wolfProvider Documentation

2025-04-22

Contents

1 イントロダクション **3**

2 OpenSSL との互換性 **4**

3 wolfProvider のビルド **5**
3.1 wolfProvider のダウンロード 5
3.2 パッケージ構成 5
3.3 OpenSSL のバージョンに関する注意事項 5
3.4 *nix 上でのビルド 5
3.4.1 OpenSSL をビルド 5
3.4.2 wolfSSL をビルド 5
3.4.3 wolfProvider をビルド 6
3.5 WinCE 上でのビルド 7
3.6 ビルドオプション (./configure に指定するオプション) 8
3.7 ビルド用マクロ定義 9

4 FIPS 140-2 のサポート **11**
4.1 デバッグログの有効化/無効化 11
4.2 ログ出力レベルの制御 11
4.3 コンポーネント単位のログ出力の制御 12
4.4 カスタムログ出力コールバックの設定 13

5 移植性 **15**
5.1 スレッド対応 15
5.2 動的メモリ使用 15
5.3 ログ出力 15

6 wolfProvider のロード **16**
6.1 OpenSSL をエンジンを利用できるように構成 16
6.2 OpenSSL コンフィギュレーションファイルによる wolfProvider のロード 16
6.3 wolfProvider 静的エントリポイント 17

7 wolfProvider の設計 **18**
7.1 wolfProvider エントリーポイント 18
7.2 wolfProvider アルゴリズムコールバック登録 18

8 その他のオープンソースソフトウェアとの統合に関する注意事項 **20**
8.1 cURL 20
8.2 stunnel 20
8.3 OpenSSH 20

9 サポートと OpenSSL バージョン追加 **21**

1 イントロダクション

wolfCrypt エンジン (wolfProvider) は、wolfCrypt および wolfCrypt FIPS 暗号化ライブラリを OpenSSL エンジンフレームワークに適合させるためのライブラリです。wolfProvider は、共有または静的ライブラリとして OpenSSL エンジン互換の実装を提供し、現在 OpenSSL を使用しているアプリケーションが FIPS および非 FIPS ユースケースで wolfCrypt 暗号化ライブラリを活用できるようにします。

wolfProvider は、wolfSSL(libwolfssl) と OpenSSL にリンクする個別のスタンドアロンライブラリとして構成されています。wolfProvider は、wolfCrypt ネイティブ API を内部的にラップする OpenSSL エンジンです。wolfProvider の概要図、そしてアプリケーションや OpenSSL とどのように関連しているかを下の図 1 に示します。

wolfProvider の設計とアーキテクチャの詳細については、wolfProvider の設計 の章をご覧ください。

wolfProvider Overview

wolfProvider は、**libwolfprovider** という名前の共有ライブラリとしてコンパイルされます。これは、アプリケーションまたはコンフィギュレーションファイルを介して OpenSSL によって実行時に動的に登録できます。wolfProvider は、アプリケーションが静的ビルドでコンパイルされたときにエンジンをロードするためのエントリポイントも提供します。

2 OpenSSL との互換性

wolfProvider は、以下のバージョンの OpenSSL を使用してテストを実行しています。これ以外のバージョンでは、変更や調整が必要になる場合があります。

- OpenSSL 1.0.2h
- OpenSSL 1.1.1b

他の OpenSSL バージョンのサポート追加を希望される場合は、info@wolfssl.jp にご連絡ください。

3 wolfProvider のビルド

3.1 wolfProvider のダウンロード

wolfProvider の最新バージョンは、wolfSSL Inc. から直接入手できます。詳細については、info@wolfssl.jpまでお問い合わせください。

3.2 パッケージ構成

wolfProvider パッケージは、以下のように構成しています。

```
certs/                (ユニットテストで使用されるテスト用証明書、鍵)
provider.conf         (wolfProviderを使用する場合のOpenSSLコンフィギュレーショ
    ンファイルサンプル)
include/
    wolfprovider/       (wolfProviderヘッダーファイル)
openssl_patches/
    1.0.2h/tests/     (OpenSSL 1.0.2h テストアプリ用パッチ)
    1.1.1b/tests/     (OpenSSL 1.1.1b テストアプリ用パッチ)
scripts/              (wolfProvider テストスクリプト)
src/                  (wolfProvider ソースファイル)
test/                 (wolfProvider テストファイル)
user_settings.h       (user_settings.hサンプル)
```

