Java Secure Socket Extension (JSSE) Reference Guide

Introduction

Data that travels across a network can easily be accessed by someone who is not the intended recipient. When the data includes private information, such as passwords and credit card numbers, steps must be taken to make the data unintelligible to unauthorized parties. It is also important to ensure that the data has not been modified, either intentionally or unintentionally, during transport. The Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols were designed to help protect the privacy and integrity of data while it is being transferred across a network.

The Java Secure Socket Extension (JSSE) enables secure Internet communications. It provides a framework and an implementation for a Java version of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Using JSSE, developers can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, or FTP) over TCP/IP. For an introduction to SSL, see Secure Sockets Layer (SSL) Protocol Overview.

By abstracting the complex underlying security algorithms and handshaking mechanisms, JSSE minimizes the risk of creating subtle but dangerous security vulnerabilities. Furthermore, it simplifies application development by serving as a building block that developers can integrate directly into their applications.

JSSE provides both an application programming interface (API) framework and an implementation of that API. The JSSE API supplements the core network and cryptographic services defined by the `java.security` and `java.net` packages by providing extended networking socket classes, trust managers, key managers, SSL contexts, and a socket factory framework for encapsulating socket creation behavior. Because the `SSLSocket` class is based on a blocking I/O model, the Java Development Kit (JDK) includes a nonblocking `SSLEngine` class to enable implementations to choose their own I/O methods.

The JSSE API supports the following security protocols:

- TLS: version 1.0, 1.1, 1.2, and 1.3 (since JDK 8u261)
- SSL (Secure Socket Layer): version 3.0

These security protocols encapsulate a normal bidirectional stream socket, and the JSSE API adds transparent support for authentication, encryption, and integrity protection.

JSSE is a security component of the Java SE platform, and is based on the same design principles found elsewhere in the Java Cryptography Architecture (JCA) framework. This framework for cryptography-related security components allows them to have implementation independence and, whenever possible, algorithm independence. JSSE uses the cryptographic service providers defined by the JCA framework.

Other security components in the Java SE platform include the Java Authentication and Authorization Service (JAAS) and the Java Security Tools. JSSE encompasses many of the same concepts and algorithms as those in JCA but automatically applies them underneath a simple stream socket API.

The JSSE API was designed to allow other SSL/TLS protocol and Public Key Infrastructure (PKI) implementations to be plugged in seamlessly. Developers can also provide alternative logic to determine if remote hosts should be trusted or what authentication key material should be sent to a remote host.

Features and Benefits

JSSE includes the following important features:

- Included as a standard component of the JDK
- Extensible, provider-based architecture
- Implemented in 100% pure Java
- Provides API support for TLS
- Provides implementations of SSL 3.0 and TLS versions 1.0, 1.1, 1.2, and 1.3 (since JDK 8u261)
- Includes classes that can be instantiated to create secure channels (`SSLSocket`, `SSLServerSocket`, and `SSLEngine`)
- Provides support for cipher suite negotiation, which is part of the SSL handshaking used to initiate or verify secure communications
- Provides support for client and server authentication, which is part of the normal SSL handshaking
- Provides support for HTTP encapsulated in the SSL protocol, which allows access to data such as web pages using HTTPS
- Provides server session management APIs to manage memory-resident SSL sessions
- Provides support for the certificate status request extension (OCSP stapling), which saves client certificate validation round-trips and resources
- Provides support for the Server Name Indication (SNI) extension, which extends the TLS protocols to indicate what server name the client is attempting to connect to during handshaking
- Provides support for endpoint identification during handshaking, which prevents man-in-the-middle attacks
- Provides support for cryptographic algorithm constraints, which provides fine-grained control over algorithms negotiated by JSSE

JSSE Standard API

The JSSE standard API, available in the `javax.net` and `javax.net.ssl` packages, provides:

- Secure sockets and server sockets.
- A nonblocking engine for producing and consuming streams of TLS data (`SSLEngine`).
- Factories for creating sockets, server sockets, SSL sockets, and SSL server sockets. By using socket factories, you can encapsulate socket creation and configuration behavior.
- A class representing a secure socket context that acts as a factory for secure socket factories and engines.
- Key and trust manager interfaces (including X.509-specific key and trust managers), and factories that can be used for creating them.
- A class for secure HTTP URL connections (HTTPS).

SunJSSE Provider

Oracle's implementation of Java SE includes a JSSE provider named `SunJSSE`, which comes preinstalled and preregistered with the JCA. This provider supplies the following cryptographic services:

- An implementation of the security protocols SSL 3.0 and TLS 1.0, 1.1, 1.2, and 1.3 (since JDK 8u261).
- An implementation of the most common TLS cipher suites, which encompass a combination of authentication, key agreement, encryption, and integrity protection.
- An implementation of an X.509-based key manager that chooses appropriate authentication keys from a standard JCA keystore.
- An implementation of an X.509-based trust manager that implements rules for certificate chain path validation.
- An implementation of PKCS12 as JCA keystore type "pkcs12". Storing trust anchors in PKCS12 is not supported. Users should store trust anchors in the Java keystore (JKS) format and save private keys in PKCS12 format.

