wolfSSL Java JNI and JSSE Provider Documentation

2025-04-22

Contents

1 イントロダクション

wolfSSL JNI/JSSE は、Java Secure Socket Extension のプロバイダー実装です。また、ネイティブの wolfSSL SSL/TLS ライブラリの薄い JNI ラッパーも含んでいます。

Java Secure Socket Extension (**JSSE**) フレームワークは、セキュリティプロバイダのインストールをサポートしています。セキュリティプロバイダーは、SSL/TLS など、Java JSSE セキュリティ API で使用される機能のサブセットを実装できます。

このドキュメントでは、wolfSSL の JSSE プロバイダーの実装 "**wolfJSSE**" について説明しています。wolfJSSE は、ネイティブの wolfSSL SSL/TLS ライブラリをラップします。このインターフェースにより、Java アプリケーションは TLS 1.3までの現在の SSL/TLS 標準、FIPS 140-2 および 140-3 サポート、パフォーマンスの最適化、ハードウェア暗号化のサポート、商用サポート等々の wolfSSL を使用して得られるすべての利点を享受できます。

wolfJSSE は、"**wolfssljni**" パッケージの一部として配布されます。このパッケージには、wolfSSL 用の薄い JNI ラッパーと wolfJSSE プロバイダーの両方が含まれています。

2 システム要件

2.1 Java / JDK

wolfJSSE では、ホスト システムに Java をインストールする必要があります。ユーザーと開発者が利用できる JDK バリアントがいくつかあります。wolfSSL JNI/JSSE では以下についてテスト済みです：

- Unix/Linux:
  - Oracle JDK
  - OpenJDK
  - Zulu JDK
  - Amazon Coretto
- Android

wolfSSL JNI/JSSE のビルド システムは、現時点で Microsoft Windows で実行するようにセットアップされていません。この件について興味がある方はfacts@wolfssl.comまでお問い合わせください。

“IDE/Android” の下に含まれる Android Studio のサンプルプログラムプロジェクトは、Linux と Windows の両方でテストされています。

2.2 JUnit

ユニットテストを実行するには、開発システムに JUnit 4 がインストールされている必要があります。JUnit は、プロジェクトの Web サイトwww.junit.org からダウンロードできます。

Unix/Linux/OSX システムに JUnit をインストールするには:

1) junit.org/junit4/ から “**junit-4.13.2.jar**” と” **hamcrest-all-1.3.jar**” をダウンロードします。執筆時点では、上記の.jar ファイルは次のリンクからダウンロードできます。

   junit-4.13.2.jar hamcrest-all-1.3.jar

2) ダウンロードした Jar ファイルをシステムに配置したらそのパスを **JUNIT_HOME** にセットします。例えば：

```
$ export JUNIT_HOME=/path/to/jar/files
```

2.3 システム要件 (gcc、ant)

gcc と **ant** がそれぞれ C コードと Java コードのコンパイルに使用されます。開発システム上に上記がインストールされていることを確認してください。

注意事項: `java.sh` スクリプトは、Java のインストールフォルダとして一般的なロケーションを使用します。Java のインストールフォルダが異なる場合、`java.sh` の実行時にエラーが発生する可能性があります。この場合、`java.sh` を環境に合わせて変更する必要があります。

2.4 wolfSSL SSL/TLS ライブラリ

wolfSSL JNI/JSSE のコンパイルに先立ち、ネイティブ wolfSSL ライブラリのラッパーとして、wolfSSL C ライブラリをホスト プラットフォームにインストールし、インクルードおよびライブラリ検索パスに配置する必要があります。

2.4.1 wolfSSL と wolfCrypt C ライブラリのコンパイル

wolfJSSE で使用するために Unix/Linux 環境で wolfSSL をコンパイルしてインストールするには、wolfSSL マニュアルのビルド手順に従ってください。wolfSSL をコンパイルする最も一般的な方法は、Autoconf を使用することです。

