wolfProvider Documentation

2025-12-26

Contents

1 Introduction

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

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

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

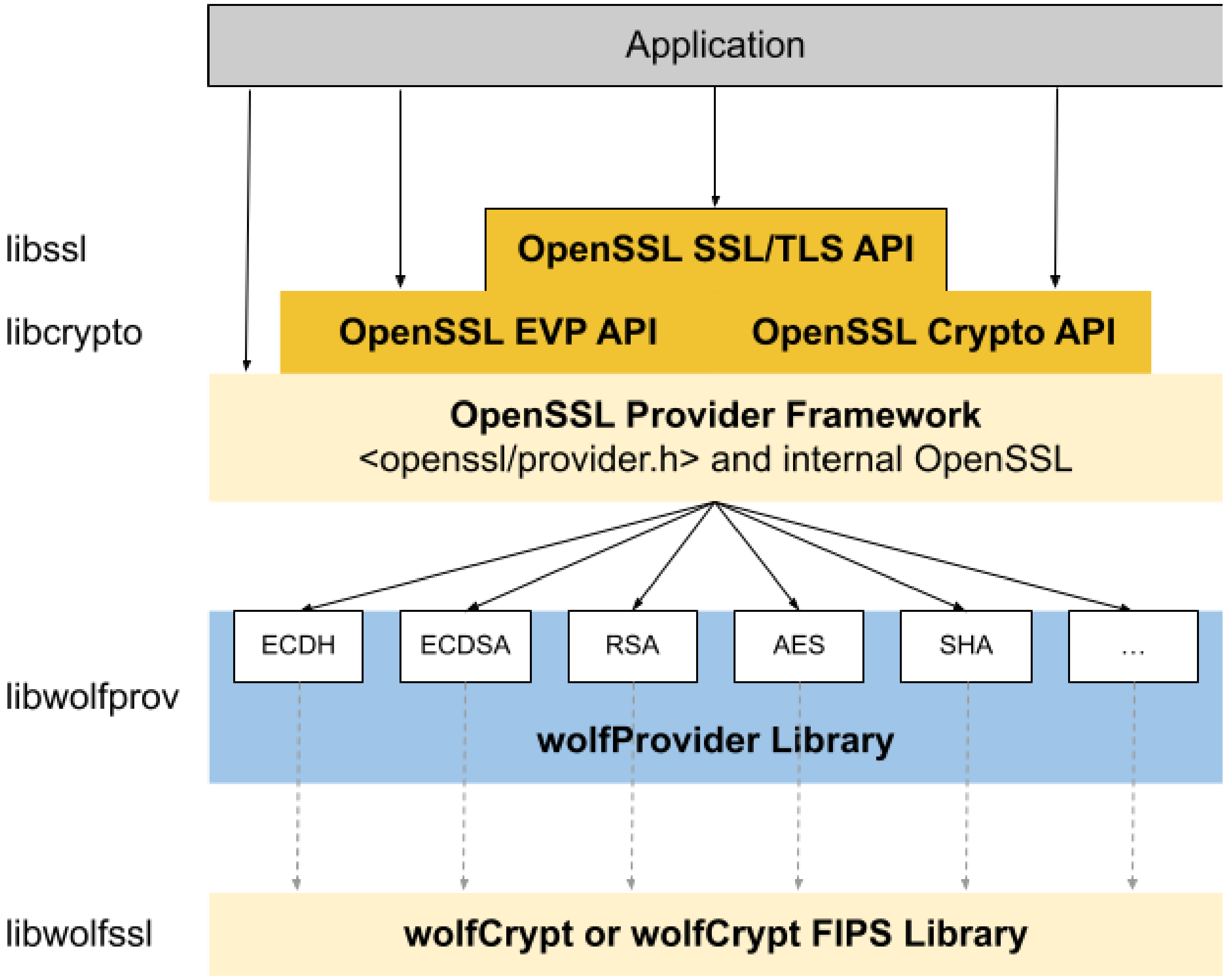

Figure 1: wolfProvider Overview

wolfProvider is compiled by default as a shared library called **libwolfprov** which can be dynamically registered at runtime by an application or OpenSSL through a config file. wolfProvider also provides an entry point for applications to load the provider when compiled in a static build.