More information about this provider is available in the SunJSSE section of the Oracle Providers Documentation.

Related Documentation

The following list contains links to online documentation and names of books about related subjects:

- **JSSE API Documentation**
  - `javax.net` package
  - `javax.net.ssl` package
  - `javax.security.cert` package
- **Java SE Security**
  - The Java SE Security Documentation index page
  - The Java SE Security home page
  - The Security Features in Java SE trail of the Java Tutorial
  - Java PKI Programmer's Guide
  - Inside Java 2 Platform Security, Second Edition: Architecture, API Design and Implementation
- **Transport Layer Security (TLS)**
  - The TLS Protocol Version 1.0
  - The TLS Protocol Version 1.1
  - The TLS Protocol Version 1.2
  - The TLS Protocol Version 1.3

- Transport Layer Security (TLS) Extensions
- HTTP Over TLS

- **U.S. Encryption Policies**
  - U.S. Department of Commerce
  - Technology CEO Council
  - Current export policies: Encryption and Export Administration Regulations (EAR)
  - NIST Computer Security Publications

Terms and Definitions

Several terms relating to cryptography are used within this document. This section defines some of these terms.

authentication

The process of confirming the identity of a party with whom one is communicating.

cipher suite

A combination of cryptographic parameters that define the security algorithms and key sizes used for authentication, key agreement, encryption, and integrity protection.

certificate

A digitally signed statement vouching for the identity and public key of an entity (person, company, and so on). Certificates can either be self-signed or issued by a Certificate Authority (CA) — an entity that is trusted to issue valid certificates for other entities. Well-known CAs include VeriSign, Entrust, and GTE CyberTrust. X509 is a common certificate format that can be managed by the JDK's `keytool`.

cryptographic hash function

An algorithm that is used to produce a relatively small fixed-size string of bits (called a hash) from an arbitrary block of data. A cryptographic hash function is similar to a checksum and has three primary characteristics: it is a one-way function, meaning that it is not possible to produce the original data from the hash; a small change in the original data produces a large change in the resulting hash; and it does not require a cryptographic key.

Cryptographic Service Provider

Sometimes referred to simply as provider for short, the Java Cryptography Architecture (JCA) defines it as a package (or set of packages) that implements one or more engine classes for specific cryptographic algorithms. An engine class defines a cryptographic service in an abstract fashion without a concrete implementation.

decryption

See encryption/decryption.

digital signature

A digital equivalent of a handwritten signature. It is used to ensure that data transmitted over a network was sent by whoever claims to have sent it and that the data has not been modified in transit. For example, an RSA-based digital signature is calculated by first computing a cryptographic hash of the data and then encrypting the hash with the sender's private key.

encryption/decryption

Encryption is the process of using a complex algorithm to convert an original message (cleartext) to an encoded message (ciphertext) that is unintelligible unless it is decrypted. Decryption is the inverse process of producing cleartext from ciphertext.

The algorithms used to encrypt and decrypt data typically come in two categories: secret key (symmetric) cryptography and public key (asymmetric) cryptography.

handshake protocol

The negotiation phase during which the two socket peers agree to use a new or existing session. The handshake protocol is a series of messages exchanged over the record protocol. At the end of the handshake, new connection-specific encryption and integrity protection keys are generated based on the key agreement secrets in the session.

key agreement

A method by which two parties cooperate to establish a common key. Each side generates some data, which is exchanged. These two pieces of data are then combined to generate a key. Only those holding the proper private initialization data can obtain the final key. Diffie-Hellman (DH) is the most common example of a key agreement algorithm.

key exchange

A method by which keys are exchanged. One side generates a private key and encrypts it using the peer's public key (typically RSA). The data is transmitted to the peer, who decrypts the key using the corresponding private key.

key manager/trust manager

Key managers and trust managers use keystores for their key material. A key manager manages a keystore and supplies public keys to others as needed (for example, for use in authenticating the user to others). A trust manager decides who to trust based on information in the truststore it manages.

keystore/truststore

A keystore is a database of key material. Key material is used for a variety of purposes, including authentication and data integrity. Various types of keystores are available, including PKCS12 and Oracle's JKS.

Generally speaking, keystore information can be grouped into two categories: key entries and trusted certificate entries. A key entry consists of an entity's identity and its private key, and can be used for a variety of cryptographic purposes. In contrast, a trusted certificate entry contains only a public key in addition to the entity's identity. Thus, a trusted certificate entry cannot be used where a private key is required, such as in a `javax.net.ssl.KeyManager`. In the JDK implementation of JKS, a keystore may contain both key entries and trusted certificate entries.

A truststore is a keystore that is used when making decisions about what to trust. If you receive data from an entity that you already trust, and if you can verify that the entity is the one that it claims to be, then you can assume that the data really came from that entity.

