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

Version 11

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

Table of Contents

Terms and Definitions

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

Device Key — 128-bit AES key assigned by Widevine and used to secure entitlements.

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.

Provision — Install a Keybox that has been uniquely constructed for a specific device.

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

Common Encryption (CENC) — standards based scheme for encryption and key management

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

OEMCrypto — the low level cryptographic library implemented by the OEM to provide key and content protection.

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.

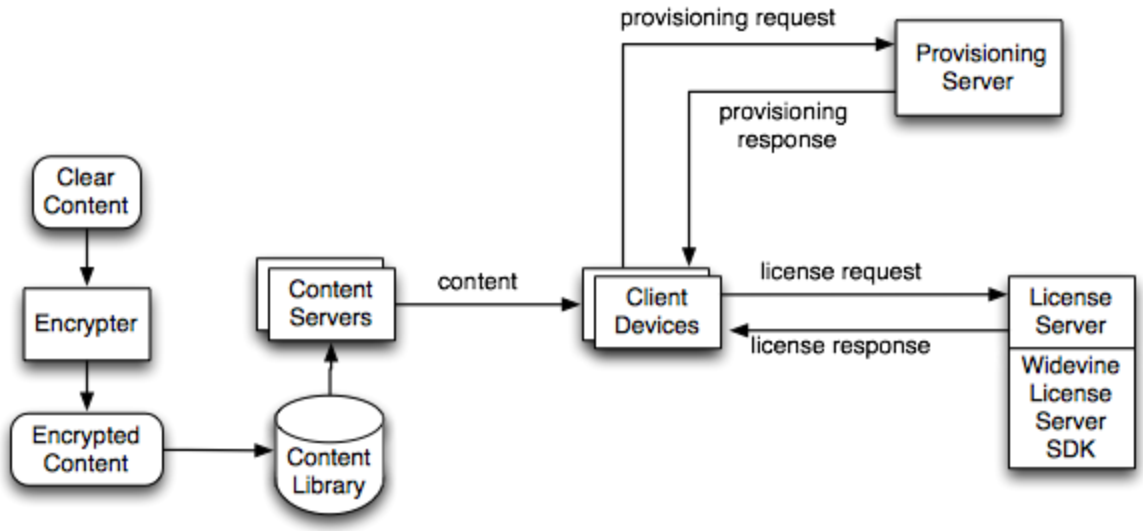
Purpose

This document describes the security APIs used in Widevine content protection for playing content 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.

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

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.



Licenses to view the content are obtained from a License Server. The security (signing and encryption) of the licenses is implemented by the License Server SDK, which is a library that is linked with the service provider's license server. A license is requested from the server using a license request (a.k.a challenge). The license response is delivered to the client.

A provisioning server may be required to distribute device-unique credentials to the devices. This process extends the chain of trust established during factory or field provisioning of the devices using the Widevine keybox by securely delivering an asymmetric device private key to the device over a secure channel.

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.

Device Provisioning

Provisioning Overview

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.

A device may also be provisioned with an RSA certificate. For most platforms, this certificate is installed in the field using the API described below in the section RSA Certificate Provisioning and License Requests. In rare cases, a platform might only have an RSA certificate baked in, and does not need to have a keybox. See the section below about Optional Features. Refer to any supplement to this guide for your platform

if you are not sure which features are required. For example, Level 1 Android devices are required to have a factory provisioned keybox.

Keyboxes may be installed on devices using Field provisioning or Factory provisioning. Field provisioning is only used for Level 3 implementations. Level 1 and 2 devices must be factory-provisioned and the keybox must be encrypted by an on-chip AES device unique secret key before being stored into non-erasable persistent memory.

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. 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.

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***

OEMCrypto APIs for Common Encryption

OEMCrypto is an interface to the trusted environment that implements the functions needed to protect and manage keys for the Widevine content protection system. 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 a key 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, and (4) a means to use the content keys to produce a decrypted stream for decoding and rendering.

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 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.

Session Context

One or more crypto sessions will be created to support A/V playback. Each session has context, or state, that must be maintained in secure memory. The required session state is summarized in the diagram below.

Most of the OEMCrypto calls require information to be retained in the session context. There may be several sessions, and each session has its own collection of keys. Each session has its own current content key and its own pair of message signing keys (mac_keys). Typically, a session has a video key and an audio key, but there may be more than two keys. There may be several sessions active at any moment. When an application wishes to switch from one resolution to another, it may create a new session with a different set of keys.

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 and the Generalized Modular DRM sections 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 RSA certificates, the functions in RSA Certificate Provisioning API are also session specific. Each session may have a different RSA key installed.

The functions in the Crypto Device Control API, Provisioning API, and Keybox Access and Validation API sections are not associated with any one session. There is only one widevine keybox on the device. These functions handle initialization of the device itself.

The figure below shows data that should be stored in the trusted environment. The widevine keybox is shared for all sessions. All of the other data in the figure is specific to a session.

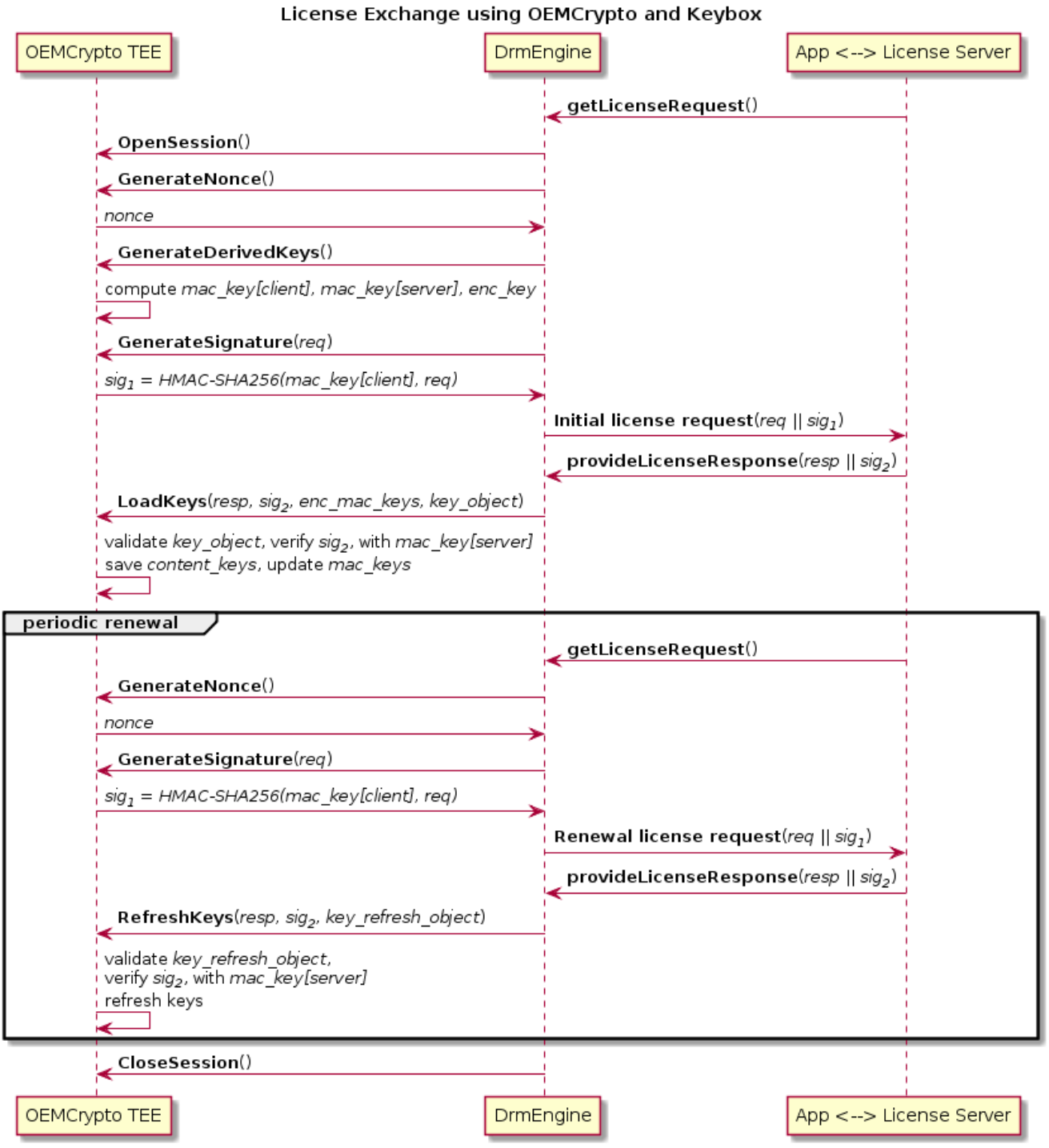
When the session is closed via OEMCrypto_CloseSession(), all of the Session Context resources must be explicitly cleared and then released.

License Signing and Verification Using Keybox

All license messages are signed to ensure that the license request and response can not be modified. The OEMCrypto implementation performs the signature generation and verification to

prevent tampering with the license messages.

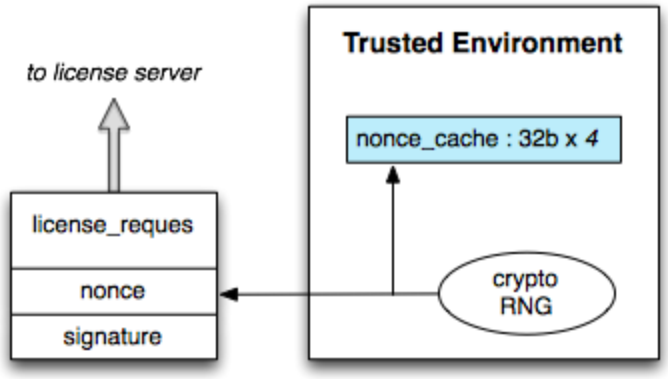
The sequence diagram below illustrates the interactions between the DrmEngine, the OEMCrypto Trusted Execution Environment (TEE) and the app, related to license signing and verification.



The app calls getLicenseRequest() to obtain an opaque license request message to send to the license server. The OEMCrypto calls OpenSession, GenerateDerivedKeys, GenerateNonce and GenerateSignature are used in the construction and signing of the request message. Once a license server response has been received, the app 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.



OEMCrypto_GenerateNonce()

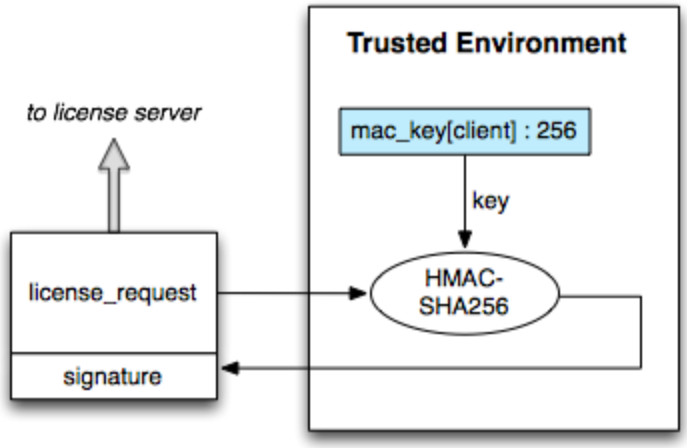
For the license initial and renewal *responses*, the OEMCrypto implementation must verify that the license response and its signature match.

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.

