WIDEVINE®

WV Modular DRM Version 12 Delta

Changes from Version 11 to 12

November 28, 2016

Table of Contents

References

DASH - 23001-7 ISO BMFF Common Encryption, 3rd edition.

DASH - 14496-12 ISO BMFF, 5th edition.

W3C Encrypted Media Extensions (EME)

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

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

Audience

This document is intended for SOC and OEM device manufacturers to upgrade an integration with Widevine content protection using Common Encryption (CENC) on consumer devices.   In particular, if you already have a working OEMCrypto v11 library, and want to upgrade to OEMCrypto v12, then this document is for you.   If you are starting from scratch, you should read WV Modular DRM Security Integration Guide for Common Encryption (CENC).

Overview

There are several new features required for OEMCrypto version 12. The following sections discuss the main new features and give some idea why the new feature is being added. You should refer to WV Modular DRM Security Integration Guide for Common Encryption (CENC) for the full documentation of the API -- this document only discusses changes to the API. In this document, when we say “OEMCrypto shall …” we mean that “an implementation of the OEMCrypto library shall …”.

Definitions

CENC - Common Encryption

DASH - Dynamic Adaptive Streaming over HTTP

CDM - Content Decryption Module -- this is the software that calls the OEMCrypto library and implements CENC.

PST - Provider Session Token - the string used to track usage information and off line licenses.

API Version Number

```
uint32_t OEMCrypto_APIVersion();
```

This function should now return 12. If it returns less than 12, then the calling code will assume that OEMCrypto does not support the new v12 features. Depending on the platform, the library may be run in backwards compatibility mode, or it may fail. See the supplemental document for your platform to see which version of OEMCrypto is required.

Buffer Size Clarification

Previously, this spec was vague about some buffer sizes. In version 12 we specify a maximum buffer size for some calls, and the unit tests now verify that OEMCrypto does not fail when buffers have that size. If any of the functions below are given a buffer that is too large, it should return the new error code OEMCrypto_ERROR_BUFFER_TOO_LARGE. An implementation of OEMCrypto is not required to limit the buffer size -- each function below may handle larger buffers than the minimum.

Buffers used in license requests are used to derive session keys and will be signed. We now require OEMCrypto to handle message and context buffers as large as 8 KiB for the following functions:

```
OEMCrypto_GenerateDerivedKeys
OEMCrypto_DeriveKeysFromSessionKey
OEMCrypto_GenerateSignature
OEMCrypto_LoadKeys
OEMCrypto_RefreshKeys
OEMCrypto_RewrapDeviceRSAKey
OEMCrypto_GenerateRSASignature
```

OEMCrypto now must handle a PST as large as 255 bytes in the following functions. OEMCrypto may store a hash of the PST as a key in the usage table, rather than storing the actual PST in the table.

```
OEMCrypto_DeactivateUsageEntry
OEMCrypto_ReportUsage
OEMCrypto_DeleteUsageEntry
OEMCrypto_ForceDeleteUsageEntry
```

Generic encryption must handle buffers as large as 100 KiB.

```
OEMCrypto_Generic_Encrypt
OEMCrypto_Generic_Decrypt
OEMCrypto_Generic_Sign
OEMCrypto_Generic_Verify
```

The random number generator must now handle requests to generate up to 32 bytes at a time:

```
OEMCrypto_GetRandom
```

The entire data buffer sizes for content decryption can be as large as a frame of video. If a frame is more than 100 KiB, the layer above OEMCrypto will break it into subframes of at most 100k. The following functions shall accept data buffers up to 100 KiB large. For most devices, the codec is the limiting factor for buffer sizes.

```
OEMCrypto_DecryptCENC
```

```
OEMCrypto_CopyBuffer
```

Key Expired Error Codes

Several new unit tests will verify that the following functions return the error code OEMCrypto_ERROR_KEY_EXPIRED if the key's timer has surpassed the duration field in the key control block. Previously, some unit tests only verified that the function returned an error.

```
    OEMCrypto_SelectKey
    OEMCrypto_DecryptCENC
    OEMCrypto_Generic_Encrypt
    OEMCrypto_Generic_Decrypt
    OEMCrypto_Generic_Sign
    OEMCrypto_Generic_Verify
```

Key ID Length

OEMCrypto may assume that the key id length is at most 16 bytes. Previously, a maximum length was unspecified. OEMCrypto shall correctly handle key id lengths that are from 1 to 16 bytes.

