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

Version 3

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

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.

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

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

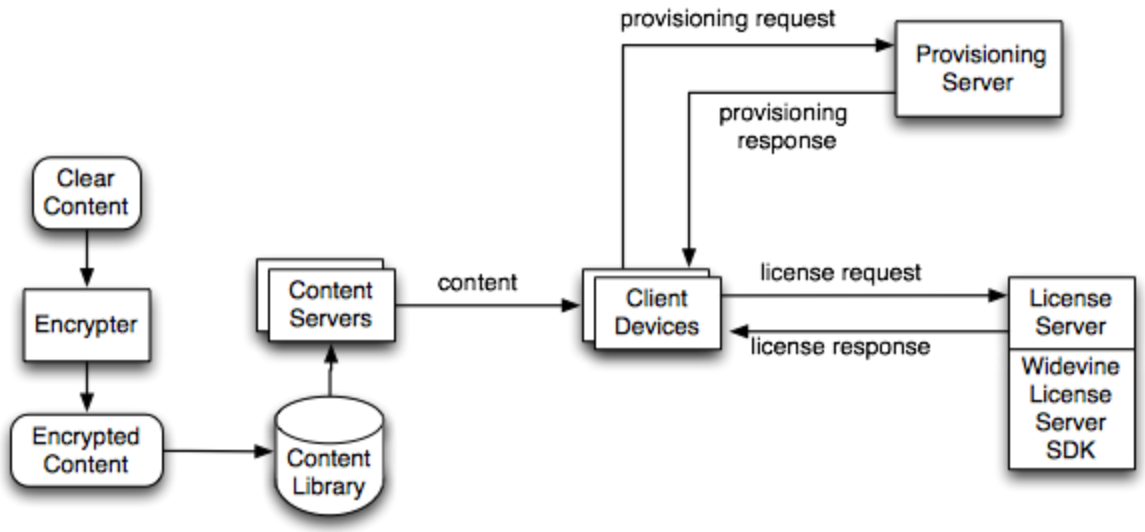
This document describes the security APIs used in Widevine content protection for playing content compatible with the *Dynamic Adaptive Streaming over HTTP* specification, ISO/IEC 23009-1 (MPEG DASH) using the DRM methods specified in ISO/IEC 23001-7: Common

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	Yes	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.

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

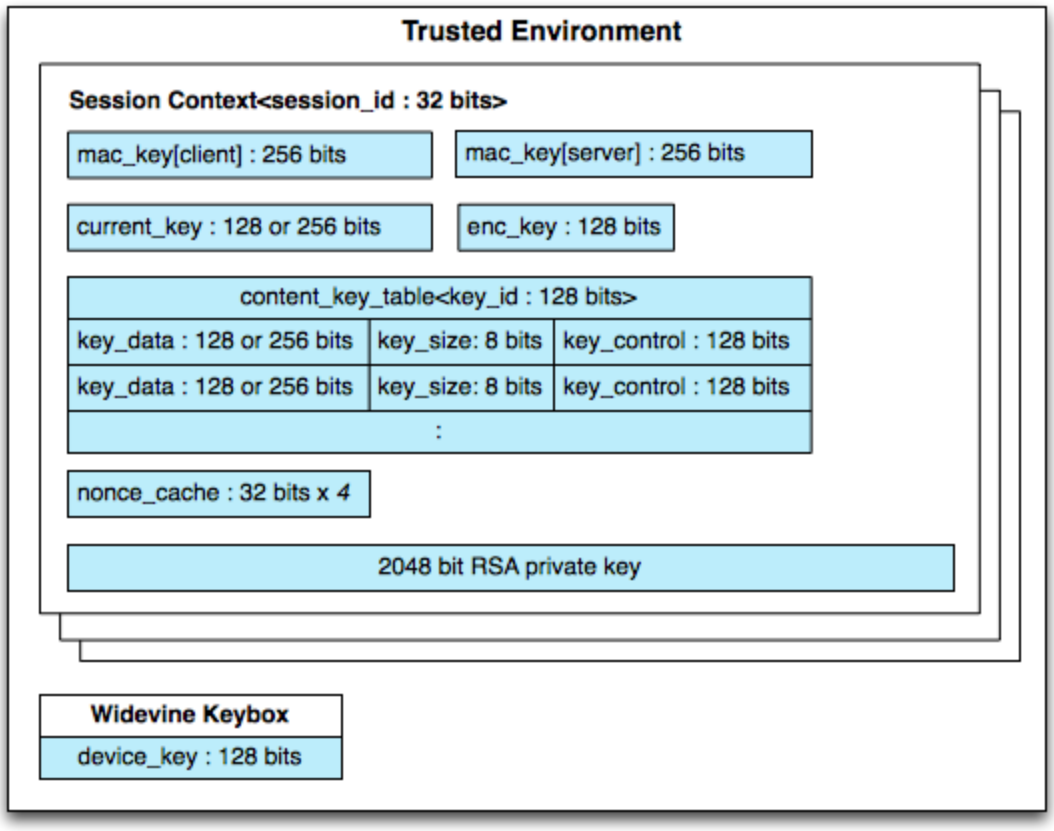
One or more crypto sessions will be created to support A/V playback. Each session has context, or state, that must be maintained in secure memory. The required session state is summarized in the diagram below.

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.

