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

Version 14

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

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 company that specifies HDCP.

Device Id — A null-terminated C-string uniquely identifying the device. 32 character maximum, including NULL termination. Used for Provisioning 2.0.

Device Key — 128-bit AES key assigned by Widevine and used to secure entitlements. 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 Provision — Install a Keybox that has been uniquely constructed for a specific device. This is done before the device is in the field.

Keybox — Widevine structure containing keys and other information used to establish a root of trust on a device. The keybox is either installed during manufacture 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.

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 cryptographic library implemented by the OEM to provide key and content protection.

Provider Session Token (PST) — the string used to track usage information and offline licenses.

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

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 2.0 — Using a Keybox to to request a DRM certificate from a provisioning server.

A device should use either Provisioning 2.0 or 3.0.

Provisioning 3.0 — Using 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) — Contains blacklist of revoked HDCP keys.

Trusted Execution Environment (TEE) — The portion of the device that contains security hardware and prevents access by non secure system resources.

References

DASH - 23001-7 ISO BMFF Common Encryption

DASH - 14496-12 ISO BMFF Amendment

W3C Encrypted Media Extensions (EME)

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

Draft International Standard ISO/IEC DIS 23001-7

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 security APIs used in Widevine content protection for playing content using Common Encryption (CENC). This includes content compatible with the *Dynamic Adaptive Streaming over HTTP* specification, ISO/IEC 23009-1 (MPEG DASH) using the DRM methods specified in ISO/IEC 23001-7: Common Encryption, on devices capable of playing premium video content. It also includes content delivered using HTTP Live Streaming (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 OEMCrypto

OEMCrypto is an interface to the trusted environment that implements the functions needed to protect and manage keys for the Widevine content protection system. We also use the word OEMCrypto to refer to an implementation of this interface. The interface provides: (1) a means to establish a signing key that can be used to verify the authenticity of messages to and from a license server, (2) a means to establish an encryption key that can be used to decrypt the key material contained in the messages, (3) a means to load encrypted content keys into the trusted environment and decrypt them, (4) a means to use the content keys to produce a decrypted stream for decoding and rendering and (5) a means to enforce license policies such as license duration and stream output protection requirements.

In this system the OEMCrypto implementation is responsible for ensuring that session keys, the decrypted content keys, and the decrypted content stream are never accessible to any user code running on the device. This is typically accomplished through a secondary processor and/or secure OS that has its own dedicated memory and runs the crypto algorithms that require access to the protected key material. In such a system, key material, or any bytes that have been decrypted with the device's root keys, are never returned back to the primary processor. The OEMCrypto implementation is also responsible for completely erasing all session-level state, including content keys and derived keys, when the session is terminated.

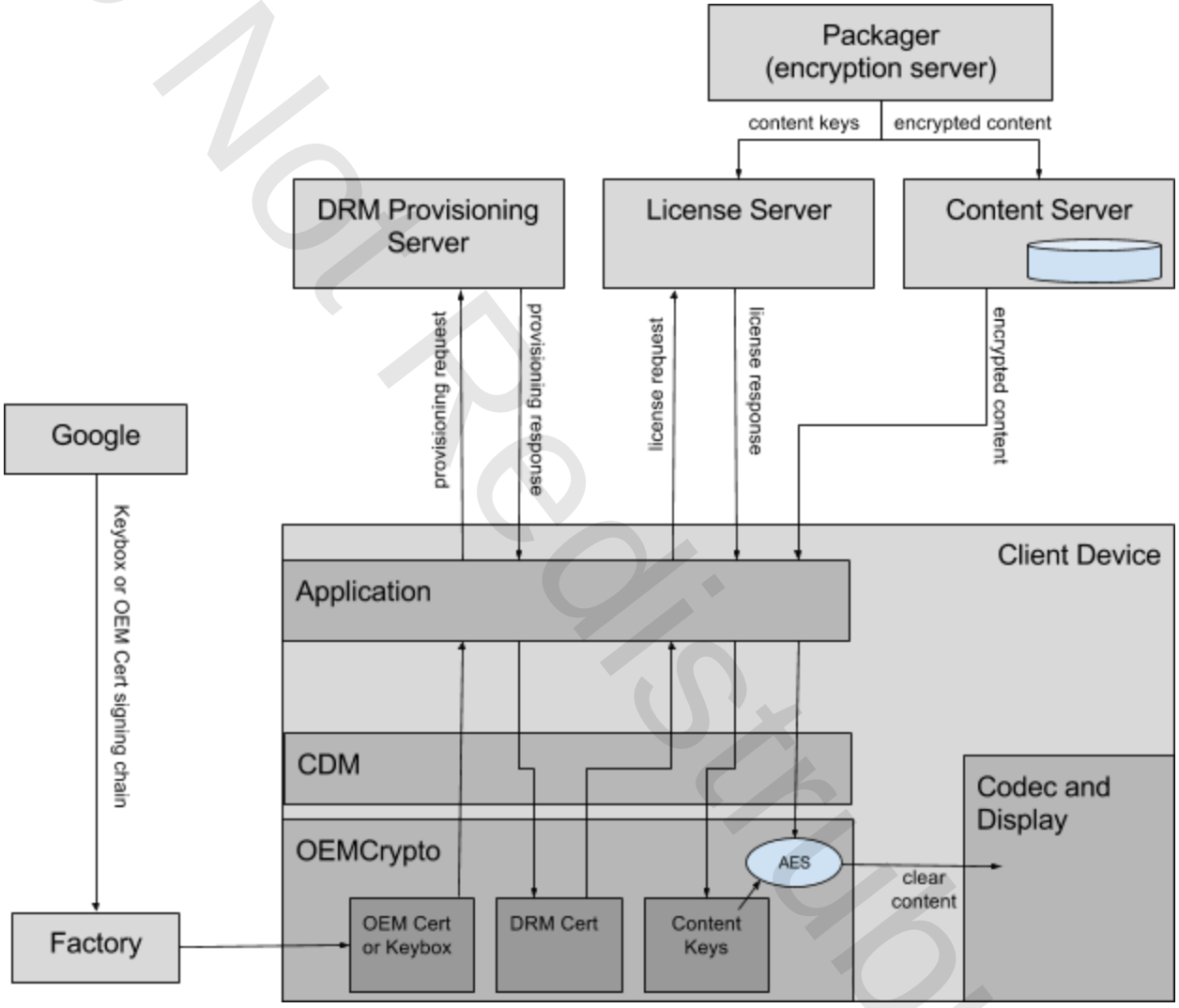
Overview of Widevine Content Protection System

The Widevine Content Protection System uses a tiered 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 on the device in the field. A provisioning request is sent from the device with either keybox information or the 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 content keys come from a license server. A set of keys and 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 will use the DRM Certificate to identify the device to the license server and attest to the devices security features.

Encrypted content is prepared using an encryption server 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 standard HTTP web servers. OEMCrypto's job is to decrypt the content, and to enforce any license restrictions such as time limits or output protection.



Security 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 clear to host CPU	Hardware Protected Video Path
Level 2	Yes	Factory	Yes	Keys never exposed in clear to host CPU	Clear video streams delivered to renderer
Level 3	No	Field	No	Clear keys exposed to host CPU	Clear video streams delivered to decoder

An OEM-provided OEMCrypto library is required for implementation of Widevine security Level 1 or 2.

Provisioning

Provisioning refers to 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 this authentication 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 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 device will use a DRM certificate to authenticate itself to 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 Provision 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

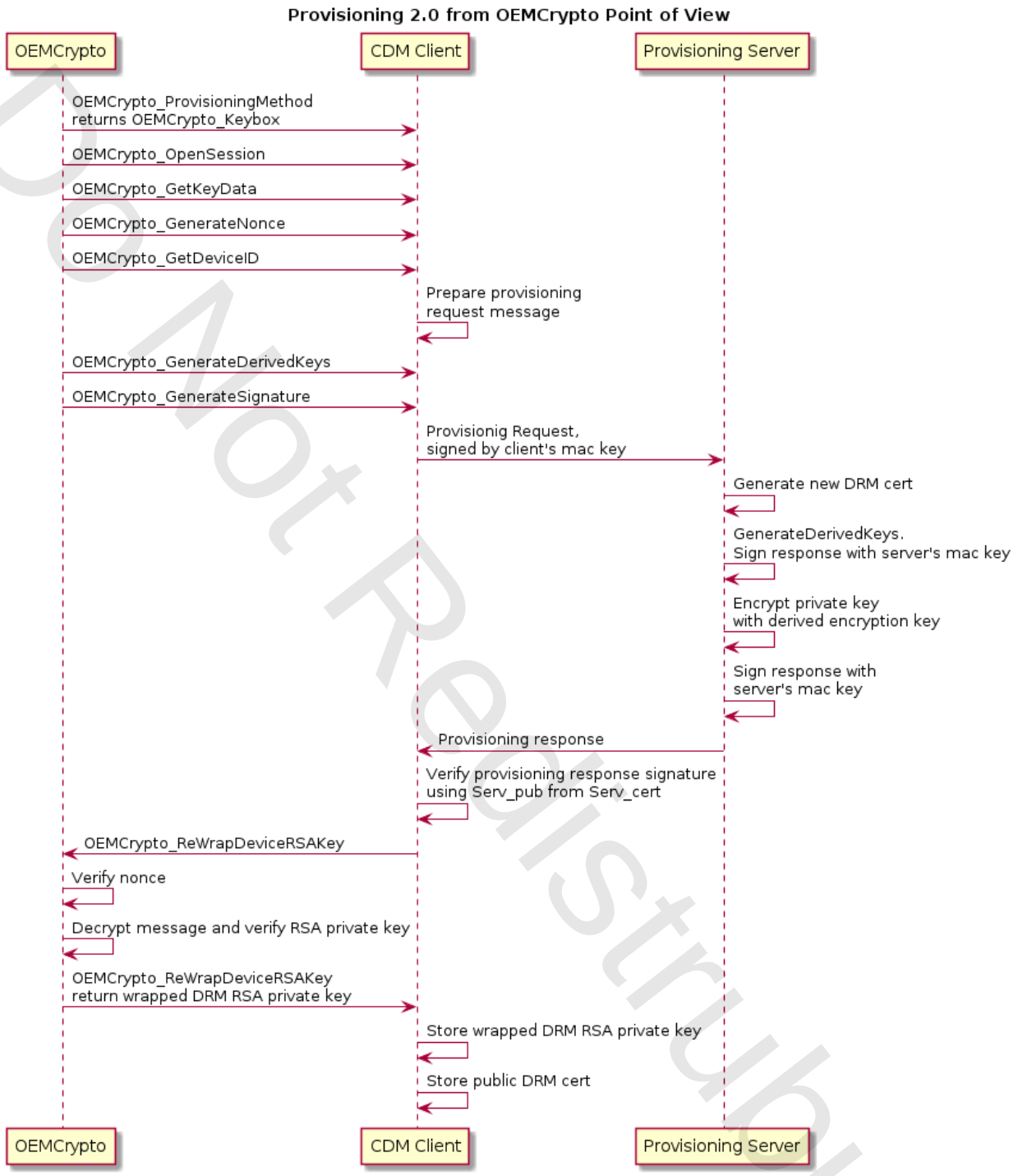
used to protect the contents of the keybox when it is stored. The device key in the keybox is used in the process of decrypting the media content played by the device. Google will support this provisioning method for the foreseeable future, but OEMs creating new devices are encouraged to use Provision 3.0 described below. Provision 3.0 is easier for OEMs to implement.

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 system ID in the keybox that ensures keyboxes are unique across manufacturers and device models. Two manufacturers 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 the 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, 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 provision, 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_RewrapDeviceRSAKey30() and make sure OEMCrypto_GenerateRSASignature 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_RewrapDeviceRSAKey30(). They can ignore the rest of this section.

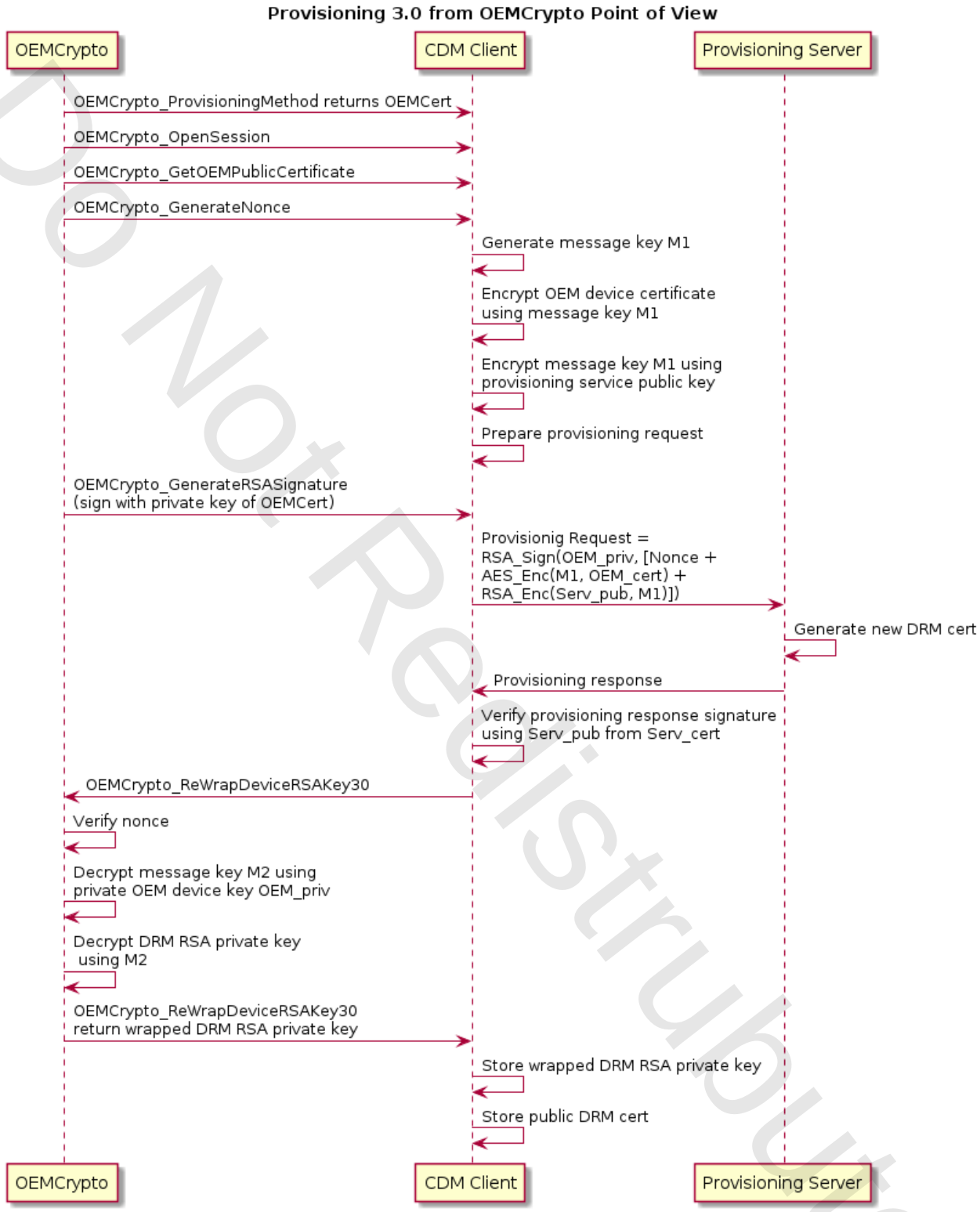
For a complete description of Provision 3.0, please see the document “Widevine Provisioning 3.0 Design”. OEMs will request a single X.509 CA certificate from Google for each make and model of the device, and use them to sign the device specific certificates which the OEM will generate for each device. 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 also load the private RSA key corresponding to the certificate when OEMCrypto_GetOEMPublicCertificate is called. It is the OEM’s responsibility to make sure that the private RSA key is not accessible to the user.

Using an OEM certificate will allow a device to talk to a provisioning server that is on a restricted network and is unable to connect to the Google provisioning server. Examples of disconnected/restricted networks are transportation industry, planes, trains, ships, and some geolocations which may have restricted access to the wider internet.

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 indicating a provisioning request 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.

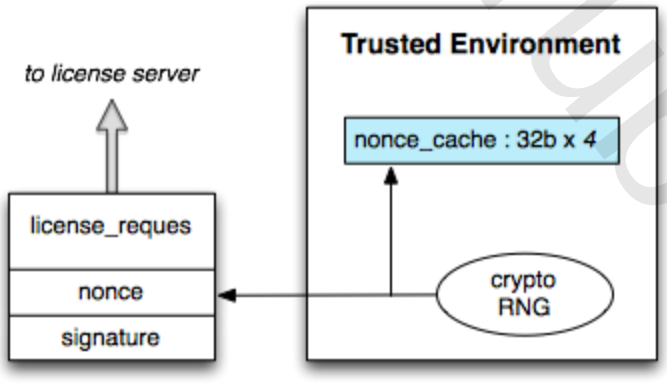
Content License Exchange and Renewal

There are two ways for content keys to be loaded into OEMCrypto -- via a content license, or from data that is controlled by an entitlement license. This section discusses content licenses and the next section talks about 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 application 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, GenerateDerivedKeys, GenerateNonce and GenerateSignature 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.

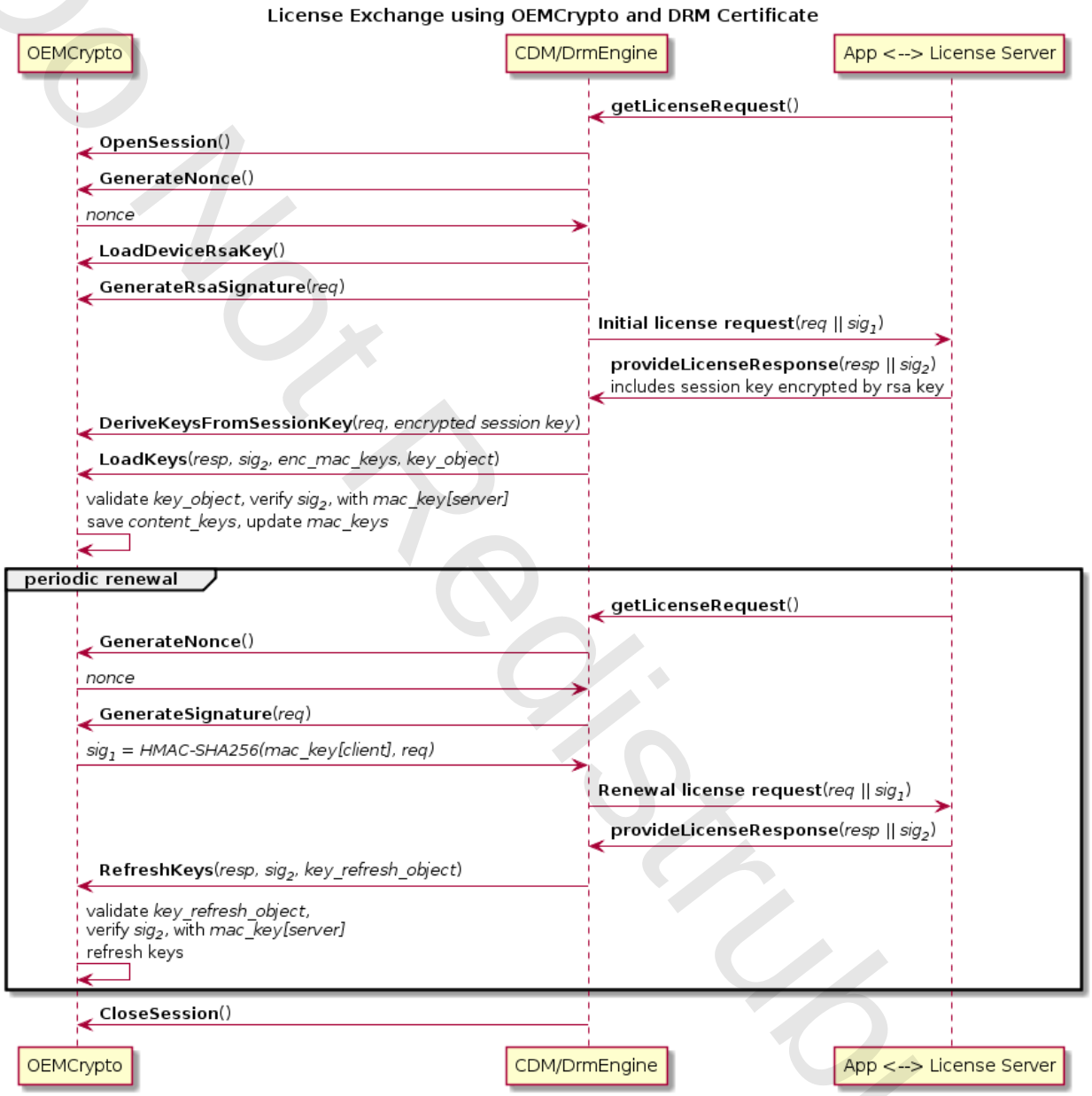
After the initial license has been processed, there is a periodic renewal request/response sequence that occurs during continued playback of the content. The OEMCrypto API calling sequence for renewal is similar to the sequence for the original license message, except that RefreshKeys is called instead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

For the license initial and renewal *requests*, the OEMCrypto implementation is required to generate a nonce and a signature that will be appended to the request. The nonce is used to prevent replay attacks. A nonce-cache is used to enforce one-time-use of each nonce. A nonce is added to the cache when created, and removed from the cache when used. 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.

