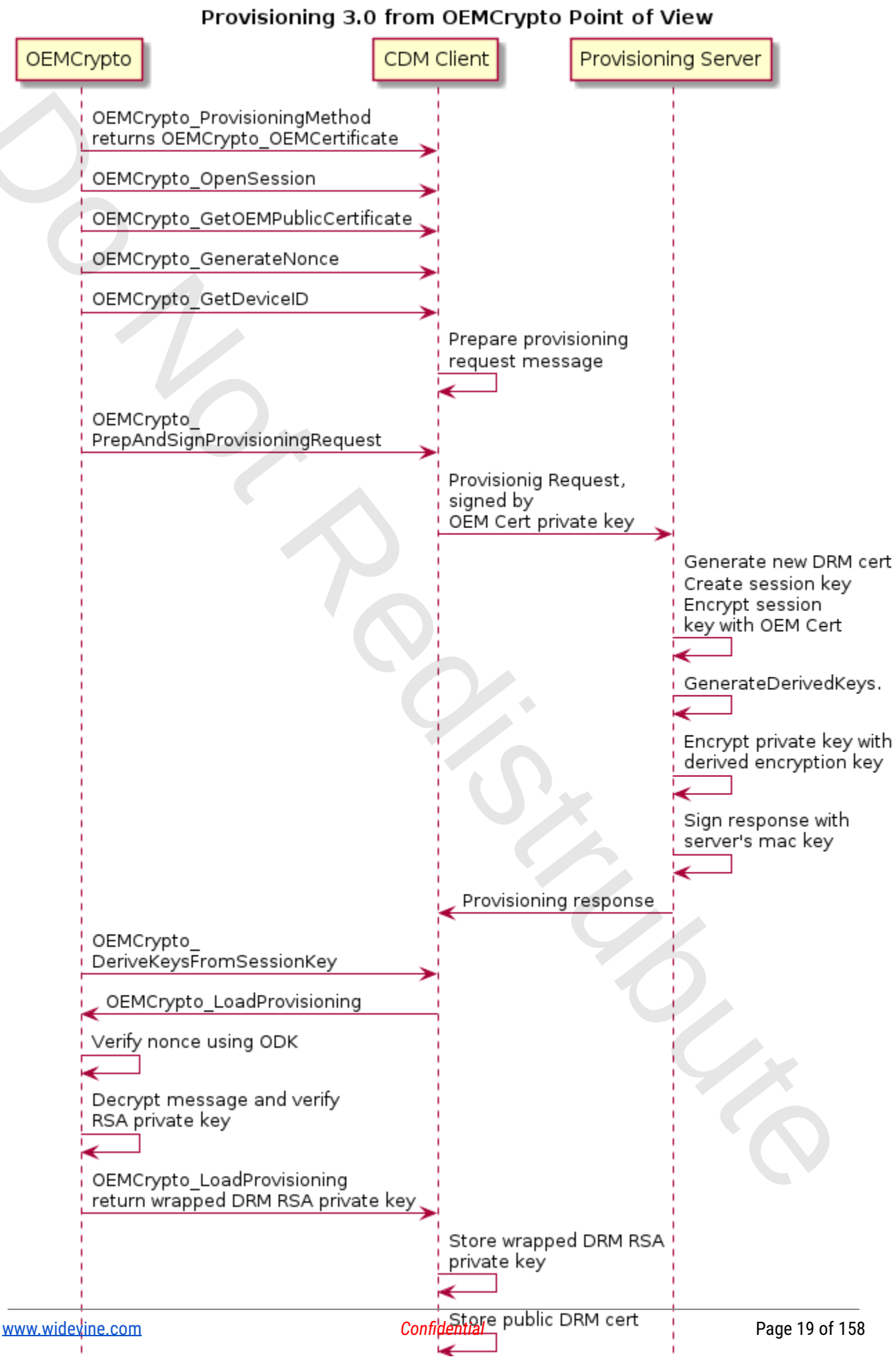
For a complete description of Provisioning 3.0, please see the document "Widevine Provisioning 3.0 Design". OEMs will request a single X.509 CA certificate from Google for each make and model of the device, and use them to sign the device specific certificates which the OEM will generate for each device. The device specific certificate will be installed on the device in the

factory. OEMCrypto will pass the certificate up to the CDM layer when the function OEMCrypto_GetOEMPublicCertificate is called. The OEMCrypto library will load the private RSA key corresponding to the certificate when OEMCrypto_LoadOEMPrivateKey is called. It is the OEM's responsibility to make sure that the private RSA key is not accessible to the user.

It is the OEM's responsibility to make sure that the private key for the OEM certificate not be accessible to user space programs, i.e. must be stored in secure NVRAM or the TEE, or wrapped by a key stored in NVRAM or the TEE. The private key should be treated with the same robustness rules that have always applied to a Widevine keybox or to content keys.

Sequence Diagrams for Provisioning 3.0

Below are sequence diagrams illustrating a Provisioning 3.0 session from an OEMCrypto viewpoint.



Notice that the request is encrypted with the key M1 by the CDM layer. This is not intended to secure content, but allows for user privacy. Similarly, the provisioning response's signature is verified by the CDM layer. This gives security to the user and is not intended to protect the video content.

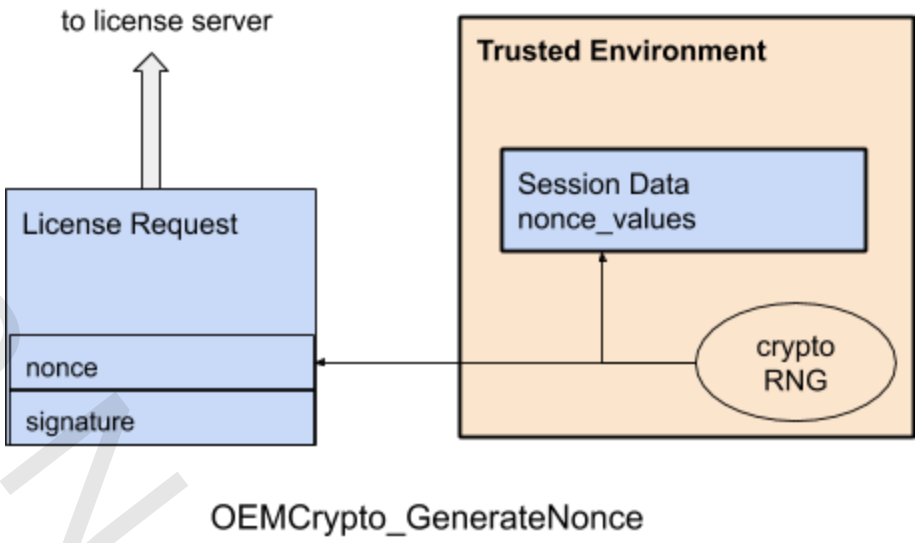
Content License Exchange and Renewal

There are two ways for content keys to be loaded into OEMCrypto -- from a content license containing the keys, or from key data that is access-controlled by an entitlement license. This section discusses content licenses and the next section discusses entitlement licenses. A content license has the keys used to decrypt content embedded in it as a blob of data that is opaque to the application. These content keys are encrypted by an encryption key shared between OEMCrypto and the server. It is OEMCrypto's responsibility to ensure that none of these keys are available to the user. A sequence diagram for the license exchange is shown below. For some cases, the content keys expire before the content is complete. In this case, the CDM will request a license renewal.

The application calls the CDM function getLicenseRequest() to obtain an opaque license request message to send to the license server. The CDM calls the OEMCrypto functions OpenSession, GenerateNonce and OEMCrypto_PrepAndSignLicenseRequest to construct and sign the request message. Once a license server response has been received, the application calls provideLicenseResponse() to initiate signature verification, input validation and key loading. This results in a call to OEMCrypto_DeriveKeysFromSessionKey to prepare signing keys and to OEMCrypto_LoadLicense to load the keys.

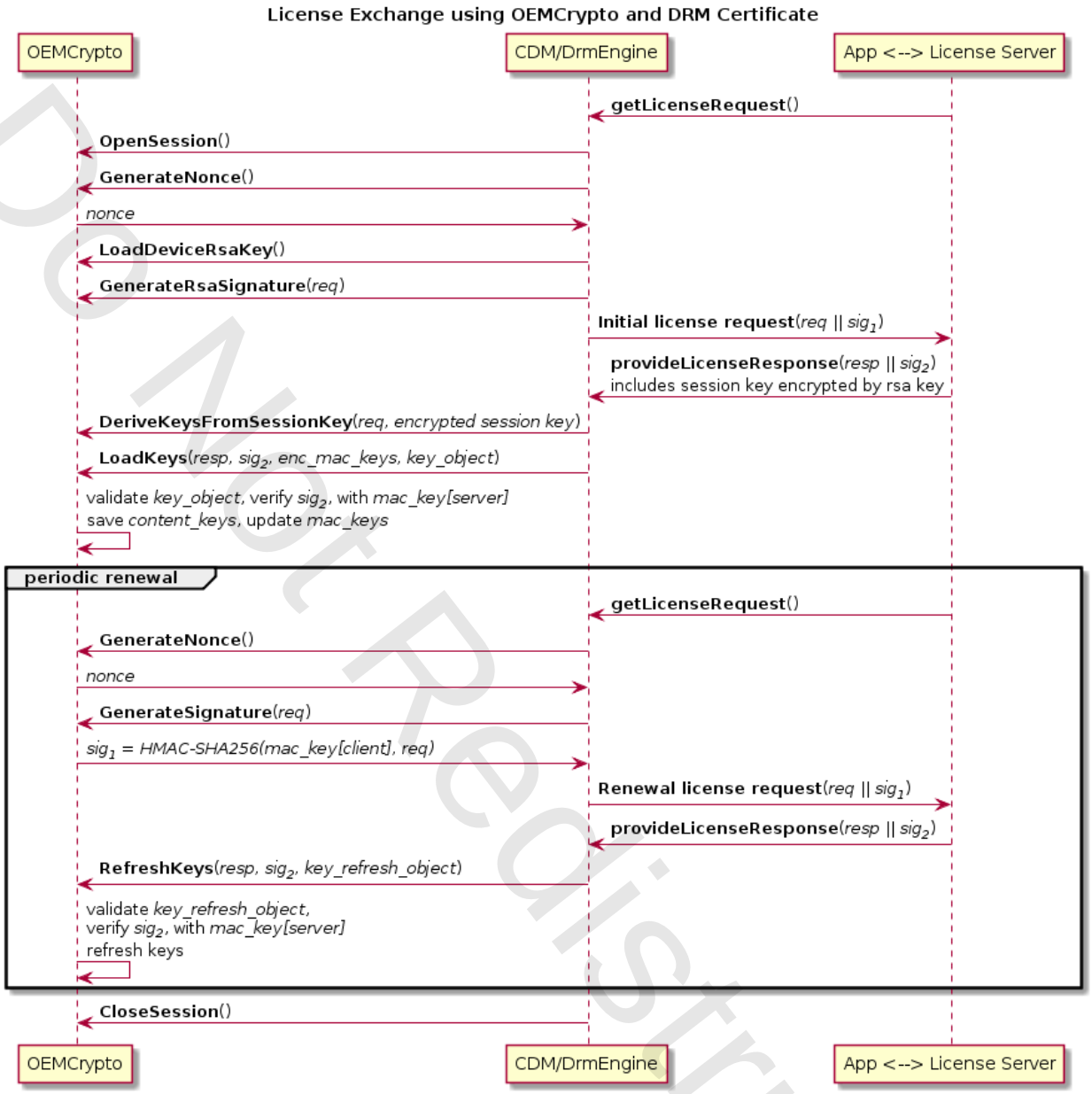
After the initial license has been processed, there is a periodic renewal request/response sequence that may occur during continued playback of the content. The OEMCrypto API calling sequence for renewal is similar to the sequence for the original license message, except that OEMCrypto_LoadRenewal is called instead of OEMCrypto_LoadLicense.

For the licensel requests, the OEMCrypto implementation is required to generate a nonce and a signature that will be included in the request. The nonce is used to prevent replay attacks. A discussion of nonce and replay control is in the section Replay Control -- Nonce and Provider Session Token (PST), below.



OEMCrypto_GenerateNonce

For the license initial and renewal *responses*, the OEMCrypto implementation must verify that the license response and its signature match. Signature verification is discussed in the section Verification of Messages from a Server, below.

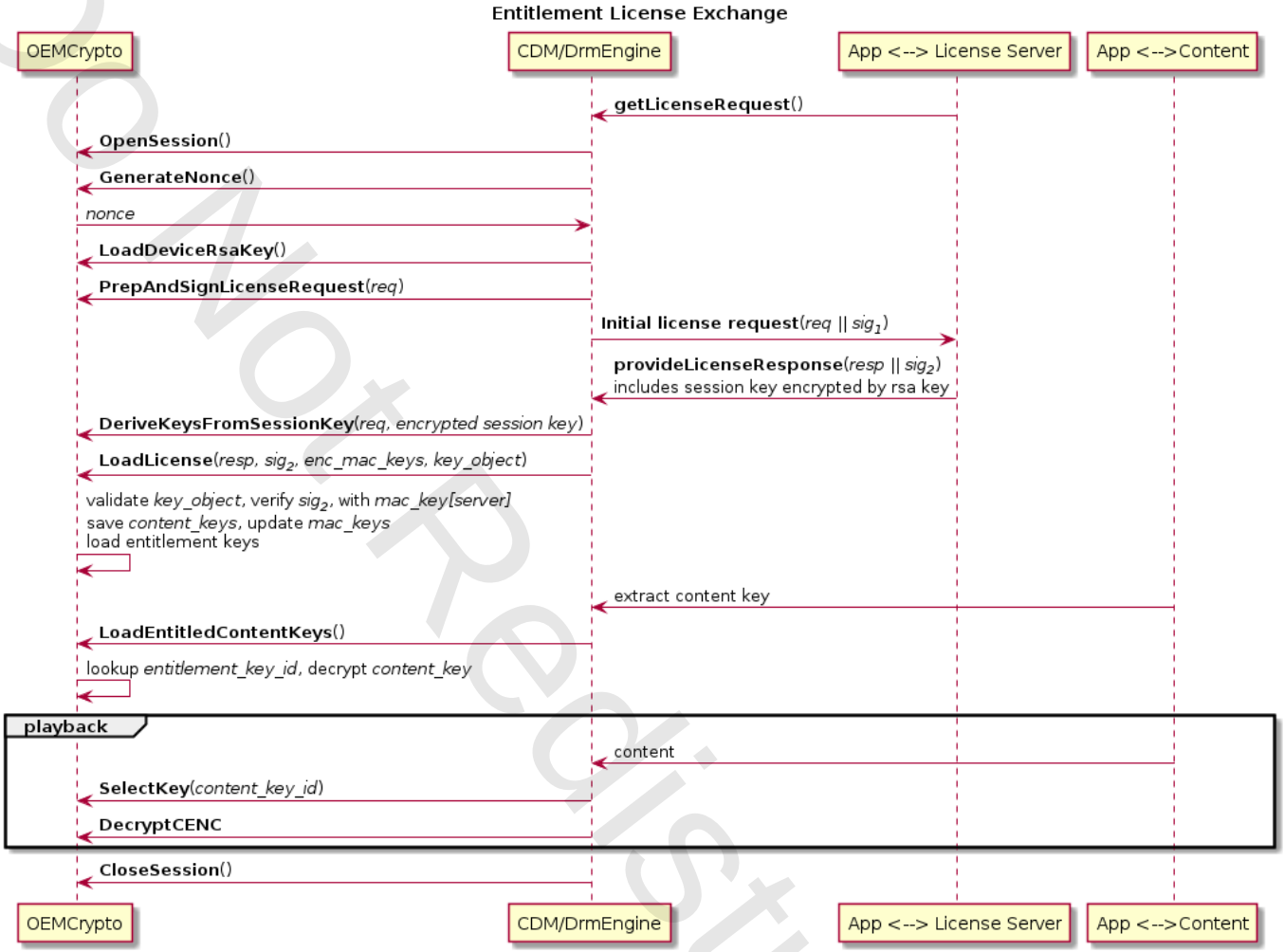


Entitlement License Exchange

Entitlement licensing is a way to provide access to content keys that may be stored elsewhere, such as in the content itself. This can be used to implement content key rotation without requiring new licenses, or access to multiple pieces of content with a single license. The device does not have to send a license request to a server for each section or piece of content. Instead, entitlement keys are delivered to the device in a single entitlement license. The device is then entitled to decode all of the content covered by that entitlement license. An entitlement license may have several entitlement keys. An entry in the session's key table will now contain both a content key and an entitlement key. The content key and the entitlement key will each have a key ID. The content key inherits the key control block from its entitlement key (see Key Control Block for details). The key control block controls policy and security requirements such

as output protection requirements, etc.

A license request for an entitlement license has the same sequence diagram as the content license above. The difference is that after LoadKeys is called to load the entitlement keys, one or more calls to LoadEntitledContentKeys is made, as seen in the diagram below.



An entitlement license can also be renewed. The renewal process for an entitlement license is the same as that for a content license.

A session with an entitlement license may have unused entitlement keys in the key table -- i.e. the entitlement key was loaded in the call to LoadLicense, but no content key is ever loaded. This is valid, but that particular key cannot be selected by a call to SelectKey.

Here are some definitions:

Entitlement License. An entitlement license is a message from a license server that contains encrypted entitlement keys.

Entitlement Keys. Entitlement keys are used to decrypt entitled content keys.

Entitled Content Keys. An entitled content key is a content key that is encrypted by an

entitlement key. It will not be contained in a license message from a license server. It will usually be embedded in the content itself. The entitled content key is used to decrypt the content.

Core Messages and ODK Library

Widevine provides a library of functions called the ODK Library. This library is delivered in source code form and should be in the same repository as the OEMCrypto headers and unit tests. Look in the directory oemcrypto/odk. These functions provide some core functionality that is common to all OEMCrypto implementations and can be written in platform independent C code. Structures and functions that start with the prefix ODK_ are part of this library. See the documents "**Widevine Core Message Serialization**" and "**License Duration and Renewal**" for a complete description of the ODK library.

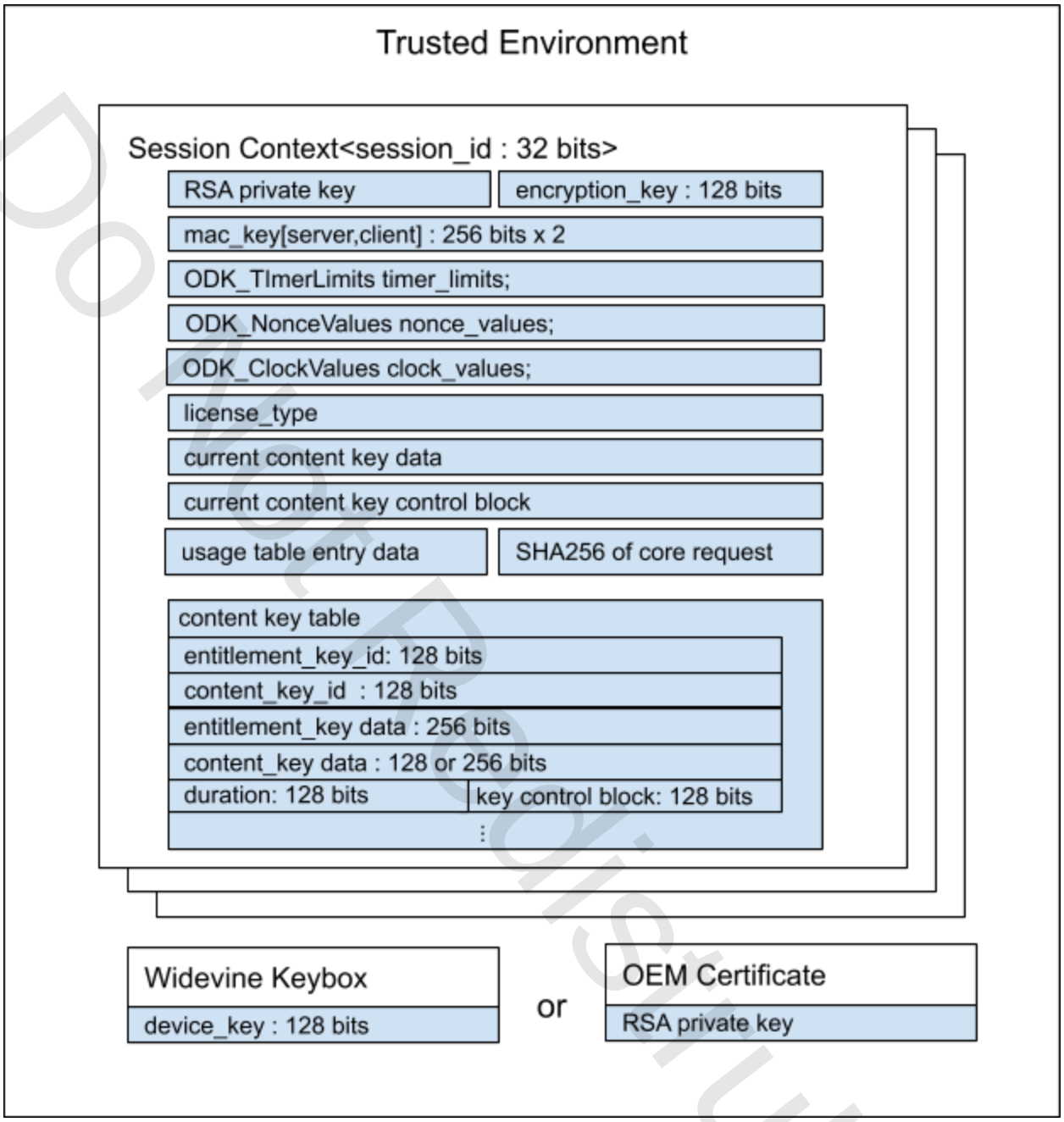
Session Context

One or more crypto sessions will be created to support media playback. An application may use a single session with multiple keys for all of its content (recommended). An application may use one session for video and a different session for audio. An application may also preload several licenses while waiting for the user to decide which video to watch. Most of the OEMCrypto calls require information to be retained in the session context. Each session has its own current content key and its own pair of message authentication keys (mac_keys). Typically, content will have several keys corresponding to audio and video at different resolutions. If the content uses key rotation, there could be as many as 20 keys in a single session. OEMCrypto shall support at least 10 simultaneous open sessions.

The following data is session specific:

- Certificate's private RSA key. A session may be asked to load a DRM Certificate's private key, or the OEM Certificate's private key. Different applications may use different DRM certificates.
- Server HMAC Verification Session Key (mac_key[server] 256 HMAC key) - used to verify messages signed by the server.
- Client HMAC Signing Session Key (mac_key[client] 256 HMAC key) - used to sign messages to the server.
- Session Wrapping Key (128 bit AES key) used to decrypt data from the server. In particular, content keys are encrypted with this key.
- Flag indicating if the license type is a content license or an entitlement license.
- Current content key (128 bit AES key for content or 256 HMAC key for generic encryption's signing and verification functionality).
- Current content key control block (described below).
- Table of keys, which contains
    - Key control blocks. Each key has its own control block because different restrictions may apply to different keys.
    - Content key id (up to 16 bytes). Used to select key for decrypt.

    - Content key data (128 bit AES, or 256 bit HMAC key)
    - Entitlement key id (up to 16 bytes). Not used for content license.
    - Entitlement key data (256 bit AES key).
- TImer limits (struct ODK_TimerLimits) for license duration restrictions. See the document “License Duration and Renewal” for a definition. This document refers to this field with the name timer_limits, but implementations may use local style guides for name choice.
- Clock Values (struct ODK_ClockValues) for tracking current timer values. See the document “License Duration and Renewal” for a definition. This document refers to this field with the name clock_values, but implementations may use local style guides for name choice.
- Nonce values (struct ODK_NonceValues) for tracking the license request’s nonce. This document refers to this field with the name nonce_values, but implementations may use local style guides for name choice.
- Usage table entry data. See the section on replay control below.
- Hash of core license request.



The functions in the Crypto Key Ladder API section are used by the application to generate a license request, and are used to install and update keys for a given session. The functions in the Decryption API section are used to select a current key for the session and to decrypt or encrypt data with the current key. Because different applications may use different DRM certificates, the functions in DRM Certificate Provisioning API are also session specific. Each session may have a different DRM key installed.

The functions in the Crypto Device Control API and Keybox Access and Provisioning API sections are not associated with any one session. There is only one active widevine keybox on the device, either a production keybox or the test keybox. These functions handle initialization of the device itself and accessing keybox information.

When the session is closed via OEMCrypto_CloseSession(), all of the Session Context resources must be explicitly cleared and then released.

Key Derivation

Communication between the client and the server must be authenticated and some elements within it must be encrypted/wrapped. The initial license request is authenticated using the session's DRM private RSA key, while subsequent messages can be authenticated using a set of session MAC keys derived from a common session key. A session wrapping key is also derived from the common session key in order to encrypt/wrap sensitive elements and keys. A provisioning request is authenticated using a session's OEM private RSA key for Provisioning 3.0. For devices with a keybox, a set of session mac keys are derived from the keybox, and these session mac keys are used to sign the provisioning request.

Derivation of the session MAC and wrapping keys is done using a context buffer consisting of the initial request message for a provisioning 2.0 request, the function OEMCrypto_GenerateDerivedKeys uses the device key from the keybox as the input key. For other messages, a session key is encrypted by the server with the RSA public key and passed into the function OEMCrypto_DeriveKeysFromSessionKey. OEMCrypto will decrypt the session key, and then use it as the input key in the key derivation algorithm.

Key derivation is based on NIST 800-108. Specifically NIST 800-108 key derivation using 128-bit AES-128-CMAC as the pseudorandom function in counter mode.

These keys are:

1. encrypt_key: used to encrypt the content key:
   encrypt_key := AES-128-CMAC(device_key, 0x01 || context_enc)

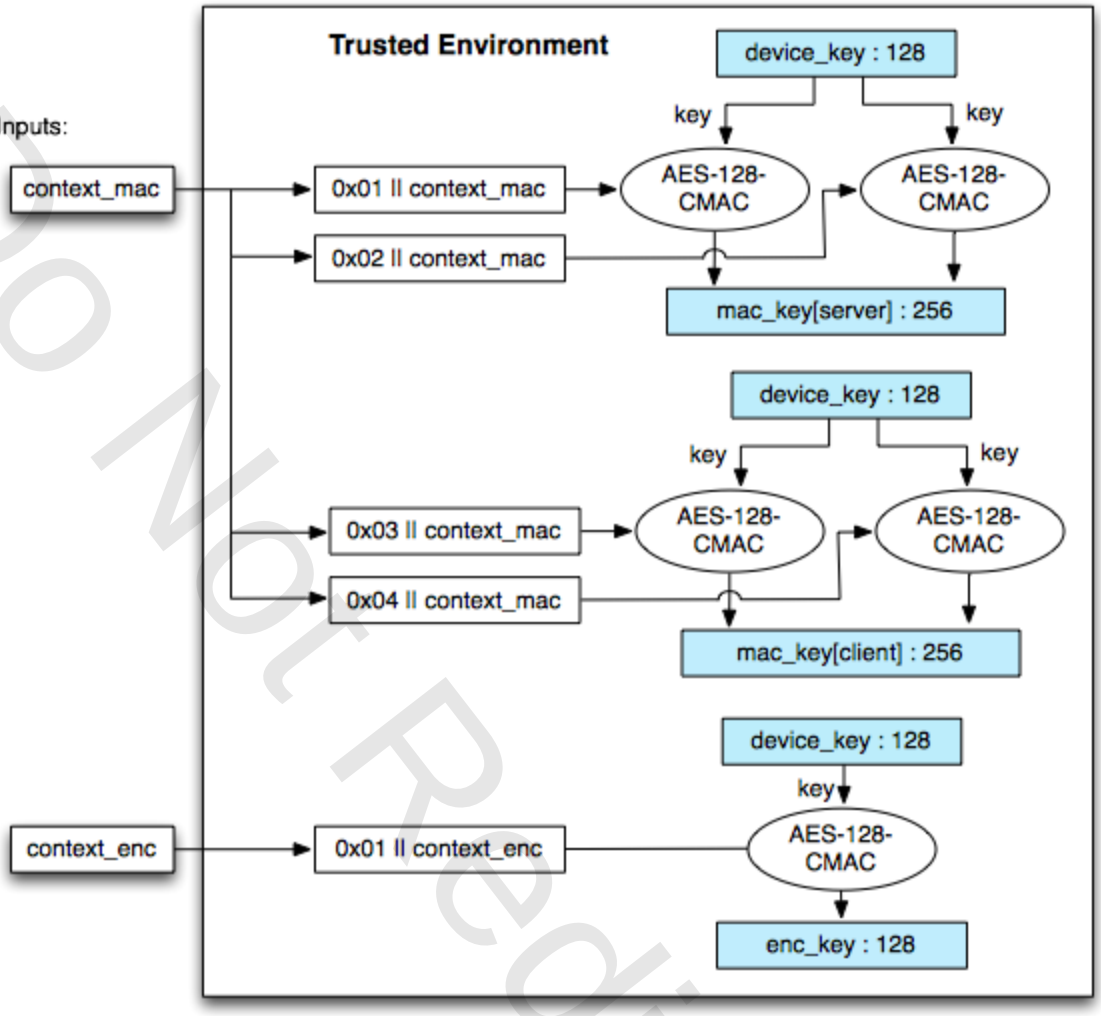
2. mac_keys: used as the hash key for the HMAC to sign and verify license messages:
   mac_key[server] || mac_key[client]

   := AES-128-CMAC(device_key, 0x01 || context_mac) ||
   AES-128-CMAC(device_key, 0x02 || context_mac) ||
   AES-128-CMAC(device_key, 0x03 || context_mac) ||
   AES-128-CMAC(device_key, 0x04 || context_mac)

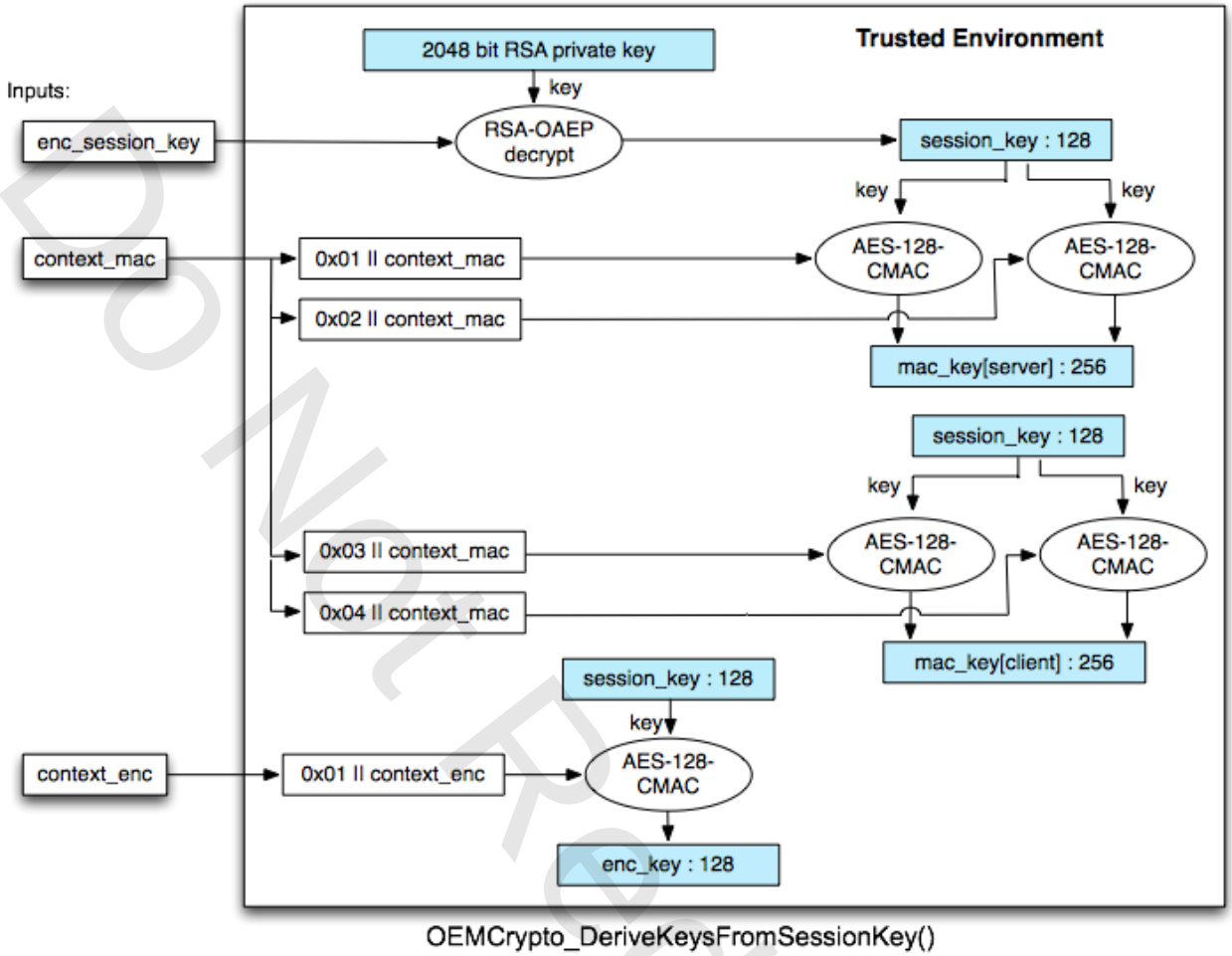
   For the case of license renewal, the mac_keys are generated by the license server, then encrypted and placed in a license response message, which is passed to OEMCrypto through OEMCrypto_LoadKeys. In this case the derivation is as follows:

   mac_keys := AES-128-CBC-decrypt(encrypt_key, iv, encrypted_mac_key)

The data *context_enc* and *context_mac* are provided as parameters to the OEMCrypto API functions that generate these keys, and "||" represents the concatenation operation on message bytes.



OEMCrypto_GenerateDerivedKeys()



OEMCrypto_DeriveKeysFromSessionKey()

Note: the mac_keys computed by either of these functions may be replaced when OEMCrypto_LoadKeys() is called, as it receives new server-generated and encrypted mac_keys.

Signing Messages Sent to a Server

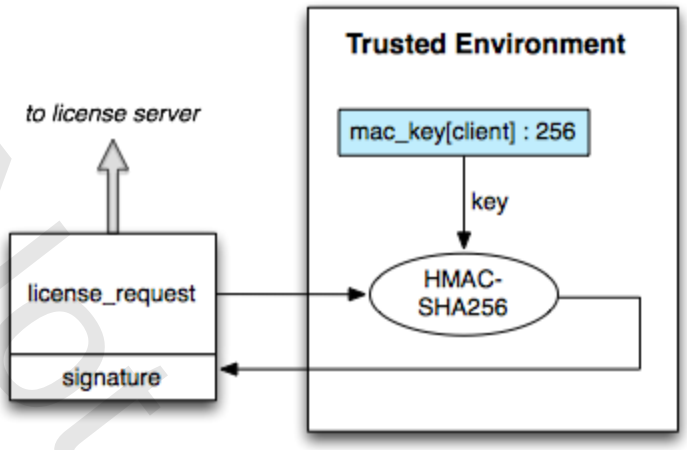
Messages sent to a server will be signed to ensure that the license request can not be modified in transit. Signing is done by OEMCrypto using either

1. the session's RSA private key, which is either
    a. paired to one of the device's DRM Certificate. This is used for an initial license request,
    b. the device's OEM Certificate. This is used for a provisioning request.
2. the session's mac_key[client], which is either
    a. derived using CMAC from one of the derivation functions described above
    b. a MAC key sent by the server and loaded by means of OEMCrypto_LoadKeys. This is used for license renewals and license release messages.

These signing functions specify a session id and should use the current RSA key or HMAC key

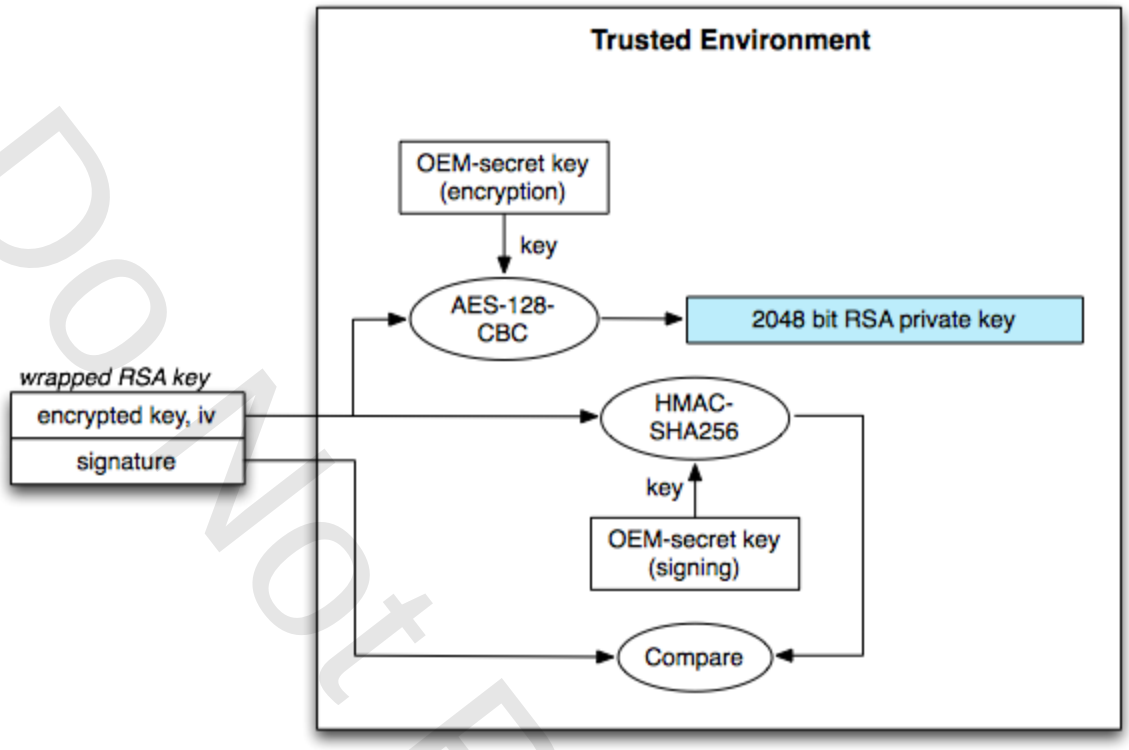
for the specified session.

