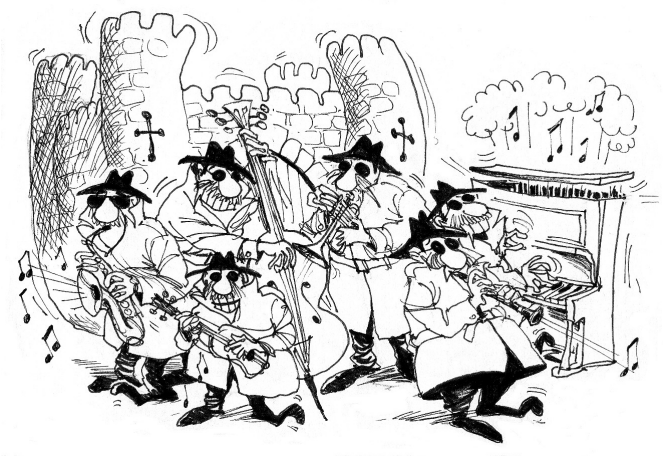
Legion of the Bouncy Castle Inc.
Java (D)TLS API and JSSE Provider

User Guide

Version: 2.0.23 Date: 05/03/26



Legion of the Bouncy Castle Inc.
(ABN 84 166 338 567)
https://www.bouncycastle.org

Copyright and Trademark Notice

Acknowledgements

Initial work on the DTLS/TLS APIs and the JSSE provider was funded through grants from the Core Infrastructure Initiative (CII) which is managed by the Linux Foundation. See www.coreinfrastructure.org for further information.

Stage two work improved the overall flexibility of the JSSE provider and was funded by Micro Focus (www.microfocus.com). Improvements included expansion of supported Cipher Suites, the addition of an initial FIPS mode and improved exception handling and logging.

Further work on the project has also been funded through support contract consulting hours made available from holders of Bouncy Castle support contracts with Keyfactor. Details about support contracts can be found at: https://www.keyfactor.com/open-source/bouncy-castle-support/.

For further information about this distribution, or to help support this work further, please contact us at office@bouncycastle.org.

Table of Contents

1 Introduction

This document is a guide to the use of the Legion of the Bouncy Castle DTLS/TLS APIs and JSSE provider.

The BC DTLS/TLS APIs and JSSE provider follow a similar model to that of the BC JCA/JCE provider. The DTLS/TLS APIs provide a low-level mechanism for making use of the DTLS and TLS protocols which is more flexible than what is provided by the JSSE APIs, but requires more knowledge and care on the part of the developer. The BC JSSE provider is built on top of the TLS API and provides functionality to support the use of the JSSE API defined in javax.net.ssl.

2 Installation

The bctls jar can be installed in either jre/lib/ext for your JRE/JDK, or in many cases, on the general class path defined for the application you are running. In the event you do install on the bctls jar on the general class path be aware that sometimes the system class loader may make it impossible to use the classes in the bctls jar without causing an exception depending on how elements of the JSSE provider are loaded elsewhere in the application.

2.1 BCJSSE Provider installation into the JRE

Once the bctls jar is installed, the provider class BouncyCastleJsseProvider may need to be installed if it is required in the application globally.

Installation of the provider can be done statically in the JVM by adding it to the provider definition to the java.security file in in the jre/lib/security directory for your JRE/JDK.

The provider can also be added during execution. If you wish to add the provider to the JVM globally during execution you can add the following imports to your code:

```
import java.security.Security
import org.bouncycastle.jsse.provider.BouncyCastleJsseProvider
```

Then insert the line

```
Security.addProvider(new BouncyCastleJsseProvider());
```

The provider can then be used by referencing the name “BCJSSE”, for example:

```
SSLContext clientContext = SSLContext.getInstance("TLS", “BCJSSE”);
```

Alternately if you do not wish to install the provider globally, but use it locally instead, it is possible to pass the provider to the getInstance() method on the JSSE class you are creating an instance of. For example:

```
SSLContext clientContext = SSLContext.getInstance("TLS",
                                    new BouncyCastleJsseProvider());
```

2.1.1 Configuring the BCJSSE Provider in FIPS mode

The BCJSSE provider has a FIPS mode which helps restricts the provider to cipher suites and parameters that can be offered in a FIPS compliant TLS client, or server, setup. FIPS mode is indicated by passing the string “fips:<provider>” to the constructor of the BouncyCastleJsseProvider class, either at runtime or via the java.security file for the JVM.

Acceptable values of <provider> are “default” which indicates that any other crypto provider can be installed or a provider name, such as “BC”, or “BCFIPS” if you are using the Bouncy Castle FIPS provider.

The security.provider section of the java.security file in the JCE would look like the following for a minimal install of the BCFIPS provider and the BCJSSE provider in FIPS mode.

```
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
security.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider fips:BCFIPS
security.provider.3=sun.security.provider.Sun
```

At runtime a similar configuration can be achieved using:

```
Security.addProvider(new BouncyCastleJsseProvider("fips:BCFIPS"));
```

2.1.2 Enabling GCM for TLS 1.2 in the BCJSSE Provider in FIPS mode

For TLS 1.2, the GCM nonce used in FIPS mode is required to be generated in the cryptographic boundary of the FIPS module providing the cryptographic services being used by the BCJSSE.

In the case of the bctls-fips distribution of the BCJSSE, the BCJSSE jar is configured already to recognise the BCFIPS provider, no work on a user's part is required.

Where the BCFIPS module is not in use and some other Java FIPS provider is being used an override for the JcaTlsCrypto class needs to be provided which is used to fetch the in-boundary nonce generator by using an implementation of the AEADNonceGeneratorFactory. Briefly the following is required:

```
public class FipsJcaTlsCrypto extends JcaTlsCrypto
{
   public FipsJcaTlsCrypto(
        JcaJceHelper helper, SecureRandom entropySource, SecureRandom nonceEntropySource)
   {
      super(helper, entropySource, nonceEntropySource);
   }

   @Override
   public AEADNonceGeneratorFactory getFipsGCMNonceGeneratorFactory()
   {
      return <in-boundary derived implementation of AEADNonceGeneratorFactory goes here>
   }
}
```

A corresponding provider class is also required, which can usually just be defined as:

```
public class FipsJcaTlsCryptoProvider extends JcaTlsCryptoProvider
{
   @Override
   public JcaTlsCrypto create(SecureRandom keyRandom, SecureRandom
nonceRandom)
   {
      return new FipsJcaTlsCrypto(getHelper(), keyRandom, nonceRandom);
   }
}
```

The provider class can then be used to construct BCJSSE provider class which will have the TLS 1.2 GCM cipher suites enabled in it. For example:

```
boolean fips = true;
JcaTlsCryptoProvider cryptoProvider = new FipsJcaTlsCryptoProvider();

Provider jsseProvider = new BouncyCastleJsseProvider(fips, cryptoProvider);
```

Note: the BCJSSE will still work if the above is not done, but you will find that, where the BCFIPS provider is not in use the TLS 1.2 GCM cipher suites will not be enabled.

2.2 Other Differences between the BCJSSE and JSSE provider

The TrustManagers produced by the BCJSSE provider will ignore private keys in any KeyStores passed to their factories. Only simple certificate entries in a KeyStore will be used to create a TrustManager.

From BC-FJA 1.0.2 the BCFIPS provider includes a NonceGenerator for GCM which guarantees failure of a nonce generation where the counter might wrap around. Earlier versions of BC-FJA are unable to support GCM in the BCJSSE as the generator is not available.

BCJSSE also supports ChaCha20 and Poly1305 in non-FIPS mode. Support for this cipher suite is available in the BCFIPS provider from 1.0.2.

3 Using the BCJSSE Provider

While usage of the JSSE provider is essentially the same as usage of the JSSE provider that ships with the JRE, there are small differences relating to some of the names used to create the various objects, and to the way in which some things behave. BCJSSE TrustManagerFactory objects will ignore private key entries in passed in KeyStore objects, and names like “SunX509” do not exist.

Note also that BCJSSE SSLContext instances can only be used with BCJSSE KeyManagerFactory and TrustManagerFactory instances.