An entry should only be added to a truststore if the user trusts that entity. By either generating a key pair or by importing a certificate, the user gives trust to that entry. Any entry in the truststore is considered a trusted entry.

It may be useful to have two different keystore files: one containing just your key entries, and the other containing your trusted certificate entries, including CA certificates. The former contains private information, whereas the latter does not. Using two files instead of a single keystore file provides a cleaner separation of the logical distinction between your own certificates (and corresponding private keys) and others' certificates. To provide more protection for your private keys, store them in a keystore with restricted access, and provide the trusted certificates in a more publicly accessible keystore if needed.

message authentication code (MAC)

Provides a way to check the integrity of information transmitted over or stored in an unreliable medium, based on a secret key. Typically, MACs are used between two parties that share a secret key in order to validate information transmitted between these parties.

A MAC mechanism that is based on cryptographic hash functions is referred to as HMAC. HMAC can be used with any cryptographic hash function, such as Secure Hash Algorithm (SHA-256), in combination with a secret shared key. HMAC is specified in RFC 2104.

public-key cryptography

A cryptographic system that uses an encryption algorithm in which two keys are produced. One key is made public, whereas the other is kept private. The public key and the private key are cryptographic inverses; what one key encrypts only the other key can decrypt. Public-key cryptography is also called asymmetric cryptography.

Record Protocol

A protocol that packages all data (whether application-level or as part of the handshake process) into discrete records of data much like a TCP stream socket converts an application byte stream into network packets. The individual records are then protected by the current encryption and integrity protection keys.

secret-key cryptography

A cryptographic system that uses an encryption algorithm in which the same key is used both to encrypt and decrypt the data. Secret-key cryptography is also called symmetric cryptography.

session

A named collection of state information including authenticated peer identity, cipher suite, and key agreement secrets that are negotiated through a secure socket handshake and that can be shared among multiple secure socket instances.

trust manager

See key manager/trust manager.

truststore

See keystore/truststore.

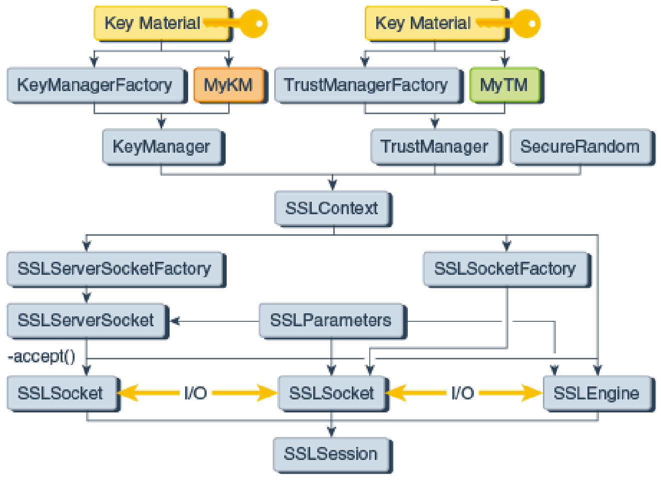
Transport Layer Security (TLS) Protocol Overview

Transport Layer Security (TLS) is the most widely used protocol for implementing cryptography on the web. TLS uses a combination of cryptographic processes to provide secure communication over a network. The page Transport Layer Security (TLS) Protocol Overview provides an introduction to TLS and the cryptographic processes it uses.

JSSE Classes and Interfaces

To communicate securely, both sides of the connection must be SSL-enabled. In the JSSE API, the endpoint classes of the connection are `SSLSocket` and `SSLEngine`. In the figure JSSE Classes Used to Create SSLSocket and SSLEngine, the major classes used to create `SSLSocket` and `SSLEngine` are laid out in a logical ordering. The text following the diagram, explains the contents of the illustration.

Classes Used to Create SSLSocket and SSLEngine



An `SSLSocket` is created either by an `SSLSocketFactory` or by an `SSLServerSocket` accepting an inbound connection. In turn, an `SSLServerSocket` is created by an `SSLServerSocketFactory`. Both `SSLSocketFactory` and `SSLServerSocketFactory` objects are created by an `SSLContext`. An `SSLEngine` is created directly by an `SSLContext`, and relies on the application to handle all I/O.

Note: When using raw `SSLSocket` or `SSLEngine` classes, you should always check the peer's credentials before sending any data. Since JDK 7, endpoint identification/verification procedures can be handled during SSL/TLS handshaking. See the method `SSLParameters.setEndpointIdentificationAlgorithm`. For example, the host name in a URL should match the host name in the peer's credentials. An application could be exploited with URL spoofing if the host name is not verified.

Core Classes and Interfaces

The core JSSE classes are part of the `javax.net` and `javax.net.ssl` packages.

SocketFactory and ServerSocketFactory Classes

The abstract `javax.net.SocketFactory` class is used to create sockets. Subclasses of this class are factories that create particular subclasses of sockets and thus provide a general framework for the addition of public socket-level functionality. For example, see SSLSocketFactory and SSLServerSocketFactory.