The function OEMCrypto_PrepAndSignRenewalRequest should use the session's mac_key[client] which was loaded with the license. If the device has a keybox, then the function OEMCrypto_PrepAndSignProvisioningRequest should use the session's derived mac_key[client] to sign a buffer using the HMAC-SHA256 algorithm.



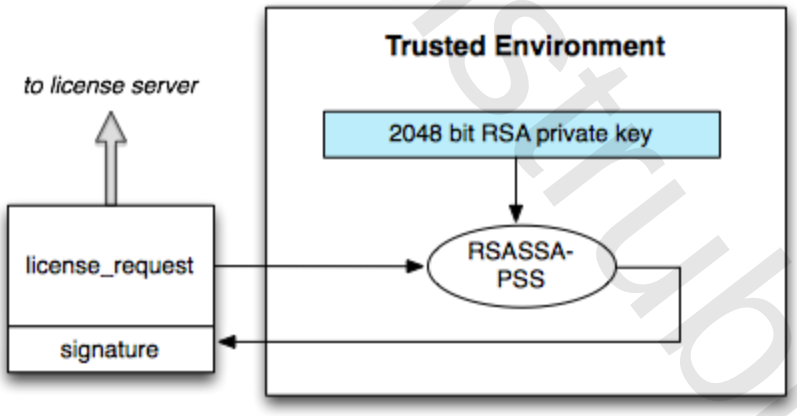
OEMCrypto_GenerateSignature()

In order to sign a message using RSA, OEMCrypto will first be asked to load the private RSA key associated with a DRM Certificate. This will be passed into OEMCrypto_LoadDRMPrivateKey() as a blob of data that was previously wrapped by the function OEMCrypto_LoadProvisioning().



OEMCrypto_LoadDeviceRSAKey()

The function OEMCrypto_PrepAndSignLicenseRequest should use the session's private RSA key to sign a buffer using the RSASSA-PSS algorithm. If the device has an OEM Certificate, then the function OEMCrypto_PrepAndSignProvisioningRequest should also use the session's private RSA key to sign a buffer using the RSASSA-PSS algorithm.



OEMCrypto_GenerateRSASignature()

Data in Messages from a Server

Several functions take pointer offsets to data that came from the server. For each of these functions, the message and its signature are passed in, as well as pointer offsets to data within

the message buffer. OEMCrypto shall verify the signature of the message, as described below, and OEMCrypto shall verify that each of the data elements is within the range of the message. In other words, the data offsets should be positive, and the offset plus the data element length must not exceed the message length. Finally, if oemcrypto is running on an architecture that requires data to be word-aligned in memory, OEMCrypto shall copy the data to a local buffer that is correctly aligned, as needed.

Verification of Messages from a Server

Messages from the server will be signed using the algorithm HMAC-SHA256 and the key mac_key[server].

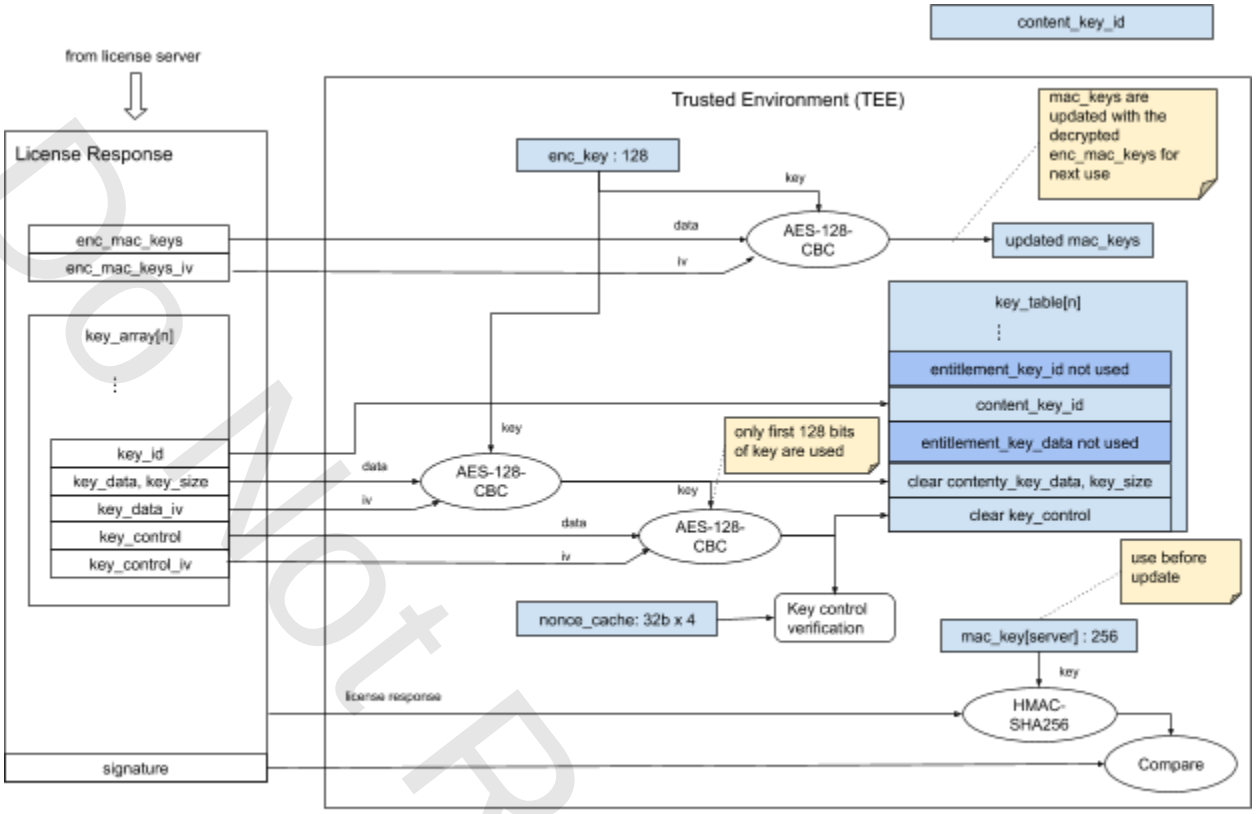
signature == HMAC-SHA256(*mac_key[server], msg*)

where *mac_key[server]* is defined in the Key Derivation section, and *msg* is a byte array provided to the OEMCrypto API function for computation of the signature.

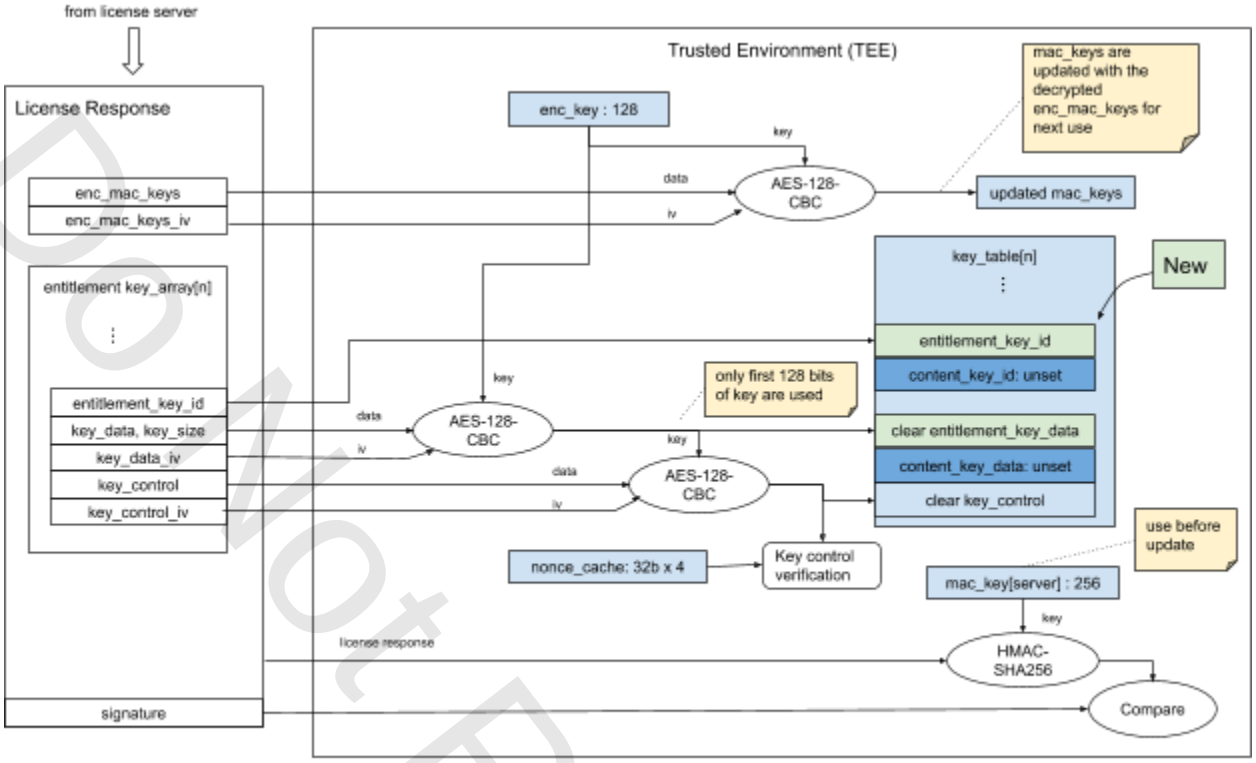
This is done by OEMCrypto in each function that processes a message. The layer above OEMCrypto will parse the message, and pass key data extracted from the message to OEMCrypto along with the message and the signature buffer. OEMCrypto shall verify that the pointers to the key data are contained in the message region, and shall verify that the signature matches the message.

Loading Keys from License

The license response from the license server will be signed by the derived key mac_key[server] and contains key data encrypted/wrapped with derived key enc_key. See the section, Key Derivation, above for a description of derived keys. When the CDM layer calls OEMCrypto_LoadKeys, one of the parameters is license_type, which will indicate if the keys are content keys, or entitlement keys.

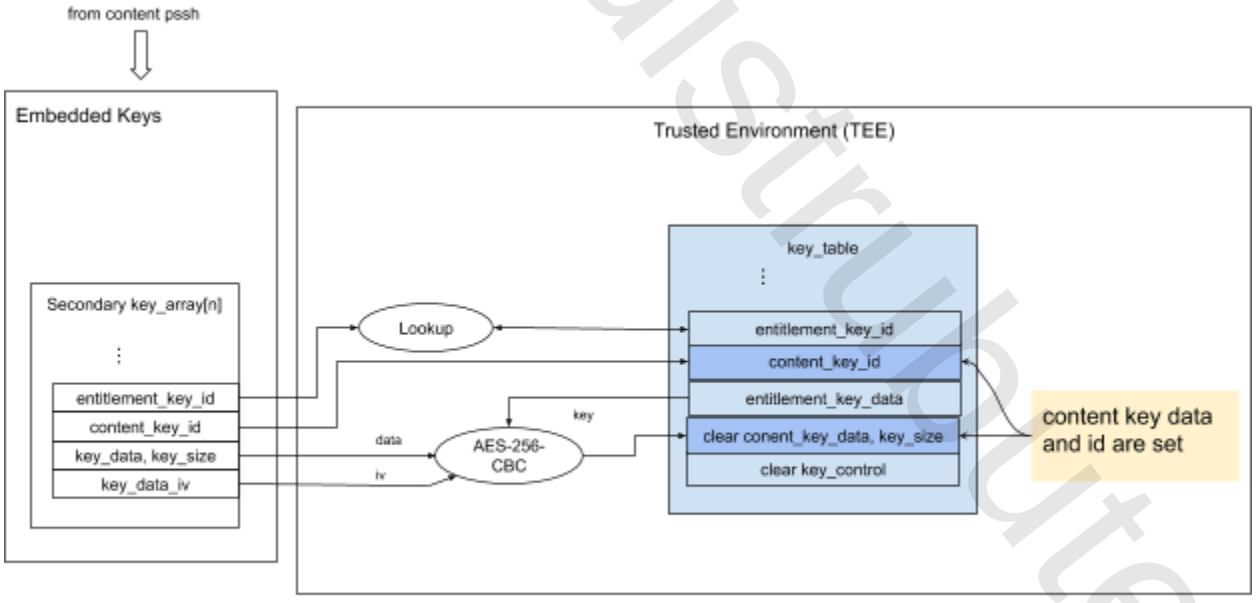


OEMCrypto_LoadKeys() and OEMCrypto_LoadLicense() with license_type = OEMCrypto_ContentLicense



OEMCrypto_LoadKeys() and OEMCrypto_LoadLicense() with license_type = OEMCrypto_EntitlementLicense

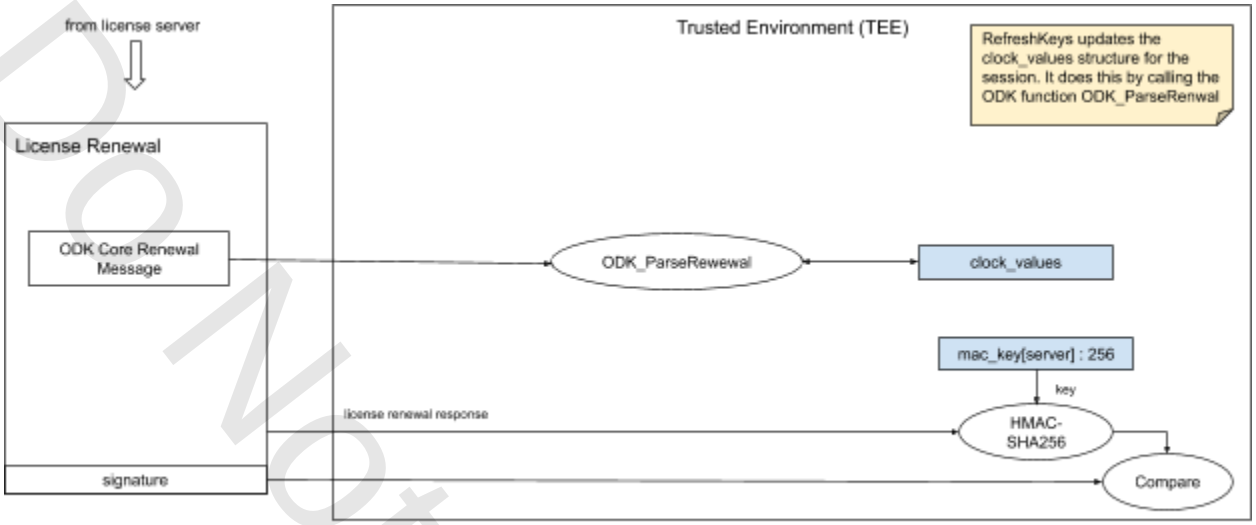
If the license_type was OEMCrypto_EntitlementLicense, then one or more calls to OEMCrypto_LoadEntitledContentKeys will be used to load the content keys.



OEMCrypto_LoadEntitledContentKeys()

Some content also requires licenses to be periodically renewed. This is performed with a call to

the OEMCrypto_LoadRenewal. If license_type was OEMCrypto_ContentLicense, then any key ids in the license refer to content key ids. If license_type was OEMCrypto_EntitlementLicense, then any key ids in the license refer to entitlement key ids.



OEMCrypto_RefreshKeys()

Key Control Block

There is a data element referred to as “key control block” associated with each content and entitlement key. The key control block specifies security constraints for the stream protected by that key, which need to be enforced by the trusted environment. These security constraints/requirements include the Secure Data Path requirement, key validity lifetime and output protection controls.

For most content, the audio and video streams have different security robustness requirements. While the video can be processed through a path entirely protected by hardware, the audio may be processed through a path that may not, due to processing that is performed on the audio stream by the primary CPU after decryption. To maintain security of the video stream, the audio and video streams are encrypted with separate keys. The key control block provides a means to enforce data path security requirements for each media stream.

The key control block is also used to securely limit the lifetime of keys, by associating a timeout value with each content key. The timeout is enforced in the trusted environment. Additionally, the key control block contains output control bits, enabling secure enforcement of the output controls such as HDCP.

When a license requires HDCP, a device may use a wireless protocol to connect to a display only if that protocol supports the version of HDCP as required by the license. Both WirelessHD (formerly WiFi Display) and Miracast support HDCP.

The key control block structure contains fields as defined below. The fields are defined to be in big-endian byte order. The 128-bit key control block is AES-128-CBC encrypted with the

content key it is associated with, using a random IV.

Key Control Block: 128 bits

Field	Description	Bits
Verification	Constant bytes “kctl”, “kc09”, “kc10”, “kc11”, ... “kc15”. A device that supports the current version of this API must support all verification strings.	32
Duration	Obsolete. OEMCrypto should NOT enforce this duration.	32
Nonce	Ensures that key control values can’t be replayed to the secure environment. See “Replay Control -- Nonce and Provider Session Token (PST)”.	32
Control Bits	Bit fields containing specific control bits, defined below	32

Control Bits definition: 32 bits

bit 31	Observe_DataPathType 0 = Ignore 1 = Observe
bit 30	Observe_HDCP 0 = Ignore 1 = Observe
bit 29	Observe_CGMS 0 = Ignore 1 = Observe
bit 28	Require_AntiRollback_Hardware 0 = not require 1 = require
bits 27..25	Reserved set to 0
bit 24	Allow_Hash_Verification If set, content encrypted by this key may be used for full decrypt path testing.
bit 23	Shared_License obsolete. must be set to 0.
bit 22	SRM_Version_Required If set, then a minimum SRM version is required for this key
bit 21	Disable_Analog_Output If set, data decrypted with this key may not be sent to analog output
bits 20..15	Minimum_Security_Patch_Level

	OEM or Device specific software patch level
bits 14..13	Replay_Control 0x0 - Session Usage table not required. 0x1 - Nonce required, create entry in Session Usage table. 0x2 - Require existing Session Usage table entry or Nonce.
bits 12..9	HDCP_Version 0x0 - No HDCP required 0x1 - HDCP version 1.0 required 0x2 - HDCP version 2.0 Type 1 required 0x3 - HDCP version 2.1 Type 1 required 0x4 - HDCP version 2.2 Type 1 required 0x5 - HDCP version 2.3 Type 1 required 0xF - Local display only. The content should not be available to any external display, including HDMI, DTCP, Miracast, or any other digital output, regardless of HDCP level.
bit 8	Allow_Encrypt 0 = Normal 1 = May be used to encrypt generic data.
bit 7	Allow_Decrypt 0 = Normal 1 = May be used to decrypt generic data.
bit 6	Allow_Sign 0 = Normal 1 = May be used to sign generic data.
bit 5	Allow_Verify 0 = Normal 1 = May be used to verify signature of generic data.
bit 4	Data_Path_Type 0 = Normal 1 = Secure only
bit 3	Nonce_Enable 0 = Ignore Nonce 1 = Verify Nonce
bit 2	HDCP 0 = HDCP not required 1 = HDCP required
bit 1..0	CGMS 0x00 - Copy freely - Unlimited copies may be made 0x02 - Copy Once - Only one copy may be made 0x03 - Copy Never

Key Control Block Algorithm

The key control block is a member of the OEMCrypto KeyObject data type, which is supplied as

the *key_array* parameters to LoadKeys() or in the ParsedLicense returned by ODK_ParseLicense. The following steps shall be followed to decrypt, verify, and apply the information in the key control block. Unless otherwise noted, these steps should be performed during key control block verification in OEMCrypto_LoadKeys.

1. Verify that the key_control pointer is non-NULL. If not, return OEMCrypto_ERROR_CONTROL_INVALID.
2. AES-128-CBC-decrypt the content key {key_data, key_data_iv, key_data_length} with enc_key.
3. AES-128-CBC-decrypt the key control block {key_control, key_control_iv} using the first 128 bits of the clear content key from step 2.
4. Verify that bytes 0..3 of the decrypted key control block contain the pattern ‘kctl’, ‘kc09’, ‘kc10’, ‘kc11’, ... or ‘kc15’. If not, return OEMCrypto_ERROR_CONTROL_INVALID. In particular, it is important that devices to not accept key control blocks for future versions, such as ‘kc16’.
5. If Require_AntiRollback_Hardware is set, and the device does not have hardware protection preventing rollback of the usage table, do not load keys and return OEMCrypto_ERROR_UNKNOWN_FAILURE.
6. If Minimum_Security_Patch_Level is greater than the OEM defined TEE patch level, do not load keys and return OEMCrypto_ERROR_UNKNOWN_FAILURE. See the section Security Patch Level for more details.
7. Apply the control fields:
    a. Replay_Control and Nonce_Enable -- if required, verify the nonce. See the section Replay Control -- Nonce and Provider Session Token (PST) for details on verifying the nonce, and for details on when to restrict replay. If the nonce verification fails, return OEMCrypto_ERROR_CONTROL_INVALID.
    b. DataPathType -- If Observe_DataPathType is 1 the DataPathType setting must be enforced, otherwise the data path type must not be changed from its current value. If DataPathType is 1, then the decrypted stream must not be generally accessible. The system must provide a secure data path, aka “trusted video path” (TVP), for the stream. If 0 there is no such constraint. If the setting is not compatible with the security level of the stream, destroy the key and return OEMCrypto_ERROR_CONTENT_KEY_INVALID. If it is not possible to immediately detect a DataPathType and stream security level mismatch, the failure may be reported and the key destroyed on next decrypt call, before decryption.
8. HDCP -- If Observe_HDCP is 1, then apply the HDCP setting. Otherwise the HDCP setting must not be changed from its current value. Should be done in OEMCrypto_SelectKey.
9. CGMS -- If Observe_CGMS is 1, then apply the CGMS field if applicable on the device. Otherwise the CGMS settings must not be changed from their current value. Should be done in OEMCrypto_SelectKey.
   If Observe_CGMS is 0, and CGMS fields are non-zero, and the device is capable of implementing “CGMS Best Effort”, then it should implement “CGMS Best Effort”.
10. Duration field -- This legacy value has been replaced by the playback duration in the license's timer limits.

11. Make the decrypted content key from step 2 available for decryption of the media stream by DecryptCENC.
12. Return OEMCrypto_SUCCESS.

Backwards Compatibility

It is valid for a key control block to have an older verification field. For example, if the verification is “kc09”, then the key control block will have zero values in any field introduced after version 9 of this API. Since all new fields have had 0 chosen to represent a default or non-restricted value, the device does not need to handle different verification codes differently. As long as the verification code is valid, the key control block may be treated with the latest field definitions.

Replay Control -- Nonce and Provider Session Token (PST)

The nonce field of the Key Control Block is a 32 bit value that is generated in the trusted environment. The OEMCrypto implementation is responsible for detecting whether it has ever before received a message with the same nonce (a possible replay attack). The nonce algorithm is defined as follows:

1. Nonce generation: a new nonce is generated by the OEMCrypto implementation at the request of the client, when OEMCrypto_GenerateNonce() is called. The nonce is placed in the license request. The OEMCrypto implementation shall generate a 32-bit cryptographically secure random number each time it is called by the client and associate it with the session.
2. Nonce monitoring: the OEMCrypto implementation is responsible for checking the nonce in each call to OEMCrypto_LoadKeys(). When the function OEMCrypto_LoadLicense, OEMCrypto_LoadProvisioning, or OEMCrypto_LoadRenewal is called, the corresponding ODK function is used to verify the nonce.

The replay control flag and the nonce enabled flag determine if a license may be used only once, may be reloaded until released, or may be reloaded indefinitely. An online license may be loaded only once, and requires a valid nonce from the nonce cache. An online license may also require that a new entry in the usage table be created. An offline license that is unlimited does not require a nonce, or a pst. An offline license that can be released requires a valid nonce and a pst when it is first loaded. On subsequent loads, the nonce does not have to be valid, but the pst must be found in the usage table. This is summarized in the following table:

License Type	Replay_Control	Nonce_Enabled	PST required?
Unlimited Offline	0x0 - Session Usage table not required	0=Ignore Nonce	No. OEMCrypto ignores pst.
Invalid - server will not send.	0x1 - Nonce required, create entry in Session Usage table	0=Ignore Nonce	n/a

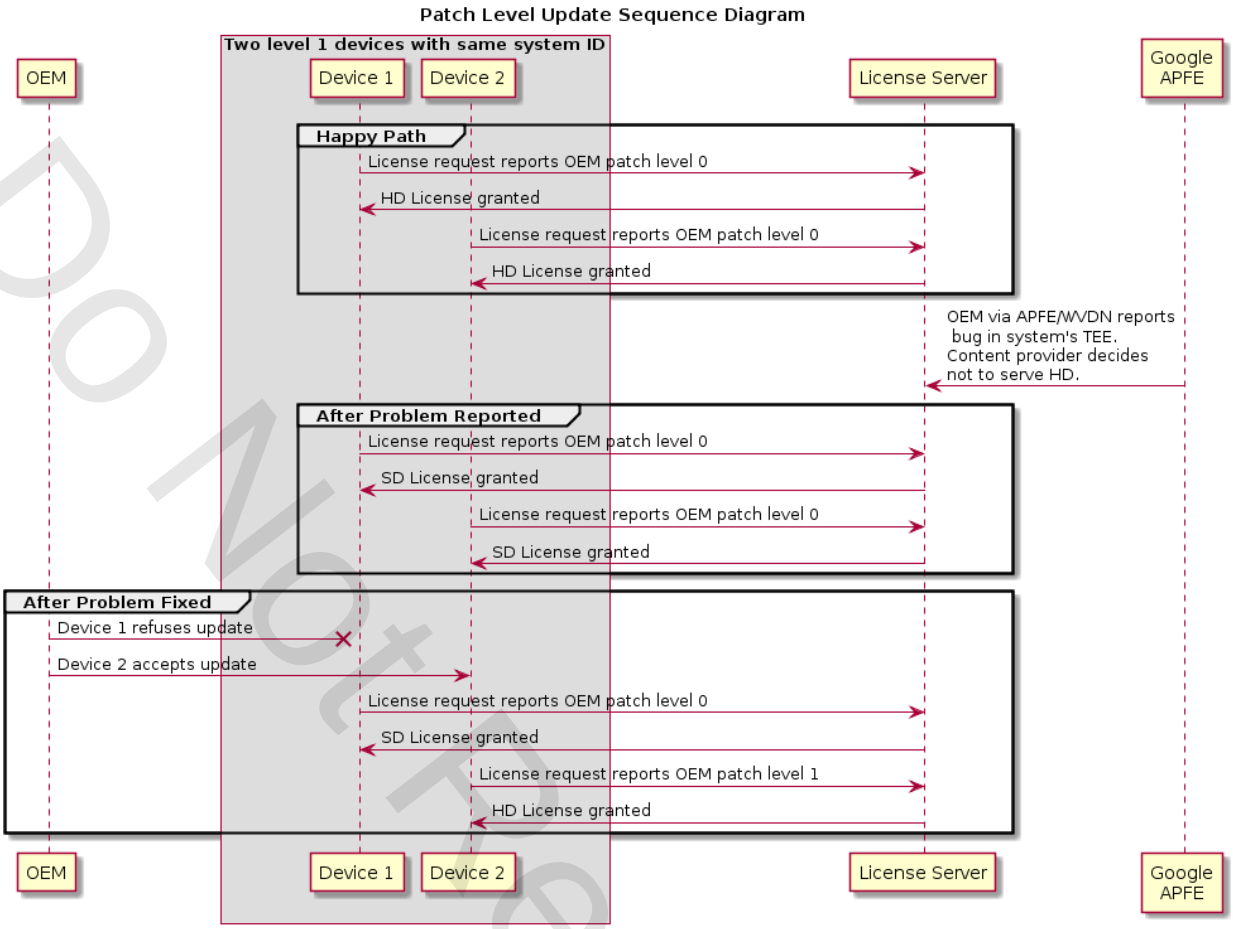
Offline	0x2 - Require existing Session Usage table entry or Nonce	0=Ignore Nonce. Nonce is verified on first load, and ignored subsequently.	Yes. OEMCrypto requires PST.
Streaming, no usage data required	0x0 - Session Usage table not required	1=Verify Nonce	No. OEMCrypto ignores pst.
Streaming, usage data required.	0x1 - Nonce required, create entry in Session Usage table	1=Verify Nonce	Yes. OEMCrypto requires PST.
Invalid - server will not send.	0x2 - Require existing Session Usage table entry or Nonce	1=Verify Nonce	n/a

Security Patch Level

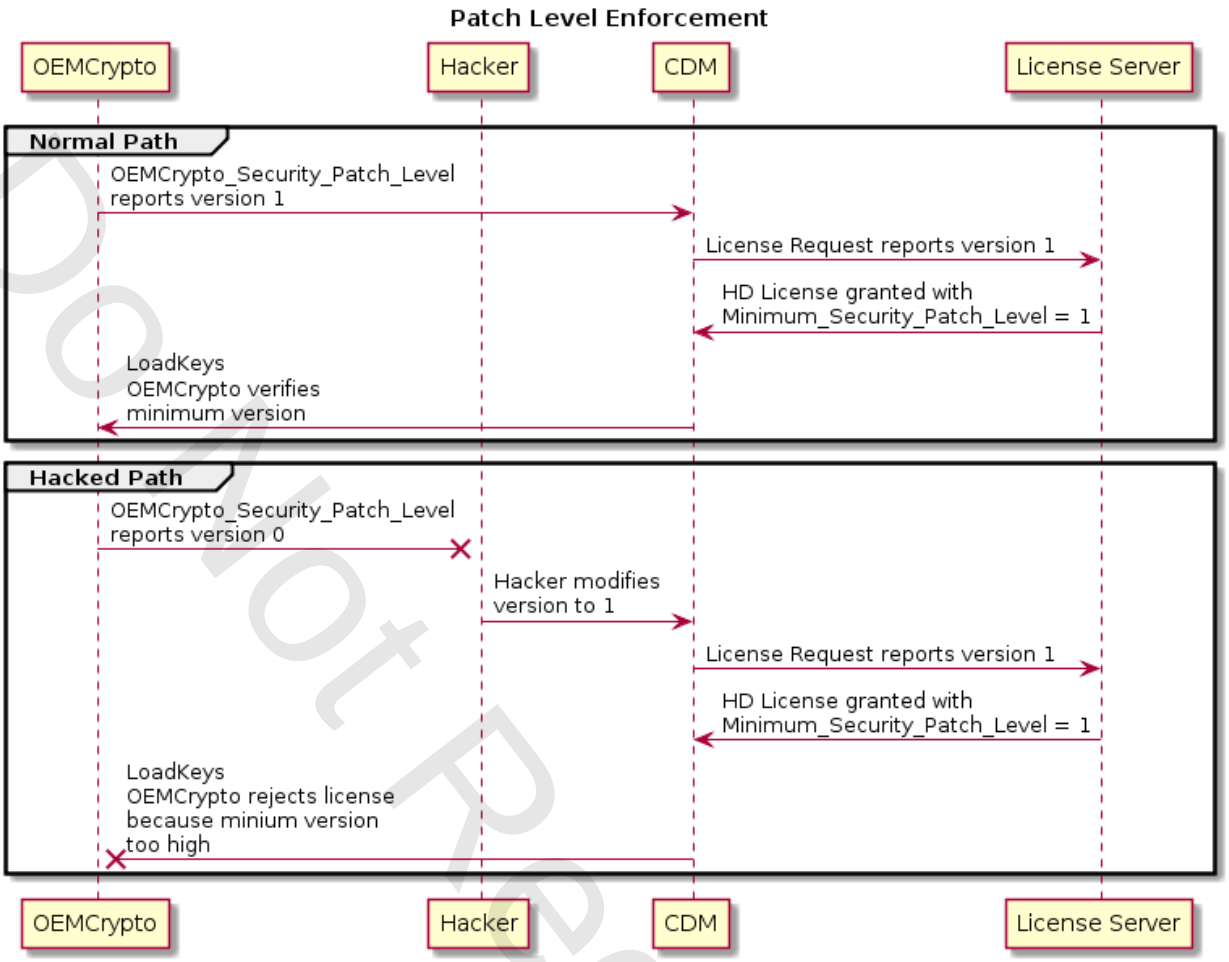
This feature addresses the desire of a content provider to serve licenses to a device only if it has a specific security patch. This feature allows the device to indicate that it has received a security patch. Notice that this feature will not distinguish between a device whose root of trust has been compromised and one that has not --- it is assumed that the root of trust is still uncompromised.
This feature will be implemented by assigning a patch level to the OEM software -- either OEMCrypto or any underlying components. Initially the patch level will be 0. The patch level would only be rolled when a security problem has been discovered, and there is a need to distinguish between devices in the field that have the new security patch from those that do not. Since this is expected to happen very rarely, the patch level will be 0 for most devices. The patch level is only used to distinguish between devices with the same Widevine system ID. Devices with different system IDs will not have their patch levels compared.

When the device sends a license request to the server, the current OEM patch level is included in the request. The server will decide which type of license to grant, and send the license response. When the function LoadKeys is called, the key control block will have the bits Minimum_Security_Patch_Level set to the patch level. If the minimum number is larger than the current patch level, the device should assume that there has been a man-in-the-middle attack, and reject the license.

Here is a top level sequence diagram showing two devices. One device is updated and the other is not.



Here is a sequence diagram showing how OEMCrypto should behave in the normal case, and in the case where there is a man-in-the-middle.



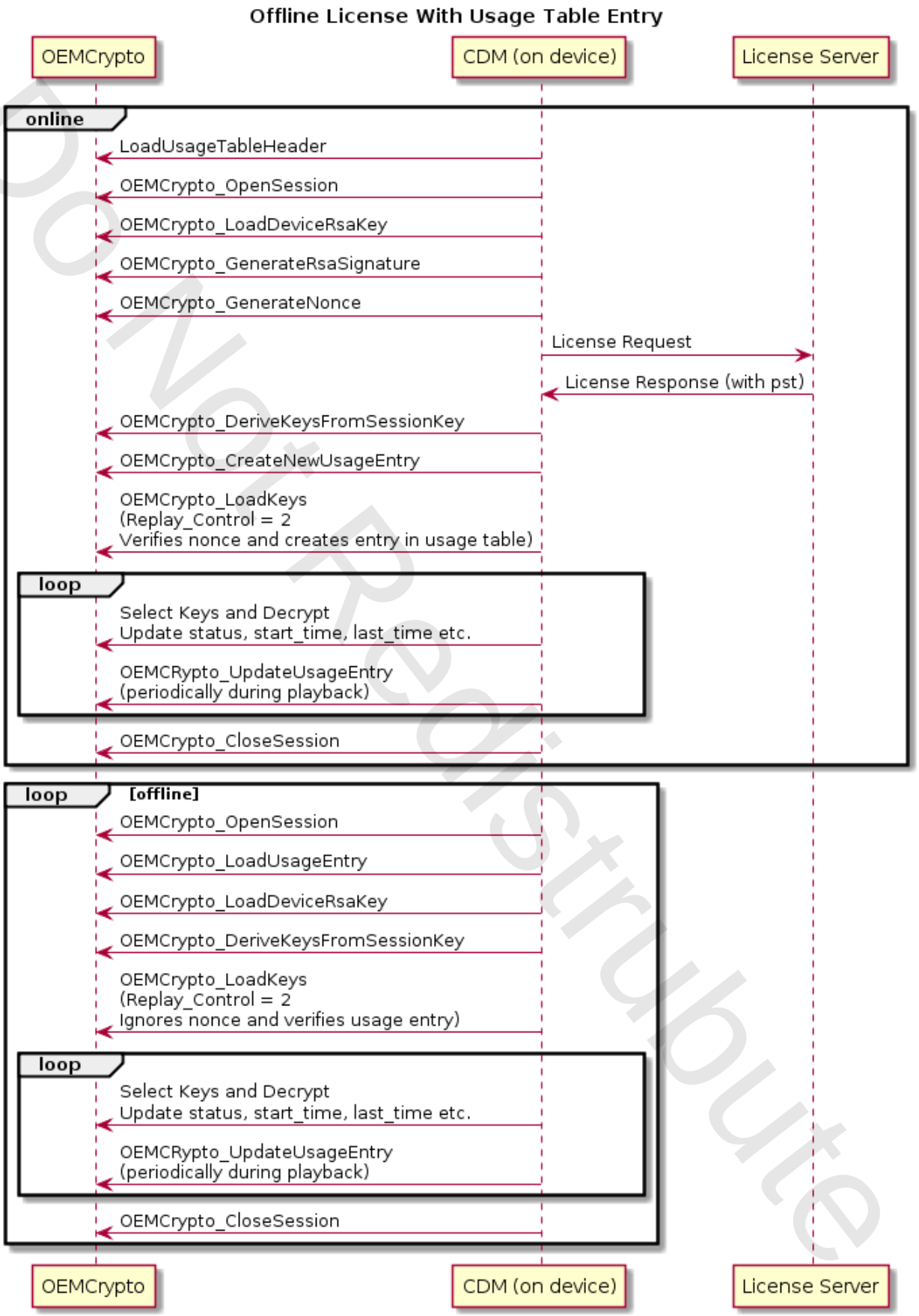
Session Usage Table and Reporting

The Session Usage Table is a feature that has two main use cases. It is used to control reloading keys for offline playback, and for reporting secure stops for online playback. Both of these use cases require a Session Usage Table that stores persistent data securely, and a secure clock or timer that cannot be rolled back by the user. In this section we define what we mean by a secure clock or timer, and describe the table. The API for reporting usage is described in the section Usage Table API, and in the function OEMCrypto_LoadKeys, and the decryption functions in the Decryption API.