Keys Per Session

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

Number of Sessions

The required number of sessions allowed is increasing from 8 to 10. This value is returned in OEMCrypto_GetMaxNumberOfSessions. Some applications use multiple sessions to pre-fetch licenses, so high end devices should support more sessions -- we recommend a minimum of 50 sessions.

HDCP Level

The key control block field controlling HDCP level has a new value: 0xF = "local display only value". This is interpreted as the content should only be displayed locally. The content should not be available to any external display, including HDMI, no matter what the HDCP level is.

Key Renewal Does Not Modify Key Control

The key control block should not be modified by the OEMCrypto_RefreshKeys. New unit tests will be added to make sure that modified control fields generate an error in OEMCrypto_RefreshKeys.

DecryptCENC Subsamples Might Not Be Pairs

The WebM VP9 spec specifies that a sample might have a final subsample that is not encrypted. Although the OEMCrypto API did not say so, most content had calls to DecryptCENC in pairs of one clear subsample followed by one encrypted subsample. This behaviour is not guaranteed, and new unit tests will be added to verify that OEMCrypto does not rely on this.

Provisioning 3.0

Provisioning 3.0 is a way for OEMs to provision their devices using an X.509 certificate generated by the OEM, instead of using a keybox generated by Google. This PKI-based approach allows for other third parties to provision devices for DRM or other services. Provisioning 3.0 is an optional feature: Keyboxes will still be supported by the CDM layer and by Google certificate provisioning servers. There is no plan to deprecate keyboxes at this time.

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

OEM's that wish to use Provisioning 3.0 certificates should return OEMCrypto_OEMCertificate from a call to OEMCrypto_GetProvisioningMethod(). They should implement OEMCrypto_GetOEMPublicCertificate(), OEMCrypto_RewrapDeviceRSAKey30() and make sure OEMCrypto_GenerateRSASignature works with the OEM certificate, as described below.

Implementations which have not yet been updated to Provioning 3.0 should return OEMCrypto_UsesKeybox from a call to OEMCrypto_GetProvisioningMethod(). They should return OEMCrypto_ERROR_NOT_IMPLEMENTED from calls to OEMCrypto_GetOEMPublicCertificate() and OEMCrypto_RewrapDeviceRSAKey30(). They can ignore the rest of this section.

For a complete description of Provision 3.0, please see the document "Widevine Provisioning 3.0 Design". OEMs will request a single X.509 CA certificate from Google for each make and model of the device, and use them to sign the device specific certificates which the OEM will generate for each device. The device specific certificate will be installed on the device in the factory. OEMCrypto will pass the certificate up to the CDM layer when the function OEMCrypto_GetOEMPublicCertificate is called. The OEMCrypto library will also load the private RSA key corresponding to the certificate when OEMCrypto_GetOEMPublicCertificate is called. It is the OEM's responsibility to make sure that the private RSA key is not accessible to the user.

