Widevine CE CDM v3.5.x Integration Guide

Document version 2.5

Revision History

Version	Date	Description	Author
0.1	2015-06-13	Initial draft.	Joey Parrish
1.0	2015-06-18	Addresses internal feedback: fixed typographical consistency problems, corrected unclear language, simplified API headings, corrected OpenSSL notes.	Joey Parrish
1.1	2015-09-11	Added new methods from v3.0.1 interface: Cdm::*AppParameter	Joey Parrish
1.2	2015-10-08	Most third party dependencies are now included, and install_third_party.sh has been removed.	Joey Parrish
1.3	2015-12-04	v3.0.4 interface: updated type names and added kPersistentUsageRecord.	Joey Parrish
1.4	2015-12-15	v3.0.5: Add openssl_config gyp variable.	Joey Parrish
2.0	2016-07-18	v3.1.0: New certificate provisioning flow, per-origin storage, CENC 3.0 support, decryption of HLS content, generic crypto operations and querying allowed key usage.	Gene Morgan, Jacob Trimble, John Bruce
2.1	2016-09-02	v3.1.1: Corrected provisioning URL parameter from signedMessage to signedRequest.	Joey Parrish
2.2	2016-12-12	Begin v3.2.x updates.	Gene Morgan
2.2	2016-12-19	Add Cdm::setVideoResolution() Add Cdm::isProvisioned(), Cdm::removeProvisioning	Gene Morgan
2.2	2016-01-24	Add IStorage::list() Add Cdm::listStoredLicenses	Gene Morgan
2.2	2016-01-25	Support Provisioning v3.0	Gene Morgan

		Removed default service certificates. Client must supply valid service certificates for the provisioning and license servers (see setServerCertificate() and setDeviceProvisioningServerCertificate() for details.	
2.3	2017-05-03	Support Provisioning v3.0 Added APIs for managing Service Certificates and Usage Table Entries. Added section describing Service Certificates. Added section describing Provisioning Server message formats.	Gene Morgan
2.4	2017-07-20	Define new load() method that loads sublicenses for a session. Add brief description of SubLicenses and SubSessions.	Gene Morgan
2.5	2017-11-22	Updates for OEMCrypto v13; update third_party materials.	Gene Morgan

This document replaces the documents "Widevine Security Integration Guide for CENC: EME Supplement" and "CDM Porting Guide" that shipped with v3.2.x of the CE CDM.

Table of Contents

Purpose

This document gives an overview of the Widevine CE CDM 3.5 software stack, explains high-level components and responsibilities, and gives brief explanations of all CDM APIs. It also documents the build system and explains the basics porting the CDM to a new platform.

You may also want to refer to *Widevine Modular DRM Security Integration Guide for Common Encryption (CENC)*, which documents the OEM-provided OEMCrypto library, a critical component of the system.

Audience

This document is intended for:

- SOC and OEM device manufacturers who wish to deploy Widevine content protection on embedded devices not running Android
- Application developers who wish to integrate the Widevine CDM directly into their application in order to use Widevine content protection where it is not provided by the platform

External References

Encrypted Media Extensions Specification: https://w3c.github.io/encrypted-media/

What is a CDM?

CDM stands for "Content Decryption Module". The term comes from the *Encrypted Media Extensions Specification* (EME). This is a client-side component that provides content protection services to an application, such as generating license requests and performing decryption.

Although EME is specified in the context of a web browser, the Widevine CDM can be used for content protection in other platforms and applications as well.

The Widevine CE CDM is intended for consumer electronics (CE) devices other than Android. Android has its own Widevine implementation and uses a different API.

One CDM instance can have multiple sessions. Sessions are contexts for key management, and are defined in more detail in the EME specification. One instance of the CE CDM library can have multiple CDM instances.

The CDM instance is created and managed by a component that the EME specification calls a User Agent. The user agent role may be fulfilled by a browser or a native application, as shown in the following sections.

CE CDM Software Stack

For Browsers (HTML5/EME stack)

Figure 1 shows the architecture and playback flow for a browser integration of the Widevine CE CDM.