2 OpenSSL Version Compatability

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

- OpenSSL 3.0.0
- OpenSSL 3.5.0

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

3 Building wolfProvider

3.1 Getting wolfProvider Source Code

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

3.2 wolfProvider Package Structure

The general wolfProvider package is structured as follows:

```
certs/                      (Test certificates and keys, used with unit tests)
examples/                   (Code examples)
include/
    wolfprovider/           (wolfProvider header files)
IDE/                        (Integration examples)
scripts/                    (wolfProvider scripts for testing and building)
src/                        (wolfProvider source files)
test/                       (wolfProvider test files)
provider.conf               (Example OpenSSL config file using wolfProvider)
provider-fips.conf          (Example OpenSSL config file using wolfProvider
   FIPS)
user_settings.h             (EXAMPLE user_settings.h)
```

3.3 Building on *nix

The quickest method is to use the `scripts/build-wolfprovider.sh` script as follows:

```
./scripts/build-wolfprovider.sh
```

It will clone, configure, compile, and install OpenSSL and wolfSSL with a default set of options. Two methods are available to override these defaults:

Setting the various environment variables prior to calling the script:

```
OPENSSL_TAG=openssl-3.2.0 WOLFSSL_TAG=v5.7.2-stable WOLFPROV_DEBUG=1 ./scripts
   /build-wolfprovider.sh
```

Specifying arguments for the script to parse:

```
./scripts/build-wolfprovider.sh --openssl-ver=openssl-3.2.0 --wolfssl-ver=v5
   .7.2-stable --debug
```

Of course, these methods can be combined to achieve the desired build combination as well.

For a full list of environment variables and script arguments do `./scripts/build-wolfprovider.sh --help`.

If desired, each component can be manually compiled using the following guide.

3.3.1 Building OpenSSL

A pre-installed version of OpenSSL may be used with wolfProvider, or OpenSSL can be recompiled for use with wolfProvider. 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
```

3.3.2 Building wolfSSL

If using a FIPS-validated version of wolfSSL with wolfProvider, follow the build instructions provided with your specific FIPS validated source bundle and Security Policy. In addition to the correct "–enable-fips" configure option, wolfProvider 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
```

If available, it may be easier to instead `make` then run the `./fips-hash.sh` utility and then `make` once again. This utility automates the process of updating fips_test.c with the testwolfcrypt hash output.

To build non-FIPS wolfSSL for use with wolfProvider:

```
cd wolfssl-X.X.X

./configure --enable-opensslcoexist --enable-cmac --enable-keygen --enable-sha
     --enable-des3 --enable-aesctr --enable-aesccm --enable-x963kdf --enable-
    compkey CPPFLAGS="-DHAVE_AES_ECB -DWOLFSSL_AES_DIRECT -DWC_RSA_NO_PADDING -
    DWOLFSSL_PUBLIC_MP -DHAVE_PUBLIC_FFDHE -DWOLFSSL_DH_EXTRA -
    DWOLFSSL_PSS_LONG_SALT -DWOLFSSL_PSS_SALT_LEN_DISCOVER -DRSA_MIN_SIZE=1024"
     --enable-certgen --enable-aeskeywrap --enable-enckeys --enable-base16 --
    with-eccminsz=192
make
sudo make install
```

Add `--enable-aesgcm-stream` if available for better AES-GCM support. Add `--enable-curve25519` to include support for X25519 Key Exchange. Add `--enable-curve448` to include support for X448 Key Exchange. Add `--enable-ed25519` to include support for Ed25519 signatures and certificates.. Add `--enable-ed448` to include support for Ed448 signature and certificates.

Add `--enable-pwdbased` to the configure command above if PKCS#12 is used in OpenSSL.

Add to CPPFLAGS `-DHAVE_FFDHE_6144 -DHAVE_FFDHE_8192 -DFP_MAX_BITS=16384` to enable predefined 6144-bit and 8192-bit DH parameters.

Add to `--enable-hmac-copy` if performing HMAC repeatedly with the same key to improve performance. (Available with wolfSSL 5.7.8+.)

Add `--enable-sp=yes,asm' '--enable-sp-math-all'` to use SP Integer maths. Replace -DFP_MAX_BITS=16384 with -DSP_INT_BITS=8192‘ when used.