3.3 OpenSSL のバージョンに関する注意事項

次に示すアルゴリズムを利用するには、併記した OpenSSL バージョンが必要です。

- SHA-3：OpenSSL バージョン 1.1.1 以降が必要
- EC_KEY_METHOD：OpenSSL バージョン 1.1.1 以降が必要

3.4 *nix 上でのビルド

3.4.1 OpenSSL をビルド

すでにインストールされた OpenSSL を使用することも、新しく 1 から OpenSSL をコンパイルして使用することもできます。以下に、*nix(Linux, Unix) 上で OpenSSL をコンパイルする一般的な手法を示します。より詳しいビルド手順をお求めでしたら、OpenSSL のINSTALL ファイルやドキュメントをご参照ください。

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

3.4.2 wolfSSL をビルド

wolfProvider で wolfSSL FIPS 版を使用するには、特定の FIPS 検証済みソースバンドルやセキュリティポリシーで指定されたビルド手順に従う必要があります。まずコンフィギュレーションオプションとして`--enable-fips` が必要で、かつ **WOLFSSL_PUBLIC_MP** が定義された状態で wolfSSL をコンパイルします。以下に、Linux で「wolfCrypt Linux FIPSv2 バンドル」をビルドする例を示します。

```
cd wolfssl-X.X.X-commercial-fips-linuxv
./configure **--enable-fips=v2 CFLAGS=" -DWOLFSSL_PUBLIC_MP " **
make
./wolfcrypt/test/testwolfcrypt
#< ここで、fips_test.c内のverifyCoreを開き、testwolfcryptスクリプトが出力する
    ハッシュ値に更新してください >--

make
./wolfcrypt/test/testwolfcrypt

#< すべてのテストでPASSできるはずです >--

sudo make install
```

非 FIPS 版の wolfSSL を使用する場合は、以下のようになります。

```
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
```

なお wolfSSL を GitHub リポジトリから取得された場合は、`./configure` を実行する前に `autogen.sh` スクリプトを実行する必要があります。これにより、configure スクリプトが生成されます。

```
./autogen.sh
```

3.4.3 wolfProvider をビルド

Linux などの *nix ライクな環境で wolfProvider をビルドする場合は、autoconf をご利用ください。wolfProvider をビルドするには、wolfProvider のルートディレクトリ上で次のコマンドを実行します。

```
./configure
make
```

wolfProvider を GitHub リポジトリから取得された場合は、`./configure` を実行する前に `autogen.sh` スクリプトを実行する必要があります。

```
./autogen.sh
```

任意の数のビルドオプションを `./configure` に追加できます。利用可能なビルドオプションの一覧については、後の「ビルドオプション」セクションに掲載しているほか、次のコマンドを実行することで表示できます。

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

wolfProvider は通常、システムにインストールされたデフォルトの OpenSSL ライブラリを使用します。`--with-openssl` オプションにより、特定のディレクトリに存在する OpenSSL を使用することもできます。

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

デフォルト以外の OpenSSL を使用する場合、ライブラリ検索パスに追加しなければならないケースがあります。Linux システムでは、次のように LD_LIBRARY_PATH を編集します。

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

wolfProvider をビルドしインストールするには、以下のコマンドを実行します。

```
make
make install
```

インストール時には root 権限を求められる場合があります。その際は、コマンドの前に sudo を付加します。

```
sudo make install
```

ビルド結果をテストするには、wolfProvider のルートディレクトリで以下のコマンドを実行します。

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

次のように、autoconf を使用して実行することもできます。

```
make check
```

ライブラリが見つからない場合、`error while loading shared libraries: libssl.so.3` のようなエラーが発生します。環境変数 LD_LIBRARY_PATH を編集することで解決しないかお試しください。

3.5 WinCE 上でのビルド

wolfProvider との互換性を保つために、wolfCrypt の user_settings.h ファイルに以下の定義があることをご確認ください。

```
#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
```

使用するアルゴリズムと機能に応じて、user_settings.h ファイルに wolfProvider フラグを追加します。wolfProvider ディレクトリにある user_settings.h ファイルで、wolfProvider ユーザー設定フラグを参照できます。