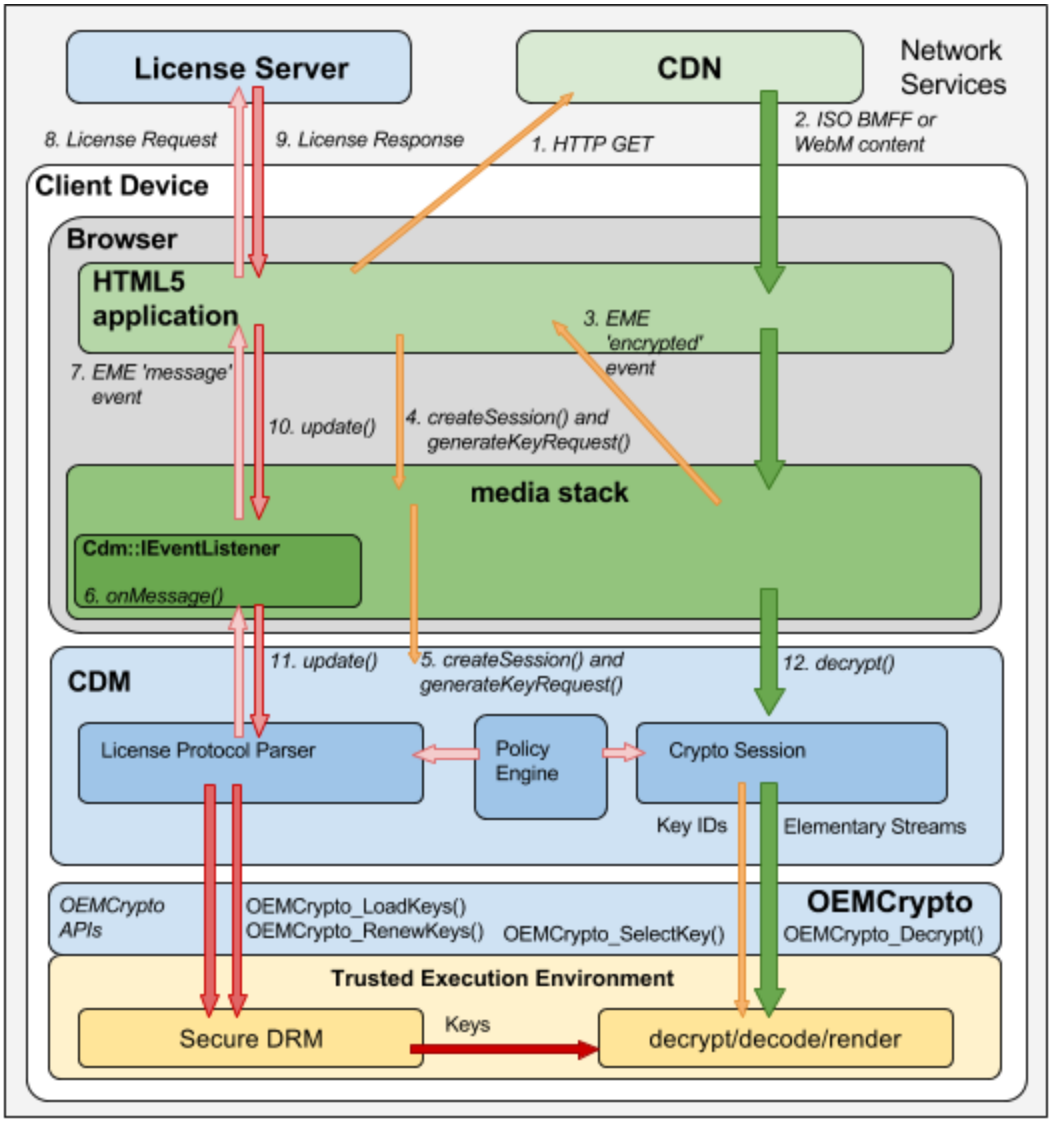

Figure 1. HTML5/EME Playback with Widevine CDM

Please note the basic components. The browser's media stack communicates with the CDM, which in turn communicates with the trusted execution environment through OEMCrypto. The

media stack implements `IEventListener`, which the CDM will use to send information asynchronously to the media stack and browser.

Also, please note the flow of events and information, numbered from 1 to 12 in this diagram. The HTML5 application requests content from the CDN (1-2), and feeds it to the browser's media stack. The media stack detects that the content is encrypted and sends an `'encrypted'` event (3) to the application. The application uses EME APIs `createSession()` and `generateRequest()`, which are then proxied to the CDM itself (4-5). The CDM sends a message to the media stack, which is then proxied to the HTML5 application (6-7). The HTML5 application relays the license request to the license server, and passes the response back to the CDM via the EME `update()` method (8-11). Finally, the media stack asks the CDM to decrypt the content (12).

For Native Applications

Figure 2 shows the architecture and playback flow for a native application using the Widevine CE CDM.