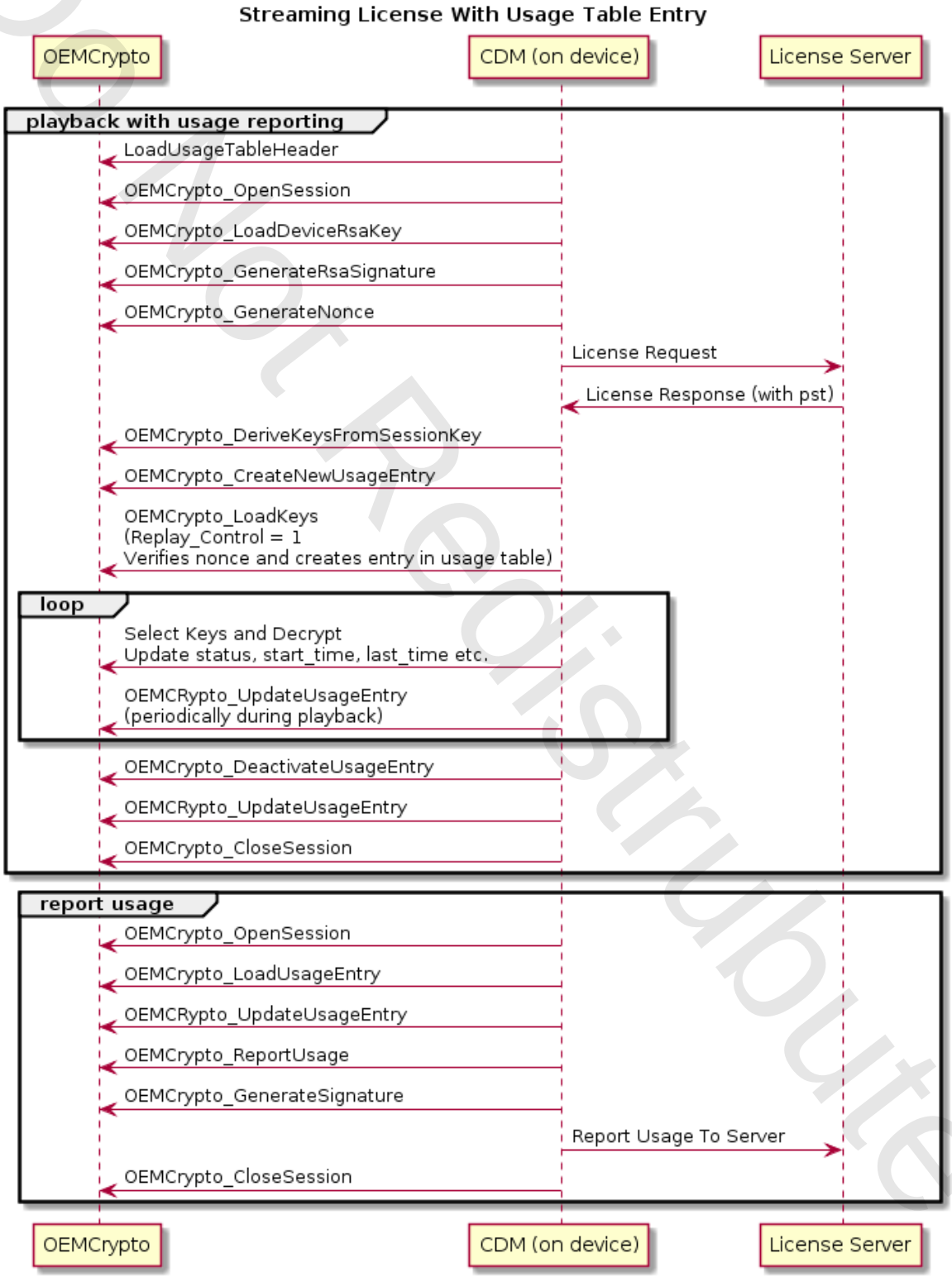
Keys that are intended for offline playback will need to be loaded several times, without access to a new license response. The API is designed so that the first time such a key is loaded, it must have a valid nonce matching the license request. The key will then be loaded into the usage table. For any subsequent calls to LoadKeys, the key will be verified with the usage table instead of using a nonce, and that session will be associated with the existing entry in the Usage Table.

The Usage Entry will be associated with a Provider Session Token (PST). A PST is associated with a session on the server, and its entry may persist after an OEMCrypto Session has been closed. Entries in the table may be created from a call to OEMCrypto_CreateNewUsageEntry, and will be deleted with a call to OEMCrypto_ShrinkUsageTable or when it is overwritten with a call to OEMCrypto_MoveEntry. The table contains session signing keys, so it must be encrypted or stored in secure memory to prevent inspection; the table will be used to report usage times, so it must not be user modifiable; and the session records license release times, so the user should not be able to rollback to a previous valid table. The table will be modified when LoadKeys is called or when any of the Usage Table API functions are called. In particular, during video playback, the table will be updated approximately once every minute.

Below is the sequence diagram for an offline license.



Keys that are designed for “secure stop” will be added to the usage table and will also require a nonce. After the application has finished using this key, the application will request that the entry in the table will be marked as inactive. After that, the key cannot be used for decryption, but usage times will still be available to send to the server for bookkeeping purposes. The sequence diagram for a streaming license with secure stop is below.



An entry in the Usage Table will store the start and stop times for when the key was used. With this in mind, the TEE will have a clock, which we define below in the description of OEMCrypto_ReportUsage. For all levels of secure clock, OEMCrypto shall force the clock to advance only. If the clock hits end-of-time and wraps back to 0, every entry in the usage table will be deleted and all keys will be deleted -- using 64 bits for seconds, this should only happen if the clock is being modified by a rogue application.

Each entry in the Session Usage table contains the following data:

```
 {
  uint8_t verification[8];  // must always be "USEENTRY"
  uint64_t time_of_license_signed;  //set when license is signed.
  uint64_t time_of_first_decrypt;  // set when first decrypt is called.
  uint64_t time_of_last_decrypt;  // updated after any decrypt when usage entry
updated.
  uint64_t generation_number;
  uint32_t index;  // index in header's array of generation numbers.
  enum USAGE_ENTRY_STATUS status;
  uint8_t  server_mac_key[MAC_KEY_SIZE];
  uint8_t  client_mac_key[MAC_KEY_SIZE];
  size_t pst_length;
  uint8_t pst[MAX_PST_SIZE];
}
```

Entries will be created and associated with an open session, and stored in protected memory by OEMCrypto. In order to persist the entry, the CDM layer will ask OEMCrypto for an updated entry. OEMCrypto will encrypt and sign the entry and pass it back to the CDM layer. The CDM layer will be responsible for saving the data to the file system or similar persistent memory. After the session has been closed, all memory used by OEMCrypto for that usage entry may be released.

Before version 16 of this API, time_of_license_received was stored instead of time_of_license_signed. For any practical bookkeeping purposes, these events are essentially at the same time.

OEMCrypto will also maintain a list of all existing Usage Entries -- those that are currently in memory associated with an open session, and those that have been saved to the file system. This structure is called the Usage Table Header:

```
 {
  uint8_t verification[8];  // must always be "UTHEADER"
  uint64_t master_generation_number;
  uint32_t entry_count;
  uint64_t entry_generation_number[variable size];  // matches entry's gn.
}
```

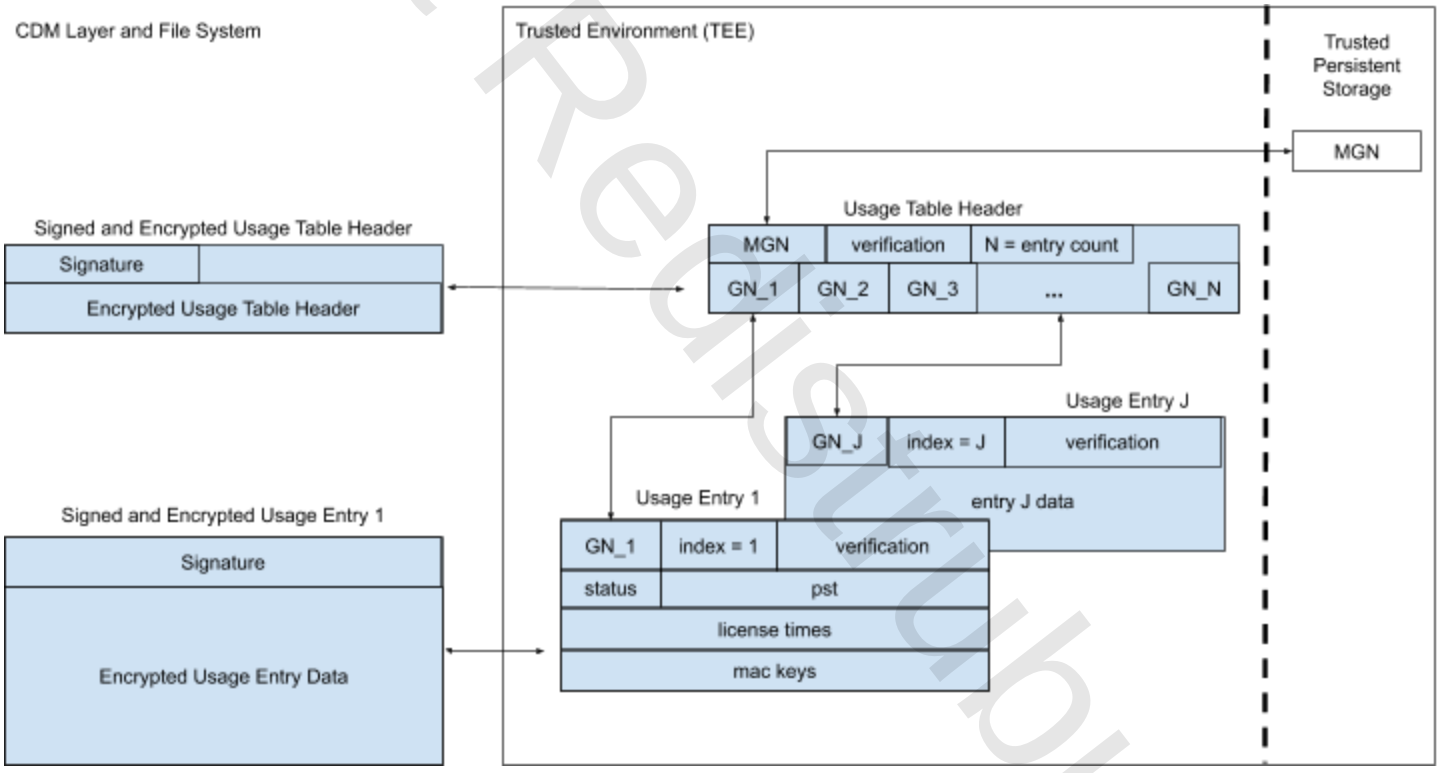
As with each usage entry, the header will be stored in protected memory by OEMCrypto. In order to persist the header, the CDM layer will ask OEMCrypto for an updated header. OEMCrypto will encrypt and sign the header and pass it back to the CDM layer. The CDM layer will be responsible for saving the data to the file system or similar persistent memory. After every session has been closed, and OEMCrypto_Terminate is called, all memory used by

OEMCrypto for the usage header may be released.

Since the usage table is used to report to the server that a license has been released and marked as inactive, OEMCrypto must prevent rollback of the data. In order to do this, OEMCrypto will mark each entry in the usage table with a generation number. This number should be the same as the entry's generation number in the usage table header. The usage table header has an array of generation numbers -- one that matches each entry, and it has a master generation number. The master generation number is also stored in secure persistent storage by OEMCrypto. Whenever a usage entry is updated, its generation number is incremented, and the master generation number is incremented, and both entry and header are encrypted and signed and saved to insecure storage. Rollback of the whole table is prevented by having OEMCrypto prevent rollback of the master generation number.

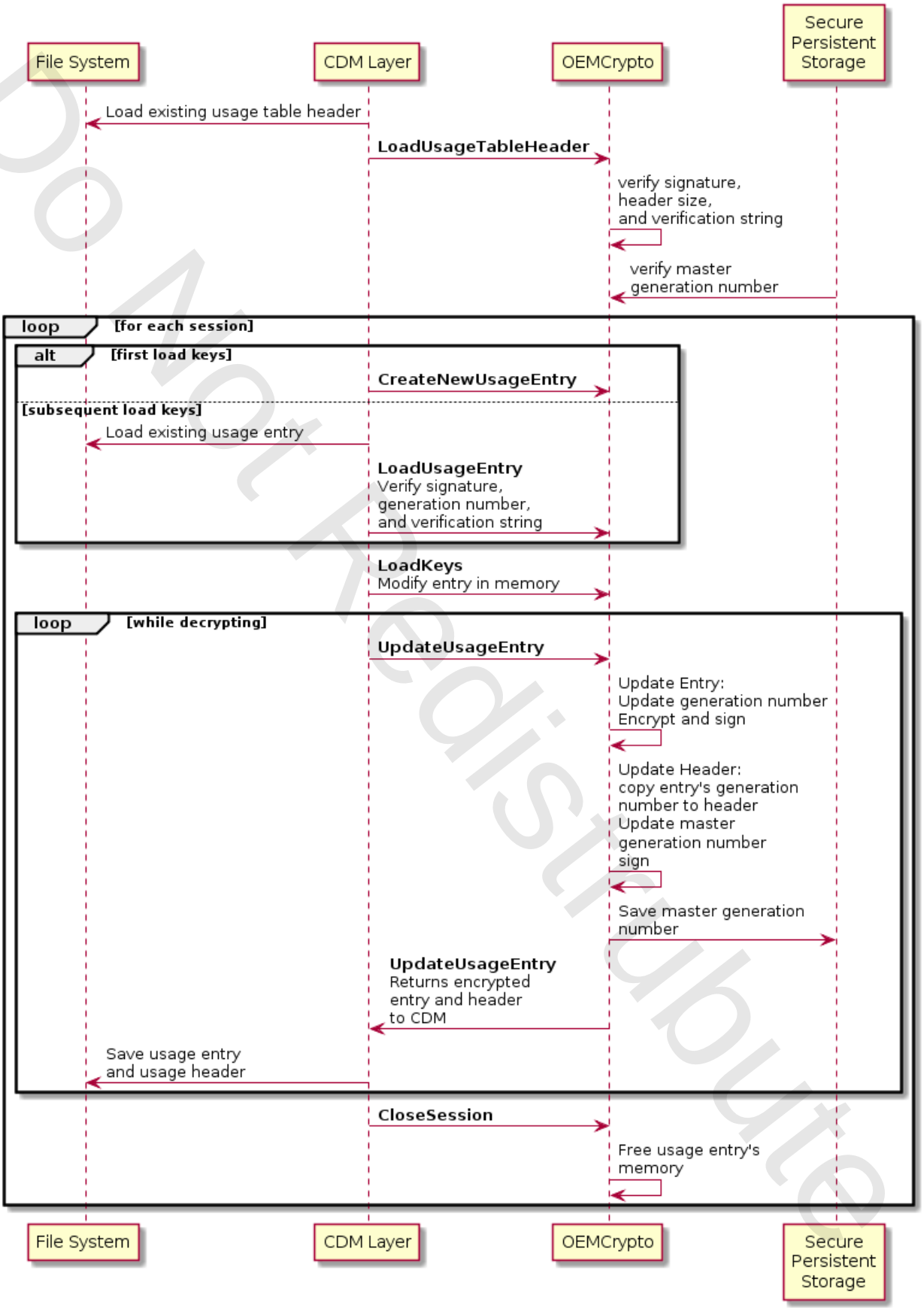
To allow for accidental system crashes, the system can allow for the table to be rolled back by one generation number. However, more than one generation will trigger an error and invalidate the table. When the table is invalidated, all entries will be considered invalid.

Here is a diagram showing that encrypted data is stored on the file system and that part of the table will be resident in the secure memory of the TEE.



Below is a sequence diagram showing the flow of data when saving a usage entry and the usage table header.

Usage Tables



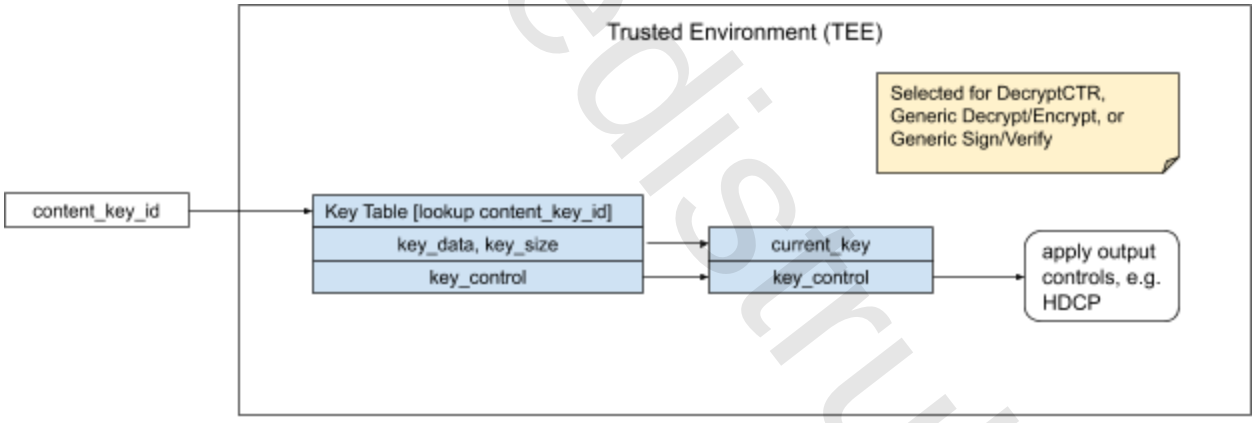
Entries in the table may have the following status values:

```
enum OEMCrypto_Usage_Entry_Status {
        kUnused = 0, // decrypt not yet called
        kActive = 1, // keys not released
        kInactiveUsed = 3, // keys released after use.
        kInactiveUnused = 4, // keys released before use.
};
```

New entries will have a status of kUnused. On the first decrypt call for a session, the status is changed to kActive. When a license is no longer needed, the method OEMCrypto_DeactivateUsageEntry is called to change the state to either kInactiveUused or kInactiveUsed. Once a session's entry has been marked "inactive", the keys in that session may no longer be used to decrypt or encrypt data. The entry will be kept until a usage report has been sent to the server and an acknowledgement has returned. The entry may still be loaded into a session, but the session may not be used to decrypt content -- that session will only be used to generate a usage report. The usage report is used to securely confirm to the license server that a license is no longer active.

Content Decryption

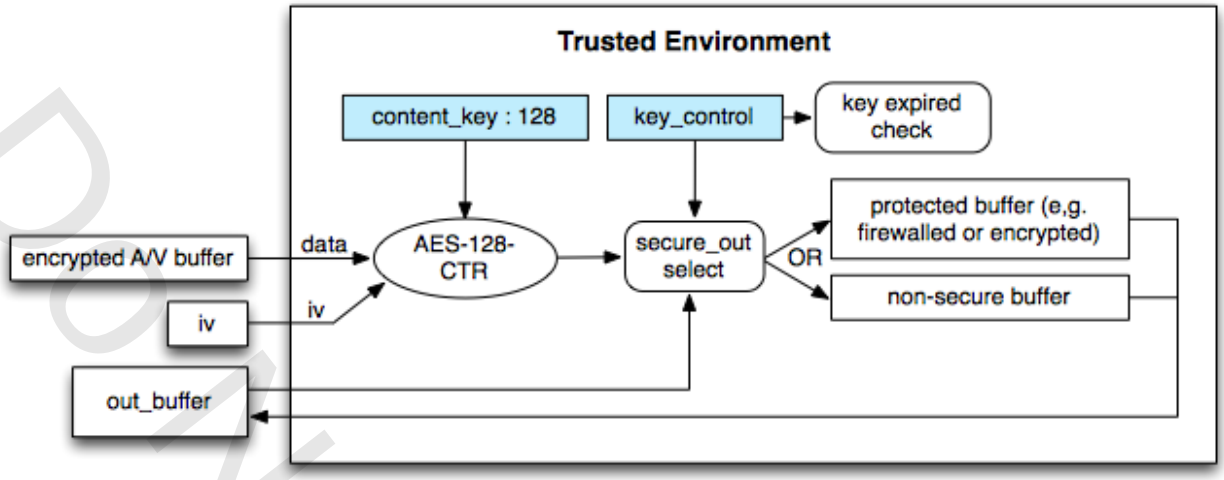
OEMCrypto_SelectKey() is used to prepare one of the previously loaded keys for decryption.



OEMCrypto_SelectKey()

For an entitlement license, If the device uses a hardware key ladder, it may be more convenient to store the encrypted content key in the key table. If that is the case, then SelectKey will first latch in the entitlement key and decrypt the content key. Then it will latch in the content key.

Once the content_key is loaded, OEMCrypto_DecryptCENC is used to decrypt content. *enc_key* encrypts *content_key* using AES-128-CBC with random IV. *content_key* encrypts *content* using AES-128-CTR or AES-128-CBC with random IV.



OEMCrypto_DecryptCTR()

Generic Crypto

OEMCrypto may also be used to encrypt, decrypt, sign or verify generic application data. This may be used by an application to deal with business data instead of just protected media. Keys for generic crypto operations are loaded and selected as for media keys, described above.

OEMCrypto may not use a content key for generic operations unless permission is given in the key control block. The flags Allow_Encrypt, Allow_Decrypt, Allow_Sign and Allow_Verify must be set in a key's key control block in order for the key to be used in the function OEMCrypto_Generic_Encrypt, OEMCrypto_Generic_Decrypt, OEMCrypto_Generic_Sign, and OEMCrypto_Generic_Verify respectively.

HDCP SRM Update

Some content providers have requested that Widevine deliver the HDCP SRM (System Renewability Message). This is a small file, currently less than 5kB, that contains lists of Key Selection Vectors (i.e. key IDs) that should not be negotiated for HDCP. The device is supposed to validate the signature on the SRM, store the SRM in non-volatile memory and use it during authentication to decide if a downstream device is allowed to receive content, as required by the HDCP specification.

Devices that support HDCP v2.2 or higher, and expect to display 4k content, should implement the SRM update function, OEMCrypto_LoadSRM.

According to the HDCP specification, the SRM is signed by the DCP private key, and must be verified by the device. Each SRM has a version number, and the device must not install a less recent version of the file. This makes testing this feature problematic. With that in mind, the

SRM update functions will only be superficially tested by the standard suite of unit tests. See the discussion about the function RemoveSRM below for more information.

In addition to loading the SRM, some keys will have the flag **MinimumSRMVersion** set, and the parameter srm_restriction_data will be passed into OEMCrypto_LoadLicense. The SRM restriction data will tell the device what the required minimum SRM version number is.

Be aware that some content providers wish to require HDCP but do not wish to require a minimum SRM. The key control block flags SRMVersionRequired may be set or may be unset for various values of HDCP_Version. If SRMVersionRequired is not set, then the device should NOT enforce the SRM blacklist. This can be used to bypass a compromised SRM that has been installed on a device by a rogue entity at the discretion of the content provider.

In order to test this functionality, it will be necessary to install a new SRM file. In order to run several tests, or to run the test several times, the device will need to delete the SRM file. This functionality should **not** be available on production devices. Widevine will create a brief set of unit tests which will use this function. The OEM will need to take extra care verifying this feature, because there will be no automated tests.

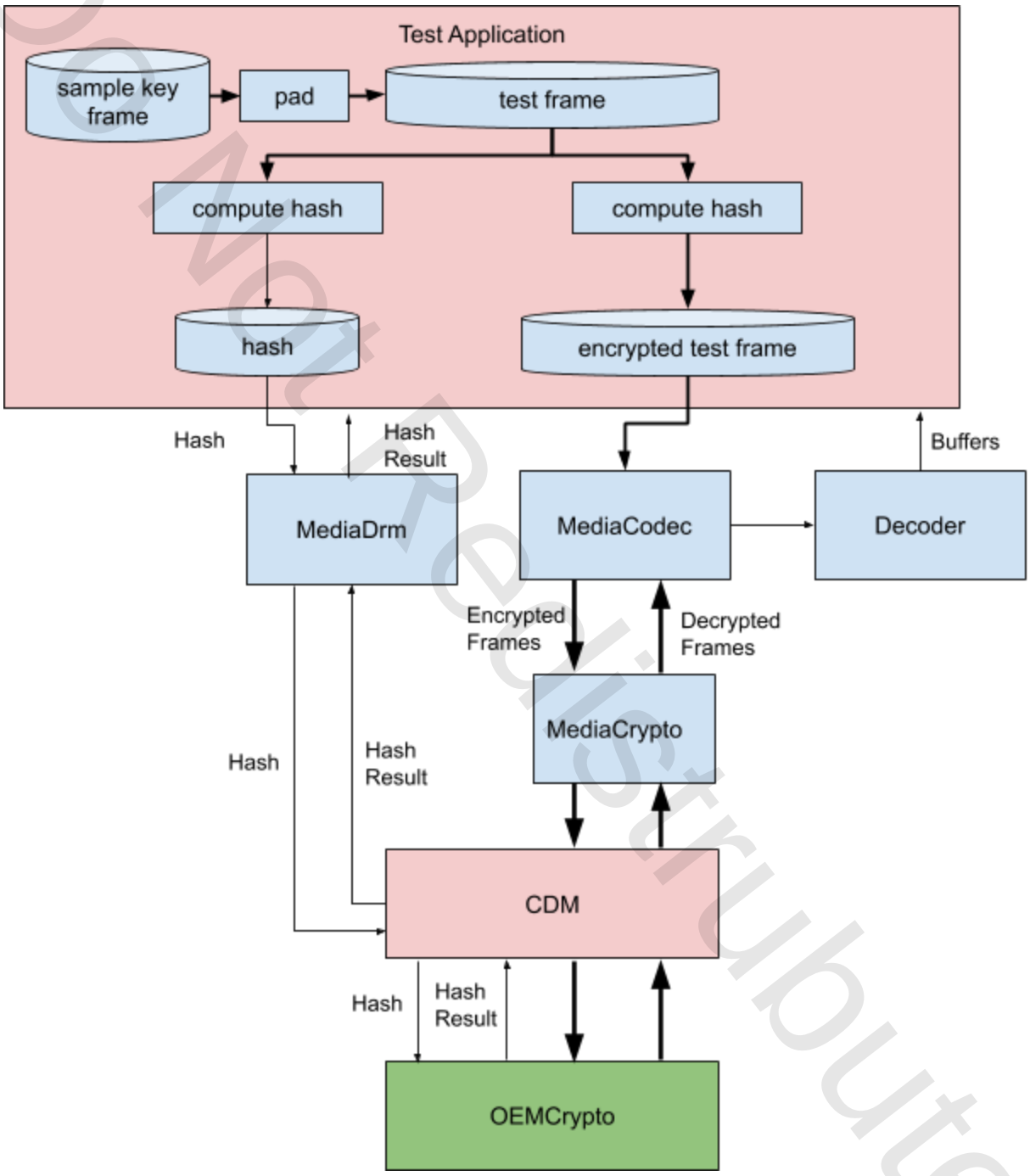
Full Decrypt Path Testing

In order to verify that the full decryption path works for secure buffers with the various pattern decryption standards, some new API functions will be added to verify that the frame to be displayed is correct. While testing the full decrypt path, the keys would be installed as usual. Then, before each frame is decrypted the function OEMCrypto_SetDecryptHash will be called. This sets a hash of next frame. Then the function OEMCrypto_GetHashErrorCode will be called. If the hash of any frame does not match the hash set by the test application, this will return an error code.

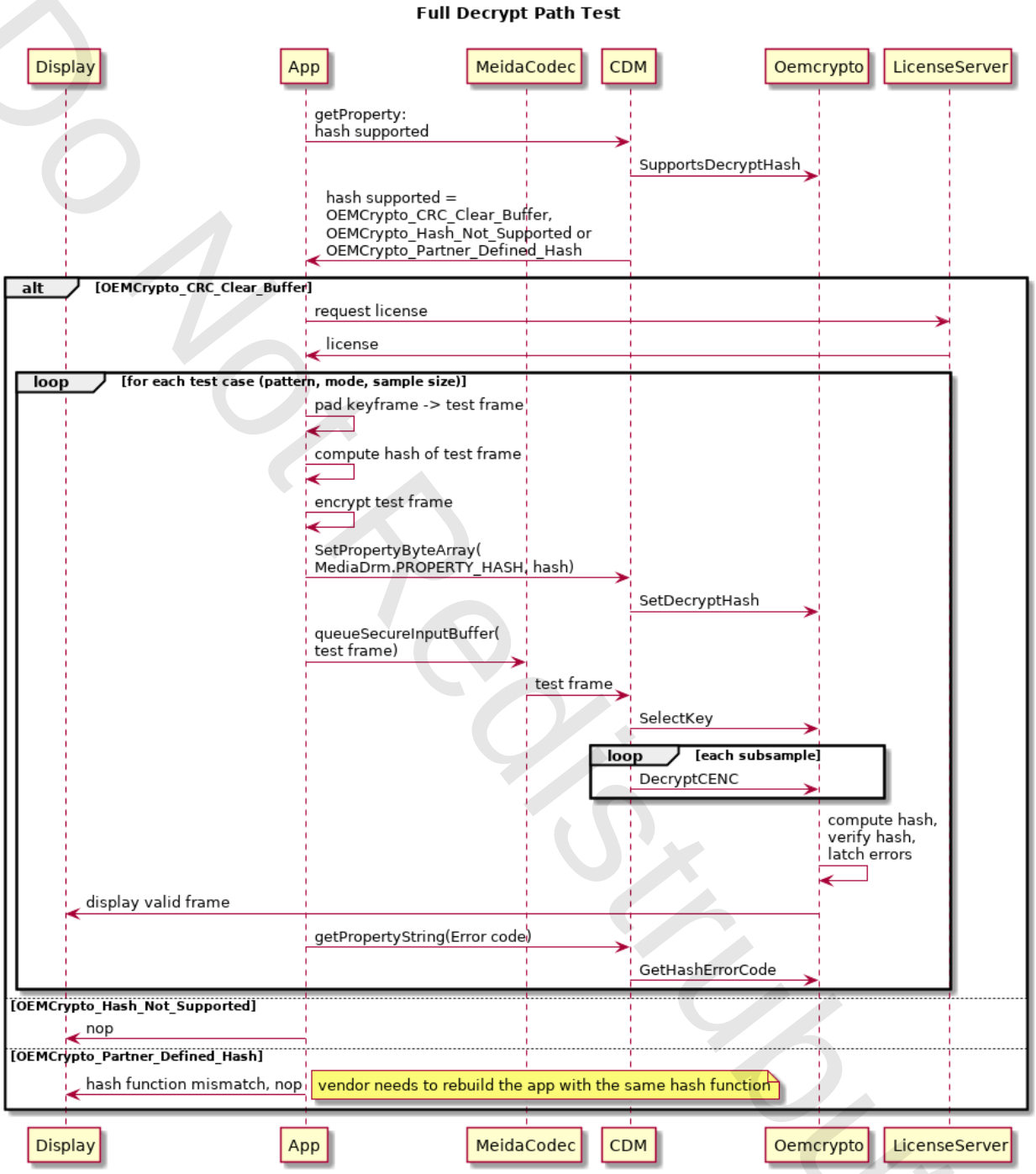
Hashes will only be verified if the key has the Allow_Hash_Verification bit set in the key control block. This bit will only be set on test content.

The main reason to do this is so that the contents of a secure buffer can be verified. Many chip makers have been using a different code path for DecryptCENC when the output buffer is secure than when the output buffer is not secure. This new feature will be used to verify that decryption is working correctly for secure buffers using real video content. This will make it much easier for you to verify that your version of OEMCrypto is working correctly in the future.

Below is a diagram illustrating the data flow for the Android platform. Partners using the source CE CDM release will need to ensure that each frame corresponds to one sample, and that both clear and encrypted subsamples are passed to the CDM layer. Also, partners will need to help write the test application, because each platform handles playback differently.



Below is a sequence diagram for Android. Again, a CE CDM platform would need to coordinate with a Widevine engineer to ensure that the test application sets the hash file correctly. Also, a CE CDM platform needs to ensure each sample is exactly one frame.

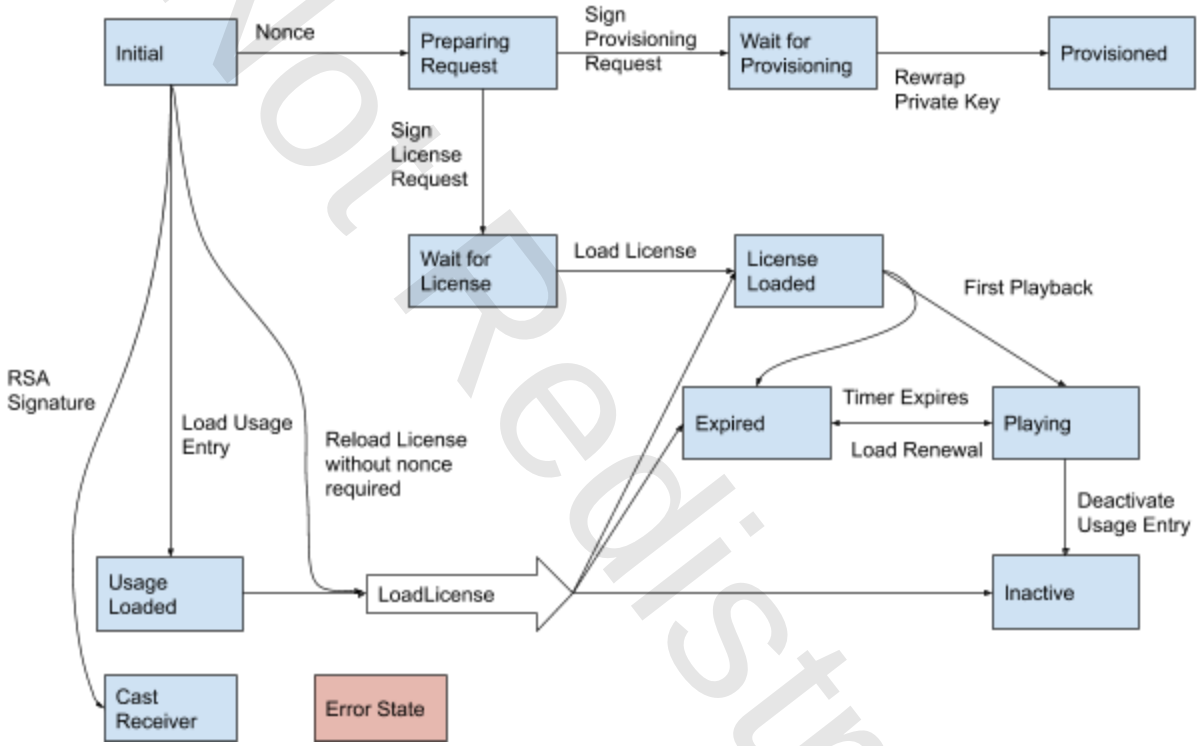


OEMCrypto State Model

To understand proper usage of OEMCrypto functions, it is helpful to think of an OEMCrypto session as a state machine. OEMCrypto implementers are not required to use a state machine. For v16, Widevine does have the following requirements.

1. Only one of OEMCrypto_LoadLicense or OEMCrypto_ReloadLicense can be called in each session, and that function shall only be called once in each session.
2. Only one of OEMCrypto_LoadUsageEntry and OEMCrypto_CreateUsageEntry can be called in each session, and that function sahll only be called once in each session.
3. OEMCrypto_GenerateNonce can only be called once in each session, and it must be called before signing either a provisioning request or a license request.

The diagram below shows the possible transitions among each state:



More details can be found in the document “OEMCrypto State Diagram”.

Threading Model Clarification

This section is a clarification and rewording, and is not intended to be new information.

Applications using the CDM and OEMCrypto may be multithreaded. On some OSes, there may be several applications that are active at the same time. The OEMCrypto threading model is a list of assumptions that OEMCrypto may make about which functions may be called simultaneously with which other functions.

Even though we make guarantees to partners about how the CDM layer will and won’t call OEMCrypto as regards threading, partners cannot rely on only being called in that way for

security, as malicious actors could call OEMCrypto in other ways. At best, our Threading Guarantees say “If we call you in this manner, you should not have errors because of threading.” OEMCrypto must not leak or become compromised if they are called outside those guarantees. The guarantees merely permit them to fail to work correctly when called outside those bounds.

To specify these rules, this document uses the terms “read lock” and “write lock”. When a thread holds a “write lock” on a mutex, no other thread may hold a lock for reading or writing on the same mutex. When a thread holds a “read lock” on a mutex, no other thread may hold a lock for writing on that mutex, but other threads may simultaneously hold a lock for reading. The rules below are for the CDM layer. OEMCrypto may assume that the CDM layer holds the appropriate read or write lock before calling the OEMCrypto function.

We will use a model that assumes there is one mutex for the entire OEMCrypto system. The system mutex will have both read and write locks on it. All functions will hold at least a read lock on the system. Also, each session will have its own mutex. Each of the session mutexes will have write locks on it.

There are four classes of functions for threading, which are described below.

Initialization and Termination Functions

Initialization and termination functions are called sequentially, as if they hold a write lock on the OEMCrypto system. These functions include

- OEMCrypto_SetSandbox
- OEMCrypto_Initialize
- OEMCrypto_InstallKeyboxOrOEMCert
- OEMCrypto_LoadTestKeybox
- OEMCrypto_Terminate

All other functions will be called after OEMCrypto_Initialize and before OEMCrypto_Terminate.

Property Functions

Property functions and functions that do not modify the system are used to gather information about the system. OEMCrypto may assume these are called between the initialization and termination functions, as if the CDM holds a read lock on the OEMCrypto system.

These functions include OEMCrypto_GetKeyData, OEMCrypto_GetRandom, OEMCrypto_APIVersion, and almost all functions that do not need a session or the usage table.

