wolfCrypt JCE Provider and JNI Wrapper Manual

2025-06-16

Contents

1 Introduction

The JCE (Java Cryptography Extension) supports the installation of custom Cryptographic Service Providers. Those providers can implement a subset of the underlying cryptographic functionality used by the Java Security API.

This manual describes the details and usage of the wolfCrypt JCE provider. The wolfCrypt JCE provider (wolfJCE) uses JNI to wrap the native wolfCrypt cryptography library for compatibility with the Java Security API. The Github repository for wolfCrypt JNI/JCE is located here.

The wolfcrypt-jni package contains both the wolfCrypt JNI wrapper and wolfJCE JCE provider. The JNI wrapper can be used independently if desired.

2 Requirements

2.1 Java / JDK

wolfJCE requires Java to be installed on the host system. There are several JDK variants available to users and developers - including the Oracle JDK and OpenJDK. wolfJCE has currently been tested with OpenJDK, Oracle JDK, Amazon Coretto, Zulu, Temurin, Microsoft JDK, and Android. Some JDK implementations such as OpenJDK and Android do not require JCE providers to be code signed, whereas the Oracle JDK does. For details on code signing, please see Chapter 7

2.2 JUnit

JUnit is required to be installed on the development system in order to run unit tests. JUnit4 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.jar** " and " **hamcrest-all-1.3.jar** "from junit.org/junit4/. At the time of writing, the mentioned .jar files could be downloaded from the following links:

Junit: junit-4.13.jar

Hamcrest: hamcrest-all-1.3.jar

2) Place these JAR files on your system and set `JUNIT_HOME` to point to the directory location they are at. For example:

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

2.3 make and ant

"make" and "ant" are used to compile native C code and Java code, respectively. Please ensure that these are installed on your development machine.

2.4 wolfSSL / wolfCrypt Library

As a wrapper around the native wolfCrypt library, wolfSSL must be installed and placed on the include and library search paths. wolfJCE can be compiled against either the FIPS 140-2/3 or non-FIPS version of the wolfSSL/wolfCrypt native library.

2.4.1 Compiling wolfSSL/wolfCrypt

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

You can build and install a wolfSSL (wolfssl-x.x.x), wolfSSL FIPS release (wolfssl-x.x.x-commercial-fips), or wolfSSL FIPS Ready release. With any of these archives, you will need to use the `--enable-jni` ./configure option in addition to any other package-specific configure option requirements (ex: `--enable-fips`).

wolfSSL Standard Build:

```
$ cd wolfssl-x.x.x
$ ./configure --enable-jni
$ make check
$ sudo make install
```

wolfSSL FIPSv1 Build:

```
$ cd wolfssl-x.x.x-commercial-fips
$ ./configure --enable-fips --enable-jni
$ make check
$ sudo make install
```

wolfSSL FIPSv2 Build:

```
$ cd wolfssl-x.x.x-commercial-fips
$ ./configure --enable-fips=v2 --enable-jni
$ make check
$ sudo make install
```

wolfSSL FIPSv5 Build:

```
$ cd wolfssl-x.x.x-commercial-fips
$ ./configure --enable-fips=v5 --enable-jni
$ make check
$ sudo make install
```

wolfSSL FIPS Ready Build:

```
$ cd wolfssl-x.x.x-commercial-fips
$ ./configure --enable-fips=ready --enable-jni
$ 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
```

3 Compilation

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

Before running make, copy the correct "makefile" for your system, depending if you are on Linux/Unix or MacOS. For example, if you were on Linux:

```
$ cd wolfcrypt-jni
$ cp makefile.linux makefile
```

If you are instead on Mac OSX:

```
$ cd wolfcrypt-jni
$ cp makefile.macosx makefile
```

Then compile the native C JNI source code with "make". This will generate the native JNI shared library (`libwolfcryptjni.so/dylib`).

```
$ cd wolfcrypt-jni
$ make
```

To compile the Java sources, "ant" is used. There are several ant targets to compile either the JNI or JCE (includes JNI) packages, in either debug or release mode. Running regular "ant" will give usage options:

```
$ ant
...
build:
[echo] wolfCrypt JNI and JCE
[echo]
    ----------------------------------------------------------------------------

[echo] USAGE:
[echo] Run one of the following targets with ant:
[echo] build-jni-debug | builds debug JAR with only wolfCrypt JNI classes
[echo] build-jni-release | builds release JAR with only wolfCrypt JNI classes
[echo] build-jce-debug | builds debug JAR with JNI and JCE classes
[echo] build-jce-release | builds release JAR with JNI and JCE classes
[echo]
    ----------------------------------------------------------------------------
```