3.1 Conventions

3.1.1 Use of getDefault()

Things like SSLSocketFactory.getDefault() will return the class from the BCJSSE provider only if the BCJSSE provider is either ahead of or replacing the regular JRE JSSE provider. In order that BCJSSE implementations of KeyManagerFactory and/or TrustManagerFactory are instantiated in this scenario, the following java.security file properties/values should be set:

```
ssl.KeyManagerFactory.algorithm=PKIX
ssl.TrustManagerFactory.algorithm=PKIX
```

3.1.2 Instance Names

The core KeyManagerFactory has the instance name “X.509”, and the aliases “X509” and “PKIX”.

The core TrustManagerFactory has the instance name “PKIX”, and the aliases “X.509” and “X509”.

SSLContext instances are available for the algorithms “TLSv1”, “TLSv1.1”, “TLSv1.2”, and “TLSv1.3”. This choice affects which protocols will be enabled for client connections. In each case the algorithm name represents the highest version of TLS enabled and lower versions are also enabled. The SSLContext algorithms “TLS” and "DEFAULT" will enable TLSv1.2 and earlier, i.e. they currently behave like "TLSv1.2". Servers also always default to enabling TLSv1.2 and earlier. (In the next release, the default will include TLSv1.3).

“SSL” is an alias for “TLS” and “SSLv3” is an alias for “TLSv1”. Although "SSLv3" is still a supported protocol version, it is never enabled by default.

The system properties "jdk.tls.client.protocols" and "jdk.tls.server.protocols" can be used to explicitly set the enabled protocols in the case of client connections and server connections respectively. These override the settings based on the SSLContext algorithm.

Note that the security property jdk.tls.disabledAlgorithms might disable the use of certain protocols. In recent JDKs this typically affects “SSLv3”, “TLSv1” and “TLSv1.1”. Protocols disabled in this way are not usable in any configuration.

3.2 A basic Server

A basic server only requires a KeyManager to identify itself with. The utility class described in Appendix E – Utility Classes used in Examples is used to generate the necessary credentials and the simple protocol it executes is also described there. The actual KeyManager is then created by a

KeyManagerFactory which has been initialised using the server's key store and key store password.

Note: in this case the BCFIPS and BCJSSE providers have just been added to the provider list – no attempt has been made to replace the regular JSSE provider, so SSLSocketFactory.getDefault() will still refer to the regular JSSE provider.

```
import java.security.Security;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;

import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider;
import org.bouncycastle.jsse.provider.BouncyCastleJsseProvider;

/**
 * Basic SSL Server - using the '!' protocol.
 */
public class TLSServerExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleFipsProvider());
        Security.addProvider(new BouncyCastleJsseProvider());

        SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");
        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance(
                                                        "PKIX", "BCJSSE");

        keyMgrFact.init(Utils.createServerKeyStore(), Utils.SERVER_PASSWORD);

        sslContext.init(keyMgrFact.getKeyManagers(), null, null);

        SSLServerSocketFactory fact = sslContext.getServerSocketFactory();
        SSLServerSocket sSock = (SSLServerSocket)fact.createServerSocket(
                                                            Utils.PORT_NO);
        SSLSocket sslSock = (SSLSocket)sSock.accept();
        Protocol.doServerSide(sslSock);
    }
}
```

3.3 A basic Client

In the case of a basic client the only requirement is that is has some way of identifying the server. This is done by providing a TrustManager to validate incoming credentials. The TrustManager is created using a TrustManagerFactory which has been initialised with a key store containing the certificates needed to validate the certificate path the server will present.

```
import java.security.Security;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;

import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider;
```

```
import org.bouncycastle.jsse.provider.BouncyCastleJsseProvider;

/**
 * Basic SSL Client - using the '!' protocol.
 */
public class TLSClientExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleFipsProvider());
        Security.addProvider(new BouncyCastleJsseProvider());

        SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");
        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance(
                                                              "PKIX", "BCJSSE");
        trustMgrFact.init(Utils.createServerTrustStore());

        sslContext.init(null, trustMgrFact.getTrustManagers(), null);

        SSLSocketFactory fact = sslContext.getSocketFactory();
        SSLSocket    cSock = (SSLSocket)fact.createSocket(
                                        Utils.HOST, Utils.PORT_NO);

        Protocol.doClientSide(cSock);
    }
}
```

3.4 Using Client Authentication

Basic client authentication requires a server which has the ability to validated a client using a TrustManager and if client authentication is compulsory the SSLServerSocket.setNeedClientAuth() method needs to be called with “true”.

The basic changes required have been made to the original server example and reproduced below.

```
import java.security.Security;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.TrustManagerFactory;

import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider;
import org.bouncycastle.jsse.provider.BouncyCastleJsseProvider;

/**
 * Basic SSL Server - using the '!' protocol.
 */
public class TLSServerWithClientAuthExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleFipsProvider());
        Security.addProvider(new BouncyCastleJsseProvider());
```

```
        SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");
        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance(
                                                        "PKIX", "BCJSSE");
        keyMgrFact.init(Utils.createServerKeyStore(), Utils.SERVER_PASSWORD);

        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance(
                                                        "PKIX", "BCJSSE");
        trustMgrFact.init(Utils.createClientTrustStore());

        sslContext.init(
            keyMgrFact.getKeyManagers(), trustMgrFact.getTrustManagers(), null);

        SSLServerSocketFactory fact = sslContext.getServerSocketFactory();
        SSLServerSocket        sSock = (SSLServerSocket)fact.createServerSocket(
                                                            Utils.PORT_NO);
        sSock.setNeedClientAuth(true);

        SSLSocket sslSock = (SSLSocket)sSock.accept();

        Protocol.doServerSide(sslSock);
    }
}
```

Just as a TrustManager needs to introduced to the server to allow it to identify the client a KeyManager also needs to be added to the client's SSLContext to allow the client to identify itself to the server.

The code example that follows shows a basic client that is capable of authenticating itself to a server. Note that the trust anchor for the certificate path identifying the client must also have been made available to the TrustManager created for the server.

As you can see in the example, from the client's point of view, simply providing the KeyManager is enough. No additional settings for this are required.

```
import java.security.Security;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider;
import org.bouncycastle.jsse.provider.BouncyCastleJsseProvider;

/**
 * Basic SSL Client - using the '!' protocol.
 */
public class TLSClientWithClientAuthExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        Security.addProvider(new BouncyCastleFipsProvider());
        Security.addProvider(new BouncyCastleJsseProvider());

        SSLContext sslContext = SSLContext.getInstance("TLS", "BCJSSE");
        KeyManagerFactory keyMgrFact = KeyManagerFactory.getInstance(
```

```
                                                        "PKIX", "BCJSSE");
        keyMgrFact.init(Utils.createClientKeyStore(), Utils.CLIENT_PASSWORD);

        TrustManagerFactory trustMgrFact = TrustManagerFactory.getInstance(
                                                        "PKIX", "BCJSSE");
        trustMgrFact.init(Utils.createServerTrustStore());

        sslContext.init(
            keyMgrFact.getKeyManagers(), trustMgrFact.getTrustManagers(), null);

        SSLSocketFactory fact = sslContext.getSocketFactory();
        SSLSocket        cSock = (SSLSocket)fact.createSocket(
                                        Utils.HOST, Utils.PORT_NO);

        Protocol.doClientSide(cSock);
    }
}
```

3.5 SunJSSE Compatibility Issues

3.5.1 Endpoint Identification

Endpoint identification was introduced in 1.0.7 in a manner that tried to be as compatible with SunJSSE as possible. Unfortunately, SunJSSE SSL sockets have special access to an internal API (specifically access to the 'originalHostName' of an InetAddress), which third party providers like BCJSSE do not have. HttpsURLConnection (for example) will also call a (private API) 'setHost' method on SunJSSE SSL sockets (only). A workaround is therefore required to let BCJSSE's endpoint identification work correctly.

The recommended approach is to supply a custom SSLSocketFactory to set the correct hostname on a BCJSSE SSL socket directly. We provide the org.bouncycastle.jsse.util.SetHostSocketFactory[1] for this purpose, which takes a URL and uses it to call the BC extension method 'setHost' on any BCSSLSocket instances it creates.