Windows CE 用の wcecompat、wolfCrypt、および OpenSSL をビルドし、それらのパスを参照できるようにします。

wolfProvider ディレクトリでソースファイルを開き、OpenSSL、wolfCrypt、および user_settings.h パスを使用しているディレクトリに変更します。INCLUDES セクションと TARGETLIBS セクションのパスを更新する必要があります。

Visual Studio で wolfProvider プロジェクトをロードします。ベンチマークまたは単体テストを実行する場合は、「bench.c」、または「unit.h」と「unit.c」のいずれかを含めます。

プロジェクトをビルドすると、実行可能ファイル wolfProvider.exe が作成されます。この実行可能ファイルに--help の引数をつけて実行すると、オプションの一覧を表示できます。wolfProvider を静的エンジンとして使用するには--static を付けて実行する必要があります。

3.6 ビルドオプション (./configure に指定するオプション)

ライブラリの構築方法をカスタマイズするために `./configure` スクリプトに追加できるオプションを以下に示します。

デフォルトでは、ビルド時間を半分にするために共有ライブラリのみをビルドします。必要に応じて、静的ライブラリをビルドしたり共有ライブラリのビルドを無効化したりできます。

オプション	デフォルト	意味
–enable-static	**無効**	静的ライブラリとしてビルド
–enable-shared	有効	共有ライブラリとしてビルド
–enable-debug	**無効**	wolfProvider のデバッグ出力を有効にする
–enable-coverage	**無効**	コードカバレッジレポートを作成する用ビルド
–enable-usersettings	**無効**	user_settings.h を使用し Makefile の CFLAGS を使用しない
–enable-dynamic-provider	有効	wolfProvider をダイナミックエンジンとしてロードする
–enable-singlethreaded	**無効**	wolfProvider をシングルスレッド環境で使用する
–enable-digest	有効	ダイジェストの生成に wc_Hash API を使用する
–enable-sha	有効	SHA-1 を有効にする
–enable-sha224	有効	SHA2-224 を有効にする
–enable-sha256	有効	SHA2-256 を有効にする
–enable-sha384	有効	SHA2-384 を有効にする
–enable-sha512	有効	SHA2-512 を有効にする
–enable-sha3	有効	SHA3 を有効にする
–enable-sha3-224	有効	SHA3-224 を有効にする
–enable-sha3-256	有効	SHA3-256 を有効にする
–enable-sha3-384	有効	SHA3-384 を有効にする
–enable-sha3-512	有効	SHA3-512 を有効にする
–enable-cmac	有効	CMAC を有効にする
–enable-hmac	有効	HMAC を有効にする
–enable-des3cbc	有効	3DES-CBC を有効にする
–enable-aesecb	有効	AES-ECB を有効にする
–enable-aescbc	有効	AES-CBC を有効にする
–enable-aesctr	有効	AES-CTR を有効にする
–enable-aesgcm	**無効**	AES-GCM を有効にする
–enable-aesccm	**無効**	AES-CCM を有効にする
–enable-rand	有効	RAND を有効にする
–enable-rsa	有効	RSA を有効にする
–enable-dh	有効	DH を有効にする
–enable-evp-pkey	有効	EVP_PKEY APIs を有効にする
–enable-ec-key	有効	ECC using EC_KEY を有効にする
–enable-ecdsa	有効	ECDSA を有効にする
–enable-ecdh	有効	ECDH を有効にする
–enable-eckg	有効	EC Key Generation を有効にする
–enable-p192	有効	EC Curve P-192 を有効にする
–enable-p224	有効	EC Curve P-224 を有効にする
–enable-p256	有効	EC Curve P-256 を有効にする
–enable-p384	有効	EC Curve P-384 を有効にする
–enable-p521	有効	EC Curve P-521 を有効にする

オプション	デフォルト	意味
–with-openssl=DIR		OpenSSL のインストール場所を指定。指定しない場合はシステムのデフォルトライブラリパスとインクルードパスが使われます。

3.7 ビルド用マクロ定義

wolfProvider は、お客様が wolfProvider のビルド方法を設定できるようにするいくつかのプリプロセッサマクロを公開しています。以下にその一覧を示します。

