Widevine CE CDM v3.0 Integration Guide

Document version 1.1

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

This document replaces the documents "Widevine Security Integration Guide for CENC: EME Supplement" and "CDM Porting Guide" that shipped with v2.x of the CE CDM.

Table of Contents

Purpose

This document gives an overview of the Widevine CE CDM 3.0 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 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.

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.

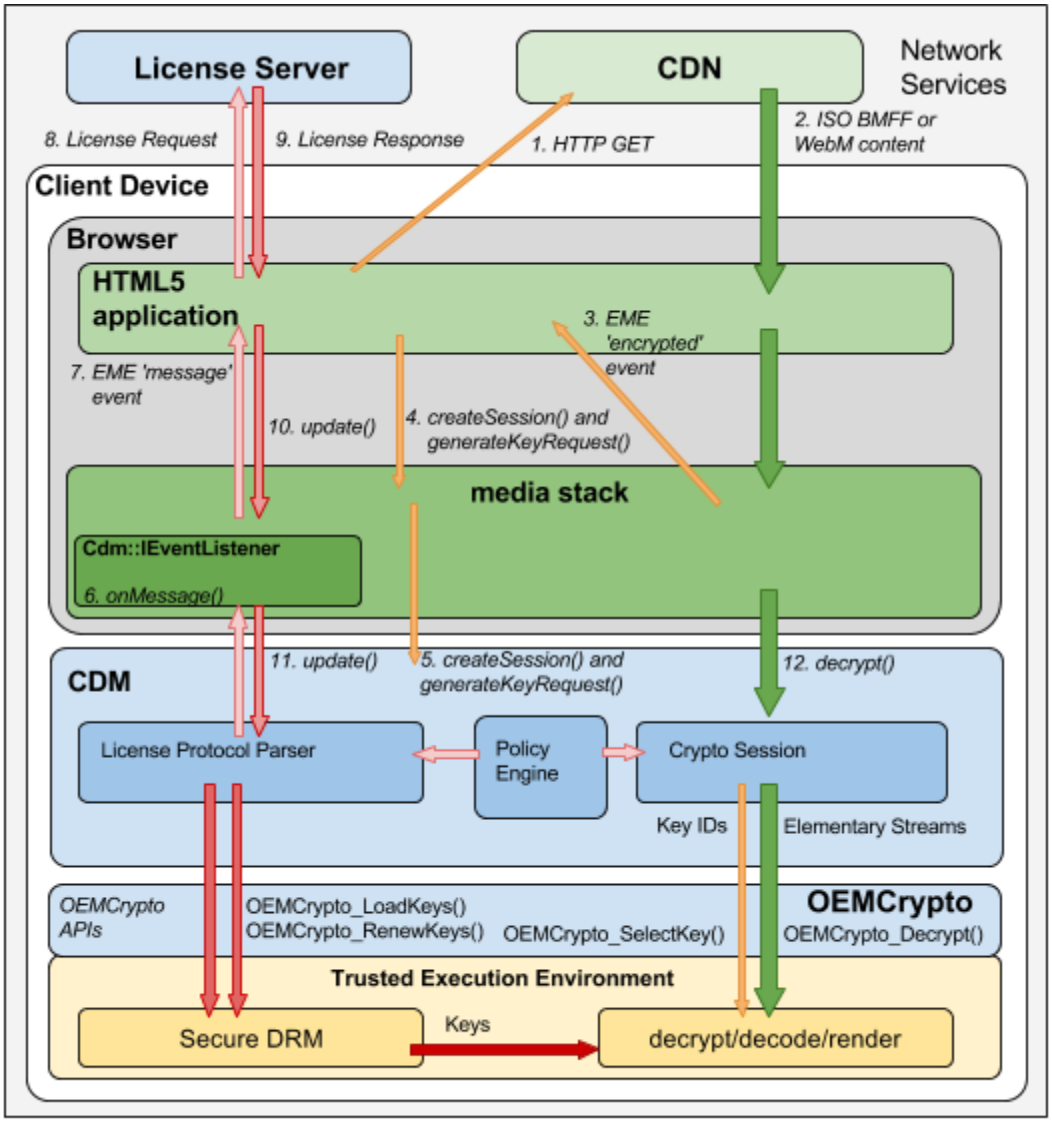

Figure 1. HTML5/EME Playback with Widevine CDM

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.