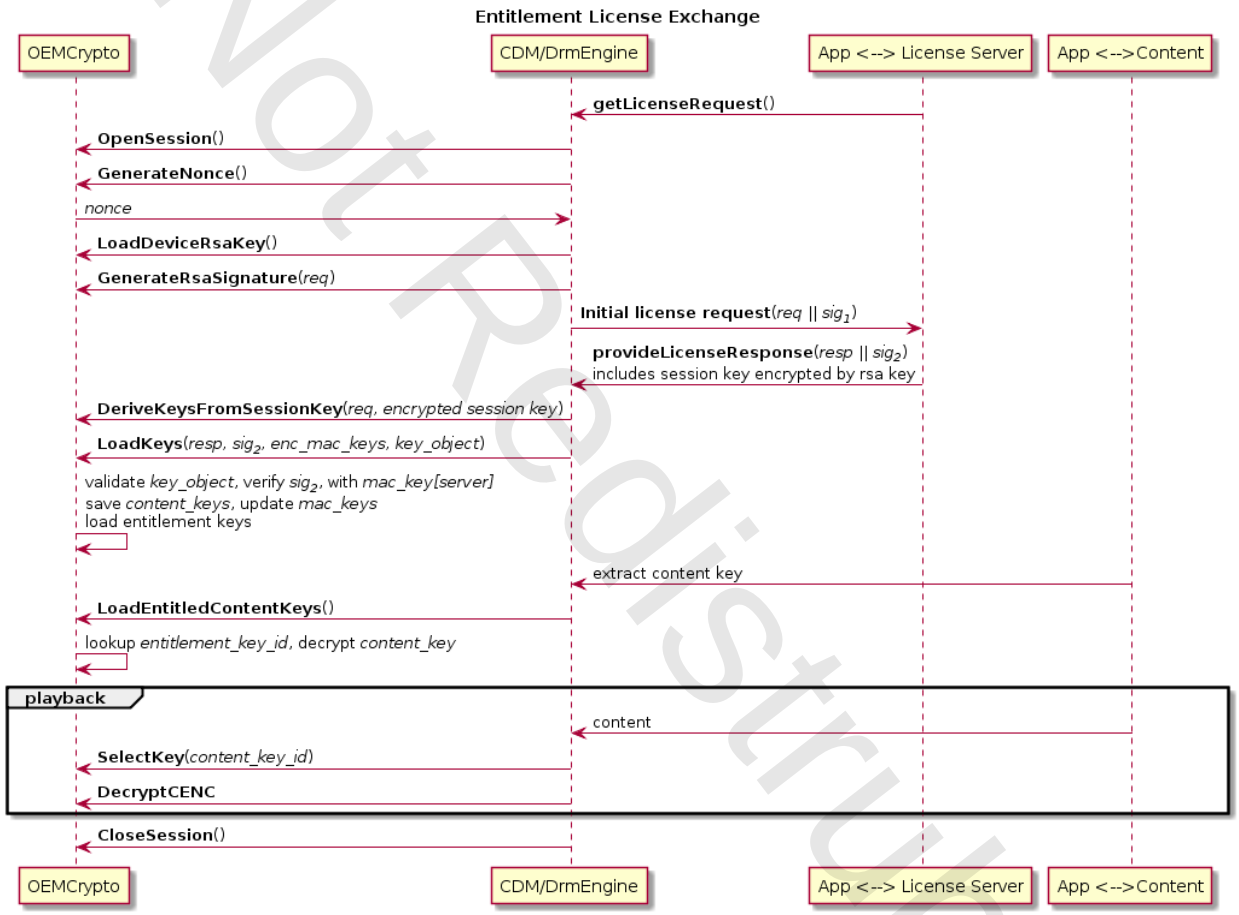


Entitlement License Exchange

An entitlement license is a way to group multiple content licenses together. Each piece of content in the group will have its content keys encrypted by an entitlement key and embedded in

the content. The device does not have to send a license request to a server for each piece of content. Instead, the entitlement key will be 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. 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 and entitlement key will share a duration and key control block.

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.



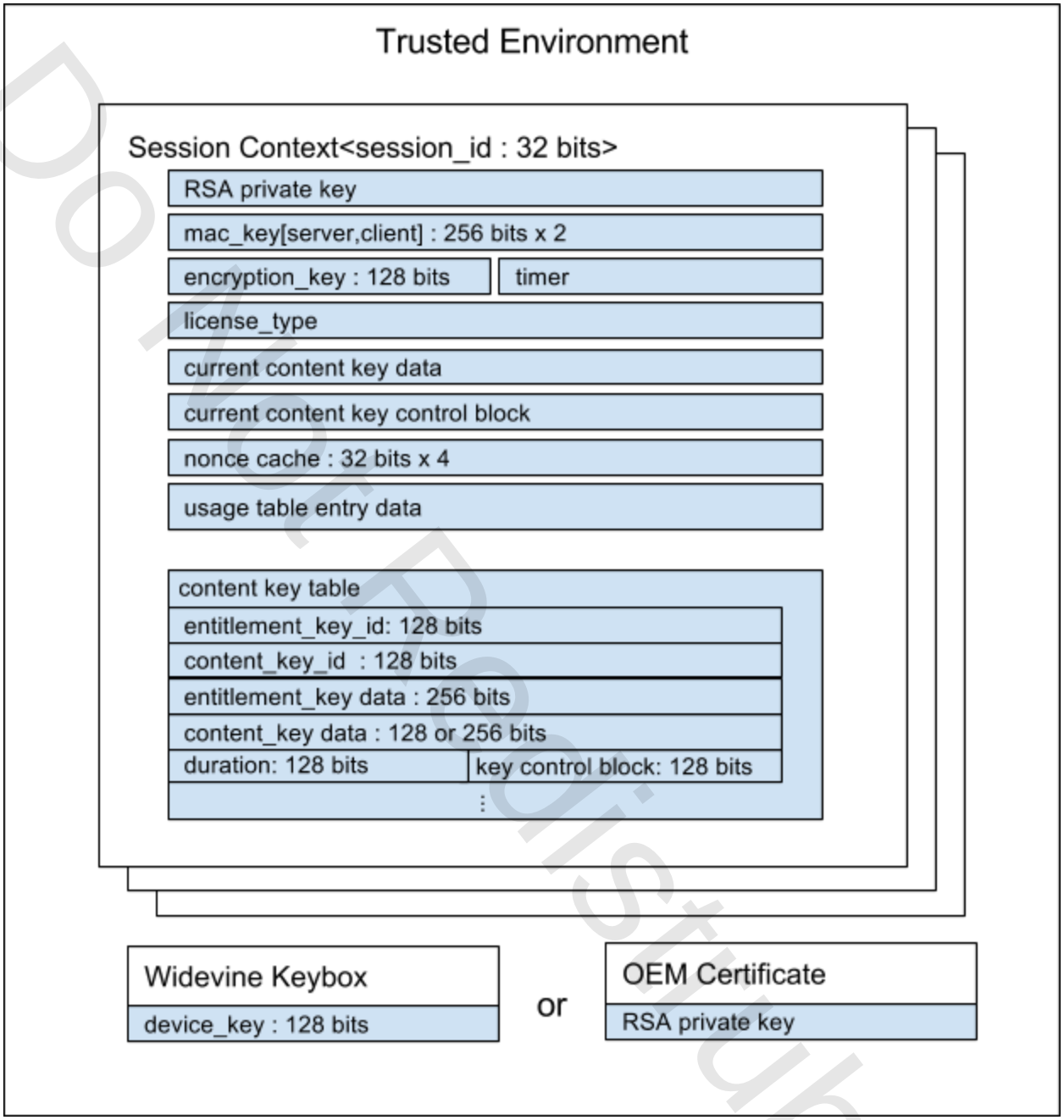
An entitlement license can also be renewed. The renewal process for an entitlement license is the same as that for a content license.

Session Context

One or more crypto sessions will be created to support A/V playback. An application may use a single session with multiple keys for all of its content. An application may use one session for video and a different session for audio. An application might also preload several license 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 signing 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 simultaneously 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 signing key (mac_key[server] 256 HMAC key) - used to verify messages signed by the server.
- Client HMAC signing key (mac_key[client] 256 HMAC key) - used to sign messages to the server.
- Encryption key (128 bit AES key) used to decrypt data from the server.
- Flag indicating if the license type is a content license or an entitlement license.
- Current content key (128 bit AES key or 256 HMAC key).
- Current content key control block (described below).
- Current content key duration limit.
- Table of keys, which contains
    - Key control blocks. Each key has its own control block because different restrictions or playback durations may apply to different keys.
    - Duration limit.
    - 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 for tracking key expiration.
- Nonce table.
- Usage table entry data. See the section on replay control below.



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.

Lifespan of Entitlement Keys

The entitlement license flow is designed to allow some flexibility in the applications ability to grant access to a variety of content by processing a single license. This is desirable because processing a license requires several RSA operations, which are expensive. A single entitlement license may cover several different pieces of content. For this reason, an entitlement license may have unused keys in the key table. Also, the content use key rotation, and the content keys for a given entitlement keys may be replaced.

Let us consider an example that illustrates why the number of entitlement keys may not match the number of content keys. There might be one entitlement key for HD video, one for SD video, and one for audio. A piece of content might have several keys embedded in it:

- One content key encrypted by the HD video entitlement key that is for decrypting HD video content.
- One content key encrypted by the SD video entitlement key that is for decrypting SD video content.
- One content key encrypted by the audio entitlement key that is for decrypting audio content.

Some content may have a subset of these keys -- for example some content may have only audio and SD video. In this case, the entitlement license might have all three keys, but only two content keys would be loaded into the session. Some entitlement licenses may have only a subset of these keys -- for example the customer did not pay for HD content. In this case, the entitlement license would only have two keys, and only two of the content keys will be loaded. It is the applications responsibility to only load those content keys it is entitled to.

Key Derivation

Communication with the server must be signed and partially encrypted. Each function below will specify if a signature or encryption is done using the session's RSA private key, or with a set of derived keys. Derived keys are AES keys derived from a context buffer using AES-128-CMAC and a secret input key. The server will use the same key and context to derive the same keys. OEMCrypto shall prevent user access of the input key and the derived keys.

The input context is usually generated from the message which will be sent to the server. 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)

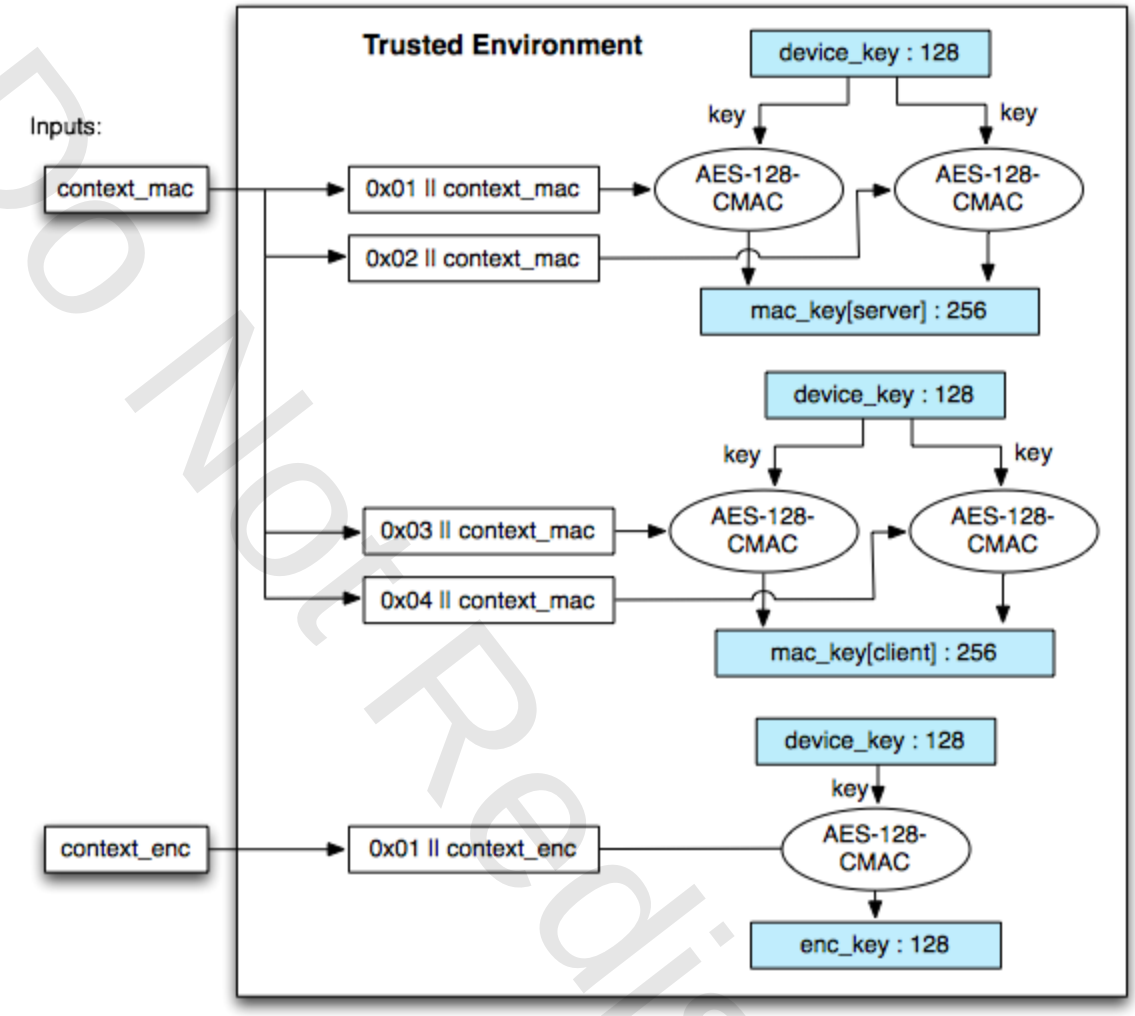
2. mac_keys: used as the hash key for the HMAC to sign and verify license messages:
   mac_key[server] || mac_key[client]

   := 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)

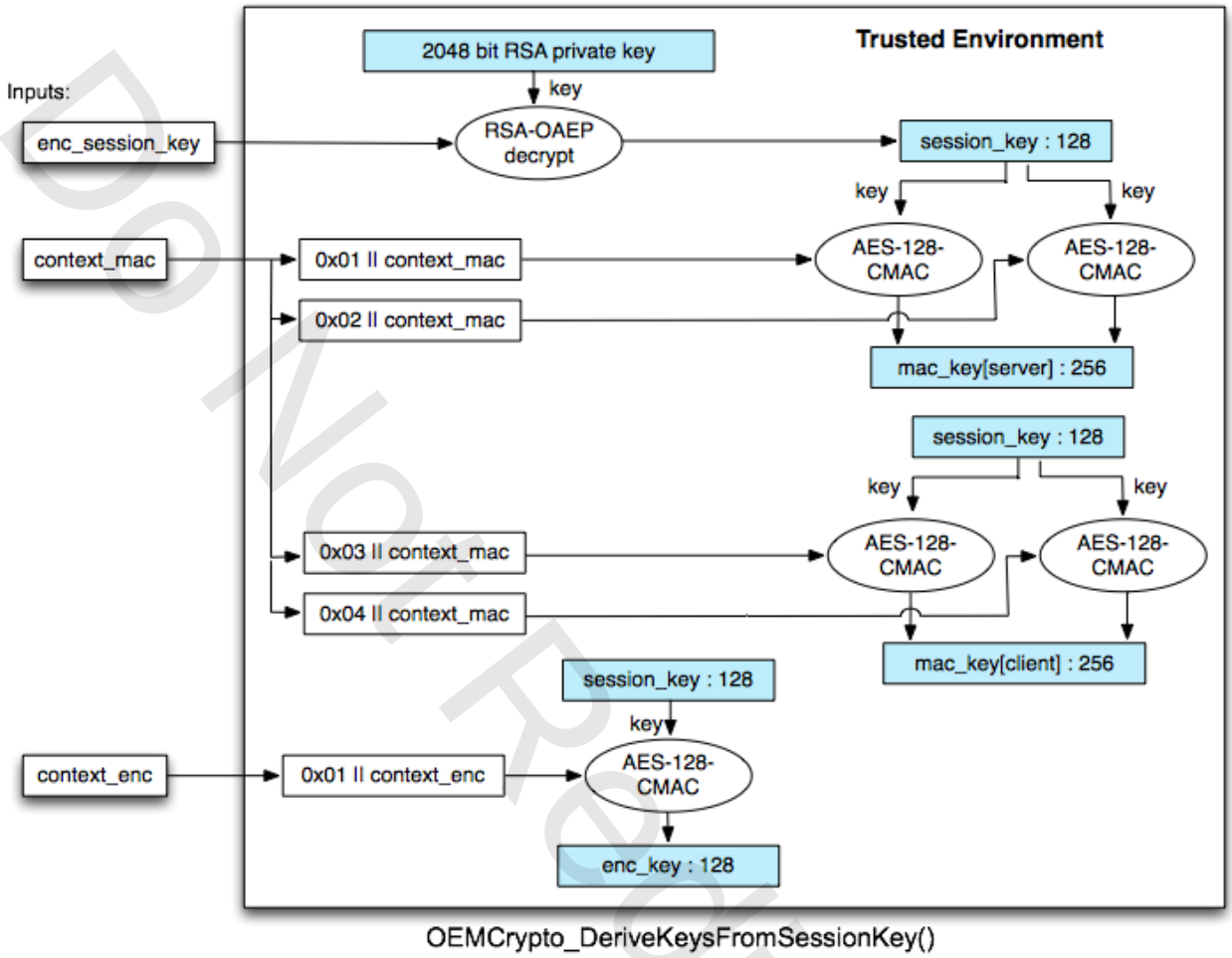
   For the case of license renewal, the mac_keys are generated by the license server, then encrypted and placed in a license response message, which is passed to OEMCrypto through OEMCrypto_LoadKeys. In this case the derivation is as follows:

   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()



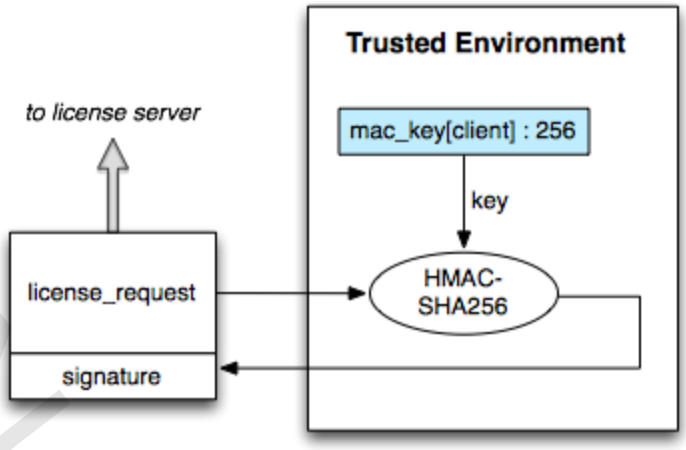
OEMCrypto_DeriveKeysFromSessionKey()

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

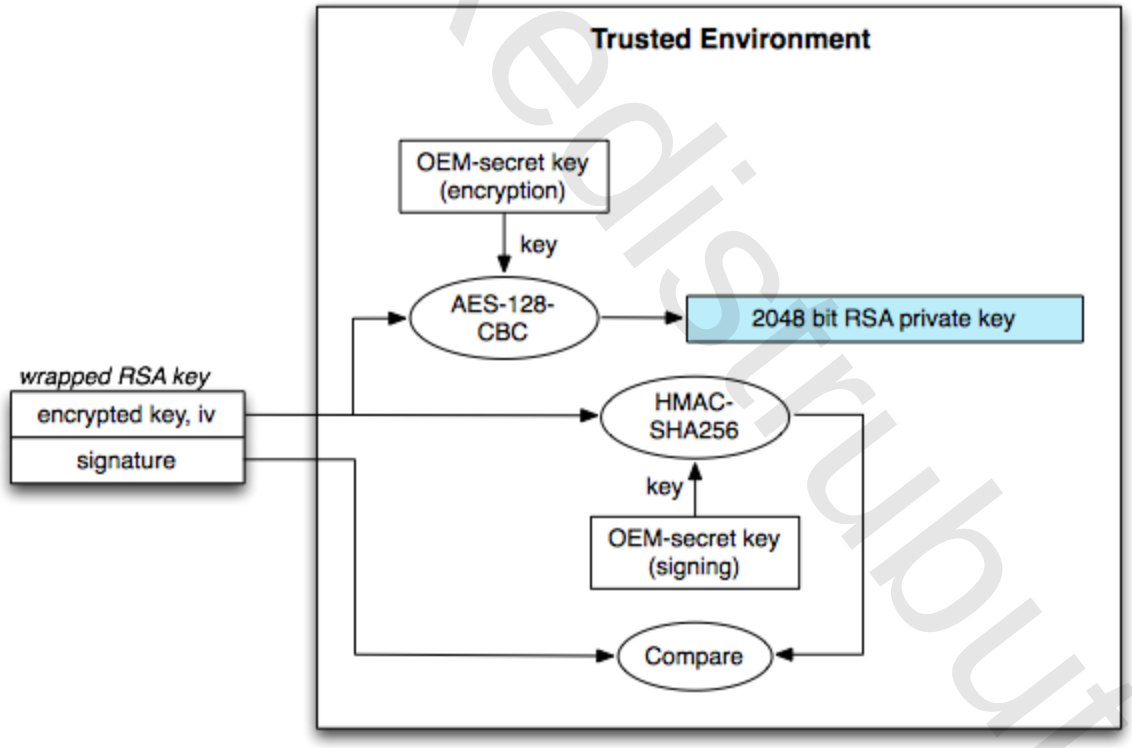
Messages sent to a server will be signed to ensure that the license request can not be modified in transit. Signing is done by OEMCrypto using either the session's derived mac_key[client] or the session's private RSA key. These functions specify a session id and should use the current RSA key or HMAC key for the specified session.