マクロ定義	意味
WOLFPROVIDER_DEBUG	デバッグシンボル、最適化レベル、デバッグロギングを使用して wolfProvider をビルドします
WP_NO_DYNAMIC_PROVIDER	wolfProvider をダイナミックエンジンとしてビルドしない。ダイナミックエンジンとは、OpenSSL が実行時に動的にロードするエンジンのことです。
WP_SINGLE_THREADED	wolfProvider をシングルスレッドモードでビルドする。このマクロ定義によりグローバルリソースの使用の排他用に内部的に使用するロック機構を取り除きます。
WP_USE_HASH	ハッシュアルゴリズムを wc_Hash API を使って有効にする
WP_HAVE_SHA1	SHA-1 を有効にする
WP_HAVE_SHA224	SHA-2 224 を有効にする
WP_HAVE_SHA256	SHA-2 256 を有効にする
WP_HAVE_SHA384	SHA-2 384 を有効にする
WP_HAVE_SHA512	SHA-2 512 を有効にする
WP_SHA1_DIRECT	SHA-1 を wc_Sha API を使って有効にする。WP_USE_HASH とは同時に指定できません。
WP_SHA224_DIRECT	SHA-2 224 を wc_Sha224 API を使って有効にする。WP_USE_HASH とは同時に指定できません。
WP_SHA256_DIRECT	SHA-2 256 を wc_Sha256 API を使って有効にする。WP_USE_HASH とは同時に指定できません。
WP_HAVE_SHA3_224	SHA-3 224 を有効にする（OpenSSL 1.0.2 では利用不可）
WP_HAVE_SHA3_256	SHA-3 256 を有効にする（OpenSSL 1.0.2 では利用不可）
WP_HAVE_SHA3_384	SHA-3 384 を有効にする（OpenSSL 1.0.2 では利用不可）
WP_HAVE_SHA3_512	SHA-3 512 を有効にする（OpenSSL 1.0.2 では利用不可）
WP_HAVE_EVP_PKEY	EVP_PKEY API を使用する機能を有効にする（RSA, DH 等も含む）
WP_HAVE_CMAC	CMAC を有効にする
WP_HAVE_HMAC	HMAC を有効にする
WP_HAVE_DES3CBC	DES3-CBC を有効にする
WP_HAVE_AESECB	AES-ECB を有効にする
WP_HAVE_AESCBC	AES-CBC を有効にする
WP_HAVE_AESCTR	AES-countee mode を有効にする
WP_HAVE_AESGCM	AES-GCM を有効にする
WP_HAVE_AESCCM	AES-CCM を有効にする
WP_HAVE_RANDOM	wolfCrypt の疑似乱数生成実装を有効にする
WP_HAVE_RSA	RSA 操作 (署名、検証、鍵生成等) を有効にする
WP_HAVE_DH	Diffie-Hellman 操作 (鍵生成、共有シークレット計算等) を有効にする
WP_HAVE_ECC	楕円曲線暗号を有効にする
WP_HAVE_EC_KEY	EC_KEY_METHOD のサポートを有効にする（OpenSSL 1.0.2 では利用不可）

マクロ定義	意味
WP_HAVE_ECDSA	ECDSA を有効にする
WP_HAVE_ECDH	EC Diffie-Hellman 操作を有効にする
WP_HAVE_ECKEYGEN	EC 鍵生成を有効にする
WP_HAVE_EC_P192	EC Curve P192 を有効にする
WP_HAVE_EC_P224	EC Curve P224 を有効にする
WP_HAVE_EC_P256	EC Curve P256 を有効にする
WP_HAVE_EC_P384	EC Curve P384 を有効にする
WP_HAVE_EC_P512	EC Curve P512 を有効にする
WP_HAVE_DIGEST	ダイジェストアルゴリズムをベンチマークとユニットテストのコードに含めてコンパイルする
WOLFPROVIDER_USER_SETTINGS	ユーザーの指定した定義を user_settings.h ファイルから読み込む