We also provide org.bouncycastle.jsse.util.URLConnectionUtil as a convenient replacement for typical usages of java.net.URL, which will handle setting up of the SetHostSocketFactory, as shown in the example below:

```
// main code block
{
    URL serverURL = "https://...";
    URLConnectionUtil util = new URLConnectionUtil();
    HttpsURLConnection conn =
    (HttpsURLConnection)util.openConnection(serverURL);
}
```

1 Note that we earlier recommended org.bouncycastle.jsse.util.SNIHostSocketFactory for this purpose, which will still work for endpoint identification, but has the downside that the socket still lacks the hostname for other purposes like logging and session caching.

The SetHostSocketFactory configured by URLConnectionUtil will delegate socket creation to whatever is already configured for HttpsURLConnection, or you can configure a different delegate using the appropriate URLConnectionUtil constructor.

There are other options available based on system properties, which we don't recommend due to the possible security implications and the system-wide effect of setting properties:

1. Set the system property "jdk.tls.trustNameService" to "true". However, note this advice from the Java 8u51 release notes:
   If an application does need to perform reverse name lookup for raw IP addresses in SSL/TLS connections, and encounter endpoint identification compatibility issue, System property "jdk.tls.trustNameService" can be used to switch on reverse name lookup. Note that if the name service is not trustworthy, enabling reverse name lookup may be susceptible to MITM attacks.

2. Set the system property "org.bouncycastle.jsse.client.assumeOriginalHostName" to "true". With this enabled, BCJSSE SSL sockets will assume that calling InetAddress.getHostName will return the same value as the 'originalHostName'. This has the benefit of only affecting BCJSSE, but this assumption will only be correct if you are always creating sockets with a hostname (as opposed to an IP address), and even then is subject to implementation changes outside of our control. If the assumption proves false, then it will in effect be relying on reverse name lookup as for "jdk.tls.trustNameService" and the same caveats apply as mentioned above.

3.5.2 Renegotiation

BCJSSE does not support renegotiation, and we do not intend to add support for it. We consider it an underspecified mechanism, and inherently insecure. Also, the JSSE API makes distinguishing between pre- and post-renegotiation application data cumbersome and error-prone. It would be very easy for user code to ignore the distinction (or to have subtle bugs) leading to serious security problems.

We nevertheless support the measures in RFC 5746 (Renegotiation Indication Extension) and RFC 7627 (Extended Master Secret) that address known weaknesses with renegotiation. This avoids initial handshake failures with peers that require support for those (as up-to-date implementations should).

Note that TLS 1.3 also forbids renegotiation, but includes post-handshake client authentication (RFC 8446 4.6.2) and Key/IV update (RFC 8446 4.6.3) instead. BCJSSE intends to implement both of these as part of the (future) TLS 1.3 support.

3.6 BC Extensions

BCJSSE returns SSLSocket objects which implement the org.bouncycastle.jsse BCSSLSocket interface. Similarly, SSLEngine objects implement org.bouncycastle.jsse.BCSSLEngine. These BCJSSE-specific interfaces provide extended functionality not available via the standard JSSE API, (or not available in earlier JDKs).

The BCSSLSocket is defined as follows:

```
/**
 * A BCJSSE-specific interface to expose extended functionality
 * on {@link javax.net.ssl.SSLSocket} implementations.
 */
public interface BCSSLSocket
{
    String getApplicationProtocol();

    BCApplicationProtocolSelector<SSLSocket>
getBCHandshakeApplicationProtocolSelector();

    void setBCHandshakeApplicationProtocolSelector(
BCApplicationProtocolSelector<SSLSocket> selector);

    void setBCSessionToResume(BCExtendedSSLSession session);

    BCExtendedSSLSession getBCHandshakeSession();

    BCExtendedSSLSession getBCSession();

   /**
    * Returns an accessor for extended SSL connection data. This
    * method will initiate the initial handshake if necessary and
    * then block until the handshake has been established. If an
    * error occurs during the initial handshake, this method returns
    * <code>null</code>.
    *
    * @return A {@link BCSSLConnection} instance.
    */
    BCSSLConnection getConnection();

    String getHandshakeApplicationProtocol();

    /**
     * Returns a {@link BCSSLParameters} with properties reflecting
     * the current configuration.
     * @return the current {@link BCSSLParameters parameters}
     */
    BCSSLParameters getParameters();

    /**
     * Allows explicit setting of the 'host' {@link String} when the
     * {@link SocketFactory} methods that include it as an argument
     * are not used.
     * <p>
     * Must be called prior to attempting to connect the socket
     * to have any effect.
     * </p>
     *
     * @param host the server host name with which to connect, or
     * <code>null</code> for the loopback address.
     */
    void setHost(String host);

    /**
```

```
     * Sets parameters according to the properties in a
     * {@link BCSSLParameters}.
     * <p>
     * Note that any properties set to null will be ignored,
     * which will leave the corresponding settings unchanged.
     * </p>
     *
     * @param parameters the {@link BCSSLParameters parameters} to set
     */
    void setParameters(BCSSLParameters parameters);
}
```

with org.bouncycastle.jsse.BCSSLConnection defined as:

```
/**
 * A BCJSSE-specific interface providing access to extended connection-specific
 * functionality.
 */
public interface BCSSLConnection
{
    /**
     * Returns the application protocol negotiated for this connection, or
     * an empty {@code String} if none was negotiated.
     * See <a href="https://tools.ietf.org/html/rfc7301">RFC 7301</a> for
     * details.
     *
     * @return The negotiated application protocol, or an empty {@code String}.
     */
    String getApplicationProtocol();
    /**
     * Request TLS Channel Bindings for this connection. See
     * <a href="https://tools.ietf.org/html/rfc5929">RFC 5929</a> for details.
     *
     * @param channelBinding An IANA-registered "Channel-binding unique prefix"
     *          valid for TLS e.g. "tls-unique" or "tls-server-end-point".
     * @return A copy of the channel binding data as a {@link byte[]}, or
     *         null if the binding is unavailable for this connection.
     */
    byte[] getChannelBinding(String channelBinding);

    /**
     * Returns the SSL session in use by this connection
     * @return The {@link BCExtendedSSLSession}.
     */
    BCExtendedSSLSession getSession();
}
```

3.6.1 Recovering the TLS unique ID

The channelBinding string passed to BCSSLConnection.getChannelBinding represents an IANA registered “Channel-binding unique prefix” which is listed as valid for TLS, for example: “tls-unique”. You can make use of these BCJSSE specific interfaces to create a function to retrieve a channel binding like “tls-unique” by following the example that follows.

```
public byte[] getTlsUnique(Socket sock)
{
    BCSSLConnection bcon = ((BCSSLSocket)sock).getConnection();
    if (bcon != null)
    {
```

```
            return bcon.getChannelBinding("tls-unique");
        }
        return null;
    }
```

3.6.2 Specifying the TLS session to resume

Some FTP(S) servers require that the data connection resume the specific TLS session from the control connection. Ordinarily session selection is handled automatically (via the SSLSessionContext cache), which is not a reliable method for ensuring the same-session requirement.

For this and similar cases, BCJSSE provides the BCSSLSocket.getBCSession method (likewise BCSSLEngine.getBCSession) to return the TLS session as a BC-specific type BCExtendedSSLSession (otherwise behaving the same as SSLSocket.getSession). A subsequent connection attempt can then be configured to resume that specific session on a new socket (or engine) via the BCSSLSocket.setBCSessionToResume method (likewise BCSSLEngine.setBCSessionToResume). This method can only be called before any handshaking occurs.

This causes the connection to (attempt to) resume the specified session, instead of delegating session selection to the SSLSessionContext.

3.7 PKCS12-PBMAC1 KeyStore

From release 2.0.23 the BCJSSE provider includes an implementation of the PKCS#12 KeyStore which uses the PBMAC1 algorithm (RFC 9579) for the integrity MAC. PBMAC1 replaces the legacy PKCS#12 password-based MAC with a construction based on PBKDF2 plus an HMAC, allowing the KDF parameters and the underlying HMAC primitive to be chosen explicitly. The KeyStore is registered under the type name "PKCS12-PBMAC1" and is provided by the BCJSSE provider.