Autoconf を使用して wolfSSL を設定する場合、--enable-jni オプションを使用する必要があります:

```
$ cd wolfssl-X.X.X
$ ./configure --enable-jni
$ make
```

“make check” が正常にパスすることを確認してから、ライブラリをインストールします:

```
$ make check
$ sudo make install
```

これにより、システムのデフォルトのインストールフォルダに wolfSSL ライブラリがインストールされます。多くのプラットフォームでは、次のフォルダです:

```
/usr/local/lib
/usr/local/include
```

wolfSSL が非標準のライブラリ インストールフォルダにインストールされている場合、LD_LIBRARY_PATH (Unix/Linux) または DYLD_LIBRARY_PATH (OSX) を更新する必要がある場合があります：

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/wolfssl/install
```

3 wolfSSL JNI と wolfJSSE のコンパイル

wolfJSSE をコンパイルする方法を 3 つ、ここで紹介します。Unix コマンドライン、Android Studio ビルド、汎用 IDE ビルドを使用します。

3.1 Unix コマンドライン

このセクションの手順を実行する前に、第 2 章に記載のシステム要件とされているものがインストールされていることを確認してください。

パッケージのルートフォルダの `java.sh` スクリプトは、ネイティブの JNI C ソース ファイルをコンパイルして Unix/Linux または Mac OSX 用の共有ライブラリとするために使用されます。

このスクリプトは、OSX (Darwin) から Linux までの OS を自動検出してインクルード パスと共有ライブラリ拡張タイプをセットアップしようとします。さらに、このスクリプトは JNI C ソース ファイルに対して gcc を直接呼び出して、`./lib/libwolfssljni.so` または `./lib/libwolfssljni.jnilib` を生成します。

```
$ ./java.sh

Compiling Native JNI library:
    WOLFSSL_INSTALL_DIR = /usr/local
    Detected Linux host OS
        Linux x86_64
    Java Home = /usr/lib/jvm/java-8-openjdk-amd64
    Generated ./lib/libwolfssljni.so
```

`/usr/local` にインストールされていないネイティブ wolfSSL ライブラリに対してリンクする場合には、wolfSSL インストールフォルダを表す引数を `java.sh` に渡す必要があります。

例えば：

```
$ ./java.sh /path/to/wolfssl/install
```

Java ソースファイルのビルドには `ant` を使います:

```
$ ant
```

`ant` に対して指定可能なビルドターゲット：

- `ant` (アプリケーションに必要な jar のみビルド）
- `ant test` (jar とテスト実行に必要なテストをビルド。要 JUNIT のセットアップ）
- `ant examples` (jar とサンプルプログラムをビルド)
- `ant clean` (Java アーティファクトをクリーンアップ)
- `ant cleanjni` (ネイティブアーティファクトをクリーンアップ)

次のコマンドは、プロジェクトで wolfJSSE を使用するために必要な wolfJSSE jar とネイティブ コードをビルドします。JUnit テストをコンパイルして実行するには、コマンド `ant test` を使用します：

```
$ ant test
```

コマンドを実行すると、テストがコンパイルされ、主な wolfJSSE コードとテスト結果の出力が、wolfJSSE テストスイートと wolfSSL JNI テストスイートの最後に合格した全テストの要約とともに表示されます。ビルドが成功すると、最後に “BUILD SUCCESSFUL” というメッセージが表示されます。

```
[junit] WolfSSLTrustX509 Class
[junit] Testing parse all_mixed.jks ... passed
[junit] Testing loading default certs ... passed
```

```
[junit] Testing parse all.jks ... passed
[junit] Testing verify ... passed
...

build:

BUILD SUCCESSFUL
Total time: 18 seconds
```

wolfJSSE にバンドルされているサンプルプログラムをビルドして実行するためには `ant examples` を使います：