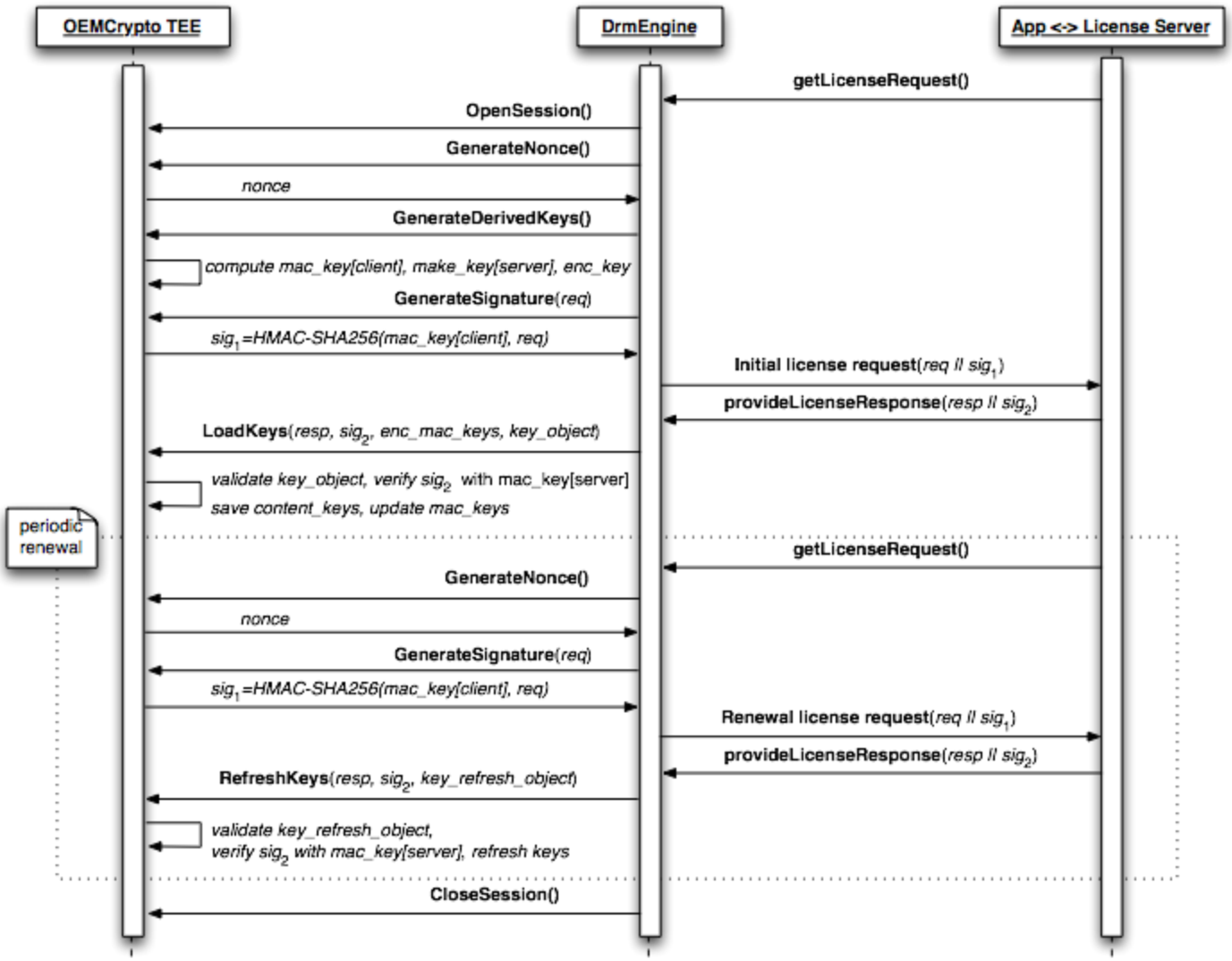


When the session is closed via OEMCrypto_CloseSession(), all of the Session Context resources must be explicitly cleared and then released.

License Signing and Verification

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.

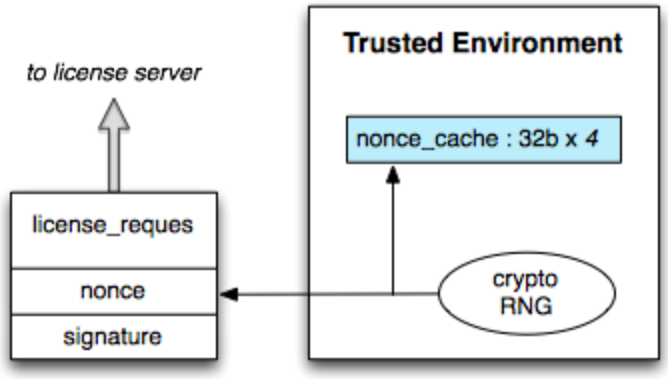


The app calls getLicenseRequest() to obtain an opaque license request message to send to the license server. The OEMCrypto calls OpenSession, GenerateDerivedKeys, GenerateNonce and GenerateSignature are used in the construction and signing of the request message. Once a license server response has been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

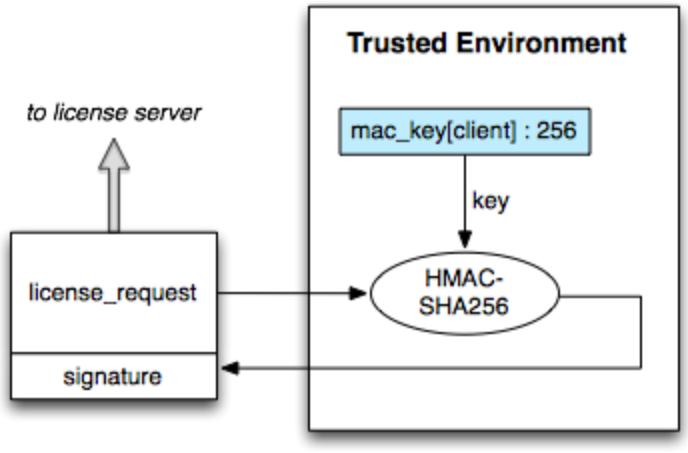
For the license initial and renewal *responses*, the OEMCrypto implementation must verify that the license response and its signature match.

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.

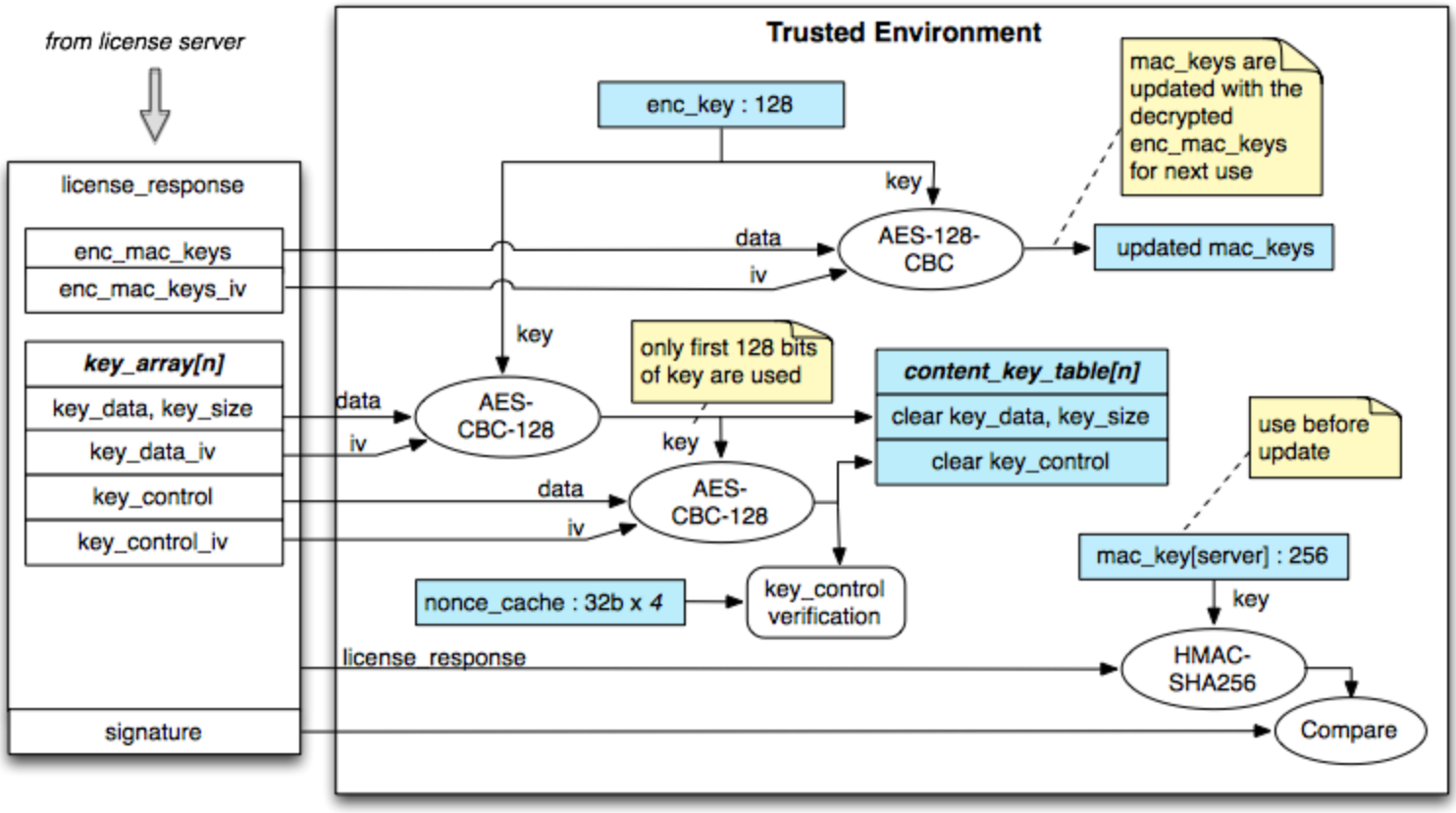
Note: When verifying the signature, the string comparison between the input signature and the recomputed signature should be a constant-time operation, to avoid leaking timing info.