A KeyStore of this type is created in the usual way by passing the type name and the BCJSSE provider name to KeyStore.getInstance():

```
KeyStore keyStore = KeyStore.getInstance("PKCS12-PBMAC1", "BCJSSE");
keyStore.load(null, null);
```

When the KeyStore is later written using KeyStore.store(OutputStream, char[]), the resulting PKCS#12 file will use PBMAC1 with HMAC-SHA512 and a PBKDF2-derived 256-bit key as the integrity MAC. A KeyStore created with the type "PKCS12-PBMAC1" can also load existing PKCS#12 files that use the legacy MAC; reading is unaffected by the choice of write-time MAC.

3.7.1 Configuring the PBMAC1 parameters

The default MAC parameters can be overridden when storing a KeyStore by passing an org.bouncycastle.jsse.PKCS12StoreParameter to KeyStore.store(LoadStoreParameter). The PKCS12StoreParameter.PBMAC1WithPBKDF2Builder helper allows the salt, iteration count, derived key size, PRF and underlying HMAC algorithm to be configured. The builder produces an AlgorithmIdentifier suitable for passing to PKCS12StoreParameter.Builder.setMacAlgorithm():

```
KeyStore keyStore = KeyStore.getInstance("PKCS12-PBMAC1", "BCJSSE");
```

```
keyStore.load(in, password);

byte[] salt = new byte[20];
new SecureRandom().nextBytes(salt);

AlgorithmIdentifier macAlg = PKCS12StoreParameter.pbmac1WithPBKDF2Builder()
.setSalt(salt)
.setIterationCount(16384)
.setKeySize(256)
.setPrf(PKCSObjectIdentifiers.id_hmacWithSHA256)
.setMac(PKCSObjectIdentifiers.id_hmacWithSHA512)
.build();

PKCS12StoreParameter storeParam = PKCS12StoreParameter.builder(out, password)
.setMacAlgorithm(macAlg)
.build();
keyStore.store(storeParam);
```

The values shown above are the defaults applied by PBMAC1WithPBKDF2Builder; only setSalt() must be specified to obtain a usable AlgorithmIdentifier. Calling setMacAlgorithm() on PKCS12StoreParameter.Builder is the only mechanism required to switch a KeyStore from the legacy MAC to PBMAC1 on store: the in-memory KeyStore representation is the same in both cases.

3.7.2 Loading legacy PKCS#12 files with non-ASCII passwords

Some legacy PKCS#12 files were written by tools that encoded passwords as ISO-8859-1 (Latin-1) rather than the UTF-8/BMPString form required by PKCS#12. To load such files, the org.bouncycastle.jsse.PKCS12LoadStoreParameter builder provides a setUseISO8859d1ForDecryption() option which, when enabled, will fall back to ISO-8859-1 password encoding if the standard encoding fails:

```
KeyStore keyStore = KeyStore.getInstance("PKCS12-PBMAC1", "BCJSSE");

PKCS12LoadStoreParameter loadParam =
new PKCS12LoadStoreParameter.Builder(in,
new KeyStore.PasswordProtection(password))
.setUseISO8859d1ForDecryption(true)
.build();

keyStore.load(loadParam);
```

4 Using the low-level (D)TLS API

The BCJSSE provider implements TLS over a low-level (D)TLS API located in the org.bouncycastle.tls package. This API can be used directly instead of through the JSSE standard API. It is designed to allow great flexibility to the programmer to modify or extend all aspects of the implementation. This could mean simple configuration options like cipher suites, versions, elliptic curves, or more complex changes like adding support for a new TLS extension.

For most users, using Bouncy Castle's TLS functionality via the BCJSSE provider will be the best choice, especially when working with third party applications that will usually be written to the JSSE standard API. However, there may be use cases for which the JSSE does not expose the necessary functionality, where you need to use DTLS, or for some other reason the JSSE is not appropriate. In some cases, a BC-specific extension may be available to cover the use case, but it is also possible to dispense with the JSSE altogether.

Some familiarity with the TLS specification (as described in RFC 5246 and other related RFCs) is advisable. If you are planning to go down this path it may also be useful to refer to the source code for the TLS test suite for example usage, and to study how the BCJSSE provider itself uses the API.

4.1 TlsCrypto

The low-level (D)TLS API delegates all cryptographic operations to the TlsCrypto API, which is located in the org.bouncycastle.tls.crypto package. Use of this API is via an instance of the TlsCrypto interface; a single instance can be shared across many connections. e.g. the BCJSSE provider uses a single instance for each SSLContext.

There are 2 existing implementations of TlsCrypto available in the regular TLS distribution:

- org.bouncycastle.tls.crypto.impl.bc.BcTlsCrypto
- org.bouncycastle.tls.crypto.impl.jcajce.JcaTlsCrypto

Another option is to write a completely new implementation, or perhaps more realistically to subclass one of these in order to tweak its behaviour. Just keep in mind that TlsCrypto objects should be shareable.

4.1.1 BcTlsCrypto

BcTlsCrypto relies on the lightweight BC crypto API for implementations of cryptographic primitives. The BcTlsCrypto is not available in the BCFIPS TLS jar.

4.1.2 JcaTlsCrypto

JcaTlsCrypto delegates operations to any installed cryptographic providers via the Java Cryptography Architecture (JCA). Although we expect that this will typically be used with the "BC", or “BCFIPS”, provider, it is not a requirement. However, be aware that the set of cipher suites available to the TLS API is constrained by the capabilities of the available providers.

4.2 TLS

The low-level TLS API provides the foundation on which the BCJSSE provider is built, so anything that can be done with the BCJSSE provider can also be done in the low-level TLS API.

4.2.1 Outline of a simple TLS Client

Sample code for a simple TLS client:

```
// TlsCrypto to support client functionality
TlsCrypto crypto = new BcTlsCrypto(new SecureRandom());

…

InetAddress address = InetAddress.getByName("www.example.com");
int port = 443;

Socket s = new Socket(address, port);
TlsClient client = new DefaultTlsClient(crypto) {
            // MUST implement TlsClient.getAuthentication() here
};
TlsClientProtocol protocol = new TlsClientProtocol(
            s.getInputStream(), s.getOutputStream());

// Performs a TLS handshake
protocol.connect(client);

// Read/write to protocol.getInputStream(), protocol.getOutputStream()
…

protocol.close();
```

DefaultTlsClient provides a reasonable default set of cipher suites, like what a typical web browser might support. Other options are PSKTlsClient if you are using pre-shared keys, or SRPTlsClient if using the Secure Remote Password protocol for authentication.

Note that before this code can be used, an implementation of TlsClient.getAuthentication() will have to be provided (see code comment above). That implementation should return an instance of the TlsAuthentication interface, and that instance MUST perform certificate validation in TlsAuthentication.notifyServerCertificate. It could be a simple check that the server certificate is exactly the expected one, or a full path validation as specified in RFC 5280, or something in between, but it is vital to understand that the security of the resulting TLS connection depends on this step.

TlsAuthentication.getClientCredentials will also need to be implemented if you wish for your client to authenticate with the server.

4.2.2 Outline of a simple TLS Server

Writing a TLS server can be rather more work than a TLS client, depending on the range of functionality needed. The top-level code should be quite similar though:

```
// TlsCrypto to support server functionality
TlsCrypto crypto = new BcTlsCrypto(new SecureRandom());

…

int port = 443;
ServerSocket ss = new ServerSocket(port);
Socket s = ss.accept();
```

```
…

TlsServer server = new DefaultTlsServer(crypto) {
      // Override e.g. TlsServer.getRSASignerCredentials() or
      // similar here, depending on what credentials you wish to use.
};

TlsServerProtocol protocol = new TlsServerProtocol(
                  s.getInputStream(), s.getOutputStream());

// Performs a TLS handshake
protocol.accept(server);

// Read/write to protocol.getInputStream(), protocol.getOutputStream()
…

protocol.close();
```

PKSTlsServer or SRPTlsServer are other options here, similar to the client case. Here also, before this can be useful for anything, the issue of what credentials the server will use has to be addressed (see code comments). To support client authentication requires overriding TlsServer.getCertificateRequest() and TlsServer.notifyClientCertificate, and the latter MUST perform certificate validation before the client is considered authenticated.

The TlsClient or TlsServer implementation is where most customisation will likely take place, though it is possible to subclass TlsClientProtocol or TlsServerProtocol, if necessary.

