wolfSSL Java JNI and JSSE Provider Documentation

2025-04-30

Contents

1 Introduction

wolfSSL JNI/JSSE is a provider implementation of the Java Secure Socket Extension (JSSE). It also includes a thin JNI wrapper around the native wolfSSL SSL/TLS library. If you are instead looking for a JCE (Java Cryptography Extension) provider, see wolfCrypt JNI/JCE. wolfSSL actively maintains both of these providers (JSSE and JCE).

The Java Secure Socket Extension (**JSSE**) framework supports the installation of security providers. These providers can implement a subset of the functionality used by the Java JSSE security APIs, including SSL/TLS.

This document describes wolfSSL's JSSE provider implementation, named "**wolfJSSE / wolfSSL-Provider**". wolfJSSE wraps the native wolfSSL SSL/TLS library. This interface gives Java applications access to all the benefits of using wolfSSL, including current SSL/TLS standards up to TLS 1.3, FIPS 140-2 and 140-3 support, performance optimizations, hardware cryptography support, commercial support, and more!

wolfJSSE is distributed as part of the " **wolfssljni** "package.

2 Requirements

2.1 Java / JDK

wolfJSSE requires Java to be installed on the host system. There are several JDK variants available to users and developers. wolfSSL JNI/JSSE has been tested on the following:

- Unix/Linux:
  - Oracle JDK
  - OpenJDK
  - Zulu JDK
  - Amazon Coretto
- Mac OSX
- Windows (Visual Studio)
- Android Studio
- Android AOSP

2.2 JUnit

In order to run the unit tests, JUnit 4 is required to be installed on the development system. JUnit can be downloaded from the project website at www.junit.org.

To install JUnit4 on a Unix/Linux/OSX system:

1) Download " **junit-4.13.2.jar** " and " **hamcrest-all-1.3.jar** "from junit.org/junit4/. At the time of writing, the above mentioned .jar files could be downloaded from the following links:

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

2) Place these JAR files on your system and set **JUNIT_HOME** to point to that location. Ex:

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

2.3 System Dependencies (gcc and ant)

gcc and **ant** are used to compile native C code and Java code, respectively. Please ensure that these are installed on your development machine. Alternatively, you can choose to compile with **maven** instead of **ant**.

2.4 wolfSSL SSL/TLS Library

wolfSSL JNI/JSSE is a wrapper around the native wolfSSL library. As such, the wolfSSL C library must be installed on the host platform and placed on the include and library search paths before compiling wolfSSL JNI/JSSE.

2.4.1 Compiling wolfSSL and wolfCrypt C Library

To compile and install wolfSSL in a Unix/Linux environment for use with wolfJSSE, please follow build instructions in the wolfSSL Manual. The most common way to compile wolfSSL is using Autoconf.

When using Autoconf to configure wolfSSL, the `--enable-jni` option will need to be used:

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

Verify `make check` passes successfully, then install the library:

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

This will install the wolfSSL library to your system default installation location. On many platforms this is:

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

If wolfSSL has been installed to a non-standard library installation location, you may need to update LD_LIBRARY_PATH (Unix/Linux) or DYLD_LIBRARY_PATH (OSX):

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

3 wolfSSL JNI and wolfJSSE Compilation

There are three ways to compile wolfJSSE listed in this chapter, including using the Unix Command Line, Android Studio Build and Generic IDE Build.

3.1 Unix Command Line

Before following steps in this section, please ensure that the dependencies listed in Chapter 2 are installed.

The `java.sh` script in the root package directory is used to compile the native JNI C source files into a shared library for either Unix/Linux or Mac OSX. This script tries to auto-detect between OSX (Darwin) and Linux to set up include paths and shared library extension type. This script directly calls gcc on the JNI C source files, producing `./lib/libwolfssljni.so` or `./lib/libwolfssljni.jnilib`.

```
$ ./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
```

If linking against a native wolfSSL library not installed to `/usr/local`, one argument may be passed to `java.sh` which represents the wolfSSL installation location. For example:

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

To compile the Java sources, `ant` is used:

```
$ ant
```

Available build targets for `ant` are :

- `ant` (only builds the jar necessary for an app to use)
- `ant test` (builds the jar and tests then runs the tests, requires JUNIT setup)
- `ant examples` (builds the jar and example cases)
- `ant clean` (cleans all Java artifacts)
- `ant cleanjni` (cleans native artifacts)