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

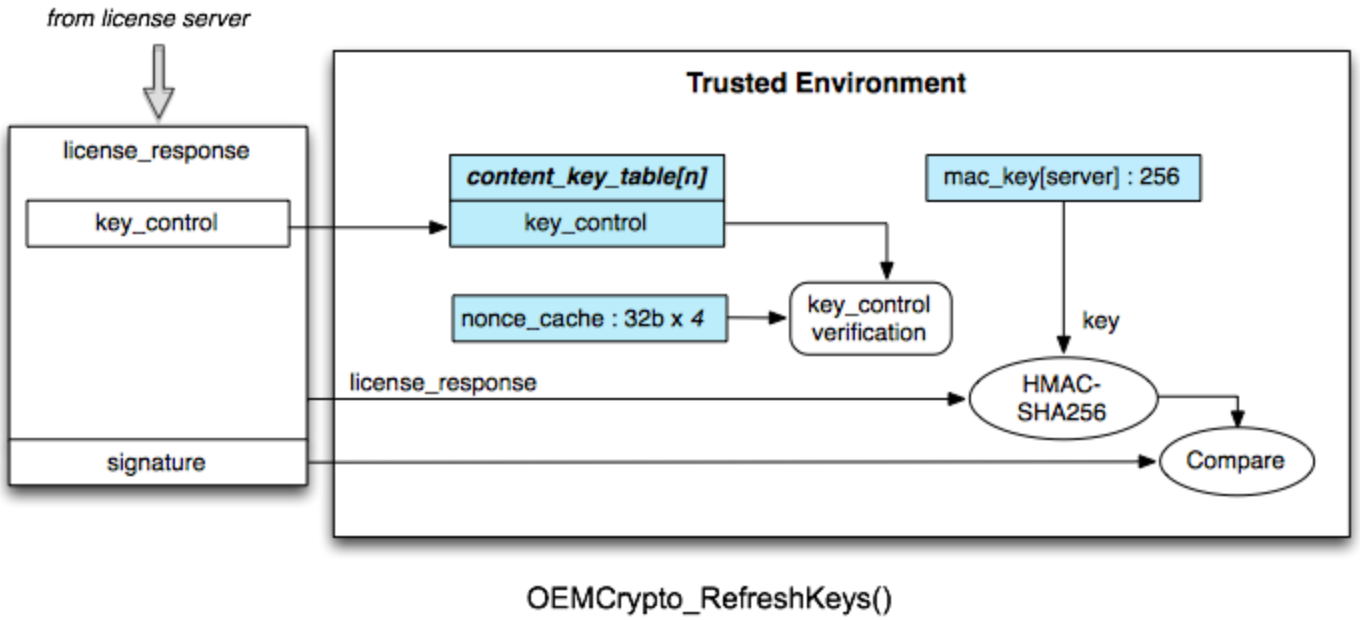


OEMCrypto_GenerateSignature()

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: enc_key + mac_keys

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.

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]

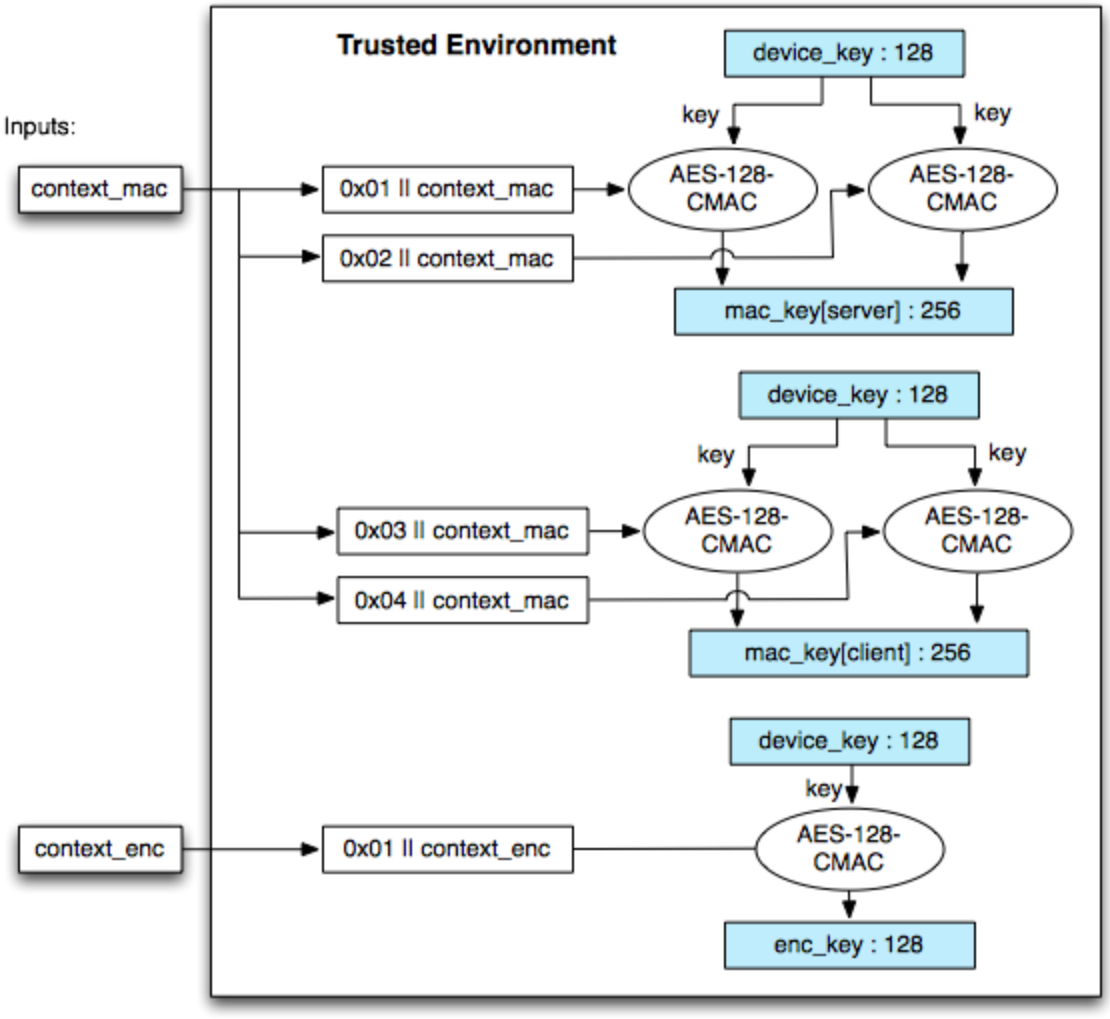
   := AES-128-CMAC(device_key, 0x01 || context_mac) ||
   AES-128-CMAC(device_key, 0x02 || context_mac) ||
   AES-128-CMAC(device_key, 0x03 || context_mac) ||
   AES-128-CMAC(device_key, 0x04 || context_mac)

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’.	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	*bit 30*	*bit 29*
Observe_DataPathType 0=Ignore 1=Observe	Observe_HDCP 0=Ignore 1=Observe	Observe_CGMS 0=Ignore 1=Observe

bits 28..9	*bit 8*	*bit 7*	*bit 6*	*bits 5*
Reserved set to 0	Allow_Encrypt 0=Normal 1=May be used to encrypt generic data.	Allow_Decrypt 0=Normal 1=May be used to decrypt generic data.	Allow_Sign 0=Normal 1=May be used to sign generic data.	Allow_Verify 0=Normal 1=May be used to verify signature of generic data.

bit 4	*bit 3*	*bit 2*	*bits 1..0*
Data_Path_Type 0=Normal 1=Secure only	Nonce_Enable 0=Ignore Nonce 1=Verify Nonce	HDCP 0=HDCP not required 1=HDCP required	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’. If not, return OEMCrypto_ERROR_CONTROL_INVALID.
5. Apply the control fields:
    a. NonceEnable -- if 1, verify the nonce. See the next section for details on verifying the nonce. 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 be 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.
6. 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.
7. 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.
8. Duration field -- on each DecryptCTR 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.
9. Make the decrypted content key from step 2 available for decryption of the media stream by DecryptCTR.
10. Return OEMCrypto_SUCCESS.