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

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

The following functions are for provisioning 2.0 only, and may return OEMCrypto_ERROR_NOT_IMPLEMENTED if the device uses an OEM certificate.

```
OEMCryptoResult OEMCrypto_LoadTestKeybox();
OEMCryptoResult OEMCrypto_IsKeyboxValid();
OEMCryptoResult OEMCrypto_GetDeviceID(...);
OEMCryptoResult OEMCrypto_GetKeyData(...);
```

The following is a list of new or modified functions.

```
OEMCrypto_ProvisioningMethod  OEMCrypto_GetProvisioningMethod()
enum OEMCrypto_ProvisioningMethod  {
      OEMCrypto_ProvisioningError  = 0,
      OEMCrypto_DrmCertificate  = 1,
      OEMCrypto_Keybox  = 2,
      OEMCrypto_OEMCertificate  = 3
}
```

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

- **DrmCertificate** means the device has a DRM certificate built into the system. This can only be used by certain level 3 devices.
- **Keybox** means the device has a unique keybox. For level 1 devices this keybox should be installed in the factory.
- **OEMCertificate** means the device has a factory installed OEM certificate.
- **ProvisioningError** indicates a serious problem with the OEMCrypto library.

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

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

Parameters

- `session` - (in) this function affects the specified session only.
- `public_cert` (out) the buffer where the public certificate is stored.
- `public_cert_length` - (in/out) on input, this is the available size of the buffer. On output, this is the number of bytes needed for the certificate.

Returns

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

```
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)
```

This function's signature and purpose have not changed. However, if it is called before calling

OEMCrypto_LoadDeviceRSAKey, it uses the OEM certificate's private key to sign the message. The only padding_scheme allowed for an OEM certificate is 0x1 - RSASSA-PSS with SHA1. Any other padding scheme must generate an error.

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

This function is similar to OEMCrypto_RewrapDeviceRSAKey. It should verify 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. The wrapped RSA key will be loaded in the future with OEMCrypto_LoadDeviceRSAKey.