Remove `-DWOLFSSL_PSS_LONG_SALT -DWOLFSSL_PSS_SALT_LEN_DISCOVER` and add `--enable-fips=v2` to the configure command above if building from a FIPS v2 bundle and not the git repository. Change `--enable-fips=v2` to `--enable-fips=ready` if using a FIPS Ready bundle.

If '–with-eccminsz=192' is not supported by wolfSSL, add '-DECC_MIN_KEY_SZ=192' to the CPPFLAGS.

"

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
```

3.3.3 Building wolfProvider

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

```
./configure
make
```

If building wolfProvider 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
```

wolfProvider 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 wolfProvider, 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 wolfProvider 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.

3.4 Building on WinCE

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

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

In the wolfProvider 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 wolfProvider 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 wolfProvider.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 wolfProvider as a static provider.

3.5 Build Options (./configure Options)

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

By default, wolfProvider 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
–disable-option-checking	**Disabled**	
–enable-silent-rules	**Disabled**	less verbose build output (undo: “make V=1”)
–disable-silent-rules	**Disabled**	verbose build output (undo: “make V=0”)
–enable-static	**Disabled**	Build static libraries
–enable-pic[=PKGS]	**Use Both**	try to use only PIC/non-PIC objects
–enable-shared	**Enabled**	Build shared libraries
–enable-fast-install[=PKGS]	**Enabled**	optimize for fast installation
–enable-aix-soname=aix|svr4|both	**aix**	shared library versioning (aka “SONAME”) variant to provide on AIX

Option	Default Value	Description
–enable-dependency-tracking	**Disabled**	do not reject slow dependency extractors
–disable-dependency-tracking	**Disabled**	speeds up one-time build
–disable-libtool-lock	**Disabled**	avoid locking (might break parallel builds)
–enable-debug	**Disabled**	Enable wolfProvider 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	**Enabled**	Enable loading wolfProvider as a dynamic provider
–enable-singlethreaded	**Disabled**	Enable wolfProvider single threaded
–with-openssl=DIR		OpenSSL installation location to link against. If not set, use the system default library and include paths.
–with-wolfssl=DIR		wolfSSL installation location to link against. If not set, use the system default library and include paths.

3.6 Build Defines

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

Define	Description
WOLFPROVIDER_USER_SETTINGS	Read user-specified defines from user_settings.h.
WOLFPROV_DEBUG	Output debug information
WP_CHECK_FORCE_FAIL	Force failure checking for testing purposes
WP_ALLOW_NON_FIPS	Allow certain non-FIPS algorithms in FIPS mode
WP_HAVE_AESCCM	AES encryption in CCM (Counter with CBC-MAC) mode
WP_HAVE_AESCFB	AES encryption in CFB (Cipher Feedback) mode
WP_HAVE_AESCBC	AES encryption in CBC (Cipher Block Chaining) mode
WP_HAVE_AESCTR	AES encryption in CTR (Counter) mode
WP_HAVE_AESCTS	AES encryption in CTS (Ciphertext Stealing) mode
WP_HAVE_AESECB	AES encryption in ECB (Electronic Codebook) mode
WP_HAVE_AESGCM	AES encryption in GCM (Galois/Counter Mode) mode
WP_HAVE_CMAC	CMAC (Cipher-based Message Authentication Code) support
WP_HAVE_DES3CBC	Triple DES encryption in CBC mode
WP_HAVE_DH	Diffie-Hellman key exchange support
WP_HAVE_DIGEST	General digest/hash algorithm support
WP_HAVE_ECC	General Elliptic Curve Cryptography support
WP_HAVE_EC_P192	P-192 elliptic curve support