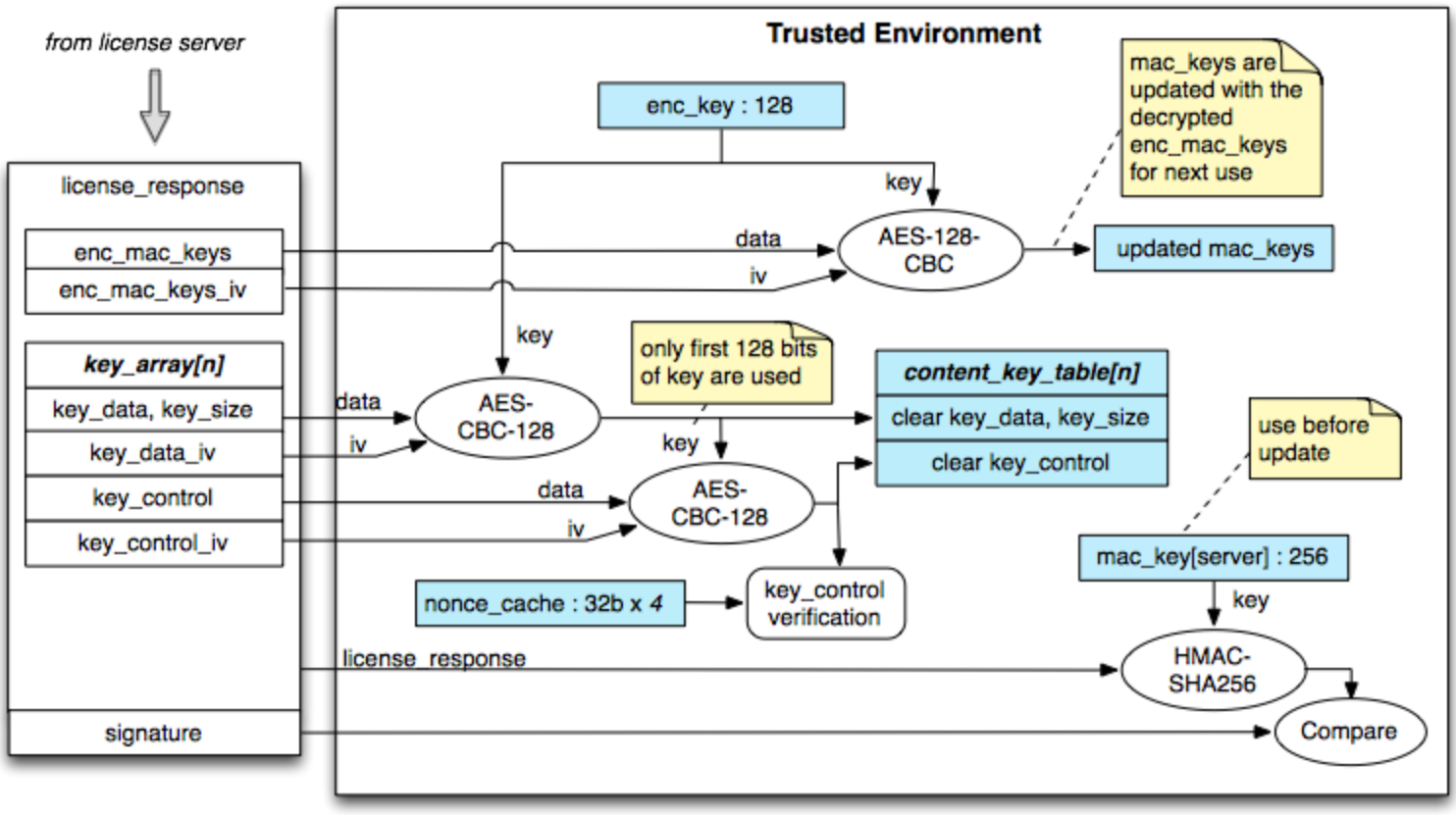
Note: When verifying the signature, the string comparison between the input signature and the recomputed signature should be a constant-time operation, to avoid leaking timing info.

The signatures for license initial and renewal requests are generated through the API call OEMCrypto_GenerateSignature().

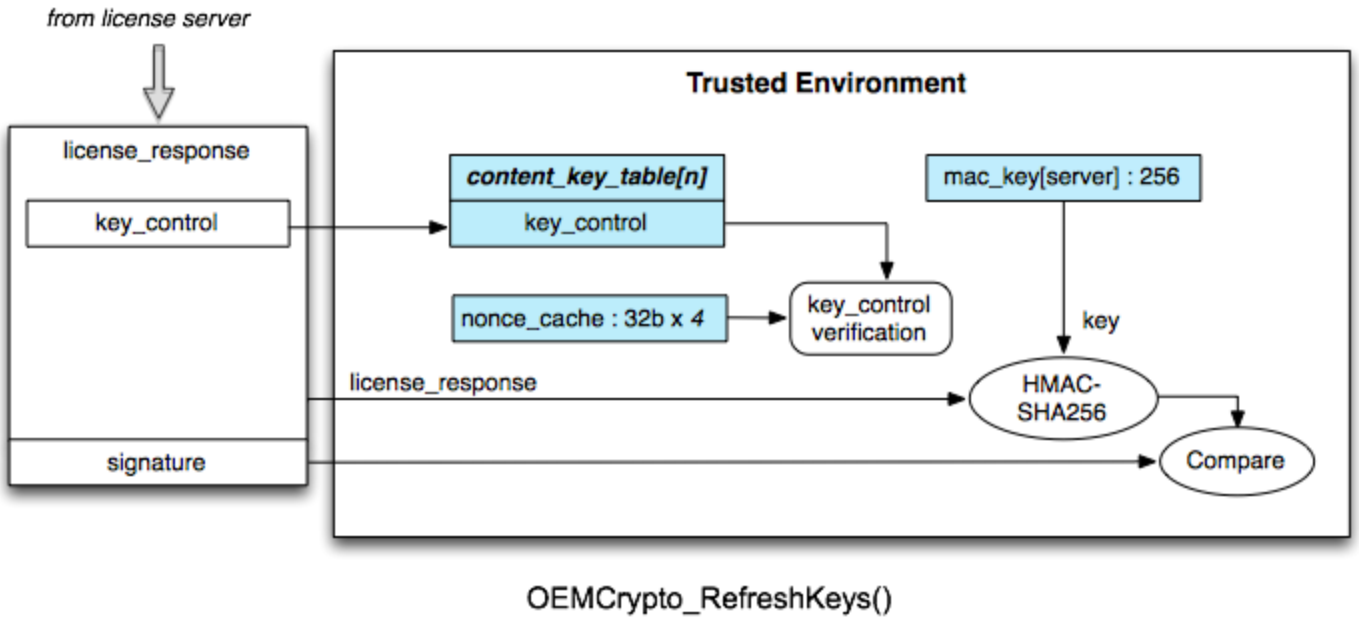


OEMCrypto_GenerateSignature()

The signature on the initial and renewal license response responses are verified within the OEMCrypto_Loadkeys() and OEMCrypto_RefreshKeys(), respectively. The signing algorithm is HMAC-SHA256.



OEMCrypto_LoadKeys()



OEMCrypto_RefreshKeys()

In addition to verifying the signature on the response messages, the implementations of OEMCrypto_LoadKeys() and OEMCrypto_RefreshKeys() must verify that the key_array entries are contained in the memory address range of the license response.

Key Derivation from Keybox: enc_key + mac_keys

For devices with a keybox, license signing and key encryption both depend on the device_key from the keybox. In order to avoid reusing the device_key for multiple purposes, separate keys are derived from the device_key, and the device_key is not used directly for any other purpose. Like the device key, these keys are never revealed in clear form.

Since version 9 of this API, most devices also support RSA Certificates, and will use a session key instead of the device_key from the keybox to derive the the encryption and signing keys. The algorithm described below is still used.

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]*

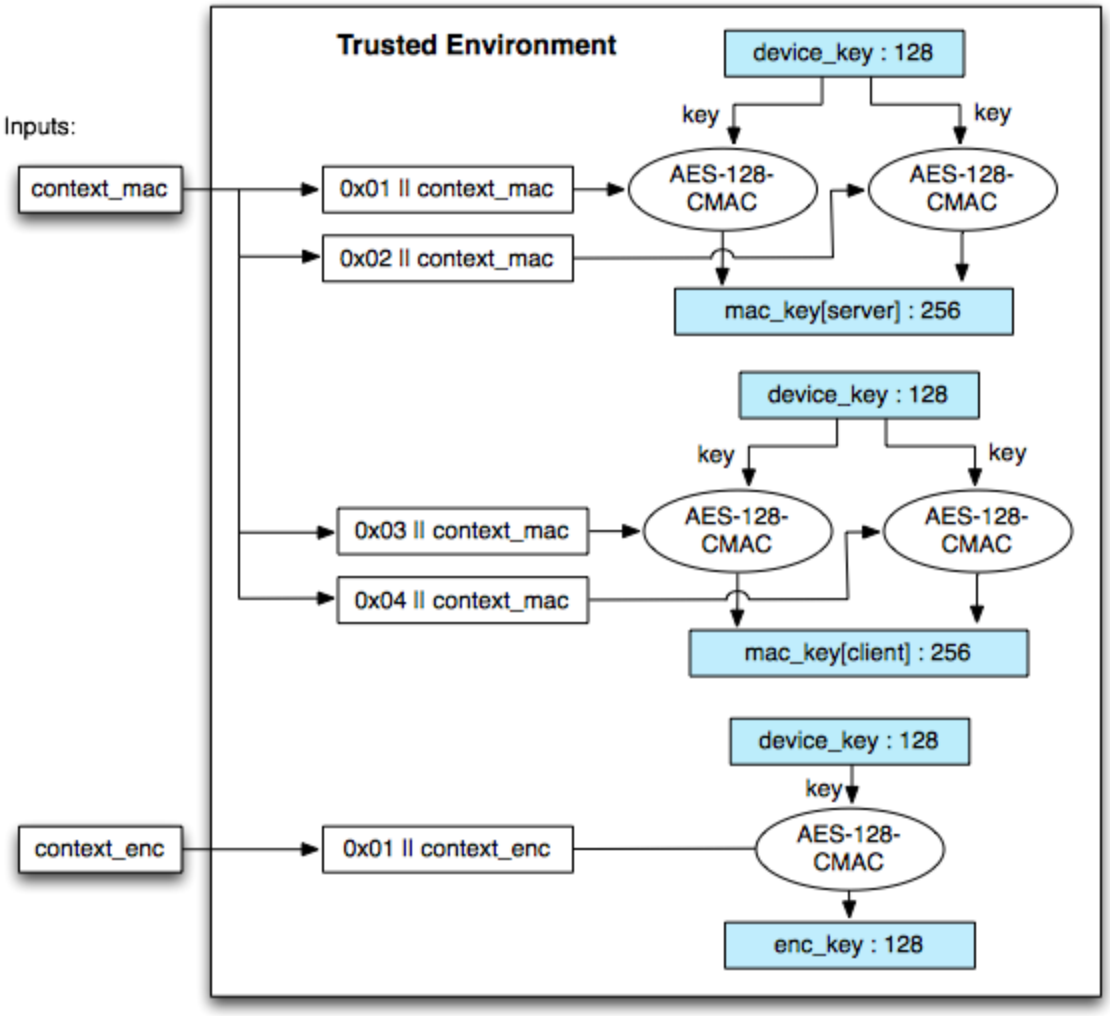
:= *AES-128-CMAC(device_key, 0x01* || *context_mac)* ||
AES-128-CMAC(device_key, 0x02 || *context_mac)* ||
AES-128-CMAC(device_key, 0x03 || *context_mac)* ||
AES-128-CMAC(device_key, 0x04 || *context_mac)*

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

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

where *context_enc* and *context_mac* are provided as parameters to the OEMCrypto API function generates these keys, and "||" represents the concatenation operation on message bytes.

The API call for generating the derived keys is OEMCrypto_GenerateDerivedKeys().



OEMCrypto_GenerateDerivedKeys()

Note: the mac_keys computed by OEMCrypto_GenerateDerivedKeys() will be replaced when OEMCrypto_LoadKeys() is called, as it receives new server-generated and encrypted mac_keys.

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 Android 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”, or “kc11”. 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 “Nonce Algorithm”.	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..21	Reserved set to 0
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 required 0x3 - HDCP version 2.1 required 0x4 - HDCP version 2.2 Type 1 required
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’ or ‘kc11’. 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 OEM/TEE Patch Level Enforcement for more details.
7. Apply the control fields:
    a. Replay_Control and Nonce_Enable -- if required, verify the nonce. See the next section (Nonce Algorithm) for details on verifying the nonce, and the following section (Replay Control and Nonce Requirements) 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.

Nonce Algorithm

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 algorithm is defined as follows:

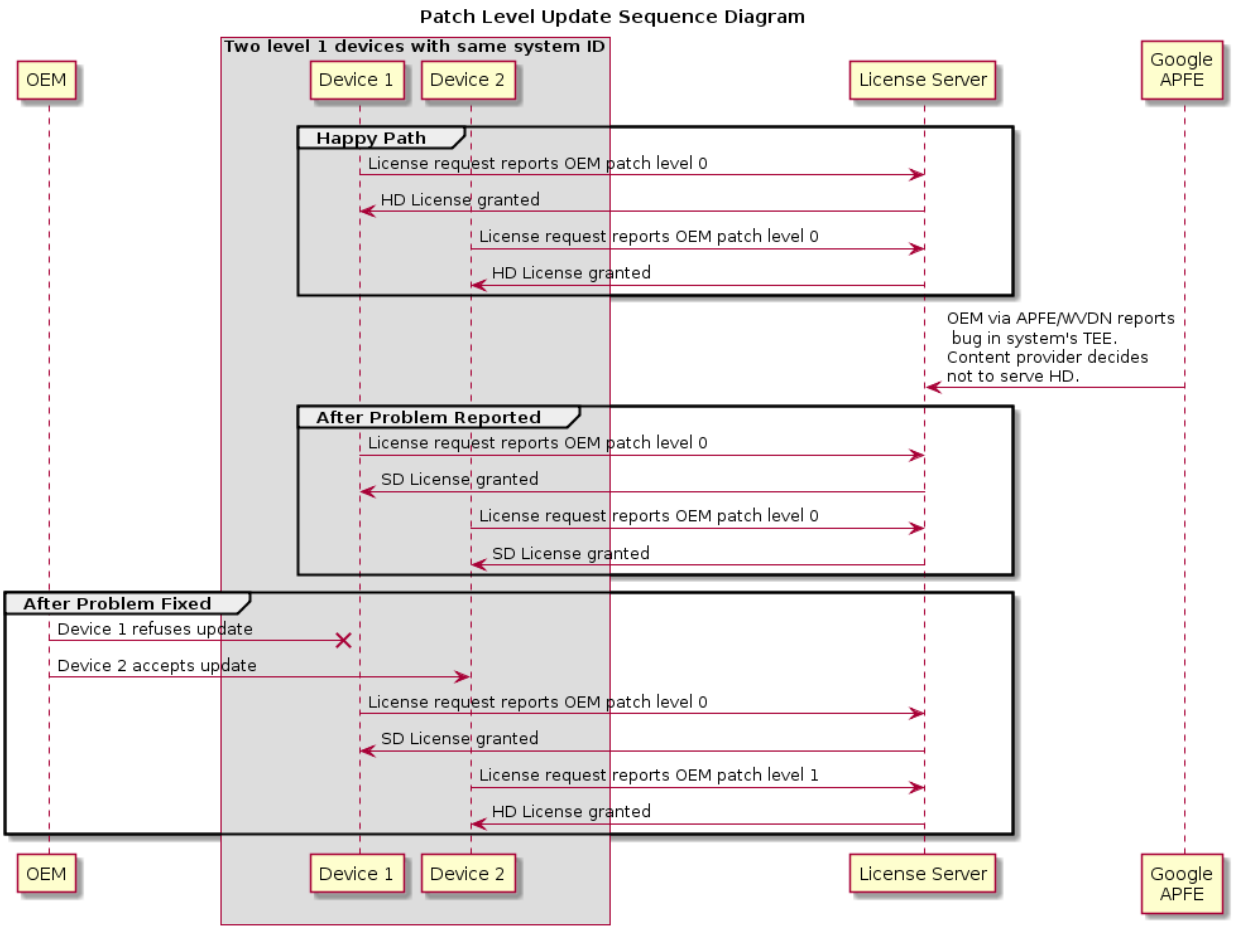
1. Nonce generation: a new nonce is generated by the OEMCrypto implementation at the request of the client, when OEMCrypto_GenerateNonce() is called. The nonce is placed in the license request. The OEMCrypto implementation shall generate a 32-bit cryptographically secure random number each time it is 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() and OEMCrypto_RefreshKeys(), and rejecting any keys whose nonce is not in the cache. If a nonce is in the cache, accept the key 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.