4 FIPS 140-2 のサポート

wolfProvider は，FIPS で検証されたバージョンの wolfCrypt に対して適切にコンパイルされた場合にのみ、FIPS140-2 に対応した動作を行うよう設計しています。この使用シナリオには、wolfSSL Inc. から入手した、適切にライセンスされ、検証されたバージョンの wolfCrypt が必要です。

wolfCrypt FIPS ライブラリは、非 FIPS モードに「切り替える」ことができません。通常の非 FIPS 版 wolfCrypt と FIPS 版 wolfCrypt は、それぞれ別々のソースコードパッケージで提供しています。

wolfProvider を FIPS 版 wolfCrypt を使用するようにコンパイルすると、FIPS で検証されたアルゴリズム、モード、および鍵サイズのサポートおよび登録エンジンコールバックのみが含まれます。OpenSSL ベースのアプリケーションが FIPS 検証が行われていないアルゴリズムを呼び出す場合、実行は wolfProvider に入らず、OpenSSL 構成に基づいて、デフォルトの OpenSSL エンジンまたは他の登録済みエンジンプロバイダーによって処理される可能性があります。

注:wolfCrypt 以外に実装された FIPS アルゴリズムを別のプロバイダーから呼び出す場合、それらのアルゴリズムは wolfProvider および FIPS 版 wolfCrypt のスコープに含みません。FIPS 認証取得に際し、問題となる可能性があります。

FIPS 版 wolfCrypt(140-2/140-3) の使用に関する詳細については、wolfSSL(info@wolfssl.jp) までお問い合わせください。# ログ出力

wolfProvider は、情報提供とデバッグを目的としたログメッセージの出力をサポートしています。デバッグログ出力を有効にするには、最初にデバッグサポートを有効にして wolfProvider をコンパイルする必要があります。autoconf を使用している場合、これは `./configure` に `--enable-debug` オプションを加えることで実現できます。

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

autoconf/configure を使用しない場合は、wolfProvider ライブラリをコンパイルする際に `WOLFPROVIDER_DEBUG` を定義します。

4.1 デバッグログの有効化/無効化

デバッグサポートを有効化してコンパイルを行った後、以下に示す wolfProvider コントロールコマンドを使用して実行時にデバッグを有効にする必要があります。“0” を指定すると、ログ出力が無効になります。`PROVIDER_ctrl_cmd()` API を使用してログ出力を有効にする例を以下に示します。

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

wolfProvider がデバッグサポート無効の状態でコンパイルされた場合、`PROVIDER_ctrl_cmd()` で `enable_debug` を設定しようとすると失敗 (0) が返されます。

4.2 ログ出力レベルの制御

wolfProvider は以下のログ出力レベルをサポートします。これらは、`include/wolfprovider/wp_logging.h` で、`wolfProvider_LogType enum` の一部として定義しています。

ログ出力レベル	意味	レベル値
WP_LOG_ERROR	エラーログを出力	0x0001
WP_LOG_ENTER	関数に入った際にログを出力	0x0002

ログ出力レベル	意味	レベル値
WP_LOG_LEAVE	関数を抜ける際にログを出力	0x0004
WP_LOG_INFO	情報提供のメッセージをログを出力	0x0008
WP_LOG_VERBOSE	暗号化/復号のデータを含めた詳細ログを出力	0x0010
WP_LOG_LEVEL_DEFAULT	デフォルトのログレベル（VERBOSE 以外を全て含む）	WP_LOG_ERROR | WP_LOG_ENTER | WP_LOG_LEAVE | WP_LOG_INFO
WP_LOG_LEVEL_ALL	全てのログを出力	WP_LOG_ENTER | WP_LOG_LEAVE | WP_LOG_INFO | WP_LOG_VERBOSE
WP_LOG_ERROR		

デフォルトの wolfProvider ログ出力レベルには、`WP_LOG_ERROR`、`WP_LOG_ENTER`、`WP_LOG_LEAVE`、`WP_LOG_INFO` を含みます。すなわち、詳細ログ (`WP_LOG_VERBOSE`) を除くすべてのログが出力されます。

ログレベルは、`PROVIDER_ctrl_cmd()`API または OpenSSL 構成ファイル設定のいずれかを介して、実行時に "**log_level**" エンジン制御コマンドを使用して制御できます。例えば、"log_level" 制御コマンドを使用してエラーログと情報ログのみを有効にするには、アプリケーションで次のように実装します。

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

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

4.3 コンポーネント単位のログ出力の制御

wolfProvider では、コンポーネントごとにログを出力できます。コンポーネントは `include/wolfprovider/wp_logging.h` の `wolfProvider_LogComponents` に定義しています。