Session Initialization and Usage Table Functions

Session initialization functions are OEMCrypto_OpenSession and OEMCrypto_CloseSession. Usage table functions are functions that modify the usage table header or update the master generation number. This category includes the functions to create or load the usage table header, create or load a usage table entry, update or deactivate a usage entry, generate a usage report, move usage entries, or shrink the usage table header. These functions will not be called at the same time as any other functions, as if the CDM holds a write lock on the

OEMCrypto system.

This category does **not** include functions that only modify data within a usage entry, such as the decryption functions, like OEMCrypt_DecryptCENC, because that does not update the generation number of the entry or the usage header.

Session Functions

Session functions are all of the functions that take a session as a parameter. This category includes OEMCrypto_GenerateNonce, the derive key functions, the key-loading functions, and the decryption functions.

These functions can be called simultaneously with any property function, or simultaneously with any session function from another session. They will not be called simultaneously with any session function for the same session.

These functions may behave as if the CDM layer has at least a read lock on the OEMCrypto system, and a write lock on the individual session they target.

VM and Sandbox Support

Although the CDM is usually running on a set top box or a mobile device where there is only one process that interacts with OEMCrypto, there are situations, like on an in-flight entertainment system or on a desktop computer supporting multiple browsers, where several processes or virtual machines (VM) will interact with OEMCrypto. On these devices, each CDM has its own file system where it stores DRM certificates, offline licenses, and usage table data. We will use the term “sandbox” to refer to the file system and to the process or VM in which the CDM instance is running. The CDM layer has no knowledge of other running sandboxes. If OEMCrypto is interacting with several sandboxes, then we will assume that it has a way to distinguish between different sandboxes. For example, OEMCrypto could look at the process id (PID) of the CDM or it could look at the VM id.

Device manufacturers that wish to use a sandbox model must coordinate with the provider of OEMCrypto to make sure that OEMCrypto can operate properly within a sandbox. **This feature is a special case, and is not commonly supported**.

The CDM can only guarantee the threading rules specified in this document for threads within the same sandbox. For example, there is no guarantee that the CDM layer from different sandboxes will not call OpenSession at the same time.

Each CDM instance will call OEMCrypto_SetSandbox and OEMCrypto_Initialize once before any other calls, and will call OEMCrypto_Terminate after use. Neither of these shall interfere with other sandboxes. In particular, OEMCrypto_Terminate shall not close any sessions which were opened in another sandbox. This expectation only holds for OEMCrypto implementations that are designed to work with a sandbox. As mentioned above: if you are a device maker, and wish to use sandbox support for your device, you must ensure that your OEMCrypto provider supports sandboxes. All other providers of OEMCrypto will assume that there is a single CDM instance.

Another issue that may arise when using multiple sandboxes is the uniqueness of the

generation number in the usage table header. Because each CDM instance will have a separate file system, each CDM will have its own usage table header, and usage entries in the table. In order to distinguish among the different headers, the CDM will specify a sandbox ID. This sandbox ID is a string of bytes that uniquely identifies this CDM instance as belonging to a specific sandbox. This allows OEMCrypto to recreate the map from sandbox to the sandbox's persistent data when the sandbox's process or VM is shutdown and started in a new process or VM. The sandbox ID will be sent to OEMCrypto just before OEMCrypto_Initialize is called.

Device manufacturers and OEMCrypto providers who wish to support this new feature are strongly encouraged to coordinate integration testing with each other, and with a Widevine engineer.

Optional Features

Because the Widevine Modular DRM software is shared on a variety of platforms, some of the APIs described below are not needed on all platforms. This section describes what functionality will be missing if certain feature sets are not implemented.

On some platforms, such as Android, there is a strict list of features that must be supported in order to be certified. Please see the supplement to this document for your platform if there are any doubts.

The unit tests in oemcrypto_test.cpp are designed so that these features are not tested if they are not implemented. In general, if a feature is not implemented, then the OEMCrypto library should return OEMCrypto_ERROR_NOT_IMPLEMENTED for those functions.

"Not Very Optional" Optional Features

These features are used by many content providers and are required for L1 devices except in special cases.

Load Certificate Functionality - Some devices have a hard-coded DRM certificate. This is only allowed for Level 3 devices. All other devices must support RSA. Required for L1.

Provisioning 2.0 or Provisioning 3.0: Provisioning 2.0 means using a keybox as a root of trust. Provisioning 3.0 means using an OEM Cert as a root of trust. Widevine requires one of these for L1, but they are mutually exclusive.

Generic Crypto - The generic crypto functionality is used by some applications to encrypt user account information and business logic. It is required on Android.

Usage Tables - This is required for persistent licenses and release or secure stop. These features are used by most content providers. OEMCrypto must store the usage table generation number in secure memory to satisfy L1 robustness rules. Required on Android.

Entitlement Keys - Used for grouping licenses and for key rotation. Used by YouTube for key rotation. Used by ATSC for grouped licenses. Required for all devices.

"Very Optional" Optional Features

These features are only used in special cases or for debugging and are not needed on most production devices.

HDCP SRM Updates - SRM Updates are used to install a new HDCP device revocation list. This is not required by any content providers at this time. Some studios are requesting it.

Full Decrypt Path Testing - This is recommended for testing only. Not required for any production device.

Sandbox Support - Only used on specialized devices, such as Virtual Machines or desktop applications. Not required by any content providers. Several in-vehicle entertainment systems are planning to use this to distinguish between separate seats in a car.

OEMCrypto API for CENC

The OEMCrypto API is defined in the file OEMCryptoCENC.h. The description of each function is below.

Crypto Device Control API

The Crypto Device Control API involves initialization of and mode control for the security hardware.

OEMCrypto_SetSandbox

```
OEMCryptoResult OEMCrypto_SetSandbox(const uint8_t* sandbox_id,
                                     size_t sandbox_id_length);
```

This tells OEMCrypto which sandbox the current process belongs to. Any persistent memory used to store the generation number should be associated with this sandbox id. OEMCrypto can assume that this sandbox will be tied to the current process or VM until OEMCrypto_Terminate is called. See the section "VM and Sandbox Support" above for more details.

If OEMCrypto does not support sandboxes, it will return OEMCrypto_ERROR_NOT_IMPLEMENTED. On most platforms, this function will just return OEMCrypto_ERROR_NOT_IMPLEMENTED. If OEMCrypto supports sandboxes, this function returns OEMCrypto_SUCCESS on success, and OEMCrypto_ERROR_UNKNOWN_FAILURE on failure.

The CDM layer will call OEMCrypto_SetSandbox once before OEMCrypto_Initialize. After this function is called and returns success, it will be OEMCrypto's responsibility to keep calls to usage table functions separate, and to accept a call to OEMCrypto_Terminate for each sandbox.

Parameters

[in] sandbox_id: a short string unique to the current sandbox.

[in] sandbox_id_length: length of sandbox_id.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INIT_FAILED failed to initialize crypto hardware

OEMCrypto_ERROR_NOT_IMPLEMENTED - sandbox functionality not supported

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system. It is called once before OEMCrypto_Initialize.

Version

This method is new in version 15 of the API.

OEMCrypto_Initialize

```
OEMCryptoResult OEMCrypto_Initialize(void);
```

Initialize the crypto firmware/hardware.

Parameters

None

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INIT_FAILED failed to initialize crypto hardware

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method is supported by all API versions.

OEMCrypto_Terminate

```
OEMCryptoResult OEMCrypto_Terminate(void);
```

Closes the crypto operation and releases all related resources.

Parameters

None

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_TERMINATE_FAILED failed to de-initialize crypto hardware

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system. No other functions will be called before the system is re-initialized.

Version

This method is supported by all API versions.

Crypto Key Ladder API

The crypto key ladder is a mechanism for staging crypto keys for use by the hardware crypto engine. Keys are always encrypted for transmission. Before a key can be used, it must be decrypted (typically using the top key in the key ladder) and then added to the key ladder for upcoming decryption operations. The Crypto Key Ladder API requires the device to provide hardware support for AES-128 CTR and CBC modes and prevent clear keys from being exposed to the insecure OS.

OEMCrypto_OpenSession

```
OEMCryptoResult OEMCrypto_OpenSession(OEMCrypto_SESSION *session);
```

Open a new crypto security engine context. The security engine hardware and firmware shall acquire resources that are needed to support the session, and return a session handle that identifies that session in future calls.

This function shall call ODK_InitializeSessionValues to initialize the session’s clock values, timer values, and nonce values. ODK_InitializeSessionValues is described in the document “License Duration and Renewal”, to initialize the sessions clock values.

Parameters

[out] session: an opaque handle that the crypto firmware uses to identify the session.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_TOO_MANY_SESSIONS failed because too many sessions are open

OEMCrypto_ERROR_OPEN_SESSION_FAILED there is a resource issue or the security engine is not properly initialized.

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Initialization Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_CloseSession

```
OEMCryptoResult OEMCrypto_CloseSession(OEMCrypto_SESSION session);
```

Closes the crypto security engine session and frees any associated resources. If this session is associated with a Usage Entry, all resident memory associated with it will be freed. It is the CDM layer’s responsibility to call OEMCrypto_UpdateUsageEntry before closing the session.

Parameters

[in] session: handle for the session to be closed.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION no open session with that id.

OEMCrypto_ERROR_CLOSE_SESSION_FAILED illegal/unrecognized handle or the security engine is not properly initialized.

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Initialization Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_GenerateDerivedKeys

```
OEMCryptoResult OEMCrypto_GenerateDerivedKeys(OEMCrypto_SESSION session,
                            const OEMCrypto_SharedMemory *mac_key_context,
                            size_t mac_key_context_length,
                            const OEMCrypto_SharedMemory *enc_key_context,
                            size_t enc_key_context_length);
```

Generates three secondary keys, mac_key[server], mac_key[client], and encrypt_key, for handling signing and content key decryption under the license server protocol for CENC.

Refer to the Key Derivation section above for more details. This function computes the AES-128-CMAC of the enc_key_context and stores it in secure memory as the encrypt_key. It then computes four cycles of AES-128-CMAC of the mac_key_context and stores it in the mac_keys -- the first two cycles generate the mac_key[server] and the second two cycles generate the mac_key[client]. These two keys will be stored until the next call to

OEMCrypto_LoadKeys(). The device key from the keybox is used as the key for the AES-128-CMAC.

Parameters

[in] session: handle for the session to be used.

[in] mac_key_context: pointer to memory containing context data for computing the HMAC generation key.

[in] mac_key_context_length: length of the HMAC key context data, in bytes.

[in] enc_key_context: pointer to memory containing context data for computing the encryption key.

[in] enc_key_context_length: length of the encryption key context data, in bytes.

Results

mac_key[server]: the 256 bit mac key is generated and stored in secure memory.

mac_key[client]: the 256 bit mac key is generated and stored in secure memory.

enc_key: the 128 bit encryption key is generated and stored in secure memory.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_NO_DEVICE_KEY

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support mac_key_context and enc_key_context sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffers are too large.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 12.

OEMCrypto_DeriveKeysFromSessionKey

```
OEMCryptoResult OEMCrypto_DeriveKeysFromSessionKey(
                    OEMCrypto_SESSION session,
                    const uint8_t* derivation_key,
                    size_t derivation_key_length,
                    const OEMCrypto_SharedMemory *mac_key_context,
                    size_t mac_key_context_length,
                    const OEMCrypto_SharedMemory *enc_key_context,
                    size_t enc_key_context_length);
```

Generates three secondary keys, mac_key[server], mac_key[client] and encrypt_key, for handling signing and content key decryption under the license server protocol for CENC.

This function is similar to OEMCrypto_GenerateDerivedKeys, except that it uses a session key to generate the secondary keys instead of the Widevine Keybox device key. These three keys will be stored in secure memory until the next call to LoadLicense or LoadProvisioning.

If the session's private key is an RSA key, then the session key is passed in encrypted by the device RSA public key as the derivation_key, and must be decrypted with the RSA private key before use.

If the sesion's private key is an ECC key, then the session key is the SHA256 of the shared secret key calculated by ECDH between the device's ECC private key and the derivation_key. See the document "OEMCrypto Elliptic Curve Support" for details.

Once the enc_key and mac_keys have been generated, all calls to LoadKeys or LoadLicense proceed in the same manner for license requests using RSA or using a Widevine keybox token.

Verification

If the RSA key's allowed_schemes is not kSign_RSASSA_PSS, then no keys are derived and the error OEMCrypto_ERROR_INVALID_RSA_KEY is returned. An RSA key cannot be used for both deriving session keys and also for PKCS1 signatures.

Parameters

[in] session: handle for the session to be used.
[in] derivation_key: session key, encrypted with the public RSA key (from the DRM certifcate) using RSA-OAEP.
[in] derivation_key_length: length of derivation_key, in bytes.
[in] mac_key_context: pointer to memory containing context data for computing the HMAC generation key.
[in] mac_key_context_length: length of the HMAC key context data, in bytes.
[in] enc_key_context: pointer to memory containing context data for computing the encryption key.
[in] enc_key_context_length: length of the encryption key context data, in bytes.

Results

mac_key[server]: the 256 bit mac key is generated and stored in secure memory.

mac_key[client]: the 256 bit mac key is generated and stored in secure memory.
enc_key: the 128 bit encryption key is generated and stored in secure memory.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_DEVICE_NOT_RSA_PROVISIONED
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INVALID_CONTEXT
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support mac_key_context and enc_key_context sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffers are too large.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_GenerateNonce

```
OEMCryptoResult OEMCrypto_GenerateNonce(
                            OEMCrypto_SESSION session,
                            uint32_t* nonce);
```

Generates a 32-bit nonce to detect possible replay attack on the key control block. The nonce is stored in secure memory and will be used in the license or provisioning request.

Because the nonce will be used to prevent replay attacks, it is desirable that a rogue application cannot rapidly call this function until a repeated nonce is created randomly. This is called a nonce flood. With this in mind, if more than **200 nonces** are requested within one second, OEMCrypto will return an error after the 200th and not generate any more nonces for the rest of the second. After an error, if the application waits at least one second before requesting more nonces, then OEMCrypto will reset the error condition and generate valid nonces again.

The nonce should be stored in the sessions ODK_NonceValue field by calling the function ODK_SetNonceValue(&nonce_values, nonce). The ODK functions are documented in “Widevine Core Message Serialization”.

This function shall only be called at most once per open session. It shall only be called before signing either a provisioning request or a license request. If an attempt is made to generate a nonce while in the wrong state, an error of OEMCrypto_ERROR_INVALID_CONTEXT is returned.

Parameters

[in] session: handle for the session to be used.

[out] nonce: pointer to memory to receive the computed nonce.

Results

nonce: the nonce is also stored in secure memory. Each session should store 4 nonces.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Initialization Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_PrepAndSignLicenseRequest

```
OEMCryptoResult OEMCrypto_PrepAndSignLicenseRequest(
                              OEMCrypto_SESSION session,
                              uint8_t* message,
                              size_t message_length,
                              size_t* core_message_size,
                              uint8_t* signature,
                              size_t* signature_length);
```

OEMCrypto will use ODK_PrepareCoreLicenseRequest to prepare the core message. If it returns OEMCrypto_SUCCESS, then OEMCrypto shall sign the the message body using the DRM certificate’s private key. If it returns an error, the error should be returned by OEMCrypto to the CDM layer. ODK_PrepareCoreLicenseRequest is described in the document “Widevine Core Message Serialization”.

The message body is the buffer starting at `message + core_message_size`, and with length `message_length-core_message_size`. The reason OEMCrypto only signs the message body and not the entire message is to allow a v16 device to request a license from a v15 license server.

If the session's private RSA key has an "allowed_schemes" bit field, then it must be 0x1 (RSASSA-PSS with SHA1). If not, then an error of OEMCrypto_ERROR_SIGNATURE_FAILURE shall be returned.

OEMCrypto shall compute a hash of the core license request. The core license request is the buffer starting at `message` and with length `core_message_size`. The has will be saved with the session and verified that it matches a hash in the license response.

OEMCrypto shall also call the function **ODK_InitializeClockValues**, described in the document "License Duration and Renewal", to initialize the sessions clock values.

Refer to the Signing Messages Sent to a Server section above for more details about the signature algorithm.

NOTE: if signature pointer is null and/or input signature_length is zero, this function returns OEMCrypto_ERROR_SHORT_BUFFER and sets output signature_length to the size needed to receive the output signature.

Parameters

[in/out] message: Pointer to memory for the entire message. Modified by OEMCrypto via the ODK library.
[in] message_length: length of the entire message buffer.
[in/out] core_message_size: length of the core message at the beginning of the message. (in) size of buffer reserved for the core message, in bytes. (out) actual length of the core message, in bytes.
[out] signature: pointer to memory to receive the computed signature.
[in/out] signature_length: (in) length of the signature buffer, in bytes. (out) actual length of the signature, in bytes.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_SHORT_BUFFER if signature buffer is not large enough to hold the signature.

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_SIGNATURE_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support message sizes as described in the section

OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_PrepAndSignRenewalRequest

```
OEMCryptoResult OEMCrypto_PrepAndSignRenewalRequest(
                            OEMCrypto_SESSION session,
                            uint8_t* message,
                            size_t message_length,
                            size_t* core_message_size,
                            uint8_t* signature,
                            size_t* signature_length);
```

OEMCrypto will use ODK_PrepareCoreRenewalRequest, as described in the document “Widevine Core Message Serialization”, to prepare the core message.

If it returns an error, the error should be returned by OEMCrypto to the CDM layer. If it returns OEMCrypto_SUCCESS, then OEMCrypto computes the signature using the renewal mac key which was delivered in the license via LoadLicense.

If nonce_values.api_level is 16, then OEMCrypto shall compute the signature of the entire message using the session’s client renewal mac key. The entire message is the buffer starting at `message` with length `message_length`.

If nonce_values.api_major_version is 15, then OEMCrypto shall compute the signature of the message body using the session’s client renewal mac key. The message body is the buffer starting at `message+core_message_size` with length `message_length-core_message_size`. If the session has not had a license loaded, it will use the usage entries client mac key to sign the message body.

This function generates a HMAC-SHA256 signature using the mac_key[client] for license request signing under the license server protocol for CENC.

The key used for signing should be the mac_key[client] that was generated for this session or loaded for this session by OEMCrypto_LoadKeys, OEMCrypto_LoadLicense, or OEMCrypto_LoadUsageEntry.

Refer to the Signing Messages Sent to a Server section above for more details.

If a usage entry has been loaded, but keys have not been loaded through OEMCrypto_LoadKeys, then the derived mac keys and the keys in the usage entry may be different. In this case, the mac keys specified in the usage entry should be used.

NOTE: if signature pointer is null and/or input signature_length is zero, this function returns OEMCrypto_ERROR_SHORT_BUFFER and sets output signature_length to the size needed to receive the output signature.

Parameters

[in/out] message: Pointer to memory for the entire message. Modified by OEMCrypto via the ODK library.

[in] message_length: length of the entire message buffer.

[in/out] core_message_size: length of the core message at the beginning of the message. (in) size of buffer reserved for the core message, in bytes. (out) actual length of the core message, in bytes.

[out] signature: pointer to memory to receive the computed signature.

[in/out] signature_length: (in) length of the signature buffer, in bytes. (out) actual length of the signature, in bytes.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_SHORT_BUFFER if signature buffer is not large enough to hold the signature.

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_PrepAndSignProvisioningRequest

```
OEMCryptoResult OEMCrypto_PrepAndSignProvisioningRequest(
                          OEMCrypto_SESSION session,
```

```
    uint8_t* message,
    size_t message_length,
    size_t* core_message_size,
    uint8_t* signature,
    size_t* signature_length);
```

OEMCrypto will use ODK_PrepareCoreRenewalRequest, as described in the document "Widevine Core Message Serialization", to prepare the core message. If it returns an error, the error should be returned by OEMCrypto to the CDM layer. If it returns OEMCrypto_SUCCESS, then OEMCrypto shall sign compute the signature of the entire message. The entire message is the buffer starting at `message` with length `message_length`.

For a device that has a keybox, i.e. Provisioning 2.0, OEMCrypto will sign the response with the session's derived client mac key from the previous call to OEMCrypto_GenerateDerivedKeys.

For a device that has an OEM Certificate, i.e. Provisioning 3.0, OEMCrypto will sign the response with the private key associated with the OEM Certificate. The key shall have been loaded by a previous call to OEMCrypto_LoadDRMPrivateKey.

Refer to the Signing Messages Sent to a Server section above for more details.

NOTE: if signature pointer is null and/or input signature_length is zero, this function returns OEMCrypto_ERROR_SHORT_BUFFER and sets output signature_length to the size needed to receive the output signature.

Parameters

[in/out] message: Pointer to memory for the entire message. Modified by OEMCrypto via the ODK library.

[in] message_length: length of the entire message buffer.

[in/out] core_message_size: length of the core message at the beginning of the message. (in) size of buffer reserved for the core message, in bytes. (out) actual length of the core message, in bytes.

[out] signature: pointer to memory to receive the computed signature.

[in/out] signature_length: (in) length of the signature buffer, in bytes. (out) actual length of the signature, in bytes.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_SHORT_BUFFER if signature buffer is not large enough to hold the signature.

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_LoadSRM

```
OEMCryptoResult OEMCrypto_LoadSRM(const uint8_t* buffer,
                                  size_t buffer_length);
```

Verify and install a new SRM file. The device shall install the new file only if verification passes. If verification fails, the existing SRM will be left in place. Verification is defined by DCP, and includes verification of the SRM’s signature and verification that the SRM version number will not be decreased. See the section HDCP SRM Update above for more details about the SRM. This function is for devices that support HDCP v2.2 or higher and wish to receive 4k content.

Parameters

[in] bufer: buffer containing the SRM

[in] buffer_length: length of the SRM, in bytes.

Returns

OEMCrypto_SUCCESS - if the file was valid and was installed.
OEMCrypto_ERROR_INVALID_CONTEXT - if the SRM version is too low, or the file is corrupted.
OEMCrypto_ERROR_SIGNATURE_FAILURE - If the signature is invalid.
OEMCrypto_ERROR_BUFFER_TOO_LARGE - if the buffer is too large for the device.
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

The size of the buffer is determined by the HDCP specification.

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_LoadKeys

```
typedef struct {
  size_t offset;
  size_t length;
} OEMCrypto_Substring;

OEMCryptoResult OEMCrypto_LoadKeys(OEMCrypto_SESSION session,
                                   const uint8_t* message,
                                   size_t message_length,
                                   const uint8_t* signature,
                                   size_t signature_length,
                                   OEMCrypto_Substring enc_mac_keys_iv,
                                   OEMCrypto_Substring enc_mac_keys,
                                   size_t key_array_length,
                                   const OEMCrypto_KeyObject* key_array,
                                   OEMCrypto_Substring pst,
                                   OEMCrypto_Substring srm_restriction_data,
                                   OEMCrypto_LicenseType license_type);

typedef enum OEMCrypto_LicenseType {
  OEMCrypto_ContentLicense = 0,
  OEMCrypto_EntitlementLicense = 1
};

typedef struct {
    OEMCrypto_Substring key_id;
    OEMCrypto_Substring key_data_iv;
    OEMCrypto_Substring key_data;
    OEMCrypto_Substring key_control_iv;
    OEMCrypto_Substring key_control;
} OEMCrypto_KeyObject;

typedef struct {
  uint8_t verification[8];  // must be "HDCPDATA"
  uint32_t  minimum_srm_version;  // version number
} SRM_Restriction_Data;
```

Install a set of keys for performing decryption in the current session. This function will be deprecated and will only be used for legacy license from a license server that does not yet support the v16 interface.

The relevant fields have been extracted from the License Response protocol message, but the entire message and associated signature are provided so the message can be verified (using HMAC-SHA256 with the derived mac_key[server]). If the signature verification fails, ignore all other arguments and return OEMCrypto_ERROR_SIGNATURE_FAILURE. Otherwise, add the keys to the session context.

The keys will be decrypted using the current encrypt_key (AES-128-CBC) and the IV given in the KeyObject. Each key control block will be decrypted using the first 128 bits of the corresponding content key (AES-128-CBC) and the IV given in the KeyObject.

If its length is not zero, enc_mac_keys will be used to create new mac_keys. After all keys have been decrypted and validated, the new mac_keys are decrypted with the current encrypt_key and the offered IV. The new mac_keys replaces the current mac_keys for future calls to OEMCrypto_RefreshKeys(). The first 256 bits of the mac_keys become the mac_key[server] and the following 256 bits of the mac_keys become the mac_key[client].

The mac_key and encrypt_key were generated and stored by the previous call to OEMCrypto_GenerateDerivedKeys() or OEMCrypto_DeriveKeysFromSessionKey(). The nonce was generated and stored in the session's nonce_values by the previous call to OEMCrypto_GenerateNonce().

This session's elapsed time clock is started at 0. The clock will be used in OEMCrypto_DecryptCENC().

NOTE: The calling software must have previously established the mac_keys and encrypt_key with a call to OEMCrypto_DeriveKeysFromSessionKey().

Refer to the Verification of Messages from a Server section above for more details.

If the parameter license_type is OEMCrypto_ContentLicense, then the fields key_id and key_data in an OEMCrypto_KeyObject are loaded in to the content_key_id and content_key_data fields of the key table entry. In this case, entitlement key ids and entitlement key data is left blank.

If the parameter license_type is OEMCrypto_EntitlementLicense, then the fields key_id and key_data in an OEMCrypto_KeyObject are loaded in to the entitlement_key_id and entitlement_key_data fields of the key table entry. In this case, content key ids and content key data will be loaded later with a call to OEMCrypto_LoadEntitledContentKeys().

OEMCrypto may assume that the key_id_length is at most 16. However, OEMCrypto shall correctly handle key id lengths from 1 to 16 bytes.

OEMCrypto shall handle at least 20 keys per session. This allows a single license to contain separate keys for 3 key rotations (previous interval, current interval, next interval) times 4 content keys (audio, SD, HD, UHD) plus up to 8 keys for watermarks.

After a call to OEMCrypto_LoadKeys, oemcrypto should clear the encrypt_key for the session.

Verification

The following checks should be performed. If any check fails, an error is returned, and none of the keys are loaded.

1. The signature of the message shall be computed, and the API shall verify the computed signature matches the signature passed in. If not, return OEMCrypto_ERROR_SIGNATURE_FAILURE. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison).
2. If there already is a license loaded into this session, return OEMCrypto_ERROR_LICENSE_RELOAD.
3. The enc_mac_keys substring must either have zero length, or satisfy the range check. I.e. `(offset < message_length) && (offset + length < message_length) && (offset < offset+length)`, and `offset+length` does not cause an integer overflow.

If it does not have zero length, then enc_mac_keys_iv must not have zero length, and must also satisfy the range check. If not, return OEMCrypto_ERROR_INVALID_CONTEXT. If the length is zero, then OEMCrypto may assume that the offset is also zero.

4. The API shall verify that each substring in each KeyObject points to a location in the message. I.e. `(offset < message_length) && (offset + length < message_length) && (offset < offset+length)` and `offset+length` does not cause an integer overflow, for each of `key_id`, `key_data_iv`, `key_data`, `key_control_iv`, `key_control`. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
5. Each key's control block, after decryption, shall have a valid verification field. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
6. If any key control block has the Nonce_Enabled bit set, that key's Nonce field shall match a nonce in the cache. If not, return OEMCrypto_ERROR_INVALID_NONCE. If there is a match, remove that nonce from the cache. Note that all the key control blocks in a particular call shall have the same nonce value.
7. If any key control block has the Require_AntiRollback_Hardware bit set, and the device does not protect the usage table from rollback, then do not load the keys and return OEMCrypto_ERROR_UNKNOWN_FAILURE.
8. If the key control block has a nonzero Replay_Control, then the verification described below is also performed.
9. If the key control block has the bit SRMVersionRequired is set, then the verification described below is also performed. If the SRM requirement is not met, then the key control block's HDCP_Version will be changed to 0xF - local display only.
10. If key_array_length == 0, then return OEMCrypto_ERROR_INVALID_CONTEXT.
11. If this session is associated with a usage table entry, and that entry is marked as "inactive" (either kInactiveUsed or kInactiveUnused), then the keys are not loaded, and the error OEMCrypto_ERROR_LICENSE_INACTIVE is returned.
12. The data in enc_mac_keys_iv is not identical to the 16 bytes before enc_mac_keys. If it is, return OEMCrypto_ERROR_INVALID_CONTEXT.

Usage Table and Provider Session Token (pst)

If a key control block has a nonzero value for Replay_Control, then all keys in this license will have the same value for Replay_Control. In this case, the following additional checks are performed.

- The substring `pst` must have nonzero length and must satisfy the range check described above. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
- The session must be associated with a usage table entry, either created via OEMCrypto_CreateNewUsageEntry or loaded via OEMCrypto_LoadUsageEntry.
- If Replay_Control is 1 = Nonce_Required, then OEMCrypto will perform a nonce check as described above. OEMCrypto will verify that the usage entry is newly created with OEMCrypto_CreateNewUsageEntry. If an existing entry was reloaded, an error OEMCrypto_ERROR_INVALID_CONTEXT is returned and no keys are loaded. OEMCrypto will then copy the pst and the mac keys to the usage entry, and set the

status to Unused. This Replay_Control prevents the license from being loaded more than once, and will be used for online streaming.

- If Replay_Control is 2 = “Require existing Session Usage table entry or Nonce”, then OEMCrypto will behave slightly differently on the first call to LoadKeys for this license.
    - If the usage entry was created with OEMCrypto_CreateNewUsageEntry for this session, then OEMCrypto will verify the nonce for each key. OEMCrypto will copy the pst and mac keys to the usage entry. The license received time of the entry will be updated to the current time, and the status will be set to Unused.
    - If the usage entry was loaded with OEMCrypto_LoadUsageEntry for this session, then OEMCrypto will **NOT** verify the nonce for each key. Instead, it will verify that the pst passed in matches that in the entry. Also, the entry’s mac keys will be verified against the current session’s mac keys. This allows an offline license to be reloaded but maintain continuity of the playback times from one session to the next.
    - If the nonce is not valid and a usage entry was not loaded, the return error is OEMCrypto_ERROR_INVALID_NONCE.
    - If the loaded usage entry has a pst that does not match, OEMCrypto returns the error OEMCrypto_ERROR_WRONG_PST.
    - If the loaded usage entry has mac keys that do not match the license, OEMCrypto returns the error OEMCrypto_ERROR_WRONG_KEYS.

Note: If LoadKeys updates the mac keys, then the new updated mac keys will be used with the Usage Entry -- i.e. the new keys are stored in the usage table when creating a new entry, or the new keys are verified against those in the usage table if there is an existing entry. If LoadKeys does not update the mac keys, the existing session mac keys are used.
Sessions that are associated with an entry will need to be able to update and verify the status of the entry, and the time stamps in the entry.

Devices that do not support the Usage Table will return OEMCrypto_ERROR_INVALID_CONTEXT if the Replay_Control is nonzero.

Timer Update

After verification, the session’s clock and timer values are updated by calling the function ODK_InitializeV15Values as described in the document “Widevine Core Message Serialization”.

SRM Restriction Data

If any key control block has the flag SRMVersionRequired set, then the following verification is also performed.

1. The substring srm_restriction_data must have nonzero length and must satisfy the range check described above. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
2. The first 8 bytes of srm_restriction_data must match the string “HDCPDATA”. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
3. The next 4 bytes of srm_restriction_data will be converted from network byte order. If

the current SRM installed on the device has a version number less than this, then the SRM requirement is not met. If the device does not support SRM files, or OEMCrypto cannot determine the current SRM version number, then the SRM requirement is not met.

Note: if the current SRM version requirement is not met, LoadKeys will still succeed and the keys will be loaded. However, those keys with the SRMVersionRequired bit set will have their HDCP_Version increased to 0xF - local display only. Any future call to SelectKey for these keys while there is an external display will return OEMCrypto_ERROR_INSUFFICIENT_HDCP at that time.

Parameters

[in] session: crypto session identifier.

[in] message: pointer to memory containing message to be verified.

[in] message_length: length of the message, in bytes.

[in] signature: pointer to memory containing the signature.

[in] signature_length: length of the signature, in bytes.

[in] enc_mac_keys_iv: IV for decrypting new mac_key. Size is 128 bits.

[in] enc_mac_keys: encrypted mac_keys for generating new mac_keys. Size is 512 bits.

[in] key_array_length: number of keys present.

[in] key_array: set of keys to be installed.

[in] pst: the Provider Session Token.

[in] srm_restriction_data: optional data specifying the minimum SRM version.

[in] license_type: specifies if the license contains content keys or entitlement keys.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_NO_DEVICE_KEY

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_SIGNATURE_FAILURE

OEMCrypto_ERROR_INVALID_NONCE

OEMCrypto_ERROR_TOO_MANY_KEYS

OEMCrypto_ERROR_NOT_IMPLEMENTED

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

OEMCrypto_ERROR_LICENSE_RELOAD

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_LoadLicense

```
typedef struct {
  size_t offset;
  size_t length;
} OEMCrypto_Substring;

OEMCryptoResult OEMCrypto_LoadLicense(OEMCrypto_SESSION session,
                                      const uint8_t* message,
                                      size_t message_length,
                                      size_t core_message_length,
                                      const uint8_t* signature,
                                      size_t signature_length);
```

Install a set of keys for performing decryption in the current session.

First, OEMCrypto shall verify the signature of the message using HMAC-SHA256 with the derived mac_key[server]. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison). The signature is over the entire message buffer starting at `message` with length `message_length`. If the signature verification fails, ignore all other arguments and return OEMCrypto_ERROR_SIGNATURE_FAILURE. Otherwise, add the keys to the session context.

NOTE: The calling software must have previously established the mac_keys and encrypt_key with a call to OEMCrypto_DeriveKeysFromSessionKey().

Refer to the Verification of Messages from a Server section above for more details.

The function ODK_ParseLicense is called to parse the message. If it returns an error, OEMCrypto shall return that error to the CDM layer. The function ODK_ParseLicense is described in the document “Widevine Core Message Serialization”.

Below, all fields are found in the struct ODK_ParsedLicense parsed_license returned by ODK_ParseLicense.

The keys will be decrypted using the current encrypt_key (AES-128-CBC) and the IV given in the KeyObject. Each key control block will be decrypted using the first 128 bits of the corresponding content key (AES-128-CBC) and the IV given in the KeyObject.

If its length is not zero, enc_mac_keys will be used to create new mac_keys. After all keys have

been decrypted and validated, the new mac_keys are decrypted with the current encrypt_key and the offered IV. The new mac_keys replaces the current mac_keys for future signing renewal requests and loading renewal responses. The first 256 bits of the mac_keys become the mac_key[server] and the following 256 bits of the mac_keys become the mac_key[client]. If enc_mac_keys is null, then there will not be a call to OEMCrypto_LoadRenewal for this session and the current mac_keys may be deleted.

If the field license_type is OEMCrypto_ContentLicense, then the fields key_id and key_data in an OEMCrypto_KeyObject are loaded in to the content_key_id and content_key_data fields of the key table entry. In this case, entitlement key ids and entitlement key data is left blank.

If the field license_type is OEMCrypto_EntitlementLicense, then the fields key_id and key_data in an OEMCrypto_KeyObject are loaded in to the entitlement_key_id and entitlement_key_data fields of the key table entry. In this case, content key ids and content key data will be loaded later with a call to OEMCrypto_LoadEntitledContentKeys().

OEMCrypto may assume that the key_id_length is at most 16. However, OEMCrypto shall correctly handle key id lengths from 1 to 16 bytes.

OEMCrypto shall handle multiple keys, as described in the section on Resource Rating Tiers in this document.

After a call to OEMCrypto_LoadLicense, oemcrypto should clear the encrypt_key for the session.

Verification

The following checks should be performed. If any check fails, an error is returned, and none of the keys are loaded.

13. The signature of the message shall be computed, and the API shall verify the computed signature matches the signature passed in. If not, return OEMCrypto_ERROR_SIGNATURE_FAILURE. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison).
14. If there already is a license loaded into this session, return OEMCrypto_ERROR_LICENSE_RELOAD.
15. The enc_mac_keys substring must either have zero length, or satisfy the range check. I.e. `(offset < message_length) && (offset + length < message_length) && (offset < offset+length)`, and `offset+length` does not cause an integer overflow. If it does not have zero length, then enc_mac_keys_iv must not have zero length, and must also satisfy the range check. If not, return OEMCrypto_ERROR_INVALID_CONTEXT. If the length is zero, then OEMCrypto may assume that the offset is also zero.
16. The API shall verify that each substring in each KeyObject points to a location in the message. I.e. `(offset < message_length) && (offset + length < message_length) && (offset < offset+length)` and `offset+length` does not cause an integer overflow, for each of `key_id`, `key_data_iv`, `key_data`, `key_control_iv`, `key_control`. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.

17. Each key's control block, after decryption, shall have a valid verification field. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
18. If any key control block has the Nonce_Enabled bit set, that key's Nonce field shall match a nonce in the cache. If not, return OEMCrypto_ERROR_INVALID_NONCE. If there is a match, remove that nonce from the cache. Note that all the key control blocks in a particular call shall have the same nonce value.
19. If any key control block has the Require_AntiRollback_Hardware bit set, and the device does not protect the usage table from rollback, then do not load the keys and return OEMCrypto_ERROR_UNKNOWN_FAILURE.
20. If the key control block has a nonzero Replay_Control, then the verification described below is also performed.
21. If the key control block has the bit SRMVersionRequired is set, then the verification described below is also performed. If the SRM requirement is not met, then the key control block's HDCP_Version will be changed to 0xF - local display only.
22. If key_array_length == 0, then return OEMCrypto_ERROR_INVALID_CONTEXT.
23. If this session is associated with a usage table entry, and that entry is marked as "inactive" (either kInactiveUsed or kInactiveUnused), then the keys are not loaded, and the error OEMCrypto_ERROR_LICENSE_INACTIVE is returned.
24. The data in enc_mac_keys_iv is not identical to the 16 bytes before enc_mac_keys. If it is, return OEMCrypto_ERROR_INVALID_CONTEXT.

Usage Table and Provider Session Token (pst)

The function ODK_ParseLicense takes several parameters that may need more explanation.