OEMCrypto Patch Level Enforcement

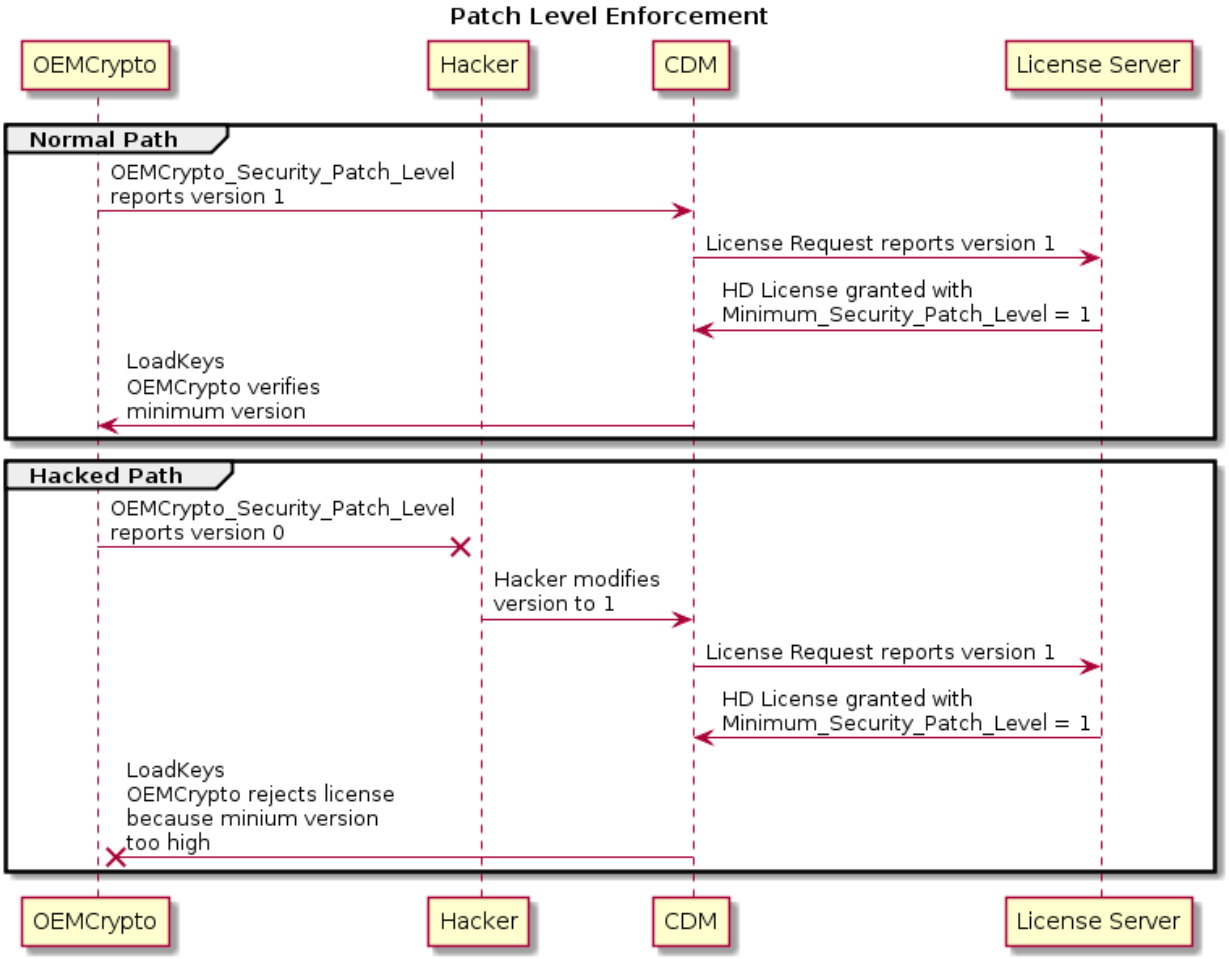
This feature addresses the desire of a content provider to serve licenses to a device only if it has a specific security patch. This feature allows the device to indicate that it has received a security patch. Notice that this feature will not distinguish between a device whose root of trust has been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



Replay Control and Nonce Requirements

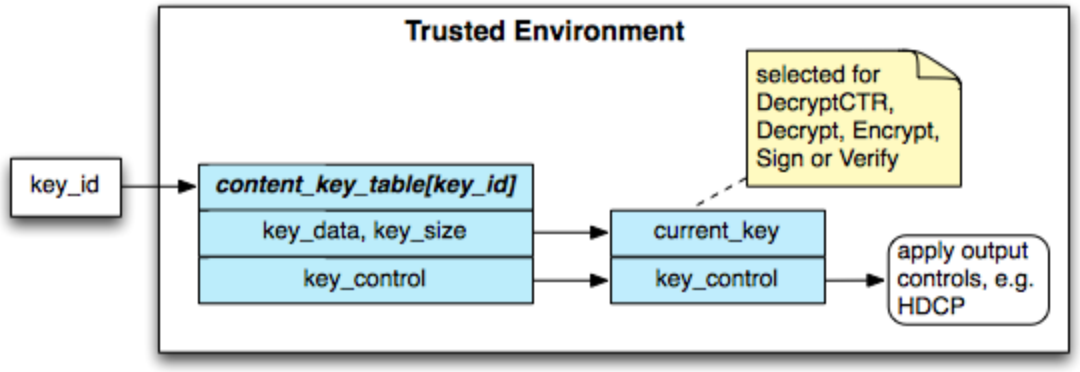
The replay control flag and the nonce enabled flag determine if a license may be used only once, may be reloaded until released, or may be reloaded indefinitely. An online license may be loaded only once, and requires a valid nonce from the nonce cache. An online license may also require that a new entry 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	Yes. OEMCrypto requires PST.
Streaming, no usage data required	0x0 - Session Usage table not required	1=Verify Nonce	No. OEMCrypto ignores pst.
Streaming, usage data required.	0x1 - Nonce required, create entry in Session Usage table	1=Verify Nonce	Yes. OEMCrypto requires PST.
Invalid - server will not send.	0x2 - Require existing Session Usage table entry or Nonce	1=Verify Nonce	n/a

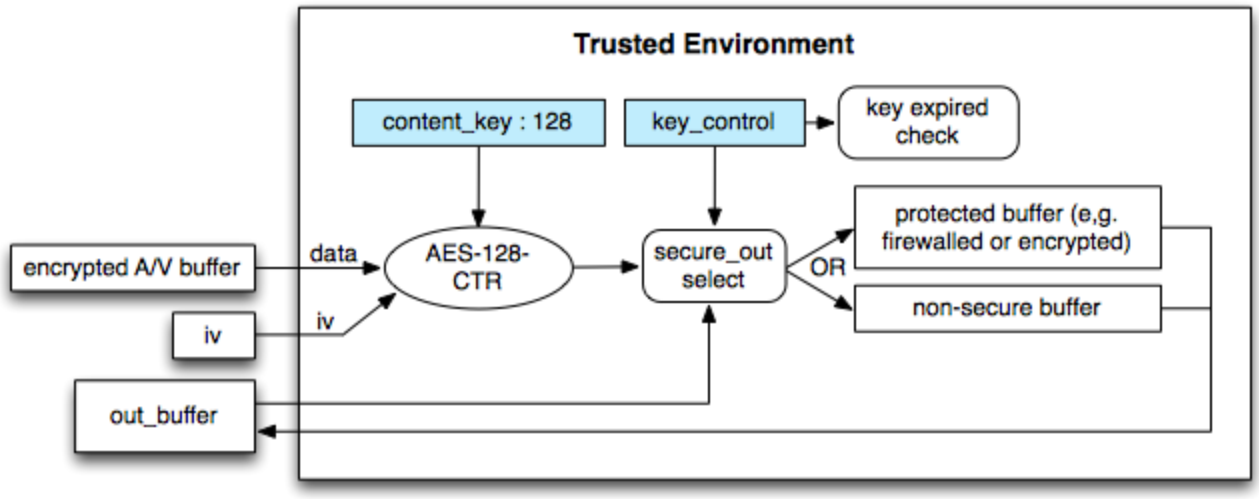
Content Decryption

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



OEMCrypto_SelectKey

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()

RSA Certificate Provisioning and License Requests

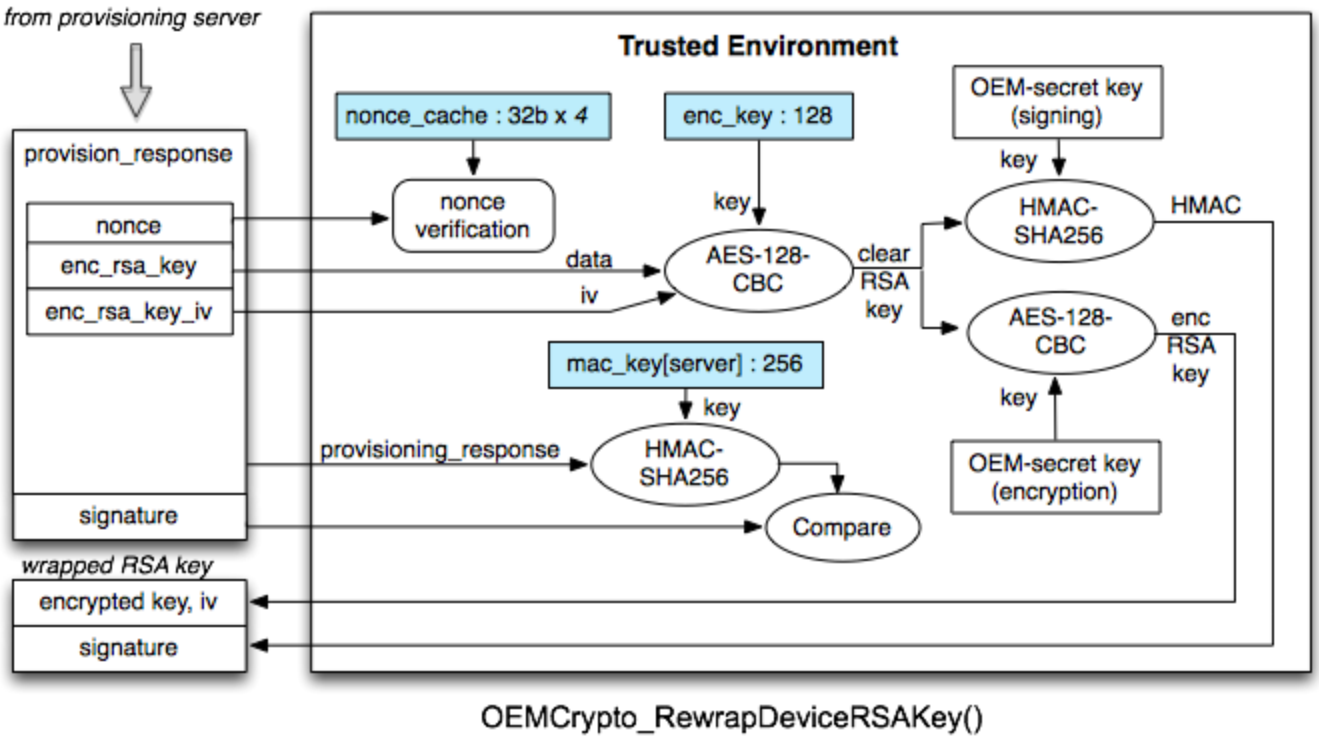
This section describes new features added in March, 2013, producing a V2.1 revision to the license protocol. The basic flow described in the previous sections can be modified to allow an application to use an RSA signed certificate for license requests instead of the Widevine Keybox. This allows the license server to grant a license without keeping a list of Widevine keybox system IDs and system keys. The device obtains a certificate from a provisioning server using the Widevine keybox as a root of trust. This logic flow adds only four new API functions because it leverages the existing OEMCrypto API.