ログ対象コンポーネント	意味	コンポーネントを示す値
WP_LOG_RNG	乱数生成コンポーネント	0x0001
WP_LOG_DIGEST	ダイジェストコンポーネント (SHA-1/2/3)	0x0002
WP_LOG_MAC	MAC 機能コンポーネント (HMAC, CMAC)	0x0004
WP_LOG_CIPHER	暗号化コンポーネント (AES, 3DES)	0x0008
WP_LOG_PK	公開鍵コンポーネント (RSA, ECC)	0x0010
WP_LOG_KE	鍵合意コンポーネント (DH, ECDH)	0x0020
WP_LOG_PROVIDER	エンジン特有	0x0040
WP_LOG_COMPONENTS_ALL	全コンポーネント	WP_LOG_RNG | WP_LOG_DIGEST | WP_LOG_MAC | WP_LOG_CIPHER | WP_LOG_PK | WP_LOG_KE | WP_LOG_PROVIDER

ログ対象コンポーネント	意味	コンポーネントを示す値
WP_LOG_COMPONENTS_DEFAULT	デフォルトコンポーネント (all).	WP_LOG_COMPONENTS_ALL

デフォルトでは、すべてのコンポーネントを対象としてログを出力します (WP_LOG_COMPONENTS_DEFAULT)。

ログ出力の対象とするコンポーネントは、PROVIDER_ctrl_cmd()API または OpenSSL 構成ファイル設定のいずれかを介して、実行時に "**log_components**" エンジン制御コマンドを使用して制御できます。たとえば、Digest および Cipher アルゴリズムのみのログ出力を有効にするには、次のようにします。

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

ret = PROVIDER_ctrl_cmd(e, " log_components " , WP_LOG_DIGEST | WP_LOG_CIPHER,
NULL, NULL, 0);
if (ret != 1) {
    printf( " Failed to set log components\n " );
}
```

4.4 カスタムログ出力コールバックの設定

デフォルトでは、wolfProvider は **fprintf()** を使用してデバッグログメッセージを **stderr** に出力します。

ログメッセージの出力方法や出力場所を変更したい場合は、カスタムログ出力コールバック関数を記述して wolfProvider に登録します。その際、include/wolfprovider/wp_logging.h に示す wolfProvider_Logging_cb のプロトタイプ宣言と一致させる必要があります。

```
/**
* 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);
```

その後、"**set_logging_cb**" エンジン制御コマンドを使用して、コールバック関数を wolfProvider に登録できます。例えば、PROVIDER_ctrl_cmd()API を使用してカスタムログ出力コールバック関数を設定するには次のようにします。

```
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;
    PROVIDER* e;
...
    ret = PROVIDER_ctrl_cmd(e, " set_logging_cb " , 0, NULL, (void(*)(void))
       my_Logging_cb, 0);
    if (ret != 1) {
```

```
        /* failed to set logging callback */
    }
...
}
```

5 移植性

wolfProvider は、関連する wolfCrypt および OpenSSL ライブラリの移植性を活用するように設計しています。

5.1 スレッド対応

wolfProvider はスレッドセーフであり、必要に応じて wolfCrypt のミューテックスロックメカニズム `wc_LockMutex()`、`wc_UnLockMutex()` を使用します。wolfCrypt には、サポートしているプラットフォーム用に抽象化されたミューテックス操作があります。

5.2 動的メモリ使用

wolfProvider は OpenSSL のメモリ割り当て関数を使用して、OpenSSL の動作との一貫性を維持します。wolfProvider の内部で使用される割り当て関数には、`OPENSSL_malloc()`、`OPENSSL_free()`、`OPENSSL_zalloc()`、`OPENSSL_realloc()` があります。

5.3 ログ出力

wolfProvider はデフォルトで `fprintf()` により stderr にログを出力します。アプリケーションは、カスタムロギング関数を登録することでこれをオーバーライドできます。詳しくは 5 章をご覧ください。

wolfProvider をコンパイルする際、以下のマクロを追加することでログの動作を調整できます。

WOLFPROVIDER_USER_LOG - ログ出力の関数名を定義するマクロ。お客様はこれを fprintf の代わりに使用するカスタムログ関数として定義できます。

WOLFPROVIDER_LOG_PRINTF - `fprintf(stderr)` ではなく、代わりに `printf(stdout)` を使用するように定義します。`WOLFPROVIDER_USER_LOG` またはカスタムロギングコールバックを使用している場合は適用されません。

6 wolfProvider のロード

6.1 OpenSSL をエンジンを利用できるように構成

アプリケーションが OpenSSL エンジンを使用および使用する方法については、OpenSSL のドキュメントをご参照ください。

- OpenSSL 1.0.2
- OpenSSL 1.1.1

アプリケーションがエンジンを使用するための方法はいくつかあります。最も単純なものとして、OpenSSL にバンドルされているすべての PROVIDER 実装をロードして登録し、アプリケーションで次のコードを呼び出す方法があります。(上記の OpenSSL ドキュメントから引用しています)

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

/* Load all bundled PROVIDERs into memory and make them visible */
PROVIDER_load_builtin_providers();

/* Register all of them for every algorithm they collectively implement */
PROVIDER_register_all_complete();
```