The parameter usage_entry_present shall be set to true if a usage entry was created or loaded for this session. This parameter is used by ODK_ParseLicense for usage entry verification.

The parameter initial_license_load shall be false if the usage entry was loaded. If there is no usage entry or if the usage entry was created with OEMCrypto_CreateNewUsageEntry, then initial_license_load shall be true.

If a usage entry is present, then it shall be verified after the call to ODK_ParseLicense.

If initial_license_load is true:

1. OEMCrypto shall copy the PST from the parsed license to the usage entry.
2. OEMCrypto shall verify that the server and client mac keys were updated by the license. The server and client mac keys shall be copied to the usage entry.

If initial_license_load is false:

1. OEMCrypto shall verify the PST from the parsed license matches that in the usage entry. If not, then an error OEMCrypto_ERROR_WRONG_PST is returned.
2. OEMCrypto shall verify that the server and client mac keys were updated by the license. OEMCrypto shall verify that the server and client mac keys match those in the usage entry. If not the error OEMCrypto_ERROR_WRONG_KEYS is returned.

If a key control block has a nonzero value for Replay_Control, then all keys in this license will have the same value for Replay_Control. In this case, the following additional checks are performed.

- The substring `pst` must have nonzero length and must satisfy the range check described above. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
- The session must be associated with a usage table entry, either created via OEMCrypto_CreateNewUsageEntry or loaded via OEMCrypto_LoadUsageEntry.
- If Replay_Control is 1 = Nonce_Required, then OEMCrypto will perform a nonce check as described above. OEMCrypto will verify that the usage entry is newly created with OEMCrypto_CreateNewUsageEntry. If an existing entry was reloaded, an error OEMCrypto_ERROR_INVALID_CONTEXT is returned and no keys are loaded. OEMCrypto will then copy the pst and the mac keys to the usage entry, and set the status to Unused. This Replay_Control prevents the license from being loaded more than once, and will be used for online streaming.
- If Replay_Control is 2 = "Require existing Session Usage table entry or Nonce", then OEMCrypto will behave slightly differently on the first call to LoadKeys for this license.
    - If the usage entry was created with OEMCrypto_CreateNewUsageEntry for this session, then OEMCrypto will verify the nonce for each key. OEMCrypto will copy the pst and mac keys to the usage entry. The license received time of the entry will be updated to the current time, and the status will be set to Unused.
    - If the usage entry was loaded with OEMCrypto_LoadUsageEntry for this session, then OEMCrypto will **NOT** verify the nonce for each key. Instead, it will verify that the pst passed in matches that in the entry. Also, the entry's mac keys will be verified against the current session's mac keys. This allows an offline license to be reloaded but maintain continuity of the playback times from one session to the next.
    - If the nonce is not valid and a usage entry was not loaded, the return error is OEMCrypto_ERROR_INVALID_NONCE.
    - If the loaded usage entry has a pst that does not match, OEMCrypto returns the error OEMCrypto_ERROR_WRONG_PST.
    - If the loaded usage entry has mac keys that do not match the license, OEMCrypto returns the error OEMCrypto_ERROR_WRONG_KEYS.

Note: If LoadKeys updates the mac keys, then the new updated mac keys will be used with the Usage Entry -- i.e. the new keys are stored in the usage table when creating a new entry, or the new keys are verified against those in the usage table if there is an existing entry. If LoadKeys does not update the mac keys, the existing session mac keys are used. Sessions that are associated with an entry will need to be able to update and verify the status of the entry, and the time stamps in the entry.

Devices that do not support the Usage Table will return OEMCrypto_ERROR_INVALID_CONTEXT if the Replay_Control is nonzero.

SRM Restriction Data

If any key control block has the flag SRMVersionRequired set, then the following verification is also performed.

4. The substring srm_restriction_data must have nonzero length and must satisfy the range check described above. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.

5. The first 8 bytes of srm_restriction_data must match the string “HDCPDATA”. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
6. The next 4 bytes of srm_restriction_data will be converted from network byte order. If the current SRM installed on the device has a version number less than this, then the SRM requirement is not met. If the device does not support SRM files, or OEMCrypto cannot determine the current SRM version number, then the SRM requirement is not met.

Note: if the current SRM version requirement is not met, LoadKeys will still succeed and the keys will be loaded. However, those keys with the SRMVersionRequired bit set will have their HDCP_Version increased to 0xF - local display only. Any future call to SelectKey for these keys while there is an external display will return OEMCrypto_ERROR_INSUFFICIENT_HDCP at that time.

Parameters

[in] session: crypto session identifier.

[in] message: pointer to memory containing data.

[in] message_length: length of the message, in bytes.

[in] core_message_length: length of the core submessage, in bytes.

[in] signature: pointer to memory containing the signature.

[in] signature_length: length of the signature, in bytes.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_NO_DEVICE_KEY

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_SIGNATURE_FAILURE

OEMCrypto_ERROR_INVALID_NONCE

OEMCrypto_ERROR_TOO_MANY_KEYS

OEMCrypto_ERROR_NOT_IMPLEMENTED

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

OEMCrypto_ERROR_LICENSE_RELOAD

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_LoadEntitledContentKeys

```
OEMCryptoResult OEMCrypto_LoadEntitledContentKeys(
    OEMCrypto_SESSION session,
    const uint8_t* message,
    size_t message_length,
    size_t key_array_length,
    const OEMCrypto_EntitledContentKeyObject* key_array);

typedef struct {
  OEMCrypto_Substring entitlement_key_id;
  OEMCrypto_Substring content_key_id;
  OEMCrypto_Substring content_key_data_iv;
  OEMCrypto_Substring content_key_data;
} OEMCrypto_EntitledContentKeyObject;
```

Load content keys into a session which already has entitlement keys loaded. This function will only be called for a session after a call to OEMCrypto_LoadKeys with the parameter type license_type equal to OEMCrypto_EntitlementLicense. This function may be called multiple times for the same session.

If the session does not have license_type equal to OEMCrypto_EntitlementLicense, return OEMCrypto_ERROR_INVALID_CONTEXT and perform no work.

For each key object in key_array, OEMCrypto shall look up the entry in the key table with the corresponding entitlement_key_id.

1. If no entry is found, return OEMCrypto_KEY_NOT_ENTITLED.
2. If the entry already has a content_key_id and content_key_data, that id and data are erased.
3. The content_key_id from the key_array is copied to the entry’s content_key_id.
4. The content_key_data decrypted using the entitlement_key_data as a key for **AES-256-CBC** with an IV of *content_key_data_iv*. Wrapped content is padded using PKCS#7 padding. Notice that the entitlement key will be an AES 256 bit key. The clear content key data will be stored in the entry’s content_key_data.

Entries in the key table that do **not** correspond to anything in the key_array are **not** modified or removed.

For devices that use a hardware key ladder, it may be more convenient to store the encrypted content key data in the key table, and decrypt it when the function SelectKey is called.

Parameters

[in] session: handle for the session to be used.

[in] message: pointer to memory containing message to be verified.

[in] message_length: length of the message, in bytes.

[in] key_array_length: number of keys present.

[in] key_array: set of key updates.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_KEY_NOT_ENTITLED

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is new in API version 14.

OEMCrypto_RefreshKeys

```
OEMCryptoResult OEMCrypto_RefreshKeys(OEMCrypto_SESSION session,
                       const uint8_t* message,
                       size_t message_length,
                       const uint8_t* signature,
                       size_t signature_length,
                       size_t num_keys,
                       const OEMCrypto_KeyRefreshObject* key_array);

typedef struct {
    OEMCrypto_Substring key_id;
    OEMCrypto_Substring key_control_iv;
    OEMCrypto_Substring key_control;
} OEMCrypto_KeyRefreshObject;
```

Updates the license clock values to allow playback to continue. This function is being deprecated and is only used for version v15 licenses -- i.e. offline license saved before an update or licenses from a server that has not update to the v16 license server SDK.

OEMCrypto shall compute the signature of the message using mac_key[server], and shall verify the computed signature matches the signature passed in. If not, return OEMCrypto_ERROR_SIGNATURE_FAILURE. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful

comparison).

The key control from the first OEMCrypto_KeyRefreshObject in the key_array shall be extracted. If it is encrypted, as described below, it shall be decrypted. The duration from the key control shall be extracted and converted to host byte order. This duration shall be passed to the function ODK_RefreshV15Values as the parameter new_key_duration.

If the KeyRefreshObject's key_control_iv has zero length, then the key_control is not encrypted. If the key_control_iv is specified, then key_control is encrypted with the first 128 bits of the corresponding content key.

If the KeyRefreshObject's key_id has zero length, then it is an error for the key_control_iv to have nonzero length. OEMCrypto shall return an error of OEMCrypto_ERROR_INVALID_CONTEXT.

If the session's license_type is OEMCrypto_ContentLicense, and the KeyRefreshObject's key_id is not null, then the entry in the keytable with the matching content_key_id is used.

If the session's license_type is OEMCrypto_EntitlementLicense, and the KeyRefreshObject's key_id is not null, then the entry in the keytable with the matching entitlment_key_id is used.

The function ODK_RefreshV15Values shall be called to update the clock values. See the document "Widevine Core Message Serialization" for the documentation of the ODK library functions.

If ODK_RefreshV15Values returns

- ODK_SET_TIMER: Success. The timer should be reset to the specified timer value.
- ODK_DISABLE_TIMER: Success, but disable timer. Unlimited playback is allowed.
- ODK_TIMER_EXPIRED: Set timer as disabled. Playback is **not** allowed.
- ODK_STALE_RENEWAL: This renewal is not the most recently signed. It is rejected. Return this error
- Any other error - OEMCrypto shall pass any other error up to the caller of OEMCrypto_RefreshKeys.

NOTE: OEMCrypto_LoadKeys() must be called first to load the keys into the session.

Parameters

[in] session: handle for the session to be used.
[in] message: pointer to memory containing message to be verified.
[in] message_length: length of the message, in bytes.
[in] signature: pointer to memory containing the signature.
[in] signature_length: length of the signature, in bytes.
[in] num_keys: number of keys present.
[in] key_array: set of key updates.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_NO_DEVICE_KEY

OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INVALID_CONTEXT
OEMCrypto_ERROR_SIGNATURE_FAILURE
OEMCrypto_ERROR_INVALID_NONCE
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_NO_CONTENT_KEY
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_LoadRenewal

```
OEMCryptoResult OEMCrypto_LoadRenewal(OEMCrypto_SESSION session,
                                      const uint8_t* message,
                                      size_t message_length,
                                      size_t core_message_length,
                                      const uint8_t* signature,
                                      size_t signature_length);
```

Updates the clock values and resets the renewal timer for the current session.

OEMCrypto shall verify the signature of the entire message using the session’s renewal mac key for the server. The entire message is the buffer starting at `message` with length `message_length`. If the signature does not match, OEMCrypto returns OEMCrypto_ERROR_SIGNATURE_FAILURE.

OEMCrypto shall verify that nonce_values.api_major_version is 16. If not, return the error OEMCrypto_ERROR_INVALID_CONTEXT. Legacy licenses will use the function OEMCrypto_RefreshKeys instead of OEMCrypto_LoadRenewal.

If the signature passes, OEMCrypto shall use the function ODK_ParseRenewal, as described in the document “Widevine Core Message Serialization” to parse and verify the message. If ODK_ParseRenewal returns an error OEMCrypto returns the error to the CDM layer.

The function ODK_ParseRenewal updates the clock values for the session, and may return ODK_SET_TIMER, ODK_DISABLE_TIMER or ODK_TIMER_EXPIRED on success. These values shall be handled by OEMCrypto, as discussed in the document “License Duration and Renewal”.

NOTE: OEMCrypto_LoadLicense() must be called first to load the keys into the session.

Verification

The following checks should be performed. If any check fails, an error is returned, and none of the keys are loaded.

1. The signature of the message shall be computed using mac_key[server], and the API shall verify the computed signature matches the signature passed in. If not, return OEMCrypto_ERROR_SIGNATURE_FAILURE. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison).
2. The API shall verify that each substring in each KeyObject has zero length or satisfies the range check described in the discussion of OEMCrypto_LoadKeys. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
3. Each key’s control block shall have a valid verification field. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
4. If the key control block has the Nonce_Enabled bit set, the Nonce field shall match one of the nonces in the cache. If not, return OEMCrypto_ERROR_INVALID_NONCE. If there is a match, remove that nonce from the cache. Note that all the key control blocks in a particular call shall have the same nonce value.
5. If a key ID is specified, and that key has not been loaded into this session, return OEMCrypto_ERROR_NO_CONTENT_KEY.

Parameters

[in] session: handle for the session to be used.
[in] message: pointer to memory containing message to be verified.
[in] message_length: length of the message, in bytes.
[in] core_message_length: length of the core submessage, in bytes.
[in] signature: pointer to memory containing the signature.
[in] signature_length: length of the signature, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INVALID_CONTEXT
OEMCrypto_ERROR_SIGNATURE_FAILURE

OEMCrypto_ERROR_INVALID_NONCE

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_NO_CONTENT_KEY

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

ODK_STALE_RENEWAL

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 12.

OEMCrypto_QueryKeyControl

```
OEMCryptoResult
OEMCrypto_QueryKeyControl(OEMCrypto_SESSION session,
                          const uint8_t* content_key_id,
                          size_t content_key_id_length,
                          uint8_t* key_control_block,
                          size_t* key_control_block_length);
```

Returns the decrypted key control block for the given content_key_id. This function is for application developers to debug license server and key timelines. It only returns a key control block if LoadKeys was successful, otherwise it returns OEMCrypto_ERROR_NO_CONTENT_KEY. The developer of the OEMCrypto library must be careful that the keys themselves are not accidentally revealed.

Note: returns control block in original, **network byte order**. If OEMCrypto converts fields to host byte order internally for storage, it should convert them back. Since OEMCrypto might not store the nonce or validation fields, values of 0 may be used instead.

Verification

The following checks should be performed.

1. If key_id is null, return OEMCrypto_ERROR_INVALID_CONTEXT.
2. If key_control_block_length is null, return OEMCrypto_ERROR_INVALID_CONTEXT.
3. If *key_control_block_length is less than the length of a key control block, set it to the correct value, and return OEMCrypto_ERROR_SHORT_BUFFER.
4. If key_control_block is null, return OEMCrypto_ERROR_INVALID_CONTEXT.
5. If the specified key has not been loaded, return OEMCrypto_ERROR_NO_CONTENT_KEY.

Parameters

[in] session: handle for the session to be used.

[in] content_key_id: The unique id of the key of interest.

[in] content_key_id_length: The length of key_id, in bytes. From 1 to 16, inclusive.

[out] key_control_block: A caller-owned buffer.

[in/out] key_control_block_length. The length of key_control_block buffer.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

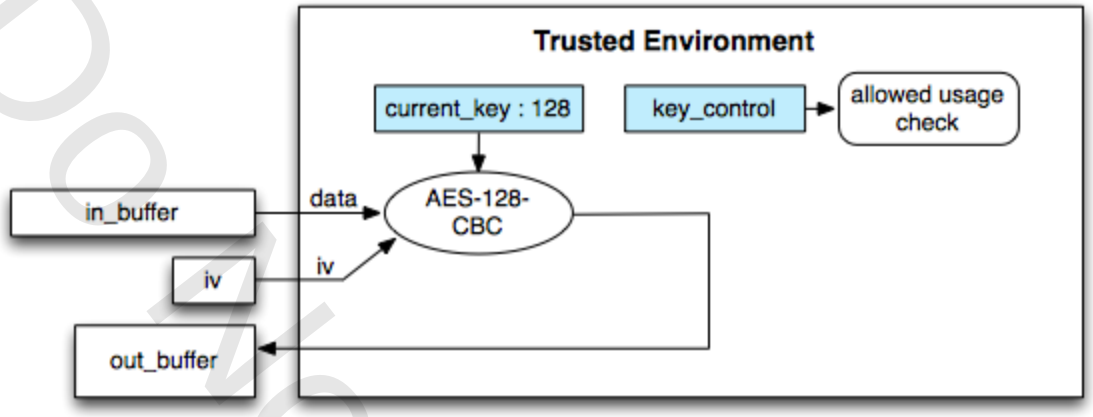
This method is new in API version 10.

Decryption API

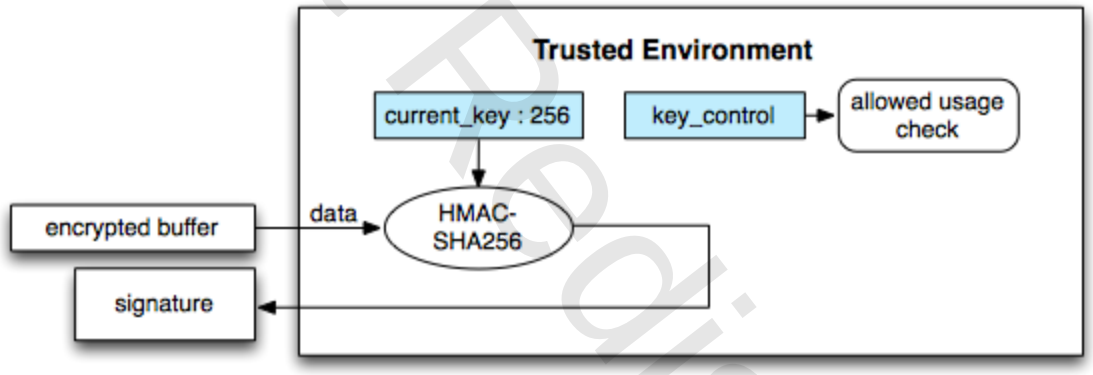
Devices that implement the Key Ladder API must also support a secure decode or secure decode and rendering implementation. This can be done by either decrypting into buffers secured by hardware protections and providing these secured buffers to the decoder/renderer or by implementing decrypt operations in the decoder/renderer.

In a Security Level 2 implementation where the video path is not protected, the audio and video streams are decrypted using OEMCrypto_DecryptCENC() and buffers are returned to the media player in the clear.

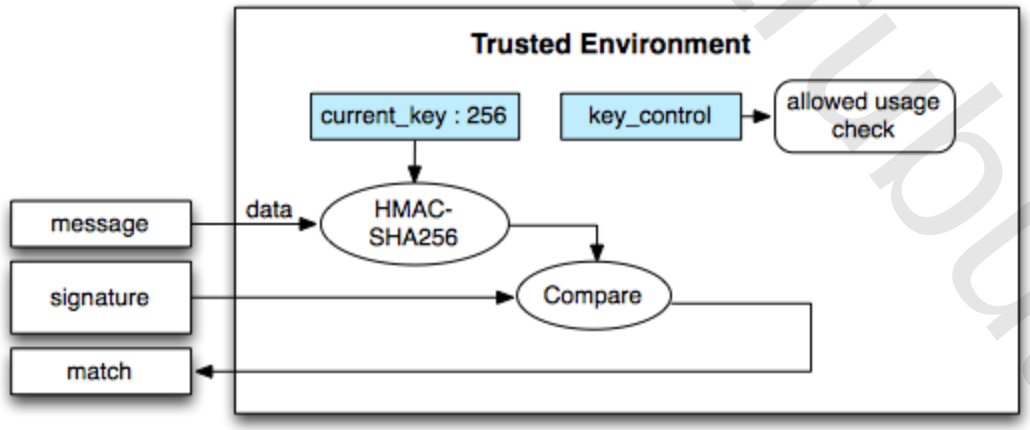
Generic Modular DRM allows an application to encrypt, decrypt, sign and verify arbitrary user data using a content key. This content key is securely delivered from the server to the client device using the same factory installed root of trust as a media content keys.



OEMCrypto_Generic_Decrypt(), OEMCrypto_Generic_Encrypt()



OEMCrypto_Generic_Sign()



OEMCrypto_Generic_Verify()

OEMCrypto_SelectKey

```
OEMCryptoResult OEMCrypto_SelectKey(OEMCrypto_SESSION session,
                                    const uint8_t* content_key_id,
                                    size_t content_key_id_length,
                                    OEMCryptoCipherMode cipher_mode);

typedef enum OEMCryptoCipherMode {
    OEMCrypto_CipherMode_CTR,
    OEMCrypto_CipherMode_CBC,
} OEMCryptoCipherMode;
```

Select a content key and install it in the hardware key ladder for subsequent decryption operations (OEMCrypto_DecryptCENC()) for this session. The specified key must have been previously "installed" via OEMCrypto_LoadKeys(), OEMCrypto_LoadLicense, or OEMCrypto_LoadEntitledContentKeys().

A key control block is associated with the key and the session, and is used to configure the session context. The Key Control data is documented in "Key Control Block Definition".

Step 1: Lookup the content key data via the offered key_id. The key data includes the key value, and the key control block.

Step 2: Latch the content key into the hardware key ladder. Set permission flags based on the key's control block.

Step 3: use the latched content key to decrypt (AES-128-CTR or AES-128-CBC) buffers passed in via OEMCrypto_DecryptCENC(). If the key is 256 bits it will be used for OEMCrypto_Generic_Sign or OEMCrypto_Generic_Verify as specified in the key control block. If the key will be used for OEMCrypto_Generic_Encrypt or OEMCrypto_Generic_Decrypt then the cipher mode will always be OEMCrypto_CipherMode_CBC. Continue to use this key for this session until OEMCrypto_SelectKey() is called again, or until OEMCrypto_CloseSession() is called.

Verification

1. If the key id is not found in the keytable for this session, then the key state is not changed and OEMCrypto shall return OEMCrypto_ERROR_NO_CONTENT_KEY.
2. If the key control block has the bit Disable_Analog_Output set, then the device should disable analog video output. If the device has analog video output that cannot be disabled, then the key is not selected, and OEMCrypto_ERROR_ANALOG_OUTPUT is returned. This step is optional -- SelectKey may return OEMCrypto_SUCCESS and delay the error until a call to OEMCrypto_DecryptCENC.
3. If the key control block has HDCP required, and the device cannot enforce HDCP, then the key is not selected, and OEMCrypto_ERROR_INSUFFICIENT_HDCP is returned. This step is optional -- SelectKey may return OEMCrypto_SUCCESS and delay the error until a call to OEMCrypto_DecryptCENC.
4. If the key control block has a nonzero value for HDCP_Version, and the device cannot enforce at least that version of HDCP, then the key is not selected, and OEMCrypto_ERROR_INSUFFICIENT_HDCP is returned.

Parameters

[in] session: crypto session identifier.

[in] content_key_id: pointer to the content Key ID.

[in] content_key_id_length: length of the content Key ID, in bytes. From 1 to 16, inclusive.

[in] cipher_mode: whether the key should be prepared for CTR mode or CBC mode when used in later calls to DecryptCENC. This should be ignored when the key is used for Generic Crypto calls.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_KEY_EXPIRED - if the session's timer has expired

OEMCrypto_ERROR_INVALID_SESSION crypto session ID invalid or not open

OEMCrypto_ERROR_NO_DEVICE_KEY failed to decrypt device key

OEMCrypto_ERROR_NO_CONTENT_KEY failed to decrypt content key

OEMCrypto_ERROR_CONTROL_INVALID invalid or unsupported control input

OEMCrypto_ERROR_KEYBOX_INVALID cannot decrypt and read from Keybox

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_KEY_EXPIRED
OEMCrypto_ERROR_ANALOG_OUTPUT

OEMCrypto_ERROR_INSUFFICIENT_HDCP

OEMCrypto_ERROR_NO_CONTENT_KEY

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a "Session Function" and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_DecryptCENC

```
OEMCryptoResult OEMCrypto_DecryptCENC(
    OEMCrypto_SESSION session,
    const OEMCrypto_SampleDescription* samples,  // an array of samples.
    size_t samples_length,  // the number of samples.
    const OEMCrypto_CENCEncryptPatternDesc* pattern);
```

Decrypts or copies a series of input payloads into output buffers using the session context indicated by the `session` parameter. The input payload is delivered in the form of samples.

The samples are subdivided into subsamples. “Samples” and “subsamples” are defined as in the ISO Common Encryption standard (**ISO/IEC 23001-7:2016**). The `samples` parameter contains a list of samples, each of which has its own input and output buffers. Each sample contains a `buffers` field that contains the input and output buffers in its `input_data` and `output` fields, respectively.

Each sample contains an array of subsample descriptions in its `subsamples` field. Each subsample is defined as a number of clear bytes followed by a number of encrypted bytes. Subsamples are consecutive inside the sample; the clear bytes of the second subsample begin immediately after the encrypted bytes of the first subsample. This follows the definition in the ISO-CENC standard.

Decryption mode is AES-128-CTR or AES-128-CBC depending on the value of `cipher_mode` previously passed in to `OEMCrypto_SelectKey`. For the encrypted portion of subsamples, the content key associated with the session is latched in the active hardware key ladder and is used for the decryption operation. For the clear portion of subsamples, the data is simply copied.

After decryption, all the `input_data` bytes are copied to the location described by the `output` field. The `output` field is an `OEMCrypto_DestBufferDesc`, which could be one of:

1. The structure `OEMCrypto_DestBufferDesc` contains a pointer to a clear text buffer. The OEMCrypto library shall verify that key control allows data to be returned in clear text. If it is not authorized, this method should return an error.
2. The structure `OEMCrypto_DestBufferDesc` contains a handle to a secure buffer.
3. The structure `OEMCrypto_DestBufferDesc` indicates that the data should be sent directly to the decoder and renderer.

Depending on your platform’s needs, you may not need to support all three of these options.

SINGLE-SAMPLE DECRYPTION AND SINGLE-SUBSAMPLE DECRYPTION:

If the OEMCrypto implementation is not able to handle the amount of samples and subsamples passed into it, it should return `OEMCrypto_ERROR_BUFFER_TOO_LARGE`, in which case the CDM can respond by breaking the samples up into smaller pieces and trying to decrypt each of them individually. It is possible that the CDM will break the samples array up into pieces that are still too large, in which case OEMCrypto may return `OEMCrypto_ERROR_BUFFER_TOO_LARGE` again.

If the OEMCrypto implementation cannot handle multiple samples at once, it may return `OEMCrypto_ERROR_BUFFER_TOO_LARGE` any time it receives more than one sample in a single call to `OEMCrypto_DecryptCENC`.

Similarly, if the OEMCrypto implementation cannot handle multiple subsamples at once, it may return `OEMCrypto_ERROR_BUFFER_TOO_LARGE` any time it receives more than one subsample in a single call to `OEMCrypto_DecryptCENC`.

The exact way that the CDM code breaks up the samples array is not guaranteed by this specification. The CDM may break down the array of samples into many arrays each containing one sample. The CDM may break down samples into subsamples and pass individual

subsamples into OEMCrypto, just like in OEMCrypto v15. The CDM may break down individual subsamples into smaller subsamples, just like in OEMCrypto v15.

If OEMCrypto requests that the CDM break samples into subsamples, the “samples” passed into `OEMCrypto_DecryptCENC` will no longer be full samples. When a full sample is passed into `OEMCrypto_DecryptCENC`, the first subsample in the subsample array will have the `OEMCrypto_FirstSubsample` flag set in its `subsample_flags` field and the last subsample array will have the `OEMCrypto_LastSubsample` flag set in its `subsample_flags` field. If this is not the case, OEMCrypto will need to accumulate more subsamples from successive calls to `OEMCrypto_DecryptCENC` to receive the full sample.

The first subsample in the sample will always have `OEMCrypto_FirstSubsample` set and the last subsample will always have the `OEMCrypto_LastSubsample` flag set, even if those subsamples are passed in separate calls to `OEMCrypto_DecryptCENC`. This is the same as in OEMCrypto v15. The decrypted data will not be used until after the subsample with the flag `OEMCrypto_LastSubsample` has been sent to OEMCrypto. This can be relied on by OEMCrypto for optimization by not doing decrypt until the last subsample has been received. However, a device that can do decrypt of more than one subsample at a time will always have better performance if it can receive those subsamples in one `OEMCrypto_Decrypt` call rather than as individual subsamples.

Although the exact way that the CDM code breaks up the samples array when it receives `OEMCrypto_ERROR_BUFFER_TOO_LARGE` is not guaranteed by this specification, here is a sample way it *might* work:

1. It tries to pass the array of samples to `OEMCrypto_DecryptCENC`.
2. If OEMCrypto returns `OEMCrypto_ERROR_BUFFER_TOO_LARGE`, it tries to pass each sample individually into `OEMCrypto_DecryptCENC`.
3. If OEMCrypto returns `OEMCrypto_ERROR_BUFFER_TOO_LARGE`, it tries to pass the clear and encrypted parts of each subsample individually into `OEMCrypto_DecryptCENC`. At this point, (and in the subsequent steps) it is replicating the behavior of OEMCrypto v15 and lower.
4. If OEMCrypto returns `OEMCrypto_ERROR_BUFFER_TOO_LARGE`, it breaks each piece of a subsample into smaller pieces, down to the minimum subsample size required by the device’s resource rating tier. It passes these pieces into `OEMCrypto_DecryptCENC`.
5. If OEMCrypto returns `OEMCrypto_ERROR_BUFFER_TOO_LARGE`, the device has failed to meet its resource rating tier requirements. It returns an error.

Because this process requires a lot of back-and-forth between the CDM and OEMCrypto, partners are *strongly* recommended to support decrypting full samples or even multiple samples in their OEMCrypto implementation.

ISO-CENC SCHEMES:

The ISO Common Encryption standard (**ISO/IEC 23001-7:2016**) defines four “schemes” that may be used to encrypt content: ‘cenc’, ‘cens’, ‘cbc1’, and ‘cbcs’. Starting with v16, OEMCrypto only supports ‘cenc’ and ‘cbcs’. The schemes ‘cens’ and ‘cbc1’ are not supported.

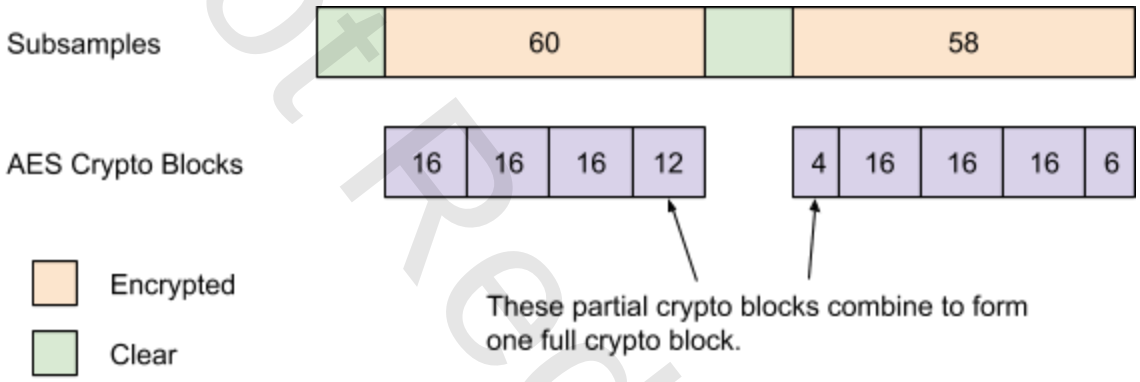
The decryption mode, either `OEMCrypto_CipherMode_CTR` or

OEMCrypto_CipherMode_CBC, was already specified in the call to OEMCrypto_SelectKey. The encryption pattern is specified by the fields in the parameter pattern. A description of partial encryption patterns for 'cbcs' can be found in the ISO-CENC standard, section 10.4.

'cenc' SCHEME:

The 'cenc' scheme is OEMCrypto_CipherMode_CTR without an encryption pattern. All the bytes in the encrypted portion of each subsample are encrypted. In the pattern parameter, both the encrypt and skip fields will be zero.

The length of a crypto block in AES-128 is 16 bytes. In the 'cenc' scheme, if an encrypted subsample has a length that is not a multiple of 16 bytes, then all the bytes of the encrypted subsample must be decrypted, but the next encrypted subsample will begin by completing the incomplete crypto block from the previous encrypted subsample. **The following diagram provides an example:**



To help with this, the block_offset field of each subsample will contain the number of bytes the initial crypto block of that subsample should be offset by. In the example above, the block_offset for the first subsample would be 0 and the block_offset for the second subsample would be 12. 'cenc' is the only mode that allows for a nonzero block_offset. This field satisfies $0 \leq block\_offset < 16$.

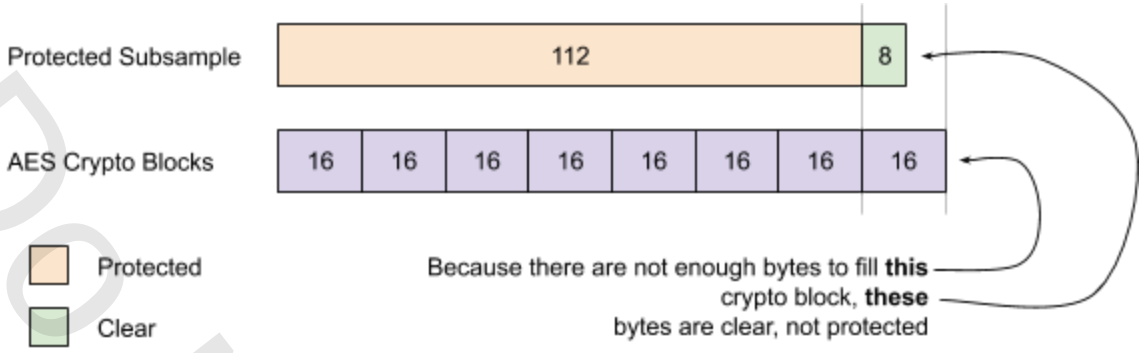
'cbcs' SCHEME:

The 'cbcs' scheme is OEMCrypto_CipherMode_CBC with an encryption pattern. Only some of the bytes in the encrypted portion of each subsample are encrypted. In the pattern parameter, the encrypt and skip fields will usually be non-zero. This mode allows devices to decrypt FMP4 HLS content, SAMPLE-AES HLS content, as well as content using the DASH 'cbcs' scheme.

The skip field of OEMCrypto_CENCEncryptPatternDesc may also be zero. If the skip field is zero, then patterns are not in use and all crypto blocks in the encrypted part of the subsample are encrypted. It is not valid for the encrypt field to be zero.

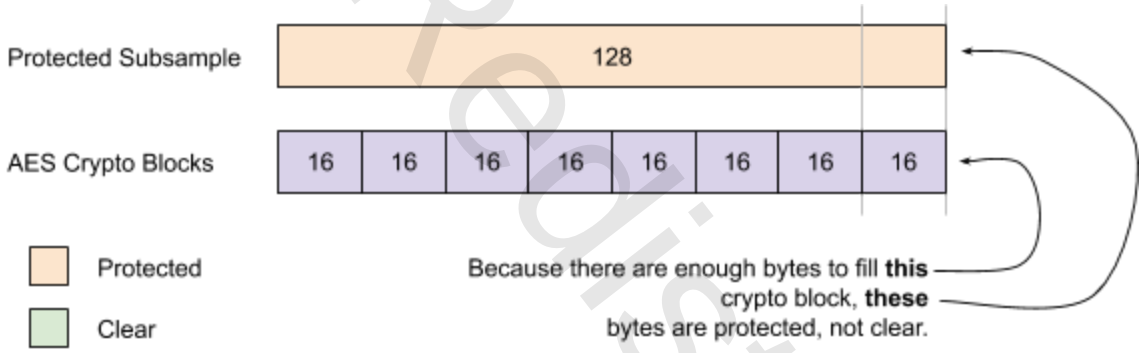
The length of a crypto block in AES-128 is 16 bytes. In the 'cbcs' scheme, if the encrypted part of a subsample has a length that is not a multiple of 16 bytes, then the final bytes that do not make up a full crypto block are clear and should never be decrypted. **The following diagram**

provides an example:



Whether any given protected block is actually encrypted also depends on the pattern. But the bytes at the end that do not make up a full crypto block will *never* be encrypted, regardless of what the pattern is. Even if the pattern says to decrypt every protected block, these bytes are clear and should not be decrypted.

Of course, if the encrypted subsample has a length that *is* a multiple of 16 bytes, all the bytes in it *are* protected, and they may need to be decrypted following the pattern. **The following diagram provides an example:**



INITIALIZATION VECTOR BETWEEN SUBSAMPLES:

The IV is specified for the initial subsample in a sample in the `iv` field of the `OEMCrypto_SampleDescription`. OEMCrypto is responsible for correctly updating the IV for subsequent subsamples according to the ISO Common Encryption standard (**ISO/IEC 23001-7:2016**). Section 9.5.2.3 covers 'cenc' and section 9.5.2.5 covers 'cbcs'. A summary of the ISO-CENC behavior follows:

For ‘cenc’, the IV at the end of each subsample carries forward to the next subsample and becomes the IV at the beginning of the next subsample. If the subsample ends on a crypto block boundary, then the IV should be incremented as normal at the end of the crypto block. If the subsample ends in the middle of a crypto block, the same IV should continue to be used until the crypto block is completed in the next subsample. Only increment the IV after the partial crypto block is completed.

For ‘cbcs’, the IV is reset at the beginning of each subsample. Each subsample should start with

the IV that was passed into `OEMCrypto_DecryptCENC`.

To phrase it another way: In ‘cenc’, the encrypted portions of the subsamples can be concatenated to form one continuous ciphertext. In ‘cbcs’, each encrypted portion of a subsample is a separate ciphertext. Each separate ciphertext begins with the IV specified in the `iv` field of the `OEMCrypto_SampleDescription`.

INITIALIZATION VECTOR WITHIN SUBSAMPLES:

Once it has the IV for each subsample, OEMCrypto is responsible for correctly updating the IV for each crypto block of each encrypted subsample portion, as outlined in the ISO Common Encryption standard (**ISO/IEC 23001-7:2016**). Section 9.5.1 includes general information about IVs in subsample decryption. A summary of the ISO-CENC behavior follows:

For ‘cenc’, the subsample’s IV is the counter value to be used for the initial encrypted block of the subsample. The IV length is the AES block size. For subsequent encrypted AES blocks, OEMCrypto must calculate the IV by incrementing the lower 64 bits (byte 8-15) of the IV value used for the previous block. The counter rolls over to zero when it reaches its maximum value (0xFFFFFFFFFFFFFFFF). The upper 64 bits (byte 0-7) of the IV do not change.