Changes to Session

In addition to the existing state variable for a session, such as a nonce table, encryption keys, the session needs to store an RSA key pair in secure memory.

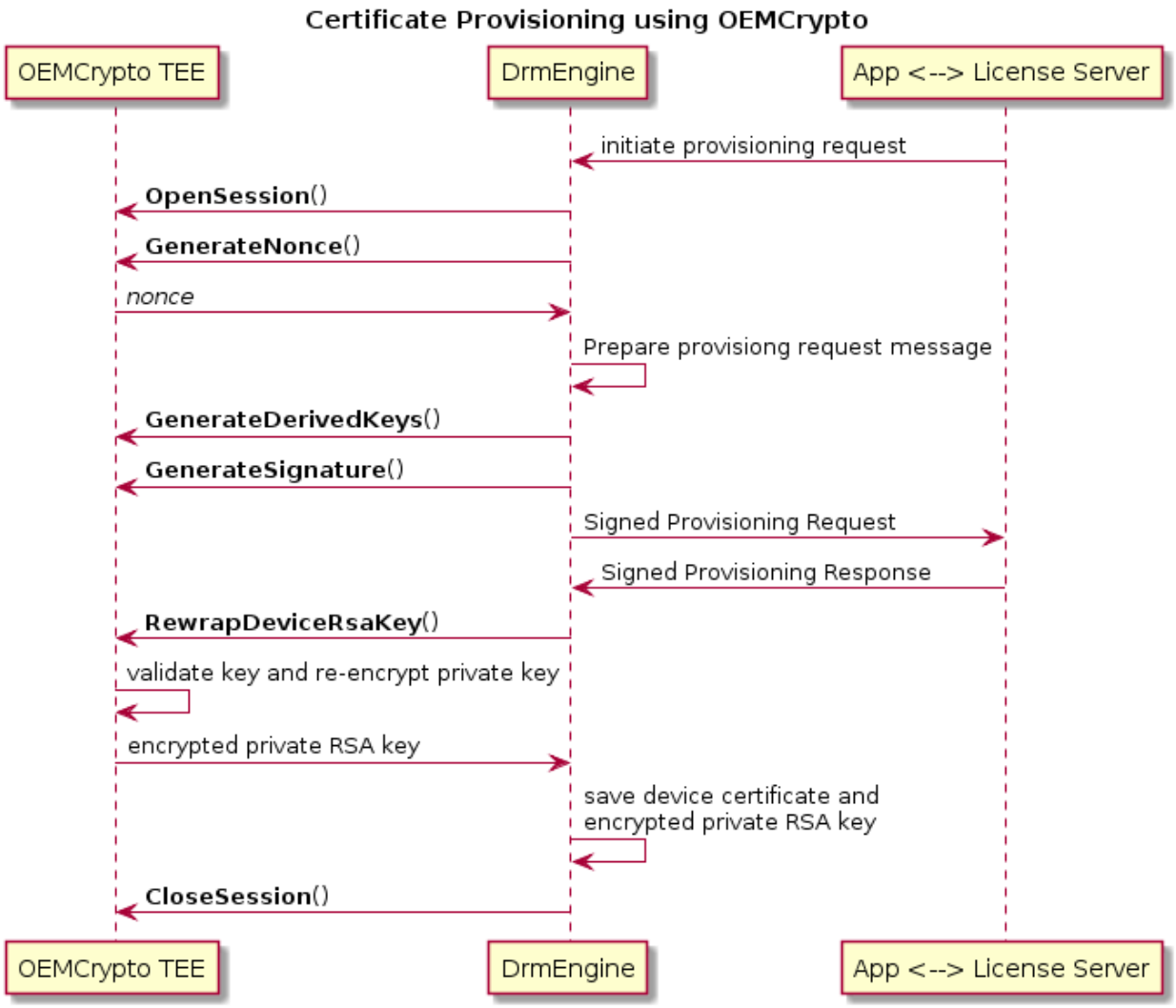
RSA Certificate Provisioning

There is one API function for provisioning a device with an RSA certificate. The RSA provisioning request is generated and signed in a similar way to the license request described above. This is sent to a provisioning server which can decrypt the Widevine keybox and send a provisioning response back. This response message contains a certificate and an RSA key pair.



OEMCrypto_RewrapDeviceRSAKey()

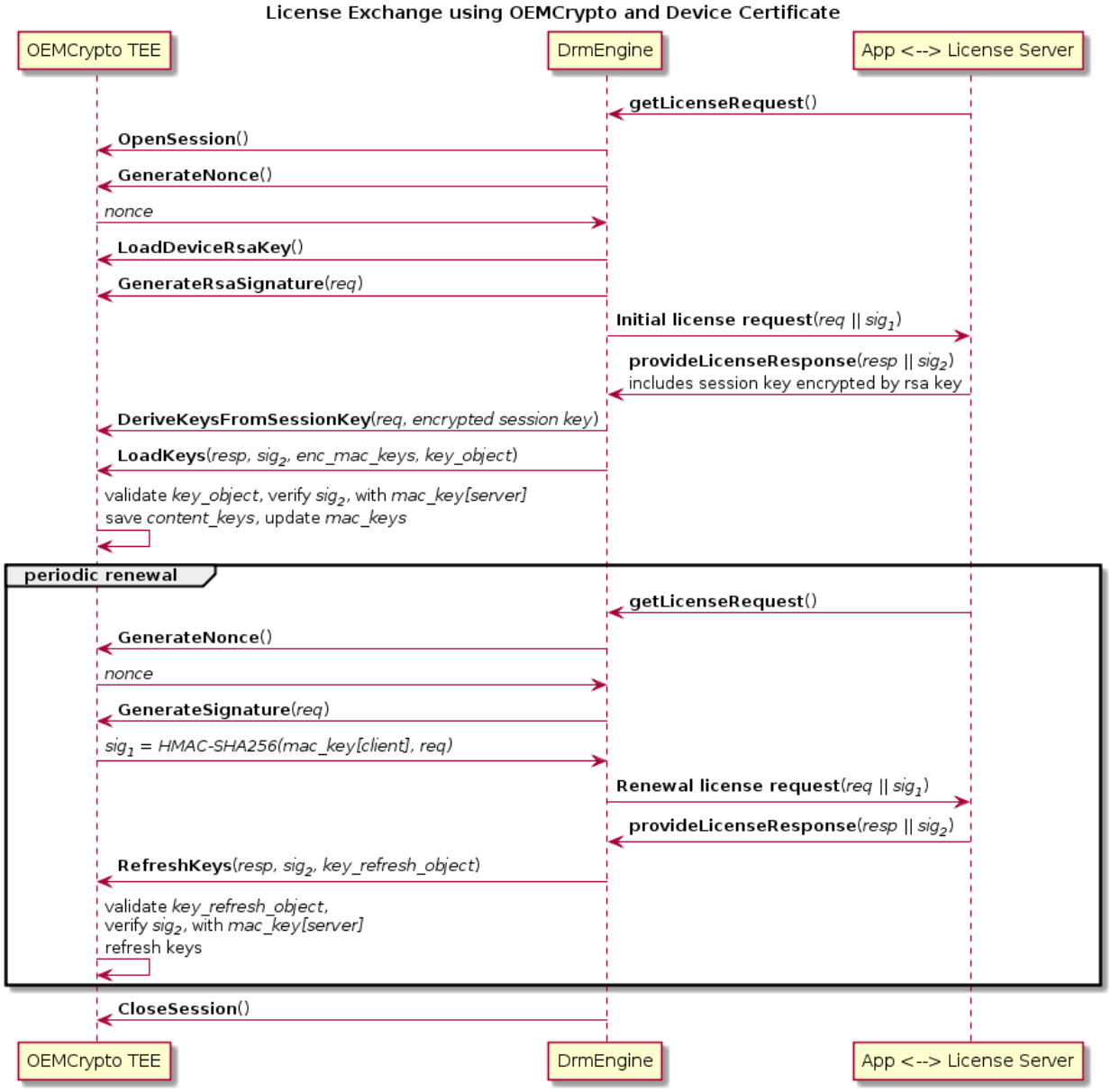
In the function OEMCrypto_RewrapDeviceRSAKey(), the device uses the encryption key, generated previously in OEMCrypto_GenerateDerivedKeys(), to decrypt the RSA private key and store it in secure memory. The device verifies the provisioning response message in much the same way it does in OEMCrypto_LoadKeys(). After decrypting the RSA key, it re-encrypts the private key using either the Widevine keybox device key, or an OEM specific device key --- this is called wrapping the key. This wrapped key is stored on the filesystem and passed back to the device whenever an RSA signed license request is needed.



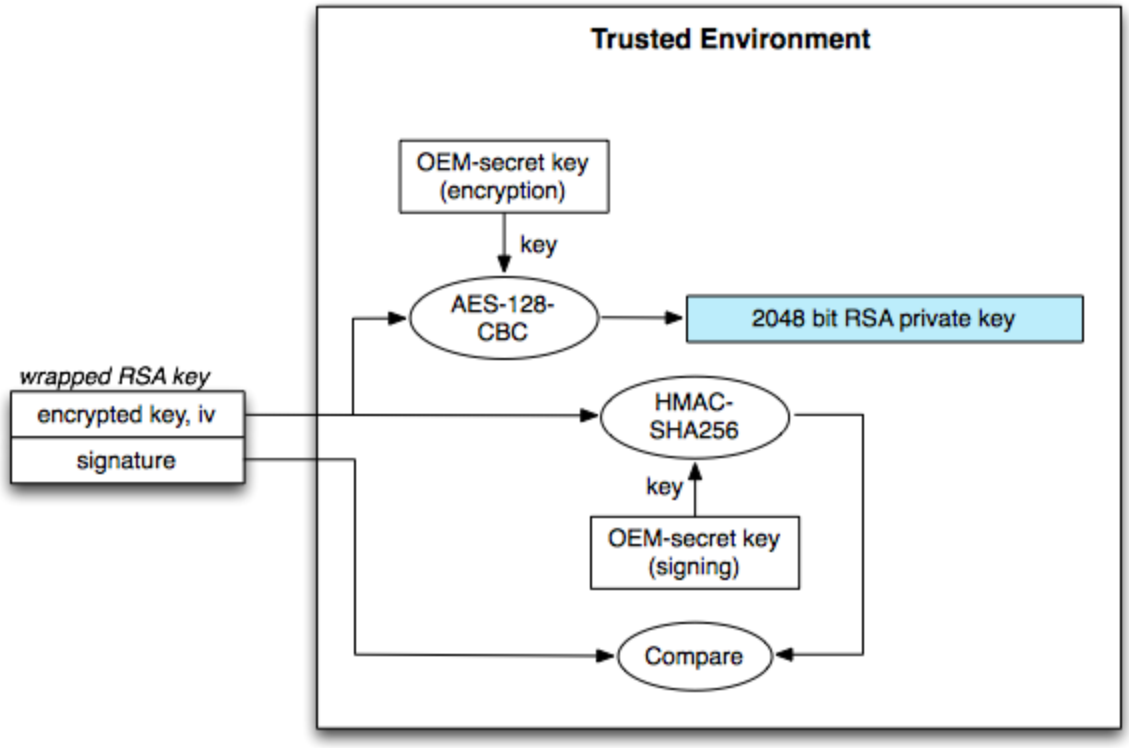
License Request Signed by RSA Certificate

Three functions, OEMCrypto_LoadDeviceRSAKey(), OEMCrypto_GenerateRSASignature(), and OEMCrypto_DeriveKeysFromSessionKey() are used to implemented the license exchange protocol when using a device certificate as the device root of trust. The following diagram shows

OEMCrypto call sequence during the license exchange:

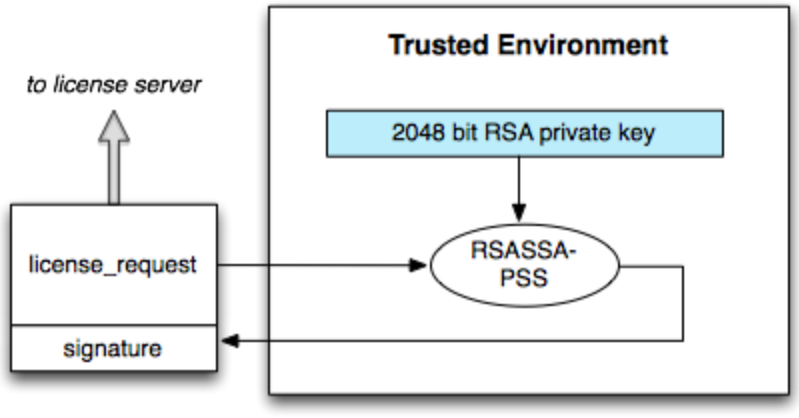


The first function is OEMCrypto_LoadDeviceRSAKey(), is passed a wrapped RSA key pair. It unwraps the key pair and stores it in secure memory.



