wolfProvider Documentation

2024-11-21

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.

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

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)
provider.conf               (Example OpenSSL config file using wolfProvider)
include/
    wolfprovider/           (wolfProvider header files)
scripts/                       (wolfProvider test scripts)
src/                             (wolfProvider source files)
test/                            (wolfProvider test files)
user_settings.h          (EXAMPLE user_settings.h)
```

3.3 Building on *nix

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
```

To build non-FIPS wolfSSL for use with wolfProvider:

```
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
```

3.3.3 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
–enable-static	**Disabled**	Build static libraries
–enable-shared	Enabled	Build shared libraries
–enable-debug	**Disabled**	Enable wolfProvider debugging support
–enable-coverage	**Disabled**	Build to generate code coverage stats

Option	Default Value	Description
–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_DEBUG	Build wolfProvider with debug symbols, optimization level, and debug logging.
WP_NO_DYNAMIC_PROVIDER	Do not build wolfProvider with dynamic provider support. Dynamic providers are ones that can be loaded into OpenSSL at runtime.
WP_SINGLE_THREADED	Build wolfProvider in single-threaded mode. This removes the need for locking around global resources used internally.
WP_USE_HASH	Enable digest algorithms using the wc_Hash API.
WP_HAVE_SHA1	Enable SHA-1 digest algorithm.
WP_HAVE_SHA224	Enable SHA-2 digest algorithm with digest size 224.
WP_HAVE_SHA256	Enable SHA-2 digest algorithm with digest size 256.

Define	Description
WP_HAVE_SHA384	Enable SHA-2 digest algorithm with digest size 384.
WP_HAVE_SHA512	Enable SHA-2 digest algorithm with digest size 512.
WP_SHA1_DIRECT	Enable the SHA-1 digest algorithm using the wc_Sha API. Incompatible with WP_USE_HASH.
WP_SHA224_DIRECT	Enable the SHA-2 224 digest algorithm using the wc_Sha224 API. Incompatible with WP_USE_HASH.
WP_SHA256_DIRECT	Enable the SHA-2 256 digest algorithm using the wc_Sha256 API. Incompatible with WP_USE_HASH.
WP_HAVE_SHA3_224	Enable SHA-3 digest algorithm with digest size 224. Not available in OpenSSL 1.0.2.
WP_HAVE_SHA3_256	Enable SHA-3 digest algorithm with digest size 256. Not available in OpenSSL 1.0.2.
WP_HAVE_SHA3_384	Enable SHA-3 digest algorithm with digest size 384. Not available in OpenSSL 1.0.2.
WP_HAVE_SHA3_512	Enable SHA-3 digest algorithm with digest size 512. Not available in OpenSSL 1.0.2.
WP_HAVE_EVP_PKEY	Enable functionality that uses the EVP_PKEY API. This includes things like RSA, DH, etc.
WP_HAVE_CMAC	Enable CMAC algorithm.
WP_HAVE_HMAC	Enable HMAC algorithm.
WP_HAVE_DES3CBC	Enable DES3-CBC algorithm.
WP_HAVE_AESECB	Enable AES algorithm with ECB mode.
WP_HAVE_AESCBC	Enable AES algorithm with CBC mode.

Define	Description
WP_HAVE_AESCTR	Enable AES algorithm with countee mode.
WP_HAVE_AESGCM	Enable AES algorithm with GCM mode.
WP_HAVE_AESCCM	Enable AES algorithm with CCM mode.
WP_HAVE_RANDOM	Enable wolfCrypt random implementation.
WP_HAVE_RSA	Enable RSA operations (e.g. sign, verify, key generation, etc.).
WP_HAVE_DH	Enable Diffie-Hellman operations (e.g. key generation, shared secret computation, etc.).
WP_HAVE_ECC	Enable support for elliptic curve cryptography.
WP_HAVE_EC_KEY	Enable support for EC_KEY_METHOD. Not available in OpenSSL 1.0.2.
WP_HAVE_ECDSA	Enable ECDSA algorithm.
WP_HAVE_ECDH	Enable EC Diffie-Hellman operations.
WP_HAVE_ECKEYGEN	Enable EC key generation.
WP_HAVE_EC_P192	Enable EC curve P192.
WP_HAVE_EC_P224	Enable EC curve P224.
WP_HAVE_EC_P256	Enable EC curve P256.
WP_HAVE_EC_P384	Enable EC curve P384.
WP_HAVE_EC_P512	Enable EC curve P512.
WP_HAVE_DIGEST	Compile code in benchmark program and unit tests for use with digest algorithms.
WOLFPROVIDER_USER_SETTINGS	Read user-specified defines from user_settings.h.