The function OEMCrypto_GenerateSignature should use the session's derived mac_key[client] to sign a buffer using the HMAC-SHA256 algorithm.



OEMCrypto_GenerateSignature()

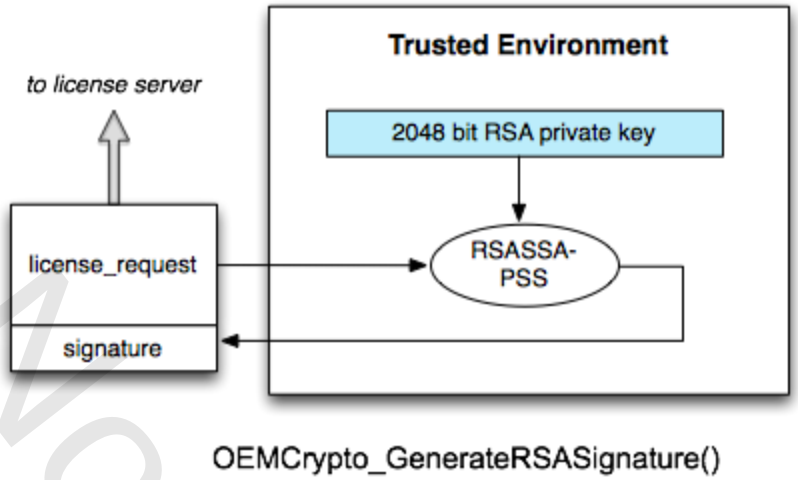
In order to sign a message using RSA, OEMCrypto will first be asked to load the private RSA key associated with a DRM Certificate. This will be passed into OEMCrypto_LoadDeviceRSAKey() as a blob of data that was previously wrapped by the function OEMCrypto_RewrapDeviceRSAKey() or OEMCrypto_RewrapDeviceRSAKey30().



OEMCrypto_LoadDeviceRSAKey()

The function OEMCrypto_GenerateRSASignature should use the session's private RSA key to

sign a buffer using the RSASSA-PSS algorithm.



Data in Messages from a Server

Several functions take pointers to data that came from the server. For each of these functions, the message and its signature are passed in, as well as pointers to data within the message buffer. OEMCrypto shall verify the signature of the message, as described below, and OEMCrypto shall verify that each the data is within the range of the message. In other words, the data pointers should be within the range of the message, and the pointer plus the length of the data should also be within the range of the message. 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.

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.

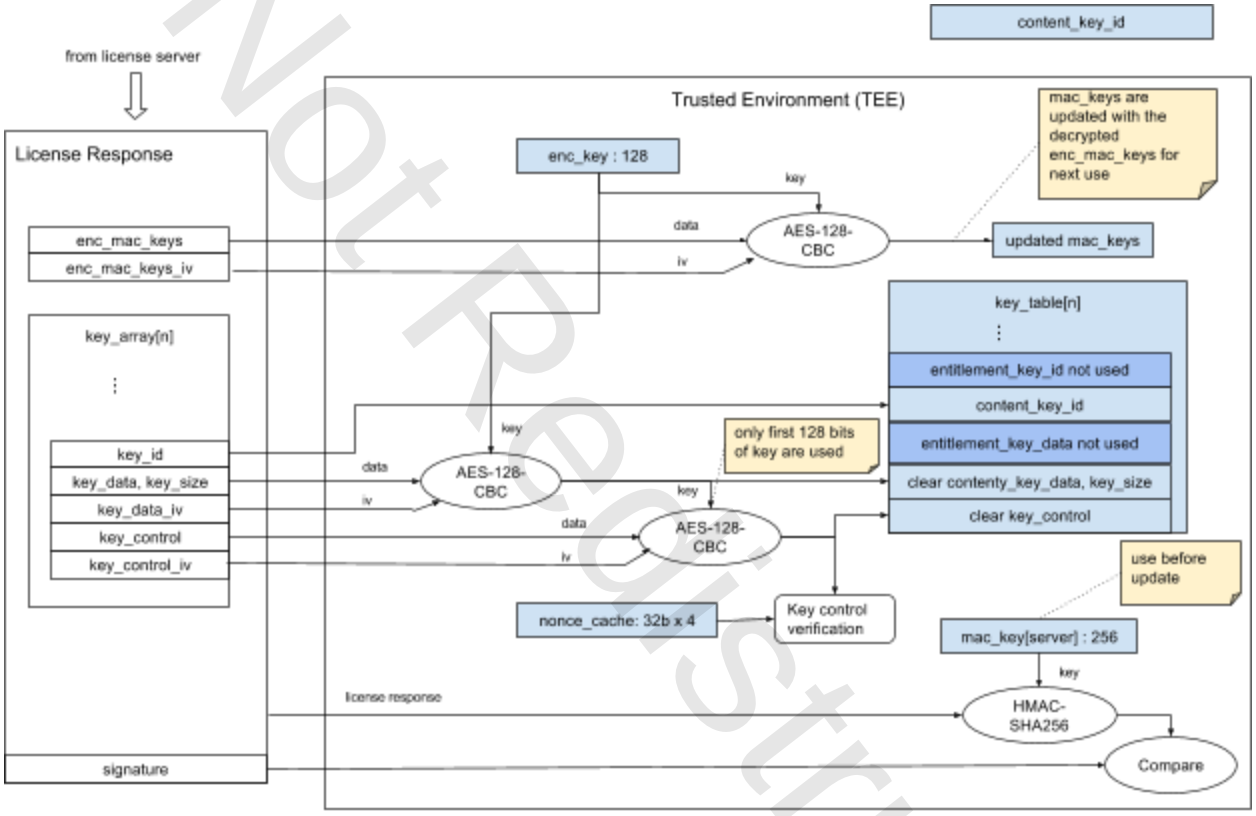
Note: When verifying the signature, the string comparison between the input signature and the recomputed signature should be a constant-time operation, to avoid leaking timing info.

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

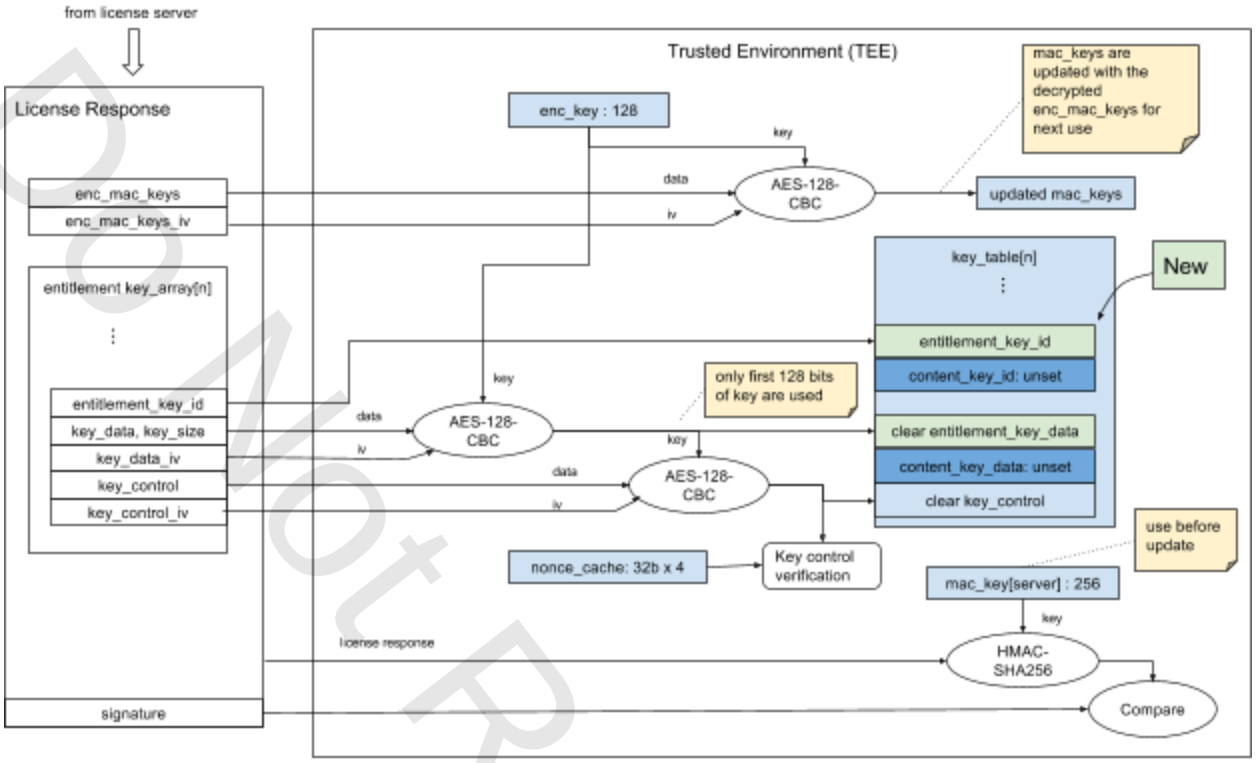
matches the message.

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 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.

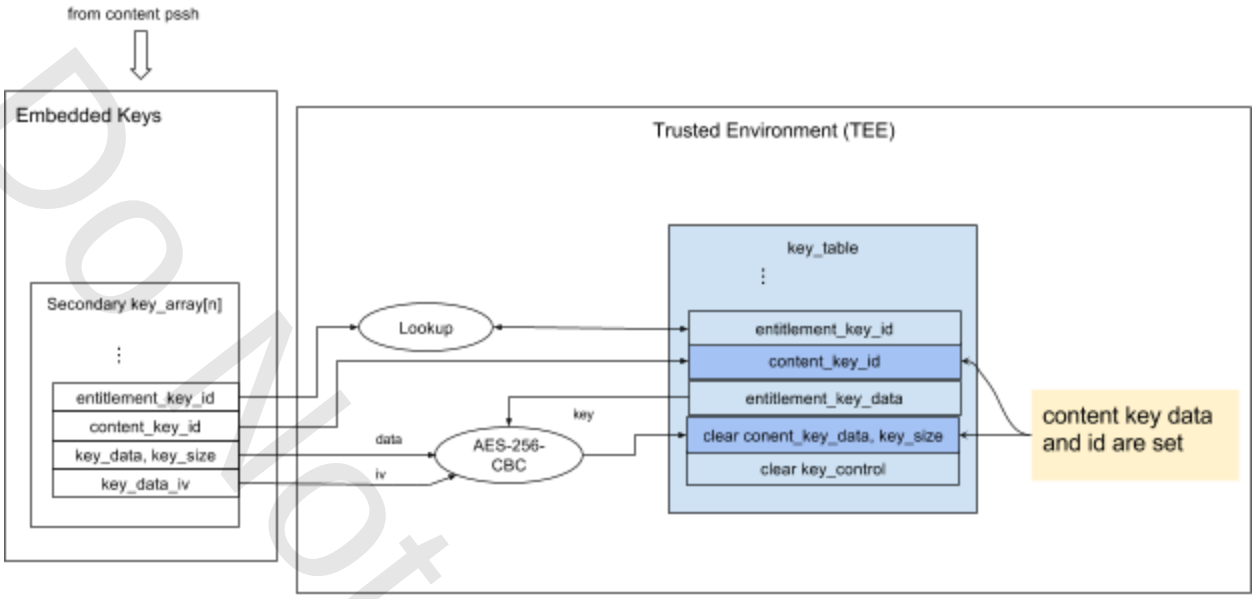


OEMCrypto_LoadKeys() with license_type = OEMCrypto_ContentLicense



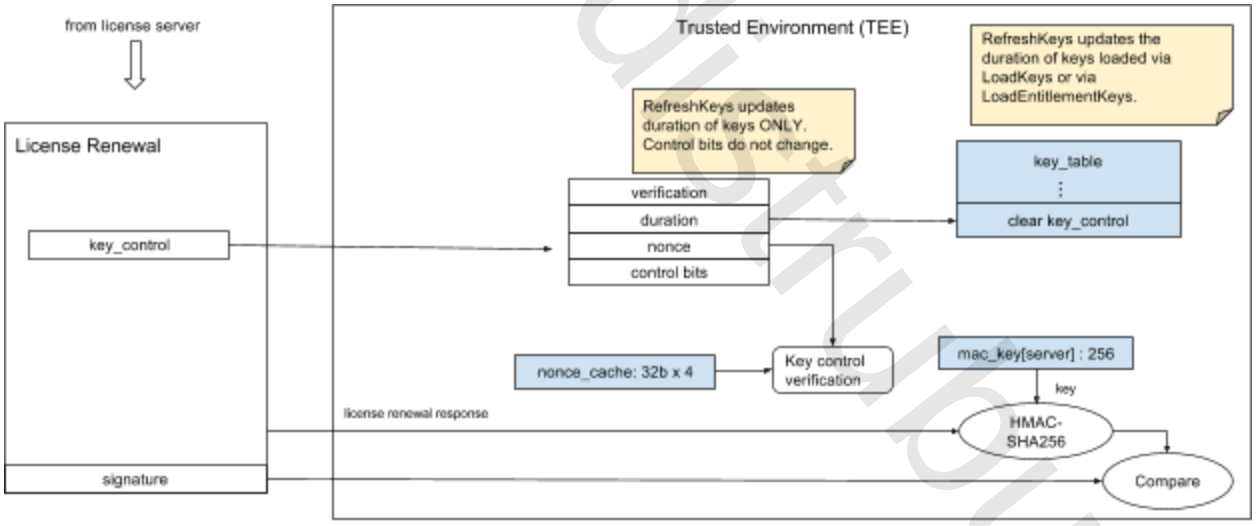
OEMCrypto_LoadKeys() with license_type = OEMCrypto_EntitlementLicense

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_RefreshKeys. 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 key control block associated with each content key. The key control block specifies

security constraints for the stream protected by each content key, which need to be enforced by the trusted environment. These security constraints include the data path security requirement, key validity lifetime and output controls.

On most devices, the video and audio paths have differing security requirements. While the video path can be entirely protected by hardware, the audio path 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 on 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.

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”, ... “kc14”. A device that supports the current version of this API must support all verification strings.	32
Duration	Maximum number of seconds during which the key can be used after being set. Interpret 0 as unlimited.	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..24	Reserved set to 0
bit 23	Shared_License If set, this is a shared license, and must follow a master license.
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 0xF - Local display only. The content should not be available to any external display, including HDMI, no matter what the HDCP level is.
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(). 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 ‘kc14’. If not, return OEMCrypto_ERROR_CONTROL_INVALID. In particular, it is important that devices to not accept key control blocks for future versions.
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.
10. Duration field -- on each DecryptCENC call for this session, compare elapsed time to this value. If elapsed time exceeds this setting and the key has not been renewed, return from the decrypt call with a return value of OEMCrypto_ERROR_KEY_EXPIRED. The elapsed time clock starts counting at 0 when OEMCrypto_LoadKeys is called, and is reset to 0 when OEMCrypto_RefreshKeys is called. Duration is in seconds. Each session will have a separate elapsed time clock.
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. If the generated value is already in the nonce cache, generate a new nonce value.
2. Nonce monitoring: the OEMCrypto implementation is responsible for checking the nonce in each call to OEMCrypto_LoadKeys(), OEMCrypto_RefreshKeys() and other functions that process data from the server, and rejecting any message whose nonce is not in the cache. If a nonce is in the cache, accept the message and remove the nonce from the cache.
3. Nonce expiration: A session should maintain at least 4 of the most recently generated nonces. Older nonce values should be removed.