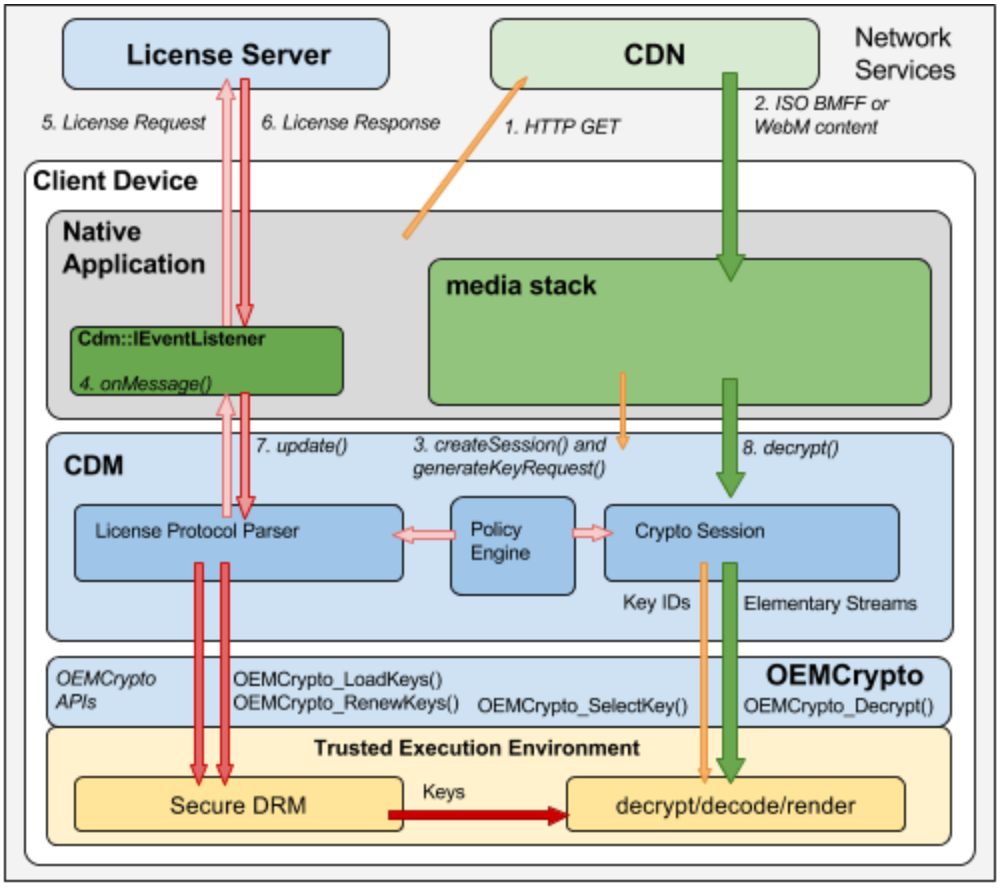

Figure 2. Native Application Playback with Widevine CDM

Please note the basic components. Like in the HTML5 diagram above, the media stack communicates with the CDM, which in turn communicates with the trusted execution environment through OEMCrypto. The application implements `IEventListener`, which the CDM will use to send information asynchronously to the application.

The flow of events and information, numbered 1-8, is very similar to the HTML5 diagram above, but with less proxying.

The application requests content from the CDN (1-2), and feed it to the media stack. The media stack detects that the content is encrypted and calls `createSession()` and `generateRequest()` on the CDM (3). The CDM sends a message to the application, which relays the license request to the license server and passes the response back to the CDM via the `update()` method (4-7). Finally, the media stack asks the CDM to decrypt the content (8).

Responsibilities

The application is responsible for:

- All networking and communications
- All persistent storage (must implement interface `IStorage`)
- All time-related functionality (must implement `IClock`)
- All asynchronous timers (must implement `ITimer`)
- Relaying messages to the license server and provisioning server (must implement `IEventListener`)
- Managing service certificates for authenticating license server and provisioning server transactions
- Managing the lifetime of CDM instances and the sessions contained within them

The CDM is responsible for:

- Generating license requests and interpreting license responses
- Enforcing content policies
- Managing crypto resources
- *NOTE: The CDM has no networking or filesystem access except through the application.*

OEMCrypto is responsible for:

- Securely storing content keys
- Securely decrypting content
- Providing a root of trust for the system

NOTE: Some applications may use the CDM for license exchange only and bypass the CDM for decryption, using instead either direct calls to OEMCrypto or to a private interface in the Trusted Execution Environment (TEE).

Build Requirements

The CE CDM requires a few things to build. The root of the build system uses a python script, so python is required:

- python2 (v2.7+).

Additional dependent tools and libraries are collected into the third_party folder:

- gyp (open-source build automation tool, uses python)
- protobuf (v2.6.1 specifically, used by the CDM as part of the license protocol)
- gTest and gMock (v1.8.0, for unit tests)
- jsmn (open-source JSON parser, used for processing an HLS playlist)
- boringssl (rev: f7412cb072cc6b1847140e0c4f8b3ceeccd0e708)
- fuzz (currently in limited use, will be used for fuzz-testing the OEMCrypto and CDM interfaces)

Application Responsibilities

Networking

The CDM relies on the application to relay messages to servers. The CDM has no networking capability of its own. The application must implement the `IEventListener` interface to receive messages from the CDM via `IEventListener::onMessage()`. The CDM may also require the owner of IEventListener to provision a device certificate. (See below.)

Device Certificate Provisioning

Starting with v3.1, the Widevine CE CDM requires a unique certificate for each device and for each origin. If, when creating a session, a device certificate is not available, the CDM will require the user agent to relay a message to the provisioning server to get a device certificate. That certificate will be stored (via the IStorage interface) and used for future sessions.

During a call to generateRequest() or load(), if Device Certificate Provisioning is required, the CDM will issue an IEventListener::onDirectIndividualizationRequest() callback instead of processing the call immediately. The callback contains a message to be sent to the provisioning server. The response must be passed back to to the CDM via Cdm::update() the same as any other message. After this, the CDM will handle the generateRequest() or load() calls.