The abstract `javax.net.ServerSocketFactory` class is analogous to the `SocketFactory` class, but is used specifically for creating server sockets.

Socket factories are a simple way to capture a variety of policies related to the sockets being constructed, producing such sockets in a way that does not require special configuration of the code that asks for the sockets:

- Due to polymorphism of both factories and sockets, different kinds of sockets can be used by the same application code just by passing different kinds of factories.
- Factories can themselves be customized with parameters used in socket construction. For example, factories could be customized to return sockets with different networking timeouts or security parameters already configured.
- The sockets returned to the application can be subclasses of `java.net.Socket` (or `javax.net.ssl.SSLSocket`), so that they can directly expose new APIs for features such as compression, security, record marking, statistics collection, or firewall tunneling.

SSLSocketFactory and SSLServerSocketFactory Classes

The `javax.net.ssl.SSLSocketFactory` class acts as a factory for creating secure sockets. This class is an abstract subclass of javax.net.SocketFactory.

Secure socket factories encapsulate the details of creating and initially configuring secure sockets. This includes authentication keys, peer certificate validation, enabled cipher suites, and the like.

The `javax.net.ssl.SSLServerSocketFactory` class is analogous to the `SSLSocketFactory` class, but is used specifically for creating server sockets.

Obtaining an SSLSocketFactory

The following ways can be used to obtain an `SSLSocketFactory`:

- Get the default factory by calling the `SSLSocketFactory.getDefault()` static method.
- Receive a factory as an API parameter. That is, code that must create sockets but does not care about the details of how the sockets are configured can include a method with an `SSLSocketFactory` parameter that can be called by clients to specify which `SSLSocketFactory` to use when creating sockets (for example, `javax.net.ssl.HttpsURLConnection`).
- Construct a new factory with specifically configured behavior.

The default factory is typically configured to support server authentication only so that sockets created by the default factory do not leak any more information about the client than a normal TCP socket would.

Many classes that create and use sockets do not need to know the details of socket creation behavior. Creating sockets through a socket factory passed in as a parameter is a good way of isolating the details of socket configuration, and increases the reusability of classes that create and use sockets.

You can create new socket factory instances either by implementing your own socket factory subclass or by using another class which acts as a factory for socket factories. One example of such a class is `SSLContext`, which is provided with the JSSE implementation as a provider-based configuration class.

SSLSocket and SSLServerSocket Classes

The `javax.net.ssl.SSLSocket` class is a subclass of the standard Java `java.net.Socket` class. It supports all of the standard socket methods and adds methods specific to secure sockets. Instances of this class encapsulate the SSLContext under which they were created. There are APIs to control the creation of secure socket sessions for a socket instance, but trust and key management are not directly exposed.

The `javax.net.ssl.SSLServerSocket` class is analogous to the `SSLSocket` class, but is used specifically for creating server sockets.

To prevent peer spoofing, you should always verify the credentials presented to an `SSLSocket`. See Cipher Suite Choice and Remote Entity Verification.

Note: Due to the complexity of the SSL and TLS protocols, it is difficult to predict whether incoming bytes on a connection are handshake or application data, and how that data might affect the current connection state (even causing the process to block). In the Oracle JSSE implementation, the `available()` method on the object obtained by `SSLSocket.getInputStream()` returns a count of the number of application data bytes successfully decrypted from the SSL connection but not yet read by the application.

Obtaining an SSLSocket

Instances of `SSLSocket` can be obtained in one of the following ways:

- An `SSLSocket` can be created by an instance of SSLSocketFactory via one of the several `createSocket()` methods of that class.
- An `SSLSocket` can be created through the `accept()` method of the `SSLServerSocket` class.

Cipher Suite Choice and Remote Entity Verification

The SSL/TLS protocols define a specific series of steps to ensure a protected connection. However, the choice of cipher suite directly affects the type of security that the connection enjoys. For example, if an anonymous cipher suite is selected, then the application has no way to verify the remote peer's identity. If a suite with no encryption is selected, then the privacy of the data cannot be protected. Additionally, the SSL/TLS protocols do not specify that the credentials received must match those that peer might be expected to send. If the connection were somehow redirected to a rogue peer, but the rogue's credentials were acceptable based on the current trust material, then the connection would be considered valid.

When using raw `SSLSocket` and `SSLEngine` classes, you should always check the peer's credentials before sending any data. The `SSLSocket` and `SSLEngine` classes do not automatically verify that the host name in a URL matches the host name in the peer's credentials. An application could be exploited with URL spoofing if the host name is not verified. Since JDK 7, endpoint identification/verification procedures can be handled during SSL/TLS handshaking. See the `SSLParameters.getEndpointIdentificationAlgorithm` method.

Protocols such as HTTPS (HTTP Over TLS) do require host name verification. Since JDK 7, the HTTPS endpoint identification is enforced during handshaking for `HttpsURLConnection` by default. See the `SSLParameters.getEndpointIdentificationAlgorithm` method. Alternatively, applications can use the `HostnameVerifier` interface to override the default HTTPS host name rules.