4.3 DTLS

DTLS clients and servers uses different protocol classes and a different transport than the TLS ones, but the same TlsClient and/or TlsServer implementation will usually work fine with the DTLS protocol classes. There are some minor exceptions, e.g. DTLS does not support cipher suites based on stream ciphers like RC4.

Example of client code differences from the TLS client example:

```
DatagramSocket socket = new DatagramSocket();
socket.connect(InetAddress.getByName("www.example.com"), 5556);

int mtu = 1500;
DatagramTransport transport = new UDPTransport(socket, mtu);

TlsClient client = ...;
DTLSClientProtocol protocol = new DTLSClientProtocol();
DTLSTransport dtls = protocol.connect(client, transport);

// Send/receive packets using dtls methods

…

dtls.close();
```

The server is again very similar:

```
DatagramTransport transport = ...;
TlsServer server = ...;
```

```
DTLSServerProtocol protocol = new DTLSServerProtocol();
DTLSTransport dtls = protocol.accept(server, transport);

// Send/receive packets using dtls methods
…

dtls.close();
```

4.4 Using Heartbeats

TLS/DTLS heartbeats are described in RFC 6520 and can be supported by overriding the methods `getHeartbeat()` and `getHeartbeatPolicy()` on the TlsPeer interface in either your client or server implementation. Currently this is supported for DTLS only.

The `getHeartbeat()` method returns an implementation of a `TlsHeartbeat` interface. This is used internally to provide heartbeat details. A default implementation based on a simple counter of `TlsHeartbeat` in the the `DefaultTlsHeartbeat` class.

The `getHearteatPolicy()` method needs to return one of the values defined in `HeartbeatMode`. Currently there are two values defned `HeartbeatMode.peer_not_allowed_to_send` and `HeartbeatMode.peer_allowed_to_send`. The default implementation of client and server objects returns `HeartbeatMode.peer_not_allowed_to_send`. Where you want the opposite party to know that it is appropriate to send heartbeat messages you should override the `getHeartbeatPolicy()` method to return `HeartbeatMode.peer_allowed_to_send`.

Appendix A – Security Properties

The following standard security properties are respected by the BCJSSE provider:

keystore.type: (via java.security.KeyStore.getDefaultType()) The default keystore type.

keystore.type.compat: Controls compatibility mode for JKS and PKCS12 keystore types.

jdk.certpath.disabledAlgorithms: Algorithm restrictions for certification path (CertPath) processing.

jdk.tls.disabledAlgorithms: Algorithm restrictions for TLS processing.

ssl.KeyManagerFactory.algorithm: (via javax.net.ssl.KeyManagerFactory.getDefaultAlgorithm()) The default key manager factory algorithm.

ssl.TrustManagerFactory.algorithm: (via javax.net.ssl.TrustManagerFactory.getDefaultAlgorithm()) The default trust manager factory algorithm.

Appendix B – System Properties

The following standard system properties are respected by the BCJSSE provider:

com.sun.net.ssl.checkRevocation: Used to enable revocation checking when the PKIX trust manager creates default PKIX parameters.

com.sun.net.ssl.requireCloseNotify: Whether to require a close_notify warning alert is sent before a connection is closed, as specified in TLS but not always implemented correctly. BCJSSE uses a default value of “true”.

java.home: Java installation directory.

javax.net.ssl.keyStore: Name of the key store file that holds the endpoint's private key. This is used to create a KeyManager internally.

javax.net.ssl.keyStorePassword: Password to use to get access to the key store file holding the endpoint's private key.

javax.net.ssl.keyStoreType: Type of the key store file holding the endpoint's private key.

javax.net.ssl.keyStoreProvider: The name of the provider which provides implementation support for the key store specified in the javax.net.ssl.keyStore property.

javax.net.ssl.sessionCacheSize: Configures the default value of the 'sessionCacheSize' property for session contexts; see `SSLSessionContext.setSessionCachSize(int).`

javax.net.ssl.trustStore: Name of the key store file that holds the trust anchors for validating the end points we are talking to. This is used to create a TrustManager internally.

javax.net.ssl.trustStorePassword: Password to use to get access to the key store file holding the trust anchors.

javax.net.ssl.trustStoreType: Type of the key store file holding the trust anchors.

javax.net.ssl.trustStoreProvider: The name of the provider which provides implementation support for the key store specified in the javax.net.ssl.trustStore property.

jdk.tls.allowLegacyMasterSecret: Whether to permit connections that did not negotiate the extended_master_secret extension.

jdk.tls.allowLegacyResumption: Whether to permit resumption of a session where the original connection did not negotiate the extended_master_secret extension.

jdk.tls.client.cipherSuites: Specify the default enabled cipher suites for TLS clients, in a comma-separated list.

jdk.tls.client.enableCAExtension: If set to “true”, enables the certificate_authorities extension, an optional ClientHello extension introduced in TLS 1.3.

jdk.tls.client.enableStatusRequestExtension: Enables “OCSP stapling” per the status_request extension from RFC 6066.

jdk.tls.client.protocols: To enable specific TLS protocols, specify them in a comma-separated list within quotation marks and all other supported protocols will be disabled. For example, if the value of this property is `"TLSv1,TLSv1.1"`, then the default protocol settings is only for TLSv1 and TLSv1.1 with nothing else available.

jdk.tls.client.SignatureSchemes: Contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the client side.

jdk.tls.client.useCompatibilityMode: A boolean property which defaults to true. When set to true, enables compatibility mode per RFC 8446 4.1.2. In compatibility mode a client not offering a pre-TLS 1.3 session will generate a new 32-byte value for the 'legacy_session_id' field.

jdk.tls.ephemeralDHKeySize: Set this value to customize the size of ephemeral DH keys (in bits). This value defaults to 2048, with a minimum of 1024.

jdk.tls.maxCertificateChainLength: Specifies the maximum allowed length of the certificate chain in TLS handshaking.

jdk.tls.maxHandshakeMessageSize: Specifies the maximum allowed size, in bytes, for the handshake message in TLS handshaking.

jdk.tls.namedGroups: Defines the set of enabled curves and/or finite-field groups, in order of preference. The format is a comma-separated list of group names e.g. "secp256r1, secp384r1, secp521r1". See RFC 4492, RFC 7919, and/or the IANA's TLS Supported Groups Registry for the full list of group names.

jdk.tls.server.cipherSuites: Specify the default enabled cipher suites for TLS servers, in a comma-separated list.

jdk.tls.server.enableCAExtension: If set to “true”, enables the certificate_authorities extension, an optional CertificateRequest extension introduced in TLS 1.3.

jdk.tls.server.protocols: To enable specific TLS server protocols, specify them in a comma-separated list within quotation marks and all other supported protocols will be disabled. For example, if the value of this property is `"TLSv1,TLSv1.1"`, then the default protocol settings is only for TLSv1 and TLSv1.1 with nothing else available.

jdk.tls.server.SignatureSchemes: Contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the client side .

jdk.tls.trustNameService: Can be used to switch on reverse name lookup where it is needed for end point identification. Note: enabling reverse name lookup may be susceptible to MITM attacks when a name service is not trustworthy - see also section 3.5 before you use this.

jdk.tls.useExtendedMasterSecret: If set to “false”, disables negotiation of the extended_master_secret extension.

jsse.enableFFDHE: If set to “false”, disables the TLS FFDHE mechanisms defined in RFC 7919.

jsse.enableSNIExtension: If set to “true” enable the Server Name Indication (SNI) extension.

sun.security.ssl.allowLegacyHelloMessages: If set to "true", allows connections to peers that do not implement secure renegotiation per RFC 5746.