OEMCrypto_LoadDeviceRSAKey()

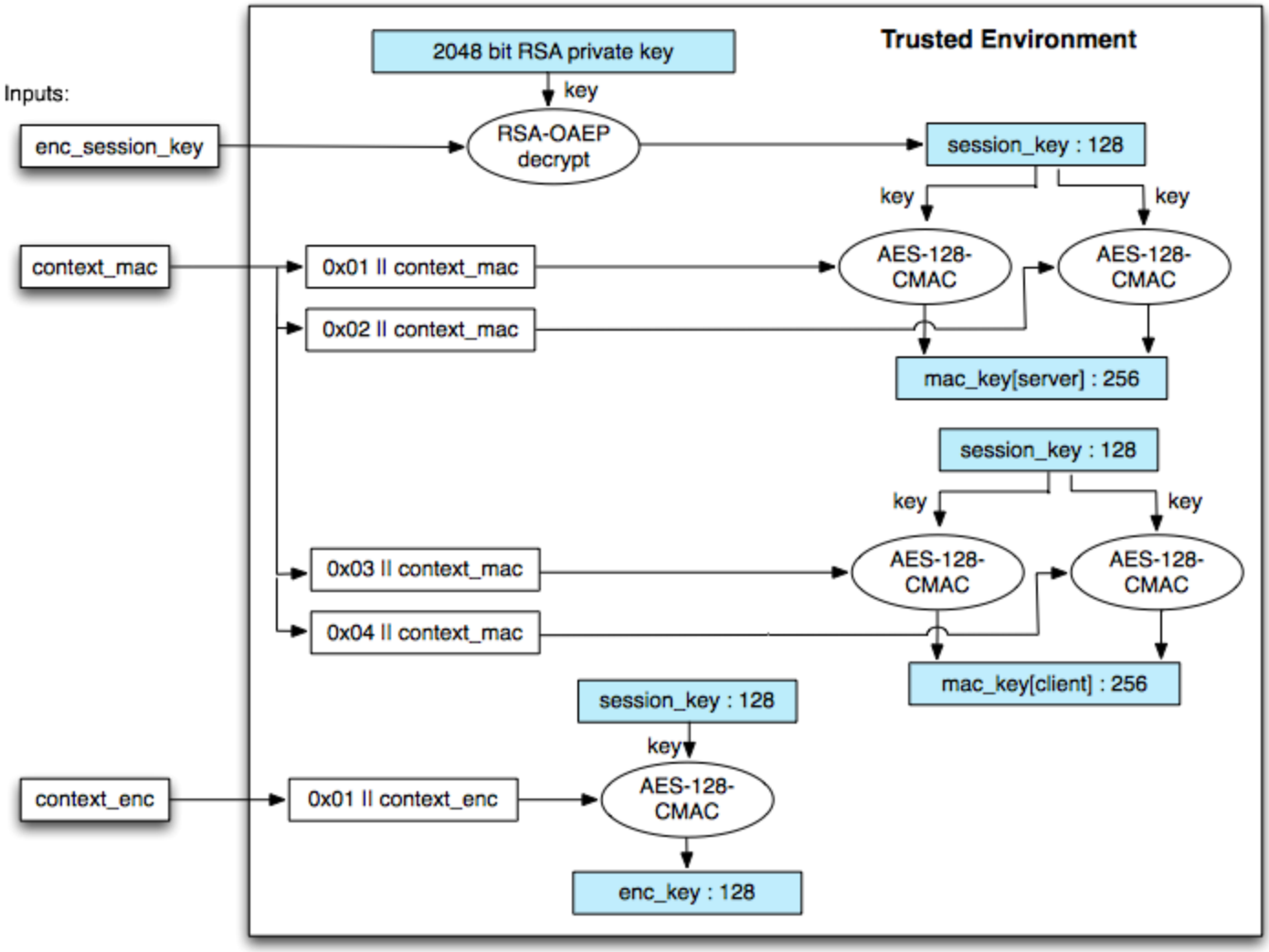
The second function, OEMCrypto_GenerateRSASignature(), signs a message using the device RSA private key.



OEMCrypto_GenerateRSASignature()

The third function, OEMCrypto_DeriveKeysFromSessionKey(), is similar to OEMCrypto_GenerateDerivedKeys. It is given an encrypted session key, and two context

strings. It should decrypt the session key using the private RSA key. Then it uses the session key to generate an encryption key and mac key.



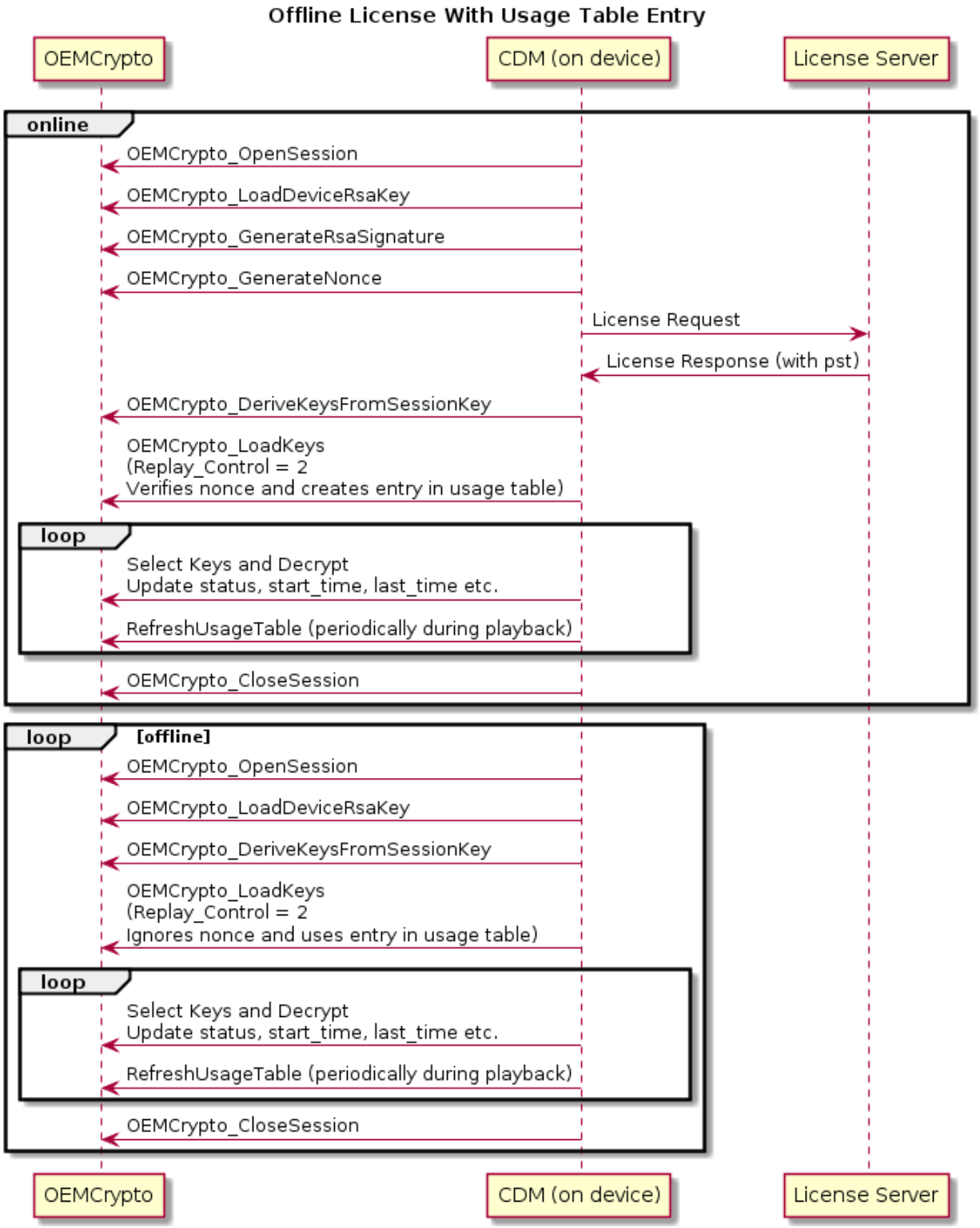
OEMCrypto_DeriveKeysFromSessionKey()

Session Usage Table and Reporting

The Session Usage Table is a feature that is new for version 9 of the OEMCrypto API. Its main two use cases are for 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.

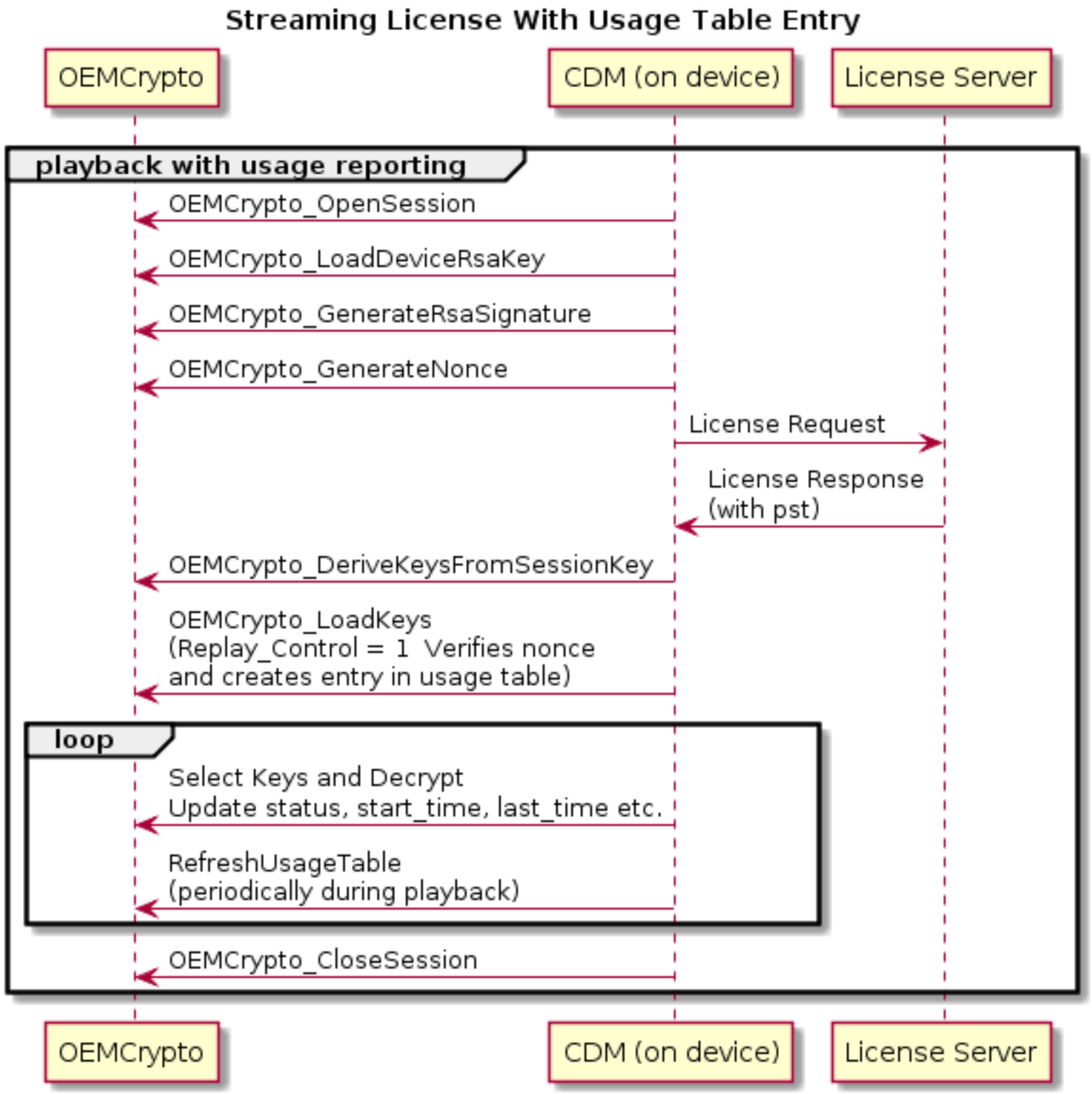
Keys that are intended for offline playback will need to be loaded several times, without access to a new license response. The API is designed so that the first time such a key is loaded, it must have a valid nonce matching the license request. The key will then be loaded into the usage table. For any 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.

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 session using this key is closed, 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.



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.

The Session Usage Table stores entries based on a Provider Session Token (or 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_LoadKeys and may be deleted from a call to OEMCrypto_DeleteUsageTable. The table must be secure from user inspection, modification, or rollback: 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.

If it is not possible to store the entire table in secure memory, the following scheme is recommended. A Generation Number is stored in secure memory. This number will be incremented once, every time the table is modified. The same number will be stored in the table, the table will then be encrypted and signed, and written to the device's file system. The encryption and signing key should be based on a device specific key, such as one derived from the device key in the keybox.

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 deleted.