If multiple sessions are created before the device is fully provisioned, the calls will return kDeferred. This indicates that a message was not generated yet, but will be once the device is provisioned. Once the device is provisioned, a call to `IEventListener::onDeferredComplete` will be made with the result of the call. (i.e. kSuccess or the error code)

Certificate Provisioning Message Formats

The request message supplied in the IEventListener::onDirectIndividualizationRequest() callback is a URL-encoded (also known as web-safe) base64 signed message that must to be sent to the provisioning server. It is safe to pass the data directly as a URL query-parameter. The default provisioning server expects this data as the signedRequest parameter as shown here:

```
https://www.googleapis.com/certificateprovisioning/v1/devicecertificates/create?key=AIzaSyB-5OLKTx2iU5mko18DfdwK5611JIjbUhE&signedRequest=<request_data>
```

The response message from the provisioning server will be in a similar format, with the actual message data in a quoted signedResponse parameter, similar to the following:

```
HTTP/1.1 200 OK
X-Google-Netmon-Label: . . .
. . .
Accept-Ranges: none
Vary: Origin,Accept-Encoding
Connection: close

{
 "kind": "certificateprovisioning#certificateProvisioningResponse",
 "signedResponse": "<response_data>"
}
```

This form is used by Widevine's provisioning server. The CDM produces only the <request_data> string for the request, and accepts only the full response message containing the "signedResponse:" field, from which it extracts the <response_data> string. Other provisioning servers may have different methods for conveying the request and response. The CDM has a Property called "provisioning_messages_are_binary", which is set false by default. It can be set to true for a provisioning server that consumes and produces binary protobuf message strings. Any other filtering or data conversion required for messages to or from a particular provisioning server to conform to these CDM-supported formats must be handled outside the CDM.

Device Certificate Provisioning v3.0

Starting with v3.2, the Widevine CE CDM has a new provisioning scheme. It replaces the keybox as an authentication object with an X.509 certificate called an OEM Certificate. This Certificate is passed to the Provisioning Server in a Provisioning Request. The Provisioing Server returns a Provisioning Response message containing a unique DRM Device Certificate and corresponding private key. The Widevine CE CDM passes the DRM Device Certificate and Private Key to OEMCrypto for verification. OEMCrypto also rewraps the private key using its own private key. The result is stored on the file system for use in future sessions. The DRM

Device Certificate is used in License Server requests to identify and authenticate the device with the license server.

Provisioning V3.0 requires OEMCrypto V12 or better. Older OEMCrypto versions are still supported by the CDM, but the "traditional" Keybox-based provisioning will be used. OEMCrypto V12 and newer systems can support either Keybox or certificate-based provisioning, but not both. The provisioning method is hard-coded into OEMCrypto. The CDM can determine the system's provisioning method and construct the correct Provisioning Request in all these cases.

Service Certificates

Service Certificates are associated with license and provisioning servers. The Service Certificate provides the public key for the service, and is used to sign and verify communications between the CDM and the server. The CDM API provides calls for registering a Service Certificate with the CDM. Applications frequently have a copy of the Service Certificate for the servers they communicate with. In addition, the License Server supports a query to obtain a copy of its Service Certificate; the CDM provides APIs to generate the request message and process the response for this query.

Storage

The application is responsible for storage on behalf of the CDM. The application must implement the `IStorage` interface, which is a simple key-value storage mechanism. The application is not required to use a filesystem for this information, but it must be persisted in some non-volatile form from one application launch to the next.

It is required to store data per-app or per-origin for security reasons (see *https://w3c.github.io/encrypted-media/#privacy-leakage* for more information). This is done by using multiple instances of `IStorage`. Passing an IStorage instance to create() will cause the resulting CDM instance to only act within that "storage". There is also a global or default `IStorage` instance passed into initialize(). Passing a NULL IStorage pointer to create() will cause the resulting CDM to use the default `IStorage` instance.

NOTE: It is important for users of your application to be able to clear stored data somehow. See http://www.w3.org/TR/encrypted-media/#privacy-storedinfo for more information.

Clock

The CDM also has no internal clock mechanism. The application must implement the `IClock` interface, which provides the current time when queried.

Timers

The CDM will sometimes need to do asynchronous processing. The CDM does not create any threads. Instead, it requires the application to implement the `ITimer` mechanism, which allows the CDM to request a callback after a specific amount of time has passed.

*NOTE: These timers are one-shot timers! If the CDM asks for a callback after 2 seconds, the application should **not** call the CDM back **every** 2 seconds.*

CDM APIs

Starting with v3.0, the CE CDM APIs are very similar to the JavaScript APIs specified in the EME spec. Here we will give a brief overview of the methods, their purpose, and any important caveats to their use.

The header "cdm/include/cdm.h" is the only header used by the integrator or application developer. More detail on methods and types is available in cdm.h, and that header should be viewed as the canonical reference for the CDM API. Should this document conflict with the header, the header takes precedence.

The CE CDM has no method corresponding to EME's `requestMediaKeySystemAccess()` method. This is used by EME for negotiating which CDM to use and what features it has, and is therefore outside the scope of the CDM.

Methods in EME that operate on a session object are implemented in the CDM as methods that have a session ID argument.

All CDM methods return a `Status` value. Some values are defined by the CDM, while others correspond to specific DOM exceptions specified in EME. Error conditions specified by EME return corresponding `Status` values. For example, where EME says certain inputs trigger a `TypeError`, we would return `kTypeError` for the same inputs.

The Status constant `kUnexpectedError` is used to cover all unexpected error cases. This status may indicate a bug or misconfiguration of the CDM, and generally should not occur. Any time you see this return value from the CDM, there should be more detailed information available in the log, including Widevine-internal error codes.

`initialize`

Used to initialize the library. Must be called before any other method.

`create`

Creates and returns a CDM instance. Multiple CDM instances can coexist. CDM instances are destroyed by a typical C++ `delete`.