There are also some BCJSSE-specific properties:

org.bouncycastle.jsse.client.assumeOriginalHostName: A BCJSSE-specific substitute for jdk.tls.trustNameService.

org.bouncycastle.jsse.client.dh.disableDefaultSuites: Affects client connections only. When set to true, no DH cipher suites will be included in the default list. Does not restrict explicit configuration. Defaults to false.

org.bouncycastle.jsse.client.dh.minimumPrimeBits: Integer property, default 2048, can be configured in the range 1024 to 16384.

org.bouncycastle.jsse.client.dh.unrestrictedGroups: boolean property, default false, if set (exact string) to "true" will accept any DH group meeting the size requirement (i.e. minimumPrimeBits above).

org.bouncycastle.jsse.client.earlyKeyShares: Configure the early key_share groups, in preference order. This is only applicable to a TLS client offering TLS 1.3 or higher (otherwise ignored). The format and names are the same as for 'jdk.tls.namedGroups'. Groups in this list will be ignored if they are not already enabled as supported groups (either by default or e.g. with 'jdk.tls.namedGroups' or other configuration).

org.bouncycastle.jsse.client.enableSessionResumption: Controls whether automatic session resumption is enabled in TLS clients (when supported).

org.bouncycastle.jsse.client.enableTrustedCAKeysExtension: Controls whether the trusted_ca_keys extension from RFC 6066 is enabled for TLS clients.

org.bouncycastle.jsse.client.omitSigAlgsCertExtension: A boolean property which defaults to true. When set to true, the client will omit the 'signature_algorithms_cert' extension if it would otherwise have the same value as the 'signature_algorithms' extension.

org.bouncycastle.jsse.config: an alternate method of passing a config string to the JSSE provider. Set to “default” to behave as normal, otherwise other configs, such as “fips:BCFIPS” can also be used.

org.bouncycastle.jsse.fips.allowRSAKeyExchange: boolean property, default false, if set to “true” will allow the use of cipher suites based on the use of RSA key exchange in fips mode. Note: the default value for this property is “false” as the SP 800-131A transition away from RSA key exchange has now taken affect.

org.bouncycastle.jsse.ec.disableChar2: boolean property, default false, if set (exact string) to “true” will disable the use of any characteristic 2, or F_{2m}, curves in TLS handshakes and key exchanges.

org.bouncycastle.jsse.keyManager.checkEKU: Controls whether BCJSSE key managers will

check for an appropriate Extended Key Usage when selecting a set of credentials to present to the TLS peer. Note that a certificate with an invalid EKU may be rejected by the peer.

org.bouncycastle.jsse.server.dh.disableDefaultSuites: Affects server connections only. When set to true, no DH cipher suites will be included in the default list. Does not restrict explicit configuration. Defaults to false.

org.bouncycastle.jsse.server.enableSessionResumption: Controls whether automatic session resumption is enabled in TLS servers (when supported).

org.bouncycastle.jsse.server.enableTrustedCAKeysExtension: Controls whether the trusted_ca_keys extension from RFC 6066 is enabled for TLS servers.

org.bouncycastle.jsse.server.omitSigAlgsCertExtension: A boolean property which defaults to true. When set to true, the server will omit the 'signature_algorithms_cert' extension if it would otherwise have the same value as the 'signature_algorithms' extension.

org.bouncycastle.jsse.trustManager.checkEKU: Controls whether BCJSSE trust managers will check for an appropriate Extended Key Usage extension as part of checking whether a given certificate chain is trusted.

Appendix C – Supported Cipher Suites

Cipher Suite Name	FIPS mode?
TLS_AES_128_CCM_8_SHA256	Y
TLS_AES_128_CCM_SHA256	Y
TLS_AES_128_GCM_SHA256	Y
TLS_AES_256_GCM_SHA384	Y
TLS_CHACHA20_POLY1305_SHA256	N
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	N
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	Y
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	Y
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	Y[2]
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	Y
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	Y
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	Y[2]
TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256	N
TLS_DHE_DSS_WITH_ARIA_128_GCM_SHA256	N
TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384	N
TLS_DHE_DSS_WITH_ARIA_256_GCM_SHA384	N
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	N
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256	N
TLS_DHE_DSS_WITH_CAMELLIA_128_GCM_SHA256	N
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	N
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256	N
TLS_DHE_DSS_WITH_CAMELLIA_256_GCM_SHA384	N
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	N
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	Y

2 Only enabled for BC-FJA 1.0.2, GCM disabled with BC-FJA 1.0.0 and BC-FJA 1.0.1 due to requirements of FIPS 140-2 IG A.5

TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	Y
TLS_DHE_RSA_WITH_AES_128_CCM	Y
TLS_DHE_RSA_WITH_AES_128_CCM_8	Y
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	Y[2]
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	Y
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	Y
TLS_DHE_RSA_WITH_AES_256_CCM	Y
TLS_DHE_RSA_WITH_AES_256_CCM_8	Y
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	Y[3]
TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256	N
TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256	N
TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384	N
TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384	N
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	N
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	N
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	N
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	N
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	N
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	N
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256	N
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	N
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	Y
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	Y
TLS_ECDHE_ECDSA_WITH_AES_128_CCM	Y
TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8	Y

TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	Y[3]
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	Y
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	Y
TLS_ECDHE_ECDSA_WITH_AES_256_CCM	Y
TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8	Y
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	Y[3]
TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256	N
TLS_ECDHE_ECDSA_WITH_ARIA_128_GCM_SHA256	N
TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384	N
TLS_ECDHE_ECDSA_WITH_ARIA_256_GCM_SHA384	N
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256	N
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256	N
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384	N
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384	N
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256	N
TLS_ECDHE_ECDSA_WITH_NULL_SHA	N
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	N
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	Y
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	Y
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	Y[4]
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	Y
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	Y
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	Y[4]
TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256	N
TLS_ECDHE_RSA_WITH_ARIA_128_GCM_SHA256	N

3 Only enabled for BC-FJA 1.0.2, GCM disabled with BC-FJA 1.0.0 and BC-FJA 1.0.1 due to requirements of FIPS 140-2 IG A.5

4 Only enabled for BC-FJA 1.0.2, GCM disabled with BC-FJA 1.0.0 and BC-FJA 1.0.1 due to requirements of FIPS 140-2 IG A.5

TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384	N
TLS_ECDHE_RSA_WITH_ARIA_256_GCM_SHA384	N
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	N
TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	N
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384	N
TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	N
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	N
TLS_ECDHE_RSA_WITH_NULL_SHA	N
TLS_RSA_WITH_3DES_EDE_CBC_SHA	N
TLS_RSA_WITH_AES_128_CBC_SHA	Y[5]
TLS_RSA_WITH_AES_128_CBC_SHA256	Y[5]
TLS_RSA_WITH_AES_128_CCM	Y[5]
TLS_RSA_WITH_AES_128_CCM_8	Y[5]
TLS_RSA_WITH_AES_128_GCM_SHA256	Y[5,6]
TLS_RSA_WITH_AES_256_CBC_SHA	Y[5]
TLS_RSA_WITH_AES_256_CBC_SHA256	Y[5]
TLS_RSA_WITH_AES_256_CCM	Y[5]
TLS_RSA_WITH_AES_256_CCM_8	Y[5]
TLS_RSA_WITH_AES_256_GCM_SHA384	Y[5,6]
TLS_RSA_WITH_ARIA_128_CBC_SHA256	N
TLS_RSA_WITH_ARIA_128_GCM_SHA256	N
TLS_RSA_WITH_ARIA_256_CBC_SHA384	N
TLS_RSA_WITH_ARIA_256_GCM_SHA384	N
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	N
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	N
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256	N
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	N

5 Now disabled by default due to the ending of the RSA PKCS1.5 transition period.

6 Only enabled for BC-FJA 1.0.2, GCM disabled with BC-FJA 1.0.0 and BC-FJA 1.0.1 due to requirements of FIPS 140-2 IG A.5

TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	N
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384	N
TLS_RSA_WITH_NULL_SHA	N
TLS_RSA_WITH_NULL_SHA256	N

Appendix D – Logging

D.1 In the low-level API

The low-level TLS/DTLS API provides callbacks and notifyAlertRaised/Received methods to allow developers to capture activity and events for logging purposes. The source of the JSSE provider can be used to provide some guidance as to how this would be done.

D.2 In the JSSE provider

The JSSE provider uses Java Logging for logging errors and other events of interest. Java Logging is part of the standard class library of the JRE, so no extra library dependencies are needed.

Java Logging is enabled by default, with the output written to the standard error console, i.e. "System.err", but it can be configured either programmatically, or via a configuration properties file. The default configuration is in the /lib/logging.properties in the JRE directory. The system property "java.util.logging.config.file" can be used to specify an alternate location from which to read the configuration.