Use the build target that matches your needs. For example, if you want to build the wolfJCE provider in release mode, run:

```
$ ant build-jce-release
```

To run the JUnit tests, run `ant test`. This will compile only the tests that match the build that was done (JNI vs. JCE) and run those tests.

```
$ ant test
```

To clean the both Java JAR and native library, run:

```
$ ant clean
$ make clean
```

3.1 API Javadocs

Running `ant` will generate a set of Javadocs under the `wolfcrypt-jni/docs/javadoc` directory. To view the Javadoc index, open the following file in a web browser:

```
wolfcrypt-jni/docs/javadoc/index.html
```

4 Installation

There are two ways that wolfJCE can be installed and used - either at runtime inside a single Java application, or at the system level for all Java applications to use.

4.1 Installation at Runtime

To install and use wolfJCE at runtime inside a single application, first make sure that "**libwolfcryptjni.so**" (or "**libwolfcryptjni.dylib**" if on MacOS) 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 MacOS, you can use `DYLD_LIBRARY_PATH` instead:

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

Next, place the wolfCrypt JNI/JCE JAR file (**wolfcrypt-jni.jar**) on your Java classpath. You can do this by adjusting your system classpath settings or at compile time and runtime 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.insertProviderAt()` to insert the wolfCryptProvider in the Java Provider list as the top most priority provider. Note that provider location 1 is the highest priority location.

```
import com.wolfssl.provider.jce.WolfCryptProvider;

public class TestClass {
   public static void main(String args[]) {
      ...
      Security.insertProviderAt(new WolfCryptProvider(), 1);
      ...
   }
}
```

To print a list of all installed providers from a Java application for verification, you can do:

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

4.2 Installation at OS / System Level (Java <= 8)

wolfJCE can be installed at the system level so that any Java application consuming Java Security APIs for cryptography can leverage wolfJCE.

To install the wolfJCE provider at the system level, copy the JAR into the correct Java installation directory for your OS and JDK and verify the shared library is on your library search path.

Add the wolfCrypt JNI/JCE JAR file (**wolfcrypt-jni.jar**) and shared library (**libwolfcryptjni.so** or **libwolfcryptjni.dylib**) to the following directory:

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

For example, on Ubuntu with OpenJDK this may be similar to:

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

Next add an entry to the `java.security` file that looks similar to the following, adding a provider entry for WolfCryptProvider:

```
security.provider.N=com.wolfssl.provider.jce.WolfCryptProvider
```

The java.security file will be located at:

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

Replacing "N" with the order of precedence you would like the WolfCryptProvider to have in comparison to other providers in the file.

5 Package Design

wolfJCE is bundled together with the “wolfcrypt-jni” JNI wrapper library. Since wolfJCE depends on the underlying JNI bindings for wolfCrypt, it is compiled into the same native library file and JAR file as wolfcrypt-jni.

For users wanting to use only the JNI wrapper, it is possible to compile a version of “wolfcrypt-jni.jar” that does not include the JCE provider classes.

wolfJCE / wolfCrypt JNI package structure:

```
wolfcrypt-jni /
    .github                         GitHub Actions workflows
    AUTHORS
    COPYING
    ChangeLog.md                    Version ChangeLog
    IDE/                            IDE Project Files
        WIN/                        Visual Studio Project Files
    LICENSING
    README.md                       Main README
    README_JCE.md                   wolfJCE README
    build.xml                       ant build script
    docs /                          Javadocs
    examples/                       Example applications and certs/keys
        certs/                      Example cert/keys/KeyStores
        provider/                   JCE example apps
    jni/                            Native C JNI binding source files
    lib/                            Compiled library artifacts
    makefile.linux                  Linux-specific Makefile
    makefile.osx                    OSX-specific Makefile
    pom.xml                         Maven build file
    rpm/                            Linux rpm files
    scripts/                        Test scripts (Facebook Infer, etc)
    src/                            Source code
        main/java                   Java source files
```

The wolfJCE provider source code is located in the `src/main/java/com/wolfssl/provider/jce` directory, and is part of the “**com.wolfssl.provider.jce**” Java package.

The wolfCrypt JNI wrapper is located in the `src/main/java/com/wolfssl/wolfcrypt` directory and is part of the “**com.wolfssl.wolfcrypt**” Java package. Users of JCE will not need to use this package directly, as it will be consumed by the wolfJCE classes.