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	1=Verify Nonce	n/a

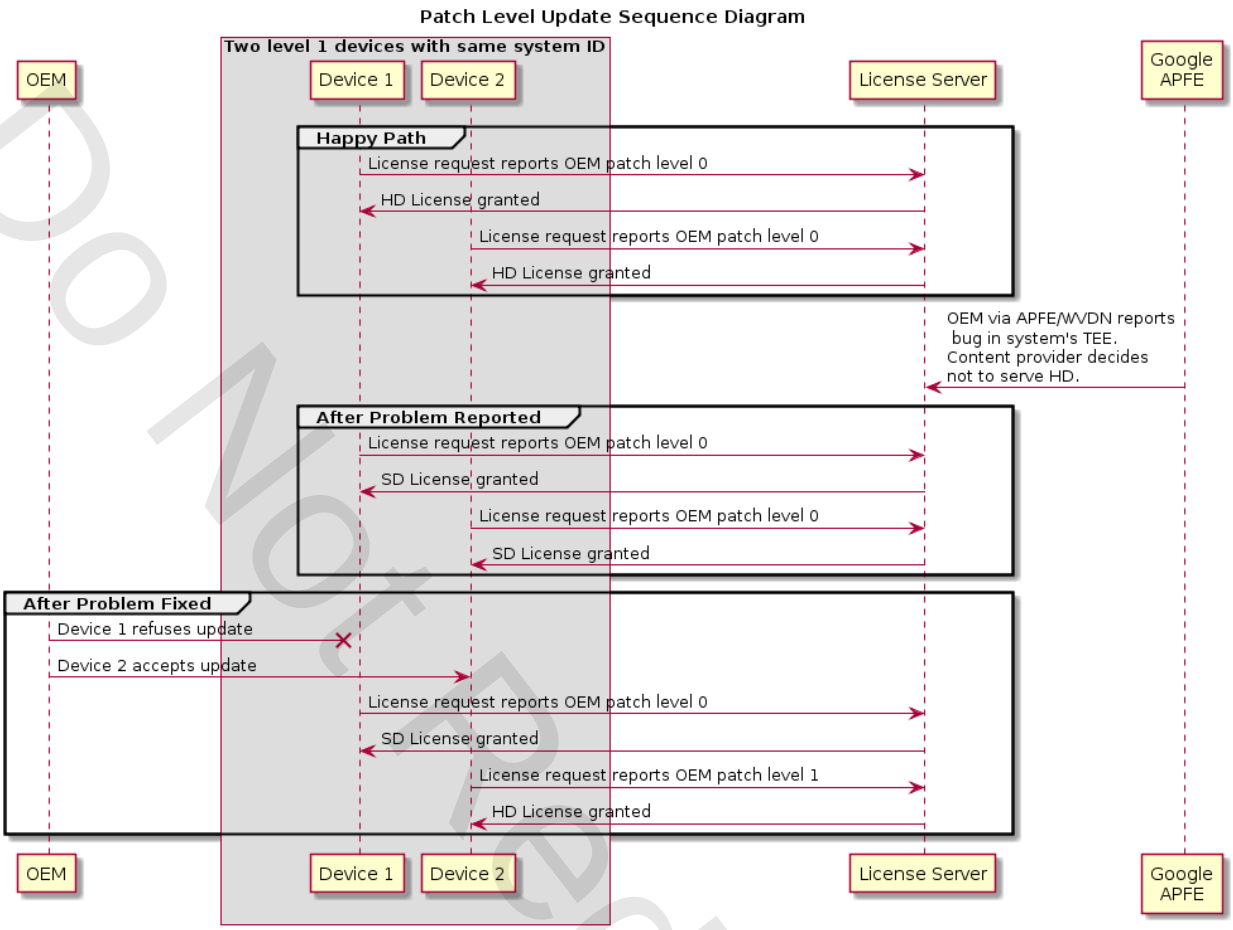
	Nonce		

Security Patch Level

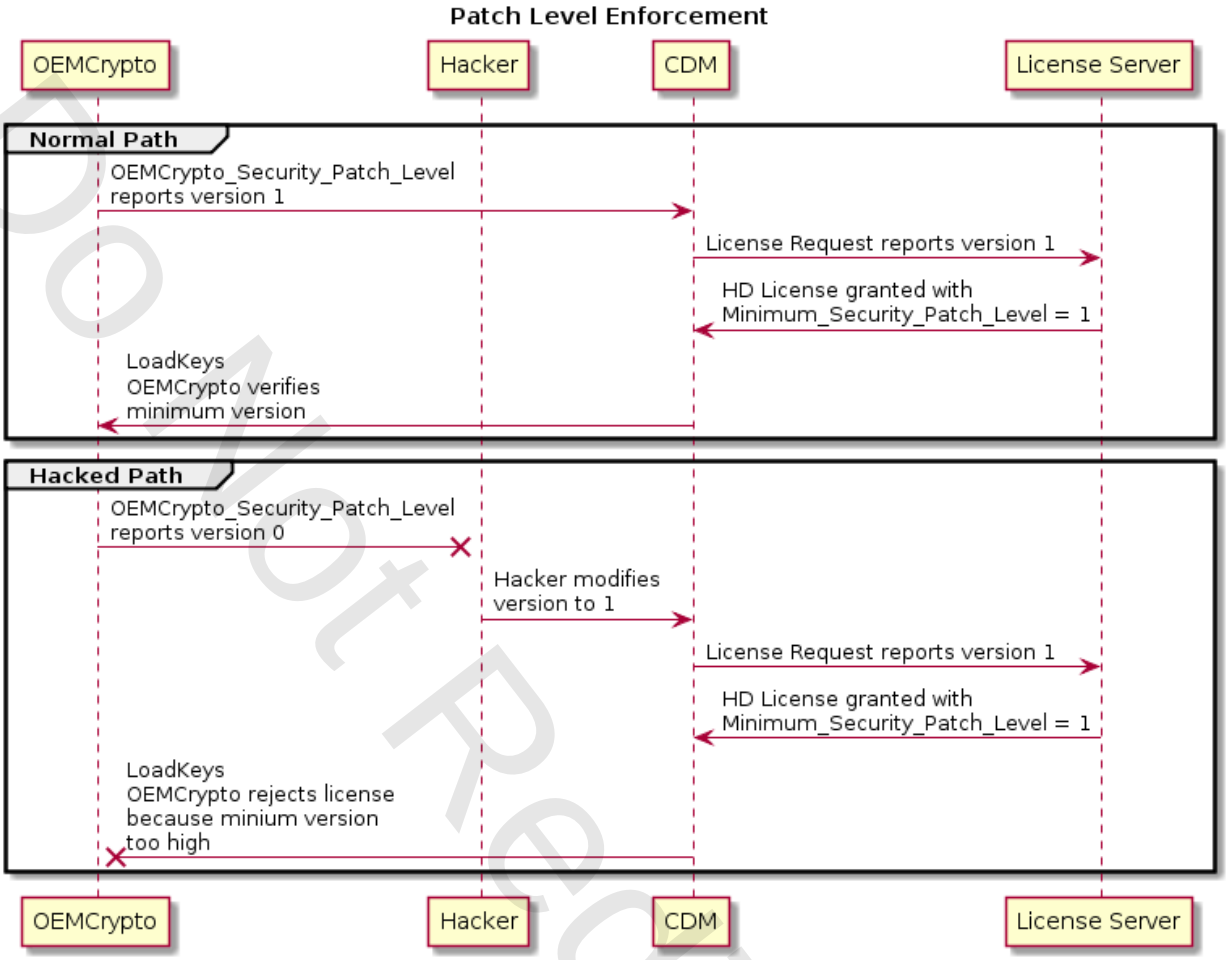
This feature addresses the desire of a content provider to serve licenses to a device only if it has a specific security patch. This feature allows the device to indicate that it has received a security patch. Notice that this feature will not distinguish between a device whose root of trust has been 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.

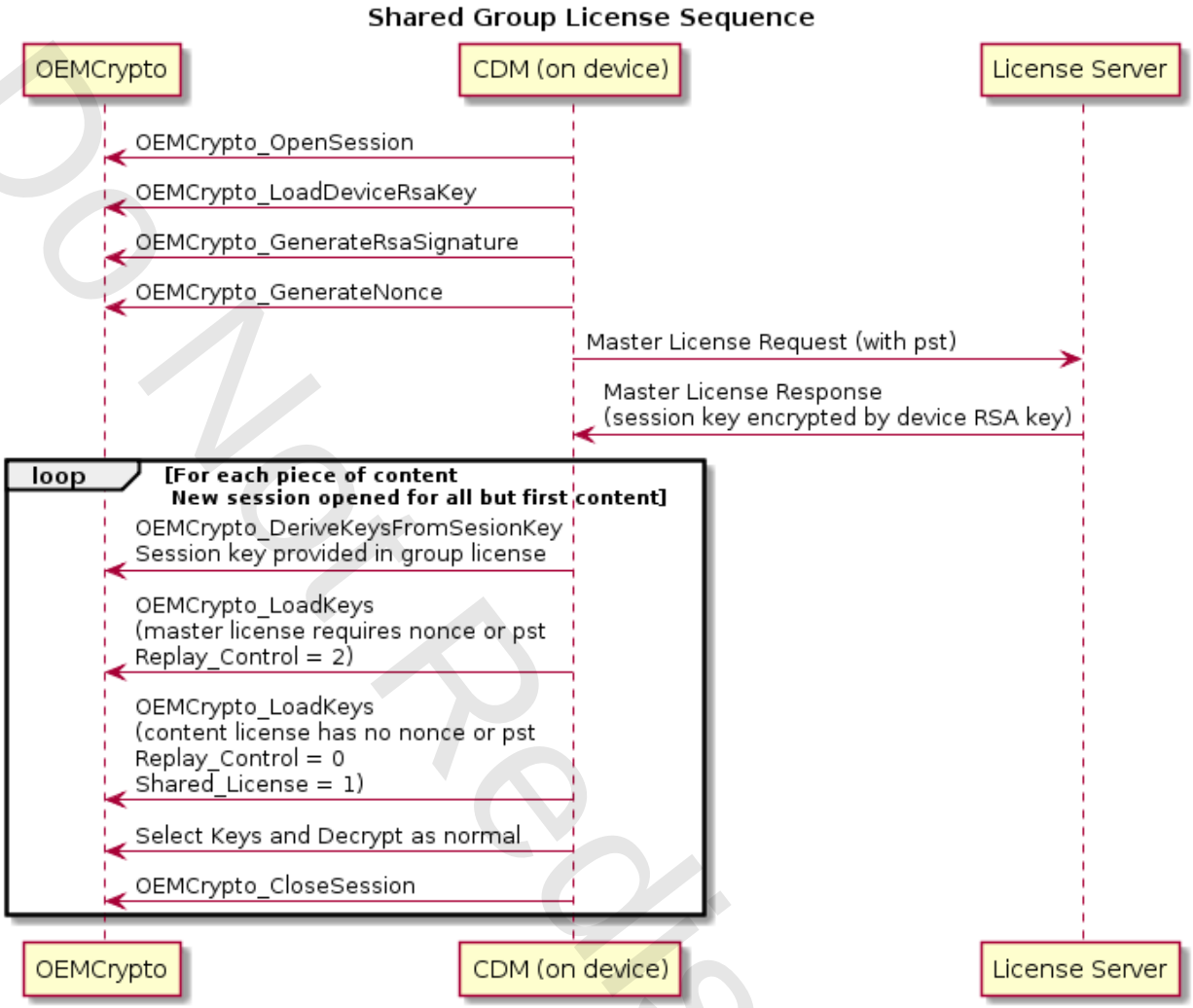


Here is a sequence diagram showing how OEMCrypto should behave in the normal case, and in the case where there is a man-in-the-middle.



Shared Group License

The term shared group license refers to a collection of content that share the same session key. The goal is to require a single round trip between the device and the license server for the master license, but allow devices to share the content licenses for content in that group. Each piece of content will have its own content keys. From OEMCrypto's point of view, the license request begins the same as a standard license request for an offline license. When the master license is loaded, it will update the mac keys. A second call to LoadKeys is made with the content license that is signed by these mac keys. Key control blocks in the second license will have the Shared_License bit set. Below is a sequence diagram.



Session Usage Table and Reporting

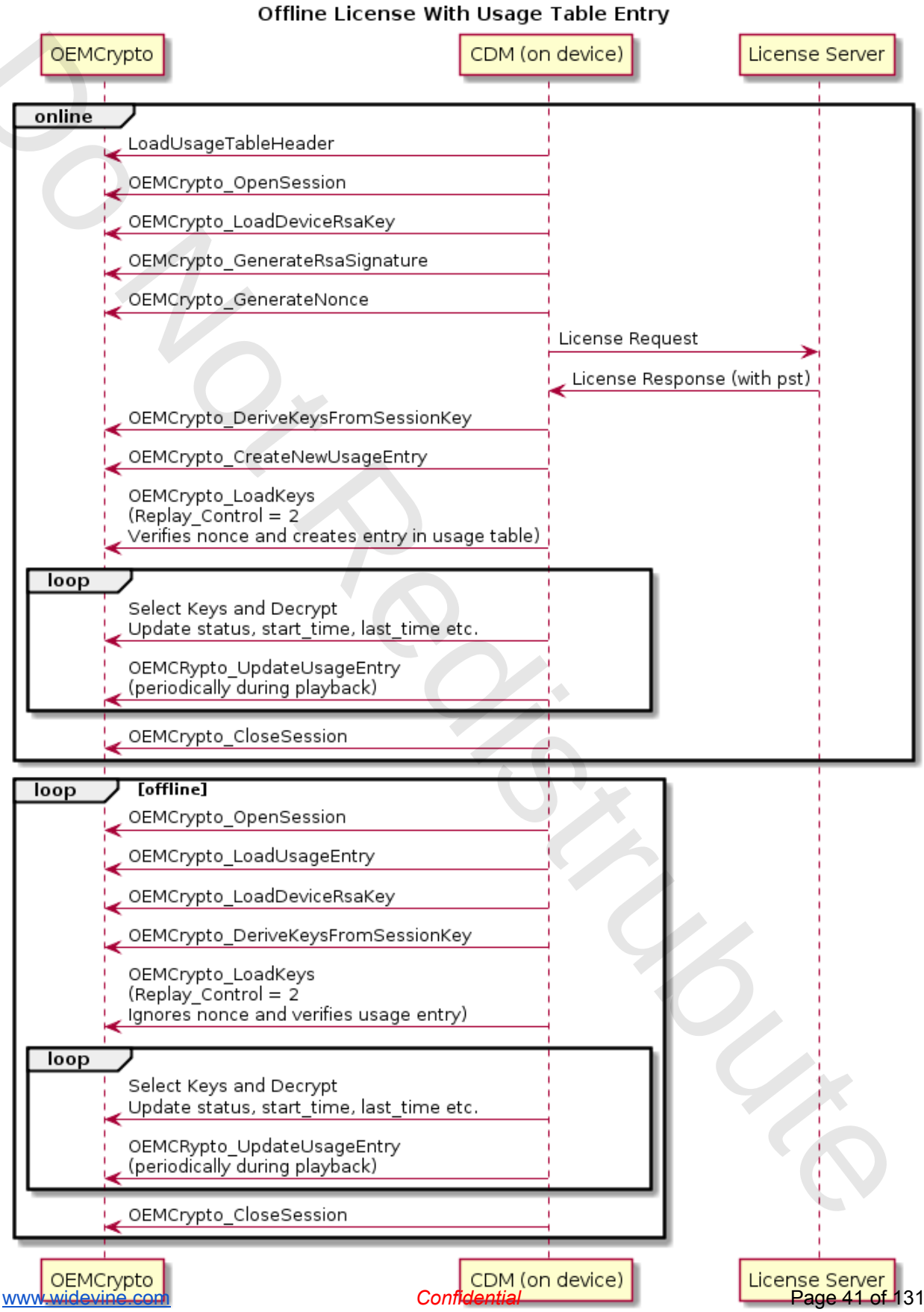
The Session Usage Table is a feature that has two main use cases. It is used to control reloading keys for offline playback, and for reporting secure stops for online playback. Both of these use cases require a Session Usage Table that stores persistent data securely, and a secure clock or timer that 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

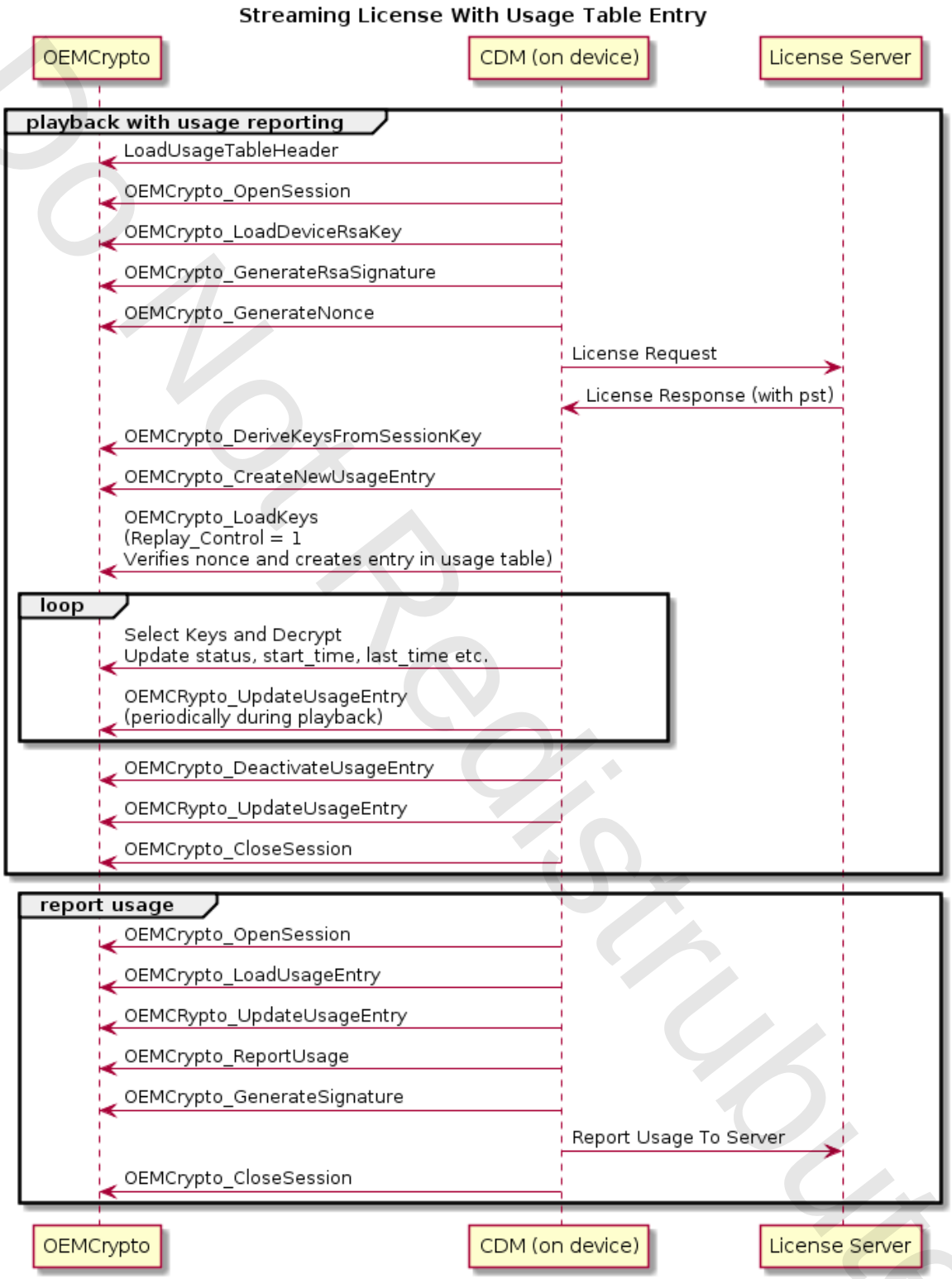
instead of using a nonce, and that session will be associated with the existing entry in the Usage Table.

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_received;  //set when loadKeys is called.
  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.