The root name space for all logging messages from the provider is "org.bouncycastle.jsse".

Please see
https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html
for more details on Java Logging configuration

Appendix E – Utility Classes used in Examples

E.1 Simple Protocol Class

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import org.bouncycastle.util.Strings;
/**
 * Simple protocol to execute.
 */
public class Protocol
{
    /**
     * Carry out the '!' protocol - client side.
     */
    static void doClientSide(
        Socket cSock)
        throws IOException
    {
        OutputStream     out = cSock.getOutputStream();
        InputStream      in = cSock.getInputStream();
        out.write(Strings.toByteArray("World"));
        out.write('!');
        int ch = 0;
        while ((ch = in.read()) != '!')
        {
            System.out.print((char)ch);
        }
        System.out.println((char)ch);
    }
    /**
     * Carry out the '!' protocol - server side.
     */
    static void doServerSide(
        Socket sSock)
        throws IOException
    {
        System.out.println("session started.");
        InputStream in = sSock.getInputStream();
        OutputStream out = sSock.getOutputStream();
        out.write(Strings.toByteArray("Hello "));
        int ch = 0;
        while ((ch = in.read()) != '!')
        {
            out.write(ch);
        }
        out.write('!');
        sSock.close();
        System.out.println("session closed.");
    }
}

import java.math.BigInteger;
```

E.2 Utils Class

```
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Date;
import javax.security.auth.x500.X500Principal;
import javax.security.auth.x500.X500PrivateCredential;
import org.bouncycastle.asn1.x509.BasicConstraints;
import org.bouncycastle.asn1.x509.Extension;
import org.bouncycastle.asn1.x509.KeyUsage;
import org.bouncycastle.cert.X509CertificateHolder;
import org.bouncycastle.cert.jcajce.JcaX509CertificateConverter;
import org.bouncycastle.cert.jcajce.JcaX509ExtensionUtils;
import org.bouncycastle.cert.jcajce.JcaX509v1CertificateBuilder;
import org.bouncycastle.cert.jcajce.JcaX509v3CertificateBuilder;
import org.bouncycastle.operator.ContentSigner;
import org.bouncycastle.operator.jcajce.JcaContentSignerBuilder;
import org.bouncycastle.util.encoders.Base64;
/**
 * Certificate/Key Utilities for the examples.
 */
public class Utils
{
    /**
     * Host name for our examples to use.
     */
    static final String HOST = "localhost";

    /**
     * Port number for our examples to use.
     */
    static final int PORT_NO = 9020;
    /**
     * Names and passwords for the key store entries we need.
     */
    public static String ROOT_ALIAS = "root";
    public static String INTERMEDIATE_ALIAS = "intermediate";
    public static String END_ENTITY_ALIAS = "end";
    public static final char[] SERVER_PASSWORD = "serverPassword".toCharArray();
    public static final char[] CLIENT_PASSWORD = "clientPassword".toCharArray();
    private static long baseTime = 0x15c57a33402L;

    private static final byte[] rootPrivateKey = Base64.decode(
        "MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQD" +
        "DYu2zJUsZAKQ31RzVqteZQwf4lxi3T8TCP8DSQ7Ke4IQp3DDKVP" +
        "9NUwVHRr/s0OZphln6JyUBkSHuQ2hTx4UQRoef2g06WLlFAHi1R" +
        "wr0QefkdGwhPgSuXMWh4dZ1AYGwuIK3KUIUfUU7x5aiwSO6Lyj5" +
        "BYTQQqeX4VFMmp1SMMU7tI88R5bAOkiJw1Wz/840BfVowPTR4Wt" +
        "TGgq33OJ8gb6GH8k1t8CdvmuFArV5D+iPApiluwVVCVINkSN0Aj" +
        "XwmkjtHUMOA+qC7aeZkzke4vPSSy1QABIzmAXIq1zUxS6o9DUqE" +
        "H9gLF1e91uqwKjmjj9SYnhZxHumEEx42J/XAgMBAAECggEAWdQZ" +
```

```
        "SYQrTx7q4RpzK87kWXumZgV9oQWlBdOOwHzMWdwKFz67FcLXL4M" +
        "sSZU+9s8iJ8DTjD1D98D0cxj9lYsE47Mxdm4nJ7yTzSQG2v0DDc" +
        "JhLTjTX8MmHs3bNO5iDSA4snlZ64Cl90qSsoWz/TbDyL0W3spJQ" +
        "gBrEdpO6OOq0ZZ54zekawgyG677aJbzInAG2o9b066HvGRSWNb3" +
        "Celw7RKvjPOohKPOWbSm2W/5gnlSnTaAUgm7W8A1AClTt7scyqg" +
        "PEtThxQHiBGorI6UGjVuO0xoT1MgYr2QWKaYJydo8mFaygaJxVJ" +
        "Hs1PeFQIhuJn7rA/F5BO8cFpuZGrYPUQKBgQDjJYQ7pDrrgRF8O" +
        "TDkvbR5lQdBEWHlK8MMD+1kyptwc8UpK2phZjqLOMofsLhURkgm" +
        "Fzc0UOEx03MdGJvHrWgGQRBc+0JHvYLzCepbOFumjkSPwbb2yQH" +
        "R9QfOPbDRpaqdFNTJnm4lQHZdTGTR4UvDX1X0PuCksRAVtPRA6P" +
        "sBsQKBgQDcNJ5H/ZwkSpT8ZA9GzdVJtxoCLjQPyi1AYYZp0xDUo" +
        "D0h6+JnDljFnsWnpy9OcoJAA6pCkQe+6Cm0vlLvMQ8eD9rcQ/+s" +
        "JFacr7lE3K9bYt56PBTLHyE+WYy90mOVu7FtLfOLz9XDjzyGMn2" +
        "ELuFrUjxlnI7ZCbpZh/GwXiXUBwKBgQDPTPbwg4KuWb2+dGd16t" +
        "ghuevD63w/bX/1qzeJrArORynh19ifiW/WjX6SC3M+nmHMOZXNL" +
        "h9HnOXK4SGSy2RLiOfJJBoqZP90lVEH7VhfmiliVXWIpov9tLVp" +
        "+Q09WAdsko1ccDWv07Pyk/zTOt0tMf29CgF07I90cBAWiUpDEQK" +
        "BgBycTZBm+BmTAyaDzaRSbArm2l88J5GBoD2ELlWjkcU+iJLWth" +
        "TTvV730RCGXVQg9qFgmIeLlmkMexa7v8TKJ/+s6a/Cuf5gvkwfX" +
        "MAAuFv0TZmuIrl9cvFJ60pigoPa3iOkW8dnmouNGb0J5Fr/SFSM" +
        "W8KMA9dZNzgYvKNAqEOTAoGBALIUD1PsOGciRA8htw3jA8hhaH8" +
        "rM+UeQEMC87QnsMEYTuXmkvsDHDNpkcs//X3woQBww+ll1qfByP" +
        "Wj4/GNn4vPjwah4M+6c2xFUez3hLpexD0qoeOS3udAXDGfvBiAT" +
        "zXkaQ1kp2LHPuQdBMGRM4vnbDYGjtq40khezAfHErK0");

private static final byte[] rootPublicKey = Base64.decode(
        "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAw2LtsyV" +
        "LGQCkN9Uc1arXmUMH+JcYt0/Ewj/A0kOynuCEKdwwylT/TVMFR0" +
        "a/7NDmaYZZ+iclAZEh7kNoU8eFEEaHn9oNOli5RQB4tUcK9EHn5" +
        "HRsIT4ErlzFoeHWdQGBsLiCtylCFH1FO8eWosEjui8o+QWE0EKn" +
        "l+FRTJqdUjDFO7SPPEeWwDpIicNVs//ONAX1aMD00eFrUxoKt9z" +
        "ifIG+hh/JNbfAnb5rhQK1eQ/ojwKYpbsFVQlSDZEjdAI18JpI7R" +
        "1DDgPqgu2nmZM5HuLz0kstUAASM5gFyKtc1MUuqPQ1KhB/YCxdX" +
        "vdbqsCo5o4/UmJ4WcR7phBMeNif1wIDAQAB");