The `privacy_mode` argument controls the encryption of client identification. If `privacy_mode` is true, the CDM will use a server certificate to encrypt the client identification in the license request. We strongly recommend the use of privacy mode whenever possible.

`setServiceCertificate`

This was previously named setServerCertificate().

Allows the application to provide a service certificate to the CDM. This certificate is used to encrypt client identification information in license requests as part of "privacy mode" (See `create()`). It is also needed to sign and verify the messages for certificate provisioning, and the certificate holds a provider_id string that is needed for constructing the provisioning request.

It is preferred that the service certificate be supplied by the client application, but the CDM client can also request a Service Certificate from the License Server. The API calls **getServiceCertificateRequest** and **parseServiceCertificateResponse** (see below) are provided to assist with this operation.

In previous versions of the CDM (v3.2 and earlier), a Service Certificate would be automatically fetched if needed, as part of the license request flow. This is no longer supported. If a Service Certificate is needed and none has been registered with the CDM, an error will be returned.

`getServiceCertificateRequest`

Generate a License Server protocol message to get a copy of the server's Service Certificate. The returned string should be sent to the License Server, which will reply with a Service Certificate Response message. This message should be passed to **parseServiceCertificateResponse** as discussed below.

`parseServiceCertificateResponse`

Process the message from the License Server in response to a Service Certificate request. If no error is reported, the Service Certificate is registered with the CDM Session and will be used as needed until the next **parseServiceCertificateResponse** or **setServiceCertificate** call, or until the session ends. A copy of the Service Certificate is also returned that may be stored and used for future sessions (by passing the string in a call to **setServiceCertificate**)

`createSession`

Creates a new CDM session. A session is a context for a license and its associated keys. The output parameter `session_id` will be used to identify the session in calls to all other CDM methods.

If `session_type` is `kPersistentLicense`, the session will be stored on disk for offline use, and can be subsequently loaded using `load()`.

If `session_type` is `kPersistentUsageRecord`, the session will persist, but the keys will not. A record of session usage will be sent on `remove()`.

`generateRequest`

Generates a license request to be relayed to the license server by the application. The request will be delivered to the application synchronously via `IEventListener::onMessage()`.

If a server certificate needs to be provisioned before requesting a license, the CDM will issue a server certificate request before the license request (both will be of type `kLicenseRequest`). Applications will not need to handle the additional message differently and should simply relay each request to the license server.

If multiple sessions are created while the device is unprovisioned, this method may return `kDeferred`. This indicates that a message has not been sent yet and will be sent once the device is provisioned. Once the message is sent, `IEventListener::onDeferredComplete()` will be called with the result of this function. (i.e. `kSuccess` or the error code)

`isProvisioned`

Returns true if the device has been provisioned (i.e., has a Device Certificate). Provisioning is performed per-origin - the test applies only to the current origin of the CDM instance as determined by its IStorage object.

`removeProvisioning`

Deletes the current Device Certificate. Distinct Device Certificates are maintained for each origin - only the current origin of the CDM instance, as determined by its IStorage object, is affected.

`removeUsageTable`

Deletes the current usage table. This is performed using an existing OEMCrypto call.

`listStoredLicenses`

Returns a vector of Key Set ID strings, representing the licenses currently stored in the current (origin-specific) file system.

`listUsageRecords`

Returns a vector of Key Set ID strings, representing the usage records currently stored in the current (origin-specific) file system.

deleteUsageRecord

Accepts a Key Set ID string, and deletes the usage record for that ID if it exists.

deleteAllUsageRecords

Delete all usage records currently stored in the current (origin-specific) file system.

load

There are two variants of the load function:

(1) **load(const std::string& session_id)**

Loads a persisted session from storage. The `session_id` parameter should be the same session ID generated by the CDM when the session was first created. Note that sessions of type `kTemporary` may not be loaded again, and that sessions of type `kPersistentLicense` that are still open may not be loaded a second time.Similar to `generateRequest`(), this may return `kDeferred` if the device is unprovisioned.

(2) **load(const std::string& session_id,**
InitDataType init_data_type,
const std::string& init_data)

Initiates the loading of key data embedded in the pssh. The session must have been already opened and loaded with a license granting permission to use the embedded keys. For more information about embedding keys for use in key rotation contact Google's Widevine Engineering team.

update

Used to pass messages from the license server to the CDM. When the application receives a message via `IEventListener::onMessage()` and relays it to the license server, the HTTP response (not including HTTP headers) should be given back to the CDM via `update()`.

getExpiration

Used to query key expiration time for a session.

getKeyStatuses

Used to query the statuses of all keys for a session. Typical key statuses are `kUsable` and `kExpired`. If a key is not in a `kUsable` state, it can't be used to decrypt content.

`getKeyAllowedUsages`

Used to query how a particular key may be used. Keys may be constrained to a particular usage or set of of usages in addition to or instead of normal content decryption. Key usages include: media content decryption (to a buffer in the clear), media content decryption (to a secure buffer), generic encryption, generic decryption, generic signing, and generic signature verification. The allowed usage for a key is independent of the key status, which indicates whether or not a media content key is currently usable.

`close`

Used to close an active session and release temporary resources associated with it. If it is of type `kPersistentLicense`, the session will not be removed from storage. No release messages (type `kLicenseRelease`) will be generated.