Define	Description
WP_HAVE_EC_P224	P-224 elliptic curve support
WP_HAVE_EC_P256	P-256 elliptic curve support
WP_HAVE_EC_P384	P-384 elliptic curve support
WP_HAVE_EC_P521	P-521 elliptic curve support
WP_HAVE_ECDH	ECDH (Elliptic Curve Diffie-Hellman) key exchange support
WP_HAVE_ECDSA	ECDSA (Elliptic Curve Digital Signature Algorithm) support
WP_HAVE_ECKEYGEN	Elliptic curve key generation support
WP_HAVE_ED25519	Ed25519 elliptic curve signature support
WP_HAVE_ED448	Ed448 elliptic curve signature support
WP_HAVE_GMAC	GMAC (Galois/Counter Mode Authentication) support
WP_HAVE_HKDF	HKDF (HMAC-based Key Derivation Function) support
WP_HAVE_HMAC	HMAC (Hash-based Message Authentication Code) support
WP_HAVE_KRB5KDF	Kerberos 5 Key Derivation Function support
WP_HAVE_MD5	MD5 hash algorithm support
WP_HAVE_MD5_SHA1	MD5+SHA1 combination support
WP_HAVE_PBE	Password-Based Encryption support
WP_HAVE_RANDOM	Random number generation support
WP_HAVE_RSA	RSA encryption and signature support
WP_HAVE_SHA1	SHA1 hash algorithm support
WP_HAVE_SHA224	SHA224 hash algorithm support
WP_HAVE_SHA256	SHA256 hash algorithm support
WP_HAVE_SHA384	SHA384 hash algorithm support
WP_HAVE_SHA3	SHA3 family hash algorithm support
WP_HAVE_SHA3_224	SHA3-224 hash algorithm support
WP_HAVE_SHA3_256	SHA3-256 hash algorithm support
WP_HAVE_SHA3_384	SHA3-384 hash algorithm support
WP_HAVE_SHA3_512	SHA3-512 hash algorithm support
WP_HAVE_SHA512	SHA512 hash algorithm support
WP_HAVE_SHA512_224	SHA512/224 hash algorithm support
WP_HAVE_SHA512_256	SHA512/256 hash algorithm support
WP_HAVE_SHAKE_256	SHAKE256 extendable output function support
WP_HAVE_TLS1_PRF	TLS1 Pseudo-Random Function support
WP_HAVE_X25519	X25519 elliptic curve support
WP_HAVE_X448	X448 elliptic curve support
WP_RSA_PSS_ENCODING	RSA-PSS (Probabilistic Signature Scheme) encoding support

4 FIPS 140-3 Support

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

NOTE : If targeting FIPS compliance, and non-wolfCrypt FIPS algorithms are called from a different provider, those algorithms are outside the scope of both wolfProvider 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 Logging

wolfProvider supports output of log messages for informative and debug purposes. To enable debug logging, wolfProvider 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 `WOLFPROV_DEBUG` when compiling the wolfProvider library.

5.1 Controlling Logging Levels

wolfProvider supports the following logging levels. These are defined in the "include/wolfprovider/wp_logging.h" header file as part of the wolfProvider_LogType enum:

Log Enum	Description	Log Enum Value
WP_LOG_ERROR	Logs errors	0x0001
WP_LOG_ENTER	Logs when entering functions	0x0002
WP_LOG_LEAVE	Logs when leaving functions	0x0004
WP_LOG_INFO	Logs informative messages	0x0008
WP_LOG_VERBOSE	Verbose logs, including encrypted/decrypted/digested data	0x0010
WP_LOG_LEVEL_DEFAULT	Default log level, all except verbose level	WP_LOG_ERROR | WP_LOG_ENTER | WP_LOG_LEAVE | WP_LOG_INFO
WP_LOG_LEVEL_ALL	All log levels are enabled	WP_LOG_ERROR | WP_LOG_ENTER | WP_LOG_LEAVE | WP_LOG_INFO | WP_LOG_VERBOSE

The default wolfProvider logging level includes `WP_LOG_ERROR`, `WP_LOG_ENTER`, `WP_LOG_LEAVE`, and `WP_LOG_INFO`. This includes all log levels except verbose logs (`WP_LOG_VERBOSE`).