HMAC-SHA256 signature			Generation Number		
Provider Token (pst)	Signing Keys	Time @ load key	Time @ 1st decrypt	Time @ last decrypt	Status
:	:	:	:	:	:

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

```
 {
  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 by refresh keys.
  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[variable size];
}
```

Because the signing mac keys are sensitive, these keys must be encrypted before saving them to the file system, or the entire table must be encrypted before saving to the file system.

Because the PST is not of fixed length, the entries in the usage table are also not fixed length. The table will use the PST value as the key, so each entry in the table will have a unique PST value.

When an entry is created, in LoadKeys, the value of time_of_license_received is copied from the secure clock. The server_mac_key and client_mac_key are also copied from the session to the Usage Table when the entry is created in LoadKeys.

An entry in the Usage Table will be associated with an open session when a call to LoadKeys is made. This association will be used to update time_of_last_decrypt whenever the Usage Table is updated. The association is not saved with the usage table -- if the entry is to be updated, a new session will be opened.

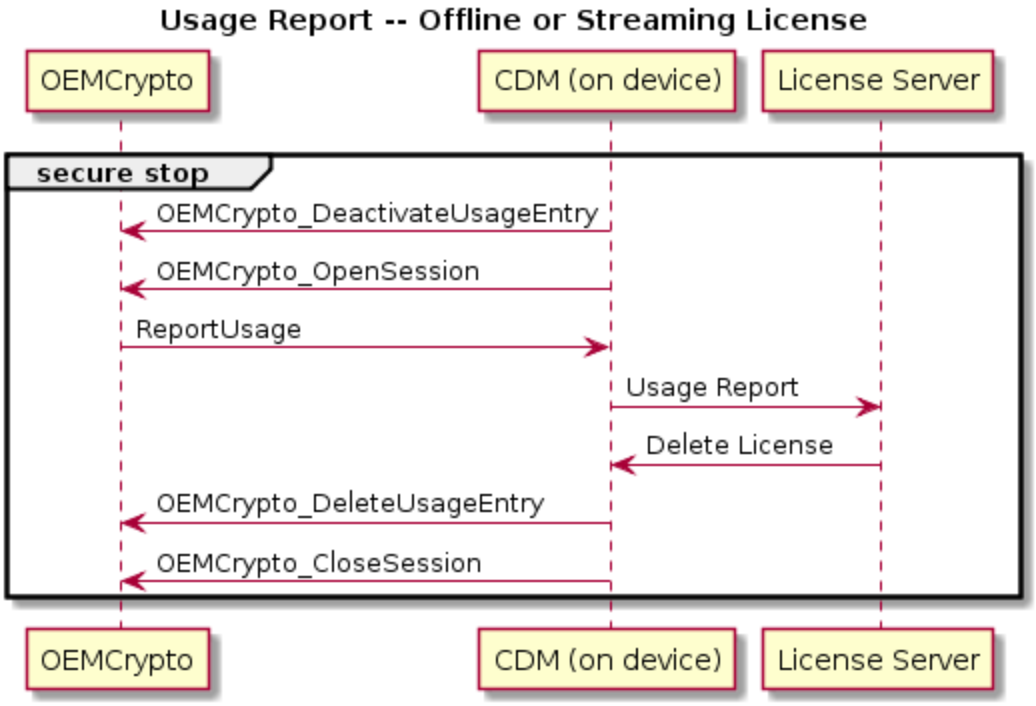
While the amount of persistent insecure memory is probably not a significant limitation, the session usage table must be kept in secure RAM in the TEE, and that will likely impose a limit on some devices. When out of memory, OEMCrypto should remove entries from the table that are not associated with a currently open session using LRU (least recently used) on time_of_license_received. There should be room in the table for at least 50 entries.

Entries in the table may have the following status values:

```
enum UsageEntryStatus {
        kUnused = 0, // decrypt not yet called
        kActive = 1, // keys not released
        kInactive = 2, // keys released
};
```

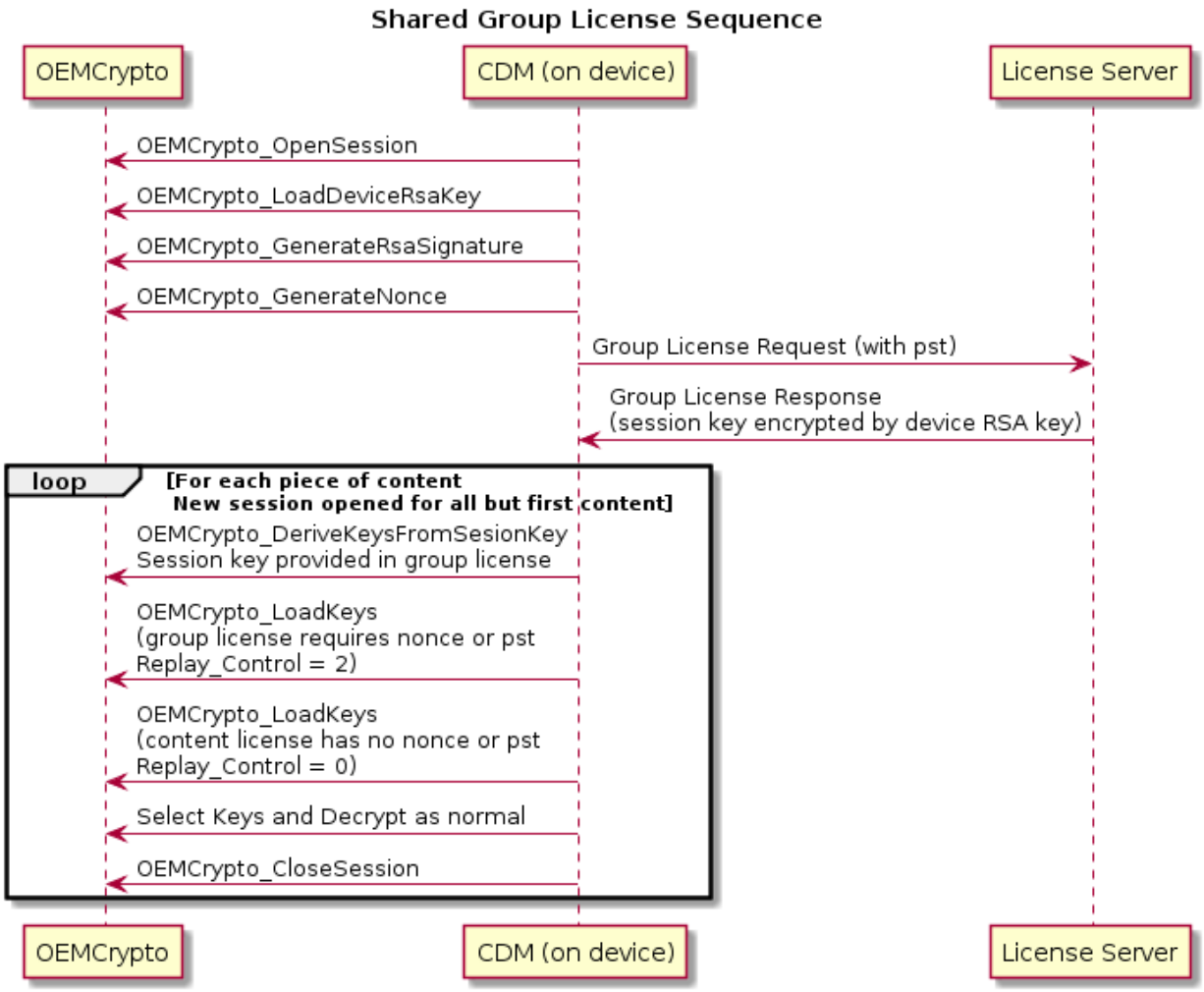
Once an entry has been marked “inactive”, any license or session associated with that entry in the table may no longer be used to decrypt or encrypt data. The entry will be kept until a usage report has been sent to the server and an acknowledgement has returned.

Below is the sequence diagram for Session Usage Reporting, which illustrates how the Session Usage Table will be used to report secure stops for an online streaming license. This sequence would happen after either the offline or streaming sequences shown above.



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 group 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 group 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. Below is a sequence diagram.



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 an RSA 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 an RSA certificate to generate license requests. Many content providers prefer to use RSA 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 RSA 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 RSA certificate from the Widevine provisioning server. This essentially makes it unable to use RSA 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.

OEMCrypto API for CENC

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

There are five areas exposed by OEMCrypto APIs:

- Crypto Device Control API
- Crypto Key Ladder API
- Decryption API
- Provisioning API
- Keybox Access
- RSA Certificate Provisioning API
- Usage Table API

Device manufacturers implement the API as a static library, which is linked into the Widevine DRM plugin.

Crypto Device Control API

The Crypto Device Control API involves initialization of and mode control for the security hardware. The following table 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 table shows the APIs required for key management:

OEMCrypto_OpenSession
OEMCrypto_CloseSession
OEMCrypto_GenerateDerivedKeys
OEMCrypto_GenerateNonce
OEMCrypto_GenerateSignature
OEMCrypto_LoadKeys
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 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.

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 existing sessions may be called while this function is active.

Version

This method changed in API version 5.

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 License Signing and Verification 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

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_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.

NOTE: OEMCrypto_GenerateDerivedKeys() must be called first to establish the mac_key[client].

Refer to the License Signing and Verification section above for more details.

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 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 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

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_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);
```

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

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 License Signing and Verification section above for more details.

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 num_keys == 0, then return OEMCrypto_ERROR_INVALID_CONTEXT.

Usage Table and Provider Session Token (pst)

If a key control block has a nonzero value for Replay_Control, then all keys in this license will have the same value. 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.

- If Replay_Control is 1 = Nonce_Required, then OEMCrypto will perform a nonce check as described above. OEMCrypto will verify that the table does not already have an entry for the value of pst passed in as a parameter --- if an entry already exists, an error OEMCrypto_ERROR_INVALID_CONTEXT is returned and no keys are loaded.

OEMCrypto will then create a new entry in the table, and mark this session as using this new entry. This 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 check the Session Usage table for an existing entry with the same pst.
    - If the pst is not in the table yet, a new entry will be created in the table and this session **shall** use the new entry. In that case, the nonce will be verified for each key.
    - If an existing usage table entry is found, then this session will use that entry. In that case, the nonce will **not** be verified for each key. 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 an existing entry is not found, the return error is OEMCrypto_ERROR_INVALID_NONCE.

Note: If LoadKeys updates the mac keys, then the new updated mac keys will be used with the Usage Table -- 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.
Note: If LoadKeys creates a new entry in the usage table, OEMCrypto will increment the Usage Table’s generation number, and then sign, encrypt, and save the Usage Table.

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.

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

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 11.

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_GenerateDerivedKeys() or OEMCrypto_LoadKeys() must be called first to establish the mac_key[server].

This session's elapsed time clock is reset to 0 when this function is called. The elapsed time clock is used in OEMCrypto_DecryptCENC().

This function does not add keys to the key table. It is only used to update a key control block license duration. Refer to the License Signing and Verification section above for more details. 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.

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.

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

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_QueryKeyControl