These two commands will build the wolfJSSE jar and native code necessary to use wolfJSSE in a project. To compile and run JUnit tests the command `ant test` is used:

```
$ ant test
```

This will compile tests and the main wolfJSSE code and output with the results of the tests will be displayed along with a summary of total tests that passed at the end of the wolfJSSE testsuite and wolfSSL JNI testsuite. A successful build will display the message “BUILD SUCCESSFUL” at the end.

```
[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
```

To compile and run examples bundled with wolfJSSE the command `ant examples` is used:

```
$ ant examples
```

3.2 Android Studio Build

An example Android Studio project is located in the directory `IDE/Android`. This is an example Android Studio project file for wolfssljni / wolfJSSE. This project should be used for reference only.

More details on tool and version information used when testing can be found in the `wolfssljni/IDE/Android/README.md`. The following steps outline what is required to run this example on an Android device or emulator.

3.2.1 1. Add Native wolfSSL Library Source Code to Project

This example project is already set up to compile and build the native wolfSSL library source files, but the wolfSSL files themselves have not been included in this package. You must download or link an appropriate version of wolfSSL to this project using one of the options below.

The project looks for the directory `wolfssljni/IDE/Android/app/src/main/cpp/wolfssl` for wolfSSL source code. This can added in multiple ways:

- OPTION A: Download the latest wolfSSL library release from www.wolfssl.com, unzip it, rename it to `wolfssl`, and place it in the direcotry `wolfssljni/IDE/Android/app/src/main/cpp/`.

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

- OPTION B: Alternatively GitHub can be used to clone wolfSSL:

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

- OPTION C: A symbolic link to a wolfssl directory on the system by using:

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

3.2.2 2. Update Java Symbolic Links (Only applies to Windows Users)

The following Java source directory is a Unix/Linux symlink:

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

This will not work correctly on Windows, and a new Windows symbolic link needs to be created in this location. To do so:

1) Open Windows Command Prompt (Right click, and “Run as Administrator”)
2) Navigate to `wolfssljni\IDE\Android\app\src\main\java\com`
3) Delete the existing symlink file (it shows up as a file called “wolfssl”)

```
del wolfssl
```

4) Create a new relative symbolic link with `mklink`:

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

3.2.3 3. Convert Example JKS files to BKS for Android Use

On an Android device BKS format key stores are expected. To convert the JKS example bundles to BKS use the following commands. Note: you will need to download a version of the bcprov JAR from the Bouncy Castle website:

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

For exmaple, when using bcprov-ext-jdk15on-169.jar:

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

3.2.4 4. Push BKS files to Android Device or Emulator

Push BKS bundles up to the device along with certificates. To do this start up the emulator/device and use adb push. An example of this would be the following commands from root wolfssljni directory. This step may be done after the starting Android Studio and compiling the project, but must be done before running the app or test cases.

```
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. Import and Build the Example Project with Android Studio

1) Open the Android Studio project by double clicking on the `Android` folder in wolfssljni/IDE/. Or, from inside Android Studio, open the `Android` project located in the wolfssljni/IDE directory.
2) Build the project and run MainActivity from app -> java/com/example.wolfssl. This will ask for permissions to access the certificates in the /sdcard/ directory and then print out the server certificate information on success.
3) OPTIONAL: The androidTests can be run after permissions has been given. app->java->com.wolfssl->provider.jsse.test->WolfSSLJSSETestSuite and app->java->com.wolfssl->test->WolfSSLTestSuite

3.3 Generic IDE Build

For generic IDE builds create a new project in the IDE, then add source files from `src/java`. This will be the following packages:

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

Run java.sh from the command line or have the IDE execute `java.sh` to generate the native shim layer linking against wolfSSL.

Add native library reference to the project. It should look in the lib directory for libwolfssl.jnilib (i.e. wolfssljni/lib/).

To compile test cases add the packages `com.wolfssl.provider.jsse.test` and `com.wolfssl.test` from the directory `src/test`. The project will also need Junit to run the tests.

If adding in optional examples then the source code in `examples/provider/` can be added to the project. Optionally the IDE can execute `examples/provider/ClientJSSE.sh`. One of the difficult parts to adding in the examples is making sure the path to keystores is known to the IDE when it runs the examples, if trying to use the default keystores.

4 Installation

There are two ways that wolfJSSE can be installed and used, either at runtime or globally at the system level.

4.1 Installation at Runtime

To install and use wolfJSSE at runtime, first make sure that `libwolfssljni.so` is on your system library search path. On Linux, you can modify this path with:

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

On OSX, use `DYLD_LIBRARY_PATH` instead of `LD_LIBRARY_PATH`.

Next, place the wolfSSL JNI / wolfJSSE JAR file(s) (wolfssl.jar, wolfssl-jsse.jar) on your Java classpath. You can do this by adjusting your system classpath settings, or at compile time and runtime of a specific application like so:

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

Finally, in your Java application, add the provider at runtime by importing the provider class and calling `Security.addProvider()`:

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

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

To print a list of all installed Security Providers for verification:

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

4.2 Installation at OS / System Level

4.2.1 Unix/Linux

To install the wolfJSSE provider at the system/OS level, copy the `wolfssl.jar` and/or `wolfssl-jsse.jar` into the correct Java installation directory for your OS and verify the `libwolfssljni.so` or `libwolfssljni.jnilib` shared library is on your library search path.

Add the JAR files (`wolfssl.jar`, `wolfssl-jsse.jar`) and shared library (`libwolfssljni.so`) to the following directory:

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

On Ubuntu with OpenJDK this path may be similar to:

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

Next, add an entry to the `java.security` file similar to the following:

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

The java.security file will be located at:

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

Replacing “N” with the order of priority you would like wolfJSSE to have in comparison to other providers in the file. To place the WolfSSLProvider as the top priority provider, the following line would be added to the providers list in the `java.security` file. You will also need to re-number the priority numbers on the other providers listed in the java.security file. The highest priority is 1.

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

4.2.2 Android OSP (AOSP)

For instructions on installing wolfJSSE as a system security provider in the Android OSP (AOSP) source tree, please reference the separate document titled “Installing a JSSE Provider in Android OSP”.

5 Package Design

wolfJSSE is bundled together with the wolfSSL JNI wrapper in the "**wolfssljni**" package. Since wolfJSSE depends on the underlying JNI bindings for wolfSSL, it is compiled into the same native library file as the JNI wrapper.

wolfJSSE / wolfSSL JNI package structure:

```
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
```

The **wolfJSSE** provider source code is located in the `src/java/com/wolfssl/provider/jsse` directory, and is part of the `**com.wolfssl.provider.jsse**` Java package.

The **wolfSSL JNI** wrapper is located in the `src/java/com/wolfssl` directory and is part of the " **com.wolfssl** " Java package. Users of JSSE will not need to use this package directly, as it will be consumed by the wolfJSSE classes.

Once wolfSSL JNI and wolfJSSE have been compiled, there are two JAR files and one native shared library that have been generated. These are located in the `./lib` directory. The native shared library could also be named `libwolfssljni.jnilib` depending on the operating system.

```
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 Supported Algorithms and Classes

wolfJSSE extends or implements the following JSSE classes:

```
javax.net.ssl.SSLContextSpi
    SSL, TLS, DEFAULT, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3
javax.net.ssl.KeyManagerFactorySpi
    PKIX, X509, SunX509
javax.net.ssl.TrustManagerFactorySpi
    PKIX, X509, SunX509
javax.net.ssl.SSLEngine
javax.net.ssl.SSLSession / ExtendedSSLSession
javax.net.ssl.X509KeyManager / X509ExtendedKeyManager
javax.net.ssl.X509TrustManager / X509ExtendedTrustManager
javax.net.ssl.SSLServerSocket
javax.net.ssl.SSLServerSocketFactory
javax.net.ssl.SSLSocket
javax.net.ssl.SSLSocketFactory
javax.net.ssl.SSLSessionContext
java.security.cert.X509Certificate
javax.security.cert.X509Certificate
```


7 Usage

For usage, please follow the Oracle/OpenJDK Javadocs for the classes specified in the previous chapter. Note that you will need to explicitly request the “wolfJSSE” provider if it has been set lower in precedence than other providers that offer the same algorithm in the `java.security` file.