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:

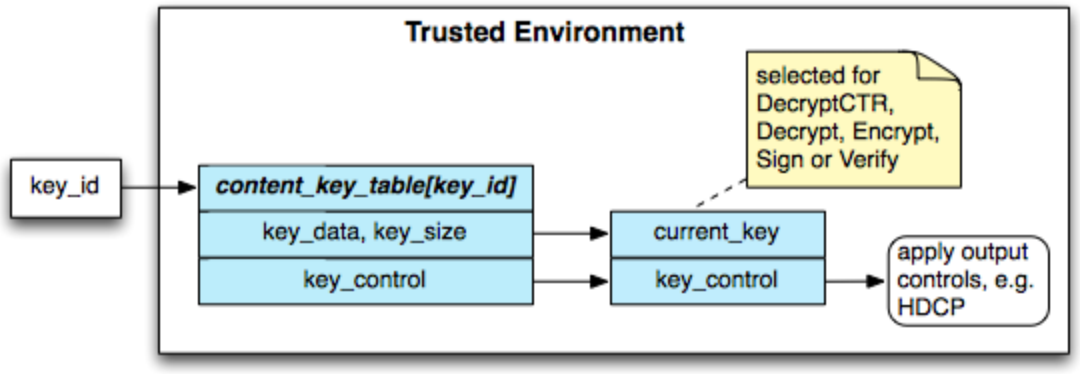
1. Nonce generation: a new nonce is generated by the OEMCrypto implementation at the

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.

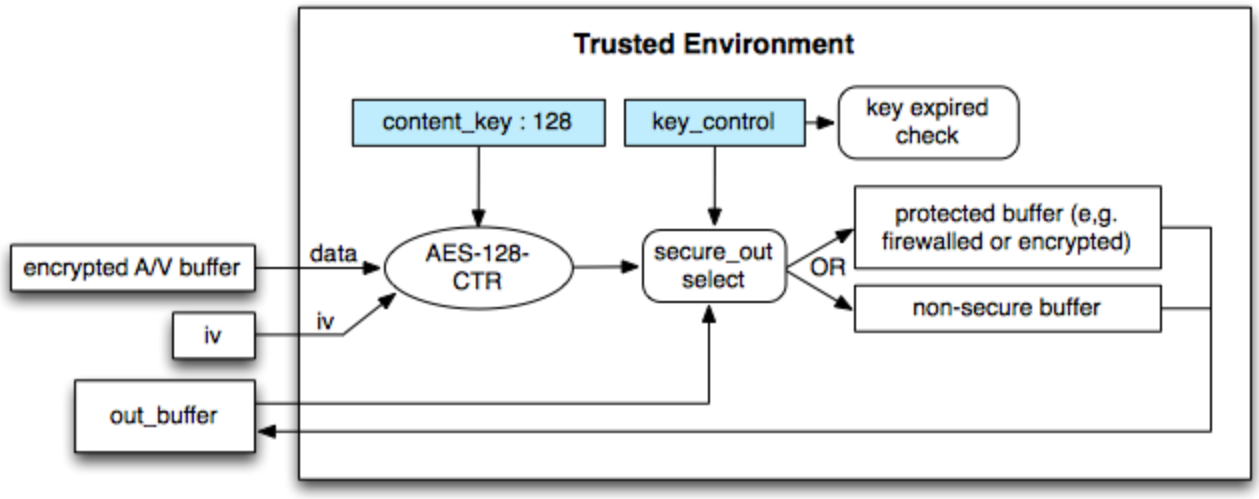
Content Decryption

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



OEMCrypto_SelectKey

Once the content_key is loaded, OEMCrypto_DecryptCTR 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 with random IV.



OEMCrypto_DecryptCTR()

RSA Certificate Provisioning and License Requests

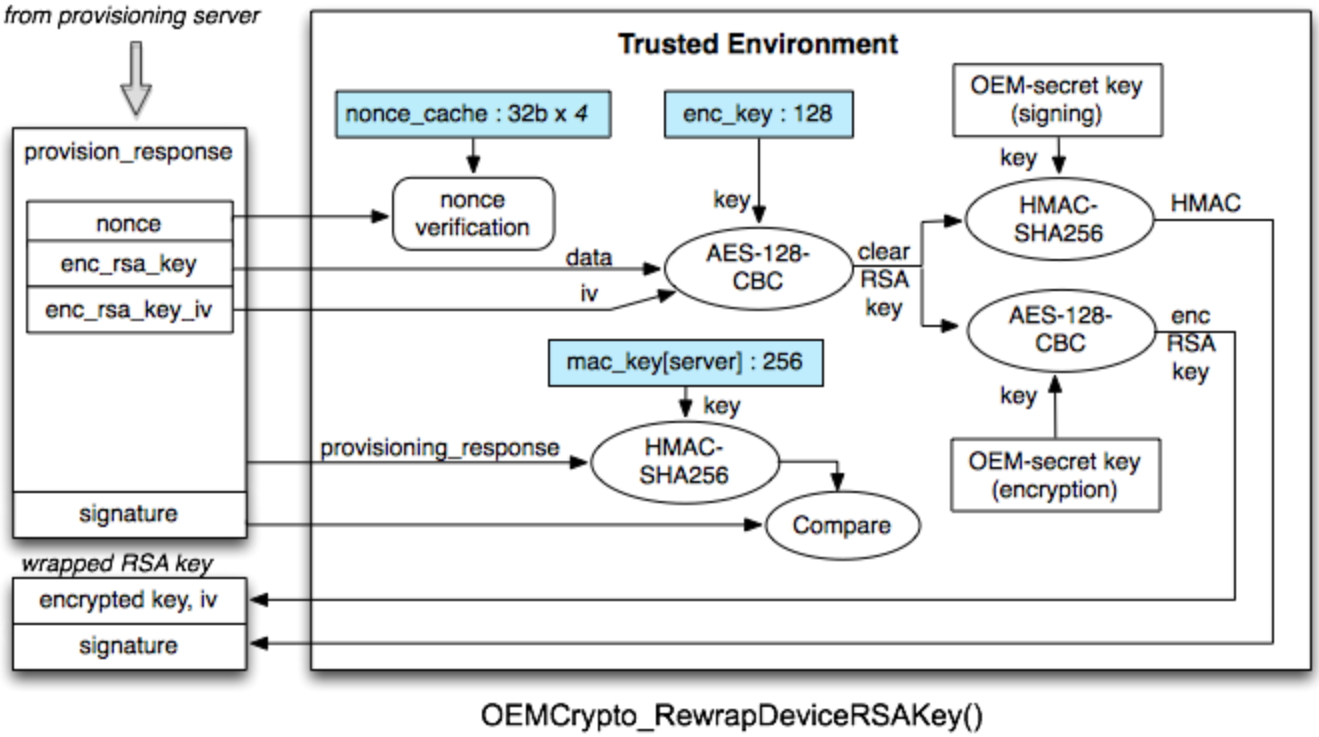
This section describes new features added in March, 2013, producing a V2.1 revision to the license protocol. The basic flow described in the previous sections can be modified to allow an application to use an RSA signed certificate for license requests instead of the Widevine Keybox. This allows the 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.