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.
}
```

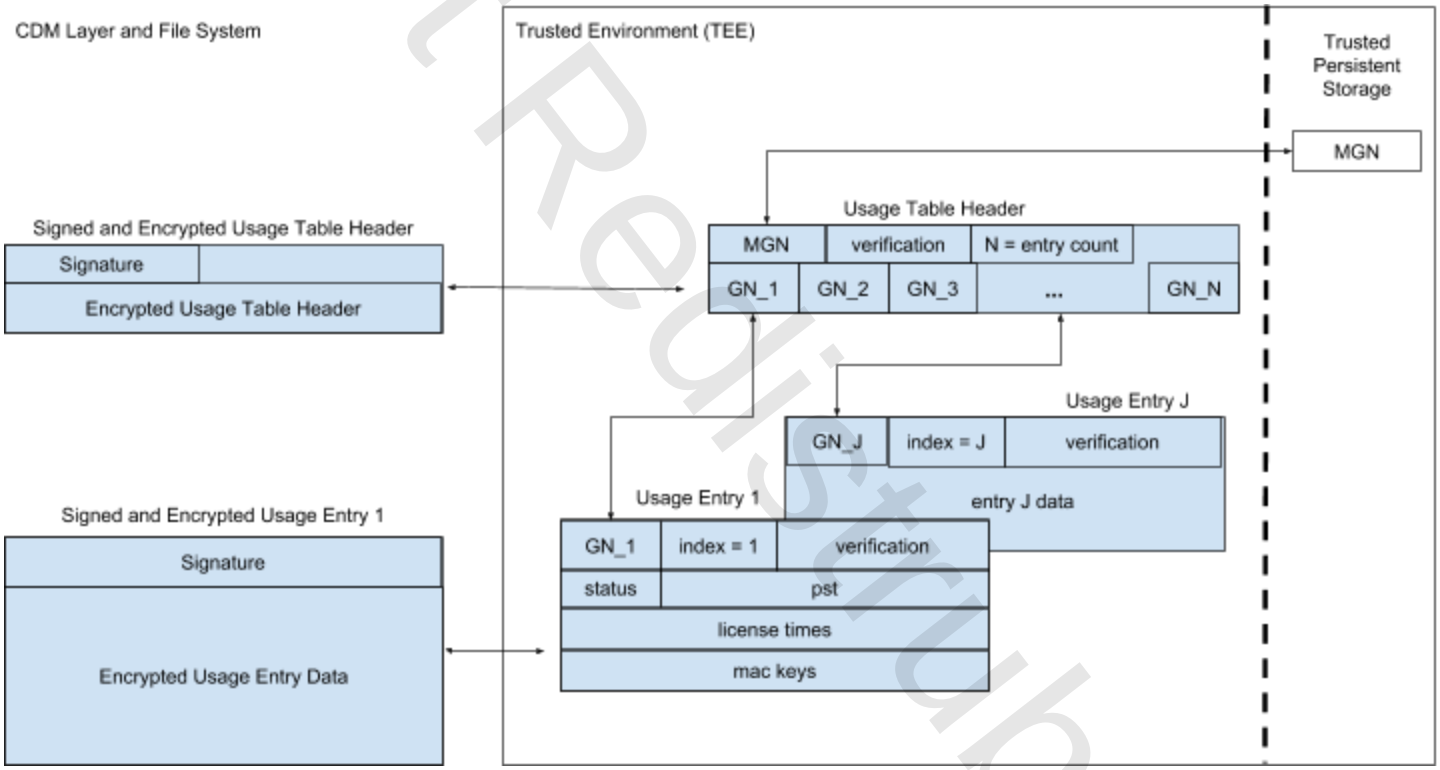
As with each usage entry, the header will be stored in protected memory by OEMCrypto. In order to persist the header, the CDM layer will ask OEMCrypto for an updated header. OEMCrypto will encrypt and sign the header and pass it back to the CDM layer. The CDM layer will be responsible for saving the 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

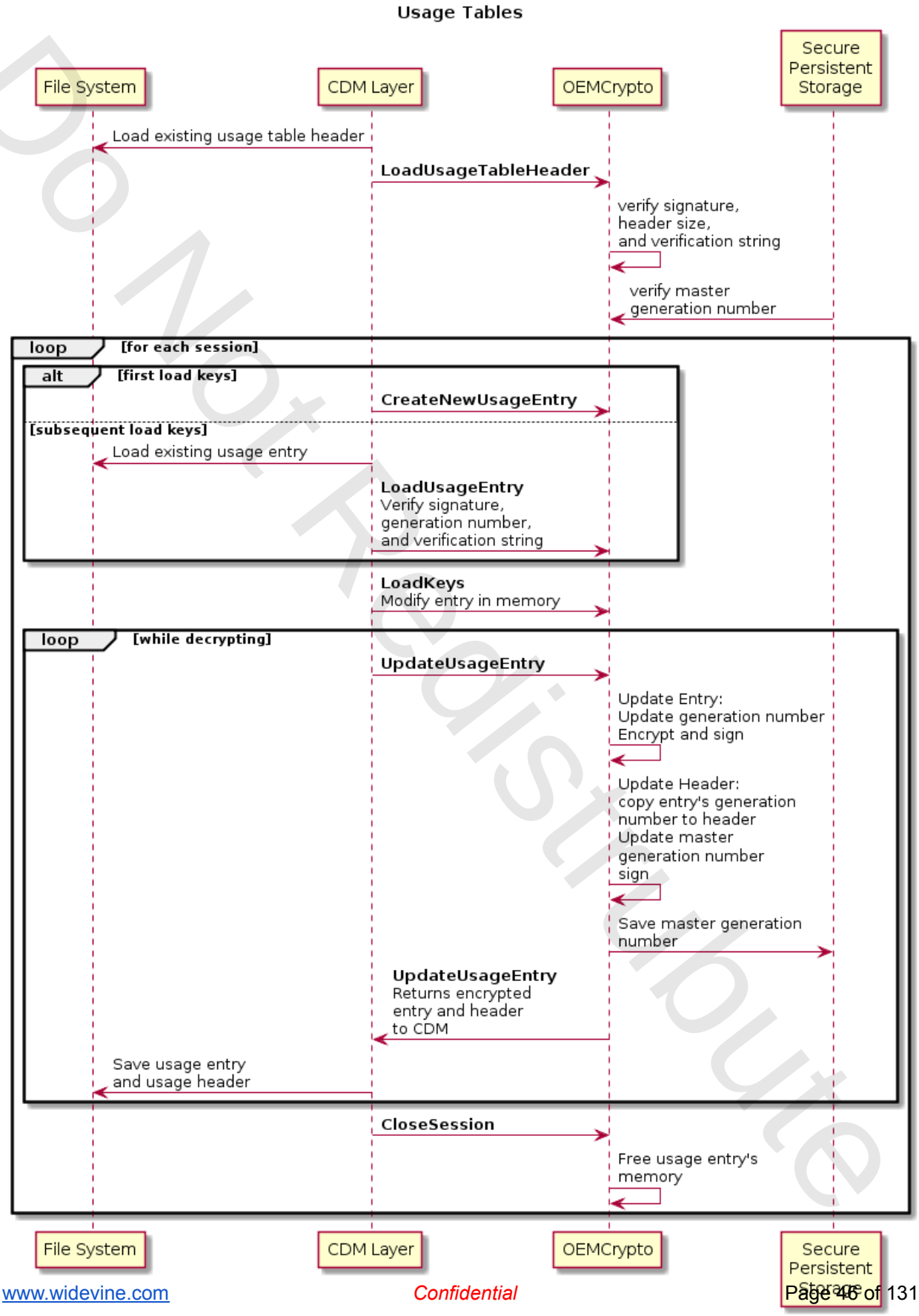
marked as inactive, OEMCrypto must prevent rollback of the data. In order to do this, OEMCrypto will mark each entry in the usage table with a generation number. This number should be the same as the entry's generation number in the usage table header. The usage table header has an array of generation 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.



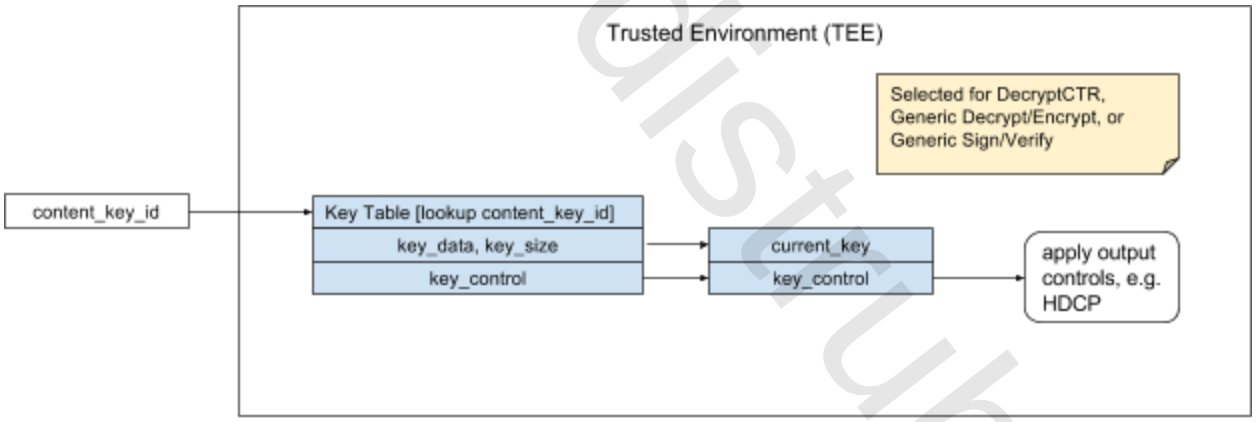
Entries in the table may have the following status values:

```
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 in active.

Content Decryption

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

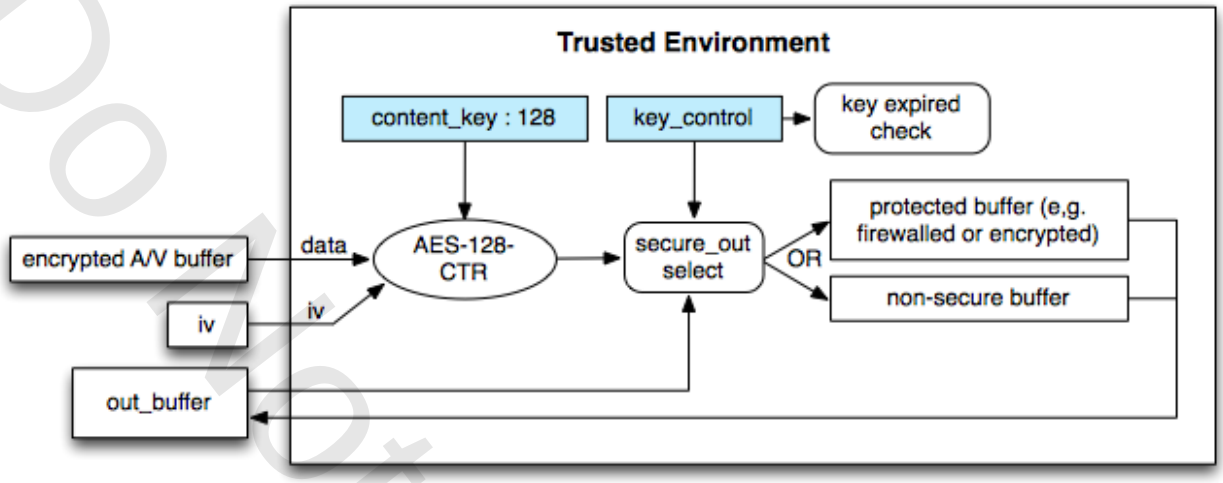


OEMCrypto_SelectKey()

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_LoadKeys. 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.

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.

- Keybox functionality. If OEMCrypto_GetKeyData is not implemented, then the device will not use a keybox to generate license requests, or to request a DRM certificate. These devices will need to have an RSA certificate installed separately.
- Certificate functionality. If OEMCrypto_GenerateRSASignature is not implemented, then it will not use a DRM certificate to generate license requests. Many content providers prefer to use DRM certificates to generate license requests because it allows them to use a stand-alone server instead of relaying requests to a Widevine server. All devices must either have a keybox or support DRM certificates. Most platforms will support both.

- Load Certificate functionality. If a device does have a keybox, but does not implement OEMCrypto_RewrapDeviceRSAKey, it will not be able to request an DRM certificate from the Widevine provisioning server. This essentially makes it unable to use DRM certificates.
- Generic Crypto. If Generic_Encrypt is not implemented, then the generic cryptographic API is not tested. Some applications use modular DRM functionality and root of trust to send secure data, such as business data or account data, from the application to the server. These functions are not used to play DRM protected video or audio.
- Usage Tables. Usage tables are a way to store usage information and track validity of offline licenses. If a device does not support usage tables, it will not be able to process secure stops or securely report termination of an offline license. Content providers may limit HD licenses to such devices.
- HDCP SRM Updates. Devices that support HDCP v2.2 or higher should support these functions. Devices that do not, do not have to implement OEMCrypto_LoadSRM or OEMCrypto_GetCurrentSRMVersion. Devices with a local display only, i.e. no video output ports, may return OEMCrypto_LOCAL_DISPLAY_ONLY from OEMCrypto_GetCurrentSRMVersion.

OEMCrypto API for CENC

The OEMCrypto API is defined in the file OEMCryptoCENC.h.

There are several areas exposed by OEMCrypto APIs:

- Crypto Device Control API
- Crypto Key Ladder API
- Decryption API
- Keybox Access and Provisioning 2.0 API
- OEM Certificate Access and Provisioning 3.0 API
- Validation and Feature Support API
- DRM Certificate Provisioning API
- Usage Table API

Crypto Device Control API

The Crypto Device Control API involves initialization of and mode control for the security hardware. The following list shows the device control methods:

OEMCrypto_Initialize

OEMCrypto_Terminate

OEMCrypto_Initialize

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

Initializes the crypto hardware.

Parameters

None

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INIT_FAILED failed to initialize crypto hardware

Threading

No other function calls will be made while this function is running. This function will not be called again before OEMCrypto_Terminate().

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

No other OEMCrypto calls are made while this function is running. After this function is called, no other OEMCrypto calls will be made until another call to OEMCrypto_Initialize() is made.

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.

The following list shows the APIs required for key management:

- OEMCrypto_OpenSession
- OEMCrypto_CloseSession
- OEMCrypto_GenerateDerivedKeys
- OEMCrypto_DeriveKeysFromSessionKey
- OEMCrypto_GenerateNonce
- OEMCrypto_GenerateSignature
- OEMCrypto_LoadSRM
- OEMCrypto_LoadKeys
- OEMCrypto_LoadEntitledContentKeys
- OEMCrypto_RefreshKeys
- OEMCrypto_QueryKeyControl

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.

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.

Threading

No other Open/Close session calls will be made while this function is running. Functions on other existing sessions may be called while this function is active.

Version

This method changed in API version 5.

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.

Threading

No other Open/Close session calls will be made while this function is running. Functions on other existing sessions may be called while this function is active.

Version

This method changed in API version 13.

OEMCrypto_GenerateDerivedKeys

```
OEMCryptoResult OEMCrypto_GenerateDerivedKeys(OEMCrypto_SESSION session,
                            const uint8_t *mac_key_context,
                            uint32_t mac_key_context_length,
                            const uint8_t *enc_key_context,
                            uint32_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

Buffer Sizes

OEMCrypto shall support mac_key_context and enc_key_context sizes of at least 8 KiB.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffers are too large.

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

OEMCrypto_DeriveKeysFromSessionKey

```
OEMCryptoResult OEMCrypto_DeriveKeysFromSessionKey(
                      OEMCrypto_SESSION session,
                      const uint8_t* enc_session_key,
                      size_t enc_session_key_length,
```

```
    const uint8_t *mac_key_context,
    size_t mac_key_context_length,
    const uint8_t *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 LoadKeys. The session key is passed in encrypted by the device RSA public key, and must be decrypted with the RSA private key before use.

Once the enc_key and mac_keys have been generated, all calls to LoadKeys and RefreshKeys 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] enc_session_key: session key, encrypted with the public RSA key (from the DRM certifcate) using RSA-OAEP.
n_key_l[in] enc_sessioength: length of session_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

Buffer Sizes

OEMCrypto shall support mac_key_context and enc_key_context sizes of at least 8 KiB.

OEMCrypto shall return OEMCrypto_ERROR_BUFFER_TOO_LARGE if the buffers are too large.

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

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 for the next call to LoadKeys.

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. With this in mind, if more than 20 nonces are requested within one second, OEMCrypto will return an error after the 20th 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.

Parameters

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

Results

nonce: the nonce is also stored in secure memory. At least 4 nonces should be stored for each session.

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INVALID_SESSION

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 5.

OEMCrypto_GenerateSignature

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

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 the most recent successful call to any one of

- OEMCrypto_GenerateDerivedKeys,
- OEMCrypto_DeriveKeysFromSessionKey,
- OEMCrypto_LoadKeys, or
- OEMCrypto_LoadUsageEntry.

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 set to zero, this function returns OEMCrypto_ERROR_SHORT_BUFFER and sets output signature_length to the size needed to receive the output signature.

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: pointer to memory to received the computed signature. May be null (see note above).

[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 buffer.

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_UNKNOWN_FAILURE

OEMCrypto_ERROR_BUFFER_TOO_LARGE

Buffer Sizes

OEMCrypto shall support message sizes of at least 8 KiB.

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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

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

Buffer Sizes

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

Threading

This function will not be called at the same time as other functions.

Version

This method changed in API version 13.

OEMCrypto_LoadKeys

```
OEMCryptoResult OEMCrypto_LoadKeys(OEMCrypto_SESSION session,
                                   const uint8_t* message,
                                   size_t message_length,
                                   const uint8_t* signature,
                                   size_t signature_length,
                                   const uint8_t* enc_mac_keys_iv,
                                   const uint8_t* enc_mac_keys,
                                   size_t num_keys,
                                   const OEMCrypto_KeyObject* key_array,
                                   const uint8_t* pst,
                                   size_t pst_length,
                                   const uint8_t *srm_restriction_data,
                                   OEMCrypto_LicenseType license_type);
typedef enum OEMCrypto_LicenseType {
  OEMCrypto_ContentLicense = 0,
  OEMCrypto_EntitlementLicense = 1
};

typedef struct {
    const uint8_t* key_id;
    size_t         key_id_length;
    const uint8_t* key_data_iv;
    const uint8_t* key_data;
    size_t         key_data_length;
    const uint8_t* key_control_iv;
    const uint8_t* key_control;
} OEMCrypto_KeyObject;

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

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

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 it is not null, 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]. If enc_mac_keys is null, then there will not be a call to OEMCrypto_RefreshKeys for this session and the current mac_keys should remain unchanged.

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 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_GenerateDerivedKeys(), OEMCrypto_DeriveKeysFromSessionKey(), or a previous call to OEMCrypto_LoadKeys().

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.

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. The enc_mac_keys pointer must be either null, or point inside the message. If the pointer `enc_mac_keys` is not null, the API shall verify that the two pointers `enc_mac_keys_iv` and `enc_mac_keys` point to locations in the message. I.e. `(message <= p && p+p_length <= message+message_length)` for `p` in each of `enc_mac_keys_iv, enc_mac_keys`. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
3. The API shall verify that each pointer in each KeyObject points to a location in the message. I.e. `(message <= p && p+p_length <= message+message_length)` for `p` in each of `key_id, key_data_iv, key_data, key_control_iv, key_control`. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
4. Each key's control block, after decryption, shall have a valid verification field. If not, return OEMCrypto_ERROR_INVALID_CONTEXT.
5. 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.
6. 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.
7. If the key control block has a nonzero Replay_Control, then the verification described below is also performed.
8. 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.
9. If num_keys == 0, then return OEMCrypto_ERROR_INVALID_CONTEXT.
10. If any key control block has the Shared_License bit set, and this call to LoadKeys is not replacing keys loaded from a previous call to LoadKeys, then the keys are not loaded, and the error OEMCrypto_ERROR_MISSING_MASTER is returned.
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.

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 pointer `pst` must not be null, and must point to a region in the message, i.e. (message <= pst && pst+pst_length <= message+message_length). 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.

1. The pointer srm_restriction_data must be non null and must point to a location in the message. I.e. `(message <= p && p+12 <= message+message_length)`. 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_key_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] num_keys: number of keys present.

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

[in] pst: the Provider Session Token.

[in] pst_length: the length of pst.

[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

Buffer Sizes

OEMCrypto shall support message sizes of at least 8 KiB.

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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 14.

OEMCrypto_LoadEntitledContentKeys

```
OEMCryptoResult OEMCrypto_LoadEntitledContentKeys(
    OEMCrypto_SESSION session,
    size_t num_keys,
    const OEMCrypto_EntitledContentKeyObject* key_array);

typedef struct {
  const uint8_t* entitlement_key_id;
  size_t entitlement_key_id_length;
  const uint8_t* content_key_id;
  size_t content_key_id_length;
  const uint8_t* content_key_data_iv;
  const uint8_t* content_key_data;
  size_t content_key_data_length;
} 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_LOADED.
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] num_keys: 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_LOADED

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

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 {
```

```
    const uint8_t* key_id;
    size_t key_id_length;
    const uint8_t* key_control_iv;
    const uint8_t* key_control;
} OEMCrypto_KeyRefreshObject;
```

Updates an existing set of keys for continuing decryption in the current session.

The relevant fields have been extracted from the Renewal Response protocol message, but the entire message and associated signature are provided so the message can be verified (using HMAC-SHA256 with the current mac_key[server]). If any verification step fails, an error is returned. Otherwise, the key table in trusted memory is updated using the key_control block. When updating an entry in the table, only the duration, nonce, and nonce_enabled fields are used. All other key control bits are not modified.

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

This session's elapsed time clock is reset to 0 when this function is called. The elapsed time clock is used in OEMCrypto_DecryptCENC() and the other Decryption API functions to determine if the key has expired.

This function does not add keys to the key table. It is only used to update a key control block license duration. This function is used to update the duration of a key, only. It is not used to update key control bits.

If the KeyRefreshObject's key_control_iv is null, 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 is null, then this refresh object should be used to update the duration of all keys for the current session. In this case, key_control_iv will also be null and the control block will not be encrypted.

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 updated.

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 updated.

If the key_id is not null, and no matching entry is found in the key table, then return OEMCrypto_KEY_NOT_LOADED.

Aside from the key's duration, no other values in the key control block should be updated by this function.

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 pointer in each KeyObject points to a location in the message, or is null. I.e. `(message <= p && p+p_length <= message+message_length)` for `p` in each of `key_id`,`key_control_iv`, `key_control`. 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_KEY_NOT_LOADED.

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_KEY_NOT_LOADED

Buffer Sizes

OEMCrypto shall support message sizes of at least 8 KiB.

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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

OEMCrypto_QueryKeyControl

```
OEMCryptoResult
OEMCrypto_QueryKeyControl(const 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] 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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

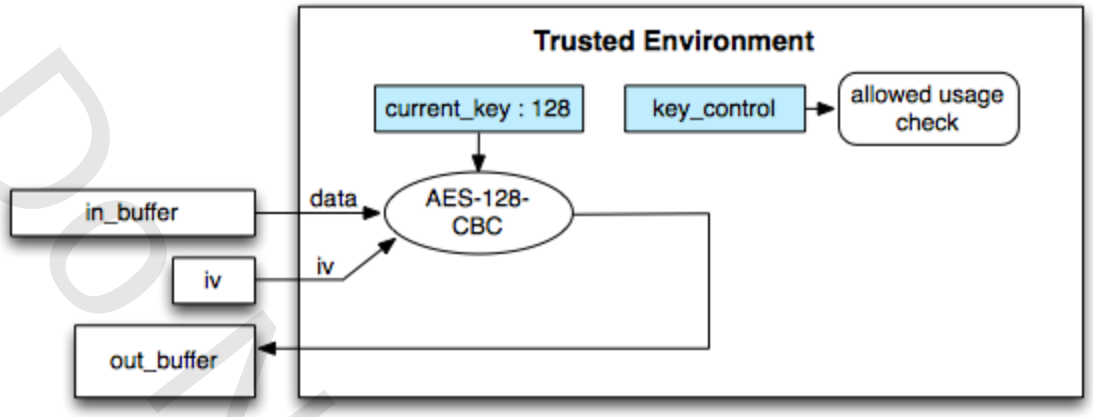
This method is new in API version 10.