For example, to use the wolfJSSE provider with the SSLContext class for TLS 1.2 an application would create a SSLContext object like so:

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

8 Examples

9 wolfSSL JNI Examples

The `examples` directory contains examples for the wolfSSL thin JNI wrapper. To view examples for the wolfSSL JSSE provider, look in the `./examples/provider` directory.

Examples should be run from the package root directory, and using the provided wrapper scripts. The wrapper scripts set up the correct environment variables for use with the wolfjni jar included in the wolfssljni package.

9.1 Notes on Debug and Logging

wolfJSSE debug logging can be enabled by using `-Dwolfjsse.debug=true` at runtime.

wolfSSL native debug logging can be enabled by using `-Dwolfssl.debug=true` at runtime, if native wolfSSL has been compiled with `--enable-debug`.

JDK debug logging can be enabled using the `-Djavax.net.debug=all` option.

9.2 wolfSSL JNI Example Client and Server

Example client/server applications that use wolfSSL JNI:

Server.java - Example wolfSSL JNI server
Client.java - Example wolfSSL JNI client

These examples can be run with the provided bash scripts:

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

To view usage and available options for the examples, use the `-?` argument:

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

10 wolfJSSE Provider Examples

The `examples/provider` directory contains examples for the wolfSSL JSSE provider (wolfJSSE).

Examples should be run from the package root directory, and using the provided wrapper scripts. The wrapper scripts set up the correct environment variables for use with the wolfJSSE provider included in the wolfssljni package. For example to run the example JSSE server and client, after compiling wolfSSL and wolfssljni:

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

10.1 Notes on Debug and Logging

wolfJSSE debug logging can be enabled by using `-Dwolfjsse.debug=true` at runtime.

wolfSSL native debug logging can be enabled by using `-Dwolfssl.debug=true` at runtime, if native wolfSSL has been compiled with `--enable-debug`.

JDK debug logging can be enabled using the `-Djavax.net.debug=all` option.

10.2 wolfJSSE Example Client and Server

Example client/server applications that use wolfJSSE along with the SSLSocket API.

ServerJSSE.java - Example wolfJSSE server
ClientJSSE.java - Example wolfJSSE client

These examples can be run with the provided bash scripts:

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

10.3 ClientSSLSocket.java

Very minimal JSSE client example using SSLSocket. Does not support all the options that ClientJSSE.java does.

Example usage is:

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

Example usage for connecting to the wolfSSL example server is:

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

The password for client.jks is: “wolfSSL test”

10.4 MultiThreadedSSLClient.java

Multi threaded SSLSocket example that connects a specified number of client threads to a server. Intended to test multi-threading with wolfJSSE.

This example creates a specified number of client threads to a server located at 127.0.0.1:11118. This example is set up to use the SSLSocket class. It makes one connection (handshake), sends/receives data, and shuts down.

A random amount of time is injected into each client thread before:

1) The SSL/TLS handshake
2) Doing I/O operations after the handshake

The maximum amount of sleep time for each of those is “maxSleep”, or 3 seconds by default. This is intended to add some randomness into the client thread operations.

Example usage:

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

This example is designed to connect against the MultiThreadedSSLServer example:

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

This example also prints out average SSL/TLS handshake time, which is measured in milliseconds on the “startHandshake()” API call.

10.5 MultiThreadedSSLServer.java

SSLServerSocket example that creates a new thread per client connection.

This server waits in an infinite loop for client connections, and when connected creates a new thread for each connection. This example is compiled when `ant examples` is run in the package root.

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

For multi threaded client testing, test against MultiThreadedSSLClient.sh. For example, to connect 10 client threads:

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

10.6 ProviderTest.java

This example tests the wolfSSL provider installation. It lists all providers installed on the system, tries to look up the wolfSSL provider, and if found, prints out the information about the wolfSSL provider. Finally, it tests what provider is registered to provide TLS to Java.

This app can be useful for testing if wolfJSSE has been installed correctly at the system level.

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

Note, if wolfJSSE has not been installed at the OS system level, wolfJSSE will not show up as an installed provider when this example is run.

10.7 ThreadedSSLSocketClientServer.java

SSLSocket example that connects a client thread to a server thread.

This example creates two threads, one server and one client. The examples are set up to use the SSLSocket and SSLServerSocket classes. They make one connection (handshake) and shut down.

Example usage:

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