Widevine CE CDM 16.4.0 Integration Guide

Document version 16.4.0

Revision History

Version	Date	Description	Author
2.8	2018-03-28	Updated the version number to 14.0.0. Clarified that the IV must be 16 bytes. Documented which compilers Widevine verifies the source code with. Updated BoringSSL revision. Replaced openssl_config with asm_target_arch. Documented that CE CDM 14.0.0 only supports OEMCrypto v14. Added explicit references to Mock OEMCrypto and called out that *Mock OEMCrypto is not suitable for production devices and is for testing only*. Updated the provisioning URLs.	John Bruce
2.9	2018-06-25	Removed Clang 3.8 & 3.9 from the list of internal compilers, added Clang 4.0 & 5.0. Made the host's responsibilites when calling Cdm::initialize() more explicit. Updated listed versions of Protobuf and BoringSSL. Updated documentation of what versions of Protobuf are supported.	John Bruce
2.10	2018-09-05	Clarified what CPU architectures Widevine tests on. Aded GCC 7.3.0 to the list of Widevine compilers. Clarified the expected behavior of ITimer. Updated listed version of BoringSSL.	John Bruce
15.0.0	2019-02-28	Revised document version numbers to reflect matching CE CDM version. Removed document versions from before CE CDM 14.0.0 from this table. Added mention of C++11 requirement. Updated included versions of Protobuf and BoringSSL. Emphasized that Widevine CE CDM works with a wide range of Protobuf versions. Renamed one of the overloads of load() to loadEmbeddedKeys(). Renamed "Mock OEMCrypto" to "Reference OEMCrypto" to reflect the name change that occurred in 14.2.0. Updated the service certificate	John Bruce

		documentation to reflect that it is now possible to install separate service certificates for the Provisioning Service and Licensing Service. Cleaned up some typos. Updated decrypt() documentation to explain the optional Session ID parameter.	
15.1.0	2019-03-29	Corrected the release date of document version 15.0.0. Revised the version numbers in the Purpose section. Removed POSIX requirement from Supported Compilers. Added Recoverable OEMCrypto Errors section. Updated provisioning information to reflect the removal of callback-based provisioning and the requirement of manual provisioning. Documented new methods for querying OEMCrypto's information.	John Bruce
15.2.0	2019-06-28	Emphasized that removeUsageTable() is almost never the right method to call. Added documentation for forceRemove(). Updated BoringSSL and Protobuf versions. Widevine now specifies what version of jsmn it requires.	John Bruce
15.3.0	2020-02-05	Added second overload to initialize(). Added second overload to create(). Noted that sessions can now be opened without a service certificate being installed. Noted the new error code kNeedsServiceCertificate. Clarified when privacy mode is recommended; Widevine no longer recommends privacy mode for most devices. Fixed the incorrect name of parseAndLoadServiceCertificateResponse().	John Bruce
16.2.0	2020-03-25	Further clarified when an app does and does not want to call removeUsageTable(). Removed confusing sentence from Storage section. "Recoverable OEMCrypto Errors" renamed just "Recoverable Errors" and expanded to include other errors with well-known recovery paths. Clarified whether build.py supports Python 3. Added a section about supported non-secure crypto libraries, now that OpenSSL is supported again. Clarified that *origins* are provisioned, not *devices*. Updated the section on decrypt() to reflect the new API in CE CDM 16. Updated the section on platform properties to remove outdated information and refer to the new platform_properties.gypi. Updated the section on platform assumptions, which referred to a nonexistent Lock implementation. Updated included BoringSSL revision. Updated the versions of GCC and Clang we use internally. Added updated guidance on reducing compiler compatibility problems.	John Bruce, Alex Lee
16.3.0	2020-06-26	Added GCC 9.3.0 to the list of tested compilers.	John Bruce
16.4.0	2020-10-08	Updated version number and range of supported OEMCrypto versions. Clarified that Usage Tables are an expected feature and will become mandatory in a future release.	John Bruce

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 16.4.0 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 will also need 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. CE CDM 16.4.0 is compatible with OEMCrypto v16.3 or v16.4. OEMCrypto v16.4 is recommended. OEMCrypto v16.4 is *required* on some devices.