Decryption API

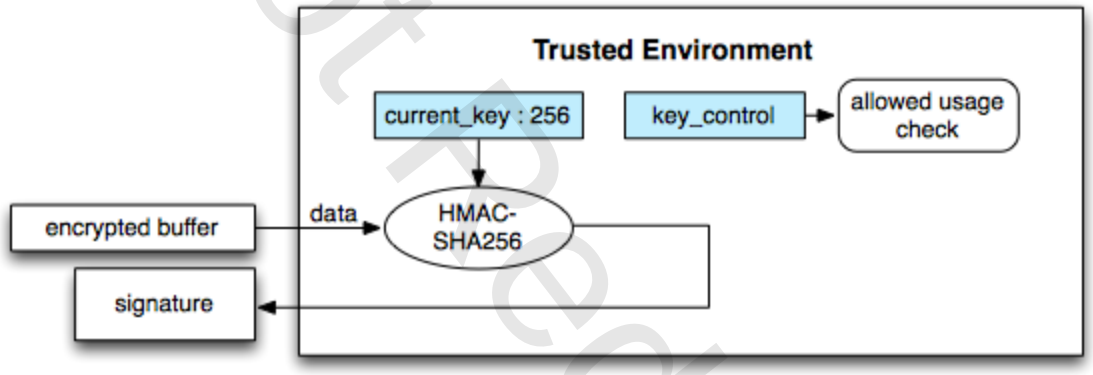
Devices that implement the Key Ladder API must also support a secure decode or secure decode and rendering implementation. This can be done by either decrypting into buffers secured by hardware protections and providing these secured buffers to the decoder/renderer or by implementing decrypt operations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

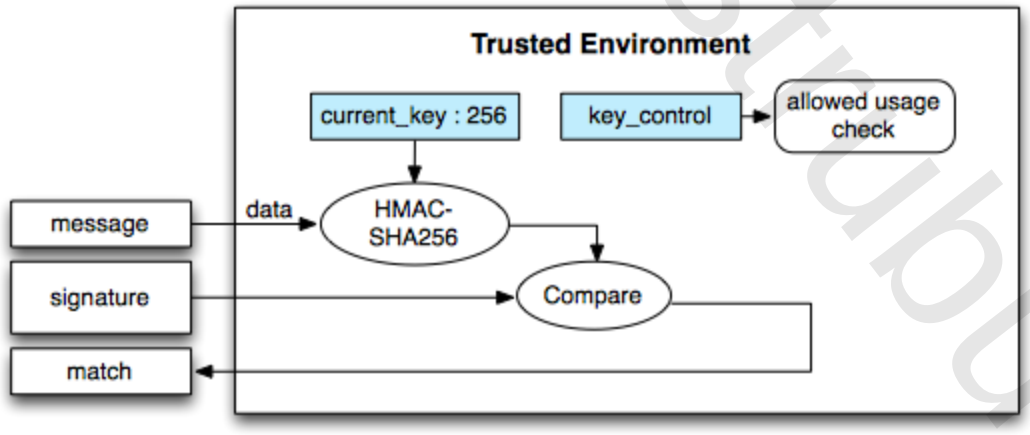
Generic Modular DRM allows an application to encrypt, decrypt, sign and verify arbitrary user data using a content key. This content key is securely delivered from the server to the client device using the same factory installed root of trust as a media content keys.



OEMCrypto_Generic_Decrypt(), OEMCrypto_Generic_Encrypt()



OEMCrypto_Generic_Sign()



OEMCrypto_Generic_Verify()

The following list shows the APIs required for decryption:

OEMCrypto_SelectKey

```
OEMCryptoResult OEMCrypto_SelectKey(const 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() or OEMCrypto_RefreshKeys().

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 and timers 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_KEY_NOT_LOADED.
2. If the current key's control block has a nonzero Duration field, then the API shall verify that the duration is greater than the session's elapsed time clock before the key is used.

OEMCrypto may return OEMCrypto_ERROR_KEY_EXPIRED from OEMCrypto_SelectKey, or SelectKey may return success from select key and the decrypt or generic crypto call will return OEMCrypto_ERROR_KEY_EXPIRED.

3. 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.
4. 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.
5. 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 key'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_KEY_NOT_LOADED

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 8.

OEMCrypto_DecryptCENC

```
OEMCryptoResult
OEMCrypto_DecryptCENC(OEMCrypto_SESSION session,
                      const uint8_t *data_addr,
                      size_t data_length,
                      bool is_encrypted,
                      const uint8_t *iv,
                      size_t block_offset, // used for CTR mode only.
                      OEMCrypto_DestBufferDesc* out_buffer,
                      const OEMCrypto_CENCEncryptPatternDesc* pattern,
                      unit8_t subsample_flags);
typedef enum OEMCryptoBufferType {
    OEMCrypto_BufferType_Clear,
    OEMCrypto_BufferType_Secure,
    OEMCrypto_BufferType_Direct
} OEMCrytoBufferType;

typedef struct {
    OEMCryptoBufferType type;
    union {
        struct {                  // type == OEMCrypto_BufferType_Clear
            uint8_t* address;
            size_t max_length;
        } clear;
        struct {                  // type == OEMCrypto_BufferType_Secure
            void* handle;
            size_t max_length;
            size_t offset;
        } secure;
        struct {                  // type == OEMCrypto_BufferType_Direct
            bool is_video;
        } direct;
    } buffer;
} OEMCrypto_DestBufferDesc;

#define OEMCrypto_FirstSubsample 1
#define OEMCrypto_LastSubsample 2

typedef struct {
```

```
    size_t encrypt; // number of 16 byte blocks to decrypt.
    size_t skip;   // number of 16 byte blocks to leave in clear.
    size_t offset;  // deprecated.
} OEMCrypto_CENCEncryptPatternDesc;
```

Decrypts or copies the payload in the buffer referenced by the *data_addr parameter into the buffer referenced by the out_buffer parameter, using the session context indicated by the session parameter. Decryption mode is AES-128-CTR or AES-128-CBC depending on the value of cipher_mode passed in to OEMCrypto_SelectKey. If is_encrypted is true, the content key associated with the session is latched in the active hardware key ladder and is used for the decryption operation. If is_encrypted is false, the data is simply copied.

After decryption, the data_length bytes are copied to the location described by out_buffer. This could be one of

1. The structure out_buffer 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 out_buffer contains a handle to a secure buffer.
3. The structure out_buffer indicates that the data should be sent directly to the decoder and renderer.

NOTES:

For CTR mode, IV points to the counter value to be used for the initial encrypted block of the input buffer. The IV length is the AES block size. For subsequent encrypted AES blocks the IV is calculated 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 CBC mode, IV points to the initial vector for cipher block chaining. Within each subsample, OEMCrypto is responsible for updating the IV as prescribed by CBC mode. The calling layer above is responsible for updating the IV from one subsample to the next if needed.

This method may be called several times before the decrypted data is used. For this reason, the parameter subsample_flags may be used to optimize decryption. 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 decrypted data will not be used until after OEMCrypto_LastSubsample has been set. If an implementation decrypts data immediately, it may ignore subsample_flags.

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

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.

The decryption mode, either OEMCrypto_CipherMode_CTR or OEMCrypto_CipherMode_CBC, was 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 can be found in the document **Draft International Standard ISO/IEC DIS 23001-7**. Search for the codes “cenc”, “cbc1”, “cens” or “cbcs”.

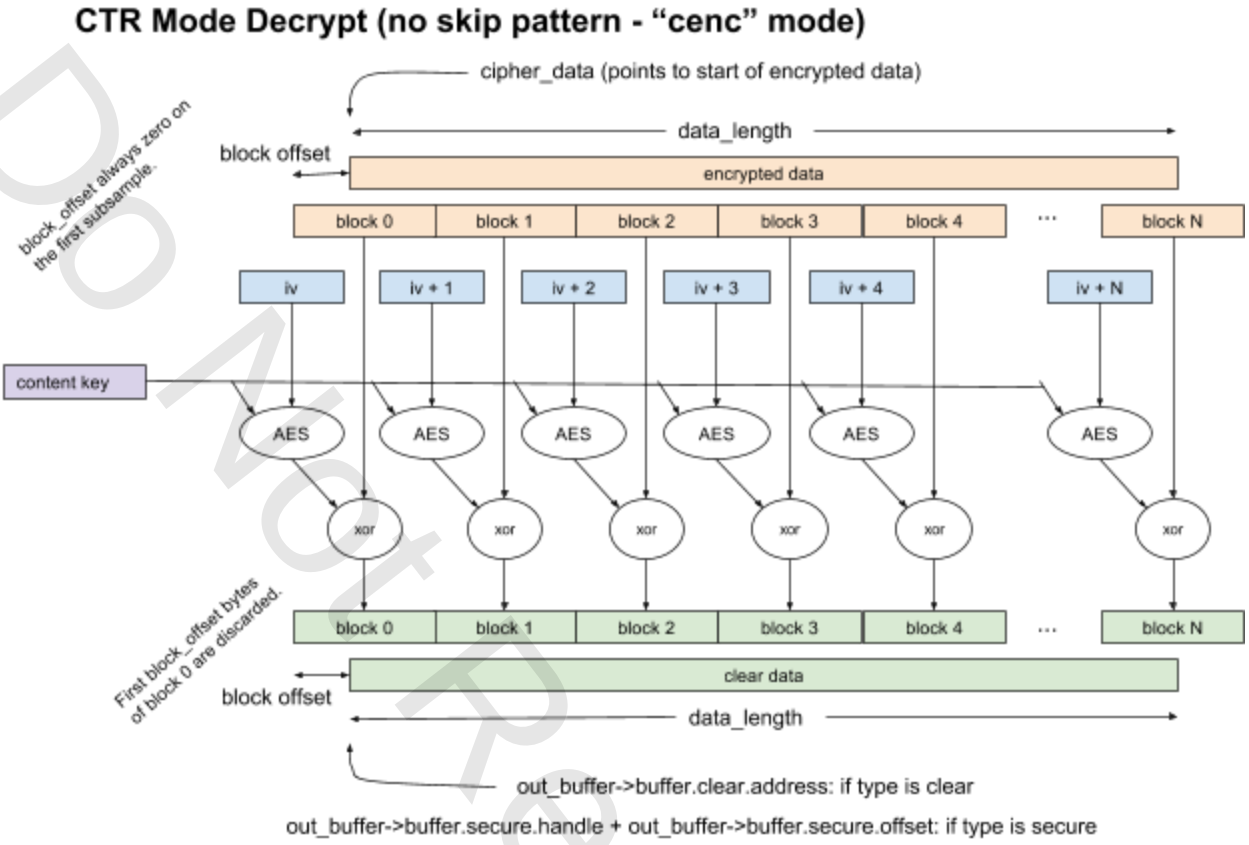
The most common mode is “cenc”, which is OEMCrypto_CipherMode_CTR without a pattern. The entire subsample is either encrypted or clear, depending on the flag is_encrypted. In the structure pattern, both encrypt and skip will be 0. This is the only mode that allows for a nonzero block_offset.

A less common mode is “cens”, which is OEMCrypto_CipherMode_CTR with an encryption pattern. For this mode, OEMCrypto may assume that an encrypted subsample will have a length that is a multiple of 16, the AES block length.

The mode “cbc1” is OEMCrypto_CipherMode_CBC without a pattern. In the structure pattern, both encrypt and skip will be 0. If an encrypted subsample has a length that is not a multiple of 16, the final partial block will be in the clear.

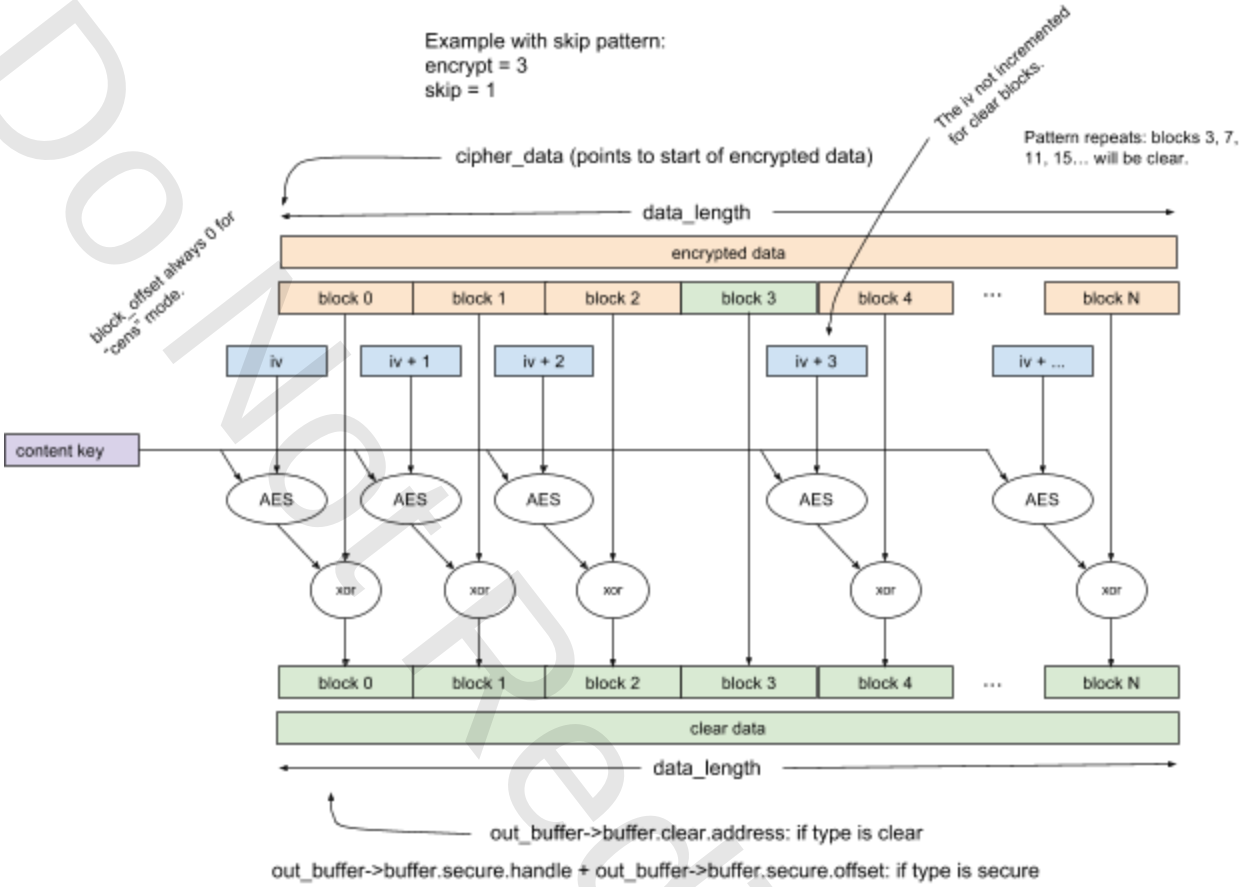
The mode “cbcs” is OEMCrypto_CipherMode_CBC with an encryption pattern. This mode allows devices to decrypt HLS content. If an encrypted subsample has a length that is not a multiple of 16, the final partial block will be in the clear.

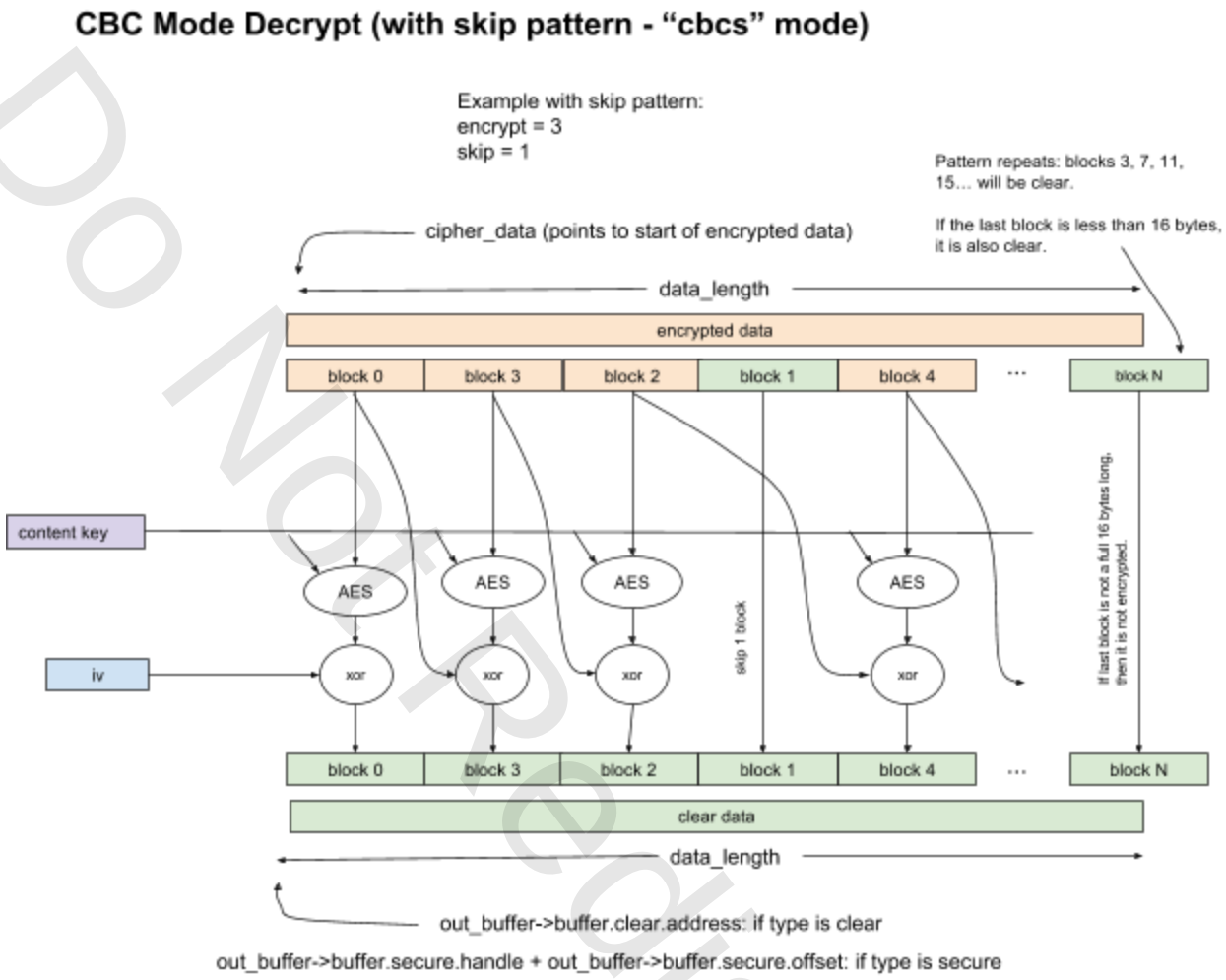
A sample may be broken up into a mix of clear and encrypted subsamples. In order to support the VP9 standard, the breakup of a subsample into clear and encrypted subsamples is not always in pairs.



If OEMCrypto assembles all of the subsamples into a single buffer and then decrypts, it can assume that the block offset is 0.

CTR Mode Decrypt (with skip pattern - “cens” mode)





Verification

The following checks should be performed if is_encrypted is true. If any check fails, an error is returned, and no decryption is performed.

1. If the current key’s control block has a nonzero Duration field, then the API shall verify that the duration is greater than the session’s elapsed time clock. If not, return OEMCrypto_ERROR_KEY_EXPIRED.
2. 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.
3. 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.
4. 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.
5. 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, then return OEMCrypto_ERROR_INSUFFICIENT_HDCP.

6. 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.

If the flag is_encrypted is false, then no verification is performed. This call shall copy clear data even when there are no keys loaded, or there is no selected key.

Parameters

[in] session: crypto session identifier.

[in] data_addr: An unaligned pointer to this segment of the stream.

[in] data_length: The length of this segment of the stream, in bytes.

[in] is_encrypted: True if the buffer described by data_addr, data_length is encrypted. If is_encrypted is false, only the data_addr and data_length parameters are used. The iv and offset arguments are ignored.

[in] iv: The initial value block to be used for content decryption.

This is discussed further below.

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

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

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

[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_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

Buffer Sizes

OEMCrypto shall support subsample sizes (i.e. data_length) of at least 100 KiB.

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.

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 13. This method changed its name in API version 11.

OEMCrypto_CopyBuffer

```
OEMCryptoResult
OEMCrypto_CopyBuffer(const uint8_t *data_addr,
                     size_t data_length,
                     OEMCrypto_DestBufferDesc* out_buffer,
                     unit8_t subsample_flags);
```

Copies the payload in the buffer referenced by the *data_addr 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_addr or out_buffer is null, return OEMCrypto_ERROR_INVALID_CONTEXT.

Parameters

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

[in] data_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

Buffer Sizes

OEMCrypto shall support subsample sizes (i.e. data_length) up to 100 KiB.

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.

Threading

This function may be called simultaneously with any other functions.

Version

This method is changed in API version 12.

OEMCrypto_Generic_Encrypt

```
OEMCryptoResult OEMCrypto_Generic_Encrypt(
                    OEMCrypto_SESSION session,
                    const uint8_t* in_buffer,
                    size_t buffer_length,
                    const uint8_t* iv,
                     OEMCrypto_Algorithm algorithm,
                    uint8_t* 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 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 encrypted.