アプリケーションが OpenSSL 設定ファイルを使用するように設定されている場合、追加のプロバイダー設定ステップをそこで行うことができます。OpenSSL を構成する方法については、OpenSSL ドキュメントをご覧ください。

- OpenSSL 1.0.2 ドキュメント
- OpenSSL 1.1.1 ドキュメント

アプリケーションはデフォルトの OpenSSL 構成ファイル (`openssl.cnf`) や、`OPENSSL_CONF` 環境変数によって設定された構成の `[openssl_conf]` セクションを呼び出して、読み取り、使用できます。

```
OPENSSL_config(NULL);
```

OpenSSL コンフィギュレーションファイルを使用する代わりに、アプリケーションは `PROVIDER_*` API を使用して明示的に wolfProvider の初期化やアルゴリズムの登録を行うこともできます。一例として、wolfProvider の初期化とすべてのアルゴリズムの登録を行う場合を以下に示します。

```
PROVIDER* e = NULL;

e = PROVIDER_by_id( “ wolfprovider ” );
if (e == NULL) {
    printf( “ Failed to find wolfProvider\n ” );
    /* error */
}
PROVIDER_set_default(e, PROVIDER_METHOD_ALL);

/* アプリケーションの実装 */

PROVIDER_finish(e);
PROVIDER_cleanup();
```

6.2 OpenSSL コンフィギュレーションファイルによる wolfProvider のロード

OpenSSL を使用するアプリケーションがコンフィギュレーションファイルを処理するように設定されている場合、wolfProvider は OpenSSL コンフィギュレーションファイルからロードできます。

wolfProvider ライブラリをコンフィギュレーションファイルに追加する方法の例を以下に示します。[wolfssl_section] は、必要に応じてエンジン制御コマンド (enable_debug) を設定するように変更できます。

```
openssl_conf = openssl_init

[openssl_init]
providers = provider_section

[provider_section]
wolfSSL = wolfssl_section

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

6.3 wolfProvider 静的エントリポイント

wolfProvider を静的ライブラリとして使用する場合、アプリケーションは次のエントリポイントを呼び出して wolfProvider をロードできます。