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 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 `WOLFPROVIDER_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_ENTER | WP_LOG_LEAVE | WP_LOG_ERROR | 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_PROVIDER	All provider specific logs	0x0040
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_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 only 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_wolfprovider.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_openssl_bc.c	wolfProvider OpenSSL binary compatibility abstraction layer, used for supporting wolfProvider across multiple OpenSSL versions.
wp_aes_block.c	wolfProvider AES-ECB and AES-CBC implementation.
wp_aes_cbc_hmac.c	wolfProvider AES-CBC-HMAC implementation.
wp_aes_ccm.c	wolfProvider AES-CCM implementation.
wp_aes_ctr.c	wolfProvider AES-CTR implementation.
wp_aes_gcm.c	wolfProvider AES-GCM implementation.
wp_des3_cbc.c	wolfProvider 3DES-CBC implementation.
wp_dh.c	wolfProvider DH implementation.
wp_digest.c	wolfProvider message digest implementations (SHA-1, SHA-2, SHA-3).
wp_ecc.c	wolfProvider ECDSA and ECDH implementation.
wp_mac.c	wolfProvider HMAC and CMAC implementations.
wp_random.c	wolfProvider RAND implementation.
wp_rsa.c	wolfProvider RSA implementation.
wp_tls_prf.c	wolfProvider TLS 1.0 PRF implementation.

8.1 wolfProvider Entry Points

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

8.2 wolfProvider Algorithm Callback Registration

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

These API calls include the following:

```
PROVIDER_set_id(e, wolfprovider_id)
PROVIDER_set_name(e, wolfprovider_name)
PROVIDER_set_digests(e, wp_digests)
PROVIDER_set_ciphers(e, wp_ciphers)
PROVIDER_set_RAND(e, wp_random_method)
PROVIDER_set_RSA(e, wp_rsa())
PROVIDER_set_DH(e, wp_dh_method)
PROVIDER_set_ECDSA(e, wp_ecdsa())
PROVIDER_set_pkey_meths(e, wp_pkey)
PROVIDER_set_pkey_asn1_meths(e, wp_pkey_asn1)
PROVIDER_set_EC(e, wp_ec())
PROVIDER_set_ECDH(e, wp_ecdh())
PROVIDER_set_destroy_function(e, wolfprovider_destroy)
PROVIDER_set_cmd_defns(e, wolfprovider_cmd_defns)
PROVIDER_set_ctrl_function(e, wolfprovider_ctrl)
```

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

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 OpenSSL configuration file or programmatically via PROVIDER API calls.

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

9.1 cURL

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

1. Add wolfProvider provider 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_PROVIDERS environment variable to point to location of wolfProvider shared library file:

```
$ export OPENSSL_PROVIDERS=/path/to/wolfprovider/library/dir
```

9.2 stunnel

stunnel has been tested with wolfProvider. Notes coming soon.

9.3 OpenSSH

OpenSSH needs to be compiled with OpenSSL provider support using the `--with-ssl-provider` 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-provider
$ make
$ sudo make install
```

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

```
$ export OPENSSL_CONF=/path/to/openssl.cnf
$ export OPENSSL_PROVIDERS=/path/to/wolfprovider/library/dir
```


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.