For ‘cbcs’, the subsample’s IV is the initialization vector for the initial encrypted block of the subsample. Within each subsample, each crypto block is used as the IV for the next crypto block, as prescribed by AES-CBC.

NOTES:

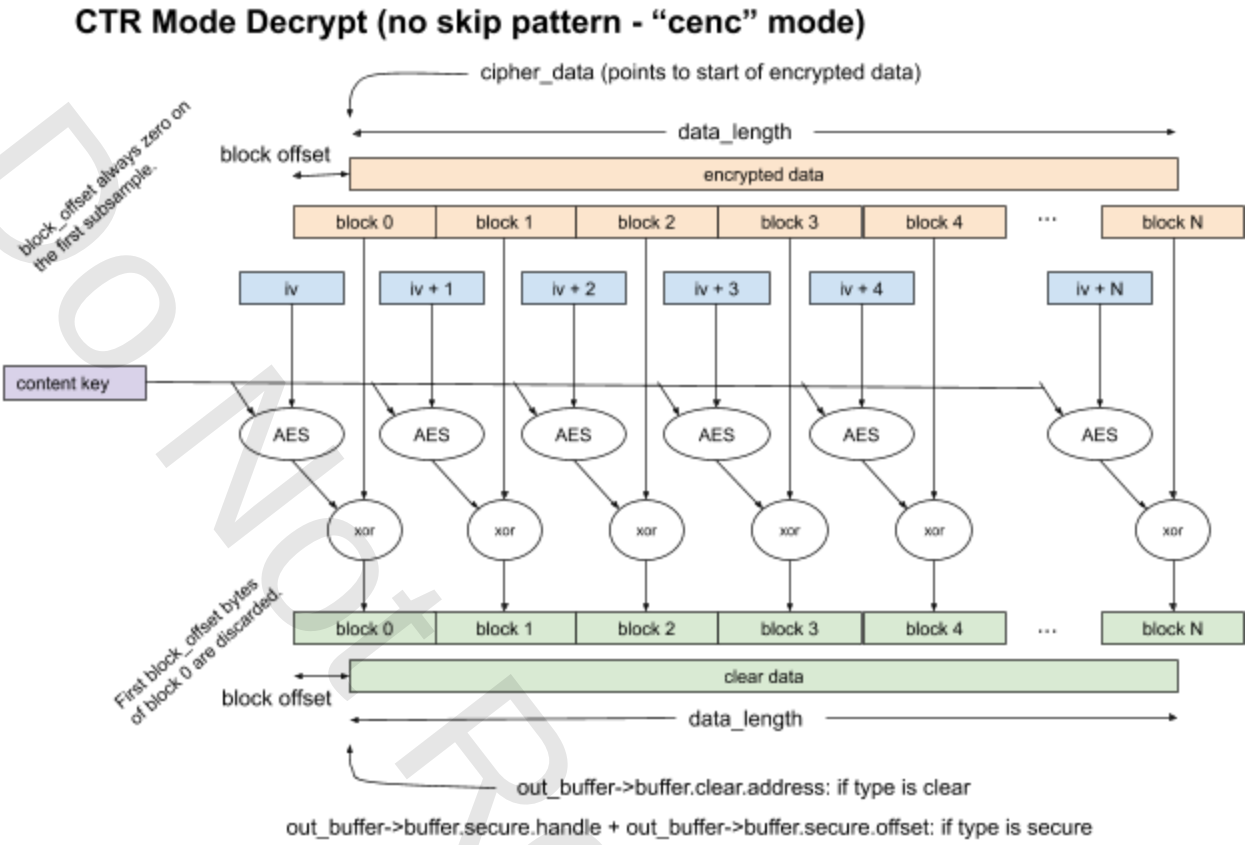
If the destination buffer is secure, an offset may be specified. `OEMCrypto_DecryptCENC` begins storing data `buffers.output.secure.offset` bytes after the beginning of the secure buffer.

If the session has an entry in the Usage Table, then OEMCrypto must update the `time_of_last_decrypt`. If the status of the entry is “unused”, then change the status to “active” and set the `time_of_first_decrypt`.

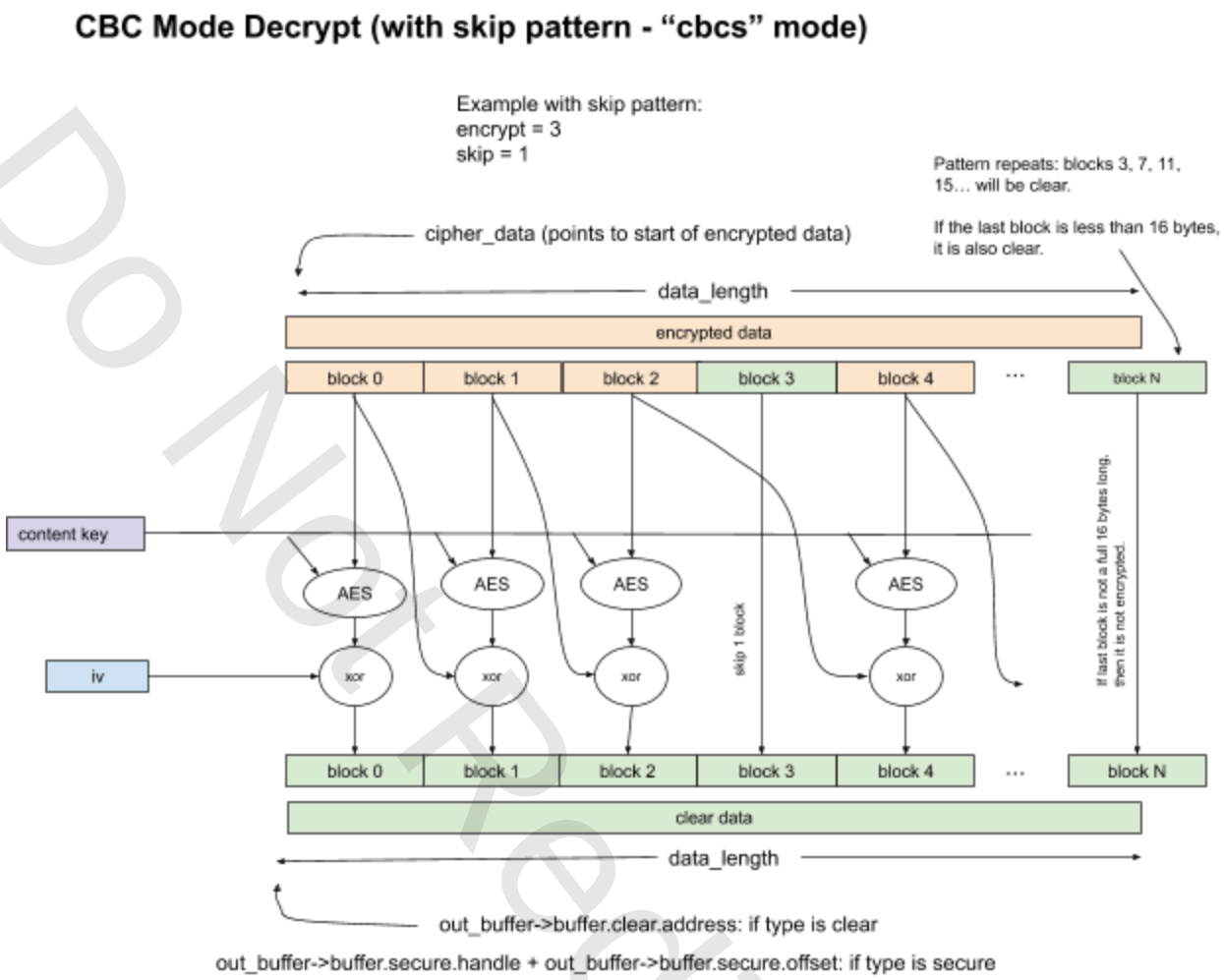
OEMCrypto cannot assume that the buffers of consecutive samples are consecutive in memory.

A subsample may consist entirely of encrypted bytes or clear bytes. In this case, the clear or the encrypted part of the subsample will be zero, indicating that no bytes of that kind appear in the subsample.

The ISO-CENC spec implicitly limits both the skip and encrypt values to be 4 bits, so they are at most 15.



If OEMCrypto assembles all of the encrypted subsample portions into a single buffer and then decrypts it in one pass, it can assume that the block offset is 0.



Verification

The total size of all the subsamples cannot exceed the total size of the input buffer. OEMCrypto integrations should validate this and return `OEMCrypto_ERROR_UNKNOWN_FAILURE` if the subsamples are larger than the input buffer. No decryption should be performed in this case.

If the subsamples all contain only clear bytes, then no further verification is performed. This call shall copy clear data even when there are no keys loaded, or there is no selected key.

If this is the first use of a key for this session, then OEMCrypto shall call ODK_AttemptFirstPlayback to update the session’s clock values and verify playback is allowed. If this is not the first use of a key for this session, then OEMCrypto shall call ODK_UpdateLastPlaybackTime. See the document “License Duration and Renewal” for handling the return value of these ODK functions.

The following checks should be performed if any subsamples contain any encrypted bytes. If any check fails, an error is returned, and no decryption is performed.

1. If the current key’s control block has the Data_Path_Type bit set, then the API shall verify that the output buffer is secure or direct. If not, return `OEMCrypto_ERROR_DECRYPT_FAILED`.

2. If the current key control block has the bit Disable_Analog_Output set, then the device should disable analog video output. If the device has analog video output that cannot be disabled, then `OEMCrypto_ERROR_ANALOG_OUTPUT` is returned. (See note on delayed error conditions below)
3. If the current key's control block has the HDCP bit set, then the API shall verify that the buffer will be displayed locally, or output externally using HDCP only. If not, return `OEMCrypto_ERROR_INSUFFICIENT_HDCP`. (See note on delayed error conditions below)
4. If the current key's control block has a nonzero value for HDCP_Version, then the current version of HDCP for the device and the display combined will be compared against the version specified in the control block. If the current version is not at least as high as that in the control block, and the device is not able to restrict displays with HDCP levels lower than what's in the control block, return OEMCrypto_ERROR_INSUFFICIENT_HDCP. If the device is able to restrict those displays, return OEMCrypto_WARNING_MIXED_OUTPUT_PROTECTION. (See note on delayed error conditions below)
5. If the current session has an entry in the Usage Table, and the status of that entry is either kInactiveUsed or kInactiveUnused, then return the error `OEMCrypto_ERROR_LICENSE_INACTIVE`.
6. If a Decrypt Hash has been initialized via `OEMCrypto_SetDecryptHash`, and the current key's control block does not have the Allow_Hash_Verification bit set, then do not compute a hash and return `OEMCrypto_ERROR_UNKNOWN_FAILURE`.

Delayed Error Conditions

On some devices, the HDCP subsystem is not directly connected to the OEMCrypto TA. This means that returning the error OEMCrypto_ERROR_INSUFFICIENT_HDCP at the time of the decrypt call is a performance hit. However, some devices have the ability to tag output buffers with security requirements, such as the required HDCP level.

For those devices, when a call to OEMCrypto_DecryptCENC is made using a key that requires HDCP output, and if the HDCP level on the output does not meet the required level.

- OEMCrypto may tag the output buffer as requiring HDCP at the required level and return OEMCrypto_SUCCESS.
- Output shall not be sent to the display.
- On the second or third call to OEMCrypto_DecryptCENC with the same key, OEMCrypto shall return OEMCrypto_ERROR_INSUFFICIENT_HDCP.

For those devices, when a call to OEMCrypto_DecryptCENC is made using a key that requires HDCP output, and if the HDCP level on some of the displays does not meet the required level.

- OEMCrypto may tag the output buffer as requiring HDCP at the required level and return OEMCrypto_SUCCESS.
- Output shall only be sent to the display with sufficient output control, e.g. the local display.
- On the second or third call to OEMCrypto_DecryptCENC with the same key, OEMCrypto shall return OEMCrypto_WARNING_MIXED_OUTPUT_PROTECTION.

In either case, a call to OEMCrypto_GetHDCPCapability shall return the current HDCP level.

Parameters

[in] session: Crypto session identifier. The crypto session in which decrypt is to be performed.

[in] samples: A caller-owned array of `OEMCrypto_SampleDescription` structures. Each entry in this array contains one sample of the content.

[in] samples_length: The length of the array pointed to by the `samples` parameter.

[in] pattern: A caller-owned structure indicating the encrypt/skip pattern as specified in the ISO-CENC standard.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_NO_DEVICE_KEY

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_DECRYPT_FAILED

OEMCrypto_ERROR_KEY_EXPIRED

OEMCrypto_ERROR_INSUFFICIENT_HDCP

OEMCrypto_ERROR_ANALOG_OUTPUT

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_OUTPUT_TOO_LARGE

OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support subsample sizes and total input buffer sizes as specified by its resource rating tier.

OEMCrypto shall return `OEMCrypto_ERROR_BUFFER_TOO_LARGE` if the buffer is larger than the supported size. If OEMCrypto returns `OEMCrypto_ERROR_BUFFER_TOO_LARGE`, the CDM will break the buffer into smaller chunks. For high performance devices, OEMCrypto should handle larger buffers. We encourage OEMCrypto implementers not to artificially restrict the maximum buffer size.

If OEMCrypto detects that the output data is too large, and breaking the buffer into smaller subsamples will not work, then it returns `OEMCrypto_ERROR_OUTPUT_TOO_LARGE`. This error will bubble up to the application, which can decide to skip the current frame of video or to switch to a lower resolution.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16. This method changed its name in API version 11.

OEMCrypto_DestBufferDesc Structure

```
typedef enum OEMCryptoBufferType {
    OEMCrypto_BufferType_Clear,
    OEMCrypto_BufferType_Secure,
    OEMCrypto_BufferType_Direct
} OEMCryptoBufferType;

typedef struct {
    OEMCryptoBufferType type;
    union {
        struct {                    // type == OEMCrypto_BufferType_Clear
            OEMCrypto_SharedMemory* address;
            size_t address_length;
        } clear;
        struct {                    // type == OEMCrypto_BufferType_Secure
            void* handle;
            size_t handle_length;
            size_t offset;
        } secure;
        struct {                    // type == OEMCrypto_BufferType_Direct
            bool is_video;
        } direct;
    } buffer;
} OEMCrypto_DestBufferDesc;
```

This structure is used as parameters in the OEMCrypto_DecryptCENC and OEMCrypto_CopyBuffer functions. This describes the type and access information for the memory to receive decrypted data.

The OEMCrypto API supports a range of client device architectures. Different architectures have different methods for acquiring and securing buffers that will hold portions of the audio or video stream after decryption. Three basic strategies are recognized for handling decrypted stream data:

1. Return the decrypted data in the clear into normal user memory (ClearBuffer). The caller uses normal memory allocation methods to acquire a buffer, and supplies the memory address of the buffer in the descriptor.
2. Place the decrypted data into protected memory (SecureBuffer). The caller uses a platform-specific method to acquire the protected buffer and a user-memory handle that references it. The handle is supplied to the decrypt call in the descriptor. If the buffer is filled with several OEMCrypto calls, the same handle will be used, and the offset will be incremented to indicate where the next write should take place.

3. Place the decrypted data directly into the audio or video decoder fifo (Direct). The caller will use platform-specific methods to initialize the fifo and the decoders. The decrypted stream data is not accessible to the caller. This is used on some platforms only.

Fields

[in] type: A tag that indicates which variant of the union is valid for this instance of the structure.

[variant] clear: This variant is valid when the type is `OEMCrypto_BufferType_Clear`. This `OEMCrypto_DestBufferDesc` indicates output should be written to a clear buffer.

[in] address: A pointer to the address in memory to begin writing output.

[in] address_length: The length of the buffer that is available to contain output.

[variant] secure: This variant is valid when the type is `OEMCrypto_BufferType_Secure`. This `OEMCrypto_DestBufferDesc` indicates output should be written to a secure buffer. The decrypted output must never leave the secure area until it is output from the device.

[in] handle: An opaque handle to a secure buffer. The meaning of this handle is platform-specific.

[in] handle_length: The length of the data contained in the secure buffer.

[in] offset: An offset indicating where in the secure buffer to start writing data.

[variant] direct: This variant is valid when the type is `OEMCrypto_BufferType_Direct`. This `OEMCrypto_DestBufferDesc` indicates output should be written directly to the decoder.

[in] is_video: A flag indicating if the data is video and should be sent to the video decoder. If this is false, the data can be assumed to be audio and sent to the audio decoder.

Version

This struct changed in API version 16.

OEMCrypto_InputOutputPair Structure

```
typedef struct {
   const OEMCrypto_SharedMemory* input_data;  // source for encrypted data.
   size_t input_data_length;  // length of encrypted data.
   OEMCrypto_DestBufferDesc output_descriptor;  // destination for clear data.
} OEMCrypto_InputOutputPair;
```

This structure is used as parameters in the OEMCrypto_DecryptCENC function.

Fields

[in] input_data: An unaligned pointer to this sample from the stream.

[in] input_data_length: The length of this sample in the stream, in bytes.

[in] output: A caller-owned descriptor that specifies the handling of the decrypted byte stream. See `OEMCrypto_DestbufferDesc` for details.

Version

This struct changed in API version 16.

OEMCrypto_SubSampleDescription Structure

```
typedef struct {
    size_t num_bytes_clear;
    size_t num_bytes_encrypted;
    uint8_t subsample_flags;  // is this the first/last subsample in a sample?
    size_t block_offset;  // used for CTR “cenc” mode only.
} OEMCrypto_SubSampleDescription;

#define OEMCrypto_FirstSubsample 1
#define OEMCrypto_LastSubsample 2
```

This structure is used as parameters in the OEMCrypto_DecryptCENC function. In the DASH specification, a sample is composed of multiple samples, and each subsample is composed of two regions. The first region is clear unprotected data. We also call this clear data or unencrypted data. Immediately following the clear region is the protected region. The protected region is encrypted or encrypted with a pattern. The pattern and number of bytes that are encrypted in the protected region is discussed in this document when we talk about the function OEMCryptoDecryptCENC. For historic reasons, this document also calls the protected region the encrypted region.

Fields

[in] num_bytes_clear: The number of unprotected bytes in this subsample. The clear bytes come before the encrypted bytes.

[in] num_bytes_encrypted: The number of protected bytes in this subsample. The protected bytes come after the clear bytes.

[in] subsample_flags: bitwise flags indicating if this is the first, middle, or last subsample in a sample. 1 = first subsample, 2 = last subsample, 3 = both first and last subsample, 0 = neither first nor last subsample.

[in] block_offset: This will only be non-zero for the ‘cenc’ scheme. If it is non-zero, the decryption block boundary is different from the start of the data. block_offset should be subtracted from data to compute the starting address of the first decrypted block. The bytes between the decryption block start address and data are discarded after decryption. It does not adjust the beginning of the source or destination data. This parameter satisfies $0 \leq block\_offset < 16$.

Version

This struct changed in API version 16.

OEMCrypto_SampleDescription Structure

```
typedef struct {
    OEMCrypto_InputOutputPair buffers;  // The source and destination buffers.
    uint8_t iv[16];  // The IV for the initial subsample.
    const OEMCrypto_SubSampleDescription* subsamples;  // subsamples array.
    size_t subsamples_length;  // the number of subsamples in the sample.
} OEMCrypto_SampleDescription;
```

This structure is used as parameters in the OEMCrypto_DecryptCENC function.

Fields

[in] buffers: A structure containing information about the input and output buffers.

[in] iv: A 16-byte array containing the IV for the initial subsample of the sample.

[in] subsamples: A caller-owned array of OEMCrypto_`SubSampleDescription` structures. Each entry in this array describes one subsample in the sample.

[in] subsamples_length: The length of the array pointed to by the `subsamples` parameter.

Version

This struct changed in API version 16.

OEMCrypto_CENCEncryptPatternDesc Structure

```
typedef struct {
    size_t encrypt;  // number of 16 byte blocks to decrypt.
    size_t skip;  // number of 16 byte blocks to leave in clear.
} OEMCrypto_CENCEncryptPatternDesc;
```

This structure is used as parameters in the OEMCrypto_DecryptCENC function.

Fields

[in] encrypt: The number of 16-byte crypto blocks to encrypt.

[in] skip: The number of 16-byte crypto blocks to leave in the clear.

Version

This struct changed in API version 16.

OEMCrypto_CopyBuffer

```
OEMCryptoResult OEMCrypto_CopyBuffer(
                    OEMCrypto_SESSION session,
                    const OEMCrypto_SharedMemory* data_addr,
                    size_t data_addr_length,
                    const OEMCrypto_DestBufferDesc* out_buffer_descriptor,
                    uint8_t subsample_flags);
```

Copies the payload in the buffer referenced by the *data parameter into the buffer referenced by the out_buffer parameter. The data is simply copied. The definition of OEMCrypto_DestBufferDesc and subsample_flags are the same as in OEMCrypto_DecryptCENC, above.

The main difference between this and DecryptCENC is that this function does not need an open

session, and it may be called concurrently with other functions on a multithreaded system. In particular, an application will use this to copy the clear leader of a video to a secure buffer while the license request is being generated, sent to the server, and the response is being processed. This functionality is needed because an application may not have read or write access to a secure destination buffer.

NOTES:

This method may be called several times before the data is used. The first buffer in a chunk of data will have the OEMCrypto_FirstSubsample bit set in subsample_flags. The last buffer in a chunk of data will have the OEMCrypto_LastSubsample bit set in subsample_flags. The data will not be used until after OEMCrypto_LastSubsample has been set. If an implementation copies data immediately, it may ignore subsample_flags.

If the destination buffer is secure, an offset may be specified. CopyBuffer begins storing data out_buffer->secure.offset bytes after the beginning of the secure buffer.

Verification

The following checks should be performed.

1. If either data or out_buffer is null, return OEMCrypto_ERROR_INVALID_CONTEXT.

Parameters

[in] session: crypto session identifier.

[in] data_addr: An unaligned pointer to the buffer to be copied.

[in] data_addr_length: The length of the buffer, in bytes.

[in] out_buffer: A caller-owned descriptor that specifies the handling of the byte stream. See OEMCrypto_DestbufferDesc for details.

[in] subsample_flags: bitwise flags indicating if this is the first, middle, or last subsample in a chunk of data. 1 = first subsample, 2 = last subsample, 3 = both first and last subsample, 0 = neither first nor last subsample.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_OUTPUT_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support subsample sizes and total input buffer sizes as specified by its

resource rating tier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size. If OEMCrypto returns OEMCrypto_ERROR_BUFFER_TOO_LARGE, the calling function must break the buffer into smaller chunks. For high performance devices, OEMCrypto should handle larger buffers. We encourage OEMCrypto implementers not to artificially restrict the maximum buffer size.

If OEMCrypto detects that the output data is too large, and breaking the buffer into smaller subsamples will not work, then it returns OEMCrypto_ERROR_OUTPUT_TOO_LARGE. This error will bubble up to the application, which can decide to skip the current frame of video or to switch to a lower resolution.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is changed in API version 15.

OEMCrypto_Generic_Encrypt

```
OEMCryptoResult OEMCrypto_Generic_Encrypt(
                      OEMCrypto_SESSION session,
                      const OEMCrypto_SharedMemory* in_buffer,
                      size_t in_buffer_length,
                      const uint8_t* iv,
                       OEMCrypto_Algorithm algorithm,
                      OEMCrypto_SharedMemory* out_buffer);

typedef enum OEMCrypto_Algorithm {
    OEMCrypto_AES_CBC_128_NO_PADDING = 0,
    OEMCrypto_HMAC_SHA256            = 1,
} OEMCrypto_Algorithm;
```

This function encrypts a generic buffer of data using the current key.

If the session has an entry in the Usage Table, then OEMCrypto will update the time_of_last_decrypt. If the status of the entry is “unused”, then change the status to “active” and set the time_of_first_decrypt.

OEMCrypto shall be able to handle buffers at least 100 KiB long.

Verification

The following checks should be performed. If any check fails, an error is returned, and the data is not encrypted.

1. The control bit for the current key shall have the Allow_Encrypt set. If not, return OEMCrypto_ERROR_UNKNOWN_FAILURE.
2. If this is the first use of a key for this session, then OEMCrypto shall call ODK_AttemptFirstPlayback to update the session’s clock values and verify playback is

allowed. If this is not the first use of a key for this session, then OEMCrypto shall call ODK_UpdateLastPlaybackTime. See the document “License Duration and Renewal” for handling the return value of these ODK functions.

3. If the current session has an entry in the Usage Table, and the status of that entry is either kInactiveUsed or kInactiveUnused, then return the error OEMCrypto_ERROR_LICENSE_INACTIVE.

Parameters

[in] session: crypto session identifier.
[in] in_buffer: pointer to memory containing data to be encrypted.
[in] in_buffer_length: length of the buffer, in bytes. The algorithm may restrict in_buffer_length to be a multiple of block size.
[in] iv: IV for encrypting data. Size is 128 bits.
[in] algorithm: Specifies which encryption algorithm to use. Currently, only CBC 128 mode is allowed for encryption.
[out] out_buffer: pointer to buffer in which encrypted data should be stored.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_KEY_EXPIRED
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED
OEMCrypto_ERROR_NOT_IMPLEMENTED

Buffer Sizes

OEMCrypto shall support buffers sizes of at least 100 KiB for generic crypto operations.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_Generic_Decrypt

```
OEMCryptoResult OEMCrypto_Generic_Decrypt(
```

```
    OEMCrypto_SESSION session,
    const OEMCrypto_SharedMemory* in_buffer,
    size_t in_buffer_length,
    const uint8_t* iv,
     OEMCrypto_Algorithm algorithm,
    OEMCrypto_SharedMemory* out_buffer);
```

This function decrypts a generic buffer of data using the current key.

If the session has an entry in the Usage Table, then OEMCrypto will update the time_of_last_decrypt. If the status of the entry is “unused”, then change the status to “active” and set the time_of_first_decrypt.

OEMCrypto should be able to handle buffers at least 100 KiB long.

Verification

The following checks should be performed. If any check fails, an error is returned, and the data is not decrypted.

1. The control bit for the current key shall have the Allow_Decrypt set. If not, return OEMCrypto_ERROR_DECRYPT_FAILED.
2. If the current key’s control block has the Data_Path_Type bit set, then return OEMCrypto_ERROR_DECRYPT_FAILED.
3. If this is the first use of a key for this session, then OEMCrypto shall call ODK_AttemptFirstPlayback to update the session’s clock values and verify playback is allowed. If this is not the first use of a key for this session, then OEMCrypto shall call ODK_UpdateLastPlaybackTime. See the document “License Duration and Renewal” for handling the return value of these ODK functions.
4. If the current session has an entry in the Usage Table, and the status of that entry is either kInactiveUsed or kInactiveUnused, then return the error OEMCrypto_ERROR_LICENSE_INACTIVE.

Parameters

[in] session: crypto session identifier.
[in] in_buffer: pointer to memory containing data to be encrypted.
[in] in_buffer_length: length of the buffer, in bytes. The algorithm may restrict in_buffer_length to be a multiple of block size.
[in] iv: IV for encrypting data. Size is 128 bits.
[in] algorithm: Specifies which encryption algorithm to use. Currently, only CBC 128 mode is allowed for decryption.
[out] out_buffer: pointer to buffer in which decrypted data should be stored.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_KEY_EXPIRED
OEMCrypto_ERROR_DECRYPT_FAILED
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED
OEMCrypto_ERROR_NOT_IMPLEMENTED

Buffer Sizes

OEMCrypto shall support buffers sizes of at least 100 KiB for generic crypto operations.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_Generic_Sign

```
OEMCryptoResult OEMCrypto_Generic_Sign(
                      OEMCrypto_SESSION session,
                      const OEMCrypto_SharedMemory* buffer,
                      size_t buffer_length,
                      OEMCrypto_Algorithm algorithm,
                      OEMCrypto_SharedMemory* signature,
                       size_t* signature_length);
```

This function signs a generic buffer of data using the current key.

If the session has an entry in the Usage Table, then OEMCrypto will update the time_of_last_decrypt. If the status of the entry is “unused”, then change the status to “active” and set the time_of_first_decrypt.

Verification

The following checks should be performed. If any check fails, an error is returned, and the data is not signed.

1. The control bit for the current key shall have the Allow_Sign set.
2. If this is the first use of a key for this session, then OEMCrypto shall call ODK_AttemptFirstPlayback to update the session’s clock values and verify playback is allowed. If this is not the first use of a key for this session, then OEMCrypto shall call ODK_UpdateLastPlaybackTime. See the document “License Duration and Renewal” for handling the return value of these ODK functions.
3. If the current session has an entry in the Usage Table, and the status of that entry is either kInactiveUsed or kInactiveUnused, then return the error OEMCrypto_ERROR_LICENSE_INACTIVE.

Parameters

[in] session: crypto session identifier.
[in] buffer: pointer to memory containing data to be encrypted.
[in] buffer_length: length of the buffer, in bytes.
[in] algorithm: Specifies which algorithm to use.
[out] signature: pointer to buffer in which signature should be stored. May be null on the first call in order to find required buffer size.
[in/out] signature_length: (in) length of the signature buffer, in bytes.
(out) actual length of the signature

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_KEY_EXPIRED
OEMCrypto_ERROR_SHORT_BUFFER if signature buffer is not large enough to hold the output signature.
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED
OEMCrypto_ERROR_NOT_IMPLEMENTED

Buffer Sizes

OEMCrypto shall support buffers sizes of at least 100 KiB for generic crypto operations.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_Generic_Verify

```
OEMCryptoResult OEMCrypto_Generic_Verify(
                      OEMCrypto_SESSION session,
                      const OEMCrypto_SharedMemory* buffer,
                      size_t buffer_length,
                       OEMCrypto_Algorithm algorithm,
                      const OEMCrypto_SharedMemory* signature,
                       size_t signature_length);
```

This function verifies the signature of a generic buffer of data using the current key.

If the session has an entry in the Usage Table, then OEMCrypto will update the time_of_last_decrypt. If the status of the entry is “unused”, then change the status to “active” and set the time_of_first_decrypt.

Verification

The following checks should be performed. If any check fails, an error is returned.

1. The control bit for the current key shall have the Allow_Verify set.
2. The signature of the message shall be computed, and the API shall verify the computed signature matches the signature passed in. If not, return OEMCrypto_ERROR_SIGNATURE_FAILURE.
3. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison).
4. If this is the first use of a key for this session, then OEMCrypto shall call ODK_AttemptFirstPlayback to update the session’s clock values and verify playback is allowed. If this is not the first use of a key for this session, then OEMCrypto shall call ODK_UpdateLastPlaybackTime. See the document “License Duration and Renewal” for handling the return value of these ODK functions.
5. If the current session has an entry in the Usage Table, and the status of that entry is either kInactiveUsed or kInactiveUnused, then return the error OEMCrypto_ERROR_LICENSE_INACTIVE.

Parameters

[in] session: crypto session identifier.
[in] buffer: pointer to memory containing data to be encrypted.
[in] buffer_length: length of the buffer, in bytes.
[in] algorithm: Specifies which algorithm to use.
[in] signature: pointer to buffer in which signature resides.
[in] signature_length: length of the signature buffer, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_KEY_EXPIRED
OEMCrypto_ERROR_SIGNATURE_FAILURE
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED
OEMCrypto_ERROR_NOT_IMPLEMENTED

Buffer Sizes

OEMCrypto shall support buffers sizes of at least 100 KiB for generic crypto operations.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

Factory Provisioning API

The OEMCrypto API allows for a device to be initially provisioned with a keybox or with an OEM certificate. See the Provisioning above. This section discusses functions used to install the root of trust.

Widevine keyboxes are used to establish a root of trust to secure content on a device that uses Provisioning 2.0. OEM Certificates are used to establish a root of trust to secure content on a device that uses Provisioning 3.0. Factory Provisioning a device is related to manufacturing methods. This section describes the API that installs the Widevine Keybox and the recommended methods for the OEM’s factory provisioning procedure.

Starting with API version 10, devices should have two keyboxes. One is the production keybox which may be installed in the factory, or using OEMCrypto_WrapKeyboxOrOEMCert and OEMCrypto_InstallKeyboxOrOEMCert as described below. The second keybox is a test keybox. The test keybox is the same for all devices and is used for a suite of unit tests. The test keybox will only be used temporarily while the unit tests are running, and will not be used by the general public. After the unit tests have been run, and OEMCrypto_Terminate has been called, the production keybox should be active again.

API functions marked as optional may be used by the OEM’s factory provisioning procedure and implemented in the library, but are not called from the Widevine DRM Plugin during normal operation. The following list shows the APIs required for devices using keybox provisioning:

- OEMCrypto_WrapKeyboxOrOEMCert- optional - only used by factory setup tools.
- OEMCrypto_InstallKeyboxOrOEMCert - optional - only used on some platforms.
- OEMCrypto_GetProvisioningMethod - required for keybox or oem cert. (provisioning 2.0 and 3.0)
- OEMCrypto_IsKeyboxOrOEMCertValid- required (provisioning 2.0 and 3.0)
- OEMCrypto_GetDeviceID- required (provisioning 2.0 and 3.0)

OEMCrypto_WrapKeyboxOrOEMCert

```
OEMCryptoResult OEMCrypto_WrapKeyboxOrOEMCert(
                        const uint8_t *keybox_or_cert,
                        size_t keybox_or_cert_length,
```

```
uint8_t *wrapped_keybox_or_cert,
size_t *wrapped_keybox_or_cert_length,
const uint8_t *transport_key,
size_t transport_key_length);
```

A device should be provisioned at the factory with either an OEM Certificate or a keybox. We will call this data the root of trust. During manufacturing, the root of trust should be encrypted with the OEM root key and stored on the file system in a region that will not be erased during factory reset. This function may be used by legacy systems that use the two-step WrapKeyboxOrOEMCert/InstallKeyboxOrOEMCert approach. When the Widevine DRM plugin initializes, it will look for a wrapped root of trust in the file /factory/wv.keys and install it into the security processor by calling OEMCrypto_InstallKeyboxOrOEMCert().

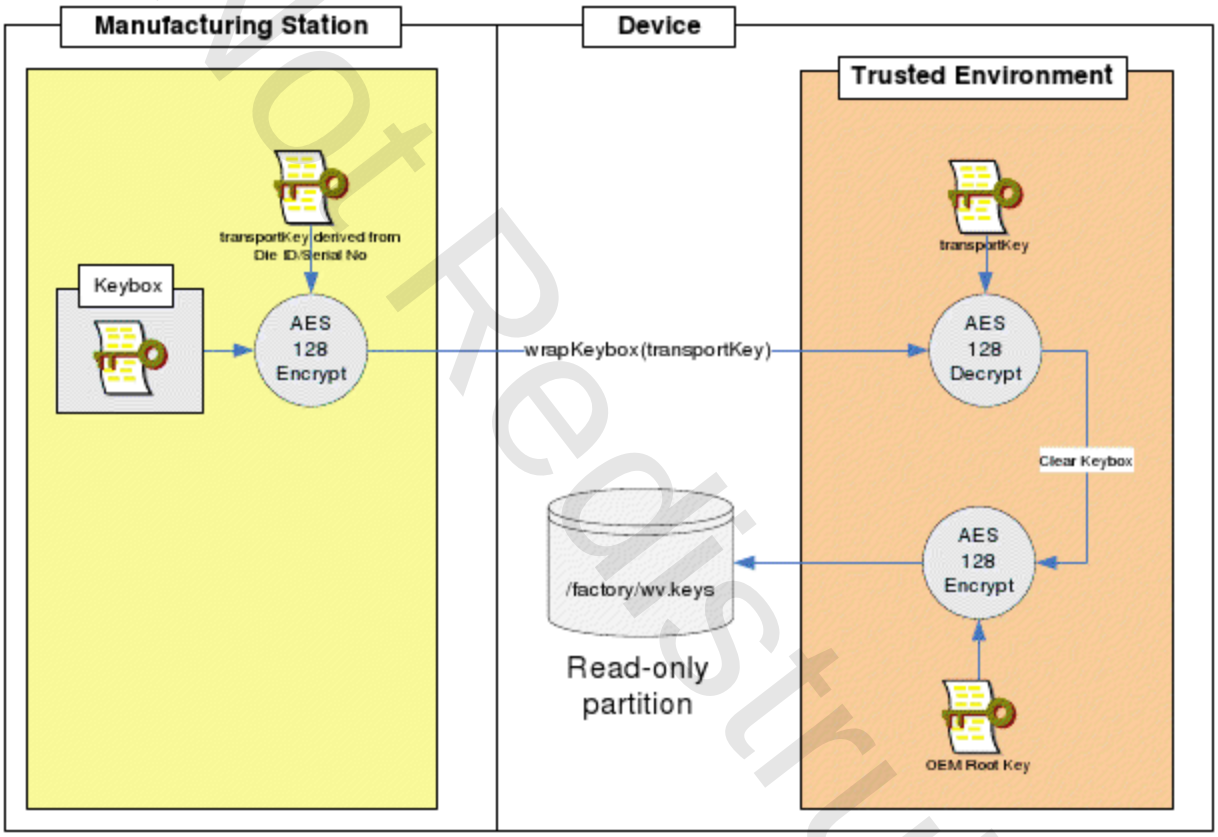


Figure 10. OEMCrypto_WrapKeyboxOrOEMCert Operation

OEMCrypto_WrapKeyboxOrOEMCert() is used to generate an OEM-encrypted root of trust that may be passed to OEMCrypto_InstallKeyboxOrOEMCert() for provisioning. The root of trust may be either passed in the clear or previously encrypted with a transport key. If a transport key is supplied, the keybox is first decrypted with the transport key before being wrapped with the OEM root key. **This function is only needed if the root of trust provisioning method involves saving the keybox or OEM Certificate to the file system.**

Parameters

[in] keybox_or_cert - pointer to root of trust data to encrypt -- this is either a keybox or an OEM

Certificate private key. May be NULL on the first call to test size of wrapped keybox. The keybox may either be clear or previously encrypted.