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.

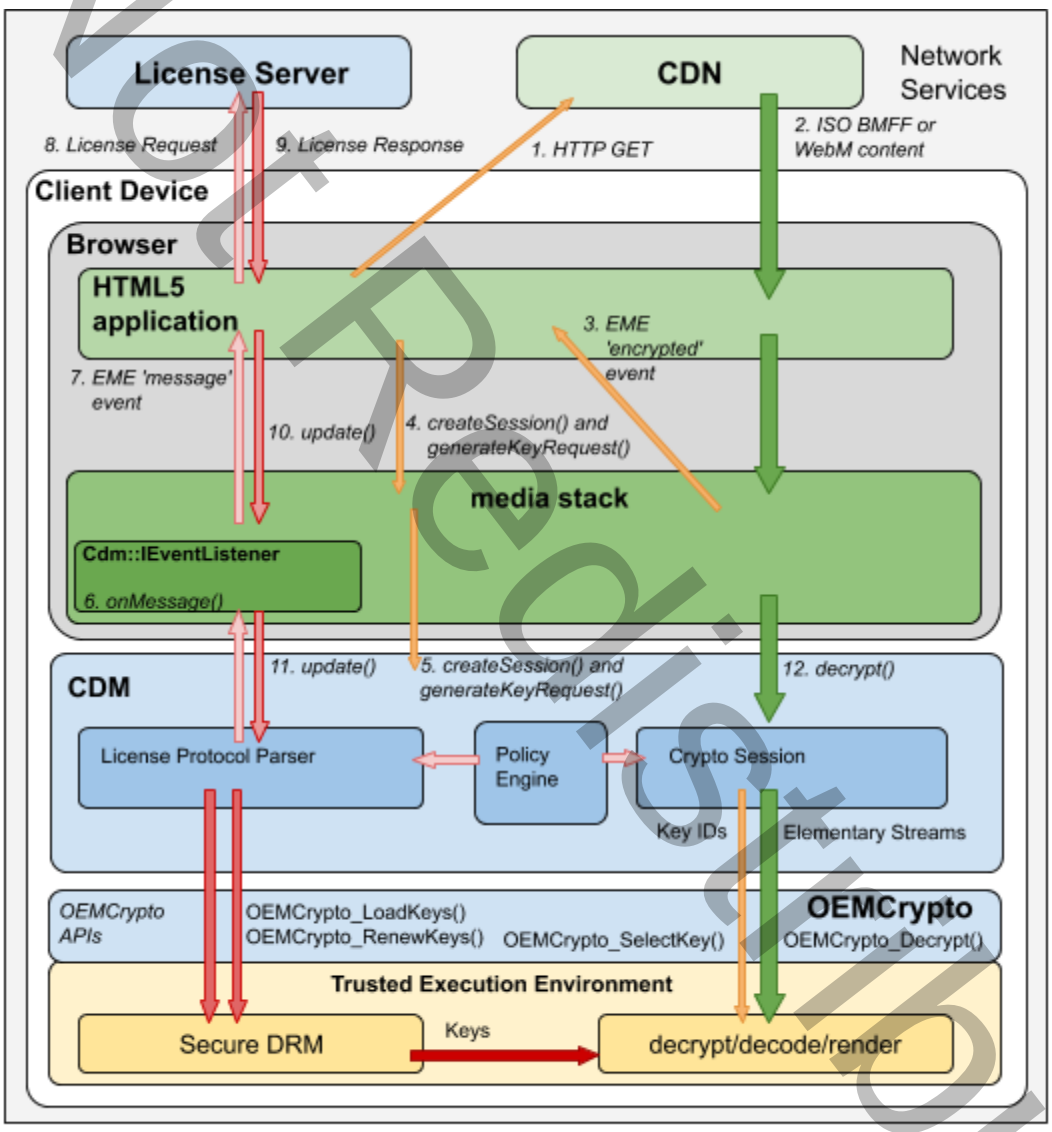


Figure 1. HTML5/EME Playback with Widevine CDM

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.