`remove`

Used to remove a persistent session and all resources associated with it. Not used for sessions of type `kTemporary`. Session must be loaded before removal.

Generates a message of type `kLicenseRelease`, which must be relayed to the license server. The server's reply to the release message must be passed to `update()` before removal is complete. Session information remains stored on disk until this process is complete. Once a session has been fully removed via `update()`, the session is considered closed, and `close()` need not be called.

A "partially-removed" session is one for which `remove()` has been called, but the release has not been confirmed via `update()`. A partially-removed session from an earlier run of the application may be fully removed in a subsequent run. Simply `load()` the session and complete the removal via `update()`.

`decrypt`

Used to decrypt content. Some applications may wish to use the CDM for license exchange only. These applications may therefore bypass the CDM for decryption and initiate decryption through OEMCrypto or directly in the TEE.

Starting with v3.1, the Widevine CDM supports all four encryption modes from the ISO-CENC 3.0 standard. These are known as cenc, cens, cbc1, and cbcs. The CDM will automatically select the correct mode depending on the input parameters passed into `decrypt()`. It bases this decision on the `pattern` and `encryption_scheme` parameters. If the `pattern` parameter is left at its default value of (0,0), then pattern usage will be disabled. If a pattern other than (0,0) is specified, then pattern usage will be enabled. In combination with the `encryption_scheme` parameter, the CDM can determine which CENC 3.0 mode to use:

	No Pattern	Pattern Specified
`kAesCtr`	cenc	cens
`kAesCbc`	cbc1	cbcs

setAppParameter, getAppParameter, removeAppParameter, clearAppParameters

These methods have been added for the convenience of Android application developers. If you have an Android app that uses the optionalParameters argument in MediaDrm.getKeyRequest, this interface will allow to maintain compatibility with your Android app on other platforms.

Parameters set through these methods are arbitrary key-value pairs to be included in license requests. These methods have no counterpart in EME, and their use is discouraged.

`genericEncrypt, genericDecrypt, genericSign, genericVerify`

These methods provide an application with basic crypto operations independent of the CDM's content decryption support. They can be used to handle encryption, decryption, signing, and signature verification for messages exchanged between the application and its server. The keys for these operations are supplied through the same licensing protocol used for content keys.

`setVideoResolution`

This method allows an application to inform the CDM of the resolution of the device (width and height in pixels). The information is used in conjunction with license constraints to determine whether a particular key can be used on this device. If this call is not made, the resolution constraint enforcement is not performed.

Using the CDM With HLS Content

Starting with v3.1, the Widevine CDM supports the CENC 3.0 cens mode. This mode can also be used to decrypt content that is in Apple's HLS format and that uses its SAMPLE-AES encryption mode. However, there are some additional considerations that applications must be aware of before attempting to do this.

Using the Correct CENC 3.0 Mode

HLS content should always be decrypted in the CENC 3.0 cens mode. This means that when calling `decrypt()`, the `encryption_scheme` parameter should be set to `kAesCbc` and the `pattern` parameter should be set to something other than (0,0). For video frames, HLS SAMPLE-AES mandates a pattern of (1,9). For audio frames, patterns are not used. However,

setting the `pattern` parameter to (0,0) will cause the CDM to operate in cbc1 mode, which is incorrect. Therefore, Widevine recommends using a pattern of (1,0) for HLS audio content, in order to activate cens mode while still decrypting every crypto block.

Extra Start Code Emulation Prevention

The Widevine CDM only handles decryption and does not know anything about the file formats the data comes from. As such, the Widevine CDM does not do any special handling for the extra start code emulation prevention in Apple's HLS format.

HLS uses H.264 video in the H.264 Annex B bytestream format. This format defines a process called "start code emulation prevention" which must be applied to the H.264 NAL Units during encoding in order to prevent false start codes from appearing in the bytestream. The HLS SAMPLE-AES specification adds to this that, after encryption is applied, start code emulation prevention must be applied a *second* time to any NAL Units that contain encrypted data, in case encryption created any new false start codes. This is applied to the entire NAL Unit, including the unencrypted portions.

The CDM expects that content passed to it with an `encryption_scheme` of `kClear` is ready to be decoded without further processing. And it expects that content passed to it with an `encryption_scheme` other than `kClear` is ready to be decrypted without further processing, after which it will be ready to be decoded.

As such, it is the responsibility of any users of the CDM to remove the extra start code emulation prevention from any H.264 NAL Units that contain encrypted data. They should *not* remove the single layer of start code emulation prevention from the NAL Units that do not contain encrypted data.

The video decoder will expect exactly one layer of start code emulation prevention to be on the video data when it decodes it. So content passed to the CDM with an `encryption_scheme` of `kClear` should already have just one layer of start code emulation prevention. And content passed to the CDM with an `encryption_scheme` of `kAesCbc` should already have its second, extra layer of start code emulation prevention removed so that it can be immediately decrypted, after which it will also only have one layer of start code emulation prevention.

Special Treatment of the Last 16 Bytes of a Video Frame

The SAMPLE-AES specification defines special handling for video frames with an encrypted area that is an exact number of crypto blocks (16 bytes) long. This handling is unique to video frames in HLS and is different from CENC 3.0 or the handling of audio frames in HLS. Applications will need to pass the last 16 bytes of any such video frames as a separate clear decrypt call.