```
$ ant examples
```

3.2 Android Studio を使ってのビルド

Android Studio プロジェクトが、ディレクトリ `IDE/Android` に用意してあります。これは、wolfssljni / wolfJSSE の Android Studio プロジェクト ファイルのサンプルプログラムです。このプロジェクトは参照用としてのみ使用してください。

テスト時に使用されるツールとバージョン情報の詳細については、`wolfssljni/IDE/Android/README.md` を参照してください。次の手順は、Android デバイスまたはエミュレーターでこのサンプルプログラムを実行するために必要です。

3.2.1 1. ネイティブ wolfSSL ライブラリのソースをプロジェクトに追加

このサンプルプロジェクトは、ネイティブ wolfSSL ライブラリのソースファイルをコンパイルしてビルドするように既に設定されています。ただし、wolfSSL ファイル自体はパッケージに含まれていないので、適切なバージョンをダウンロードしてリンクする必要があります。以下のオプションのいずれかを使用して、このプロジェクトに wolfSSL を追加します。

プロジェクトは wolfSSL ソースコードのディレクトリを `wolfssljni/IDE/Android/app/src/main/cpp/wolfssl` から探します。

これは複数の方法で追加できます：

- オプション A: www.wolfssl.com から最新の wolfSSL ライブラリ リリースをダウンロードします。解凍し、名前を “wolfssl” に変更して、ディレクトリ `wolfssljni/IDE/Android/app/src/main/cpp/` に配置します。

```
$ unzip wolfssl-X.X.X.zip
$ mv wolfssl-X.X.X wolfssljni/IDE/Android/app/src/main/cpp/wolfssl
```

- オプション B: GitHub を使用して wolfSSL をクローンすることもできます：

```
$ cd /IDE/Android/app/src/main/cpp/
$ git clone https://github.com/wolfssl/wolfssl
$ cp wolfssl/options.h.in wolfssl/options.h
```

- オプション C: システム上の wolfSSL ディレクトリへのシンボリック リンクを作成:

```
$ cd /IDE/Android/app/src/main/cpp/
$ ln -s /path/to/local/wolfssl ./wolfssl
```

3.2.2 2. Java シンボリックリンクを更新 (Windows ユーザーのみ必要)

次の Java ソース ディレクトリは、Unix/Linux の symlink です：

```
wolfssljni/IDE/Android/app/src/main/java/com/wolfssl
```

これは Windows では正しく機能しないので、新しい Windows シンボリック リンクを作成する必要があります：

1) Windows コマンド プロンプトを開きます (右クリックし、管理者として実行)。
2) `wolfssljni\IDE\Android\app\src\main\java\com` に移動。
3) 既存のシンボリックリンクファイルを削除します (“wolfssl” という名前のファイルとして表示されます)。

```
del wolfssl
```

4) `mklink` で新しい相対シンボリック リンクを作成します:

```
mklink /D wolfssl ..\..\..\..\..\..\..\src\java\com\wolfssl\
```

3.2.3 3. サンプルプログラムの JKS ファイルを Android 用の BKS に変換

Android デバイスでは、BKS 形式のキーストアが想定されています。JKS サンプルバンドルを BKS に変換するには、次のコマンドを使用します (注: Bouncy Castle の Web サイトから bcprov JAR のバージョンをダウンロードする必要があります):

```
cd examples/provider
./convert-to-bks.sh <path/to/provider>
```

例えば、bcprov-ext-jdk15on-169.jar を使用する場合:

```
cd examples/provider
./convert-to-bks.sh ~/Downloads/bcprov-ext-jdk15on-169.jar
```

3.2.4 4. BKS ファイルを Android デバイス/エミュレータにプッシュ

BKS バンドルを証明書とともにデバイスにプッシュします。エミュレーター/デバイスを起動し、“adb push” を使用します。例として、wolfssljni のルートディレクトリからの以下の様なコマンドを実行します。この手順は、Android Studio を起動してプロジェクトをコンパイルした後に行うことができますが、アプリまたはテスト ケースを実行する前に行う必要があります。