1. The control bit for the current key shall have the Allow_Encrypt set. If not, return OEMCrypto_ERROR_UNKNOWN_FAILURE.
2. If the current key’s control block has a nonzero Duration field, then the API shall verify that the duration is greater than the session’s elapsed time clock. If not, return OEMCrypto_ERROR_KEY_EXPIRED.
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] buffer_length: length of the buffer, in bytes. The algorithm may restrict 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

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 function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

OEMCrypto_Generic_Decrypt

```
OEMCryptoResult OEMCrypto_Generic_Decrypt(
                      OEMCrypto_SESSION session,
                      const uint8_t* in_buffer,
                      size_t buffer_length,
                      const uint8_t* iv,
                       OEMCrypto_Algorithm algorithm,
                      uint8_t* 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 the current key's control block has a nonzero Duration field, then the API shall verify that the duration is greater than the session's elapsed time clock. If not, return OEMCrypto_ERROR_KEY_EXPIRED.
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] buffer_length: length of the buffer, in bytes. The algorithm may restrict 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

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 function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

OEMCrypto_Generic_Sign

```
OEMCryptoResult OEMCrypto_Generic_Sign(
                    OEMCrypto_SESSION session,
                    const uint8_t* in_buffer,
                    size_t buffer_length,
                     OEMCrypto_Algorithm algorithm,
                    uint8_t* 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 the current key’s control block has a nonzero Duration field, then the API shall verify that the duration is greater than the session’s elapsed time clock. If not, return OEMCrypto_ERROR_KEY_EXPIRED.
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] 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

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 function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 14.

OEMCrypto_Generic_Verify

```
OEMCryptoResult OEMCrypto_Generic_Verify(
                      OEMCrypto_SESSION session,
                      const uint8_t* in_buffer,
                      size_t buffer_length,
                       OEMCrypto_Algorithm algorithm,
                      uint8_t* 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 the current key's control block has a nonzero Duration field, then the API shall verify that the duration is greater than the session's elapsed time clock. If not, return OEMCrypto_ERROR_KEY_EXPIRED.
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] 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

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 function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 14.

Keybox Access 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 Provisioning above. The functions in this section are for devices that are provisioned with a keybox, i.e. Provisioning 2.0.

Widevine keyboxes are used to establish a root of trust to secure content on a device that uses

Provisioning 2.0. Keybox 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_WrapKeybox and OEMCrypto_InstallKeybox 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. The test keybox does not have to be factory provisioned -- it can be hard coded into the oemcryto library because it is identical for all devices.

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_WrapKeybox - optional - only used by factory setup tools.
- OEMCrypto_InstallKeybox - optional - only used on some platforms.
- OEMCrypto_GetProvisioningMethod - required for keybox or oem cert. (provisioning 2.0 and 3.0)
- OEMCrypto_LoadTestKeybox- required if keybox provisioned (provisioning 2.0)
- OEMCrypto_IsKeyboxValid- required if keybox provisioned (provisioning 2.0)
- OEMCrypto_GetDeviceID- required if keybox provisioned (provisioning 2.0)
- OEMCrypto_GetKeyData- required if keybox provisioned (provisioning 2.0)

OEMCrypto_WrapKeybox

```
OEMCryptoResult OEMCrypto_WrapKeybox(
                        uint8_t *keybox,
                        uint32_t keyboxLength,
                        uint8_t *wrappedKeybox,
                        uint32_t *wrappedKeyBoxLength,
                        uint8_t *transportKey
                        uint32_t transportKeyLength);
```

For provisioning 2.0, during manufacturing, the keybox 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 WrapKeybox/InstallKeybox approach. When the Widevine DRM plugin initializes, it will look for a wrapped keybox in the file /factory/wv.keys and install it into the security processor by calling OEMCrypto_InstallKeybox().

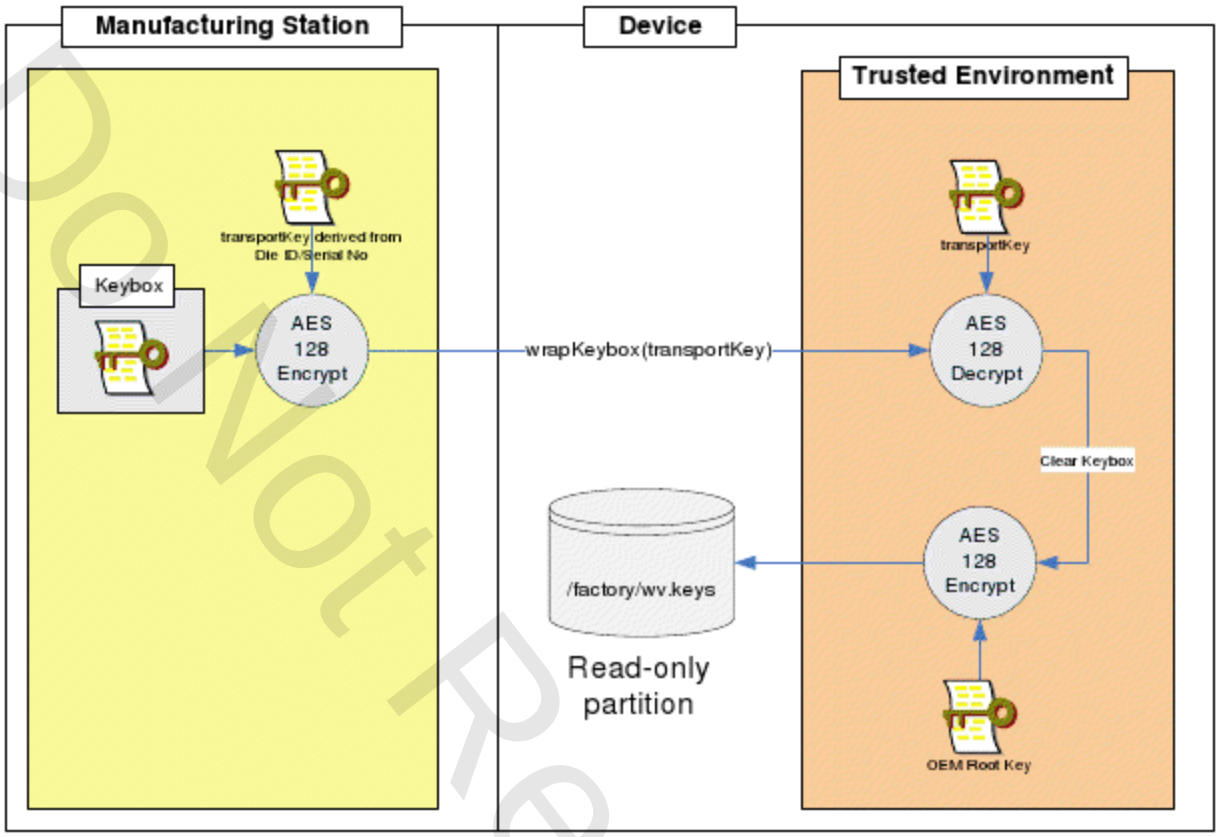


Figure 10. OEMCrypto_WrapKeybox Operation

OEMCrypto_WrapKeybox() is used to generate an OEM-encrypted keybox that may be passed to OEMCrypto_InstallKeybox() for provisioning. The keybox 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 keybox provisioning method involves saving the keybox to the file system.**

Parameters

[in] keybox - pointer to Keybox data to encrypt. May be NULL on the first call to test size of wrapped keybox. The keybox may either be clear or previously encrypted.

[in] keyboxLength - length the keybox data in bytes

[out] wrappedKeybox – Pointer to wrapped keybox

[out] wrappedKeyboxLength – Pointer to the length of the wrapped keybox in bytes

[in] transportKey – Optional. AES transport key. If provided, the keybox parameter was

previously encrypted with this key. The keybox will be decrypted with the transport key using AES-CBC and a null IV.

[in] transportKeyLength – Optional. Number of bytes in the transportKey, 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

Threading

This function is not called simultaneously with any other functions

Version

This method is supported in all API versions.

OEMCrypto_InstallKeybox

```
OEMCryptoResult OEMCrypto_InstallKeybox(
         uint8_t *keybox, uint32_t keyboxLength);
```

Decrypts a wrapped keybox and installs it in the security processor. The keybox 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 keybox installed. It looks for a wrapped keybox in the file /factory/wv.keys and if it is present, will read the file and call OEMCrypto_InstallKeybox() with the contents of the file. **This function is only needed if the keybox provisioning method involves saving the keybox to the file system.**

Parameters

[in] keybox - pointer to encrypted Keybox data as input

[in] keyboxLength - length of the keybox data in bytes

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_BAD_MAGIC

OEMCrypto_ERROR_BAD_CRC

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

OEMCrypto_ERROR_NOT_IMPLEMENTED

Threading

This function is not called simultaneously with any other functions.

Version

This method is supported in all API versions.

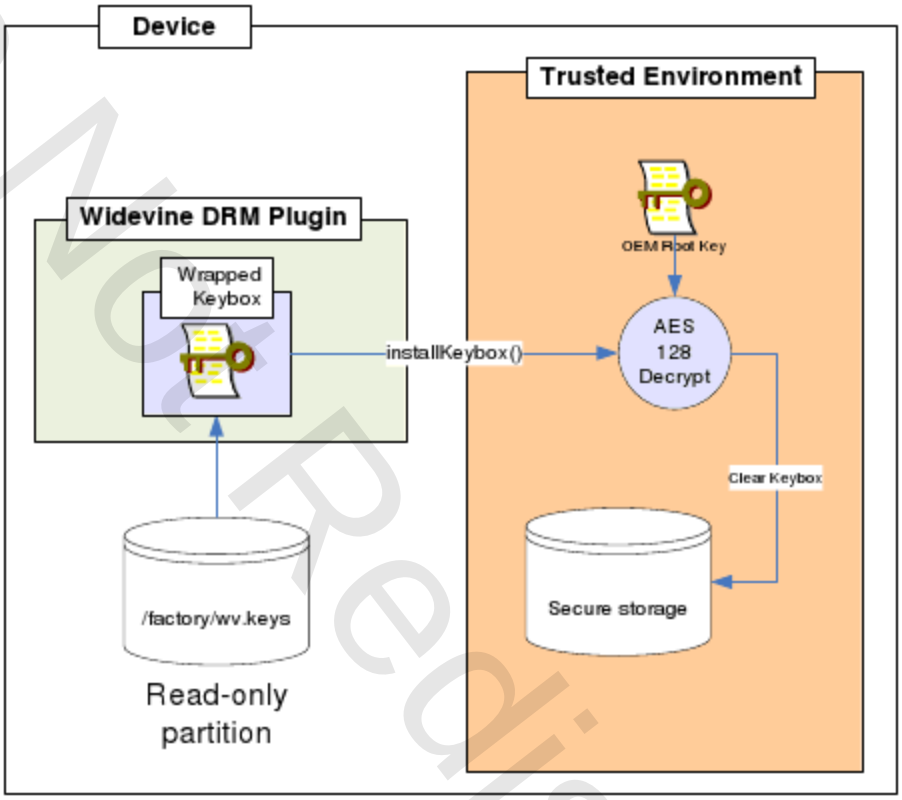


Figure 11 - Install keybox Operation

OEMCrypto_GetProvisioningMethod

```
OEMCrypto_ProvisioningMethod OEMCrypto_GetProvisioningMethod();

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 function may be called simultaneously with any session functions.

Version

This method is new API version 12.

OEMCrypto_LoadTestKeybox

```
OEMCryptoResult OEMCrypto_LoadTestKeybox(const uint8_t *buffer, size_t 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] 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.

Threading

This function is not called simultaneously with any other functions. It will be called just after OEMCrypto_Initialize().

Version

This method changed in API version 14.

OEMCrypto_IsKeyboxValid

```
OEMCryptoResult OEMCrypto_IsKeyboxValid();
```

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.

Parameters

none

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_BAD_MAGIC

OEMCrypto_ERROR_BAD_CRC

OEMCrypto_ERROR_NOT_IMPLEMENTED - this function is for Provisioning 2.0 only.

Threading

This function may be called simultaneously with any session functions.

Version

This method is supported in all API versions.

OEMCrypto_GetDeviceID

```
OEMCryptoResult OEMCrypto_GetDeviceID(
         uint8_t* deviceID,
         uint32_t *idLength);
```

Retrieve DeviceID from the Keybox.

Parameters

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

[in/out] idLength – 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_NOT_IMPLEMENTED - this function is for Provisioning 2.0 only.

Threading

This function may be called simultaneously with any session functions.

Version

This method is supported in all API versions.

OEMCrypto_GetKeyData

```
OEMCryptoResult OEMCrypto_GetKeyData(
         uint8_t* keyData, uint32_t *keyDataLength);
```

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.

Threading

This function may be called simultaneously with any session functions.

Version

This method is supported in all API versions.

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_GetProvisioningMethod - required for keybox or oem cert. (defined above)
OEMCrypto_GetOEMPublicCertificate- required if oem cert provisioned (provisioning 3.0)

OEMCrypto_GetOEMPublicCertificate

```
OEMCryptoResult OEMCrypto_GetOEMPublicCertificate(
                                OEMCrypto_SESSION session,
                                uint8_t *public_cert,
                                size_t *public_cert_length)
```

This function should place the OEM public certificate in the buffer public_cert. After a call to this function, all methods using an RSA key should use the OEM certificate's private RSA key. 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

- `session` - (in) this function affects the specified session only.
- `public_cert` (out) the buffer where the public certificate is stored.
- `public_cert_length` - (in/out) 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

Threading

This function may be called simultaneously with any session functions.

Version

This method is new API version 12.

Validation and Feature Support API

Widevine keyboxes establish a root of trust to secure content on a device.

The keybox access API provides an interface for a security processor or general CPU to access the Widevine Keybox, depending on the security level.

In a Level 1 or Level 2 implementation, only the security processor may access the keys in the keybox. The following list shows the APIs required for keybox validation:

OEMCrypto_GetRandom

```
OEMCryptoResult OEMCrypto_GetRandom(
         uint8_t* randomData, uint32_t dataLength);
```

Returns a buffer filled with hardware-generated random bytes, if supported by the hardware.

Parameters

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

[in] dataLength - 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

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 function may be called simultaneously with any session functions.

Version

This method is supported in all API versions.

OEMCrypto_APIVersion

```
uint32_t OEMCrypto_APIVersion();
```

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

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 14. Any OEM that returns this version number guarantees it passes all unit tests associated this version.

Parameters

none

Returns

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

Threading

This function may be called simultaneously with any other functions.

Version

This method changed in API version 11.

OEMCrypto_Security_Patch_Level

```
uint8_t OEMCrypto_Security_Patch_Level();
```

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 function may be called simultaneously with any other functions.

Version
This method was introduced in API version 11.

OEMCrypto_SecurityLevel

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

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 function may be called simultaneously with any other functions.

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_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.

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

Threading

This function may be called simultaneously with any other functions.

Version

This method changed in API version 10.

OEMCrypto_SupportsUsageTable

```
bool OEMCrypto_SuportsUsageTable();
```

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 function may be called simultaneously with any other functions.

Version

This method changed in API version 9.

OEMCrypto_IsAntiRollbackHwPresent

```
bool OEMCrypto_IsAntiRollbackHwPresent();
```

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 function may be called simultaneously with any other functions.

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

Threading

This function may be called simultaneously with any other functions.

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

Returns

OEMCrypto_SUCCESS

OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

This function may be called simultaneously with any other functions.

Version

This method changed in API version 12.

OEMCrypto_SupportedCertificates

```
uint32_t OEMCrypto_SupportedCertificates();
```

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 certificate are used with OEMCrypto_GenerateRSASignature with padding type set to 0x2, PKCS1 with block type 1 padding.

Threading

This function may be called simultaneously with any other functions.

Version

This method changed in API version 13.

OEMCrypto_IsSRMUpdateSupported

```
bool OEMCrypto_IsSRMUpdateSupported();
```

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 function may be called simultaneously with any other functions.

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.

Threading

This function may be called simultaneously with any other functions.

Version

This method changed in API version 13.

OEMCrypto_GetAnalogOutputFlags

```
uint32_t OEMCrypto_GetAnalogOutputFlags();
```

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 function may be called simultaneously with any other functions.

Version

This method is new in API version 14.

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.

The following list shows the APIs required for RSA provisioning and licensing:

OEMCrypto_RewrapDeviceRSAKey30 - required if oem certificate provisioned

OEMCrypto_RewrapDeviceRSAKey - required if keybox provisioned

OEMCrypto_LoadDeviceRSAKey

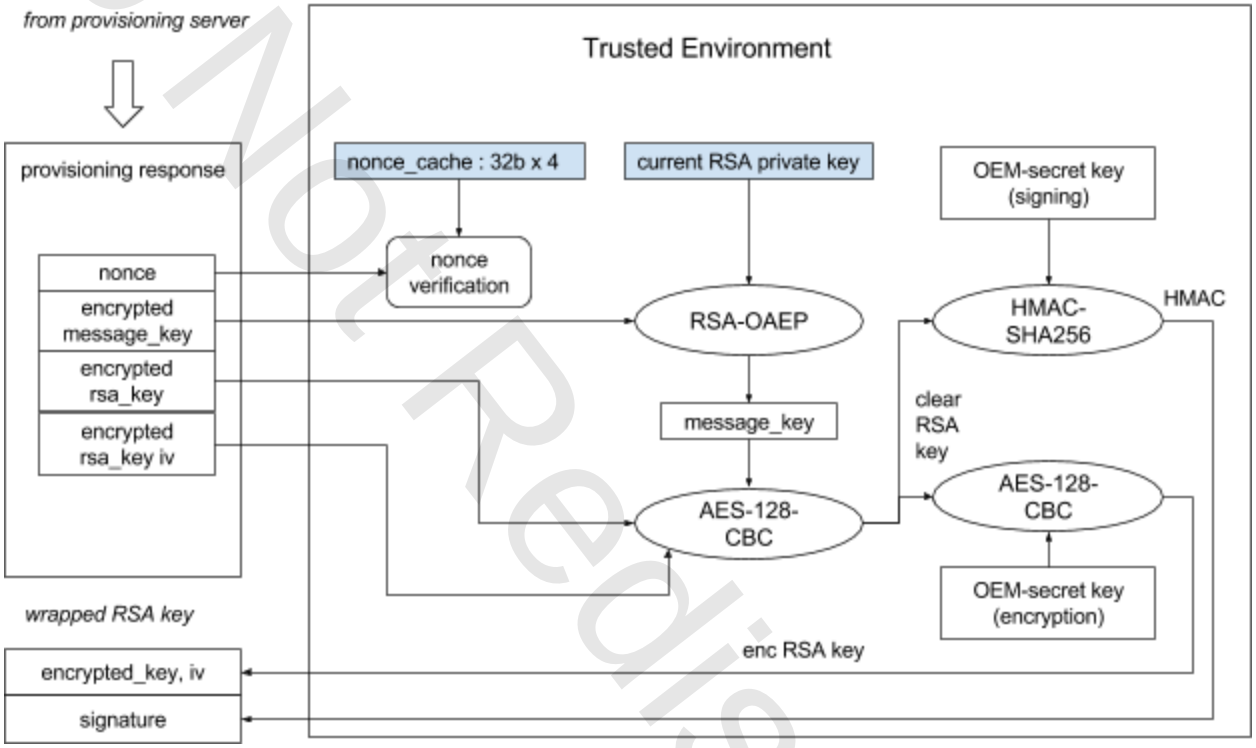
OEMCrypto_LoadTestRSAKey

OEMCrypto_GenerateRSASignature

OEMCrypto_RewrapDeviceRSAKey30

