wolfEngine Documentation

2026-03-31

Contents

1 Introduction

The wolfCrypt Engine (wolfEngine) is an OpenSSL engine for the wolfCrypt and wolfCrypt FIPS cryptography libraries. wolfEngine provides an OpenSSL engine implementation, as a shared or static library, to allow applications currently using OpenSSL to leverage wolfCrypt cryptography for FIPS and non-FIPS use cases.

wolfEngine is structured as a separate standalone library which links against wolfSSL (libwolfssl) and OpenSSL. wolfEngine implements and exposes an **OpenSSL engine implementation** which wraps the wolfCrypt native APIinternally. A high-level diagram of wolfEngine and how it relates to applications and OpenSSL is displayed below in Figure 1.

For more details on the design and architecture of wolfEngine see the wolfEngine Design chapter.

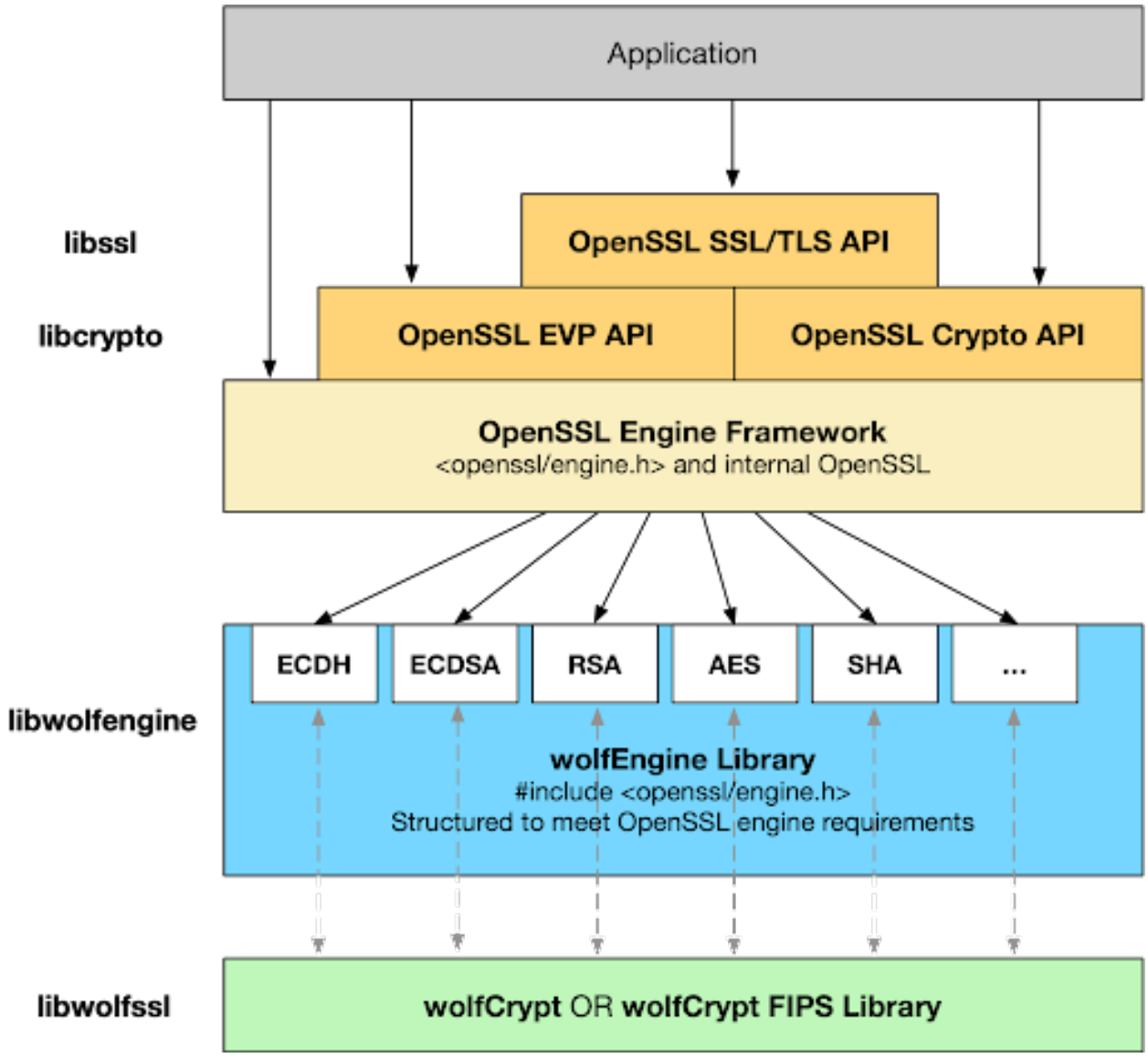

Figure 1: wolfEngine Overview

wolfEngine is compiled by default as a shared library called **libwolfengine** which can be dynamically registered at runtime by an application or OpenSSL through a config file. wolfEngine also provides an

entry point for applications to load the engine when compiled in a static build.