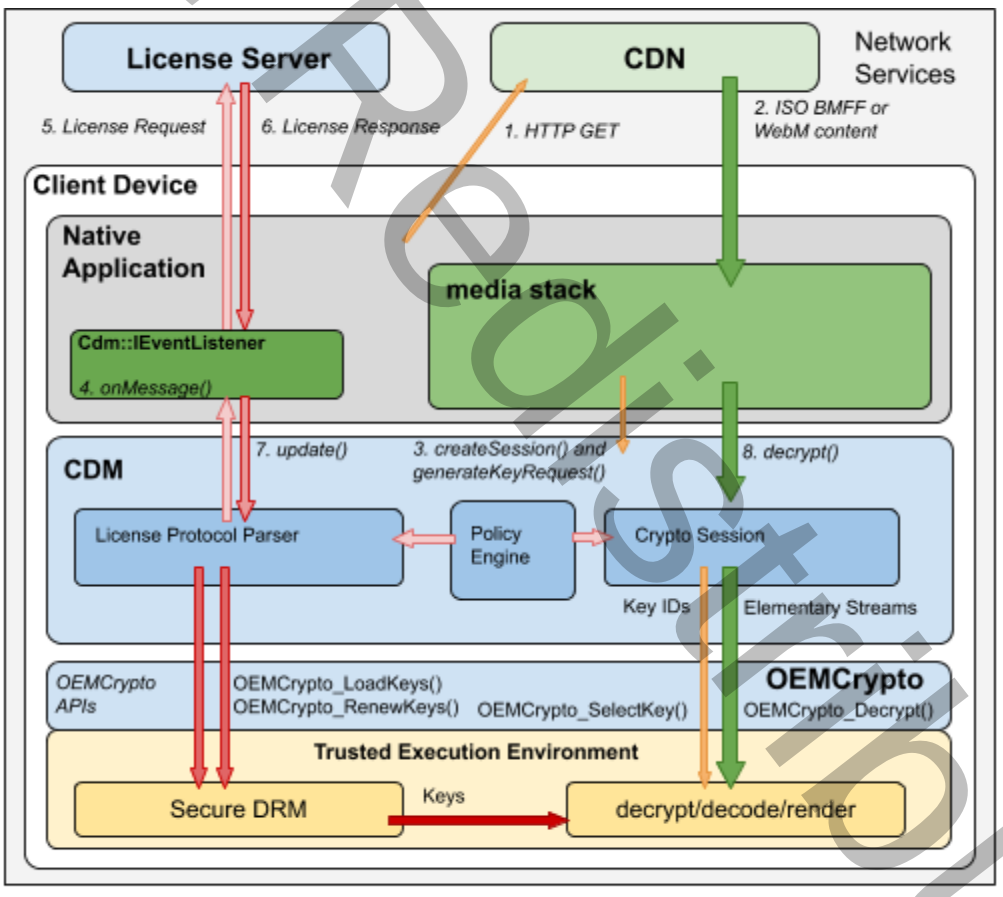


Figure 2. Native Application Playback with Widevine CDM

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. Widevine recommends using Python 3, but the script should be compatible with Python 2 as well. The minimum version of Python required is 2.7.

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

- gyp (open-source build automation tool, uses Python)
- protobuf (v3.8.0, used by the CDM as part of the license protocol. Note that the CDM does not require v3.8.0 specifically. See the section Protobuf for more details.)
- gTest and gMock (v1.8.0, for unit tests)
- jsmn (v1.0.0, open-source JSON parser, used for processing an HLS playlist)
- boringssl (rev: `0064c290d139b928e93a83900efe1367bc18dd03`)
- fuzz (currently in limited use, will be used for fuzz-testing the OEMCrypto and CDM interfaces)

The CE CDM will also require an implementation of a compatible version of OEMCrypto. For CE CDM 16.4.0, both OEMCrypto v16.3 and v16.4 are compatible. OEMCrypto v16.4 is recommended. OEMCrypto v16.4 is *required* if your device does not support usage tables.