Once wolfCrypt JNI and wolfJCE have been compiled, the output JAR and native shared library are located in the `./lib` directory. Note, these contain BOTH the wolfCrypt JNI wrapper as well as the wolfJCE provider when a JCE build is compiled.

```
lib/
    libwolfcryptjni.so (libwolfcryptjni.dylib)
    wolfcrypt-jni.jar
```

6 Supported JCE Algorithms and Classes

wolfJCE currently supports the following algorithms and classes:

```
MessageDigest Class
    MD5
    SHA-1
    SHA-256
    SHA-384
    SHA-512

SecureRandom Class
    DEFAULT (maps to HashDRBG)
    HashDRBG

Cipher Class
    AES/CBC/NoPadding
    AES/CBC/PKCS5Padding
    AES/GCM/NoPadding
    DESede/CBC/NoPadding
    RSA
    RSA/ECB/PKCS1Padding

Mac Class
    HmacMD5
    HmacSHA1
    HmacSHA256
    HmacSHA384
    HmacSHA512

Signature Class
    MD5withRSA
    SHA1withRSA
    SHA256withRSA
    SHA384withRSA
    SHA512withRSA
    SHA1withECDSA
    SHA256withECDSA
    SHA384withECDSA
    SHA512withECDSA

KeyAgreement Class
    DiffieHellman
    DH
    ECDH

KeyPairGenerator Class
    RSA
    EC
    DH

CertPathValidator Class
    PKIX
```

```
SecretKeyFactory
    PBKDF2WithHmacSHA1
    PBKDF2WithHmacSHA224
    PBKDF2WithHmacSHA256
    PBKDF2WithHmacSHA384
    PBKDF2WithHmacSHA512
    PBKDF2WithHmacSHA3-224
    PBKDF2WithHmacSHA3-256
    PBKDF2WithHmacSHA3-384
    PBKDF2WithHmacSHA3-512

KeyStore
    WKS
```

7 JAR Code Signing

The Oracle JDK/JVM require that JCE providers be signed by a code signing certificate that has been issued by Oracle. The wolfcrypt-jni package ant build script supports code signing the generated `wolfcrypt-jni.jar` file by placing a custom properties file in the root of the package directory before compilation.

To enable automatic code signing, create a file called `codeSigning.properties` and place it in the root of the `wolfcrypt-jni` package. This is a properties file which should include the following:

```
sign.alias=<signing alias in keystore>
sign.keystore=<path to signing keystore>
sign.storepass=<keystore password>
sign.tsaurl=<timestamp server url>
```

When this file is present when `ant` or `ant test` is run, it will sign `wolfcrypt-jni.jar` using the keystore and alias provided.

7.1 Using a Pre-Signed JAR File

wolfSSL Inc. (company) has a code signing certificate from Oracle that allows wolfJCE to be authenticated in the Oracle JDK. With each release of wolfJCE, wolfSSL ships pre-signed versions of the `wolfcrypt-jni.jar`, located at:

```
wolfcrypt-jni-X.X.X/lib/signed/debug/wolfcrypt-jni.jar
wolfcrypt-jni-X.X.X/lib/signed/release/wolfcrypt-jni.jar
```

These pre-signed JARs can be used with the JUnit tests, without having to re-compile the Java source files. To run the JUnit tests against this JAR file:

```
$ cd wolfcrypt-jni-X.X.X
$ cp ./lib/signed/release/wolfcrypt-jni.jar ./lib
$ ant test
```

8 Usage

For usage, please follow the Oracle/OpenJDK Javadocs for the classes specified in Chapter 6. Note that you will need to explicitly request the `wolfJCE` 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 wolfJCE provider with the MessageDigest class for SHA-1 you would create a MessageDigest object like so:

```
MessageDigest md = MessageDigest.getInstance“(SHA”-1, “”wolfJCE);
```

Please email support@wolfssl.com with any questions or feedback.

9 KeyStore Implementations

wolfJCE includes one Java KeyStore implementation, WolfSSLKeyStore (WKS). It has been designed to be compatible with wolfCrypt FIPS 140-2 / 140-3 validated modules, using cryptography algorithms and key sizes within the FIPS validated boundary. The WKS KeyStore type can also be used with non-FIPS versions of wolfSSL and wolfCrypt. WolfSSLKeyStore (WKS) KeyStores can be used along with wolfSSL JNI/JSSE as well.

9.1 JKS to WKS Migration Guide

Users of wolfJCE may wish to migrate from existing Java KeyStore files over to WolfSSLKeyStore (WKS) format to ensure use of FIPS 140-2/3 validated algorithms if using wolfCrypt FIPS. This migration guide will outline some of the steps and considerations to take into account when migrating KeyStore formats.