2 OpenSSL Version Compatability

wolfEngine has been tested against the following versions of OpenSSL. wolfEngine may work with other versions, but may require some modification or adjustment:

- OpenSSL 1.0.2h
- OpenSSL 1.1.1b

If you are interested in having wolfSSL add support to wolfEngine for other OpenSSL versions, please contact wolfSSL at facts@wolfssl.com.

3 Building wolfEngine

3.1 Getting wolfEngine Source Code

The most recent version of wolfEngine can be obtained directly from wolfSSL Inc. Contact facts@wolfssl.com for more information.

3.2 wolfEngine Package Structure

The general wolfEngine package is structured as follows:

```
certs/                          (Test certificates and keys, used with unit
   tests)
engine.conf               (Example OpenSSL config file using wolfEngine)
include/
    wolfengine/           (wolfEngine header files)
openssl_patches/
    1.0.2h/tests/         (patches for OpenSSL 1.0.2h test apps)
    1.1.1b/tests/         (patches for OpenSSL 1.1.1b test apps)
scripts/                       (wolfEngine test scripts)
src/                             (wolfEngine source files)
test/                            (wolfEngine test files)
user_settings.h          (EXAMPLE user_settings.h)
```

3.3 OpenSSL Version Caveats

Depending on the version of OpenSSL being used with wolfEngine, there are several algorithms support caveats, including:

- SHA-3 support is only available with OpenSSL versions 1.1.1+
- EC_KEY_METHOD is only available with OpenSSL versions 1.1.1+

3.4 Building on *nix

3.4.1 Building OpenSSL

A pre-installed version of OpenSSL may be used with wolfEngine (barring algorithm caveats above), or OpenSSL can be recompiled for use with wolfEngine. General instructions for compiling OpenSSL on *nix-like platforms will be similar to the following. For complete and comprehensive OpenSSL build instructions, reference the OpenSSL INSTALL file and documentation.

```
git clone https://github.com/openssl/openssl.git
cd openssl
./config no-fips -shared
make
sudo make install
```

3.4.2 Building wolfSSL

If using a FIPS-validated version of wolfSSL with wolfEngine, follow the build instructions provided with your specific FIPS validated source bundle and Security Policy. In addition to the correct "–enable-fips" configure option, wolfEngine will need wolfSSL to be compiled with " **WOLFSSL_PUBLIC_MP** " defined. For example, building the "wolfCrypt Linux FIPSv2" bundle on Linux:

```
cd wolfssl-X.X.X-commercial-fips-linuxv
./configure **--enable-fips=v2 CFLAGS”=-”DWOLFSSL_PUBLIC_MP**
make
./wolfcrypt/test/testwolfcrypt
< modify fips_test.c using verifyCore hash output from testwolfcrypt
>
make
./wolfcrypt/test/testwolfcrypt
< all algorithms should PASS >
sudo make install
```

To build non-FIPS wolfSSL for use with wolfEngine:

```
cd wolfssl-X.X.X

./configure --enable-cmac --enable-keygen --enable-sha --enable-des
--enable-aesctr --enable-aesccm --enable-x963kdf
CPPFLAGS="-DHAVE_AES_ECB -DWOLFSSL_AES_DIRECT -DWC_RSA_NO_PADDING
-DWOLFSSL_PUBLIC_MP -DECC_MIN_KEY_SZ=192 -DWOLFSSL_PSS_LONG_SALT
-DWOLFSSL_PSS_SALT_LEN_DISCOVER"

make
sudo make install
```

If cloning wolfSSL from GitHub, you will need to run the `autogen.sh` script before running `./configure`. This will generate the configure script:

```
./autogen.sh
```

3.4.3 Building wolfEngine

When building wolfEngine on Linux or other *nix-like systems, use the autoconf system. To configure and compile wolfEngine run the following two commands from the wolfEngine root directory:

```
./configure
make
```

If building wolfEngine from GitHub, run autogen.sh before running configure:

```
./autogen.sh
```

Any number of build options can be appended to ./configure. For a list of available build options, please reference the “Build Options” section below or run the following command to see a list of available build options to pass to the ./configure script:

```
./configure  --help
```

wolfEngine will use the system default OpenSSL library installation unless changed with the “–with-openssl” configure option:

```
./configure --with-openssl=/usr/local/ssl
```

The custom OpenSSL installation location may also need to be added to your library search path. On Linux, `LD_LIBRARY_PATH` is used:

```
export LD_LIBRARY_PATH=/usr/local/ssl:$LD_LIBRARY_PATH
```

To build then install wolfEngine, run:

```
make
make install
```

You may need superuser privileges to install, in which case precede the command with sudo:

```
sudo make install
```

To test the build, run the built-in tests from the root wolfEngine directory:

```
./test/unit.test
```

Or use autoconf to run the tests:

```
make check
```

If you get an error like `error while loading shared libraries: libssl.so.3` then the library cannot be found. Use the `LD_LIBRARY_PATH` environment variable as described in the section above.