Log levels can be controlled using the `wolfProv_SetLogLevel(int mask)`. For example, to turn on only error and informative logs:

```
#include <wolfprovider/wp_logging.h>

ret = wolfProv_SetLogLevel(WP_LOG_ERROR | WP_LOG_INFO);
if (ret != 0) {
    printf“(Failed to set logging level\”n);
}
```

5.2 Controlling Component Logging

wolfProvider allows logging on a per-component basis. Components are defined in the wolfProvider_LogComponents enum in `include/wolfprovider/wp_logging.h`:

Log Component Enum	Description	Component Enum Value
WP_LOG_RNG	Random number generation	0x0001
WP_LOG_DIGEST	Digests (SHA-1/2/3)	0x0002
WP_LOG_MAC	MAC functions (HMAC, CMAC)	0x0004
WP_LOG_CIPHER	Ciphers (AES, 3DES)	0x0008
WP_LOG_PK	Public Key Algorithms (RSA, ECC)	0x0010
WP_LOG_KE	Key Agreement Algorithms (DH, ECDH)	0x0020
WP_LOG_KDF	Password Based Key Derivation Algorithms	0x0040
WP_LOG_PROVIDER	All provider specific logs	0x0080
WP_LOG_COMPONENTS_ALL	Log all components	WP_LOG_RNG | WP_LOG_DIGEST | WP_LOG_MAC | WP_LOG_CIPHER | WP_LOG_PK | WP_LOG_KE | WP_LOG_KDF | WP_LOG_PROVIDER
WP_LOG_COMPONENTS_DEFAULT	Default components logged (all).	WP_LOG_COMPONENTS_ALL

The default wolfProvider logging configuration logs all components (`WP_LOG_COMPONENTS_DEFAULT`).

Components logged can be controlled using the `wolfProv_SetLogComponents(int mask)`. For example, to turn on logging only for the Digest and Cipher algorithms:

```
#include <wolfprovider/wp_logging.h>

ret = wolfProv_SetLogComponents(WP_LOG_DIGEST | WP_LOG_CIPHER);
if (ret != 0) {
    printf“(Failed to set log components\”n);
}
```

5.3 Setting a Custom Logging Callback

By default wolfProvider 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 wolfProvider. The logging callback should match the prototype of wolfProvider_Logging_cb in `include/wolfprovider/wp_logging.h`:

```
/**
* wolfProvider logging callback.
* logLevel - [IN] - Log level of message
* component - [IN] - Component that log message is coming from
* logMessage - [IN] - Log message
*/
```

```
typedef void (* wolfProvider_Logging_cb )(const int logLevel,
const int component,
const char *const logMessage);
```

The callback can then be registered with wolfProvider using the `wolfProv_SetLoggingCb(wolfProv_Logging_cb logf)`. For example:

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

int main (void)
{
    int ret;
...
    ret = wolfProv_SetLoggingCb((void(*)(void))my_Logging_cb);
    if (ret != 0) {
        /* failed to set logging callback */
    }
...
}
```

6 Portability

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

6.1 Threading

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

6.2 Dynamic Memory Usage

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

6.3 Logging

wolfProvider logs by default to stderr via `fprintf()`. Applications can override this by registering a custom logging function (see Chapter 5).

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

WOLFPROV_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.

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

7 Loading wolfProvider

7.1 Configuring OpenSSL to Enable Provider Usage

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

OpenSSL 3.0

If the application is configured to read/use an OpenSSL config file, additional provider setup steps can be done there. For OpenSSL config documentation, reference the OpenSSL documentation:

OpenSSL 3.0

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.

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

```
OSSL_PROVIDER *prov = NULL;
const char *build = NULL;
OSSL_PARAM request[] = {
    { "buildinfo", OSSL_PARAM_UTF8_PTR, &build, 0, 0 },
    { NULL, 0, NULL, 0, 0 }
};

if ((prov = OSSL_PROVIDER_load(NULL, "libwolfprov")) != NULL
    && OSSL_PROVIDER_get_params(prov, request))
    printf("Provider 'libwolfprov' buildinfo: %s\n", build);
else
    ERR_print_errors_fp(stderr);

if (OSSL_PROVIDER_self_test(prov) == 0)
    printf("Provider selftest failed\n");
else
    printf("Provider selftest passed\n");

OSSL_PROVIDER_unload(prov);
```

7.2 Loading wolfProvider from an OpenSSL Configuration File

wolfProvider 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 wolfProvider library may be added to a config file is below.