9.1.1 FIPS 140-2 / 140-3 Algorithm Requirement Considerations

FIPS 140-2 / 140-3 validation compliance from an application perspective typically means that all cryptography being called or used should come from a FIPS validated cryptographic module. Oftentimes the strictness of what cryptography needs to be FIPS validated is decided by the end consumer of the application or product. In some cases, not 100% of the cryptography in a system needs to be FIPS validated. For example an end consumer may only require cryptography used to secure data in transit be FIPS validated, and other cryptography (such as the algorithms used for key storage on the device) do not have the same FIPS requirements.

There are some use cases where the Java KeyStore objects and files being used are out of scope of the FIPS validation requirement. In those cases, it may be simpler and require fewer changes to a system to use the existing KeyStore files on a system. Typically these will be JKS or PKCS#12 format stores, as those are typically generated and consumed by Java applications using Oracle's SunJSSE and SunJCE cryptographic provider implementations.

Other applications and use cases will require all cryptography on the system to be FIPS validated. For these use cases, wolfSSL has created the WolfSSLKeyStore (WKS) store type. This migration guide will walk through some common areas and considerations when switching from other KeyStore types (ex: JKS, PKCS#12) over to the WKS type.

9.1.2 WolfSSLKeyStore Format (WKS)

The WKS KeyStore format is unique and different than other Java KeyStore types. It has been designed to use FIPS 140-2 / 140-3 validated algorithms from the wolfCrypt FIPS module to maintain FIPS validation conformance.

The WKS implementation uses AES-CBC-256 along with HMAC-SHA512 in an Encrypt-then-MAC format for encryption of PrivateKey and SecretKey objects. It uses HMAC-SHA512 for KeyStore integrity. More details on the design of the WKS type can be found in the WolfSSLKeyStore design document (WolfSSLKeyStore.md).

9.1.3 Converting Existing KeyStore Files to WKS

Existing JKS (.jks), PKCS#12 (.p12), and other Java KeyStore files will need to be converted to WKS type. This can easily be done using the Java `keytool` application.

For `keytool` to convert KeyStore files to WKS, it will need access to compiled wolfCrypt JNI/JSSE library files (.so/.dylib and .jar). After compiling wolfCrypt JNI/JCE, make sure the native JNI shared library is on the native library search path. For Linux, add the location of `libwolfcryptjni.so` to `LD_LIBRARY_PATH`, for example:

```
$ export LD_LIBRARY_PATH=/path/to/wolfcryptjni/lib:$LD_LIBRARY_PATH
```

If on MacOS, add the location of `libwolfcryptjni.dylib` to `DYLD_LIBRARY_PATH`, for example:

```
$ export DYLD_LIBRARY_PATH=/path/to/wolfcryptjni/lib:$DYLD_LIBRARY_PATH
```

`keytool` can then be used to convert between KeyStore types. Usage will be similar to:

```
$ keytool -importkeystore -srckeystore keystore.jks -destkeystore keystore.wks
     -srcstoretype JKS -deststoretype WKS -srcstorepass “password -
    deststorepass “”password -provider com.wolfssl.provider.jce.
    WolfCryptProvider --providerpath /path/to/wolfcryptjni/lib/wolfcrypt-jni.
    jar
```

The `keytool` options that need to be used are:

keytool option	Description
-importkeystore	Import from source keystore to destination
-srckeystore	Source keystore to be read from
-destkeystore	Destination keystore to write to
-srcstoretype	Type of source keystore (ex: JKS)
-deststoretype	Type of destination keystore (WKS)
-srcstorepass	Password of source keystore
-deststorepass	Password for destination keystore
-provider	Full class name of KeyStore provider to use for conversion
–providerpath	Full path to JCE provider JAR file which contains provider

After converting KeyStore files, you should have new equivalent KeyStore files but in the .wks (WolfSS-LKeyStore) format.

9.1.4 Viewing Contents of WolfSSLKeyStore (WKS) File

The Java `keytool` command can be used to view the contents of a WKS KeyStore file. Usage will be similar to:

```
keytool -list -provider com.wolfssl.provider.jce.WolfCryptProvider --
    providerpath /path/to/wolfcryptjni/lib/wolfcrypt-jni.jar -storetype WKS -
    storepass “”password -keystore keystore.wks
```

9.1.5 Changing Application Usage of KeyStore Type

Java application code typically either creates new KeyStore objects to store keys and certificates into, or opens existing KeyStore files for use.