                                                // 1 week
private static final long VALIDITY_PERIOD = 7 * 24 * 60 * 60 * 1000L;


/**
 * Create a random 2048 bit RSA key pair - okay for client and intermediate
 */
public static KeyPair generateRSAKeyPair()
    throws Exception
{
    KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA", "BC");
    kpGen.initialize(2048, new SecureRandom());
    return kpGen.generateKeyPair();
}

/**
 * Create a fixed 2048 bit RSA key pair — to keep the server cert stable
 */
public static KeyPair generateRootKeyPair()
    throws Exception
{
    KeyFactory kFact = KeyFactory.getInstance("RSA", "BC");

    return new KeyPair(
```

```
          kFact.generatePublic(new X509EncodedKeySpec(rootPublicKey)),
        kFact.generatePrivate(new PKCS8EncodedKeySpec(rootPrivateKey)));
}

/**
 * Generate a sample V1 certificate to use as a CA root certificate
 */
public static X509CertificateHolder generateRootCert(KeyPair pair)
    throws Exception
{
    JcaX509v1CertificateBuilder certBldr = new JcaX509v1CertificateBuilder(
        new X500Principal("CN=Test CA Certificate"),
        BigInteger.valueOf(1),
        new Date(baseTime),                // allow 1024 weeks for the root
        new Date(baseTime + 1024 * VALIDITY_PERIOD),
        new X500Principal("CN=Test CA Certificate"),
        pair.getPublic()
    );

    ContentSigner signer = new JcaContentSignerBuilder("SHA256withRSA")
                                .setProvider("BC").build(pair.getPrivate());

    return certBldr.build(signer);
}

/**
 * Generate a sample V3 certificate to use as an
 * intermediate CA certificate
 */
public static X509CertificateHolder generateIntermediateCert(
                PublicKey intKey, PrivateKey caKey, X509Certificate caCert)
    throws Exception
{
    long timeMillis = System.currentTimeMillis();
    JcaX509v3CertificateBuilder certBldr = new JcaX509v3CertificateBuilder(
           caCert.getSubjectX500Principal(),
           BigInteger.valueOf(1),
           new Date(timeMillis),
           new Date(timeMillis + VALIDITY_PERIOD),
           new X500Principal("CN=Test Intermediate Certificate"),
           intKey
    );

    JcaX509ExtensionUtils utils = new JcaX509ExtensionUtils();

    certBldr.addExtension(
                  Extension.authorityKeyIdentifier, false,
                  utils.createAuthorityKeyIdentifier(caCert));
    certBldr.addExtension(
                  Extension.subjectKeyIdentifier, false,
                  utils.createSubjectKeyIdentifier(intKey));
    certBldr.addExtension(
                  Extension.basicConstraints, true,
                  new BasicConstraints(0));
    certBldr.addExtension(
                  Extension.keyUsage, true,
                  new KeyUsage(KeyUsage.digitalSignature |
                               KeyUsage.keyCertSign | KeyUsage.cRLSign));

    ContentSigner signer = new JcaContentSignerBuilder("SHA256withRSA")
```

```
                                                .setProvider("BC").build(caKey);

        return certBldr.build(signer);
    }

    /**
     * Generate a sample V3 certificate to use as an end entity certificate
     */
    public static X509CertificateHolder generateEndEntityCert(
            PublicKey entityKey, PrivateKey caKey, X509Certificate caCert)
       throws Exception
    {
        long timeMillis = System.currentTimeMillis();
        JcaX509v3CertificateBuilder certBldr = new JcaX509v3CertificateBuilder(
               caCert.getSubjectX500Principal(),
               BigInteger.valueOf(1),
               new Date(timeMillis),
               new Date(timeMillis + VALIDITY_PERIOD),
               new X500Principal("CN=Test End Certificate"),
               entityKey
        );

        JcaX509ExtensionUtils utils = new JcaX509ExtensionUtils();

        certBldr.addExtension(
                        Extension.authorityKeyIdentifier, false,
                        utils.createAuthorityKeyIdentifier(caCert));
        certBldr.addExtension(
                        Extension.subjectKeyIdentifier, false,
                        utils.createSubjectKeyIdentifier(entityKey));
        certBldr.addExtension(
                        Extension.basicConstraints, true,
                        new BasicConstraints(false));
        certBldr.addExtension(
                        Extension.keyUsage, true,
                        new KeyUsage(KeyUsage.digitalSignature));

        ContentSigner signer = new JcaContentSignerBuilder("SHA256withRSA")
                                                .setProvider("BC").build(caKey);

        return certBldr.build(signer);
    }

    /**
     * Generate a X500PrivateCredential for the root entity.
     */
    public static X500PrivateCredential createRootCredential()
        throws Exception
    {
        KeyPair rootPair = generateRootKeyPair();

        X509Certificate rootCert = convertCert(generateRootCert(rootPair));

        return new X500PrivateCredential(
                        rootCert, rootPair.getPrivate(), ROOT_ALIAS);
    }

    /**
     * Generate a X500PrivateCredential for the intermediate entity.
     */
```

```
public static X500PrivateCredential createIntermediateCredential(
    PrivateKey caKey,
    X509Certificate caCert)
    throws Exception
{
    KeyPair         interPair = generateRSAKeyPair();
    X509Certificate interCert = convertCert(generateIntermediateCert(
                                    interPair.getPublic(), caKey, caCert));
    return new X500PrivateCredential(
                    interCert, interPair.getPrivate(), INTERMEDIATE_ALIAS);
}

/**
 * Generate a X500PrivateCredential for the end entity.
 */
public static X500PrivateCredential createEndEntityCredential(
    PrivateKey      caKey,
    X509Certificate caCert)
    throws Exception
{
    KeyPair         endPair = generateRSAKeyPair();
    X509Certificate endCert = convertCert(
                generateEndEntityCert(endPair.getPublic(), caKey, caCert));
    return new X500PrivateCredential(
                 endCert, endPair.getPrivate(), END_ENTITY_ALIAS);
}

private static X509Certificate convertCert(X509CertificateHolder certHolder)
    throws CertificateException
{
    return new JcaX509CertificateConverter()
                      .setProvider("BC").getCertificate(certHolder);
}

/**
 * Create a server trust store.
 *
 * @return a key store containing the example server certificate
 */
public static KeyStore createServerTrustStore()
    throws Exception
{
    X500PrivateCredential serverCred = createRootCredential();

    KeyStore keyStore = KeyStore.getInstance("JKS");
    keyStore.load(null, null);

    keyStore.setCertificateEntry(
                    serverCred.getAlias(), serverCred.getCertificate());
    return keyStore;
}

/**
 * Create a client trust store.
 *
 * @return a key store containing the example client trust anchor
 */
public static KeyStore createClientTrustStore()
    throws Exception
```

```
    {
                // for brevity we use the same TA as the server.
        return createServerTrustStore();
    }

    /**
     * Create a server key store.
     *
     * @return a key store containing the example server
     *          certificate and private key
     */
    public static KeyStore createServerKeyStore()
        throws Exception
    {
        X500PrivateCredential serverCred = createRootCredential();
        KeyStore keyStore = KeyStore.getInstance("JKS");
        keyStore.load(null, null);

        keyStore.setKeyEntry(
            serverCred.getAlias(), serverCred.getPrivateKey(), SERVER_PASSWORD,
            new X509Certificate[] { serverCred.getCertificate() });

        return keyStore;
    }

    /**
     * Create a client key store - for client side authentication.
     *
     * @return a key store containing the example client
     *          certificate and private key
     */
    public static KeyStore createClientKeyStore()
        throws Exception
    {
        X500PrivateCredential serverCred = createRootCredential();
        X500PrivateCredential interCred = createIntermediateCredential(
                    serverCred.getPrivateKey(), serverCred.getCertificate());
        X500PrivateCredential clientCred = createEndEntityCredential(
                      interCred.getPrivateKey(), interCred.getCertificate());

        KeyStore keyStore = KeyStore.getInstance("JKS");
        keyStore.load(null, null);

        keyStore.setKeyEntry(
            clientCred.getAlias(), clientCred.getPrivateKey(), CLIENT_PASSWORD,
            new X509Certificate[] {
                    clientCred.getCertificate(), interCred.getCertificate() });

        return keyStore;
    }
}
```