3.5 Building on WinCE

For full wolfEngine compatibility, ensure you have the following flags in your `user_settings.h` file for wolfCrypt:

```
#define WOLFSSL_CMAC
#define WOLFSSL_KEY_GEN
#undef NO_SHA
#undef NO_DES
#define WOLFSSL_AES_COUNTER
#define HAVE_AESCCM
#define HAVE_AES_ECB
#define WOLFSSL_AES_DIRECT
#define WC_RSA_NO_PADDING
#define WOLFSSL_PUBLIC_MP
#define ECC_MIN_KEY_SZ=192
```

Add wolfEngine flags to your `user_settings.h` file depending on which algorithms and features you want to use. You can find a list of wolfEngine user settings flags in the `user_settings.h` file in wolfEngine's directory.

Build wcecompat, wolfCrypt and OpenSSL for Windows CE, and keep track of their paths.

In the wolfEngine directory, open the sources file and change the OpenSSL, wolfCrypt, and `user_settings.h` paths to the directories you are using. You will need to update the paths in the INCLUDES and TARGETLIBS sections.

Load the wolfEngine project in Visual Studio. Include either `bench.c`, or `unit.h` and `unit.c` depending on if you want to run the benchmark or unit tests.

Build the project, and you will end up with a wolfEngine.exe executable. You can run this executable with `--help` to see a full list of options. You may need to run it with the `--static` flag to use wolfEngine as a static engine.

3.6 Build Options (./configure Options)

The following are options which may be appended to the `./configure` script to customize how the wolfEngine library is built.

By default, wolfEngine only builds a shared library, with building of a static library disabled. This speeds up build times by a factor of two. Either mode can be explicitly disabled or enabled if desired.

Option	Default Value	Description
–enable-static	**Disabled**	Build static libraries
–enable-shared	Enabled	Build shared libraries
–enable-debug	**Disabled**	Enable wolfEngine debugging support
–enable-coverage	**Disabled**	Build to generate code coverage stats
–enable-usersettings	**Disabled**	Use your own user_settings.h and do not add Makefile CFLAGS
–enable-dynamic-engine	Enabled	Enable loading wolfEngine as a dynamic engine
–enable-singlethreaded	**Disabled**	Enable wolfEngine single threaded
–enable-digest	Enabled	Enable use of wc_Hash API for digesting data
–enable-sha	Enabled	Enable SHA-1
–enable-sha224	Enabled	Enable SHA2-224
–enable-sha256	Enabled	Enable SHA2-256
–enable-sha384	Enabled	Enable SHA2-384
–enable-sha512	Enabled	Enable SHA2-512
–enable-sha3	Enabled	Enable SHA3
–enable-sha3-224	Enabled	Enable SHA3-224
–enable-sha3-256	Enabled	Enable SHA3-256
–enable-sha3-384	Enabled	Enable SHA3-384
–enable-sha3-512	Enabled	Enable SHA3-512
–enable-cmac	Enabled	Enable CMAC
–enable-hmac	Enabled	Enable HMAC
–enable-des3cbc	Enabled	Enable 3DES-CBC
–enable-aesecb	Enabled	Enable AES-ECB
–enable-aescbc	Enabled	Enable AES-CBC
–enable-aesctr	Enabled	Enable AES-CTR
–enable-aesgcm	**Disabled**	Enable AES-GCM
–enable-aesccm	**Disabled**	Enable AES-CCM
–enable-rand	Enabled	Enable RAND
–enable-rsa	Enabled	Enable RSA
–enable-dh	Enabled	Enable DH
–enable-evp-pkey	Enabled	Enable EVP_PKEY APIs
–enable-ecc	Enabled	Enable ECC
–enable-ec-key	Enabled	Enable ECC using EC_KEY
–enable-ecdsa	Enabled	Enable ECDSA
–enable-ecdh	Enabled	Enable ECDH
–enable-eckg	Enabled	Enable EC Key Generation
–enable-p192	Enabled	Enable EC Curve P-192
–enable-p224	Enabled	Enable EC Curve P-224
–enable-p256	Enabled	Enable EC Curve P-256
–enable-p384	Enabled	Enable EC Curve P-384
–enable-p521	Enabled	Enable EC Curve P-521
–with-openssl=DIR		OpenSSL installation location to link against. If not set, use the system default library and include paths.
–enable-openssh	**Disabled**	Enables use with openssh

3.7 Build Defines

wolfEngine exposes several preprocessor defines that allow users to configure how wolfEngine is built. These are described in the table below.