```
OEMCryptoResult OEMCrypto_RewrapDeviceRSAKey30(
                                OEMCrypto_SESSION session,
                                const uint32_t* unaligned_nonce,
                                const uint8_t* encrypted_message_key,
                                size_t encrypted_message_key_length,
                                const uint8_t* enc_rsa_key,
                                size_t enc_rsa_key_length,
                                const uint8_t* enc_rsa_key_iv,
                                uint8_t* wrapped_rsa_key,
                                size_t* wrapped_rsa_key_length)
```

This function is similar to RewrapDeviceRSAKey, except it uses the private key from an OEM certificate to decrypt the message key instead of keys derived from a keybox. Verifies an RSA provisioning response is valid and corresponds to the previous provisioning request by checking the nonce. The RSA private key is decrypted and stored in secure memory. The RSA key is then re-encrypted and signed 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.



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. Verify that `in_wrapped_rsa_key_length` is large enough to hold the rewrapped key, returning `OEMCrypto_ERROR_SHORT_BUFFER` otherwise.
2. Verify that the nonce matches one generated by a previous call to `OEMCrypto_GenerateNonce()`. The matching nonce shall be removed from the nonce table. If there is no matching nonce, return `OEMCRYPTO_ERROR_INVALID_NONCE`. Notice that the nonce may not point to a word aligned memory location.
3. Decrypt encrypted_message_key with the OEM certificate's private RSA key using RSA-OAEP into the buffer message_key. This message key is a 128 bit AES key used only in step 4. This message_key should be kept in secure memory and protected from the user.
4. Decrypt enc_rsa_key into the buffer rsa_key using the message_key, which was found in step 3. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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] nonce: A pointer to the nonce provided in the provisioning response. (unaligned uint32_t)
[in] encrypted_message_key : message_key encrypted by private key from OEM cert.
[in] encrypted_message_key_length : length of encrypted_message_key in bytes.
[in] enc_rsa_key: Encrypted device private RSA key received from the provisioning server. Format is PKCS#8, binary DER encoded, and encrypted with message_key, using

AES-128-CBC with PKCS#5 padding.
[in] enc_rsa_key_length: length of the encrypted RSA key, in bytes.
[in] enc_rsa_key_iv: IV for decrypting RSA key. Size is 128 bits.
[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

Buffer Sizes

OEMCrypto shall support message sizes of at least 8 KiB.

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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

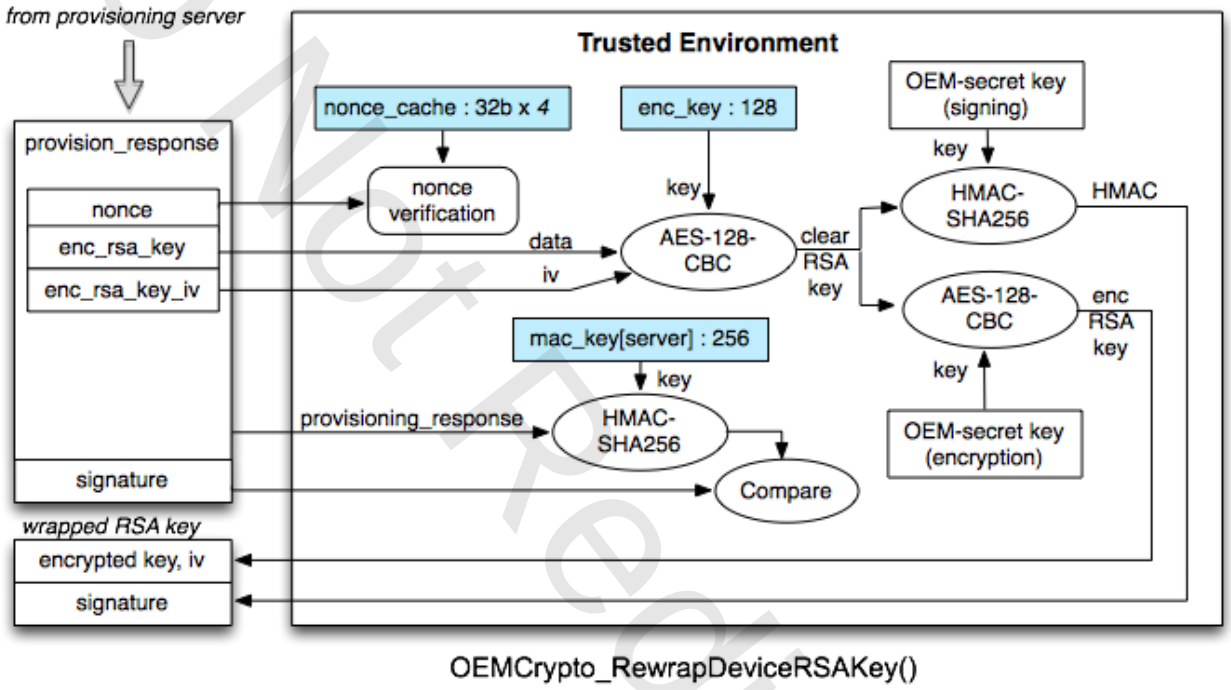
Version

This method changed in API version 12.

OEMCrypto_RewrapDeviceRSAKey

```
OEMCryptoResult OEMCrypto_RewrapDeviceRSAKey(
                      OEMCrypto_SESSION session,
                      const uint8_t* message,
                      size_t message_length,
                      const uint8_t* signature,
                      size_t signature_length,
                      const uint32_t* unaligned_nonce,
                      const uint8_t* enc_rsa_key,
                      size_t enc_rsa_key_length,
                      const uint8_t* enc_rsa_key_iv,
                      uint8_t* wrapped_rsa_key,
                      size_t *wrapped_rsa_key_length);
```

This function is similar to RewrapDeviceRSAKey30, except it uses session keys derived from the keybox instead of the OEM certificate. Verifies an RSA provisioning response is valid and corresponds to the previous provisioning request by checking the nonce. The RSA private key is decrypted and stored in secure memory. The RSA key is then re-encrypted and signed 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.



OEMCrypto_RewrapDeviceRSAKey()

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 that the nonce matches one generated by a previous call to `OEMCrypto_GenerateNonce()`. The matching nonce shall be removed from the nonce table. If there is no matching nonce, return `OEMCRYPTO_ERROR_INVALID_NONCE`.
4. Verify the message signature, using the derived signing key (mac_key[server]) from a previous call to OEMCrypto_GenerateDerivedKeys.
5. Decrypt enc_rsa_key in the buffer rsa_key using the derived encryption key (enc_key) from a previous call to OEMCrypto_GenerateDerivedKeys. 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 message to be verified.
[in] message_length: length of the message, in bytes.
[in] signature: pointer to memory containing the HMAC-SHA256 signature for `message`, received from the provisioning server.
[in] signature_length: length of the signature, in bytes.
[in] nonce: A pointer to the nonce provided in the provisioning response.

[in] enc_rsa_key: Encrypted device private RSA key received from the provisioning server. Format is PKCS#8, binary DER encoded, and encrypted with the derived encryption key, using AES-128-CBC with PKCS#5 padding.
[in] enc_rsa_key_length: length of the encrypted RSA key, in bytes.
[in] enc_rsa_key_iv: IV for decrypting RSA key. Size is 128 bits.
[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

Buffer Sizes

OEMCrypto shall support message sizes of at least 8 KiB.

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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

OEMCrypto_LoadDeviceRSAKey

```
OEMCryptoResult OEMCrypto_LoadDeviceRSAKey(
                      OEMCrypto_SESSION session,
                      const uint8_t* wrapped_rsa_key,
                      size_t wrapped_rsa_key_length);
```

Loads a wrapped RSA private key to secure memory for use by this session in future calls to OEMCrypto_GenerateRSASignature. The wrapped RSA key will be the one verified and wrapped by OEMCrypto_RewrapDeviceRSAKey. The RSA 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. If there was not bit field wrapped with the RSA key, the key will use a default value of 1 = RSASSA-PSS with SHA1.

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] 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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 9.

OEMCrypto_LoadTestRSAKey

Some platforms do not support keyboxes. On those platforms, there is a DRM certificate baked into the OEMCrypto library -- factory provisioned like a keybox is on standard devices. In order to debug and test those devices, they should be able to switch to the test DRM certificate.

```
OEMCryptoResult OEMCrypto_LoadTestRSAKey();
```

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

Threading

This function is not called simultaneously with any other functions. It will be called just after OEMCrypto_Initialize().

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 used to sign messages using the device private RSA key, specifically, it is used to sign the initial license request.
Refer to the Signing Messages Sent to a Server section above for more details.

If this function is called after OEMCrypto_LoadDeviceRSAKey for the same session, then this function should use the device RSA key that was loaded. If this function is called after a call to OEMCrypto_GetOEMPublicCertificate for the same session, then this function should use the RSA private key associated with the OEM certificate. The only padding scheme that is valid for the OEM certificate is 0x1 - RSASSA-PSS with SHA1. Any other padding scheme must

generate an error.

For devices that wish to be CAST receivers, there is a new RSA padding scheme. The padding_scheme parameter indicates which hashing and padding is to be applied to the message so as to generate the encoded message (the modulus-sized block to which the integer conversion and RSA decryption is applied). The following values are defined:
0x1 - RSASSA-PSS with SHA1.
0x2 - PKCS1 with block type 1 padding (only).

In the first case, a hash algorithm (SHA1) is first applied to the message, whose length is not otherwise restricted. In the 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 the second scheme and can return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Verification

The bitwise AND of the parameter padding_scheme and the RSA key's allowed_schemes is computed. If this value is 0, 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

Buffer Sizes

OEMCrypto shall support message sizes of at least 8 KiB.

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

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 12.

Usage Table API

The following list shows the APIs required for Usage Table maintenance and reporting:

- OEMCrypto_CreateUsageTableHeader
- OEMCrypto_LoadUsageTableHeader
- OEMCrypto_CreateNewUsageEntry
- OEMCrypto_LoadUsageEntry
- OEMCrypto_UpdateUsageEntry
- OEMCrypto_DeactivateUsageEntry
- OEMCrypto_ReportUsage
- OEMCrypto_MoveEntry
- OEMCrypto_ShrinkUsageTableHeader

The following list shows the APIs required for updating the Usage Table used by OEMCrypto before version 13 to the current version:

- OEMCrypto_CopyOldUsageEntry
- OEMCrypto_DeleteOldUsageTable

The following list shows the APIs required for testing updating the Usage Table used by OEMCrypto before version 13 to the current version:

- OEMCrypto_CreateOldUsageEntry

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.

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

Threading

This function will not be called simultaneously with any session functions.

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] buffert_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.

Threading
This function will not be called simultaneously with any session functions.

Version
This method changed in API version 13.

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.

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

Threading
This function may be called simultaneously with functions on other sessions, but not with other

functions on this session.

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 that 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.

If the entry is already loaded into another session, then this fails and returns OEMCrypto_ERROR_INVALID_SESSION.

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.

Threading

This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Version

This method changed in API version 13.

OEMCrypto_UpdateUsageEntry

```
OEMCryptoResult OEMCrypto_UpdateUsageEntry(OEMCrypto_SESSION session,
                    uint8_t* header_buffer,
                    size_t* header_buffer_length,
                    uint8_t* 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 will 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 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. As in OEMCrypto v12, 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 buffer_length is not large enough, they are set to the needed size, and 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] 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

Threading

This function may be called simultaneously with functions on other sessions, but not with other

functions on this session.

Version

This method changed in API version 13.

OEMCrypto_DeactivateUsageEntry

```
OEMCryptoResult OEMCrypto_DeactivateUsageEntry(OEMCrypto_SESSION session,
                                               uint8_t *pst,
                                               size_t pst_length);
```

This deactivates the usage entry associated with the current session. This means that the state 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 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.

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

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 function will not be called simultaneously with any session functions.

Version

This method changed in API version 13.

OEMCrypto_ReportUsage

```
OEMCryptoResult OEMCrypto_ReportUsage(OEMCrypto_SESSION session,
                                      const uint8_t *pst,
```

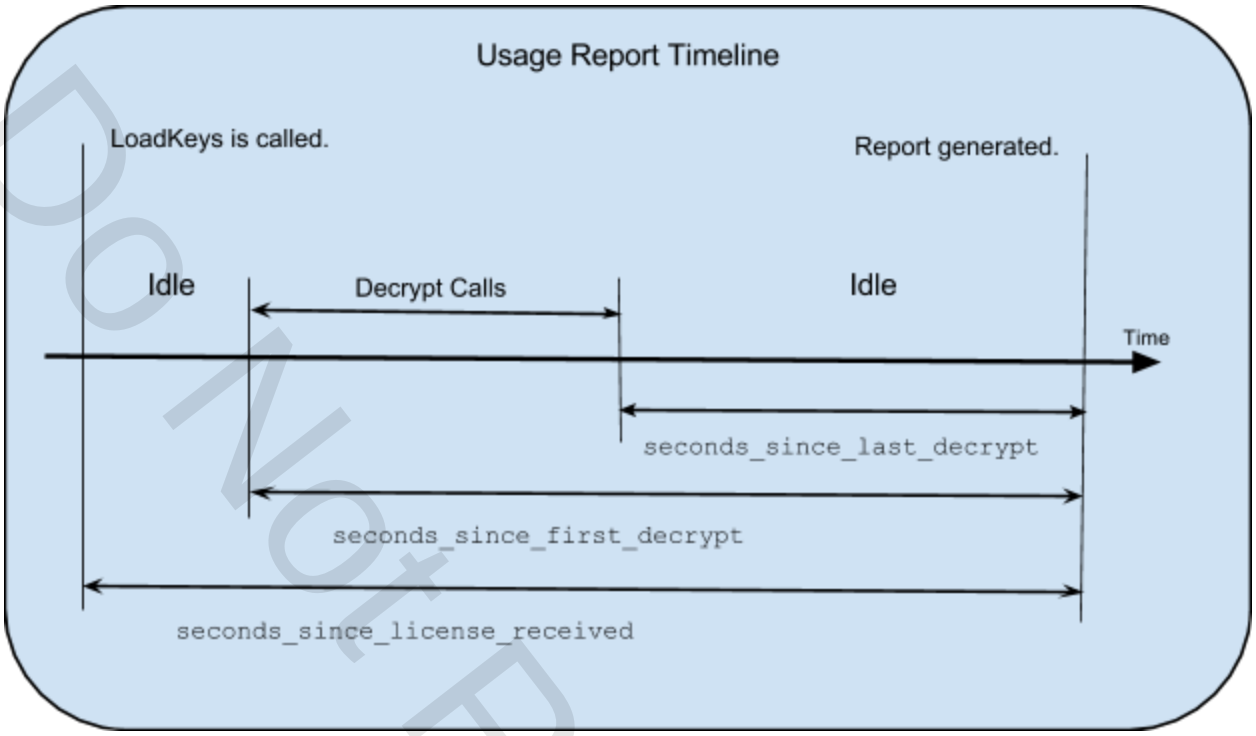
```
                                    size_t pst_length,
                                    uint8_t *buffer,
                                    size_t *buffer_length);

typedef struct OEMCrypto_PST_Report {
   uint8_t signature[20] -- HMAC SHA1 of the rest of the report.
   uint8_t padding[4];    // make int64's word aligned.
   int64_t seconds_since_license_received == now - time_of_license_received
   int64_t seconds_since_first_decrypt == now - time_of_first_decrypt
   int64_t seconds_since_last_decrypt == now - time_of_last_decrypt
   uint8_t (enum OEMCrypto_Usage_Entry_Status) status;  -- current status of pst
entry.
   uint8_t clock_security_level;
   uint8_t pst_length;
   uint8_t pst[0];
} __attribute__ ((packed)) OEMCrypto_PST_Report;
```

If the buffer_length is not sufficient to hold a report structure, set buffer_length and return OEMCrypto_ERROR_SHORT_BUFFER.

If the 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 is done in case the secure clock is not using UTC time, but is instead using something like seconds since clock installed.



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.
- 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.

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.

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 function will not be called simultaneously with any session functions.

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

Threading

This function will not be called simultaneously with any session functions.

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. If new_table_size is larger than the current size, then the header is not changed and the error 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_table_size.

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.

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

Threading

This function will not be called simultaneously with any session functions.

Version

This method is new in API version 13.

OEMCrypto_CopyOldUsageEntry

```
OEMCryptoResult OEMCrypto_CopyOldUsageEntry(OEMCrypto_SESSION session,
                                            const uint8_t*pst,
                                            size_t pst_length)
```

This function copies an entry from the old v12 table to the new table. The new entry will already have been loaded by CreateNewUsageEntry. If the device did not support pre-v13 usage tables, this may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

This is only needed for devices that are upgrading from a version of OEMCrypto before v13 to a recent version. Devices that have an existing usage table with customer's offline licenses will use this method to move entries from the old table to the new one.

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_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading
This function will not be called simultaneously with any session functions.

Version
This method is new in API version 13.

OEMCrypto_DeleteOldUsageTable

```
OEMCryptoResult OEMCrypto_DeleteOldUsageTable();
```

This function will delete the old usage table, if possible, freeing any nonvolatile secure memory. This may return OEMCrypto_ERROR_NOT_IMPLEMENTED if the device did not support pre-v13 usage tables.

This is only needed for devices that are upgrading from a version of OEMCrypto before v13 to a recent version. Devices that have an existing usage table with customer's offline licenses will use this method to move entries from the old table to the new one.

Parameters
none

Returns
OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NOT_IMPLEMENTED
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading
This function will not be called simultaneously with any session functions.

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
OEMCrypto_CreateOldUsageEntry

OEMCrypto_RemoveSRM

```
OEMCryptoResult OEMCrypto_RemoveSRM()
```

Delete the current SRM. Any valid SRM, regardless of 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 function will not be called simultaneously with any other functions.

Version
This method is new in API version 13.

OEMCrypto_CreateOldUsageEntry

```
OEMCryptoResult OEMCrypto_CreateOldUsageEntry(
             uint64_t time_since_license_received,
             uint64_t time_since_first_decrypt,
             uint64_t time_since_last_decrypt,
             OEMCrypto_Usage_Entry_Status status,
             uint8_t *server_mac_key,
             uint8_t *client_mac_key,
             const uint8_t* pst,
             size_t pst_length);
```

This forces the creation of an entry in the old usage table in order to test OEMCrypto_CopyOldUsageTable. OEMCrypto will create a new entry, set the status and compute the times at license receive, first decrypt and last decrypt. The mac keys will be copied to the entry. The mac keys are **not** encrypted, but will only correspond to a test license.

Devices that have do not support usage tables, or devices that are will not be field upgraded

from a version of OEMCrypto before v13 to a recent version may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Returns

OEMCrypto_ERROR_NOT_IMPLEMENTED

Version

This method is new in API version 13.

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	SelectKey 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 RewrapDeviceRSAKey or LoadDeviceRSAKey
36	OEMCrypto_ERROR_KEY_EXPIRED	The current key's duration 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	Shared license loaded without first loading master license.
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	Attempt to select or refresh key that is not in key table.
52	OEMCrypto_KEY_NOT_ENTITLED	Attempt to load entitled content key with no matching entitlement key

---

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