```
openssl_conf = openssl_init

[openssl_init]
providers = provider_sect

[provider_sect]
libwolfprov = libwolfprov_sect

[libwolfprov_sect]
activate = 1
```

7.3 wolfProvider Static Entrypoint

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

```
#include <wolfprovider/wp_wolfprovider.h>
wolfssl_provider_init(const OSSL_CORE_HANDLE* handle, const OSSL_DISPATCH* in,
    const OSSL_DISPATCH** out, void** provCtx);
```

8 wolfProvider Design

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

Source File	Description
wp_wolfprov.c	Contains library entry points. Calls OpenSSL IMPLEMENT_DYNAMIC_BIND_FN for dynamic loading of the library using the OpenSSL provider framework. Also includes static entry points when compiled and used as a static library.
wp_internal.c	Includes wolfprovider_bind() function, which handles registration of provider algorithm callbacks. Also includes other wolfprovider internal functionality.
wp_logging.c	wolfProvider logging framework and function implementations.
wp_aes_aead.c	wolfProvider AES-AEAD (Authenticated Encryption with Associated Data) implementation.
wp_aes_block.c	wolfProvider AES-ECB and AES-CBC implementation.
wp_aes_stream.c	wolfProvider AES stream cipher implementation.
wp_aes_wrap.c	wolfProvider AES key wrapping implementation.
wp_cmac.c	wolfProvider CMAC (Cipher-based Message Authentication Code) implementation.
wp_dec_epki2pki.c	wolfProvider encrypted private key to private key conversion implementation.
wp_dec_pem2der.c	wolfProvider PEM to DER format conversion implementation.
wp_des.c	wolfProvider DES implementation.
wp_dh_exch.c	wolfProvider DH key exchange implementation.
wp_dh_kmgmt.c	wolfProvider DH key management implementation.
wp_digests.c	wolfProvider message digest implementations (SHA-1, SHA-2, SHA-3, …).
wp_drbg.c	wolfProvider DRBG (Deterministic Random Bit Generator) implementation.
wp_ecc_kmgmt.c	wolfProvider ECC key management implementation.
wp_ecdh_exch.c	wolfProvider ECDH key exchange implementation.
wp_ecdsa_sig.c	wolfProvider ECDSA signature implementation.
wp_ecx_exch.c	wolfProvider ECX key exchange implementation (X25519, X448, …).
wp_ecx_kmgmt.c	wolfProvider ECX key management implementation.
wp_ecx_sig.c	wolfProvider ECX signature implementation (Ed25519, Ed448, …).
wp_file_store.c	wolfProvider file storage implementation.
wp_fips.c	wolfProvider FIPS validation implementation.
wp_gmac.c	wolfProvider GMAC (Galois/Counter Mode) implementation.

Source File	Description
wp_hkdf.c	wolfProvider HKDF (HMAC-based Key Derivation Function) implementation.
wp_hmac.c	wolfProvider HMAC implementation.
wp_kbkdf.c	wolfProvider KBKDF (Key-Based Key Derivation Function) implementation.
wp_kdf_exch.c	wolfProvider KDF key exchange implementation.
wp_kdf_kmgmt.c	wolfProvider KDF key management implementation.
wp_krb5kdf.c	wolfProvider Kerberos 5 KDF implementation.
wp_mac_kmgmt.c	wolfProvider MAC key management implementation.
wp_mac_sig.c	wolfProvider MAC signature implementation.
wp_params.c	wolfProvider parameter handling implementation.
wp_pbkdf2.c	wolfProvider PBKDF2 (Password-Based Key Derivation Function 2) implementation.
wp_rsa_asym.c	wolfProvider RSA asymmetric encryption implementation.
wp_rsa_kem.c	wolfProvider RSA KEM (Key Encapsulation Mechanism) implementation.
wp_rsa_kmgmt.c	wolfProvider RSA key management implementation.
wp_rsa_sig.c	wolfProvider RSA signature implementation.
wp_tls1_prf.c	wolfProvider TLS 1.0 PRF implementation.
wp_tls_capa.c	wolfProvider TLS capabilities implementation.

8.1 wolfProvider Entry Points

The main entry points into the wolfProvider library are **OSSL_provider_init()** and **wolfssl_provider_init()**.

OSSL_provider_init() is the standard OpenSSL provider entry point that is called automatically by OpenSSL when the provider is loaded dynamically. This function is defined in `wp_wolfprov.c` and serves as a wrapper that calls `wolfssl_provider_init()`.

wolfssl_provider_init() is the core initialization function that:

- Sets up the provider context
- Initializes the dispatch table with provider functions
- Handles FIPS mode configuration
- Sets up debugging if enabled
- Returns the provider's dispatch table containing function pointers for:
  - `wolfprov_teardown` - Provider cleanup
  - `wolfprov_gettable_params` - Parameter table retrieval