SSLEngine Class

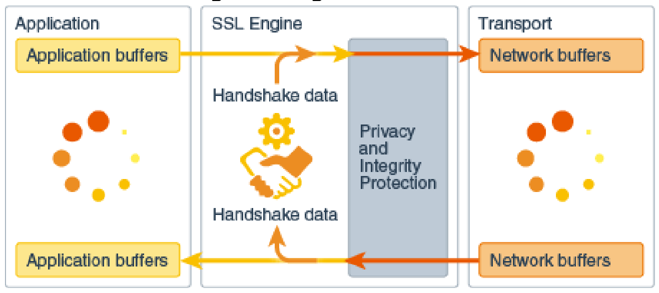
TLS is becoming increasingly popular. It is being used in a wide variety of applications across a wide range of computing platforms and devices. Along with this popularity come demands to use TLS with different I/O and threading models to satisfy the applications' performance, scalability, footprint, and other requirements. There are demands to use TLS with blocking and nonblocking I/O channels, asynchronous I/O, arbitrary input and output streams, and byte buffers. There are demands to use it in highly scalable, performance-critical environments, requiring management of thousands of network connections.

Abstraction of the I/O transport mechanism using the `SSLEngine` class in Java SE allows applications to use the TLS protocol in a transport-independent way, and thus frees application developers to choose transport and computing models that best meet their needs. Not only does this abstraction allow applications to use nonblocking I/O channels and other I/O models, it also accommodates different threading models. This effectively leaves the I/O and threading decisions up to the application developer. Because of this flexibility, the application developer must manage I/O and threading (complex topics in and of themselves), as well as have some understanding of the SSL/TLS protocols. The abstraction is therefore an advanced API: beginners should use `SSLSocket`.

Users of other Java programming language APIs such as the Java Generic Security Services (Java GSS) and the Java Simple Authentication Security Layer (Java SASL) will notice similarities in that the application is also responsible for transporting data.

The core class is `javax.net.ssl.SSLEngine`. It encapsulates a TLS state machine and operates on inbound and outbound byte buffers supplied by the user of the `SSLEngine` class. The diagram Flow of Data Through SSLEngine illustrates the flow of data from the application, through `SSLEngine`, to the transport mechanism, and back.

Flow of Data Through SSLEngine



The application, shown on the left, supplies application (plaintext) data in an application buffer and passes it to `SSLEngine`. The `SSLEngine` object processes the data contained in the buffer, or any handshaking data, to produce TLS encoded data and places it to the network buffer supplied by the application. The application is then responsible for using an appropriate transport (shown on the right) to send the contents of the network buffer to its peer. Upon receiving TLS encoded data from its peer (via the transport), the application places the data into a network buffer and passes it to `SSLEngine`. The `SSLEngine` object processes the network buffer's contents to produce handshaking data or application data.

An instance of the `SSLEngine` class can be in one of the following states:

- **Creation:** The `SSLEngine` has been created and initialized, but has not yet been used. During this phase, an application may set any `SSLEngine`-specific settings (enabled cipher suites, whether the `SSLEngine` should handshake in client or server mode, and so on). Once handshaking has begun, though, any new settings (except client/server mode) will be used for the next handshake.
- **Initial handshaking:** The initial handshake is a procedure by which the two peers exchange communication parameters until an `SSLSession` is established. Application data can't be sent during this phase.
- **Application data:** After the communication parameters have been established and the handshake is complete, application data can flow through the `SSLEngine`. Outbound application messages are encrypted and integrity protected, and inbound messages reverse the process.
- **Rehandshaking:** Either side can request a renegotiation of the session at any time during the Application Data phase. New handshaking data can be intermixed among the application data. Before starting the rehandshake phase, the application may reset the TLS communication parameters such as the list of enabled cipher suites and whether to use client authentication, but can not change between client/server modes. As before, after handshaking has begun, any new `SSLEngine` configuration settings won't be used until the next handshake.
- **Closure:** When the connection is no longer needed, the application should close the `SSLEngine` and should send/receive any remaining messages to the peer before closing the underlying transport mechanism. Once an engine is closed, it is not reusable: a new `SSLEngine` must be created.

Understanding SSLEngine Operation Statuses

To indicate the status of the engine and what actions the application should take, the `SSLEngine.wrap()` and `SSLEngine.unwrap()` methods return an `SSLEngineResult` instance, as shown in the example Using a Nonblocking SocketChannel. This `SSLEngineResult` object contains two pieces of status information: the overall status of the engine and the handshaking status.

The possible overall statuses are represented by the `SSLEngineResult.Status` enum. The following statuses are available:

- `OK`
  There was no error.
- `CLOSED`
  The operation closed the `SSLEngine` or the operation could not be completed because it was already closed.
- `BUFFER_UNDERFLOW`
  The input buffer had insufficient data, indicating that the application must obtain more data from the peer (for example, by reading more data from the network).
- `BUFFER_OVERFLOW`
  The output buffer had insufficient space to hold the result, indicating that the application must clear or enlarge the destination buffer.