HLS uses AES-CBC encryption. Because AES-CBC does not support partial crypto blocks, if the encrypted area of a frame is not an even multiple of 16 bytes long, then there will be some extra bytes at the end after the last full crypto block. In both HLS and CENC 3.0, these bytes are left clear and thus do not need to be decrypted by the CDM. If an encrypted input that is not an even multiple of crypto blocks long is passed into the CDM when using AES-CBC, the CDM will follow the CENC 3.0 standard and will automatically leave the extra bytes clear.

However, if the encrypted area is an even multiple of crypto blocks long, the HLS SAMPLE-AES specification states that video frames should be treated differently than other content. For HLS audio frames and for CENC 3.0 content, an CBC-encrypted input that is an even multiple of crypto blocks long will all be decrypted. For HLS video frames, the last crypto block (16 bytes) is instead changed into “extra bytes” and left in the clear.

Because the CDM follows the CENC 3.0 standard and does not handle this HLS video edge case specially, it is the responsibility of users of the CDM to not pass these 16 extra bytes as part of an encrypted input to `decrypt()` and instead pass them as part of a separate, clear input to `decrypt()`.

Widevine recommends that, as a simple way to do this, applications always pass the extra bytes of HLS video frames as a separate clear crypto call, regardless of how many there are. Furthermore, because encrypted data in an HLS stream is almost always followed by clear data, it is acceptable to merge the extra bytes into the next clear data call that would have happened anyway, rather than making a separate clear `decrypt()` call just for the extra bytes.

Build System

The Widevine CE CDM uses a gyp-based build system. gyp is an open-source build system written in python. We provide a simple build script called "build.py", which wraps around the gyp build system and configures the build with platform-specific settings. We will go into more detail on this in the "Porting" section below.

To build the CDM for x86, simply run "./build.py x86-64". This will produce a debug build. To produce a release build, run "./build.py x86-64 -r". Build output goes into the "out" folder. A debug build for x86 will appear in "out/x86-64/Debug". The important outputs are:

- libwidevine_ce_cdm_static.a
- libwidevine_ce_cdm_shared.so
- widevine_ce_cdm_unittest

We build the CDM as both a static and shared library as a convenience to you. You only need to link against one or the other. The header "cdm/include/cdm.h" is the only one you need in order to use the compiled CDM library.

NOTE: OEMCrypto is not linked into the CDM shared library. You must link your application against both the CDM and OEMCrypto. This allows you the flexibility to change OEMCrypto implementations when you build your app instead of when you build the CDM.

Compile-time Options and Configuration

The file "cdm/cdm.gyp" contains the main part of the build description. The "variables" section contains three main options, detailed below. The defaults in cdm.gyp can be overridden with platform-specific values in a platform-specific build file. This will be covered in the "Porting" section below. We strongly recommend that you not edit the defaults in cdm.gyp.

oemcrypto_version

There are several revisions of the OEMCrypto interface in existence (currently v9, v10, and V11, and v12). Based on this variable, the CDM will be built with adapters to allow it to interoperate with older OEMCrypto versions if that is what your platform provides.

protobuf_config

The CDM relies on protobuf as part of the Widevine license protocol. There are three values for this variable, which offer three different ways of integrating protobuf into the build:

system

The protobuf compiler (protoc) and libraries (libprotobuf-lite) are expected to be installed system-wide. When cross-compiling, protoc should be compiled for the host platform, while libprotobuf-lite should be compiled for the target platform.

This setting requires two additional variables:

- protobuf_lib - The protobuf library to link in, such as "-lprotobuf-lite" or "/usr/arm-linux-myarch/lib/libprotobuf-lite.a".
- protoc_bin - The path to protoc, such as "/usr/bin/protoc" or "/usr/local/bin/arm-linux-myarch-protoc".

target

Used for gyp-based projects which already have protobuf in their project's build.

This setting requires three additional variables:

- protobuf_lib_target - The gyp target for the target-toolchain build of libprotobuf-lite, such as "path/to/protobuf.gyp:protobuf_lite".
- protoc_host_target - The gyp target for the host-toolchain build of protoc, such as "path/to/protobuf.gyp:protoc#host".

- protoc_bin - The path to the output of protoc_host_target, typically "<(PRODUCT_DIR)/protoc".

source

This is the default, and is very useful for projects which don't have protobuf already. You provide a path to the source, and the build system will handle compilation of libprotobuf for you. Extremely convenient for cross-compiling, and highly recommended for use when porting the CDM to your target platform.

This setting requires one additional variable:

- protobuf_source - The path to protobuf v2.5.0 sources, such as "path/to/protobuf-2.5.0". You must have a valid config.h for your target platform in this folder. The supplied config.h is appropriate for linux. You may need to adjust config.h for your target platform.

openssl_config

The CDM relies on OpenSSL for privacy features. There are two values for this variable, which offer two different ways of integrating openssl into the build:

system

OpenSSL is expected to be installed system-wide for the target platform.

target

Used for gyp-based projects which already have OpenSSL in their project's build.

This setting requires one additional variable:

- openssl_target - The gyp target for the target-toolchain build of OpenSSL, such as "path/to/openssl.gyp:openssl".