9.1.5.1 Creating New KeyStore Objects

Java application code will typically create new KeyStore objects with code similar to the following, explicitly getting the a KeyStore instance of type “JKS”, or other KeyStore type:

```
import java.security.KeyStore;
...
String storePass = “”mypassword;
```

```
KeyStore store = KeyStore.getInstance“(”JKS);
store.load(null, storePass.toCharArray());
```

To convert this code to creating a KeyStore of type “WKS”, only the type passed to `getInstance()` will need to be updated:

```
KeyStore store = KeyStore.getInstance“(”WKS);
```

All application code that uses the KeyStore object should work as-is since the WolfSSLKeyStore implementation extends `KeyStoreSpi` and implements the methods in that abstract class.

9.1.5.2 Opening Existing KeyStore Files for Use Java application code that opens existing KeyStore files for use will typically do so with similar code to below:

```
import java.security.KeyStore;
...
String storePass = “”mypassword;
KeyStore store = KeyStore.getInstance“(”JKS);
store.load(new FileInputStream(keystoreFilePath),
    storePass.toCharArray());
```

To convert this code to use `WKS` type, change the call to `getInstance()` to request a KeyStore instance of `WKS`:

```
KeyStore store = KeyStore.getInstance“(”WKS);
```

Then, the actual KeyStore file on disk being read will need to already be in WolfSSLKeyStore (WKS) format. See te section above about converting KeyStore files to WKS type for instructions on how to do this.

9.1.6 Converting System CA Certificate KeyStore Files

Java JDK implementations commonly ship with their own KeyStore containing known trusted CA certificates. This will typically be either in a file called `cacerts` or `jssecacerts`. The location of this file will vary depending on Java version, but can typically be found at the following locations.

cacerts:

```
$JAVA_HOME/lib/security/cacerts         (JDK 9+)
$JAVA_HOME/jre/lib/security/cacerts     (JDK <= 8)
```

jssecacerts:

```
$JAVA_HOME/lib/security/jssecacerts     (JDK 9+)
$JAVA_HOME/jre/lib/security/jssecacerts (JDK <= 8)
```

The default `cacerts.jks` password is “**changeit**”. If using wolfCrypt FIPS 140-2 / 140-3 the minimum HMAC key size is 14 bytes. Since HMAC is used for the KeyStore integrity checks and MAC on PrivateKey/SecretKey objects, KeyStore passwords must be at least 14 characters. Because of this restriction, when converting system CA KeyStore files to WKS type, the password should be updated, to something like “**changeitchangeit**” for example.

wolfCrypt JNI/JCE includes a helper bash script which has been set up to try and detect system CA KeyStore files and convert them to WKS type. This script is located at:

```
wolfcryptjni/examples/certs/systemcerts/system-cacerts-to-wks.sh
```

This script should be run from the directory where it resides. Successful execution will result in local copies of the `cacerts` and/or `jssecacerts` KeyStore files created and placed in the same directory as the script, but in WKS format.

```
./system-cacerts-to-wks.sh
-----------------------------------------------------------------------
System CA KeyStore to WKS Conversion Script
-----------------------------------------------------------------------
JAVA_HOME already set = /path/to/java/installation
Detected Darwin/OSX host OS
Java Home = /path/to/java/installation

System cacerts found, converting from JKS to WKS:
    FROM: /path/to/java/installation/jre/lib/security/cacerts
    TO:   /path/to/wolfcryptjni/examples/certs/systemcerts/cacerts.wks
    IN PASS (default): changeit
    OUT PASS: changeitchangeit

Successfully converted JKS to WKS
```

You can then copy this `cacerts.wks` file over to your system JDK location if desired.

9.1.7 WKS KeyStore Use with wolfJSSE

wolfSSL JNI/JSSE, as of PR 178 has been modified to give preference to loading and using WolfSSLKeyStore (WKS) KeyStore types.

When auto-loading the system CA/root certificates (ex: jssecacerts, cacerts), wolfJSSE first tries to find and load a WKS equivalent file at the same location (ex: jssecacerts.wks, cacerts.wks).

Support for a new Java Security property has been added (`wolfjsse.keystore.type.required`) which can be used to restrict use of KeyStore type to the one set in this property. This can be used for example to help conform to wolfCrypt FIPS 140-2/3 crypto usage by setting to "**WKS**" when wolfJCE is also used and installed on the system.

The wolfJSSE provider example `ClientJSSE.java` and `ServerJSSE.java` have been updated with a new option to specify the KeyStore type ('`-ksformat`).

9.1.8 Support

For support or assistance in converting JKS or other KeyStore file types over to WolfSSLKeyStore (WKS) types, please email support@wolfssl.com.