Define	Description
WOLFENGINE_DEBUG	Build wolfEngine with debug symbols, optimization level, and debug logging.
WE_NO_DYNAMIC_ENGINE	Do not build wolfEngine with dynamic engine support. Dynamic engines are ones that can be loaded into OpenSSL at runtime.
WE_SINGLE_THREADED	Build wolfEngine in single-threaded mode. This removes the need for locking around global resources used internally.
WE_USE_HASH	Enable digest algorithms using the wc_Hash API.
WE_HAVE_SHA1	Enable SHA-1 digest algorithm.
WE_HAVE_SHA224	Enable SHA-2 digest algorithm with digest size 224.
WE_HAVE_SHA256	Enable SHA-2 digest algorithm with digest size 256.
WE_HAVE_SHA384	Enable SHA-2 digest algorithm with digest size 384.
WE_HAVE_SHA512	Enable SHA-2 digest algorithm with digest size 512.
WE_SHA1_DIRECT	Enable the SHA-1 digest algorithm using the wc_Sha API. Incompatible with WE_USE_HASH.
WE_SHA224_DIRECT	Enable the SHA-2 224 digest algorithm using the wc_Sha224 API. Incompatible with WE_USE_HASH.

Define	Description
WE_SHA256_DIRECT	Enable the SHA-2 256 digest algorithm using the wc_Sha256 API. Incompatible with WE_USE_HASH.
WE_HAVE_SHA3_224	Enable SHA-3 digest algorithm with digest size 224. Not available in OpenSSL 1.0.2.
WE_HAVE_SHA3_256	Enable SHA-3 digest algorithm with digest size 256. Not available in OpenSSL 1.0.2.
WE_HAVE_SHA3_384	Enable SHA-3 digest algorithm with digest size 384. Not available in OpenSSL 1.0.2.
WE_HAVE_SHA3_512	Enable SHA-3 digest algorithm with digest size 512. Not available in OpenSSL 1.0.2.
WE_HAVE_EVP_PKEY	Enable functionality that uses the EVP_PKEY API. This includes things like RSA, DH, etc.
WE_HAVE_CMAC	Enable CMAC algorithm.
WE_HAVE_HMAC	Enable HMAC algorithm.
WE_HAVE_DES3CBC	Enable DES3-CBC algorithm.
WE_HAVE_AESECB	Enable AES algorithm with ECB mode.
WE_HAVE_AESCBC	Enable AES algorithm with CBC mode.
WE_HAVE_AESCTR	Enable AES algorithm with countee mode.
WE_HAVE_AESGCM	Enable AES algorithm with GCM mode.
WE_HAVE_AESCCM	Enable AES algorithm with CCM mode.
WE_HAVE_RANDOM	Enable wolfCrypt random implementation.
WE_HAVE_RSA	Enable RSA operations (e.g. sign, verify, key generation, etc.).

Define	Description
WE_HAVE_DH	Enable Diffie-Hellman operations (e.g. key generation, shared secret computation, etc.).
WE_HAVE_ECC	Enable support for elliptic curve cryptography.
WE_HAVE_EC_KEY	Enable support for EC_KEY_METHOD. Not available in OpenSSL 1.0.2.
WE_HAVE_ECDSA	Enable ECDSA algorithm.
WE_HAVE_ECDH	Enable EC Diffie-Hellman operations.
WE_HAVE_ECKEYGEN	Enable EC key generation.
WE_HAVE_EC_P192	Enable EC curve P192.
WE_HAVE_EC_P224	Enable EC curve P224.
WE_HAVE_EC_P256	Enable EC curve P256.
WE_HAVE_EC_P384	Enable EC curve P384.
WE_HAVE_EC_P512	Enable EC curve P512.
WE_HAVE_DIGEST	Compile code in benchmark program and unit tests for use with digest algorithms.
WOLFENGINE_USER_SETTINGS	Read user-specified defines from user_settings.h.

4 FIPS 140-3 Support

wolfEngine has been designed to work with FIPS 140-3 validated versions of wolfCrypt when compiled against a FIPS-validated version of wolfCrypt. This usage scenario requires a properly licensed and validated version of wolfCrypt, as obtained from wolfSSL Inc.

Note that wolfCrypt FIPS libraries cannot be “switched” into non-FIPS mode. wolfCrypt FIPS and regular wolfCrypt are two separate source code packages.

When wolfEngine is compiled to use wolfCrypt FIPS, it will only include support and register engine callbacks for FIPS-validated algorithms, modes, and key sizes. If OpenSSL based applications call non-FIPS validated algorithms, execution may not enter wolfEngine and could be handled by the default OpenSSL engine or other registered engine providers, based on the OpenSSL configuration.

NOTE : If targeting FIPS compliance,and non-wolfCrypt FIPS algorithms are called from a different engine, those algorithms are outside the scope of wolfEngine and wolfCrypt FIPS and may not be FIPS validated.

For more information on using wolfCrypt FIPS (140-2 / 140-3), contact wolfSSL at facts@wolfssl.com.

5 Engine Control Commands

wolfEngine exposes several engine control commands allowing applications to change the behavior of wolfEngine or adjust internal settings. The following control commands are currently supported:

Control Command	Description	Value
enable_debug	Enable wolfEngine debug logging	Numeric: 1 = enable, 0 = disable. See Chapter 6 for more details on debug logging.
log_level	Set wolfEngine logging level	Numeric: Bitmask of logging levels from wolfEngine_LogType in “include/-wolfengine/we_logging.h”. See Chapter 6 for more details on debug logging.
log_components	Set components to be logged by wolfEngine.	Numeric: Bitmask of components from wolfEngine_LogComponents in “include/-wolfengine/we_logging.h. See Chapter 6 for more details on debug logging.
set_logging_cb	Set wolfEngine logging callback	Function pointer to function used to print log messages. Function must match the wolfEngine_Logging_cb prototype from we_logging.h. See Chapter 6 for more details on debug logging.