Verification and Algorithm

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

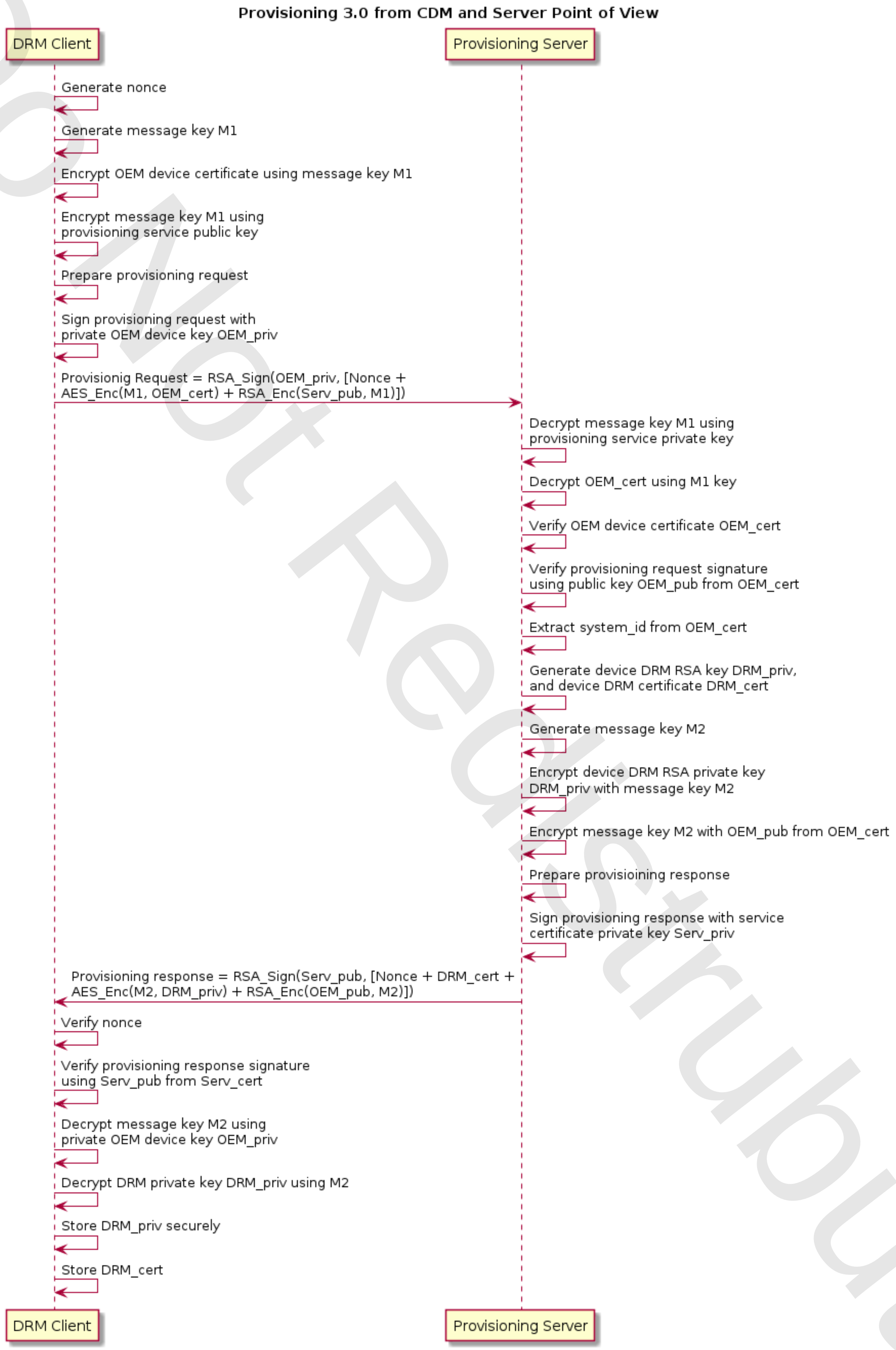
1. Verify that `in_wrapped_rsa_key_length` is large enough to hold the rewrapped key, returning `OEMCrypto_ERROR_SHORT_BUFFER` otherwise. This error does not invalidate the nonce or remove it from the nonce table.
2. Verify that the nonce matches one generated by a previous call to `OEMCrypto_GenerateNonce()`. The matching nonce shall be removed from the nonce table. If there is no matching nonce, return `OEMCRYPTO_ERROR_INVALID_NONCE`.
3. Decrypt encrypted_message_key with the OEM certificate's private RSA key using RSA-OAEP into the buffer message_key. This message key is a 128 bit AES key used only in step 4. This message_key should be kept in secure memory and protected from the user.
4. Decrypt enc_rsa_key into the buffer rsa_key using the message_key, which was found in step 3. Use enc_rsa_key_iv as the initial vector for AES_128-CBC mode, with PKCS#5 padding. The rsa_key should be kept in secure memory and protected from the user.
5. If the first four bytes of the buffer rsa_key are the string "SIGN", then the actual RSA key begins on the 9th byte of the buffer. The second four bytes of the buffer is the 32 bit field "allowed_schemes", of type RSA_Padding_Scheme, which is used in OEMCrypto_GenerateRSASignature.   The value of allowed_schemes must also be wrapped with RSA key. We recommend storing the magic string "SIGN" with the key to distinguish keys that have a value for allowed_schemes from those that should use the default allowed_schemes. Devices that do not support the alternative signing algorithms may refuse to load these keys and return an error of OEMCrypto_ERROR_NOT_IMPLEMENTED. The main use case for these alternative signing algorithms is to support devices that use X.509 certificates for authentication when acting as a ChromeCast receiver. This is not needed for devices that wish to send data to a ChromeCast.
6. If the first four bytes of the buffer rsa_key are not the string "SIGN", then the default value of allowed_schemes = 1 (kSign_RSASSA_PSS) will be used.