```
adb shell
cd sdcard
mkdir examples
mkdir examples/provider
mkdir examples/certs
exit
adb push ./examples/provider/*.bks /sdcard/examples/provider/
adb push ./examples/certs/ /sdcard/examples/
adb push ./examples/certs/intermediate/* /sdcard/examples/certs/intermediate/
```

3.2.5 5. サンプルプログラムプロジェクトを Android Studio にインポートしてビルド

1) wolfssljni/IDE/ の “Android” フォルダをダブルクリックして、Android Studio プロジェクトを開きます。または、Android Studio 内から、wolfssljni/IDE ディレクトリにある “Android” プロジェクトを開きます。

2) プロジェクトをビルドし、アプリ -> java/com/example.wolfssl から MainActivity を実行します。これにより、/sdcard/ ディレクトリ内の証明書にアクセスする許可が求められ、成功するとサーバー証明書情報が出力されます。

3) オプション: androidTests は、許可が与えられた後に実行できます。app->java->com.wolfssl->provider.jsse.test->WolfSSLJSSETestSuite および app->java->com.wolfssl->test->WolfSSLTestSuite

3.3 汎用 IDE でビルド

一般的な IDE ビルドの場合、IDE で新しいプロジェクトを作成し、`src/java` からソース ファイルを追加します。以下のパッケージになります：

```
com.wolfssl
com.wolfssl.provider.jsse
com.wolfssl.wolfcrypt
```

コマンド ラインから java.sh を実行するか、IDE で `java.sh` を実行して、wolfSSL にリンクするネイティブ シム レイヤーを生成します。

プロジェクトにネイティブ ライブラリ参照を追加します。それは lib にあるはずです libwolfssl.jnilib のディレクトリ (例: wolfssljni/lib/)。

テストケースをコンパイルするには､ディレクトリ `src/test` からパッケージ `com.wolfssl.provider.jsse.test` と `com.wolfssl.test` を追加します。プロジェクトには、テストを実行するための Junit も必要です。

サンプルプログラムをさらに追加することもできます。その場合は、`examples/provider/` のソース コードをプロジェクトに追加します。オプションで、IDE は “examples/provider/ClientJSSE.sh” を実行できます。サンプルに追加するのが難しい部分の 1 つは、デフォルトのキーストアを使用しようとする場合に、サンプルを実行するときにキーストアへのパスが IDE に認識されるようにすることです。

4 インストール

wolfJSSE をインストールして使用するには、実行時に行うかあるいはシステム レベルでグローバルでインストールするかの 2 つの方法があります。

4.1 実行時インストール

実行時に wolfJSSE をインストールして使用するには、まず “libwolfssljni.so” がシステム ライブラリの検索パスに存在していることを確認してください。Linux では、このパスを次のように変更できます:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/add
```

OSX では、LD_LIBRARY_PATH の代わりに DYLD_LIBRARY_PATH を使用します。

次に、wolfSSL JNI / wolfJSSE JAR ファイル (wolfssl.jar、wolfssl-jsse.jar) を Java クラスパスに配置します。このために、システムのクラスパス設定を調整するか、特定のアプリケーションのコンパイル時および実行時に次のようにします：

```
$ javac -classpath <path/to/jar> ...
$ java -classpath <path/to/jar> ...
```

最後に、Java アプリケーションで、プロバイダ クラスをインポートして Security.addProvider() を呼び出すことにより、実行時にプロバイダを追加します：

```
import com.wolfssl.provider.jsse.WolfSSLProvider;

public class TestClass {
    public static void main(String args[]) {
        ...
        Security.addProvider(new WolfSSLProvider());
        ...
    }
}
```

インストールされているすべてのセキュリティプロバイダのリストを確認用に出力するには、次の手順を実行します：

```
Provider[] providers = Security.getProviders();
for (Provider prov:providers) {
    System.out.println(prov);
}
```

4.2 OS / システムレベルでのインストール

4.2.1 Unix/Linux

システム/OS レベルで wolfJSSE プロバイダーをインストールするには、“wolfssl.jar” および/または “wolfssl-jsse.jar” を OS の正しい Java インストール ディレクトリにコピーし、“libwolfssljni.so” または “libwolfssljni.jnilib” 共有ライブラリがライブラリ検索パスに存在することを確認します。