We consider usage tables a standard feature, and not supporting them is an exception. Usage tables are used to save playback times for offline licenses and are needed to support several use cases that restrict playback duration. They are also used to support secure stops, which are used by several major content providers. If your device does not support usage tables but expects to support offline license, please talk to your Widevine TAM so that we may understand what is blocking this support. OEMCrypto is planning to make usage table support mandatory in a future release.

The CE CDM ships with a Reference OEMCrypto implementation in order to allow you to test the CE CDM code before your platform's OEMCrypto implementation is ready. However, **the Reference OEMCrypto is *not* suitable for production use**. It is not sufficiently secure and is not intended for use on shipping products. It is intended solely to enable testing of your CE CDM port before your own implementation of OEMCrypto is ready and does not absolve you of the need to implement OEMCrypto for your hardware.

The CE CDM also requires a non-secure crypto library in order to support Provisioning 3.0 and Privacy Mode. The CE CDM unit tests will also use this crypto library to do HTTPS. You must use one of the following crypto libraries:

- BoringSSL — A copy is included with the CE CDM and is used by default. See the above list of libraries in the third_party folder.

- OpenSSL — You will need to provide your own copy of OpenSSL. Version 1.1.0 or higher is required. You can switch from BoringSSL to OpenSSL by changing the value of the configuration variable `privacy_crypto_impl`.

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 also requires the caller 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. Before opening any sessions or generating any license requests, the caller should verify that the origin is provisioned by calling `isProvisioned()`. If the origin is not provisioned, the caller should generate a provisioning request with `getProvisioningRequest()`, exchange it with the provisioning server, and return the provisioning response to `handleProvisioningResponse()`.

Prior to CE CDM 15.0.0, the CE CDM would also automatically generate provisioning requests under certain circumstances and send them to `IEventListener::onDirectIndividualizationRequest()`, but this behavior is deprecated and that callback no longer exists.

Certificate Provisioning Message Formats

The request message supplied in response to `getProvisioningRequest()` 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/devicecerti
ficates/create?key=AIzaSyB-5OLKTx2iU5mko18DfdwK5611JIjbUhE&signed
Request=<request_data>
```

Note that there are two possible provisioning servers. The URL above is for the Production provisioning server. The Production provisioning server will only accept requests from devices that have been marked as “released” in the Widevine Integration Platform. During development and testing, you will likely be using the testing-only Reference OEMCrypto or using your own OEMCrypto on a device that has not been marked as “released” yet. These will not work with the Production provisioning server, and you should use the Staging provisioning server instead:

```
https://staging-www.sandbox.googleapis.com/certificateprovisionin
g/v1/devicecertificates/create?key=AIzaSyB-5OLKTx2iU5mko18DfdwK56
11JIjbUhE&signedRequest=<request_data>
```

Regardless of the server you use, the response message from the provisioning server will be in a similar format to the request, 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.

OEMCrypto can support either Keybox or certificate-based provisioning, but not both. The provisioning method is hard-coded into OEMCrypto.

Service Certificates

Service Certificates are only used when privacy mode is enabled. Service Certificates are associated with license and provisioning servers. The Service Certificate provides the public key for the service, and is used to encrypt parts of 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`.

There is also a global `IStorage` instance that must be passed into Cdm::initialize(). This is used for storing objects that are not associated with a specific origin. An example is the usage table header, which is persistent data that spans all per-origin usage tables.

In CE CDM version 3.5.0 and earlier, one could pass a NULL IStorage pointer into Cdm::create(). This would signal the CDM to use the default `IStorage` instance. This is no longer supported. If a non-per-origin IStorage object is to be used for a CDM instance, one can pass a copy of the pointer to the default IStorage object (i.e., the one passed in to Cdm::initialize()) to Cdm::create(). If this method is used, the default IStorage object must not be deleted until all the objects referring to it have been destroyed.

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.

Implementers only need to keep track of one timer per CDM instance. Each CDM will not request more than one timer be running simultaneously. It is an error if the CDM does this. If the CDM erroneously requests a new timer when one is already running, partners are recommended to cancel the existing timer and start the new timer.

*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.*

Managing Persistent Licenses

Acquiring Persisted Licenses

Persisted licenses are requested from license service when using a cdm message type of `kPersistentLicense`. Please refer to the `Cdm::createSession()` call for more information on creating persistent license sessions.

The output parameter `session_id` will be used to identify the persistent session in calls to all other CDM methods. The session_id will also be used to reference and manage licenses that have been persisted in client storage.

Upon calling the `Cdm::generateRequest()` API with the `kPersistentLicense` message type, a persistent license request will be generated.

LoadingPersisted Licenses

Licenses that have been acquired and stored in client storage can be reloaded using the session_id. The session_id is the output parameter identifying the session used to request the peristed license. Please refer to the `Cdm::load()` call for more information on loading persisted licenses.

Removing Persisted Licenses

Licenses that have been acquired and stored in client storage can be released and removed using the session_id. The session_id is the output parameter identifying the session used to request the peristed license. Please refer to the `Cdm::remove()` call for more information on releasing licenses.

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. Before any other method can be called, the caller must have received a successful return value from calling this function.

This function takes pointers to several interfaces that provide access to system functionality for storage, timers, and a clock. These are stored globally by the CDM library for future use. It is the responsibility of the caller to ensure that they outlive the CDM library, which likely means they must live for the lifetime of the program.

There are two overloads of the `initialize()` function. Most partners will want to use the first overload, which does not take a Sandbox ID parameter. Only partners whose OEMCrypto is using Sandbox IDs should use the second overload, which allows them to pass in their Sandbox ID.

`version`

Queries the Widevine CE CDM library's version string.

Note that this is *not* the version information for the underlying OEMCrypto integration. See `getOemCryptoBuildInfo()` below for that information.

`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 require a service certificate to encrypt the client identification in provisioning and licensing requests. Privacy mode complicates provisioning and licensing. As such, we recommend privacy mode be turned *off* for most CE devices. Privacy mode is only useful for web browsers, because they execute arbitrary, untrusted Javascript from the internet.