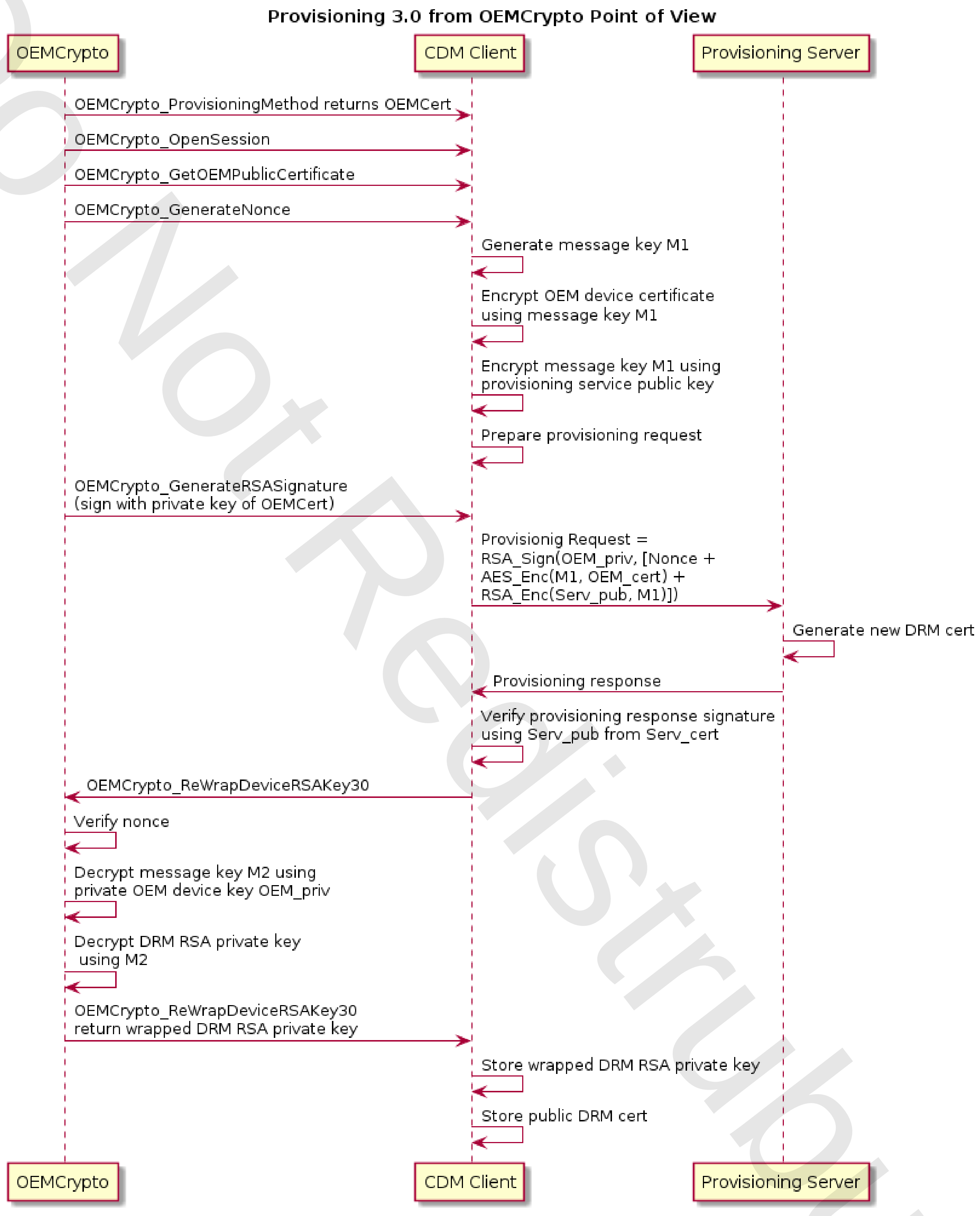
7. After possibly skipping past the first 8 bytes signifying the allowed signing algorithm, the rest of the buffer rsa_key contains an RSA device key in PKCS#8 binary DER encoded format. The OEMCrypto library shall verify that this RSA key is valid.
8. Re-encrypt the device RSA key with an internal key (such as the OEM key or Widevine Keybox key) and the generated IV using AES-128-CBC with PKCS#5 padding.
9. Copy the rewrapped key to the buffer specified by `wrapped_rsa_key` and the size of the wrapped key to `wrapped_rsa_key_length.`

Parameters
[in] session: crypto session identifier.
[in] nonce: A pointer to the nonce provided in the provisioning response.
[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

Sequence Diagrams for Provisioning 3.0

Below are sequence diagrams indicating a provisioning request from a high level, and from an OEMCrypto view point.