  - `wolfprov_get_params` - Parameter retrieval
  - `wolfprov_query` - Operation querying
  - `wolfssl_prov_get_capabilities` - Capability reporting

The provider is loaded by OpenSSL when applications request wolfProvider algorithms, and the dispatch table allows OpenSSL to call the appropriate wolfProvider functions for cryptographic operations.

8.2 wolfProvider Dispatch Table Functions

The wolfProvider dispatch table contains several key functions that handle different aspects of provider operation. Each function serves a specific purpose in the OpenSSL provider framework.

Note on OSSL Parameters (Referenced Later): OSSL parameters are a standardized way for OpenSSL to exchange configuration data and capabilities information with providers. Parameters can represent simple values (integers, strings, booleans) or complex structures, and they enable applications to query provider capabilities, configure behavior, and retrieve status information. The parameter system provides a type-safe, extensible mechanism for provider-application communication.

8.2.1 wolfprov_teardown

The `wolfprov_teardown()` function is responsible for cleaning up wolfProvider when it is unloaded by OpenSSL. It performs the following cleanup tasks:

- Frees allocated provider context and resources
- Cleans up any remaining algorithm implementations
- Removes registered callbacks and handlers
- Ensures proper memory deallocation to prevent memory leaks
- Resets any global state maintained by the provider

This function is called automatically by OpenSSL when the provider is being unloaded, ensuring that all resources are properly released.

8.2.2 wolfprov_gettable_params

The `wolfprov_gettable_params()` function returns a table of parameter descriptors that define what parameters the provider supports. This function accomplishes the following tasks:

- Defines the structure and types of parameters that can be retrieved
- Provides metadata about parameter names, types, and descriptions
- Enables OpenSSL to understand what parameters are available from the provider
- Supports parameter validation and type checking
- Allows applications to discover available provider parameters

The returned table contains parameter definitions that applications can use to query provider capabilities and configuration options.

8.2.3 wolfprov_get_params

The `wolfprov_get_params()` function retrieves specific parameter values from the provider. This function:

- Accepts parameter requests from OpenSSL or applications
- Returns the current values of requested parameters
- Handles parameter type conversion and validation
- Provides access to provider configuration and state information
- Supports both simple parameters and complex parameter structures

Common parameters that can be retrieved include provider version, supported algorithms, FIPS mode status, and other configuration details.

8.2.4 wolfssl_prov_get_capabilities

The `wolfssl_prov_get_capabilities()` function reports the cryptographic capabilities of wolfProvider to OpenSSL. It provides capability information which:

- Returns information about supported algorithms and operations
- Provides details about algorithm parameters and constraints
- Indicates FIPS compliance and validation status
- Reports performance characteristics and limitations
- Enables OpenSSL to make informed decisions about algorithm selection

The capabilities information helps OpenSSL determine when to use wolfProvider algorithms and how to configure them appropriately for different use cases.

8.2.5 wolfprov_query

The `wolfprov_query()` function is the primary mechanism for algorithm discovery and registration in wolfProvider. It serves as the central routing mechanism that:

- Handles requests from OpenSSL for specific algorithm implementations
- Returns the appropriate algorithm structure when a supported algorithm is requested
- Provides operation-specific dispatch tables for cryptographic operations
- Manages algorithm registration and lookup within the provider
- Supports both symmetric and asymmetric cryptographic operations
- Enables dynamic algorithm discovery based on OpenSSL's requirements