JAR ファイル (wolfssl.jar、wolfssl-jsse.jar) と共有ライブラリ (libwolfssljni.so) を次のディレクトリに追加します：

```
$JAVA_HOME/jre/lib/ext
```

OpenJDK を使用する Ubuntu では、このパスは次のようになります：

```
/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/ext
```

さらに、次のエントリを java.security ファイルに追加します：

```
security.provider.N=com.wolfssl.provider.jsse.WolfSSLProvider
```

java.security ファイルは次の場所にあります:

```
$JAVA_HOME /jre/lib/security/java.security
```

“N” を、ファイル内の他のプロバイダーと比較して wolfJSSE に持たせたい優先順位に置き換えます。WolfSSLProvider を最優先プロバイダとして配置するには、次の行を java.security ファイルのプロバイダリストに追加します。また、java.security ファイルにリストされている他のプロバイダーの優先番号を付け直す必要があります。最高の優先度は「1」です。

```
security.provider.1=com.wolfssl.provider.jsse.WolfSSLProvider
```

4.2.2 Android OSP (AOSP)

wolfJSSE をシステム セキュリティ プロバイダーとして Android OSP (AOSP) ソース ツリーにインストールする手順については、別のドキュメント「Installing a JSSE Provider in Android OSP」を参照してください。

5 パッケージ構成

wolfJSSE は、wolfSSL JNI ラッパーと一緒に “**wolfssljni**” パッケージにバンドルされています。wolfJSSE は wolfSSL の基礎となる JNI バインディングに依存するため、JNI ラッパーと同じネイティブ ライブラリファイルにコンパイルされます。

wolfJSSE / wolfSSL JNI パッケージ構成は以下の通りです:

```
wolfssljni/
    build.xml  ant build script
    COPYING
    docs/      Javadocs
    examples/  Example apps
    IDE/       Example IDE project, Android Studio
    java.sh    Script to build native C JNI sources
    LICENSING
    Makefile
    lib/       Output directory for compiled library
    native/    Native C JNI binding source files
    platform/  Android AOSP build files
    README.md
    rpm/       rpm spec files
    src/
        java/  Java source files
        test/  Test source files
```

wolfJSSE プロバイダーのソース コードは、`src/java/com/wolfssl/provider/jsse` ディレクトリにあり、“**com.wolfssl.provider.jsse**” Java パッケージの一部です。

wolfSSL JNI ラッパーは `src/java/com/wolfssl` ディレクトリにあり、“**com.wolfssl**” Java パッケージの一部です。このパッケージは wolfJSSE クラスによって利用されるため、JSSE のユーザーはこのパッケージを直接使用する必要はありません。

wolfSSL JNI と wolfJSSE がコンパイルされると、2 つの JAR ファイルと 1 つのネイティブ共有ライブラリが `./lib` ディレクトリに生成されます。オペレーティングシステムに応じて異なりますが、ネイティブ共有ライブラリには `libwolfssljni.jnilib` と名前をつけることもできます。

```
lib/
    libwolfSSL.so     (Native C JNI wrapper shared library)
    wolfssl.jar       (JAR with ONLY wolfSSL JNI Java classes)
    wolfssl-jsse.jar  (JAR with BOTH wolfSSL JNI and wolfJSSE classes)
```

6 サポートしているアルゴリズムとクラス

wolfJSSE は現在、次の JSSE クラスの実装を提供しています：

- SSLContext
  - TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3
- SSLEngine
- SSLSession
- SSLSocket
- SSLServerSocket
- SSLSocketFactory
- SSLServerSocketFactory
- KeyManagerFactory
- X509KeyManager
- TrustManagerFactory
- X509TrustManager
- X509Certificate

7 使用方法

使用方法については、前の章で指定されたクラスの Oracle/OpenJDK Javadoc に従ってください。“wolfJSSE” プロバイダーが、 `java.security` ファイルで同じアルゴリズムを提供している他のプロバイダーよりも優先順位が低く設定されている場合は、明示的に “wolfJSSE” プロバイダーを使用することを要求する必要あります。