Engine control commands can be set using `OpenSSL’s ENGINE_ctrl_cmd()` API. For example, to enable debug logging:

```
int ret = 0;
ret = ENGINE_ctrl_cmd(e, “”enable_debug, 1, NULL, NULL, 0);
if (ret != 1) {
    printf“(Failed to enable debug logging\”n);
}
```

Some control commands can also be set through an OpenSSL config file. More documentation on usage of OpenSSL engine control commands can be found in the OpenSSL man pages:

https://www.openssl.org/docs/man1.0.2/man3/engine.html

https://www.openssl.org/docs/man1.1.1/man3/ENGINE_ctrl_cmd.html

https://www.openssl.org/docs/man1.1.1/man3/ENGINE_ctrl_cmd_string.html

6 Logging

wolfEngine supports output of log messages for informative and debug purposes. To enable debug logging, wolfEngine must first be compiled with debug support enabled. If using Autoconf, this is done using the `--enable-debug` option to `./configure`:

```
./configure --enable-debug
```

If not using Autoconf/configure, define `WOLFENGINE_DEBUG` when compiling the wolfEngine library.

6.1 Enable / Disable Debug Logging

Once debug support has been compiled into the library, debugging must be enabled at runtime using the wolfEngine control commands specified in Section 5. To enable debug logging, use the `enable_debug` control command with the value of "1" to enable logging or "0" to disable logging. To enable logging using the `ENGINE_ctrl_cmd()` API:

```
int ret = 0;
ret = ENGINE_ctrl_cmd(e, ""enable_debug, 1, NULL, NULL, 0);
if (ret != 1) {
    printf"(Failed to enable debug logging\"n);
}
```

If wolfEngine has not been compiled with debug support enabled, an attempt to set `enable_debug` with `ENGINE_ctrl_cmd()` will return failure (0).

6.2 Controlling Logging Levels

wolfEngine supports the following logging levels. These are defined in the "include/wolfengine/we_logging.h" header file as part of the wolfEngine_LogType enum:

Log Enum	Description	Log Enum Value
WE_LOG_ERROR	Logs errors	0x0001
WE_LOG_ENTER	Logs when entering functions	0x0002
WE_LOG_LEAVE	Logs when leaving functions	0x0004
WE_LOG_INFO	Logs informative messages	0x0008
WE_LOG_VERBOSE	Verbose logs, including encrypted/decrypted/digested data	0x0010
WE_LOG_LEVEL_DEFAULT	Default log level, all except verbose level	WE_LOG_ERROR | WE_LOG_ENTER | WE_LOG_LEAVE | WE_LOG_INFO
WE_LOG_LEVEL_ALL	All log levels are enabled	WE_LOG_ENTER | WE_LOG_LEAVE | WE_LOG_INFO | WE_LOG_VERBOSE
WE_LOG_ERROR		

The default wolfEngine logging level includes `WE_LOG_ERROR`, `WE_LOG_ENTER`, `WE_LOG_LEAVE`, and `WE_LOG_INFO`. This includes all log levels except verbose logs (`WE_LOG_VERBOSE`).

Log levels can be controlled using the "**log_level**" engine control command at runtime, either through the `ENGINE_ctrl_cmd()` API or OpenSSL configuration file settings. For example, to turn on only error and informative logs using the "log_level" control command, an application would call:

```
#include <wolfengine/we_logging.h>

ret = ENGINE_ctrl_cmd(e, “”log_level, WE_LOG_ERROR | WE_LOG_INFO,
NULL, NULL, 0);
if (ret != 1) {
    printf“(Failed to set logging level\”n);
}
```

6.3 Controlling Component Logging

wolfEngine allows logging on a per-component basis. Components are defined in the wolfEngine_LogComponents enum in `include/wolfengine/we_logging.h`:

Log Component Enum	Description	Component Enum Value
WE_LOG_RNG	Random number generation	0x0001
WE_LOG_DIGEST	Digests (SHA-1/2/3)	0x0002
WE_LOG_MAC	MAC functions (HMAC, CMAC)	0x0004
WE_LOG_CIPHER	Ciphers (AES, 3DES)	0x0008
WE_LOG_PK	Public Key Algorithms (RSA, ECC)	0x0010
WE_LOG_KE	Key Agreement Algorithms (DH, ECDH)	0x0020
WE_LOG_ENGINE	All engine specific logs	0x0040
WE_LOG_COMPONENTS_ALL	Log all components	WE_LOG_RNG | WE_LOG_DIGEST | WE_LOG_MAC | WE_LOG_CIPHER | WE_LOG_PK | WE_LOG_KE | WE_LOG_ENGINE
WE_LOG_COMPONENTS_DEFAULT	Default components logged (all).	WE_LOG_COMPONENTS_ALL

The default wolfEngine logging configuration logs all components (`WE_LOG_COMPONENTS_DEFAULT`).