There are two overloads of the `create()` function. Most partners will want to use the first overload, which has fewer parameters. However, some partners have read-only storage that is pre-populated with certificates and licenses. This is currently only done for ATSC 3.0. Partners who are providing an `IStorage` that should never be written to should use the second overload and pass `true` for the `storage_is_read_only` parameter. Passing `false` for the `storage_is_read_only` parameter is identical to calling the first overload of `create()`.

`setServiceCertificate`

Allows the application to provide a service certificate to the CDM. This certificate is used to encrypt client identification information in provisioning and license requests as part of "privacy mode" (See `create()`). It is also needed to sign and verify the messages for certificate provisioning, as the certificate holds a provider_id string that is needed for constructing the provisioning request. When privacy mode is turned on, some methods will return the error `kNeedsServiceCertificate` until service certificates have been installed.

Depending on the value of the `role` parameter, the CDM may install the certificate for use with the provisioning service, the licensing service, or both. It is possible to install different service certificates for the provisioning and licensing services. Installing a certificate for one service does not remove the certificate installed for the other service.

If no service certificate is installed for the provisioning service, a default certificate for the Widevine provisioning service will be used. This is the correct service certificate for all provisioning requests unless a content provider is operating their own provisioning server.

There is no default licensing service certificate. If privacy mode is enabled, a licensing service certificate *must* be installed by the app to generate license requests.

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 **`parseAndLoadServiceCertificateResponse`** (see below) are provided to assist with this operation and can be used instead of **`setServiceCertificate`**.

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 **parseAndLoadServiceCertificateResponse** as discussed below.

`parseAndLoadServiceCertificateResponse`

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 **parseAndLoadServiceCertificateResponse** 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**)

Depending on the value of the `role` parameter, the CDM may install the certificate for use with the provisioning service, the licensing service, or both. It is possible to install different service certificates for the provisioning and licensing services. Installing a certificate for one service does not remove the certificate installed for the other service.

`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()`.

Prior to CE CDM 15.3.0, it was not possible to create a session if privacy mode was enabled and no service certificate had been installed for the licensing service. Starting in 15.3.0, it is possible to create sessions even when no service certificate is installed.

`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()`.

`getRobustnessLevel`

Retrieves the robustness level of the underlying OEMCrypto integration, either L1, L2, or L3. Most devices are L1 or L3.

Note that this function is *not* cryptographically secure and it should only be relied upon for informational purposes (e.g. determining which content to show in the UI) and not security purposes. (e.g. determining which content to allow the device to play) *Only* secure communication between OEMCrypto and the license service should be used to make security decisions.

`getResourceRatingTier`

Retrives the Resource Rating Tier reported by the underlying OEMCrypto integration. This information can be used to help determine what kind of content the device is capable of playing back. The full list of resource tiers can be found in the Widevine Modular DRM Security Integration Guide for Common Encryption.

`getOemCryptoBuildInfo`

Retrieves the build information for the underlying OEMCrypto integration. This may include version numbers, build dates, or other relevant information. The format of this string is up to the integrator, and applications wishing to use it will need to speak with device integrators directly.

`isProvisioned`

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

If this returns false, a provisioning exchange as described in Device Certificate Provisioning should be performed before opening sessions. Attempting to generate license requests or load persistent licenses *will* fail so long as the origin is not provisioned.

`getProvisioningRequest`

Generates a provisioning request. This message should be exchanged with the provisioning server as described in Certificate Provisioning Message Formats. The resulting response should be passed to `handleProvisioningResponse()`.

`handleProvisioningResponse`