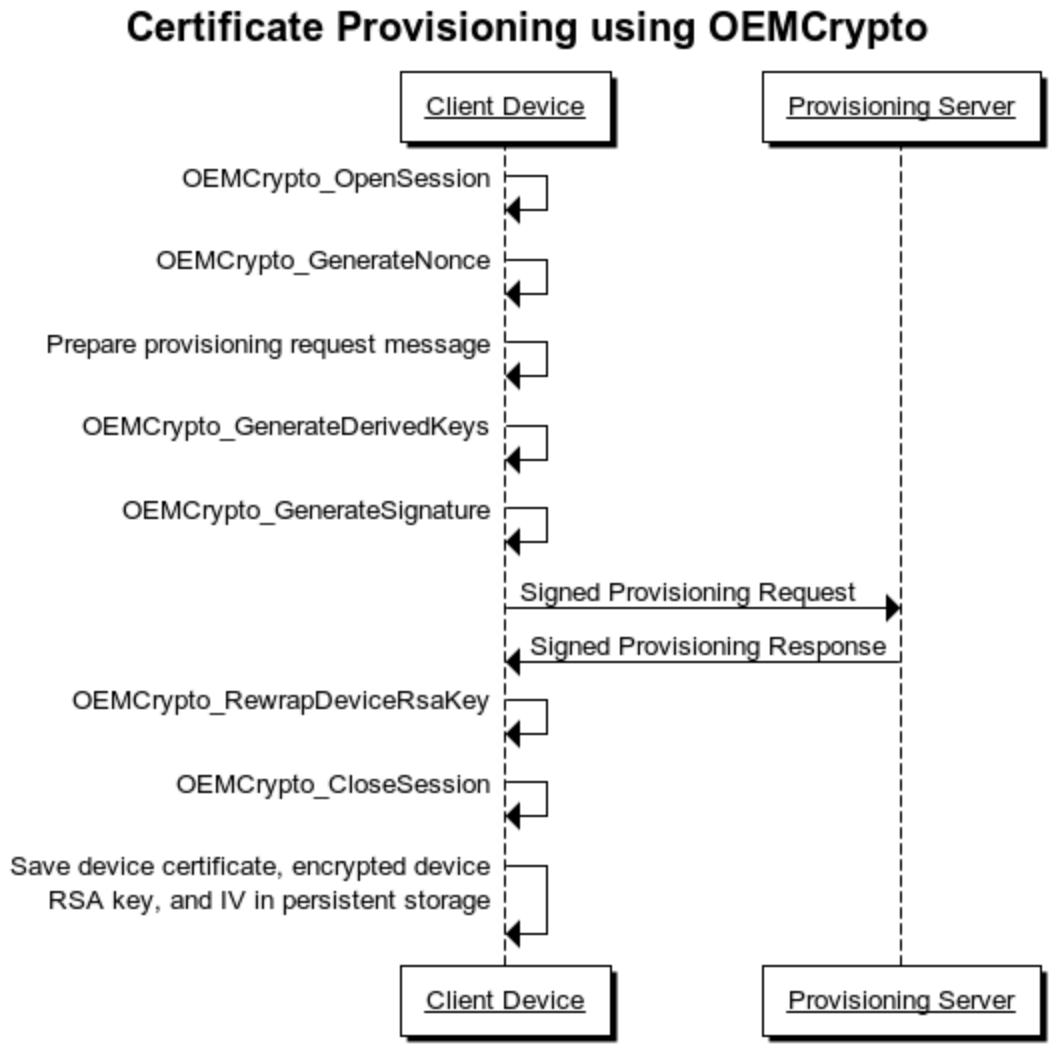
RSA Certificate Provisioning

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()

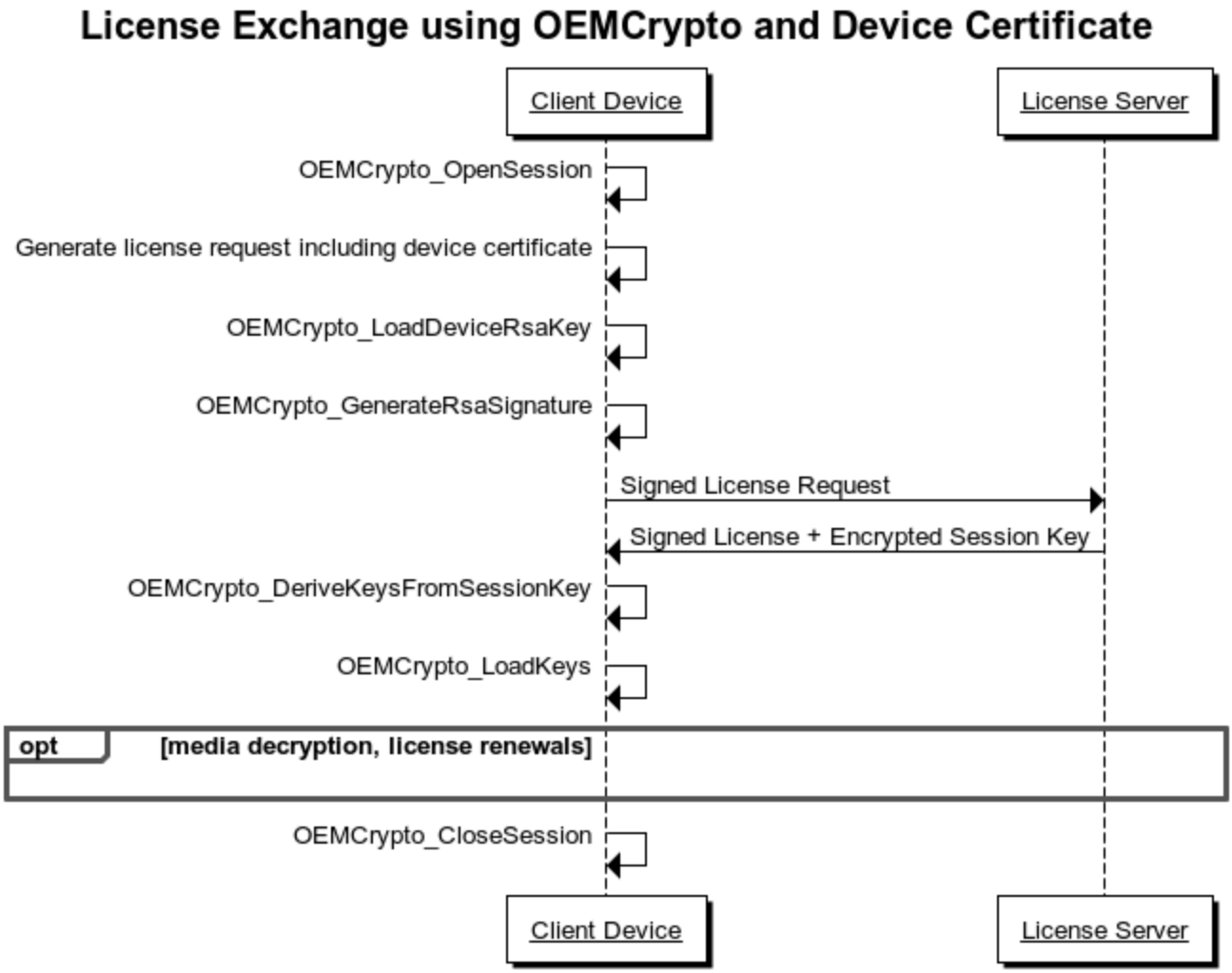
In the function OEMCrypto_RewrapDeviceRSAKey(), the device uses the encryption key, generated previously in OEMCrypto_GenerateDerivedKeys(), to decrypt the RSA private key and store it in secure memory. The device verifies the provisioning response message in much the same way it does in OEMCrypto_LoadKeys(). 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.