```
OEMCryptoResult
OEMCrypto_QueryKeyControl(const OEMCrypto_SESSION session,
                          const uint8_t* key_id,
                          size_t key_id_length,
                          uint8_t* key_control_block,
                          size_t* key_control_block_length);
```

Returns the decrypted key control block for the given 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] key_id: The unique id of the key of interest.

[in] key_id_length: The length of key_id, in bytes.

[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

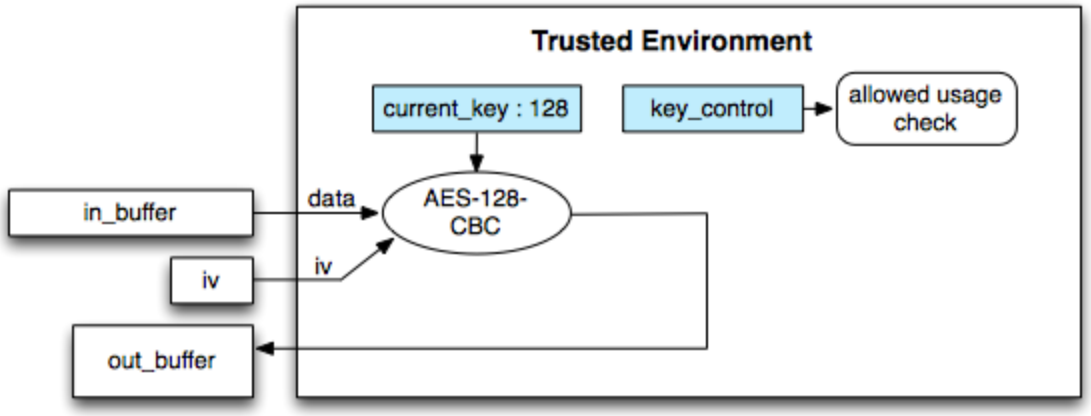
This method is new in API version 10.

Decryption API

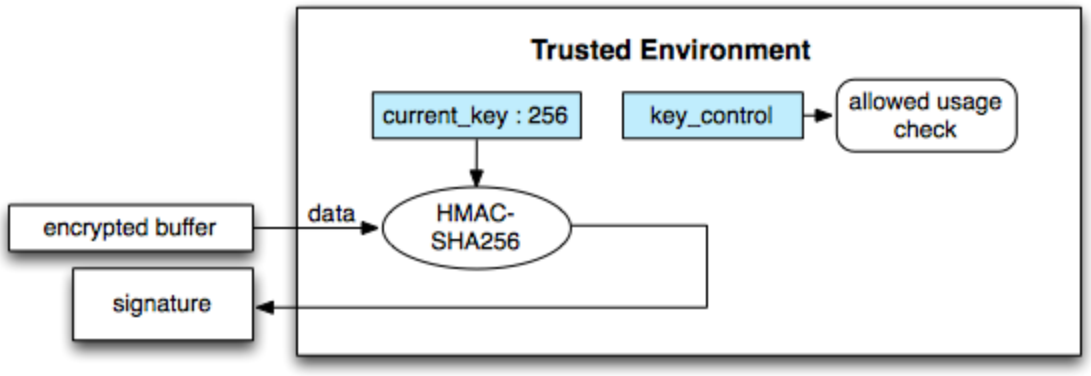
Devices that implement the Key Ladder API must also support a secure decode or secure decode and rendering implementation. This can be done by either decrypting into buffers secured by hardware protections and providing these secured buffers to the decoder/renderer or by implementing decrypt operations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

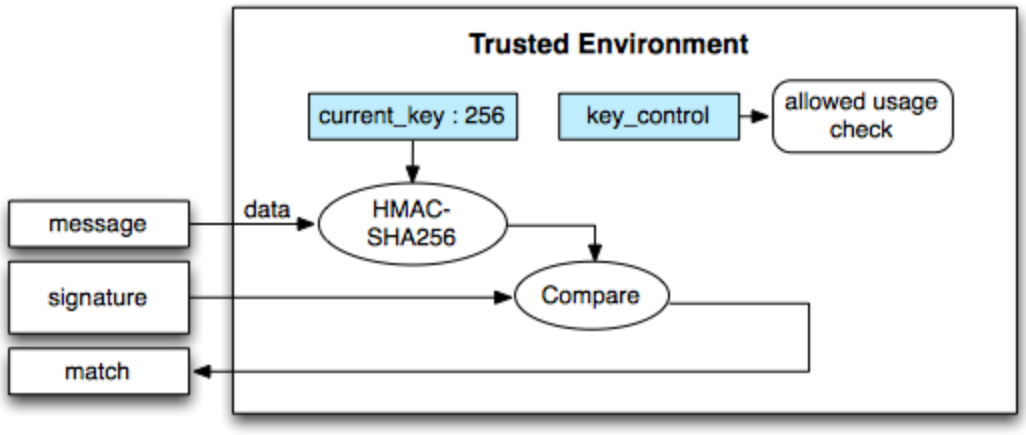
Generic Modular DRM allows an application to encrypt, decrypt, sign and verify arbitrary user data using a content key. This content key is securely delivered from the server to the client device using the same factory installed root of trust as a media content keys.



OEMCrypto_Generic_Decrypt(), OEMCrypto_Generic_Encrypt()



OEMCrypto_Generic_Sign()



OEMCrypto_Generic_Verify()

The following table shows the APIs required for decryption:

OEMCrypto_SelectKey
OEMCrypto_DecryptCENC
OEMCrypto_CopyBuffer
OEMCrypto_Generic_Encrypt
OEMCrypto_Generic_Decrypt
OEMCrypto_Generic_Sign
OEMCrypto_Generic_Verify

OEMCrypto_SelectKey

```
OEMCryptoResult OEMCrypto_SelectKey(const OEMCrypto_SESSION session,
                                    const uint8_t* key_id,
                                    size_t key_id_length);
```

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. Continue to use this key until OEMCrypto_SelectKey() is called again, or until OEMCrypto_CloseSession() is called.

Parameters

[in] session: crypto session identifier.

[in] key_id: pointer to the Key ID.

[in] key_id_length: length of the Key ID, in bytes.

Returns

OEMCrypto_SUCCESS success

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

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;  // offset into the pattern in blocks for this call.
} 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 set in the OEMCrypto_KeyObject passed in to OEMCrypto_LoadKeys. 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 rendered.

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.

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_LoadKeys. The encryption pattern is specified in by 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”.

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’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.
4. If the current key’s control block has a nonzero value for HDCP_Version, then the current version of HDCP for the device and the display combined will be compared against the version specified in the control block. If the current version is not at least as high as that in the control block, then return OEMCrypto_ERROR_INSUFFICIENT_HDCP.
5. If the current session has an entry in the Usage Table, and the status of that entry is “inactive”, then return OEMCrypto_ERROR_INVALID_SESSION.

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 $0 \leq blockoffset < 16$.

[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_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 11. 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

Threading

This function may be called simultaneously with any other functions.

Version

This method is new in API version 10.

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);
```

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.

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 “inactive”, then return OEMCrypto_ERROR_INVALID_SESSION.

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.
[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

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_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.

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 "inactive", then return OEMCrypto_ERROR_INVALID_SESSION.

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.
[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

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_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 “inactive”, then return OEMCrypto_ERROR_INVALID_SESSION.

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

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_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 “inactive”, then return OEMCrypto_ERROR_INVALID_SESSION.

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

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.

Provisioning API

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

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 table shows the APIs required for provisioning:

OEMCrypto_WrapKeybox - optional
OEMCrypto_InstallKeybox - optional
OEMCrypto_LoadTestKeybox - required

OEMCrypto_WrapKeybox

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

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. As described in section 5.5.4, the keybox may be directly encrypted and stored on the device in a single step, or it may 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().

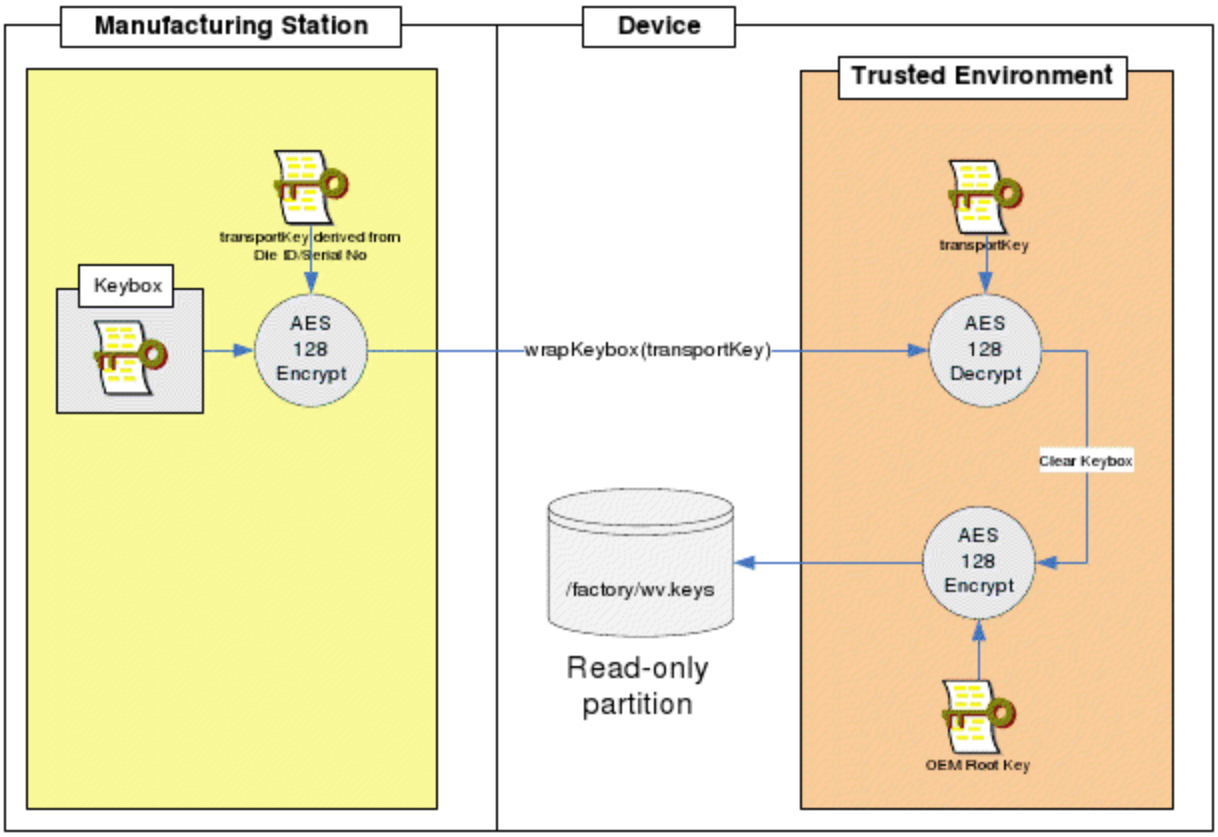


Figure 10. OEMCrypto_WrapKeybox Operation

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 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.

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.

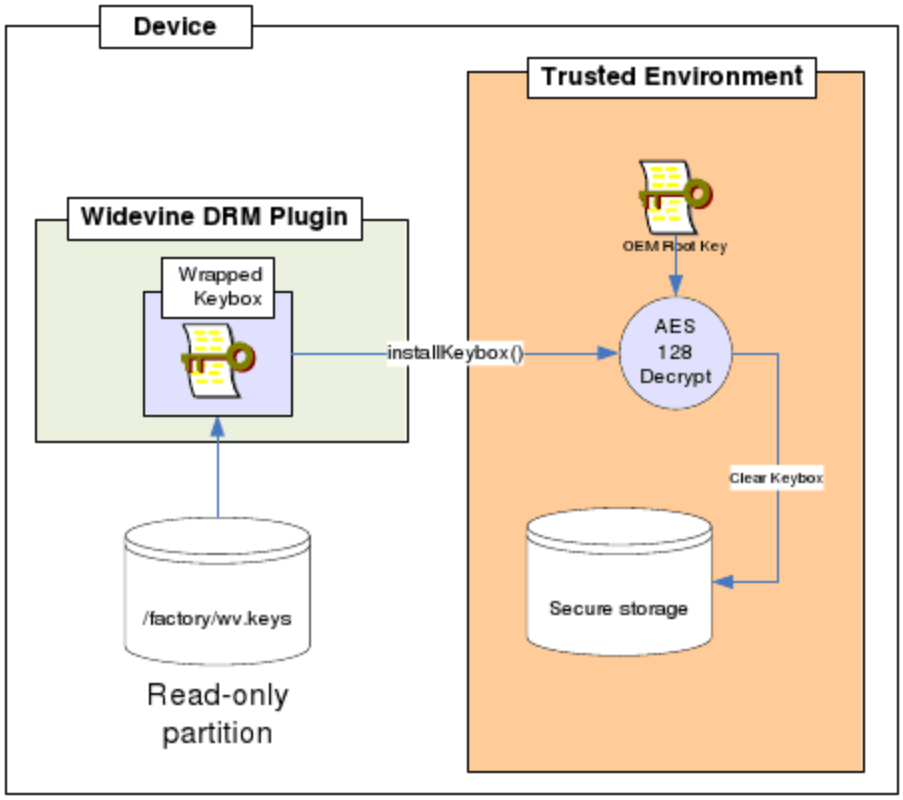


Figure 11 - Install keybox Operation

OEMCrypto_LoadTestKeybox

```
OEMCryptoResult OEMCrypto_LoadTestKeybox();
```

Temporarily use the standard 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.

The test keybox, as seen in the reference implementation of OEMCrypto is:

```
const WidevineKeybox kKeybox = {
  {
    // deviceID
    0x54, 0x65, 0x73, 0x74, 0x4b, 0x65, 0x79, 0x30,  // TestKey01
    0x31, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // ........
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // ........
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // ........
  }, {
    // key
    0xfb, 0xda, 0x04, 0x89, 0xa1, 0x58, 0x16, 0x0e,
    0xa4, 0x02, 0xe9, 0x29, 0xe3, 0xb6, 0x8f, 0x04,
  }, {
    // data
    0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x10, 0x19,
    0x07, 0xd9, 0xff, 0xde, 0x13, 0xaa, 0x95, 0xc1,
    0x22, 0x67, 0x80, 0x53, 0x36, 0x21, 0x36, 0xbd,
    0xf8, 0x40, 0x8f, 0x82, 0x76, 0xe4, 0xc2, 0xd8,
    0x7e, 0xc5, 0x2b, 0x61, 0xaa, 0x1b, 0x9f, 0x64,
    0x6e, 0x58, 0x73, 0x49, 0x30, 0xac, 0xeb, 0xe8,
    0x99, 0xb3, 0xe4, 0x64, 0x18, 0x9a, 0x14, 0xa8,
    0x72, 0x02, 0xfb, 0x02, 0x57, 0x4e, 0x70, 0x64,
    0x0b, 0xd2, 0x2e, 0xf4, 0x4b, 0x2d, 0x7e, 0x39,
  }, {
    // magic
    0x6b, 0x62, 0x6f, 0x78,
  }, {
    // Crc
    0x0a, 0x7a, 0x2c, 0x35,
  }
};
```

Parameters

none

Returns

OEMCrypto_SUCCESS success

OEMCrypto_ERROR_INSUFFICIENT_RESOURCES

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.