[in] keybox_or_cert_length - length the keybox or cert data in bytes
[out] wrapped_keybox_or_cert – Pointer to wrapped keybox or cert
[out] wrapped_keybox_or_cert_length – Pointer to the length of the wrapped keybox or certificate key in bytes
[in] transport_key – Optional. AES transport key. If provided, the keybox_or_cert parameter was previously encrypted with this key. The keybox will be decrypted with the transport key using AES-CBC and a null IV.
[in] transport_key_length – Optional. Number of bytes in the transport_key, if used.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_WRITE_KEYBOX failed to encrypt the keybox

OEMCrypto_ERROR_SHORT_BUFFER if keybox is provided as NULL, to determine the size of the wrapped keybox

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_NOT_IMPLEMENTED

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method is supported in all API versions.

OEMCrypto_InstallKeyboxOrOEMCert

```
OEMCryptoResult OEMCrypto_InstallKeyboxOrOEMCert(
         const uint8_t *keybox_or_cert, size_t keybox_or_cert_length);
```

Decrypts a wrapped root of trust and installs it in the security processor. The root of trust is unwrapped then encrypted with the OEM root key. This function is called from the Widevine DRM plugin at initialization time if there is no valid root of trust installed. It looks for wrapped data in the file /factory/wv.keys and if it is present, will read the file and call OEMCrypto_InstallKeyboxOrOEMCert() with the contents of the file. **This function is only needed if the factory provisioning method involves saving the keybox or OEM Certificate to the file system.**

Parameters

[in] keybox_or_cert - pointer to encrypted data as input

[in] keybox_or_cert_length - length of the data in bytes

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_BAD_MAGIC

OEMCrypto_ERROR_BAD_CRC

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_NOT_IMPLEMENTED

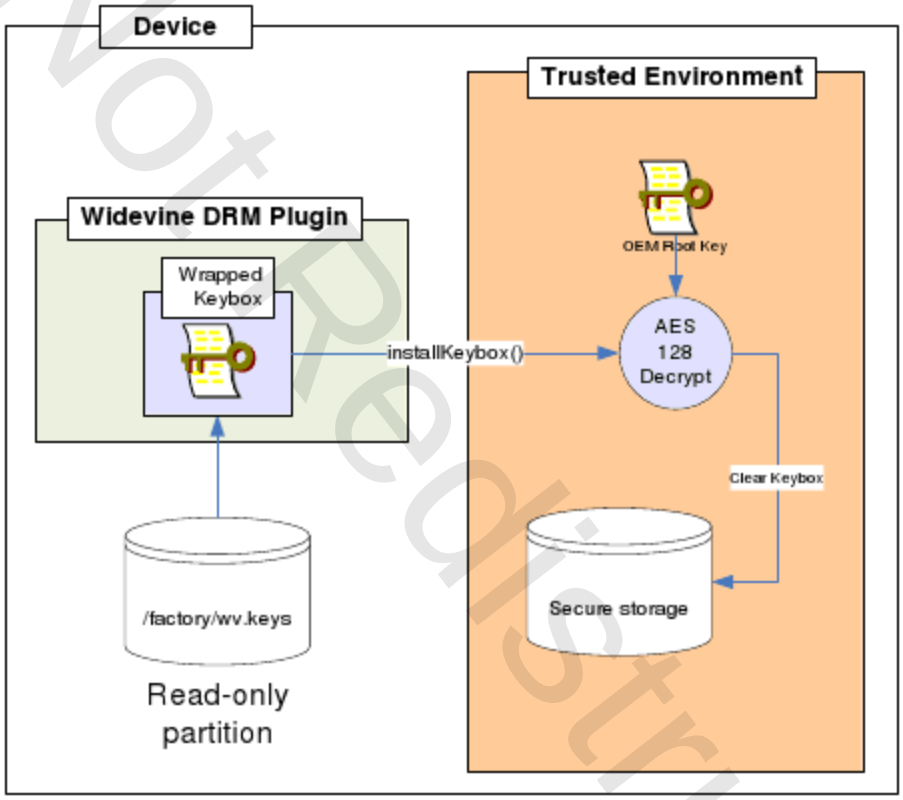
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method is supported in all API versions.



OEMCrypto_GetProvisioningMethod

```
OEMCrypto_ProvisioningMethod OEMCrypto_GetProvisioningMethod(void);

typedef enum OEMCrypto_ProvisioningMethod {
      OEMCrypto_ProvisioningError = 0,
      OEMCrypto_DrmCertificate = 1,
      OEMCrypto_Keybox = 2,
      OEMCrypto_OEMCertificate = 3
} OEMCrypto_ProvisioningMethod;
```

This function is for OEMCrypto to tell the layer above what provisioning method it uses: keybox

or OEM certificate.

Parameters
none

Returns
- **DrmCertificate** means the device has a DRM certificate built into the system. This cannot be used by level 1 devices. This provisioning method is deprecated and should not be used on new devices. OEMCertificate provisioning should be used instead.
- **Keybox** means the device has a unique keybox. For level 1 devices this keybox must be securely installed by the device manufacturer.
- **OEMCertificate** means the device has a factory installed OEM certificate. This is also called Provisioning 3.0.
- **ProvisioningError** indicates a serious problem with the OEMCrypto library.

Threading
This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version
This method is new API version 12.

OEMCrypto_IsKeyboxOrOEMCertValid

```
OEMCryptoResult OEMCrypto_IsKeyboxOrOEMCertValid(void);
```

If the device has a keybox, this validates the Widevine Keybox loaded into the security processor device. This method verifies two fields in the keybox:

- Verify the MAGIC field contains a valid signature (such as, ‘k’’b’’o’’x’).
- Compute the CRC using CRC-32-POSIX-1003.2 standard and compare the checksum to the CRC stored in the Keybox.

The CRC is computed over the entire Keybox excluding the 4 bytes of the CRC (for example, Keybox[0..123]). For a description of the fields stored in the keybox, see Keybox Definition.

If the device has an OEM Certificate, this validates the certificate private key.

Parameters
none

Returns
OEMCrypto_SUCCESS
OEMCrypto_ERROR_BAD_MAGIC
OEMCrypto_ERROR_BAD_CRC
OEMCrypto_ERROR_KEYBOX_INVALID
OEMCrypto_ERROR_INVALID_RSA_KEY
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is supported in all API versions.

OEMCrypto_GetDeviceID

```
OEMCryptoResult OEMCrypto_GetDeviceID(
        uint8_t* device_id,
        size_t* device_id_length);
```

Retrieve DeviceID from the Keybox. For devices that have an OEM Certificate instead of a keybox, this function may return OEMCrypto_ERROR_NOT_IMPLEMENTED. If the function is implemented on an OEM Certificate device, it should set the device ID to a device-unique string, such as the device serial number. The ID should be device-unique and it should be stable -- i.e. it should not change across a device reboot or a system upgrade. This shall match the device id found in the core provisioning request message.

Parameters

[out] device_id - pointer to the buffer that receives the Device ID

[in/out] device_id_length – on input, size of the caller’s device ID buffer. On output, the number of bytes written into the buffer.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_SHORT_BUFFER if the buffer is too small to return device ID

OEMCrypto_ERROR_NO_DEVICEID failed to return Device Id

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is supported in all API versions.

Keybox and Provisioning 2.0 API

The OEMCrypto API allows for a device to be initially provisioned with a keybox or with an OEM certificate. See the section Provisioning above. In a Level 1 or Level 2 implementation, only the security processor may access the keys in the keybox. The following functions are for devices that are provisioned with a keybox, i.e. Provisioning 2.0.

OEMCrypto_GetKeyData

```
OEMCryptoResult OEMCrypto_GetKeyData(
        uint8_t* key_data, size_t *key_data_length);
```

Return the Key Data field from the Keybox.

Parameters

[out] keyData - pointer to the buffer to hold the Key Data field from the Keybox

[in/out] keyDataLength – on input, the allocated buffer size. On output, the number of bytes in Key Data

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_SHORT_BUFFER if the buffer is too small to return KeyData

OEMCrypto_ERROR_NO_KEYDATA

OEMCrypto_ERROR_NOT_IMPLEMENTED - this function is for Provisioning 2.0 only.

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is supported in all API versions.

OEMCrypto_LoadTestKeybox

```
OEMCryptoResult OEMCrypto_LoadTestKeybox(const uint8_t *buffer,
                                         size_t buffer_length);
```

Temporarily use the specified test keybox until the next call to OEMCrypto_Terminate. This allows a standard suite of unit tests to be run on a production device without permanently changing the keybox. Using the test keybox is ***not*** persistent. OEMCrypto **cannot** assume that this keybox is the same as previous keyboxes used for testing.

Devices that use an OEM Certificate instead of a keybox (i.e. Provisioning 3.0) do not need to support this functionality, and may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Parameters

[in] buffer: pointer to memory containing test keybox, in binary form.

[in] buffer_length: length of the buffer, in bytes.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_NOT_IMPLEMENTED - this function is for Provisioning 2.0 only.

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system. It is called after OEMCrypto_Initialize and after OEMCrypto_GetProvisioningMethod and only if the provisoining method is OEMCrypto_Keybox,

Version

This method changed in API version 14.

OEM Certificate Access and Provisioning 3.0 API

The OEMCrypto API allows for a device to be initially provisioned with a keybox or with an OEM certificate. See the Provisioning above. The functions in this section are for devices that are provisioned with an OEM Certificate, i.e. Provisioning 3.0.

API functions marked as optional may be used by the OEM’s factory provisioning procedure and implemented in the library, but are not called from the Widevine DRM Plugin during normal operation. The following list shows the APIs required for devices using keybox provisioning:

OEMCrypto_LoadOEMPrivateKey

```
OEMCryptoResult OEMCrypto_LoadOEMPrivateKey(OEMCrypto_SESSION session);
```

After a call to this function, all session functions using an RSA key should use the OEM certificate’s private RSA key. See the section above discussing Provisioning 3.0.

Parameters

- [in] session: this function affects the specified session only.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_NOT_IMPLEMENTED - this function is for Provisioning 3.0 only.

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is new API version 16.

OEMCrypto_GetOEMPublicCertificate

```
OEMCryptoResult OEMCrypto_GetOEMPublicCertificate(
                                    uint8_t *public_cert,
                                    size_t *public_cert_length);
```

This function should place the OEM public certificate in the buffer public_cert. See the section above discussing Provisioning 3.0.

If the buffer is not large enough, OEMCrypto should update public_cert_length and return OEMCrypto_ERROR_SHORT_BUFFER.

Parameters

- [out] public_cert: the buffer where the public certificate is stored.
- [in/out] public_cert_length: on input, this is the available size of the buffer. On output, this is the number of bytes needed for the certificate.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_NOT_IMPLEMENTED - this function is for Provisioning 3.0 only.

OEMCrypto_ERROR_SHORT_BUFFER
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is new API version 16.

Validation and Feature Support API

The OEMCrypto API is flexible enough to allow different devices to support different features. This section has functions that specify the level of support for various features. These values are reported to either the application or the license server.

OEMCrypto_GetRandom

```
OEMCryptoResult OEMCrypto_GetRandom(
         uint8_t* random_data, size_t random_data_length);
```

Returns a buffer filled with hardware-generated random bytes, if supported by the hardware. If the hardware feature does not exist, return OEMCrypto_ERROR_RNG_NOT_SUPPORTED.

Parameters

[out] random_data- pointer to the buffer that receives random data

[in] random_data_length- length of the random data buffer in bytes

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_RNG_FAILED failed to generate random number

OEMCrypto_ERROR_RNG_NOT_SUPPORTED function not supported

OEMCrypto_ERROR_BUFFER_TOO_LARGE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support dataLength sizes of at least 32 bytes for random number generation.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is supported in all API versions.

OEMCrypto_APIVersion

```
uint32_t OEMCrypto_APIVersion(void);
```

This function returns the current API version number. The version number allows the calling application to avoid version mis-match errors, because this API is part of a shared library.

There is a possibility that some API methods will be backwards compatible, or backwards compatible at a reduced security level.

There is no plan to introduce forward-compatibility. Applications will reject a library with a newer version of the API.

The version specified in this document is 16. Any OEM that returns this version number guarantees it passes all unit tests associated with this version.

Parameters

none

Returns

The supported API, as specified in the header file OEMCryptoCENC.h.

Threading

This is a “Property Function” and may be called simultaneously with any other property function

or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in each API version.

OEMCrypto_MinorAPIVersion

```
uint32_t OEMCrypto_MinorAPIVersion(void);
```

This function returns the current API minor version number. The version number allows the calling application to avoid version mis-match errors, because this API is part of a shared library.

The minor version specified in this document is 2. Any OEM that returns this version number guarantees it passes all unit tests associated with this version.

Parameters

none

Returns

The supported API, as specified in the header file OEMCryptoCENC.h.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in each API version.

OEMCrypto_BuildInformation

```
const char* OEMCrypto_BuildInformation(void);
```

Report the build information of the OEMCrypto library as a short null terminated C string. The string should be at most 128 characters long. This string should be updated with each release or OEMCrypto build.

Some SOC vendors deliver a binary OEMCrypto library to a device manufacturer. This means the OEMCrypto version may not be exactly in sync with the system’s versions. This string can be used to help track which version is installed on a device.

It may be used for logging or bug tracking and may be bubbled up to the app so that it may track metrics on errors.

Since the OEMCrypto API also changes its minor version number when there are minor corrections, it would be useful to include the API version number in this string, e.g. “15.1” or “15.2” if those minor versions are released.

Parameters

none

Returns

A printable null terminated C string, suitable for a single line in a log.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in each API version.

OEMCrypto_Security_Patch_Level

```
uint8_t OEMCrypto_Security_Patch_Level(void);
```

This function returns the current patch level of the software running in the trusted environment. The patch level is defined by the OEM, and is only incremented when a security update has been added.

See the section Security Patch Level above for more details.

Parameters

none

Returns

The OEM defined version number.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method was introduced in API version 11.

OEMCrypto_SecurityLevel

```
const char* OEMCrypto_SecurityLevel(void);
```

Returns a string specifying the security level of the library.

Since this function is spoofable, it is not relied on for security purposes. It is for information only.

Parameters

none

Returns

A null terminated string. Useful value are “L1”, “L2” and “L3”.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 6.

OEMCrypto_GetHDCPCapability

```
OEMCryptoResult
OEMCrypto_GetHDCPCapability(OEMCrypto_HDCP_Capability *current,
                            OEMCrypto_HDCP_Capability *maximum);
```

Returns the maximum HDCP version supported by the device, and the HDCP version supported by the device and any connected display.

Valid values for HDCP_Capability are:

```
typedef enum OEMCrypto_HDCP_Capability {
  HDCP_NONE  = 0,  // No HDCP supported, no secure data path.
  HDCP_V1    = 1,  // HDCP version 1.0
  HDCP_V2    = 2,  // HDCP version 2.0 Type 1.
  HDCP_V2_1  = 3,  // HDCP version 2.1 Type 1.
  HDCP_V2_2  = 4,  // HDCP version 2.2 Type 1.
  HDCP_V2_3  = 5,  // HDCP version 2.3 Type 1.
  HDCP_NO_DIGITAL_OUTPUT = 0xff  // No digital output.
} OEMCrypto_HDCP_Capability;
```

The value 0xFF means the device is using a local, secure, data path instead of HDMI output. Notice that HDCP must use flag Type 1: all downstream devices will also use the same version or higher.

The maximum HDCP level should be the maximum value that the device can enforce. For example, if the device has an HDCP 1.0 port and an HDCP 2.0 port, and the first port can be disabled, then the maximum is HDCP 2.0. If the first port cannot be disabled, then the maximum is HDCP 1.0. The maximum value can be used by the application or server to decide if a license may be used in the future. For example, a device may be connected to an external display while an offline license is downloaded, but the user intends to view the content on a local display. The user will want to download the higher quality content.

The current HDCP level should be the level of HDCP currently negotiated with any connected receivers or repeaters either through HDMI or a supported wireless format. If multiple ports are connected, the current level should be the minimum HDCP level of all ports. If the key control block requires an HDCP level equal to or lower than the current HDCP level, the key is

expected to be usable. If the key control block requires a higher HDCP level, the key is expected to be forbidden.

When a key has version HDCP_V2_3 required in the key control block, the transmitter must have HDCP version 2.3 and have negotiated a connection with a version 2.2 or 2.3 receiver or repeater. The transmitter must configure the content stream to be Type 1. Since the transmitter cannot distinguish between 2.2 and 2.3 downstream receivers when connected to a repeater, it may transmit to both 2.2 and 2.3 receivers, but not 2.1 receivers.

For example, if the transmitter is 2.3, and is connected to a receiver that supports 2.3 then the current level is `HDCP_V2_3`. If the transmitter is 2.3 and is connected to a 2.3 repeater, the current level is `HDCP_V2_3` even though the repeater can negotiate a connection with a 2.2 downstream receiver for a Type 1 Content Stream.

As another example, if the transmitter can support 2.3, but a receiver supports 2.0, then the current level is `HDCP_V2`.

When a license requires HDCP, a device may use a wireless protocol to connect to a display only if that protocol supports the version of HDCP as required by the license. Both WirelessHD (formerly WiFi Display) and Miracast support HDCP.

Parameters

[out] current - this is the current HDCP version, based on the device itself, and the display to which it is connected.

[out] maximum - this is the maximum supported HDCP version for the device, ignoring any attached device.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 10.

OEMCrypto_SupportsUsageTable

```
bool OEMCrypto_SupportsUsageTable(void);
```

This is used to determine if the device can support a usage table. Since this function is spoofable, it is not relied on for security purposes. It is for information only. The usage table is

described in the section above.

Parameters

none

Returns

Returns true if the device can maintain a usage table. Returns false otherwise.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 9.

OEMCrypto_MaximumUsageTableHeaderSize

```
size_t OEMCrypto_MaximumUsageTableHeaderSize(void);
```

Estimates the maximum usage table size. If the device does not have a fixed size, this returns an estimate. A maximum size of 0 means the header is constrained only by dynamic memory allocation.

Widevine requires the size to be at least 300 entries.

Parameters

none

Returns

Returns an estimate for the maximum size of the usage table header.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_IsAntiRollbackHwPresent

```
bool OEMCrypto_IsAntiRollbackHwPresent(void);
```

Indicate whether there is hardware protection to detect and/or prevent the rollback of the usage table. For example, if the usage table contents is stored entirely on a secure file system that the

user cannot read or write to. Another example is if the usage table has a generation number and the generation number is stored in secure memory that is not user accessible.

Parameters

none

Returns

Returns true if oemcrypto uses anti-rollback hardware. Returns false otherwise.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is new in API version 10.

OEMCrypto_GetNumberOfOpenSessions

```
OEMCryptoResult OEMCrypto_GetNumberOfOpenSessions(size_t *count);
```

Returns the current number of open sessions. The CDM and OEMCrypto consumers can query this value so they can use resources more effectively.

Parameters

[out] count - this is the current number of opened sessions.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is new in API version 10.

OEMCrypto_GetMaxNumberOfSessions

```
OEMCryptoResult OEMCrypto_GetMaxNumberOfSessions(size_t *max);
```

Returns the maximum number of concurrent OEMCrypto sessions supported by the device. The CDM and OEMCrypto consumers can query this value so they can use resources more effectively. If the maximum number of sessions depends on a dynamically allocated shared resource, the returned value should be a best estimate of the maximum number of sessions.

OEMCrypto shall support a minimum of 10 sessions. Some applications use multiple sessions to pre-fetch licenses, so high end devices should support more sessions -- we recommend a minimum of 50 sessions.

Parameters

[out] max - this is the max number of supported sessions.

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 12.

OEMCrypto_SupportedCertificates

```
uint32_t OEMCrypto_SupportedCertificates(void);
```

Returns the type of certificates keys that this device supports. With very few exceptions, all devices should support at least 2048 bit RSA keys. High end devices should also support 3072 bit RSA keys. Devices that are cast receivers should also support RSA cast receiver certificates.

Beginning with OEMCrypto v14, the provisioning server may deliver to the device an RSA key that uses the Carmichael totient. This does not change the RSA algorithm -- however the product of the private and public keys is not necessarily the Euler number $\phi(n)$. OEMCrypto should not reject such keys.

Parameters

none

Returns

Returns the bitwise or of the following flags. It is likely that high end devices will support both 2048 and 3072 bit keys while the widevine servers transition to new key sizes.

- 0x1 = OEMCrypto_Supports_RSA_2048bit - the device can load a DRM certificate with a 2048 bit RSA key.
- 0x2 = OEMCrypto_Supports_RSA_3072bit - the device can load a DRM certificate with a 3072 bit RSA key.
- 0x10 = OEMCrypto_Supports_RSA_CAST - the device can load a CAST certificate. These certificates are used with OEMCrypto_GenerateRSASignature with padding type set to 0x2, PKCS1 with block type 1 padding.
- 0x100 = OEMCrypto_Supports_ECC_secp256r1 - Elliptic Curve secp256r1
- 0x200 = OEMCrypto_Supports_ECC_secp384r1 - Elliptic Curve secp384r1
- 0x200 = OEMCrypto_Supports_ECC_secp521r1 - Elliptic Curve secp521r1

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_IsSRMUpdateSupported

```
bool OEMCrypto_IsSRMUpdateSupported(void);
```

Returns true if the device supports SRM files and the file can be updated via the function OEMCrypto_LoadSRM. This also returns false for devices that do not support an SRM file, devices that do not support HDCP, and devices that have no external display support.

Parameters

none

Returns

true - if LoadSRM is supported.

false - otherwise.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_GetCurrentSRMVersion

```
OEMCryptoResult OEMCrypto_GetCurrentSRMVersion(uint16_t* version);
```

Returns the version number of the current SRM file. If the device does not support SRM files, this will return OEMCrypto_ERROR_NOT_IMPLEMENTED. If the device only supports local displays, it would return OEMCrypto_LOCAL_DISPLAY_ONLY. If the device has an SRM, but cannot use OEMCrypto to update the SRM, then this function would set version to be the current version number, and return OEMCrypto_SUCCESS, but it would return false from OEMCrypto_IsSRMUpdateSupported.

Parameters

[out] version: current SRM version number.

Returns

OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_SUCCESS
OEMCrypto_LOCAL_DISPLAY_ONLY - to indicate version was not set, and is not needed.
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_GetAnalogOutputFlags

```
uint32_t OEMCrypto_GetAnalogOutputFlags(void);
```

Returns whether the device supports analog output or not. This information will be sent to the license server, and may be used to determine the type of license allowed. This function is for reporting only. It is paired with the key control block flags Disable_Analog_Output and CGMS.

Parameters

none.

Returns

Returns a bitwise OR of the following flags.

- 0x0 = OEMCrypto_No_Analog_Output -- the device has no analog output.
- 0x1 = OEMCrypto_Supports_Analog_Output - the device does have analog output.
- 0x2 = OEMCrypto_Can_Disable_Analog_Ouptput - the device does have analog output, but it will disable analog output if required by the key control block.
- 0x4 = OEMCrypto_Supports_CGMS_A - the device supports signaling 2-bit CGMS-A, if required by the key control block

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is new in API version 14.

OEMCrypto_ResourceRatingTier

```
uint32_t OEMCrypto_ResourceRatingTier(void);
```

This function returns a positive number indicating which resource rating it supports. This value will bubble up to the application level as a property. This will allow applications to estimate what resolution and bandwidth the device is expected to support.

OEMCrypto unit tests and Android GTS tests will verify that devices do support the resource values specified in the table below at the tier claimed by the device. If a device claims to be a low end device, the OEMCrypto unit tests will only verify the low end performance values.

OEMCrypto implementers should consider the numbers below to be minimum values.

These performance parameters are for OEMCrypto only. In particular, bandwidth and codec resolution are determined by the platform.

Some parameters need more explanation. The Sample size is typically the size of one encoded frame, but might be several frames for AV1. Converting this to resolution depends on the Codec, which is not specified by OEMCrypto. Some content has the sample broken into several subsamples. The “number of subsamples” restriction requires that any content can be broken into at least that many subsamples. However, this number may be larger if DecryptCENC returns OEMCrypto_ERROR_BUFFER_TOO_LARGE. In that case, the layer above OEMCrypto will break the sample into subsamples of size “Decrypt Buffer Size” as specified in the table below. The “Decrypt Buffer Size” means the size of one subsample that may be passed into DecryptCENC or CopyBuffer without returning error OEMCrypto_ERROR_BUFFER_TOO_LARGE.

The minimum subsample buffer size is the smallest buffer that the CDM layer above OEMCrypto will use when breaking a sample into subsamples. As mentioned above, the CDM layer will only break a sample into smaller subsamples if OEMCrypto returns OEMCrypto_ERROR_BUFFER_TOO_LARGE. Because this might be a performance problem, OEMCrypto implementers are encouraged to process larger subsamples and to process multiple subsamples in a single call to DecryptCENC.

The number of keys per session is an indication of how many different track types there can be for a piece of content. Typically, content will have several keys corresponding to audio and video at different resolutions. If the content uses key rotation, there could be three keys -- previous interval, current interval, and next interval -- for each resolution.

Concurrent playback sessions versus concurrent sessions: some applications will preload multiple licenses before the user picks which content to play. Each of these licenses

corresponds to an open session. Once playback starts, some platforms support picture-in-picture or multiple displays. Each of these pictures would correspond to a separate playback session with active decryption.

The total number of keys for all sessions indicates that the device may share key memory over multiple sessions. For example, on a Tier 3 device, the device must support four sessions with 20 keys each (80 total), or 20 sessions with 4 keys each (80 total), but it does not need to support 20 sessions with 20 keys each.

The message size that is needed for a license with a large number of keys is larger than in previous versions. The message size limit applies to all functions that sign or verify messages. It also applies to the size of context buffers in the derive key functions.

Decrypted frames per second -- strictly speaking, OEMCrypto only controls the decryption part of playback and cannot control the decoding and display part. However, devices that support the higher resource tiers should also support a higher frame rate. Platforms may enforce these values. For example Android will enforce a frame rate via a GTS test.

Note on units: We will use KiB to mean 1024 bytes and MiB to mean 1024 KiB, as described at https://en.wikipedia.org/wiki/Kibibyte.

Resource Rating Tier	**1 - Low**	**2 - Medium**	**3 - High**	**4 - Very High**
Example Device	A low cost phone that only plays SD would probably be in this tier.	A high cost phone that plays SD or HD would probably be in this tier.	A UHD television or home entertainment device would probably be in this tier.	An 8k television or home entertainment device would probably be in this tier.
Minimum Sample size (see note above)	1 MiB	2 MiB	4 MiB	16 MiB
Minimum Number of Subsamples - H264 or HEVC (see note above)	10	16	32	64
Minimum Number of Subsamples - VP9 (see note above)	9	9	9	9
Minimum Number of Subsamples - AV1 (see note above)	72	144	288	576
Minimum subsample buffer size	100 KiB	500 KiB	1 MiB	4 MiB
Minimum Generic	10 KiB	100 KiB	500 KiB	1 MiB

crypto buffer size				
Minimum Number of concurrent sessions	10	20	30	40
Minimum Number of keys per session	4	20	20	30
Minimum Total Number of Keys (all sessions)	16	40	80	90
Message Size	8 KiB	8 KiB	16 KiB	32 KiB
Decrypted Frames per Second	30 fps SD	30 fps HD	60 fps HD	60 fps at 8k

Parameters

none.

Returns

Returns an integer indicating which resource tier the device supports.

Threading

This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is new in API version 15.

DRM Certificate Provisioning API

This section of functions are used to provision the device with an DRM certificate. This certificate is obtained by a device in the field from a Google/Widevine provisioning server, or from a third party server running the Google/Widevine provisioning server SDK. Since the DRM certificate may be origin or application specific, a device may have several DRM certificates installed at a time. The DRM certificate is used to authenticate the device to a license server. In order to obtain a DRM certificate from a provisioning server, the device may authenticate itself using a keybox or using an OEM certificate.

OEMCrypto_LoadProvisioning

```
OEMCryptoResult OEMCrypto_LoadProvisioning(OEMCrypto_SESSION session,
                                    const uint8_t* message,
                                    size_t message_length,
                                    size_t core_message_length,
```

```
                                    const uint8_t* signature,
                                    size_t signature_length,
                                    uint8_t* wrapped_private_key,
                                    size_t *wrapped_private_key_length);
```

Load and parse a provisioning response, and then rewrap the private key for storage on the filesystem. We recommend that the OEM use an encryption key and signing key generated using an algorithm at least as strong as that in GenerateDerivedKeys.

First, OEMCrypto shall verify the signature of the message using HMAC-SHA256 with the derived mac_key[server]. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison). The signature is over the entire message buffer starting at `message` with length `message_length`. If the signature verification fails, ignore all other arguments and return OEMCrypto_ERROR_SIGNATURE_FAILURE.

NOTE: The calling software must have previously established the mac_keys and encrypt_key with a call to OEMCrypto_DeriveKeysFromSessionKey or OEMCrypto_GenerateDerivedKeys.

The function ODK_ParseProvisioning is called to parse the message. If it returns an error, OEMCrypto shall return that error to the CDM layer. The function ODK_ParseProvisioning is described in the document “Widevine Core Message Serialization”.

Below, all fields are found in the struct ODK_ParsedLicense parsed_license returned by ODK_ParsedProvisioning.

After decrypting enc_rsa_key, If the first four bytes of the buffer are the string “SIGN”, then the actual RSA key begins on the 9th byte of the buffer. The second four bytes of the buffer is the 32 bit field “allowed_schemes”of type RSA_Padding_Scheme, which is used in OEMCrypto_GenerateRSASignature. The value of allowed_schemes must also be wrapped with RSA key. We recommend storing the magic string “SIGN” with the key to distinguish keys that have a value for allowed_schemes from those that should use the default allowed_schemes. Devices that do not support the alternative signing algorithms may refuse to load these keys and return an error of OEMCrypto_ERROR_NOT_IMPLEMENTED. The main use case for these alternative signing algorithms is to support devices that use X509 certificates for authentication when acting as a ChromeCast receiver. This is not needed for devices that wish to send data to a ChromeCast.

If the first four bytes of the buffer enc_rsa_key are not the string “SIGN”, then the default value of allowed_schemes = 1 (kSign_RSASSA_PSS) will be used.

Verification and Algorithm

The following checks should be performed. If any check fails, an error is returned, and the key is not loaded.

1. Check that all the pointer values passed into it are within the buffer specified by message and message_length.
2. Verify that `in_wrapped_rsa_key_length` is large enough to hold the rewrapped key, returning `OEMCrypto_ERROR_SHORT_BUFFER` otherwise.
3. Verify the message signature, using the derived signing key (mac_key[server]) from a previous call to OEMCrypto_GenerateDerivedKeys or

OEMCrypto_DeriveKeysFromSessionKey.

4. The function ODK_ParseProvisioning is called to parse the message.
5. Decrypt enc_rsa_key in the buffer rsa_key using the session's derived encryption key (enc_key). Use enc_rsa_key_iv as the initial vector for AES_128-CBC mode, with PKCS#5 padding. The rsa_key should be kept in secure memory and protected from the user.
6. If the first four bytes of the buffer rsa_key are the string "SIGN", then the actual RSA key begins on the 9th byte of the buffer. The second four bytes of the buffer is the 32 bit field "allowed_schemes", of type RSA_Padding_Scheme, which is used in OEMCrypto_GenerateRSASignature. The value of allowed_schemes must also be wrapped with RSA key. We recommend storing the magic string "SIGN" with the key to distinguish keys that have a value for allowed_schemes from those that should use the default allowed_schemes. Devices that do not support the alternative signing algorithms may refuse to load these keys and return an error of OEMCrypto_ERROR_NOT_IMPLEMENTED. The main use case for these alternative signing algorithms is to support devices that use X.509 certificates for authentication when acting as a ChromeCast receiver. This is not needed for devices that wish to send data to a ChromeCast.
7. If the first four bytes of the buffer rsa_key are not the string "SIGN", then the default value of allowed_schemes = 1 (kSign_RSASSA_PSS) will be used.
8. After possibly skipping past the first 8 bytes signifying the allowed signing algorithm, the rest of the buffer rsa_key contains an RSA device key in PKCS#8 binary DER encoded format. The OEMCrypto library shall verify that this RSA key is valid.
9. Re-encrypt the device RSA key with an internal key (such as the OEM key or Widevine Keybox key) and the generated IV using AES-128-CBC with PKCS#5 padding.
10. Copy the rewrapped key to the buffer specified by `wrapped_rsa_key` and the size of the wrapped key to `wrapped_rsa_key_length`.

Parameters

[in] session: crypto session identifier.

[in] message: pointer to memory containing data.

[in] message_length: length of the message, in bytes.

[in] core_message_length: length of the core submessage, in bytes.

[in] signature: pointer to memory containing the signature.

[in] signature_length: length of the signature, in bytes.

[out] wrapped_rsa_key: pointer to buffer in which encrypted RSA key should be stored. May be null on the first call in order to find required buffer size.

[in/out] wrapped_rsa_key_length: (in) length of the encrypted RSA key, in bytes.
(out) actual length of the encrypted RSA key

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INVALID_RSA_KEY
OEMCrypto_ERROR_SIGNATURE_FAILURE

OEMCrypto_ERROR_INVALID_NONCE
OEMCrypto_ERROR_SHORT_BUFFER
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_LoadDRMPrivateKey

```
OEMCryptoResult OEMCrypto_LoadDRMPrivateKey(
                      OEMCrypto_SESSION session,
                      OEMCrypto_PrivateKeyType key_type,
                      const uint8_t* wrapped_rsa_key,
                      size_t wrapped_rsa_key_length);

typedef enum OEMCrypto_PrivateKeyType {
  OEMCrypto_RSA_Private_Key,
  OEMCrypto_ECC_Private_Key,
} OEMCrypto_PrivateKeyType;
```

Loads a wrapped RSA or ECC private key to secure memory for use by this session in future calls to OEMCrypto_PrepAndSignLicenseRequest or OEMCrypto_DeriveKeysFromSessionKey. The wrapped private key will be the one verified and wrapped by OEMCrypto_LoadProvisioning. The private key should be stored in secure memory.

If the bit field “allowed_schemes” was wrapped with this RSA key, its value will be loaded and stored with the RSA key, and the key may be used with calls to OEMCrypto_GenerateRSASignature. If there was not a bit field wrapped with the RSA key, the key will be used for OEMCrypto_PrepAndSignLicenseRequest or OEMCrypto_DeriveKeysFromSessionKey

Verification

The following checks should be performed. If any check fails, an error is returned, and the RSA key is not loaded.

1. The wrapped key has a valid signature, as described in RewrapDeviceRSAKey.
2. The decrypted key is a valid private RSA key.
3. If a value for allowed_schemes is included with the key, it is a valid value.

Parameters

[in] session: crypto session identifier.
[in] key_type: indicates either an RSA or ECC key for devices that support both.
[in] wrapped_rsa_key: wrapped device RSA key stored on the device. Format is PKCS#8, binary DER encoded, and encrypted with a key internal to the OEMCrypto instance, using AES-128-CBC with PKCS#5 padding. This is the wrapped key generated by OEMCrypto_RewrapDeviceRSAKey.
[in] wrapped_rsa_key_length: length of the wrapped key buffer, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INVALID_RSA_KEY
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_LoadTestRSAKey

Some platforms do not support keyboxes or OEM Certificates. On those platforms, there is a DRM certificate baked into the OEMCrypto library. This is unusual, and is only available for L3 devices. In order to debug and test those devices, they should be able to switch to the test DRM certificate.

```
OEMCryptoResult OEMCrypto_LoadTestRSAKey(void);
```

Temporarily use the standard test RSA key until the next call to OEMCrypto_Terminate. This allows a standard suite of unit tests to be run on a production device without permanently changing the key. Using the test key is ***not*** persistent.

The test key can be found in the unit test code, oemcrypto_test.cpp, in PKCS8 form as the constant kTestRSAPKCS8PrivateKeyInfo2_2048.

Parameters

none

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_NOT_IMPLEMENTED - devices that use a keybox should not implement this function
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method is new in API version 10.

OEMCrypto_GenerateRSASignature

```
OEMCryptoResult OEMCrypto_GenerateRSASignature(
                      OEMCrypto_SESSION session,
                      const uint8_t* message,
                      size_t message_length,
                      uint8_t* signature,
                      size_t *signature_length,
                       RSA_Padding_Scheme padding_scheme);

typedef uint8_t RSA_Padding_Scheme;
```

The OEMCrypto_GenerateRSASignature method is only used for devices that are CAST receivers. This function is called after OEMCrypto_LoadDRMPrivateKey for the same session.

The parameter padding_scheme has two possible legacy values:
0x1 - RSASSA-PSS with SHA1.
0x2 - PKCS1 with block type 1 padding (only).

The only supported padding scheme is 0x2 since version 16 of this API. In this second case, the "message" is already a digest, so no further hashing is applied, and the message_length can be no longer than 83 bytes. If the message_length is greater than 83 bytes OEMCrypto_ERROR_SIGNATURE_FAILURE shall be returned.

The second padding scheme is for devices that use X509 certificates for authentication. The

main example is devices that work as a Cast receiver, like a ChromeCast, not for devices that wish to send to the Cast device, such as almost all Android devices. OEMs that do not support X509 certificate authentication need not implement this function and can return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Verification

Both the padding_scheme and the RSA key's allowed_schemes must be 0x2. If not, then the signature is not computed and the error OEMCrypto_ERROR_INVALID_RSA_KEY is returned.

Parameters

[in] session: crypto session identifier.
[in] message: pointer to memory containing message to be signed.
[in] message_length: length of the message, in bytes.
[out] signature: buffer to hold the message signature. On return, it will contain the message signature generated with the device private RSA key using RSASSA-PSS. Will be null on the first call in order to find required buffer size.
[in/out] signature_length: (in) length of the signature buffer, in bytes.
(out) actual length of the signature
[in] padding_scheme: specify which scheme to use for the signature.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER if the signature buffer is too small.
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_INVALID_CONTEXT
OEMCrypto_ERROR_INVALID_RSA_KEY
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_NOT_IMPLEMENTED - if algorithm > 0, and the device does not support that algorithm.
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support message sizes as described in the section OEMCrypto_ResourceRatingTier.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a "Session Function" and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method changed in API version 16.