Components logged can be controlled using the “ **log_components** ” engine control command at runtime, either through the `ENGINE_ctrl_cmd()` API or OpenSSL configuration file settings. For example, to turn on only logging only for the Digest and Cipher algorithms:

```
#include <wolfengine/we_logging.h>

ret = ENGINE_ctrl_cmd(e, “ **log_components** ”, WE_LOG_DIGEST | WE_LOG_CIPHER
    ,
NULL, NULL, 0);
if (ret != 1) {
    printf“(Failed to set log components\”n);
}
```


6.4 Setting a Custom Logging Callback

By default wolfEngine outputs debug log messages using **fprintf()** to **stderr**.

Applications that want to have more control over how or where log messages are output can write and register a custom logging callback with wolfEngine. The logging callback should match the prototype of wolfEngine_Logging_cb in `include/wolfengine/we_logging.h`:

```
/**
* wolfEngine logging callback.
* logLevel - [IN] - Log level of message
* component - [IN] - Component that log message is coming from
* logMessage - [IN] - Log message
*/
typedef void (* **wolfEngine_Logging_cb** )(const int logLevel,
const int component,
const char *const logMessage);
```

The callback can then be registered with wolfEngine using the " **set_logging_cb** " engine control command. For example, to use the `ENGINE_ctrl_cmd()` API to set a custom logging callback:

```
void **customLogCallback** (const int logLevel, const int component,
const char* const logMessage)
{
    (void)logLevel;
    (void)component;
    fprintf(stderr, "wolfEngine log message: %d\"n, logMessage);
}

int **main** (void)
{
    int ret;
    ENGINE* e;
...
    ret = ENGINE_ctrl_cmd(e, " **set_logging_cb** ", 0, NULL,
    (void(*)(void))my_Logging_cb, 0);
    if (ret != 1) {
        /* failed to set logging callback */
    }
...
}
```

7 Portability

wolfEngine has been designed to leverage the portability of the associated wolfCrypt and OpenSSL libraries.

7.1 Threading

wolfEngine is thread safe and uses mutex locking mechanisms from wolfCrypt (`wc_LockMutex()`, `wc_UnLockMutex()`) where necessary. wolfCrypt has mutex operations abstracted for supported platforms.

7.2 Dynamic Memory Usage

wolfEngine uses OpenSSL's memory allocation functions to remain consistent with OpenSSL behavior. Allocation functions used internally to wolfEngine include `OPENSSL_malloc()`, `OPENSSL_free()`, `OPENSSL_zalloc()`, and `OPENSSL_realloc()`.

7.3 Logging

wolfEngine logs by default to stderr via `fprintf()`. Applications can override this by registering a custom logging function (see Chapter 6).

Additional macros that may be defined when compiling wolfEngine to adjust logging behavior include:

WOLFENGINE_USER_LOG - Macro that defines the nameof function for log output. Users can define this to a custom log function to be used in place of fprintf.

WOLFENGINE_LOG_PRINTF - Define that toggles the usageof fprintf (to stderr) to use printf (to stdout) instead. Not applicable if using WOLFENGINE_USER_LOG or custom logging callback.

8 Loading wolfEngine

8.1 Configuring OpenSSL to Enable Engine Usage

For documentation on how applications use and consume OpenSSL engines, refer to the OpenSSL documentation:

OpenSSL 1.0.2 OpenSSL 1.1.1

There are several ways an application can choose to consume, register, and configure engine usage. In the simplest usage, to load and register all ENGINE implementations bundled with OpenSSL an application would need to call the following (taken from the above OpenSSL documentation):

```
/* For OpenSSL 1.0.2, need to make the “”dynamic ENGINE available */
ENGINE_load_dynamic();

/* Load all bundled ENGINEs into memory and make them visible */
ENGINE_load_builtin_engines();
/* Register all of them for every algorithm they collectively implement */
ENGINE_register_all_complete();
```

At this point, if the application is configured to read/use an OpenSSL config file, additional engine setup steps can be done there. For OpenSSL config documentation, reference the OpenSSL documentation:

OpenSSL 1.0.2 OpenSSL 1.1.1

For example, an application can read and consume the default OpenSSL config file (openssl.cnf) or config as set by OPENSSL_CONF environment variable, and default [openssl_conf] section by calling:

```
OPENSSL_config(NULL);
```

Alternatively to using an OpenSSL config file, applications can explicitly initialize and register wolfEngine using the desired ENGINE_* APIs. As one example, initializing wolfEngine and registering for all algorithms could be done using:

```
ENGINE* e = NULL;

e = ENGINE_by_id“(”wolfengine);
if (e == NULL) {
printf“(Failed to find wolfEngine\”n);
/* error */
}
ENGINE_set_default(e, ENGINE_METHOD_ALL);

/* normal application execution / behavior */

ENGINE_finish(e);
ENGINE_cleanup();
```