The example Handling BUFFER_UNDERFLOW and BUFFER_OVERFLOW illustrates how to handle the `BUFFER_UNDERFLOW` and `BUFFER_OVERFLOW` statuses of the `SSLEngine.unwrap()` method. It uses `SSLSession.getApplicationBufferSize()` and `SSLSession.getPacketBufferSize()` to determine how large to make the byte buffers.

Handling BUFFER_UNDERFLOW and BUFFER_OVERFLOW

```
SSLEngineResult res = engine.unwrap(peerNetData, peerAppData);
switch (res.getStatus()) {

case BUFFER_OVERFLOW:
// Maybe need to enlarge the peer application data buffer.
if (engine.getSession().getApplicationBufferSize() > peerAppData.capacity()) {
// enlarge the peer application data buffer
} else {
// compact or clear the buffer
}
// retry the operation
break;

case BUFFER_UNDERFLOW:
// Maybe need to enlarge the peer network packet buffer
if (engine.getSession().getPacketBufferSize() > peerNetData.capacity()) {
// enlarge the peer network packet buffer
} else {
// compact or clear the buffer
}
// obtain more inbound network data and then retry the operation
break;

// Handle other status: CLOSED, OK
...
}
```

The possible handshaking statuses are represented by the `SSLEngineResult.HandshakeStatus` enum. They represent whether handshaking has completed, whether the caller must obtain more handshaking data from the peer or send more handshaking data to the peer, and so on. The following handshake statuses are available:

- `FINISHED`
  The `SSLEngine` has just finished handshaking.
- `NEED_TASK`
  The `SSLEngine` needs the results of one (or more) delegated tasks before handshaking can continue.
- `NEED_UNWRAP`
  The `SSLEngine` needs to receive data from the remote side before handshaking can continue.
- `NEED_UNWRAP_AGAIN`
  The `SSLEngine` needs to unwrap before handshaking can continue. This value indicates that not-yet-interpreted data has been previously received from the remote side and does not need to be received again; the data has been brought into the JSSE framework but has not been processed yet.
- `NEED_WRAP`
  The `SSLEngine` must send data to the remote side before handshaking can continue, so `SSLEngine.wrap()` should be called.
- `NOT_HANDSHAKING`
  The `SSLEngine` is not currently handshaking.

Having two statuses per result allows the `SSLEngine` to indicate that the application must take two actions: one in response to the handshaking and one representing the overall status of the `wrap()` and `unwrap()` methods. For example, the engine might, as the result of a single `SSLEngine.unwrap()` call, return `SSLEngineResult.Status.OK` to indicate that the input data was processed successfully and `SSLEngineResult.HandshakeStatus.NEED_UNWRAP` to indicate that the application should obtain more TLS encoded data from the peer and supply it to `SSLEngine.unwrap()` again so that handshaking can continue. As you can see, the previous examples were greatly simplified; they would need to be expanded significantly to properly handle all of these statuses.

The example Checking and Processing Handshaking Statuses and Overall Statuses illustrates how to process handshaking data by checking handshaking status and the overall status of the `wrap()` and `unwrap()` methods.

Checking and Processing Handshaking Statuses and Overall Statuses

```
void doHandshake(SocketChannel socketChannel, SSLEngine engine,
ByteBuffer myNetData, ByteBuffer peerNetData) throws Exception {

// Create byte buffers to use for holding application data
int appBufferSize = engine.getSession().getApplicationBufferSize();
ByteBuffer myAppData = ByteBuffer.allocate(appBufferSize);
ByteBuffer peerAppData = ByteBuffer.allocate(appBufferSize);

// Begin handshake
engine.beginHandshake();
SSLEngineResult.HandshakeStatus hs = engine.getHandshakeStatus();

// Process handshaking message
while (hs != SSLEngineResult.HandshakeStatus.FINISHED &&
hs != SSLEngineResult.HandshakeStatus.NOT_HANDSHAKING) {

switch (hs) {

case NEED_UNWRAP:
// Receive handshaking data from peer
if (socketChannel.read(peerNetData) < 0) {
// The channel has reached end-of-stream
}

// Process incoming handshaking data
peerNetData.flip();
SSLEngineResult res = engine.unwrap(peerNetData, peerAppData);
peerNetData.compact();
hs = res.getHandshakeStatus();

// Check status
switch (res.getStatus()) {
case OK :
// Handle OK status
break;

// Handle other status: BUFFER_UNDERFLOW, BUFFER_OVERFLOW, CLOSED
...
}
break;

case NEED_WRAP :
// Empty the local network packet buffer.
myNetData.clear();

// Generate handshaking data
res = engine.wrap(myAppData, myNetData);
hs = res.getHandshakeStatus();

// Check status
switch (res.getStatus()) {
case OK :
myNetData.flip();

// Send the handshaking data to peer
while (myNetData.hasRemaining()) {
socketChannel.write(myNetData);
}
break;

// Handle other status:  BUFFER_OVERFLOW, BUFFER_UNDERFLOW, CLOSED
...
}
break;

case NEED_TASK :
// Handle blocking tasks
break;

// Handle other status:  // FINISHED or NOT_HANDSHAKING
...
}
}

// Processes after handshaking
...
}
```

SSLEngine for TLS Protocols

This section shows you how to create an SSLEngine object and use it to generate and process TLS data.

