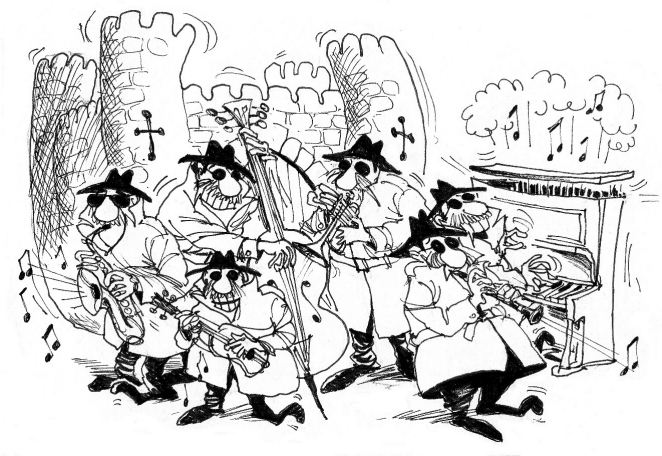
Legion of the Bouncy Castle Inc.
Java (D)TLS API and JSSE Provider

User Guide

Version: 1.0.0 Date: 06/20/17



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 https://www.cryptoworkshop.com.

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 JSSE Provider installation into the JRE

Once the bctls jar is installed, the provider class BouncyCastleFipsProvider 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 JSSE Provider in FIPS mode

The JSSE 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”));
```

2.2 Other Differences between the BCJSSE and JSSE provider

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

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.

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.

3.1.2 Instance Names

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

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

SSLContext instances are available for “TLSv1”, “TLSv1.1”, and “TLSv1.2”. In each case the name of the instance represents the lowest version of TLS supported and higher versions are supported as well. The SSLContext instance names “TLS” and DEFAULT provide support for TLSv1.2.

3.2 A basic Server

A basic server only requires a KeyManager to identify itself with. The utility class described in Appendix D – 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 BC 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.jce.provider.BouncyCastleProvider;
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 BouncyCastleProvider());
        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.jce.provider.BouncyCastleProvider;
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 BouncyCastleProvider());
        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.jce.provider.BouncyCastleProvider;
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 BouncyCastleProvider());
        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.jce.provider.BouncyCastleProvider;
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 BouncyCastleProvider());
        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 BC Extensions - Recovering the TLS unique ID

The BC JSSE returns SSLSocket objects which implement the org.bouncycastle.jsse BCSSLSocket

interface. The BCSSLSocket is defined as follows:

```
public interface BCSSLSocket
{
    BCSSLConnection getConnection();
}
```

with org.bouncycastle.jsse.BCSSLConnection defined as:

```
public interface BCSSLConnection
{
    byte[] getChannelBinding(String channelBinding);

    SSLSession getSession();
}
```

The channelBinding string 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” but 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;
}
```

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();
```

Appendix A – System Properties

The following regular JSSE provider properties are respected by the BCJSSE provider.

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.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.

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

Appendix B – Supported Cipher Suites

Cipher Suite Name	FIPS mode?
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	Y
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	N[1]
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	N
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	N
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	N
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	N
TLS_DHE_RSA_WITH_AES_128_CCM	N
TLS_DHE_RSA_WITH_AES_128_CCM_8	N
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	N
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	N
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	N
TLS_DHE_RSA_WITH_AES_256_CCM	N
TLS_DHE_RSA_WITH_AES_256_CCM_8	N
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	N
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	Y
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	N
TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8	N
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	N
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	Y

1 GCM disabled due to requirements of FIPS 140-2 IG A.5

TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	Y
TLS_ECDHE_ECDSA_WITH_AES_256_CCM	N
TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8	N
TLS_ECDHE_ECDSA_WITH_AES_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	Y
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	N
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	Y
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	N
TLS_ECDHE_RSA_WITH_AES_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	Y
TLS_RSA_WITH_AES_128_CBC_SHA	Y
TLS_RSA_WITH_AES_128_CBC_SHA256	Y
TLS_RSA_WITH_AES_128_CCM	Y
TLS_RSA_WITH_AES_128_CCM_8	N
TLS_RSA_WITH_AES_128_GCM_SHA256	N
TLS_RSA_WITH_AES_256_CBC_SHA	Y
TLS_RSA_WITH_AES_256_CBC_SHA256	Y
TLS_RSA_WITH_AES_256_CCM	Y
TLS_RSA_WITH_AES_256_CCM_8	N
TLS_RSA_WITH_AES_256_GCM_SHA384	N
TLS_RSA_WITH_NULL_SHA	N
TLS_RSA_WITH_NULL_SHA256	N

Appendix C – Logging

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.

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 D – Utility Classes used in Examples

D.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;
```

D.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;
    }
}
```