8.2 Loading wolfEngine from an OpenSSL Configuration File

wolfEngine can be loaded from an OpenSSL config file if an application using OpenSSL is set up to process a config file. An example of how the wolfEngine library may be added to a config file is below. The [wolfssl_section] could be modified to set engine control commands (enable_debug) if needed.

```
openssl_conf = openssl_init
```

```
[openssl_init]
engines = engine_section

[engine_section]
wolfSSL = wolfssl_section

[wolfssl_section]
# If using OpenSSL <= 1.0.2, change engine_id to wolfengine
(drop the "lib").
engine_id = libwolfengine
# dynamic_path = .libs/libwolfengine.so
init = 1
# Use wolfEngine as the default for all algorithms it provides.
default_algorithms = ALL
# Only enable when debugging application - produces large
amounts of output.
# enable_debug = 1
```

8.3 wolfEngine Static Entrypoint

When wolfEngine is used as a static library, applications can call the following entry point to load wolfEngine:

```
#include <wolfengine/we_wolfengine.h>
ENGINE_load_wolfengine();
```

9 wolfEngine Design

wolfEngine is composed of the following source files, all located under the “src” subdirectory of the wolfEngine package.

Source File	Description
we_wolfengine.c	Contains library entry points. Calls OpenSSL IMPLEMENT_DYNAMIC_BIND_FN for dynamic loading of the library using the OpenSSL engine framework. Also includes static entry points when compiled and used as a static library.
we_internal.c	Includes wolfengine_bind() function, which handles registration of engine algorithm callbacks. Also includes other wolfengine internal functionality.
we_logging.c	wolfEngine logging framework and function implementations.
we_openssl_bc.c	wolfEngine OpenSSL binary compatibility abstraction layer, used for supporting wolfEngine across multiple OpenSSL versions.
we_aes_block.c	wolfEngine AES-ECB and AES-CBC implementation.
we_aes_cbc_hmac.c	wolfEngine AES-CBC-HMAC implementation.
we_aes_ccm.c	wolfEngine AES-CCM implementation.
we_aes_ctr.c	wolfEngine AES-CTR implementation.
we_aes_gcm.c	wolfEngine AES-GCM implementation.
we_des3_cbc.c	wolfEngine 3DES-CBC implementation.
we_dh.c	wolfEngine DH implementation.
we_digest.c	wolfEngine message digest implementations (SHA-1, SHA-2, SHA-3).
we_ecc.c	wolfEngine ECDSA and ECDH implementation.
we_mac.c	wolfEngine HMAC and CMAC implementations.
we_random.c	wolfEngine RAND implementation.
we_rsa.c	wolfEngine RSA implementation.
we_tls_prf.c	wolfEngine TLS 1.0 PRF implementation.

General wolfEngine architecture is as follows, showing both dynamic and static entry points:

9.1 wolfEngine Entry Points

The main entry points into the wolfEngine library are either **wolfengine_bind** () or **ENGINE_load_wolfengine** (). wolfengine_bind() is called automatically by OpenSSL if wolfEngine has been loaded dynamically. ENGINE_load_wolfengine() is the entry point applications must call if wolfEngine has been built and used statically instead of dynamically.

9.2 wolfEngine Algorithm Callback Registration

wolfEngine registers algorithm structures and callbacks with the OpenSSL engine framework for all supported components of wolfCrypt FIPS. This registration happens inside `wolfengine_bind()` in `we_internal.c`. `wolfengine_bind()` receives an ENGINE structure pointer representing the wolfEngine engine. Individual algorithm/component callbacks or structures are then registered with that ENGINE structure using the appropriate API from <openssl/engine.h>.