When OpenSSL requests an algorithm (such as AES, RSA, SHA-256, etc.), `wolfprov_query()` determines if wolfProvider supports that algorithm and returns the corresponding implementation structure. This function acts as the central routing mechanism that connects OpenSSL's algorithm requests to the specific wolfProvider implementations found in the various source files (e.g., `wp_aes_block.c`, `wp_rsa_sig.c`, etc.).

9 Notes on Open Source Integration

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

wolfSSL has tested wolfProvider with numerous open source projects through automated CI/CD workflows. This chapter contains notes and tips on wolfProvider integration with the tested projects.

9.1 Tested Open Source Projects

The following Open Source Projects (OSPs) have been tested and verified to work with wolfProvider:

9.1.1 Network and Web Technologies

- cURL - Command line tool for transferring data with URLs
- gRPC - High-performance RPC framework
- libwebsockets - Lightweight C library for websockets
- Nginx - High-performance HTTP server and reverse proxy
- Qt5 Network - Qt networking module

9.1.2 Security and Authentication

- OpenSSH - Secure shell implementation
- libssh2 - SSH2 library
- libfido2 - FIDO2 library for WebAuthn
- OpenSC - Smart card tools and middleware
- pam-pkcs11 - PAM module for PKCS#11
- OpenVPN - VPN solution
- Stunnel - SSL wrapper for network services

9.1.3 System and Network Tools

- systemd - System and service manager
- tcpdump - Network packet analyzer
- rsync - File synchronization utility
- tnftp - Enhanced FTP client
- iperf - Network performance measurement tool
- IPMItool - IPMI management tool
- PPP - Point-to-Point Protocol implementation

9.1.4 Directory and Identity Services

- OpenLDAP - Lightweight Directory Access Protocol
- SSSD - System Security Services Daemon
- Net-SNMP - Network management protocol implementation

9.1.5 Cryptography and PKI

- cjose - C library for JWT
- libeac3 - Electronic Authentication Components
- libhashkit2 - Consistent hashing library
- liboauth2 - OAuth2 library

- libtss2 - TPM2 Software Stack
- tpm2-tools - TPM2 tools
- xmlsec - XML Security library
- sscep - SCEP client implementation

9.1.6 Development and Testing

- Asan - Address Sanitizer testing
- Codespell - Spell checker for source code
- Multi-Compiler - Multi-compiler testing

9.1.7 Remote Access and Display

- x11vnc - VNC server for X11
- python3-ntp - Python NTP library

9.1.8 Other Utilities

- Socat - Multipurpose relay for bidirectional data transfer
- Simple - Simple test applications

9.2 General Setup

Most of these projects require similar setup steps:

1. Clone from Github
2. Build with Autotools
3. Configure with OpenSSL
4. Make and Install
5. Use wolfProvider:

   ```
   export OPENSSL_CONF=/path/to/provider.conf
   export OPENSSL_MODULES=/path/to/wolfprov-install/lib
   ```

After running make (or the equivalent build script) the configured version of OpenSSL can be checked by running `ldd /path/to/compiled/binary`. This will provide a list of which libraries are linked against. If the incorrect version is present then setting some combination of these four environment variables before rebuilding may help:

```
export LD_LIBRARY_PATH="/path/to/wolfssl/install/lib:/path/to/openssl/install/
    lib64"
export PKG_CONFIG_PATH="/path/to/openssl/install/lib64/pkgconfig"
export LDFLAGS="-L/path/to/openssl/install/lib64"
export CPPFLAGS="-I/path/to/openssl/install/include"
```

Further, wolfProvider gives some ability to determine if the library is actually using wolfProvider. Just do `export WOLFPROV_FORCE_FAIL=1` or `WOLFPROV_FORCE_FAIL=1 /command/to/run` and if the command ends up using wolfProvider crypto it will fail.

If the project being used is included in the list of tested open source project's then the testing scripts can be referenced. These can be found in the wolfssl/wolfProvider repository on GitHub under .github/workflows/.

9.3 Testing and Validation

All of the above referenced open source project’s are continuously tested in the wolfProvider CI/CD pipeline with:

- OpenSSL version 3.5.0
- wolfSSL with both master and stable releases
- Force failure testing to ensure proper error handling
- FIPS testing is also done through a Jenkins pipeline

This comprehensive testing ensures that wolfProvider maintains compatibility with a wide range of open source projects and their various use cases.

10 Support and OpenSSL Version Adding

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