```
#include <wolfprovider/wp_wolfprovider.h>
PROVIDER_load_wolfprovider();
```

7 wolfProvider の設計

wolfProvider は次のソースファイルで構成され、すべて wolfProvider パッケージの `src` サブディレクトリの下にあります。

ソースファイル	詳細
wp_wolfprovider.c	ライブラリエントリポイントが含まれます。OpenSSL エンジンフレームワークを使用してライブラリを動的にロードするために、OpenSSL IMPLEMENT_DYNAMIC_BIND_FN を呼び出します。コンパイルして静的ライブラリとして使用する場合のスタティックエントリポイントも含みます。
wp_internal.c	エンジンアルゴリズムコールバックの登録を処理する `wolfprovider_bind()` 関数が含まれています。他の wolfProvider の内部機能も含みます。
wp_logging.c	wolfProvider ログ出力フレームワークと関数の実装
wp_openssl_bc.c	wolfProvider OpenSSL バイナリ互換抽象化レイヤーです。複数の OpenSSL バージョンで wolfProvider をサポートするために使用します。
wp_aes_block.c	wolfProvider AES-ECB および AES-CBC 実装
wp_aes_cbc_hmac.c	wolfProvider AES-CBC-HMAC 実装
wp_aes_ccm.c	wolfProvider AES-CCM 実装
wp_aes_ctr.c	wolfProvider AES-CTR 実装
wp_aes_gcm.c	wolfProvider AES-GCM 実装
wp_des3_cbc.c	wolfProvider 3DES-CBC の実装
wp_dh.c	wolfProvider DH の実装
wp_digest.c	wolfProvider メッセージダイジェストの実装 (SHA-1、SHA-2、SHA-3)
wp_ecc.c	wolfProvider ECDSA および ECDH の実装
wp_mac.c	wolfProvider HMAC および CMAC の実装
wp_random.c	wolfProvider RAND 実装
wp_rsa.c	wolfProvider RSA 実装
wp_tls_prf.c	wolfProvider TLS 1.0 PRF 実装

7.1 wolfProvider エントリーポイント

wolfProvider ライブラリへの主なエントリポイントは、**wolfprovider_bind** () または **PROVIDER_load_wolfprovider** () のいずれかです。wolfProvider が動的にロードされている場合、`wolfprovider_bind()` は OpenSSL によって自動的に呼び出されます。`PROVIDER_load_wolfprovider()` は wolfProvider 静的に構築および使用されている場合に、アプリケーションが呼び出す必要があるエントリポイントです。

7.2 wolfProvider アルゴリズムコールバック登録

wolfProvider は、FIPS 版 wolfCrypt でサポートしているすべてのコンポーネントに対して、アルゴリズム構造体とコールバックを OpenSSL エンジンフレームワークに登録します。この登録は、`wp_internal.c` の `wolfprovider_bind()` 内で行われます。`wolfprovider_bind()` は、wolfProvider エンジンを表す `PROVIDER` 構造体ポインタを受け取ります。次に、個々のアルゴリズム/コンポーネントのコールバックまたは構造体が、`<openssl/provider.h>` の適切な API を使用してその `PROVIDER` 構造体に登録されます。

これらの API 呼び出しには、以下のものが含まれます。

```
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)
```

上記の呼び出しで使用される各アルゴリズム/コンポーネントのコールバック関数または構造体 (例: wp_digests、wp_ciphers など) は、`wp_internal.c` またはそれぞれのアルゴリズムソースファイルに実装しています。

8 その他のオープンソースソフトウェアとの統合に関する注意事項

wolfProvider は、一般的な OpenSSL エンジンフレームワークとアーキテクチャに準拠しています。そのため、OpenSSL を使用するアプリケーションから OpenSSL 構成ファイルを介して、または PROVIDER API 呼び出しを介して、他のエンジン実装と同様に wolfProvider をプログラムで利用できます。

wolfSSL は、いくつかのオープンソースプロジェクトで wolfProvider をテストしました。この章には、wolfProvider とのインテグレーションに関する注意事項とヒントを示します。ただし、すべてのオープンソースプロジェクトを網羅しているわけではありません。今後も随時、wolfSSL またはコミュニティが追加のオープンソースプロジェクトで wolfProvider の動作を確認し追記します。

8.1 cURL

cURL はすでに OpenSSL 構成ファイルを利用するようにセットアップされています。wolfProvider を利用するには、次のステップを実行してください。

1. wolfProvider エンジン情報を OpenSSL 設定ファイルに追加します
2. 必要に応じて、OPENSSL_CONF 環境変数が OpenSSL 設定ファイルを指すように設定します

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

3. OPENSSL_PROVIDERS 環境変数を wolfProvider 共有ライブラリファイルの場所を指すように設定します

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

8.2 stunnel

stunnel は wolfProvider でテスト済みです。詳細は追って更新いたします。

8.3 OpenSSH

OpenSSH は、--with-ssl-provider 構成オプションを使用して、OpenSSL エンジンサポートでコンパイルする必要があります。必要に応じて--with-ssl-dir=DIR を使用して、使用されている OpenSSL ライブラリのインストール場所を指定することもできます。

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

OpenSSH には、wolfProvider を活用するための OpenSSL 構成ファイルのセットアップも必要です。必要に応じて、OPENSSL_CONF 環境変数を構成ファイルを指すように設定できます。OPENSSL_PROVIDERS 環境変数も、wolfProvider 共有ライブラリの場所に設定する必要があります。

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

9　サポートと OpenSSL バージョン追加

wolfProvider のサポートについては、info@wolfssl.jp までお問い合わせください。サポートが必要となる OpenSSL バージョンの追加を希望される場合も、ぜひお知らせください。