Usage Table API

The following list shows the APIs required for Usage Table maintenance and reporting.

OEMCrypto_CreateUsageTableHeader

```
OEMCryptoResult OEMCrypto_CreateUsageTableHeader(uint8_t* header_buffer,
                                                 size_t* header_buffer_length);
```

This creates a new Usage Table Header with no entries. If there is already a generation number stored in secure storage, it will be incremented by 1 and used as the new Master Generation Number. This will only be called if the CDM layer finds no existing usage table on the file system. OEMCrypto will encrypt and sign the new, empty, header and return it in the provided buffer.

The new entry should be created with a status of kUnused and all times times should be set to 0.

Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Parameters

[out] header_buffer: pointer to memory where encrypted usage table header is written.
[in/out] header_buffer_length: (in) length of the header_buffer, in bytes.
(out) actual length of the header_buffer

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER - if header_buffer_length is too small.
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Usage Table Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_LoadUsageTableHeader

```
OEMCryptoResult OEMCrypto_LoadUsageTableHeader(const uint8_t* buffer,
                                               size_t buffer_length);
```

This loads the Usage Table Header. The buffer’s signature is verified and the buffer is decrypted. OEMCrypto will verify the verification string. If the Master Generation Number is

more than 1 off, the table is considered bad, the headers are NOT loaded, and the error OEMCrypto_ERROR_GENERATION_SKEW is returned. If the generation number is off by 1, the warning OEMCrypto_WARNING_GENERATION_SKEW is returned but the header is still loaded. This warning may be logged by the CDM layer.

Parameters

[in] buffer: pointer to memory containing encrypted usage table header.
[in] buffer_length: length of the buffer, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER
OEMCrypto_ERROR_NOT_IMPLEMENTED - some devices do not implement usage tables.
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_WARNING_GENERATION_SKEW - if the generation number is off by exactly 1.
OEMCrypto_ERROR_GENERATION_SKEW - if the generation number is off by **more** than 1.
OEMCrypto_ERROR_SIGNATURE_FAILURE - if the signature failed.
OEMCrypto_ERROR_BAD_MAGIC - verification string does not match.
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Usage Table Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_CreateNewUsageEntry

```
OEMCryptoResult OEMCrypto_CreateNewUsageEntry(OEMCrypto_SESSION session,
                                              uint32_t *usage_entry_number);
```

This creates a new usage entry. The size of the header will be increased by 8 bytes, and secure volatile memory will be allocated for it. The new entry will be associated with the given session. The status of the new entry will be set to “unused”. OEMCrypto will set *usage_entry_number to be the index of the new entry. The first entry created will have index 0. The new entry will be initialized with a generation number equal to the master generation number, which will also be stored in the header’s new slot. Then the master generation number will be incremented. Since each entry’s generation number is less than the master generation number, the new entry will have a generation number that is larger than all other entries and larger than all previously deleted entries. This helps prevent a rogue application from deleting an entry and then loading an old version of it.

If the session already has a usage entry associated with it, the error OEMCrypto_ERROR_MULTIPLE_USAGE_ENTRIES is returned.

Parameters

[in] session: handle for the session to be used.

[out] usage_entry_number: index of new usage entry.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NOT_IMPLEMENTED - some devices do not implement usage tables.
OEMCrypto_ERROR_INSUFFICIENT_RESOURCES - if there is no room in memory to increase the size of the usage table header. The CDM layer can delete some entries and then try again, or it can pass the error up to the application.
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED
OEMCrypto_ERROR_MULTIPLE_USAGE_ENTRIES - if there already is a usage entry loaded into this session

Threading

This is a “Usage Table Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_LoadUsageEntry

```
OEMCryptoResult OEMCrypto_LoadUsageEntry(OEMCrypto_SESSION session,
                                         uint32_t usage_entry_number,
                                         const uint8_t *buffer,
                                         size_t buffer_length);
```

This loads a usage table saved previously by UpdateUsageEntry. The signature at the beginning of the buffer is verified and the buffer will be decrypted. Then the verification field in the entry will be verified. The index in the entry must match the index passed in. The generation number in the entry will be compared against the entry’s corresponding generation number in the header. If it is off by 1, a warning is returned, but the entry is still loaded. This warning may be logged by the CDM layer. If the generation number is off by more than 1, an error is returned and the entry is not loaded.

OEMCrypto shall call **ODK_ReloadClockValues**, as described in “License Duration and Renweal” to set the session’s clock values.

If the entry is already loaded into another open session, then this fails and returns OEMCrypto_ERROR_INVALID_SESSION. If the session already has a usage entry associated with it, the error OEMCrypto_ERROR_MULTIPLE_USAGE_ENTRIES is returned.

Before version API 16, the usage entry stored the time that the license was loaded. This value is now interpreted as the time that the licence request was signed. This can be achieved by simply renaming the field and using the same value when reloading an older entry.

Parameters

[in] session: handle for the session to be used.

[in] usage_entry_number: index of existing usage entry.
[in] buffer: pointer to memory containing encrypted usage table entry.
[in] buffer_length: length of the buffer, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER
OEMCrypto_ERROR_NOT_IMPLEMENTED - some devices do not implement usage tables.
OEMCrypto_ERROR_UNKNOWN_FAILURE - index beyond end of table.
OEMCrypto_ERROR_INVALID_SESSION - entry associated with another session or the index is wrong.
OEMCrypto_WARNING_GENERATION_SKEW - if the generation number is off by exactly 1.
OEMCrypto_ERROR_GENERATION_SKEW - if the generation number is off by **more** than 1.
OEMCrypto_ERROR_SIGNATURE_FAILURE - if the signature failed.
OEMCrypto_ERROR_BAD_MAGIC - verification string does not match.
OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

OEMCrypto_ERROR_MULTIPLE_USAGE_ENTRIES - if there already is a usage entry loaded into this session

Threading

This is a “Usage Table Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_UpdateUsageEntry

```
OEMCryptoResult OEMCrypto_UpdateUsageEntry(OEMCrypto_SESSION session,
                      OEMCrypto_SharedMemory* header_buffer,
                      size_t* header_buffer_length,
                      OEMCrypto_SharedMemory* entry_buffer,
                      size_t* entry_buffer_length);
```

Updates the session’s usage entry and fills buffers with the encrypted and signed entry and usage table header.

OEMCrypto shall call ODK_UpdateLastPlaybackTime to update the session’s clock values, as discussed in the document “License Duration and Renewal”. The values in the session’s clock values structure are copied to the usage entry.

OEMCrypto shall update all time and status values in the entry, and then increment the entry’s generation number. The corresponding generation number in the usage table header is also incremented so that it matches the one in the entry. The master generation number in the usage table header is incremented and the master generation number is copied to secure persistent storage. OEMCrypto will encrypt and sign the entry into the entry_buffer, and it will encrypt and sign the usage table header into the header_buffer. Some actions, such as the first

decrypt and deactivating an entry, will also increment the entry's generation number as well as changing the entry's status and time fields. The first decryption will change the status from Inactive to Active, and it will set the time stamp "first decrypt".

If the usage entry has the flag ForbidReport set, then the flag is cleared. It is the responsibility of the CDM layer to call this function and save the usage table before the next call to ReportUsage and before the CDM is terminated. Failure to do so will result in generation number skew, which will invalidate all of the usage table.

If either entry_buffer_length or header_buffer_length is not large enough, they are set to the needed size, and return OEMCrypto_ERROR_SHORT_BUFFER. In this case, the entry is not updated, ForbidReport is not cleared, generation numbers are not incremented, and no other work is done.

Parameters

[in] session: handle for the session to be used.
[out] header_buffer: pointer to memory where encrypted usage table header is written.
[in/out] header_buffer_length: (in) length of the header_buffer, in bytes.
(out) actual length of the header_buffer
[out] entry_buffer: pointer to memory where encrypted usage table entry is written.
[in/out] entry_buffer_length: (in) length of the entry_buffer, in bytes.
(out) actual length of the entry_buffer

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER
OEMCrypto_ERROR_NOT_IMPLEMENTED - some devices do not implement usage tables.
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a "Usage Table Function" and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_DeactivateUsageEntry

```
OEMCryptoResult OEMCrypto_DeactivateUsageEntry(OEMCrypto_SESSION session,
                                               const uint8_t *pst,
                                                size_t pst_length);
```

This deactivates the usage entry associated with the current session. This means that the status of the usage entry is changed to InactiveUsed if it was Active, or InactiveUnused if it was Unused. This also increments the entry's generation number, and the header's master generation number. The corresponding generation number in the usage table header is also incremented so that it matches the one in the entry. The entry's flag ForbidReport will be set.

This flag prevents an application from generating a report of a deactivated license without first saving the entry.

OEMCrypto shall call ODK_DeactivateUsageEntry to update the session's clock values, as discussed in the document "License Duration and Renewal".

It is allowed to call this function multiple times. If the state is already InactiveUsed or InactiveUnused, then this function does not change the entry or its state.

Parameters

[in] session: handle for the session to be used.
[in] pst: pointer to memory containing Provider Session Token.
[in] pst_length: length of the pst, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_INVALID_CONTEXT - an entry was not created or loaded, or the pst does not match.
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support pst sizes of at least 255 bytes.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a "Usage Table Function" and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 16.

OEMCrypto_ReportUsage

```
OEMCryptoResult OEMCrypto_ReportUsage(OEMCrypto_SESSION session,
                                      const uint8_t *pst,
                                      size_t pst_length,
                                      uint8_t *buffer,
                                      size_t *buffer_length);

typedef struct {
  uint8_t signature[20];  //  -- HMAC SHA1 of the rest of the report.
  uint8_t status;  // current status of entry. (OEMCrypto_Usage_Entry_Status)
  uint8_t clock_security_level;
  uint8_t pst_length;
```

```
  uint8_t padding;                          // make int64's word aligned.
  int64_t seconds_since_license_signed;  // now - time_of_license_signed
  int64_t seconds_since_first_decrypt;     // now - time_of_first_decrypt
  int64_t seconds_since_last_decrypt;      // now - time_of_last_decrypt
  uint8_t pst[];
} __attribute__((packed)) OEMCrypto_PST_Report;
```

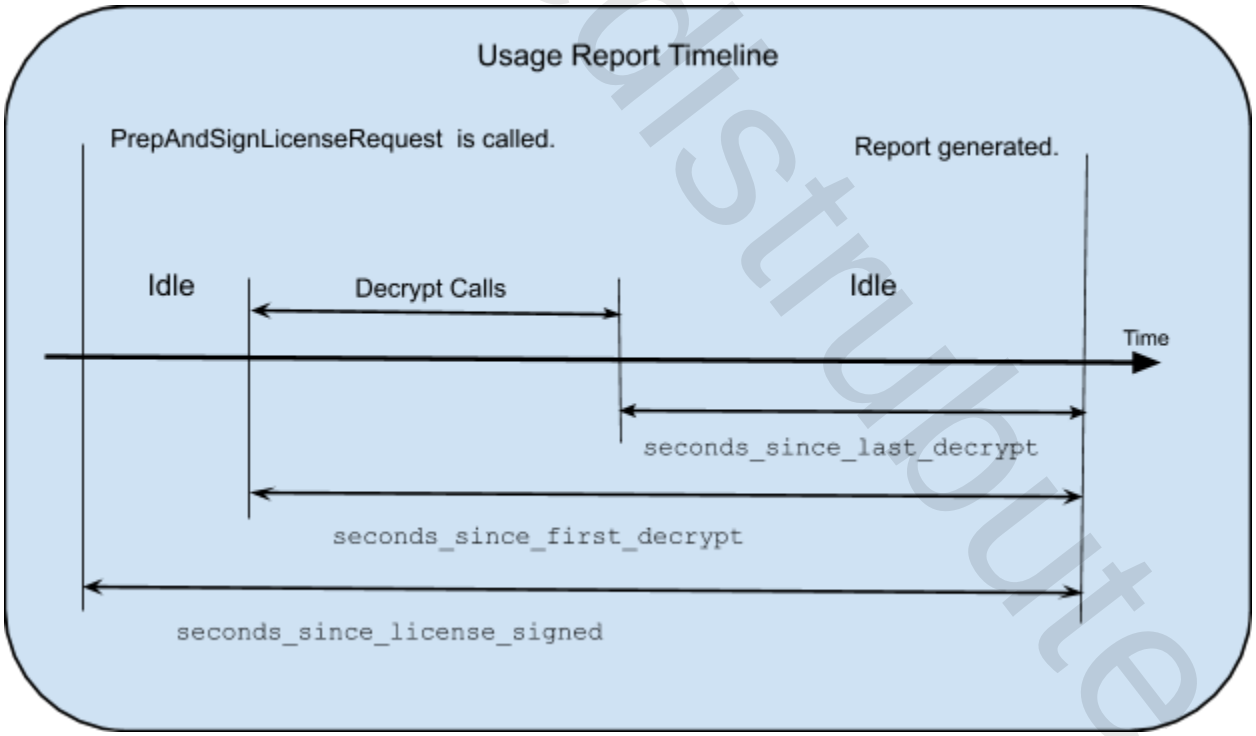
All fields of OEMCrypto_PST_Report are in network byte order.

If the buffer_length is not sufficient to hold a report structure, set buffer_length and return OEMCrypto_ERROR_SHORT_BUFFER.

If an entry was not loaded or created with OEMCrypto_CreateNewUsageEntry or OEMCRypto_LoadUsageEntry, or if the pst does not match that in the entry, return the error OEMCrypto_ERROR_INVALID_CONTEXT.

If the usage entry's flag ForbidReport is set, indicating the entry has not been saved since the entry was deactivated, then the error OEMCrypto_ERROR_ENTRY_NEEDS_UPDATE is returned and a report is not generated. Similarly, if any key in the session has been used since the last call to OEMCrypto_UpdateUsageEntry, then the report is not generated, and OEMCrypto returns the error OEMCrypto_ERROR_ENTRY_NEEDS_UPDATE.

The pst_report is filled out by subtracting the times in the Usage Entry from the current time on the secure clock. This design was chosen to avoid the device's secure clock with any external clock.

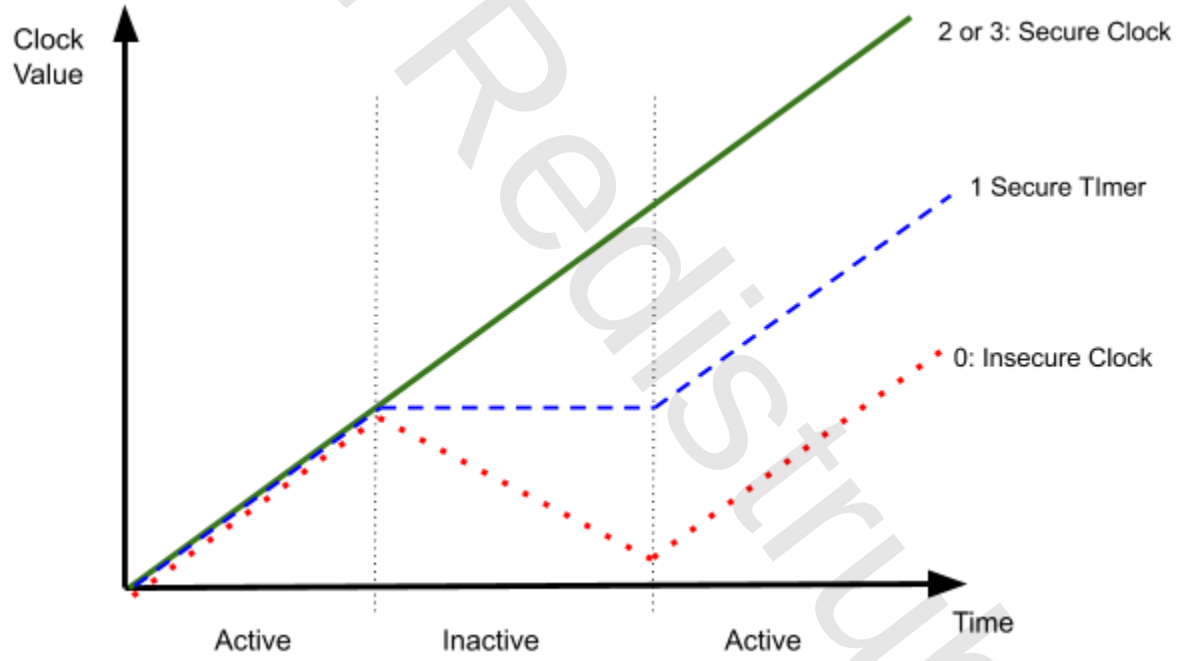


Valid values for status are:

- 0 = kUnused -- the keys have not been used to decrypt.
- 1 = kActive -- the keys have been used, and have not been deactivated.
- 2 = kInactive - deprecated. Use kInactiveUsed or kInactiveUnused.
- 3 = kInactiveUsed -- the keys have been marked inactive after being active.
- 4 = kInactiveUnused -- they keys have been marked inactive, but were never active.

The clock_security_level is reported as follows:

- 0 = Insecure Clock - clock just uses system time.
- 1 = Secure Timer - clock runs from a secure timer which is initialized from system time when OEMCrypto becomes active and cannot be modified by user software or the user while OEMCrypto is active. A secure timer cannot run backwards, even while OEMCrypto is not active.
- 2 = Secure Clock - Real-time clock set from a secure source that cannot be modified by user software regardless of whether OEMCrypto is active or inactive. The clock time can only be modified by tampering with the security software or hardware.
- 3 = Hardware Secure Clock - Real-time clock set from a secure source that cannot be modified by user software and there are security features that prevent the user from modifying the clock in hardware, such as a tamper proof battery.



After pst_report has been filled in, the HMAC SHA1 signature is computed for the buffer from bytes 20 to the end of the pst field. The signature is computed using the mac_key[client] which is stored in the usage table. The HMAC SHA1 signature is used to prevent a rogue application from using OMECrypto_GenerateSignature to forge a Usage Report.

Before version 16 of this API, seconds_since_license_received was reported instead of seconds_since_license_signed. For any practical bookkeeping purposes, these events are essentially at the same time.

Devices that do not implement a Session Usage Table may return

OEMCrypto_ERROR_NOT_IMPLEMENTED.

Parameters

[in] session: handle for the session to be used.
[in] pst: pointer to memory containing Provider Session Token.
[in] pst_length: length of the pst, in bytes.
[out] buffer: pointer to buffer in which usage report should be stored. May be null on the first call in order to find required buffer size.
[in/out] buffer_length: (in) length of the report buffer, in bytes.
(out) actual length of the report

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER - if report buffer is not large enough to hold the output report.
OEMCrypto_ERROR_INVALID_SESSION - no open session with that id.
OEMCrypto_ERROR_INVALID_CONTEXT
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_ENTRY_NEEDS_UPDATE - if no call to UpdateUsageEntry since last call to Deactivate or since key use.
OEMCrypto_ERROR_WRONG_PST - report asked for wrong pst.
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Buffer Sizes

OEMCrypto shall support pst sizes of at least 255 bytes.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffer is larger than the supported size.

Threading

This is a “Usage Table Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method changed in API version 13.

OEMCrypto_MoveEntry

```
OEMCryptoResult OEMCrypto_MoveEntry(OEMCrypto_SESSION session,
                                    uint32_t new_index);
```

Moves the entry associated with the current session from one location in the usage table header to another. This function is used by the CDM layer to defragment the usage table. This does not modify any data in the entry, except the index and the generation number. The index in the

session's usage entry will be changed to new_index. The generation number in session's usage entry and in the header for new_index will be increased to the master generation number, and then the master generation number is incremented. If there was an existing entry at the new location, it will be overwritten. It is an error to call this when the entry that was at new_index is associated with a currently open session. In this case, the error code OEMCrypto_ERROR_ENTRY_IN_USE is returned. It is the CDM layer's responsibility to call UpdateUsageEntry after moving an entry. It is an error for new_index to be beyond the end of the existing usage table header.

Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Parameters

[in] session: handle for the session to be used.
[in] new_index: new index to be used for the session's usage entry

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_BUFFER_TOO_LARGE
OEMCrypto_ERROR_ENTRY_IN_USE
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a "Usage Table Function" and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version

This method is new in API version 13.

OEMCrypto_ShrinkUsageTableHeader

```
OEMCryptoResult OEMCrypto_ShrinkUsageTableHeader(
                                uint32_t new_entry_count,
                                uint8_t* header_buffer,
                                size_t* header_buffer_length);
```

This shrinks the usage table and the header. This function is used by the CDM layer after it has defragmented the usage table and can delete unused entries. It is an error if any open session is associated with an entry that will be erased - the error OEMCrypto_ERROR_ENTRY_IN_USE shall be returned in this case, and the header shall not be modified. If new_entry_count is larger than the current size, then the header is not changed and the error OEMCrypto_ERROR_UNKNOWN_FAILURE is returned. If the header has not been previously loaded, then an error is returned. OEMCrypto will increment the master generation number in the header and store the new value in secure persistent storage. Then, OEMCrypto will encrypt and sign the header into the provided buffer. The generation numbers of all remaining entries

will remain unchanged. The next time OEMCrypto_CreateNewUsageEntry is called, the new entry will have an index of new_entry_count.

Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

If header_buffer_length is not large enough to hold the new table, it is set to the needed value, the generation number is **not** incremented, and OEMCrypto_ERROR_SHORT_BUFFER is returned.

If the header has not been loaded or created, return the error OEMCrypto_ERROR_UNKNOWN_FAILURE.

Parameters
[in] new_entry_count: number of entries in the to be in the header.
[out] header_buffer: pointer to memory where encrypted usage table header is written.
[in/out] header_buffer_length: (in) length of the header_buffer, in bytes.
(out) actual length of the header_buffer

Returns
OEMCrypto_SUCCESS success
OEMCrypto_ERROR_SHORT_BUFFER
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_ENTRY_IN_USE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading
This is a “Usage Table Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version
This method is new in API version 13.

Test and Verification Functions

Functions in this section are designed to help test OEMCrypto and the device. They are not used during normal operation. Some functions, like OEMCrypto_RemoveSRM should only be implemented on test devices. Other functions, like those that test the full decrypt data path may be supported on a production device with no added risk of security loss.

The following functions are used just for testing and verification of OEMCrypto and the CDM code.

OEMCrypto_RemoveSRM

```
OEMCryptoResult OEMCrypto_RemoveSRM(void);
```

Delete the current SRM. Any valid SRM, regardless of its version number, will be installable after this via OEMCrypto_LoadSRM.

This function should **not** be implemented on production devices, and will only be used to verify unit tests on a test device.

Parameters
none

Returns
OEMCrypto_SUCCESS - if the SRM file was deleted.
OEMCrypto_ERROR_NOT_IMPLEMENTED - always on production devices.

Threading
This is an “Initialization and Termination Function” and will not be called simultaneously with any other function, as if the CDM holds a write lock on the OEMCrypto system.

Version
This method is new in API version 13.

OEMCrypto_SupportsDecryptHash

```
uint32_t OEMCrypto_SupportsDecryptHash(void);
```

Returns the type of hash function supported for Full Decrypt Path Testing. A hash type of OEMCrypto_Hash_Not_Supported = 0 means this feature is not supported. OEMCrypto is not required by Google to support this feature, but support will greatly improve automated testing. A hash type of OEMCrypto_CRC_Clear_Buffer = 1 means the device will be able to compute the CRC 32 checksum of the decrypted content in the secure buffer after a call to OEMCrypto_DecryptCENC. Google intends to provide test applications on some platforms, such as Android, that will automate decryption testing using the CRC 32 checksum of all frames in some test content.

If an SOC vendor cannot support CRC 32 checksums of decrypted output, but can support some other hash or checksum, then the function should return OEMCrypto_Partner_Defined_Hash = 2 and those partners should modify the test application to compute the appropriate hash. An application that computes the CRC 32 hashes of test content and builds a hash file in the correct format will be provided by Widevine. The source of this application will be provided so that partners may modify it to compute their own hash format and generate their own hashes.

Returns
OEMCrypto_Hash_Not_Supported = 0;
OEMCrypto_CRC_Clear_Buffer = 1;
OEMCrypto_Partner_Defined_Hash = 2;

Threading
This is a “Property Function” and may be called simultaneously with any other property function or session function, but not any initialization or usage table function, as if the CDM holds a read lock on the OEMCrypto system.

Version

This method is new in API version 15.

OEMCrypto_SetDecryptHash

```
OEMCryptoResult OEMCrypto_SetDecryptHash(OEMCrypto_SESSION session,
                                         uint32_t frame_number,
                                         const uint8_t* hash,
                                         size_t hash_length);
```

Set the hash value for the next frame to be decrypted. This function is called before the first subsample is passed to OEMCrypto_DecryptCENC, when the subsample_flag has the bit OEMCrypto_FirstSubsample set. The hash is over all of the frame or sample: encrypted and clear subsamples concatenated together, up to, and including the subsample with the subsample_flag having the bit OEMCrypto_LastSubsample set. If hashing the output is not supported, then this will return OEMCrypto_ERROR_NOT_IMPLEMENTED. If the hash is ill formed or there are other error conditions, this returns OEMCrypto_ERROR_UNKNOWN_FAILURE. The length of the hash will be at most 128 bytes, and will be 4 bytes (32 bits) for the default CRC32 hash.

This may be called before the first call to SelectKey. In that case, this function cannot verify that the key control block allows hash verification. The function DecryptCENC should verify that the key control bit allows hash verification when it is called. If an attempt is made to compute a hash when the selected key does not have the bit Allow_Hash_Verification set, then a hash should not be computed, and OEMCrypto_GetHashErrorCode should return the error OEMCrypto_ERROR_UNKNOWN_FAILURE.

OEMCrypto should compute the hash of the frame and then compare it with the correct value. If the values differ, then OEMCrypto should latch in an error and save the frame number of the bad hash. It is allowed for OEMCrypto to postpone computation of the hash until the frame is displayed. This might happen if the actual decryption operation is carried out by a later step in the video pipeline, or if you are using a partner specified hash of the decoded frame. For this reason, an error state must be saved until the call to OEMCrypto_GetHashErrorCode is made.

Parameters

[in] session: session id for current decrypt operation
[in] frame_number: frame number for the recent DecryptCENC sample.
[in] hash: hash or CRC of previously decrypted frame.
[in] hash_length: length of hash, in bytes.

Returns

OEMCrypto_SUCCESS - if the hash was set
OEMCrypto_ERROR_NOT_IMPLEMENTED - function not implemented
OEMCrypto_ERROR_INVALID_SESSION - session not open
OEMCrypto_ERROR_SHORT_BUFFER - hash_length too short for supported hash type
OEMCrypto_ERROR_BUFFER_TOO_LARGE - hash_length too long for supported hash type
OEMCrypto_ERROR_UNKNOWN_FAILURE - other error
OEMCrypto_ERROR_SESSION_LOST_STATE

OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is new in API version 15.

OEMCrypto_GetHashErrorCode

```
OEMCryptoResult OEMCrypto_GetHashErrorCode(OEMCrypto_SESSION session,
           uint32_t* failed_frame_number);
```

If the hash set in OEMCrypto_SetDecryptHash did not match the computed hash, then an error code was saved internally. This function returns that error and the frame number of the bad hash. This will be called periodically, but might not be in sync with the decrypt loop. OEMCrypto shall not reset the error state to “no error” once any frame has failed verification. It should be initialized to “no error” when the session is first opened. If there is more than one bad frame, it is the implementer’s choice if it is more useful to return the number of the first bad frame, or the most recent bad frame.

If the hash could not be computed -- either because the Allow_Hash_Verification was not set in the key control block, or because there were other issues -- this function should return OEMCrypto_ERROR_UNKNOWN_FAILURE.

Parameters

[in] session: session id for operation.
[out] failed_frame_number: frame number for sample with incorrect hash.

Returns

OEMCrypto_SUCCESS - if all frames have had a correct hash
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_BAD_HASH - if any frame had an incorrect hash
OEMCrypto_ERROR_UNKNOWN_FAILURE - if the hash could not be computed
OEMCrypto_ERROR_SESSION_LOST_STATE
OEMCrypto_ERROR_SYSTEM_INVALIDATED

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is new in API version 15.

OEMCrypto_AllocateSecureBuffer

```
OEMCryptoResult OEMCrypto_AllocateSecureBuffer(
                                OEMCrypto_SESSION session,
                                size_t buffer_size,
                                OEMCrypto_DestBufferDesc *output_descriptor,
                                int *secure_fd);
```

Allocates a secure buffer and fills out the destination buffer information in output. The integer secure_fd may also be set to indicate the source of the buffer. OEMCrypto may use the secure_fd to help track the buffer if it wishes. The unit tests will pass a pointer to the same destination buffer description and the same secure_fd to OEMCrypto_FreeSecureBuffer when the buffer is to be freed.

This is especially helpful if the hash functions above are supported. This will only be used by the OEMCrypto unit tests, so we recommend returning OEMCrypto_ERROR_NOT_IMPLEMENTED for production devices if performance is an issue. If OEMCrypto_ERROR_NOT_IMPLEMENTED is returned, then secure buffer unit tests will be skipped.

Parameters

[in] session: session id for operation.
[in] buffer_size: the requested buffer size.
[out] output: the buffer descriptor for the created buffer. This will be passed into the OEMCrypto_DecryptCENC function.
[out] secure_fd: a pointer to platform dependant file or buffer descriptor. This will be passed to OEMCrypto_FreeSecureBuffer.

Returns

OEMCrypto_SUCCESS - if the buffer was created
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_OUTPUT_TOO_LARGE
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is new in API version 16.

OEMCrypto_FreeSecureBuffer

```
OEMCryptoResult OEMCrypto_FreeSecureBuffer(
                                OEMCrypto_SESSION session,
```

```
                                    OEMCrypto_DestBufferDesc *output_descriptor,
                                    int secure_fd);
```

Frees a secure buffer that had previously been created with OEMCrypto_AllocateSecureBuffer. Any return value except OEMCrypto_SUCCESS will cause the unit test using secure buffers to fail.

Parameters

[in] session: session id for operation.
[out] output: the buffer descriptor modified by OEMCrypto_AllocateSecureBuffer
[in] secure_fd: The integer returned by OEMCrypto_AllocateSecureBuffer

Returns

OEMCrypto_SUCCESS - if the buffer was freed
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

This is a “Session Function” and may be called simultaneously with session functions for other sessions but not simultaneously with other functions for this session. It will not be called simultaneously with initialization or usage table functions. It is as if the CDM holds a write lock for this session, and a read lock on the OEMCrypto system.

Version

This method is new in API version 16.

Errors

State Loss Errors

Some devices may be designed in such a way that secure volatile memory is lost when the device enters a sleep state. When this happens, a session may be corrupted and decryption may not continue. If OEMCrypto detects such an error condition, it may return the error code OEMCrypto_ERROR_SESSION_LOST_STATE. The CDM layer will close that session, open a new session, and load a new license.

If OEMCrypto detects an error condition that affects all sessions, it may return the error code OEMCrypto_ERROR_SYSTEM_INVALIDATED. When this happens, the CDM layer will close all sessions, call OEMCrypto_Terminate and then re-initialize the system.

Error Codes

This is a list of error codes and their uses.

0	OEMCrypto_SUCCESS	No error.
1	OEMCrypto_ERROR_INIT_FAILED	Initialization failed.

2	OEMCrypto_ERROR_TERMINATE_FAILED	Termination failed.
7	OEMCrypto_ERROR_SHORT_BUFFER	Indicates an output buffer is not long enough to hold its data. Function can be called again with a larger buffer.
8	OEMCrypto_ERROR_NO_DEVICE_KEY	Indicates the keybox does not have a device key. (deprecated)
10	OEMCrypto_ERROR_KEYBOX_INVALID	Indicates Widevine keybox is invalid.
11	OEMCrypto_ERROR_NO_KEYDATA	Indicates Widevine keybox is invalid or does not have any key data.
13	OEMCrypto_ERROR_DECRYPT_FAILED	Indicates DecryptCENC or Generic Decrypt failed.
14	OEMCrypto_ERROR_WRITE_KEYBOX	Keybox could not be installed to secure memory.
15	OEMCrypto_ERROR_WRAP_KEYBOX	OEMCrypto_WrapKeybox failed to encrypt keybox.
16	OEMCrypto_ERROR_BAD_MAGIC	Keybox has bad magic field.
17	OEMCrypto_ERROR_BAD_CRC	Keybox has bad CRC field.
18	OEMCrypto_ERROR_NO_DEVICEID	GetDeviceID failed.
19	OEMCrypto_ERROR_RNG_FAILED	GetRandom failed.
20	OEMCrypto_ERROR_ RNG_NOT_SUPPORTED	GetRandom is not implemented.
22	OEMCrypto_ERROR_OPEN_SESSION_FAILED	OpenSession failed, but not with a resource issue.
23	OEMCrypto_ERROR_CLOSE_SESSION_FAILED	CloseSession failed on valid session.
24	OEMCrypto_ERROR_INVALID_SESSION	Specified session is not open or is in a corrupted state.
25	OEMCrypto_ERROR_NOT_IMPLEMENTED	Function is not implemented.
26	OEMCrypto_ERROR_NO_CONTENT_KEY	Failed to find the specified Key ID.
27	OEMCrypto_ERROR_CONTROL_INVALID	The control block of the specified key is not valid. Returned by SelectKey.
28	OEMCrypto_ERROR_UNKNOWN_FAILURE	Any other error.
29	OEMCrypto_ERROR_INVALID_CONTEXT	Context for signing or verification is not valid, or other sanity check failed.

30	OEMCrypto_ERROR_SIGNATURE_FAILURE	Could not sign specified buffer.
31	OEMCrypto_ERROR_TOO_MANY_SESSIONS	Not enough resources to open a new session.
32	OEMCrypto_ERROR_INVALID_NONCE	Nonce in server response does not match any in table.
33	OEMCrypto_ERROR_TOO_MANY_KEYS	Not enough resources to LoadKeys.
34	OEMCrypto_ERROR_ DEVICE_NOT_RSA_PROVISIONED	Session does not have an RSA key installed.
35	OEMCrypto_ERROR_INVALID_RSA_KEY	RSA key is not valid in LoadProvisioning, RewrapDeviceRSAKey or LoadDRMPrivateKey
36	OEMCrypto_ERROR_KEY_EXPIRED	The license has expired, but is otherwise valid.
37	OEMCrypto_ERROR_ INSUFFICIENT_RESOURCES	Other resource issues, such as buffers needed for decryption.
38	OEMCrypto_ERROR_INSUFFICIENT_HDCP	An attached display does not support the minimum HDCP version.
39	OEMCrypto_ERROR_BUFFER_TOO_LARGE	The length of a buffer is too large
40	OEMCrypto_WARNING_GENERATION_SKEW	Usage table generation number off by 1.
41	OEMCrypto_ERROR_GENERATION_SKEW	Usage table generation number off by more than 1.
42	OEMCrypto_LOCAL_DISPLAY_ONLY	CurrentSRMVersion is not relevant because no external output.
43	OEMCrypto_ERROR_ANALOG_OUTPUT	SelectKey failed because analog output could not be disabled.
44	OEMCrypto_ERROR_WRONG_PST	Offline license loaded entry with wrong pst.
45	OEMCrypto_ERROR_WRONG_KEYS	Offline license loaded entry with wrong mac keys.
46	OEMCrypto_ERROR_MISSING_MASTER	Obsolete
47	OEMCrypto_ERROR_LICENSE_INACTIVE	Attempt to use keys associated with a usage entry that is inactive.
48	OEMCrypto_ERROR_ENTRY_NEEDS_UPDATE	An attempt was made to call ReportUsage without calling

		UpdateUsageEntry first.
49	OEMCrypto_ERROR_ENTRY_IN_USE	An attempt was made to shrink the usage table past or move a usage entry onto an entry that is in use.
50	reserved - do not use	
51	OEMCrypto_KEY_NOT_LOADED	Obsolete. Use ERROR_NO_CONTENT_KEY.
52	OEMCrypto_KEY_NOT_ENTITLED	Attempt to load entitled content key with no matching entitlement key
53	OEMCrypto_ERROR_BAD_HASH	At least one frame had an incorrect hash when verifying the full decrypt path
54	OEMCrypto_ERROR_OUTPUT_TOO_LARGE	Decrypt failed because output is too large.
55	OEMCrypto_ERROR_SESSION_LOST_STATE	Session data in secure memory has been lost, requiring closing the current session
56	OEMCrypto_ERROR_SYSTEM_INVALIDATED	All data in secure memory has been lost, or other resource error requiring OEMCrypto_Terminate to recover.
57	OEMCrypto_ERROR_LICENSE_RELOAD	An attempt was made to load a license into a session that already has a license
58	OEMCrypto_ERROR_MULTIPLE_USAGE_ENTRIES	An attempt was made to load multiple usage entries into a single session.
59	OEMCrypto_WARNING_MIXED_ OUTPUT_PROTECTION	May be returned by DecryptCENC to indicate that some displays do not support the required level of HDCP
1k	ODK errors start at 1000.	

RSA Algorithm Details

Message signing and encryption using RSA algorithms shall be used during the license exchange process. The specific algorithms are RSASSA-PSS (signing) and RSA-OAEP (encryption). Both of these algorithms use random values in their operation, making them non-deterministic. These algorithms are described in the PKCS#8 specification.

RSASSA-PSS Details

Message signing using RSASSA-PSS shall be performed using the default algorithm parameters specified in PKCS#1:

- Hash algorithm: SHA1
- Mask generation algorithm: SHA1
- Salt length: 20 bytes
- Trailer field: 0xbc

RSA-OAEP

Message encryption using RSA-OAEP shall be performed using the default algorithm parameters specified in PKCS#1:

- Hash algorithm: SHA1
- Mask generation algorithm: SHA1
- Algorithm parameters: empty string