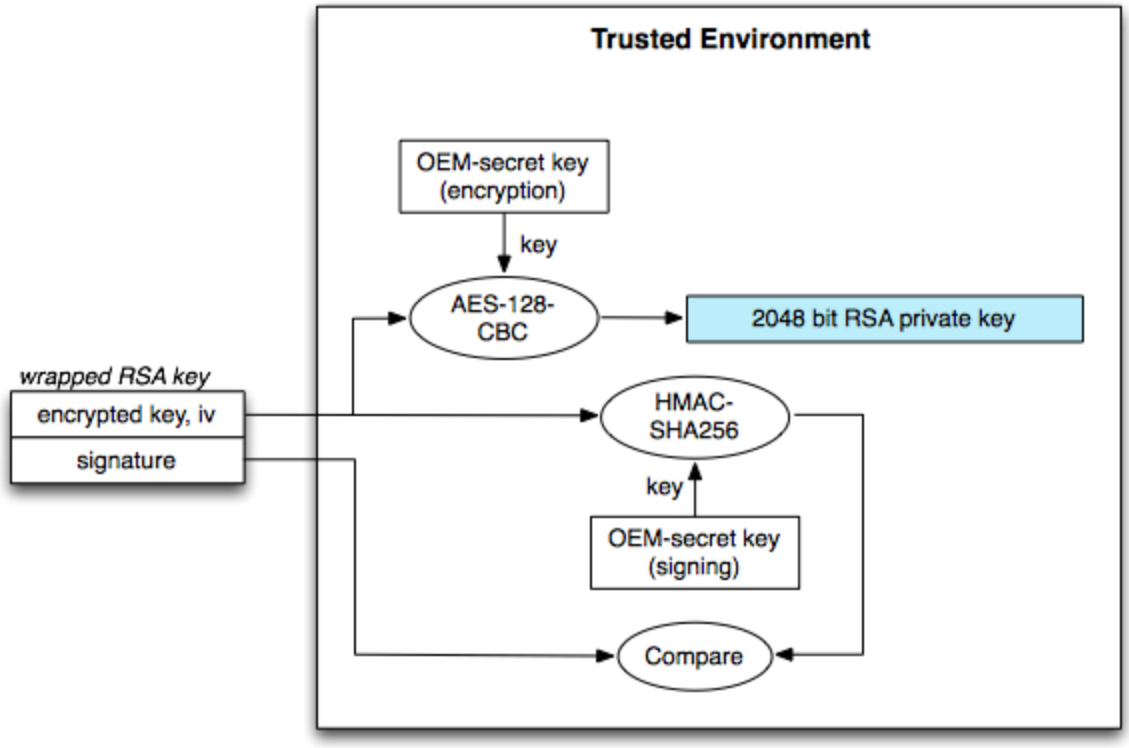
License Request Signed by RSA Certificate

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:

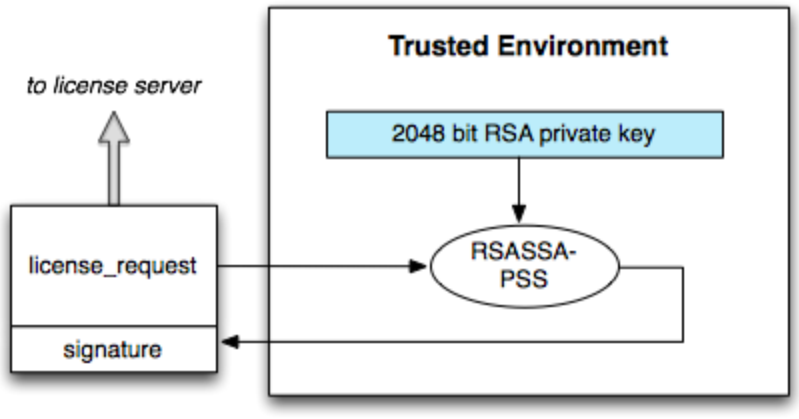


The first function is OEMCrypto_LoadDeviceRSAKey(), is passed a wrapped RSA key pair. It unwraps the key pair and stores it in secure memory.



OEMCrypto_LoadDeviceRSAKey()

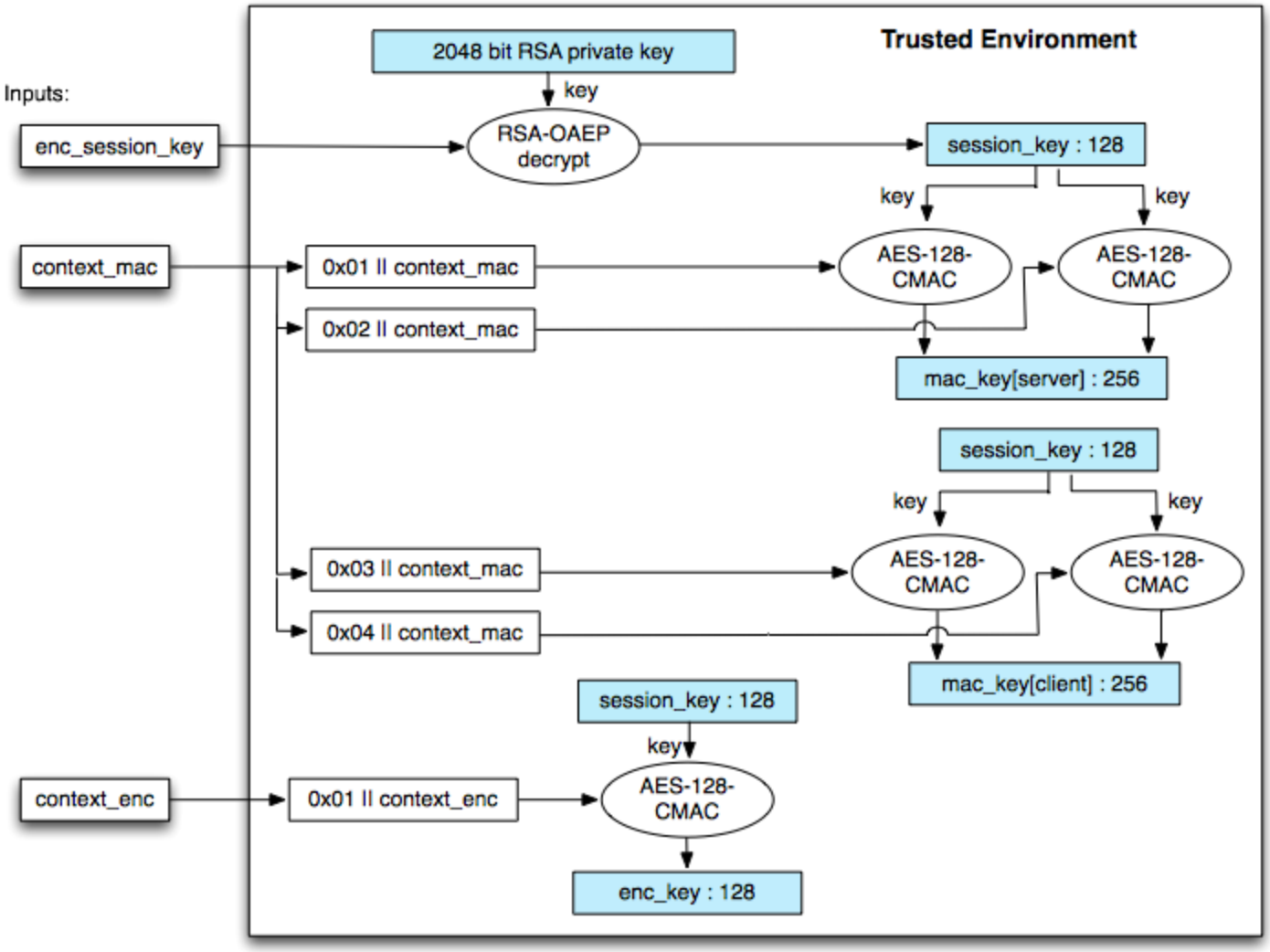
The second function, OEMCrypto_GenerateRSASignature(), signs a message using the device RSA private key.



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()

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
- Video Path API
- Provisioning API

- Keybox Access

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().

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.

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 mode and prevent clear keys from being exposed to the CPU.

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_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.

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.

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 AES CTR

mode.

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().

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_UNKNOWN_FAILURE

OEMCrypto_ERROR_INVALID_CONTEXT

Threading

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

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.

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

Threading

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

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 AES CTR mode.

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.

[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_SHORT_BUFFER if signature buffer is not large enough to hold buffer.

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

Threading

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

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);

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

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 corresponding content key (AES-128-CBC) and the IV given in the KeyObject.

After all keys have been decrypted and validated, the new mac_keys are decrypted with the current encrypt_key and the offered IV. The new mac_keys replaces the current mac_keys for future calls to OEMCrypto_RefreshKeys(). The first 256 bits of the mac_keys become the

mac_key[server] and the following 256 bits of the mac_keys become the mac_key[client].