たとえば、TLS 1.2 の SSLContext クラスで wolfJSSE プロバイダーを使用するには アプリケーションは次のように SSLContext オブジェクトを作成します：

```
SSLContext ctx = SSLContext.getInstance(“ TLSv1.2 ”, “ wolfJSSE ”);
```

8 サンプルプログラム

9 wolfSSL JNI サンプルプログラム

“examples” ディレクトリには、wolfSSL シン JNI ラッパーのサンプルプログラムが含まれています。wolfSSL JSSE プロバイダーのサンプルプログラムは ./examples/provider ディレクトリにあります。

サンプルプログラムは、パッケージのルート ディレクトリから、提供されたラッパー スクリプトを使用して実行する必要があります。ラッパー スクリプトは、wolfssljni パッケージに含まれる wolfjni jar で使用するための正しい環境変数を設定します。

9.1 デバッグとログに間する注意事項

実行時に -Dwolfjsse.debug=true‘ を使用することで、wolfJSSE デバッグログ出力を有効にできます。

ネイティブ wolfSSL が “–enable-debug” でコンパイルされている場合、実行時に “-Dwolfssl.debug=true” を使用して wolfSSL ネイティブ デバッグログ出力を有効にできます。

-Djavax.net.debug=all オプションを使用して、JDK デバッグログ出力を有効にできます。

9.2 wolfSSL JNI サンプルクライアントとサンプルサーバー

wolfSSL JNI を使用するクライアント/サーバー アプリケーションのサンプルプログラム:

Server.java - wolfSSL JNI サーバーサンプルプログラム **Client.java** - wolfSSL JNI クライアントサンプルプログラム

これらのサンプルプログラムは、提供されている bash スクリプトを使って実行できます：

```
$ cd <wolfssljni_root>
$ ./examples/server.sh <options>
$ ./examples/client.sh <options>
```

サンプルプログラムの使用方法と使用可能なオプションを表示するには、“-?” を指定します:

```
$ ./examples/server.sh --help
```


10 wolfJSSE Provider サンプルプログラム

examples/provider ディレクトリには、wolfSSL JSSE プロバイダー (wolfJSSE) のサンプルプログラムが含まれています。

サンプルプログラムは、パッケージのルート ディレクトリから、提供されたラッパー スクリプトを使用して実行する必要があります。ラッパー スクリプトは、wolfssljni パッケージに含まれる wolfJSSE プロバイダーで使用するための環境変数を正しく設定します。たとえば、サンプルの JSSE サーバーとクライアントを実行するには、wolfSSL と wolfssljni をコンパイルした後:

```
$ cd <wolfssljni_root>
$ ./examples/provider/ServerJSSE.sh
$ ./examples/provider/ClientJSSE.sh
```

10.1 デバッグとログ出力に間する注意事項

wolfJSSE デバッグ ログは、実行時に -Dwolfjsse.debug=true‘ を使用して有効にできます。

ネイティブ wolfSSL が --enable-debug でコンパイルされている場合、実行時に -Dwolfssl.debug=true‘ を使用して wolfSSL ネイティブ デバッグ ロギングを有効にできます。

-Djavax.net.debug=all オプションを使用して、JDK デバッグ ロギングを有効にできます。

10.2 wolfJSSE Example Client and Server

SSLSocket API とともに wolfJSSE を使用するクライアント/サーバー アプリケーションのサンプルプログラムです。

ServerJSSE.java - wolfJSSE サーバーサンプルプログラム
ClientJSSE.java - wolfJSSE クライアントサンプルプログラム

これらのサンプルプログラムは、提供されている bash スクリプトで実行できます：

```
$ ./examples/provider/ServerJSSE.sh <options>
$ ./examples/provider/ClientJSSE.sh <options>
```