Alternative Build Systems

The gyp-based build system we provide is not the only way to build the CDM sources. As an alternative, you can use the lists of source files in cdm.gyp to integrate the CDM source into your project's existing build system. Please note that although Widevine will not provide support for this, it should be fairly straightforward.

If your project is gyp-based, you can also refer to our gyp files from your own. Make sure the variables described above are properly set in the context of cdm.gyp.

Tests

The Widevine CE CDM comes with a suite of unit tests covering various parts of the codebase. Some tests exercise OEMCrypto, some exercise the CDM's internals (core), and some exercise the CDM APIs at the highest level.

The complete test suite can take several minutes to run, since some of the OEMCrypto tests are quite long. You can exclude any of these large test sets temporarily by commenting out the corresponding lines in cdm_unittest.gyp:

```
'includes': [
  'oemcrypto_unittests.gypi',
  'core_unittests.gypi',
  'cdm_unittests.gypi',
],
```

You may also filter the tests on the command-line using the "gtest_filter" argument. For example, to run only the CdmTest and CdmSession groups:

```
out/x86-64/Debug/widevine_ce_cdm_unittest \
      --gtest_filter=CdmTest.*:CdmSession.*
```

Or to negate a set of tests, prefix the filter with a minus sign. For example, to run all tests except the OEMCryptoClientTest and GenericCryptoTest groups:

```
out/x86-64/Debug/widevine_ce_cdm_unittest \
      --gtest_filter=-OEMCryptoClientTest.*:GenericCryptoTest.*
```

Unit test verbosity can be controlled using the "-v" argument. It may be repeated multiple times to increase verbosity. For example:

```
out/x86-64/Debug/widevine_ce_cdm_unittest # show error logs
out/x86-64/Debug/widevine_ce_cdm_unittest -v # show warnings
out/x86-64/Debug/widevine_ce_cdm_unittest -vv # show info
out/x86-64/Debug/widevine_ce_cdm_unittest -vvv # show debug
out/x86-64/Debug/widevine_ce_cdm_unittest -vvvv # show verbose
```

Porting

Assumptions and Alternatives

The default build makes several assumptions about your platform, but there are some alternatives available to you.

Locking

The file "cdm/src/lock.cpp" assumes the existence of pthread on your platform. If this is not available, one alternative is to write a simple wrapper to implement the pthread functions `pthread_mutex_{init,destroy,lock,unlock}` using your platform's locking primitives. We do not use any other part of the pthread library.

Another alternative to pthread is to exclude "lock.cpp" from the build by commenting it out in cdm.gyp, then implement this same interface differently in your application or platform. The pthread-based implementation is only 30 lines or so, and should be very easy to replace.

Logging

The file "cdm/src/log.cpp" assumes that you can log to stderr. If this is not available, one alternative is to exclude this file from the build and implement the logging interface differently in your application or platform. The stderr-based implementation is only 40 lines or so, and should be very easy to replace.

Protobuf

Protobuf is a critical component of the system, and must be available. Cross-compiling protobuf and installing it system-wide can be tricky. Therefore, we strongly recommend using the default "source" setting for the gyp variable "protobuf_config", as described above in the "Compile-time Options and Configuration" section. This will leverage our build system to handle cross-compilation for you, and does not necessitate system-wide installation.

To use this setting, config.h in the protobuf sources must be appropriate for your target platform. The supplied config.h is appropriate for linux. You can produce a config.h for your target platform in two ways. Either run protobuf's configure script using appropriate flags for your platform, or edit config.h manually to tailor to your target platform.

Adding a New Platform

Platform settings live in the "platforms" folder. When compiling with "./build.py x86-64", the settings in "platforms/x86-64/settings.gypi" and "platforms/x86-64/environment.py" are used.

To add a new platform, make a copy of the x86-64 folder and rename it to the name of your platform. For this example, we will use "HAL9000".

Next, edit "platforms/HAL9000/environment.py" to set the compilers used by your platform and any additional environment variables required by them. For example:

```
tooldir = '/usr/local/hal9000'
export_variables = {
  'CC': tooldir + 'hal9000-cc',
  'CXX': tooldir + 'hal9000-c++',
```

```
    'AR': tooldir + 'hal9000-ar',
    # The toolchain requires this env. var to work correctly:
    'CROSS_C_ROOT_PATH': '/build/sdks/hal9000/sdk',
  }
```

Next, edit "platforms/HAL9000/settings.gypi" to override settings specific to your platform. For example:

```
  'variables': {
    'oemcrypto_version': 9,
    'protobuf_config': 'source',
    'protobuf_source': '/path/to/protobuf-2.5.0',
  }, # end variables
```

Finally, run "./build.py HAL9000" to build for your platform.

Testing Against Your Platform's OEMCrypto

By default, the unit tests link against a reference implementation of OEMCrypto. To link the unit tests against your platform's OEMCrypto, edit your platform's "settings.gypi" file and set the variable "oemcrypto_lib". For example, in "platforms/HAL9000/settings.gypi":

```
  'variables': {
    'oemcrypto_version': 9,
    'oemcrypto_lib': '-lhal9000_oec',
  }, # end variables
```

Finally, run "./build.py HAL9000" to rebuild the tests.