The mac_key and encrypt_key were generated and stored by the previous call to OEMCrypto_GenerateDerivedKeys(). 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_DecryptCTR().

NOTE: OEMCrypto_GenerateDerivedKeys() must be called first to establish the mac_key and encrypt_key.

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 API shall verify that the two pointers `enc_mac_key_iv` and `enc_mac_keys` point to locations in the message. I.e. `(message <= p && p < message+message_length)` for `p` in each of `enc_mac_key_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 < 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 the nonce generated by the current nonce. 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: 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.

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

Threading

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

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 key other 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_DecryptCTR().

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 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, 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 < 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_UNKNOWN_FAILURE
OEMCrypto_ERROR_INVALID_CONTEXT
OEMCrypto_ERROR_SIGNATURE_FAILURE
OEMCrypto_ERROR_INVALID_NONCE

Threading

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

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. This section covers the latter implementation. Additional APIs will be provided if the former option needs to be supported. For encrypted HTML5 streams the device needs to provide support for AES-128-CTR-CTS mode as described here.

The following table shows the APIs required for decryption:

OEMCrypto_SelectKey
OEMCrypto_DecryptCTR

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

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_DecryptCTR()) 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) to decrypt buffers passed in via OEMCrypto_DecryptCTR(). 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

Threading

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

OEMCrypto_DecryptCTR

```
OEMCryptoResult
OEMCrypto_DecryptCTR(OEMCrypto_SESSION session,
                         const uint8_t *data_addr,
                         size_t data_length,
                         bool is_encrypted,
                         const uint8_t *iv,
                         size_t block_offset,
                         const OEMCrypto_DestBufferDesc* out_buffer,
                         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
```

Decrypts (AES-128-CTR) 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. 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:

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. DecryptCTR begins storing data out_buffer->secure.offset bytes after the beginning of the secure buffer.

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_DECRYPT_FAILED.
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 output using HDCP only. If not, return OEMCrypto_ERROR_DECRYPT_FAILED.

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] 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_UNKNOWN_FAILURE

OEMCrypto_ERROR_INVALID_CONTEXT

OEMCrypto_ERROR_DECRYPT_FAILED

Threading

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

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.

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
OEMCrypto_InstallKeybox

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().

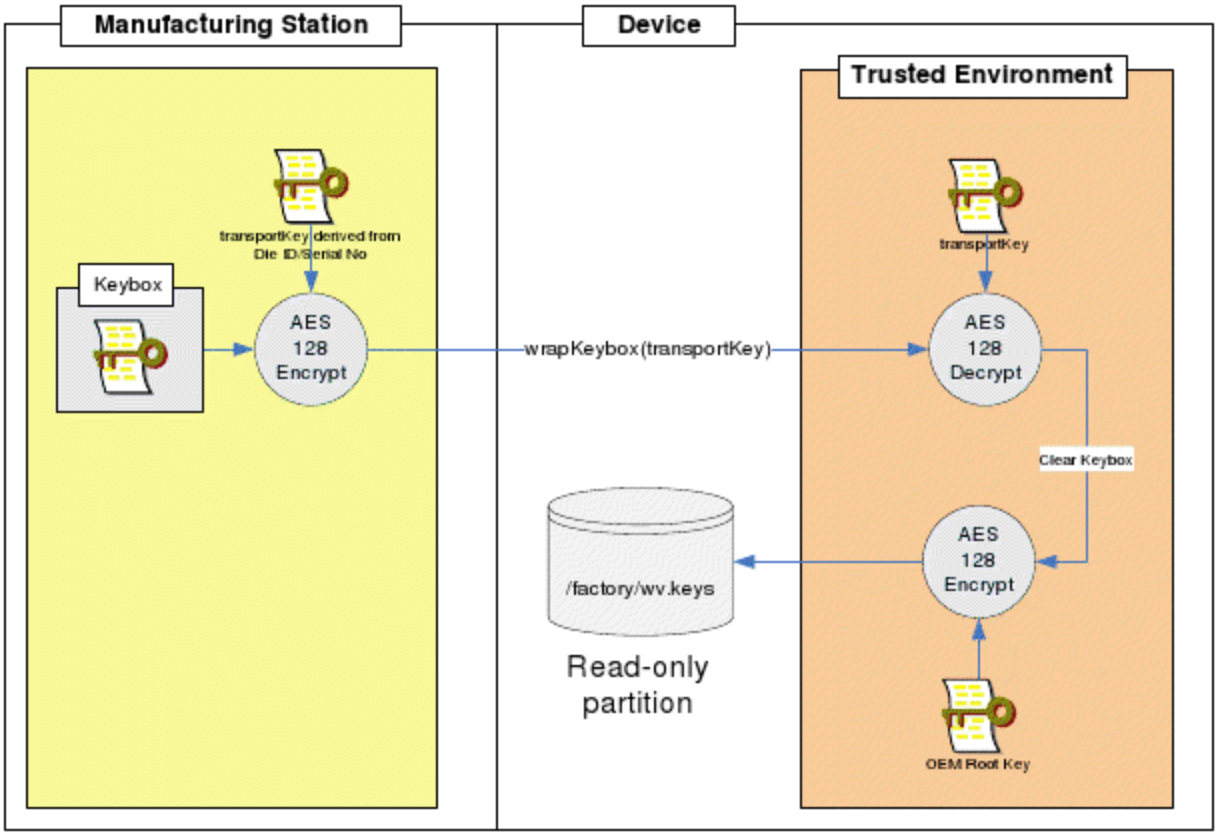


Figure 10. OEMCrypto_WrapKeybox Operation

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_NOT_IMPLEMENTED

Threading

This function is not called simultaneously with any other functions

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_WRITE_KEYBOX failed to encrypt and store Keybox

Threading

This function is not called simultaneously with any other functions

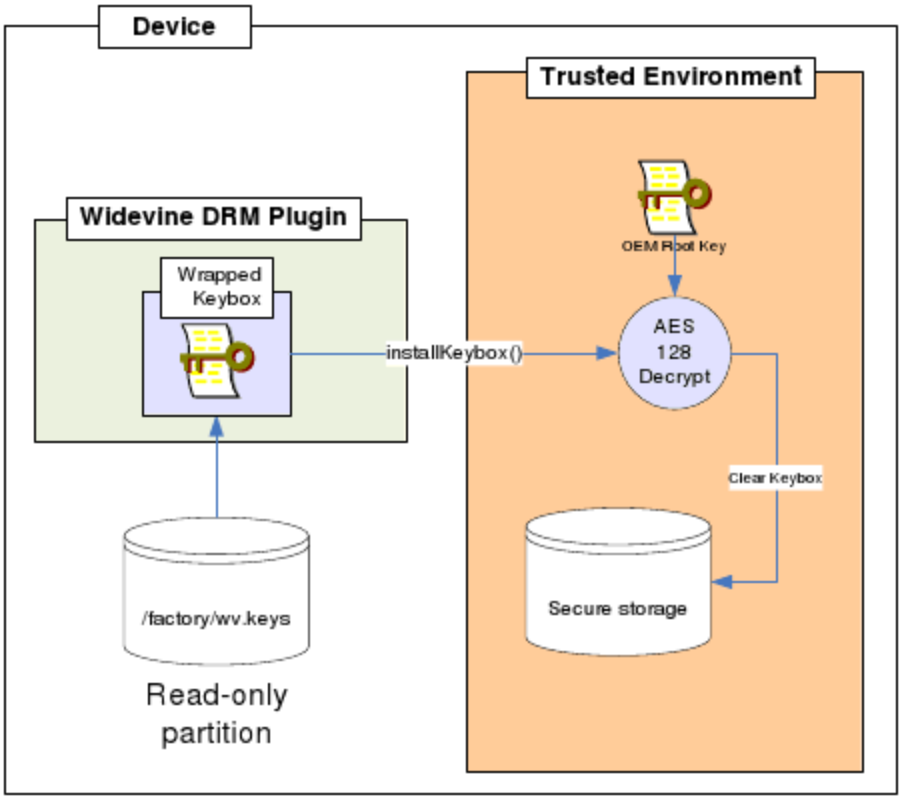


Figure 11 - Install keybox Operation

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_SecurityLevel

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.