Creating an SSLEngine Object

To create an `SSLEngine` object, you use the `SSLContext.createSSLEngine()` method. You must configure the engine to act as a client or a server, and set other configuration parameters, such as which cipher suites to use and whether to require client authentication. The `SSLContext.createSSLEngine` method creates a `javax.net.ssl.SSLEngine` object.

The example Creating an SSLEngine Client for TLS with JKS as Keystore illustrates how to create an `SSLEngine` client for TLS that uses JKS as keystore.

Note: The server name and port number are not used for communicating with the server (all transport is the responsibility of the application). They are hints to the JSSE provider to use for SSL session caching, and for Kerberos-based cipher suite implementations to determine which server credentials should be obtained.

Creating an SSLEngine Client for TLS with JKS as Keystore

```
import javax.net.ssl.*;
import java.security.*;

// Create and initialize the SSLContext with key material
char[] passphrase = "passphrase".toCharArray();

// First initialize the key and trust material
KeyStore ksKeys = KeyStore.getInstance("JKS");
ksKeys.load(new FileInputStream("testKeys"), passphrase);
KeyStore ksTrust = KeyStore.getInstance("JKS");
ksTrust.load(new FileInputStream("testTrust"), passphrase);

// KeyManagers decide which key material to use
KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
kmf.init(ksKeys, passphrase);

// TrustManagers decide whether to allow connections
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ksTrust);

// Get an instance of SSLContext for TLS protocols
sslContext = SSLContext.getInstance("TLS");
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// Create the engine
SSLEngine engine = sslContext.createSSLengine(hostname, port);

// Use as client
engine.setUseClientMode(true);
```

Generating and Processing TLS Data

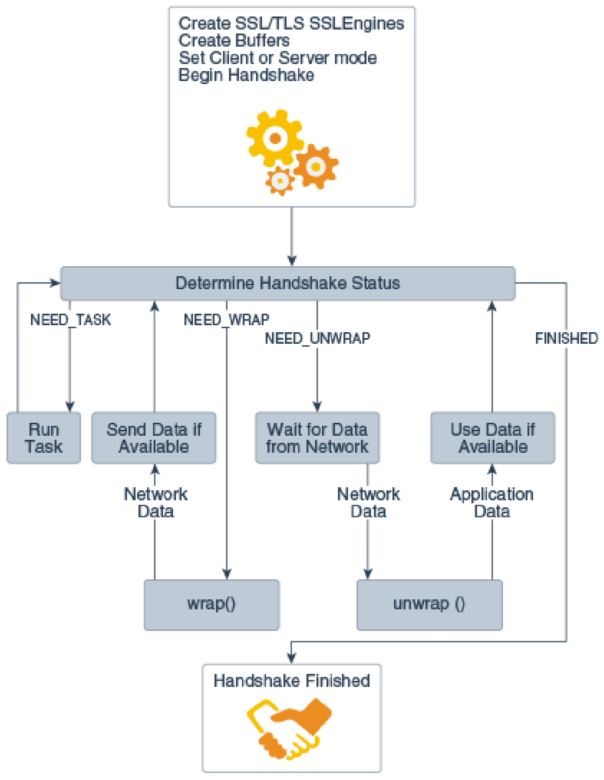
The two main `SSLEngine` methods are `wrap()` and `unwrap()`. They are responsible for generating and consuming network data respectively. Depending on the state of the `SSLEngine` object, this data might be handshake or application data.

Each `SSLEngine` object has several phases during its lifetime. Before application data can be sent or received, the SSL/TLS protocol requires a handshake to establish cryptographic parameters. This handshake requires a series of back-and-forth steps by the `SSLEngine` object.

During the initial handshaking, the `wrap()` and `unwrap()` methods generate and consume handshake data, and the application is responsible for transporting the data. The `wrap()` and `unwrap()` method sequence is repeated until the handshake is finished. Each `SSLEngine` operation generates an instance of the `SSLEngineResult` class, in which the `SSLEngineResult.HandshakeStatus` field is used to determine what operation must occur next to move the handshake along.

The figure State Machine during TLS Handshake shows the state machine during a typical TLS handshake, with corresponding messages and statuses:

State Machine during TLS Handshake



The following steps are performed before the status of the handshake is determined:

- Create SSL/TLS SSLEngines
- Create Buffers
- Set Client of Server mode
- Begin Handshake

This image illustrates some of the possible handshake statuses. The section Understanding SSLEngine Operation Statuses describes these statuses in more detail:

- `NEED_TASK`
- `NEED_WRAP`
- `NEED_UNWRAP`
- `FINISHED`

When handshaking is complete, further calls to `wrap()` will attempt to consume application data and package it for transport. The `unwrap()` method will attempt the opposite.