10.3 ClientSSLSocket.java

SSLSocket を使用した非常に最小限の JSSE クライアントのサンプルプログラムです。ClientJSSE.java が行うすべてのオプションをサポートしているわけではありません。

使用例は次のとおりです：

```
$ ./examples/provider/ClientSSLSocket.sh [host] [port] [keystore] [truststore]
```

wolfSSL サンプル サーバーに接続するための使用例は次のとおりです:

```
$ ./examples/provider/ClientSSLSocket.sh 127.0.0.1 11111 \
  ./examples/provider/client.jks ./examples/provider/ca-server.jks
```

client.jks のパスワード は: “wolfSSL test”

10.4 MultiThreadedSSLClient.java

指定された数のクライアントスレッドをサーバーに接続するマルチスレッド SSLSocket のサンプルプログラムです。wolfJSSE を使用したマルチスレッドのテストを目的としています。

このサンプルプログラムでは、127.0.0.1:11118 にあるサーバーに対して、指定された数のクライアント スレッドを作成します。このサンプルプログラムは、SSLSocket クラスを使用するように設定されています。1 つの接続 (ハンドシェイク) を行い、データを送受信し、シャットダウンします。

次のハンドシェーク実行前に、ランダムな時間が各クライアント スレッドに挿入されます。

1) SSL/TLS ハンドシェイク
2) ハンドシェイク後に I/O 操作を行う

それぞれの最大スリープ時間は “maxSleep” で、デフォルトでは 3 秒です。これは、クライアント スレッド操作にランダム性を追加することを目的としています。

使用例:

```
$ ant examples
$ ./examples/provider/MultiThreadedSSLClient.sh -n <num_client_threads>
```

このサンプルプログラムは、MultiThreadedSSLServer のサンプルプログラムに接続するように設計されています。

```
$ ./examples/provider/MultiThreadedSSLServer.sh
```

このサンプルプログラムでは、平均 SSL/TLS ハンドシェイク時間も出力されます。これは、“startHandshake()” API 呼び出しでミリ秒単位で測定されます。

10.5 MultiThreadedSSLServer.java

クライアント接続ごとに新しいスレッドを作成する SSLServerSocket のサンプルプログラムです。

このサーバーはクライアント接続を無限ループで待機し、接続されると接続ごとに新しいスレッドを作成します。このサンプルプログラムは、パッケージ ルートで “ant examples” を実行するとコンパイルされます。

```
$ ant examples
$ ./examples/provider/MultiThreadedSSLServer.sh
```

マルチスレッド クライアントのテストでは、MultiThreadedSSLClient.sh に対してテストします。たとえば、10 個のクライアントスレッドを接続するには、次のようにします：

```
$ ./examples/provider/MultiThreadedSSLClient.sh -n 10
```

10.6 ProviderTest.java

このサンプルプログラムでは、wolfSSL プロバイダーのインストールをテストします。システムにインストールされているすべてのプロバイダーを一覧表示し、wolfSSL プロバイダーの検索を試み、見つかった場合は、wolfSSL プロバイダーに関する情報を出力します。最後に、Java に TLS を提供するために登録されているプロバイダーをテストします。

このアプリは、wolfJSSE がシステムレベルで正しくインストールされているかどうかをテストするのに役立ちます。

```
$ ./examples/provider/ProviderTest.sh
```

wolfJSSE が OS システム レベルでインストールされていない場合、このサンプルプログラムを実行しても wolfJSSE はインストールされたプロバイダーとして表示されないことに注意してください。

10.7 ThreadedSSLSocketClientServer.java

クライアント スレッドをサーバー スレッドに接続する SSLSocket のサンプルプログラム。

このサンプルプログラムでは、1 つのサーバースレッドと 1 つのクライアントスレッドの 2 つのスレッドを作成します。サンプルプログラムは、SSLSocket および SSLServerSocket クラスを使用するように設定されています。両スレッドで互いに通信に 1 つの接続 (ハンドシェイク) とシャットダウンを実行します。

使用例:

```
$ ./examples/provider/ThreadedSSLSocketClientServer.sh
```