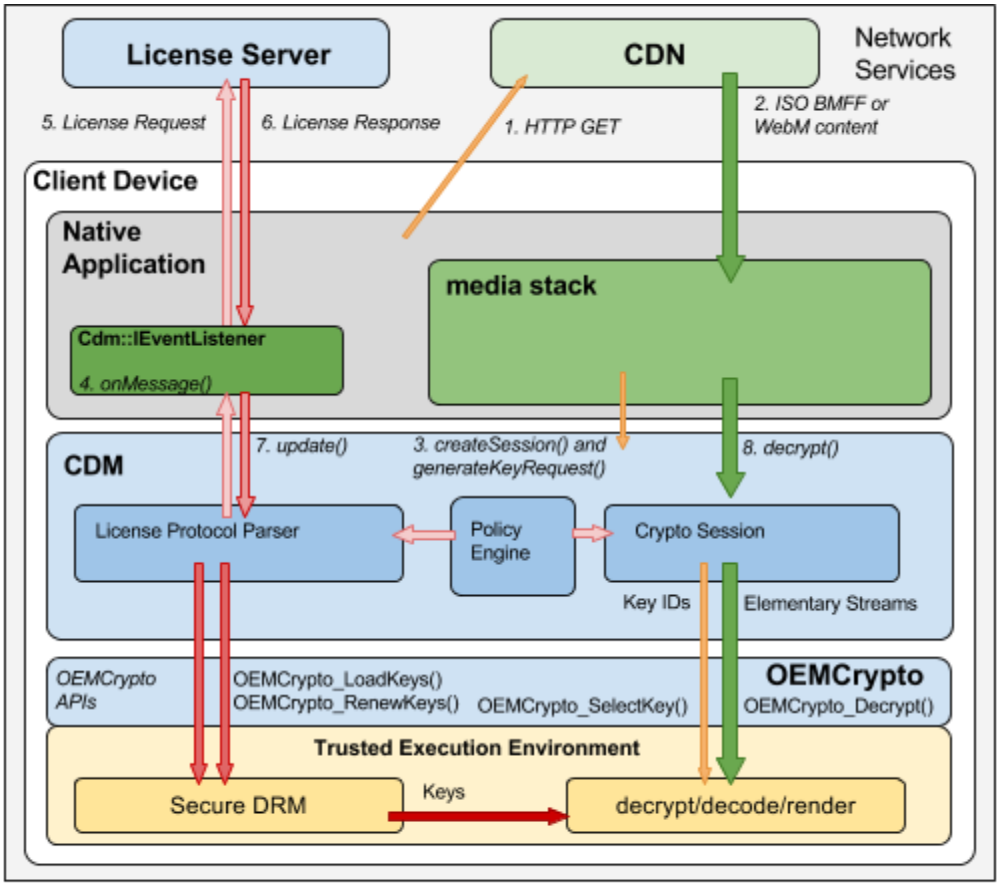

Figure 2. Native Application Playback with Widevine CDM

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 (must implement `IEventListener`)
- 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 either OEMCrypto or a private interface in the Trusted Execution Environment (TEE).

Build Requirements

The CE CDM requires a few things to build. There are a few prerequisites you must handle yourself:

- python2 (v2.7+, used by the build system)
- OpenSSL (v1.0.1g+, used by the CDM and the OEMCrypto reference implementation)
    - *NOTE: variants such as BoringSSL are also permitted.*

We provide a script which installs the rest for you. Run "./install_third_party.sh" to install these into the CDM source under the "third_party" folder:

- gyp (open-source build automation tool, uses python)
- protobuf (v2.5 specifically, used by the CDM as part of the license protocol)
- stringencoders (open-source base64 library in C)
- gTest and gMock (v1.6.0, for unit tests)

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 application to provision a device certificate at library initialization time. (See below.)

Device Certificate Provisioning

Starting with v3.0, the Widevine CE CDM requires a unique certificate for each device. If, at library initialization time, a device certificate is not available, the CDM will require the application to relay a message to the provisioning server to get a device certificate. The CDM will refuse to create sessions until this process has been completed. See the initialize() method for more details.

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.

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.

NOTE: Browsers or other multi-application systems should store data separately per-app or per-origin. See https://w3c.github.io/encrypted-media/#privacy-leakage 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 an `InvalidAccess` exception, we would return `kInvalidAccess` 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.

The library will fill in the `DeviceCertificateRequest` output parameter, which the caller must then examine. If the `needed` field is true, the caller must then provision a device certificate before using the library. Make an HTTP POST request to the URL given in `url` and pass the HTTP response to the `acceptReply()` method, then check the return value. See `DeviceCertificateRequest` in the header for more details.

`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.

`setServerCertificate`

Provides a server certificate. The certificate will be used to encrypt client identification information in license requests as part of "privacy mode". (See `create()`). May not be called if privacy mode is disabled in this CDM instance.

When privacy mode is enabled and `setServerCertificate()` is not called, a server certificate will be provisioned as part of the license request flow using the message type `kIndividualizationRequest`. This will result in one extra round trip before a license is acquired.

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

`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, this will be a server certificate request (type `kIndividualizationRequest`) instead of a license request (type `kLicenseRequest`). Note, however, that most applications will not need to pay attention to the

different message types and can simply relay to the license server what they are given in `onMessage()`.

`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 `kPersistent` 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()`.

`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.

`close`

Used to close an active session and release temporary resources associated with it. If it is of type `kPersistent`, 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, the `remove()` it again 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.

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.

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 v8, v9, and v10). 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. If you used "install_third_party.sh", the config.h produced by that script is appropriate for your host platform and may need adjustment for your target platform.

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. Just 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. If you used "install_third_party.sh", the config.h produced by that script is appropriate for your host platform. 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.