Provisioning 3.0 from CDM and Server Point of View
DRM Client
Provisioning Server
Generate nonce
Generate message key M1
Encrypt OEM device certificate using message key M1
Encrypt message key M1 using provisioning service public key
Prepare provisioning request
Sign provisioning request with private OEM device key OEM_priv
Provisionig Request = RSA_Sign(OEM_priv, [Nonce + AES_Enc(M1, OEM_cert) + RSA_Enc(Serv_pub, M1)])
Decrypt message key M1 using provisioning service private key
Decrypt OEM_cert using M1 key
Verify OEM device certificate OEM_cert
Verify provisioning request signature using public key OEM_pub from OEM_cert
Extract system_id from OEM_cert
Generate device DRM RSA key DRM_priv, and device DRM certificate DRM_cert
Generate message key M2
Encrypt device DRM RSA private key DRM_priv with message key M2
Encrypt message key M2 with OEM_pub from OEM_cert
Prepare provisioining response
Sign provisioning response with service certificate private key Serv_priv
Provisioning response = RSA_Sign(Serv_pub, [Nonce + DRM_cert + AES_Enc(M2, DRM_priv) + RSA_Enc(OEM_pub, M2)])
Verify nonce
Verify provisioning response signature using Serv_pub from Serv_cert
Decrypt message key M2 using private OEM device key OEM_priv
Decrypt DRM private key DRM_priv using M2
Store DRM_priv securely
Store DRM_cert
DRM Client
Provisioning Server


Provisioning 3.0 from OEMCrypto Point of View
OEMCrypto
CDM Client
Provisioning Server
OEMCrypto_ProvisioningMethod returns OEMCert
OEMCrypto_OpenSession
OEMCrypto_GetOEMPublicCertificate
OEMCrypto_GenerateNonce
Generate message key M1
Encrypt OEM device certificate using message key M1
Encrypt message key M1 using provisioning service public key
Prepare provisioning request
OEMCrypto_GenerateRSASignature (sign with private key of OEMCert)
Provisionig Request = RSA_Sign(OEM_priv, [Nonce + AES_Enc(M1, OEM_cert) + RSA_Enc(Serv_pub, M1)])
Generate new DRM cert
Provisioning response
Verify provisioning response signature using Serv_pub from Serv_cert
OEMCrypto_ReWrapDeviceRSAKey30
Verify nonce
Decrypt message key M2 using private OEM device key OEM_priv
Decrypt DRM RSA private key using M2
OEMCrypto_ReWrapDeviceRSAKey30 return wrapped DRM RSA private key
Store wrapped DRM RSA private key
Store public DRM cert
OEMCrypto
CDM Client
Provisioning Server

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

Key Control Block Changes

The functionality described above requires the following changes to the key control block. The key control block should still be verified in the LoadKeys function. To allow for backwards compatibility, OEMCrypto should accept a key control block for previous versions of the API.

The four verification bytes in the key control block are valid if they are “kctl”, “kc09” or “kc10”, “kc11” or “kc12”. An OEMCrypto that implements the new v12 functions described in this document should accept any of these four strings. It should not accept any other string -- in particular, it should NOT accept “kc13”.

The field HDCP_Version has a new value 0xF which indicates the device should not send content to an external display.

bit 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 0xF - Local Display Only (no HDMI allowed)