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.

OEMCrypto_GetKeyData

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

Decrypt and 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

Threading

This function may be called simultaneously with any session functions.

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.

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.

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.

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.

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_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.

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_BUFFER_TOO_SMALL` 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.
1. Validate the decrypted RSA device key by verifying that it can be loaded by the RSA implementation.
2. Generate a random initialization vector and store it in `wrapped_rsa_key_iv.`
3. 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.
6. 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: length of the encrypted RSA key, in bytes.

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_INVALID_RSA_KEY
OEMCrypto_ERROR_SIGNATURE_FAILURE
OEMCrypto_ERROR_INVALID_NONCE
OEMCrypto_ERROR_BUFFER_TO_SMALL

Threading

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

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.

Verification

The API should verify that the signature in the wrapped RSA key, and the decrypted RSA key is valid.

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_UNKNOWN_FAILURE
OEMCrypto_ERROR_INVALID_RSA_KEY

Threading

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

OEMCrypto_GenerateRSASignature

```
OEMCryptoResult OEMCrypto_GenerateRSASignature(
                      OEMCrypto_SESSION session,
```

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

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.

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

Returns

OEMCrypto_SUCCESS success
OEMCrypto_ERROR_BUFFER_TOO_SMALL if the signature buffer is too small.
OEMCrypto_ERROR_CLOSE_SESSION_FAILED illegal/unrecognized handle or the security engine is not properly initialized.

Threading

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

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 AES CTR mode.

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 two 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.

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

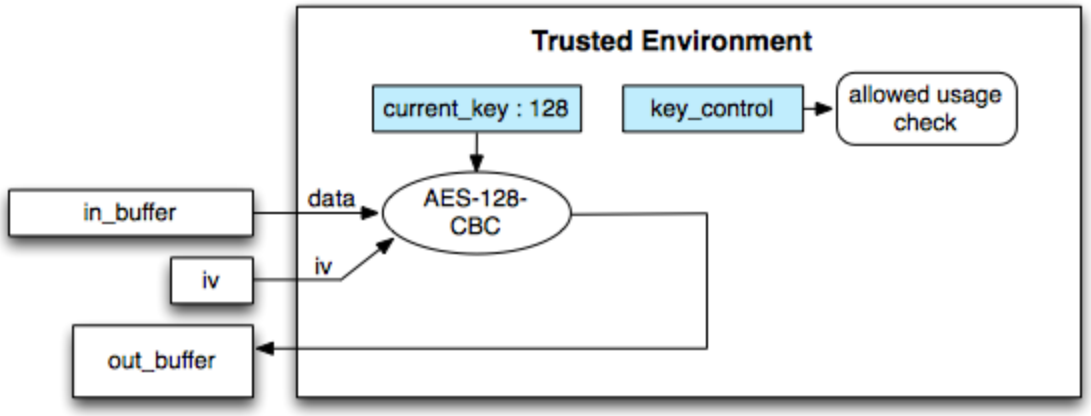
OEMCrypto_SUCCESS success
OEMCrypto_ERROR_DEVICE_NOT_RSA_PROVISIONED
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_UNKNOWN_FAILURE
OEMCrypto_ERROR_INVALID_CONTEXT

Threading

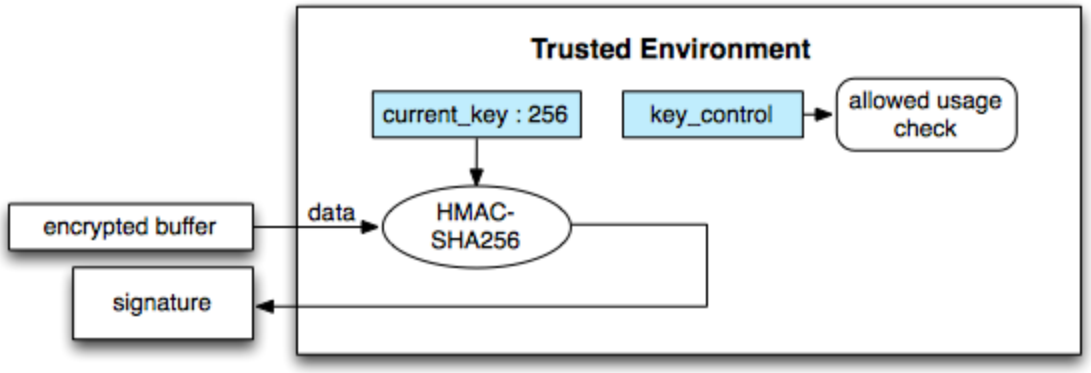
This function may be called simultaneously with functions on other sessions, but not with other functions on this session.

Generalized Modular DRM

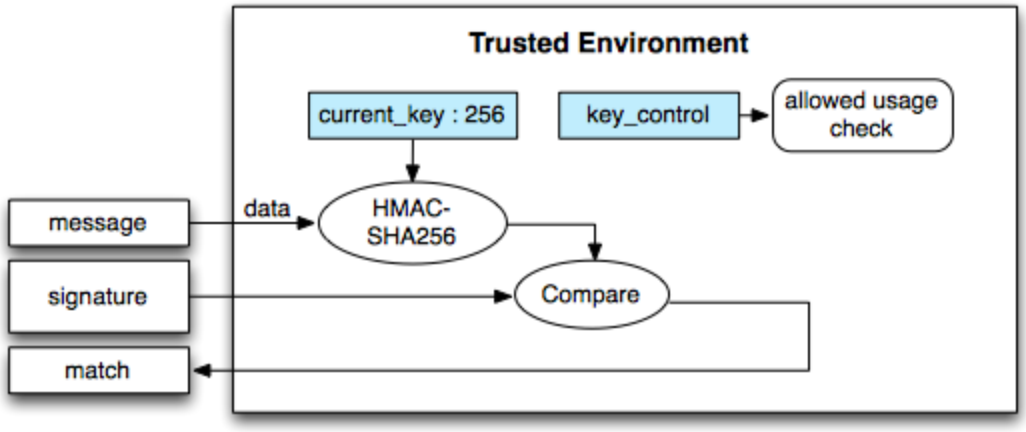
This section describes a generalization of Modular DRM and the Android MediaDrm APIs to provide the ability for operators to securely deliver session keys from their server to a client device, based on the factory-installed root of trust, and provide the ability to do encrypt, decrypt, sign and verify with the session key on arbitrary user data.



OEMCrypto_Generic_Decrypt(), OEMCrypto_Generic_Encrypt()



OEMCrypto_Generic_Sign()



OEMCrypto_Generic_Verify()

The following table shows the APIs required for Generalized Modular DRM:

OEMCrypto_Generic_Encrypt
OEMCrypto_Generic_Decrypt
OEMCrypto_Generic_Sign
OEMCrypto_Generic_Verify

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.

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.

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_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

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

functions on this session.

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.

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 the HDCP bit set, then return OEMCrypto_ERROR_DECRYPT_FAILED.

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_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

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

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.

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.

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

Returns

OEMCrypto_SUCCESS success
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_UNKNOWN_FAILURE

Threading

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

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 verfies the signature of a generic buffer of data using the current key.

Verification

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

6. The control bit for the current key shall have the Allow_Verify set.
7. 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.
8. The signature verification shall use a constant-time algorithm (a signature mismatch will always take the same time as a successful comparison).

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_SIGNATURE_FAILURE
OEMCrypto_ERROR_NO_DEVICE_KEY
OEMCrypto_ERROR_INVALID_SESSION
OEMCrypto_ERROR_UNKNOWN_FAILURE

Threading

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

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