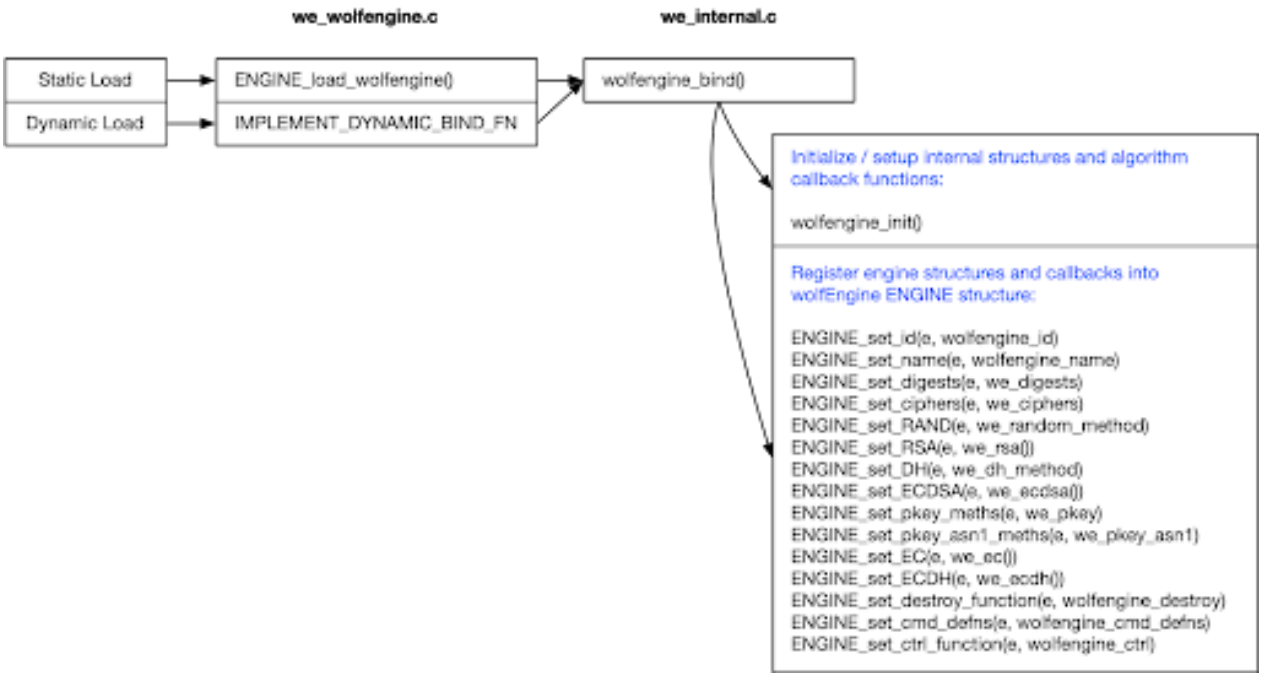

Figure 2: wolfEngine Architecture

These API calls include the following:

```
ENGINE_set_id(e, wolfengine_id)
ENGINE_set_name(e, wolfengine_name)
ENGINE_set_digests(e, we_digests)
ENGINE_set_ciphers(e, we_ciphers)
ENGINE_set_RAND(e, we_random_method)
ENGINE_set_RSA(e, we_rsa())
ENGINE_set_DH(e, we_dh_method)
ENGINE_set_ECDSA(e, we_ecdsa())
ENGINE_set_pkey_meths(e, we_pkey)
ENGINE_set_pkey_asn1_meths(e, we_pkey_asn1)
ENGINE_set_EC(e, we_ec())
ENGINE_set_ECDH(e, we_ecdh())
ENGINE_set_destroy_function(e, wolfengine_destroy)
ENGINE_set_cmd_defns(e, wolfengine_cmd_defns)
ENGINE_set_ctrl_function(e, wolfengine_ctrl)
```

Each algorithm/component callback function or structure used in the above calls (ex: we_digests, we_ciphers, etc) are implemented in either `we_internal.c` or in the respective algorithm source file.

10 Notes on Open Source Integration

wolfEngine conforms to the general OpenSSL engine framework and architecture. As such, it can be leveraged from any OpenSSL-consuming application that correctly loads and initializes engines and wolfEngine through OpenSSL configuration file or programmatically via ENGINE API calls.

wolfSSL has tested wolfEngine with several open source projects. This chapter contains notes and tips on wolfEngine integration. This chapter is not comprehensive of all open source project support with wolfEngine, and will be expanded upon as wolfSSL or the community reports testing and using wolfEngine with additional open source projects.

10.1 cURL

cURL is already set up to leverage an OpenSSL config file. To leverage wolfEngine:

1. Add wolfEngine engine information to your OpenSSL config file
2. If needed, set OPENSSL_CONF environment variable to point to your OpenSSL config file:

```
$ export OPENSSL_CONF=/path/to/openssl.cnf
```

3. Set OPENSSL_ENGINES environment variable to point to location of wolfEngine shared library file:

```
$ export OPENSSL_ENGINES=/path/to/wolfengine/library/dir
```

10.2 stunnel

stunnel has been tested with wolfEngine. Notes coming soon.

10.3 OpenSSH

To use wolfEngine with OpenSSH you will need to add `--enable-openssh` to the `./configure` line of wolfEngine.

OpenSSH needs to be compiled with OpenSSL engine support using the `--with-ssl-engine` configure option. If needed, `--with-ssl-dir=DIR` can also be used to specify the installation location of the OpenSSL library being used:

```
$ cd openssh
$ ./configure --prefix=/install/path --with-ssl-dir=/path/to/openssl/install
--with-ssl-engine
$ make
$ sudo make install
```

OpenSSH will also need an OpenSSL config file set up to leverage wolfEngine. If needed, the OPENSSL_CONF environment variable can be set to point to your config file. The OPENSSL_ENGINES environment variable may also need to be set to the location of the wolfEngine shared library:

```
$ export OPENSSL_CONF=/path/to/openssl.cnf
$ export OPENSSL_ENGINES=/path/to/wolfengine/library/dir
```


11 Support and OpenSSL Version Adding

For support with wolfEngine contact the wolfSSL support team at support@wolfssl.com. To have additional OpenSSL version support implemented in wolfEngine, contact wolfSSL at facts@wolfssl.com.