After a successful provisioning exchange with the provisioning server as described in Certificate Provisioning Message Formats, the response should be passed to this method. If this returns a successful status code, then the device is now provisioned and `isProvisioned()` will return true.

`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.

`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.

`removeUsageTable`

Deletes all the usage table files from the device. *This affects all applications and origins on the device, not just the current one.* This is a dangerous procedure and is generally not desired by most apps. If you want to delete all of your app's usage records, you should call `deleteAllUsageRecords()`.

The only valid use for this function is to recover from a broken state where the CDM cannot even load the usage table.

load

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.

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()`.

loadEmbeddedKeys

(Prior to CE CDM 15.0.0, this function shared the name `load()` *with the function still known as* `load()`*.)*

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.

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`

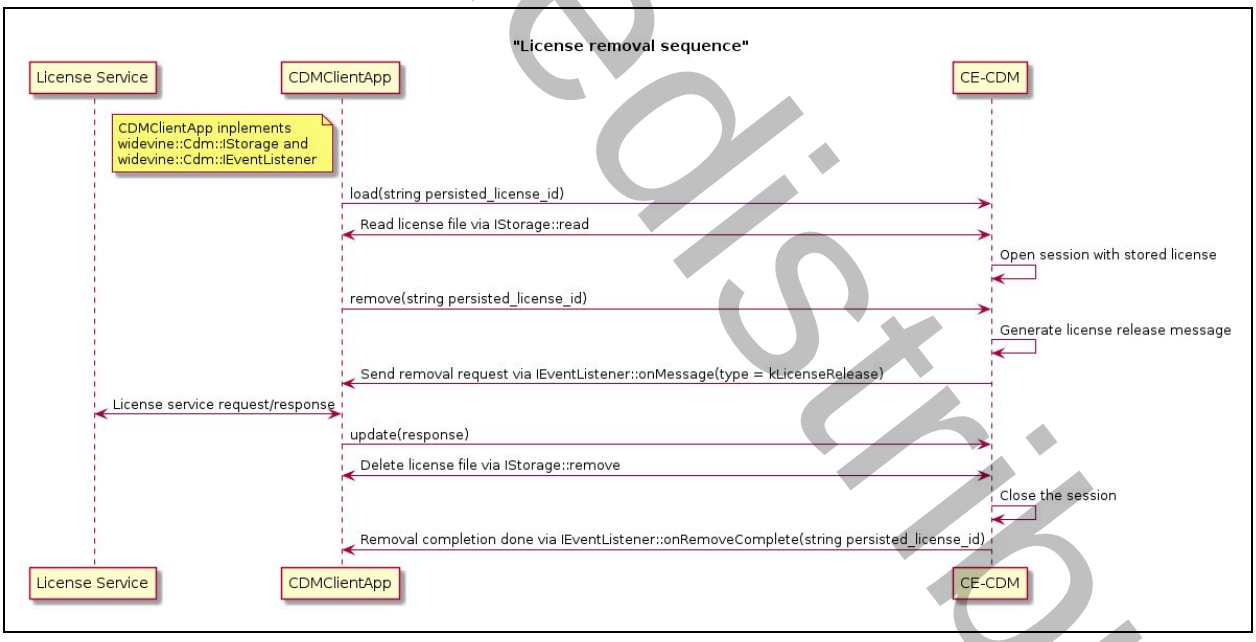
Used to close an active session and release temporary resources associated with it. If it is of type `kPersistentLicense`, the session will not be removed from storage. No release messages (type `kLicenseRelease`) will be generated.

`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()`.



forceRemove

Used to remove a persistent session without doing a server roundtrip. All usage information for the session is lost. This is generally not desired. Most applications should use `remove`.

decrypt

Used to decrypt content.

Starting with 16.2.0, the Widevine CDM supports only the ‘cenc’ and ‘cbcs’ schemas from the ISO-CENC 3.0 standard. This is different from previous releases, which supported all four ISO-CENC schemas. 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 `encryption_scheme` is `kAesCtr`, then the ‘cenc’ schema is used and it is an error to specify a `pattern` other than the default value of (0,0). If the `encryption_scheme` is `kAesCbc`, then the ‘cbcs’ schema is used and the `pattern` parameter will be respected. If the `pattern` parameter is left at its default value of (0,0) when using the ‘cbcs’ schema, then the CDM will assume ISO-CENC “full sample” encryption is in use and will treat the content as if the pattern (10,0) were specified.

It is possible to pass multiple samples to `decrypt()`. The Widevine CE CDM will try to pass as many samples to OEMCrypto in one call as you pass to `decrypt()`. It is therefore strongly recommended not to pass more samples to `decrypt()` than your OEMCrypto integration can handle in a single decryption call. This improves the throughput and efficiency of the decrypt process, as the CDM does not have to spend time breaking the decrypt request into smaller pieces until it finds a size that can be passed to OEMCrypto.

The IV must be 16 bytes long. If the content is using 8-byte IVs with an implied 8-byte counter, as recommended by the ISO-CENC standard, then it is the application’s responsibility to zero-pad the IV to 16 bytes, as specified by ISO-CENC.

Starting with Widevine CE CDM 15.0.0, it is possible to specify an explicit Session ID to use for decryption. If the specified Session ID does not have the correct key loaded into it, then `decrypt()` will return an error. Specifying a Session ID is optional. When no Session ID is specified, the CDM will find a session that has the correct key loaded and use that. If no session has the correct key loaded, `decrypt()` will return an error.

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.

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.

Recoverable Errors

It is generally up to the caller to determine how to handle any errors returned by the CE CDM. However, certain errors are considered "recoverable" in that there are specific known remedies that the caller may take in response to them.

kResourceContention

This error indicates that too many calls are accessing the OEMCrypto integration at the same time and have either exceeded its capacity or have tripped anti-abuse protections. The action that triggered this response should be retried after a delay, typically 1 second.

kSessionStateLost

This error indicates that the OEMCrypto integration has lost some state essential to the session. The current session is no longer valid and there is nothing the caller can do to recover it, however other sessions are still valid and the caller should be able to close and then recreate the current session to continue operations.

kSystemStateLost

This error indicates that the OEMCrypto integration has lost some essential internal state necessary for its continued operation. All DRM operations except total termination of the CDM are likely to fail. However, closing all CDMs and recreating them should reinitialize OEMCrypto and allow operations to continue.