Keybox Access and Validation 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 table shows the APIs required for keybox validation:

OEMCrypto_IsKeyboxValid
OEMCrypto_GetDeviceId
OEMCrypto_GetKeyData
OEMCrypto_GetRandom
OEMCrypto_APIVersion
OEMCrypto_Security_Patch_Level
OEMCrypto_SecurityLevel
OEMCrypto_GetHDCPCapability
OEMCrypto_SupportsUsageTable
OEMCrypto_IsAntiRollbackHwPresent
OEMCrypto_GetNumberOfOpenSessions
OEMCrypto_GetMaxNumberOfSessions

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

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

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

Threading

This function may be called simultaneously with any session functions.

Version

This method is supported in all API versions.

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

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 11. 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 OEM/TEE Patch Level Enforcement 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
  HDCP_V2_1  = 3,  // HDCP version 2.1
  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 v2.2 is Type 1: all downstream devices will also use HDCP v2.2.

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 prevent the rollback of the usage table. For example, if the usage table contains 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.

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.

RSA Certificate Provisioning API

As an alternative to using the Widevine Keybox device key to sign the license request, this collection of APIs provide a way to use an RSA signed certificate. The certificate is generated by a provisioning server, and the certificate is used when communicating with a license server. Communication with the provisioning server is still authenticated with the keybox.

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

OEMCrypto_RewrapDeviceRSAKey
OEMCrypto_LoadDeviceRSAKey
OEMCrypto_LoadTestRSAKey
OEMCrypto_GenerateRSASignature
OEMCrypto_DeriveKeysFromSessionKey

OEMCrypto_RewrapDeviceRSAKey

```
OEMCryptoResult OEMCrypto_RewrapDeviceRSAKey(
                        OEMCrypto_SESSION session,
                        const uint8_t* message,
                        size_t message_length,
                        const uint8_t* signature,
                        size_t signature_length,
                        uint32_t *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);
```

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

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]).
5. Decrypt enc_rsa_key using the derived encryption key (enc_key), and enc_rsa_key_iv.
6. Validate the decrypted RSA device key by verifying that it can be loaded by the RSA implementation.
7. Generate a random initialization vector and store it in `wrapped_rsa_key_iv.`
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] 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

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_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 an RSA 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 RSA 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 License Request Signed by RSA Certificate section above for more details.

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.

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_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 device RSA key (from the device 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

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.

Usage Table API

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

OEMCrypto_UpdateUsageTable
OEMCrypto_DeactivateUsageEntry
OEMCrypto_ReportUsage

OEMCrypto_DeleteUsageEntry
OEMCrypto_ForceDeleteUsageEntry
OEMCrypto_DeleteUsageTable

OEMCrypto_UpdateUsageTable

```
OEMCryptoResult OEMCrypto_UpdateUsageTable();
```

OEMCrypto should propagate values from all open sessions to the Session Usage Table. If any values have changed, increment the generation number, sign, and save the table. During playback, this function will be called approximately once per minute.

Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

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 changed in API version 9.

OEMCrypto_DeactivateUsageEntry

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

Find the entry in the Usage Table with a matching PST. Mark the status of that entry as “inactive”. If it corresponds to an open session, the status of that session will also be marked as “inactive”. Then OEMCrypto will increment Usage Table’s generation number, sign, encrypt, and save the Usage Table.
If no entry in the Usage Table has a matching PST, return the error OEMCrypto_ERROR_INVALID_CONTEXT.
Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

After modifying the table, OEMCrypto will increment the Usage Table's generation number, and then sign, encrypt, and save the Usage Table.

Parameters

[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 - no entry has matching PST.
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 9.

OEMCrypto_ReportUsage

```
OEMCryptoResult OEMCrypto_ReportUsage(OEMCrypto_SESSION session,
                                      const uint8_t *pst,
                                      size_t pst_length,
                                      OEMCrypto_PST_Report *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;
```

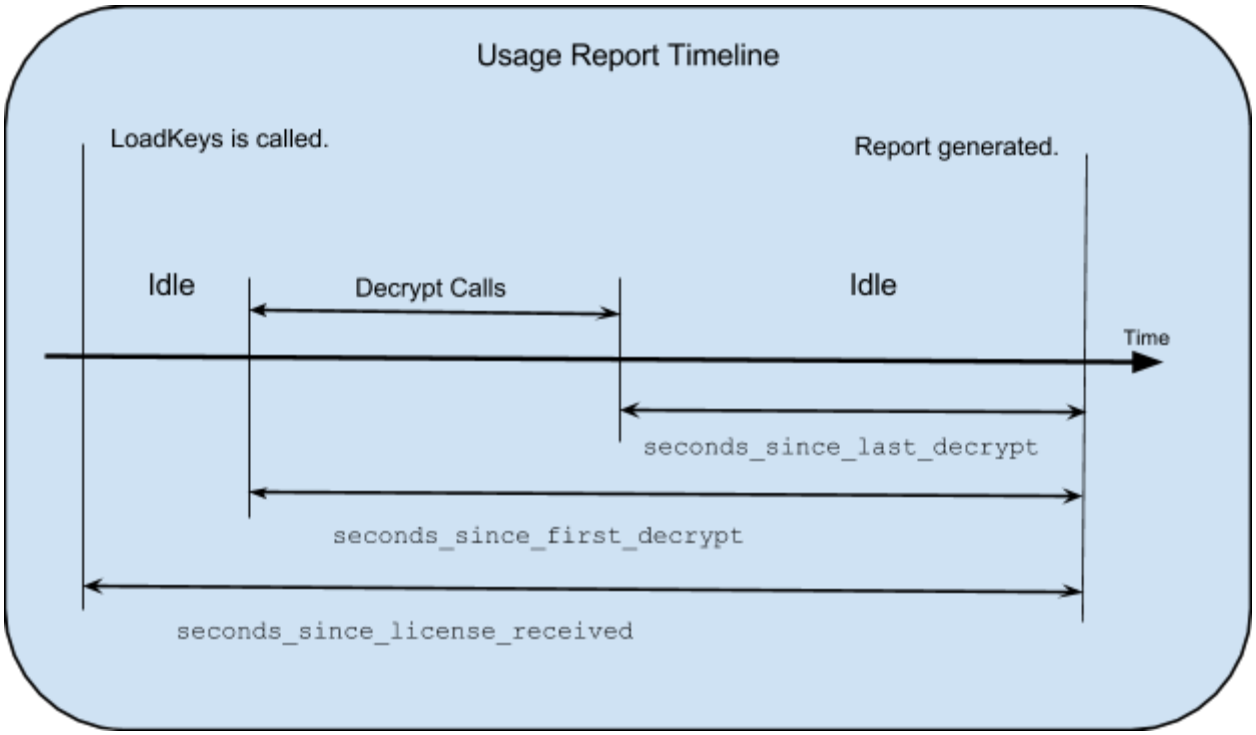
If the buffer_length is not sufficient to hold a report structure, set buffer_length and return OEMCrypto_ERROR_SHORT_BUFFER.

If no entry in the Usage Table has a matching PST, return the error OEMCrypto_ERROR_INVALID_CONTEXT.

OEMCrypto will increment Usage Table's generation number, sign, encrypt, and save the Usage Table. This is done, even though the table has not changed, so that a single rollback cannot

undo a call to DeactivateUsageEntry and still report that license as inactive.

The pst_report is filled out by subtracting the times in the Usage Table 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 -- the keys have been marked inactive.

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.

This function also copies the client_mac_key and server_mac_key from the Usage Table entry to the session. They will be used to verify a signature in OEMCrypto_DeleteUsageEntry below. This session will be associated with the entry in the Usage Table.

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 signature.
OEMCrypto_ERROR_INVALID_SESSION no open session with that id.
OEMCrypto_ERROR_INVALID_CONTEXT - no entry has matching PST.
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 9.

OEMCrypto_DeleteUsageEntry

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

This function verifies the signature of the given message using the session's mac_key[server] and the algorithm HMAC-SHA256, and then deletes an entry from the session usage table. The session should already be associated with the given entry, from a previous call to OEMCrypto_ReportUsage.

After performing all verification listed below, and deleting the entry from the Usage Table, OEMCrypto will increment the Usage Table's generation number, and then sign, encrypt, and save the Usage Table.

The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison).

Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Verification

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

1. The pointer pst is not null, and points inside the message. If not, return OEMCrypto_ERROR_UNKNOWN_FAILURE.
2. The signature of the message shall be computed, and the API shall verify the computed signature matches the signature passed in. The signature will be computed using HMAC-SHA256 and the mac_key_server. If they do not match, return OEMCrypto_ERROR_SIGNATURE_FAILURE.
3. If the session is not associated with an entry in the Usage Table, return OEMCrypto_ERROR_UNKNOWN_FAILURE.
4. If the pst passed in as a parameter does not match that in the Usage Table, return OEMCrypto_ERROR_UNKNOWN_FAILURE.

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.
[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.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_INVALID_SESSION no open session with that id.
OEMCrypto_ERROR_SIGNATURE_FAILURE
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 9.

OEMCrypto_ForceDeleteUsageEntry

```
OEMCryptoResult OEMCrypto_ForceDeleteUsageEntry(const uint8_t* pst,
                                                size_t pst_length);
```

This function deletes an entry from the session usage table. This will be used for stale entries without a signed request from the server.

After performing all verification listed below, and deleting the entry from the Usage Table, OEMCrypto will increment the Usage Table's generation number, and then sign, encrypt, and save the Usage Table.

Devices that do not implement a Session Usage Table may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

Verification

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

1. The pointer pst is not null. If not, return OEMCrypto_ERROR_UNKNOWN_FAILURE.

Parameters

[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 changed in API version 10.

OEMCrypto_DeleteUsageTable

```
OEMCryptoResult OEMCrypto_DeleteUsageTable()
```

This is called when the CDM system believes there are major problems or resource issues. The entire table should be cleaned and a new table should be created.

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 changed in API version 9.

Error Codes

This is a list of error codes and their uses.

OEMCrypto_SUCCESS	No error.
OEMCrypto_ERROR_INIT_FAILED	Initialization failed.
OEMCrypto_ERROR_TERMINATE_FAILED	Termination failed.
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.
OEMCrypto_ERROR_NO_DEVICE_KEY	Indicates the keybox does not have a device key. (deprecated)
OEMCrypto_ERROR_KEYBOX_INVALID	Indicates Widevine keybox is invalid.
OEMCrypto_ERROR_NO_KEYDATA	Indicates Widevine keybox is invalid or does not have any key data.
OEMCrypto_ERROR_DECRYPT_FAILED	Indicates DecryptCENC or Generic Decrypt failed.
OEMCrypto_ERROR_WRITE_KEYBOX	Keybox could not be installed to

	secure memory.
OEMCrypto_ERROR_WRAP_KEYBOX	OEMCrypto_WrapKeybox failed to encrypt keybox.
OEMCrypto_ERROR_BAD_MAGIC	Keybox has bad magic field.
OEMCrypto_ERROR_BAD_CRC	Keybox has bad CRC field.
OEMCrypto_ERROR_NO_DEVICEID	GetDeviceID failed.
OEMCrypto_ERROR_RNG_FAILED	GetRandom failed.
OEMCrypto_ERROR_ RNG_NOT_SUPPORTED	GetRandom is not implemented.
OEMCrypto_ERROR_OPEN_SESSION_FAILED	OpenSession failed, but not with a resource issue.
OEMCrypto_ERROR_CLOSE_SESSION_FAILED	CloseSession failed on valid session.
OEMCrypto_ERROR_INVALID_SESSION	Specified session is not open or is in a corrupted state.
OEMCrypto_ERROR_NOT_IMPLEMENTED	WrapKeybox is not implemented.
OEMCrypto_ERROR_NO_CONTENT_KEY	SelectKey failed to find the specified Key ID.
OEMCrypto_ERROR_CONTROL_INVALID	The control block of the specified key is not valid. Returned by SelectKey.
OEMCrypto_ERROR_INVALID_CONTEXT	Context for signing or verification is not valid.
OEMCrypto_ERROR_SIGNATURE_FAILURE	Could not sign specified buffer.
OEMCrypto_ERROR_ DEVICE_NOT_RSA_PROVISIONED	Session does not have an RSA key installed.
OEMCrypto_ERROR_INVALID_RSA_KEY	RSA key is not valid in RewrapDeviceRSAKey or LoadDeviceRSAKey
OEMCrypto_ERROR_INVALID_NONCE	Nonce in server response does not match any in table.
OEMCrypto_ERROR_KEY_EXPIRED	The current key's duration has expired, but is otherwise valid.

OEMCrypto_ERROR_TOO_MANY_SESSIONS	Not enough resources to open a new session.
OEMCrypto_ERROR_TOO_MANY_KEYS	Not enough resources to LoadKeys.
OEMCrypto_ERROR_ INSUFFICIENT_RESOURCES	Other resource issues, such as buffers needed for decryption.
OEMCrypto_ERROR_INSUFFICIENT_HDCP	An attached display does not support the minimum HDCP version.
OEMCrypto_ERROR_UNKNOWN_FAILURE	Any other error.

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