kOutputTooLarge

This error indicates that the frame passed to decrypt is too large for some part of the decrypt/decode pipeline to handle. There is nothing the caller can do to change this, and the frame that was being decrypted should be skipped.

kNeedsDeviceCertificate

This error indicates that either

- the device is not provisioned and it needs to be provisioned
- the device has a provisioning but it is no longer valid and the device needs to be reprovisioned

Regardless of what function returned it or the existing provisioning status of the device, the correct response to this error is to provision the device.

Note that this error may be returned by the `Cdm::update()` function. In this case, it means the licensing server will not send licenses to the device until it upgrades its existing device certificate by reprovisioning the device.

kNeedsServiceCertificate

This error indicates that privacy mode is turned on but no service certificate was set before attempting to generate a message to exchange with the licensing service. The caller should do one of two things, depending on whether it has a service certificate cached already:

- If the caller already has a service certificate cached for the licensing service it wants to contact, it should pass the certificate to `Cdm::setServiceCertificate()`.
- If the caller does not have a certificate cached, it should generate a service certificate request with `Cdm::getServiceCertificateRequest()` and exchange it with the licensing service. The response should be passed into `Cdm::parseAndLoadServiceCertificateResponse()`. The caller can optionally cache the value returned from this function for future use with `Cdm::setServiceCertificate()`.

The caller does not need to wait to get this error before providing a service certificate. It is preferable for the caller to provide the service certificate before generating any licensing messages so that this error does not even occur.

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.

Supported Compilers

The Widevine CE CDM should compile on any compiler that supports the C++11 standard. However, Widevine only verifies the code ourselves with the following compilers and CPU architectures:

- GCC 4.8.4 x86-64 Linux
- GCC 9.3.0 x86-64 Linux
- Clang 5.0.2 x86-64 Linux
- Clang 6.0.1 x86-64 Linux
- Clang 8.0.1 x86-64 Linux

Due to subtle differences in their default build flags, other compilers may or may not successfully compile the Widevine CE CDM using the default build flags. The most common cause of incompatibilities is more- or less-stringent error-checking, which is almost always fixable by making small changes to your platform-specific build flags. The sample settings.gypi file turns warnings into errors by default, so if you copy this flag, differences in your compiler's warning settings may end up treated as errors. Widevine recommends *not* using this setting in your platform's settings.gypi file.

Compile-time Options and Configuration

The file "cdm/cdm.gyp" contains the main part of the build description. It depends on the file "cdm/platform_properties.gypi", which includes build options that we expect partners may need to override. The options are documented in the platform_properties.gypi file; please refer to the comments in that file for a description of their use. The defaults in platform_properties.gypi can be overridden with platform-specific values in a platform-specific build file. This will be covered in the "Porting" section below. You should not edit the defaults in platform_properties.gypi or cdm.gyp.

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.

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 cdm.gyp 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. The Widevine CE CDM ships with a copy of Protobuf v3.8.0 in “third_party/”, which we recommend using. However, the nature of your platform and the other applications you are compiling the CE CDM into may require you to use a different version of Protobuf. For those who need a different version of Protobuf, the Widevine CE CDM theoretically works with all versions of protobuf back to v2.6. However, we only verify it with the version of Protobuf included in the “third_party/” directory.

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. For example, 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': 14,
  'protobuf_config': 'source',
  'protobuf_source': '/path/to/protobuf-2.6.0',
}, # 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 testing-only implementation of OEMCrypto known as the Reference OEMCrypto. Before you can ship your device, you must link with your own implementation of OEMCrypto that is specific to your platform. 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': 14,
  'oemcrypto_lib': '-lhal9000_oec',
}, # end variables
```

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