To send data to the peer, the application first supplies the data that it wants to send via `SSLEngine.wrap()` to obtain the corresponding SSL/TLS encoded data. The application then sends the encoded data to the peer using its chosen transport mechanism. When the application receives the SSL/TLS encoded data from the peer via the transport mechanism, it supplies this data to the `SSLEngine` via `SSLEngine.unwrap()` to obtain the plaintext data sent by the peer.

The example Using a Nonblocking SocketChannel shows an SSL application that uses a nonblocking `SocketChannel` to communicate with its peer.

Note: The example can be made more robust and scalable by using a `Selector` with the nonblocking `SocketChannel`.

In the example Using a Nonblocking SocketChannel, the string `hello` is sent to the peer by encoding it using the `SSLEngine` created in the example Creating an SSLEngine Client for TLS with JKS as Keystore. It uses information from the `SSLSession` to determine how large the byte buffers should be.

Using a Nonblocking SocketChannel

```
// Create a nonblocking socket channel
SocketChannel socketChannel = SocketChannel.open();
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress(hostname, port));

// Complete connection
while (!socketChannel.finishedConnect()) {
// do something until connect completed
}

// Create byte buffers to use for holding application and encoded data
SSLSession session = engine.getSession();
ByteBuffer myAppData = ByteBuffer.allocate(session.getApplicationBufferSize());
ByteBuffer myNetData = ByteBuffer.allocate(session.getPacketBufferSize());
ByteBuffer peerAppData = ByteBuffer.allocate(session.getApplicationBufferSize());
ByteBuffer peerNetData = ByteBuffer.allocate(session.getPacketBufferSize());

// Do initial handshake
doHandshake(socketChannel, engine, myNetData, peerNetData);

myAppData.put("hello".getBytes());
myAppData.flip();

while (myAppData.hasRemaining()) {
// Generate SSL/TLS encoded data (handshake or application data)
SSLEngineResult res = engine.wrap(myAppData, myNetData);

// Process status of call
if (res.getStatus() == SSLEngineResult.Status.OK) {
myAppData.compact();

// Send SSL/TLS encoded data to peer
while(myNetData.hasRemaining()) {
int num = socketChannel.write(myNetData);
if (num == 0) {
// no bytes written; try again later
}
}
}

// Handle other status:  BUFFER_OVERFLOW, CLOSED
...
}
```

The example Reading Data From Nonblocking SocketChannel illustrates how to read data from the same nonblocking `SocketChannel` and extract the plaintext data from it by using the `SSLEngine` created in the example Creating an SSLEngine Client for TLS with JKS as Keystore. Each iteration of this code may or may not produce plaintext data, depending on whether handshaking is in progress.

Reading Data From Nonblocking SocketChannel

```
// Read SSL/TLS encoded data from peer
int num = socketChannel.read(peerNetData);
if (num == -1) {
// The channel has reached end-of-stream
} else if (num == 0) {
// No bytes read; try again ...
} else {
// Process incoming data
peerNetData.flip();
res = engine.unwrap(peerNetData, peerAppData);

if (res.getStatus() == SSLEngineResult.Status.OK) {
peerNetData.compact();

if (peerAppData.hasRemaining()) {
// Use peerAppData
}
}
// Handle other status:  BUFFER_OVERFLOW, BUFFER_UNDERFLOW, CLOSED
...
}
```

Dealing With Blocking Tasks

During handshaking, an `SSLEngine` might encounter tasks that can block or take a long time. For example, a `TrustManager` may need to connect to a remote certificate validation service, or a `KeyManager` might need to prompt a user to determine which certificate to use as part of client authentication. To preserve the nonblocking nature of `SSLEngine`, when the engine encounters such a task, it will return `SSLEngineResult.HandshakeStatus.NEED_TASK`. Upon receiving this status, the application should invoke `SSLEngine.getDelegatedTask()` to get the task, and then, using the threading model appropriate for its requirements, process the task. The application might, for example, obtain threads from a thread pool to process the tasks, while the main thread handles other I/O.

The following code executes each task in a newly created thread:

```
if (res.getHandshakeStatus() == SSLEngineResult.HandshakeStatus.NEED_TASK) {
Runnable task;
while ((task = engine.getDelegatedTask()) != null) {
new Thread(task).start();
}
}
```

The `SSLEngine` will block future `wrap()` and `unwrap()` calls until all of the outstanding tasks are completed.

Shutting Down

For an orderly shutdown of an SSL/TLS connection, the SSL/TLS protocols require transmission of close messages. Therefore, when an application is done with the SSL/TLS connection, it should first obtain the close messages from the

`SSLEngine`, then transmit them to the peer using its transport mechanism, and finally shut down the transport mechanism. Example 6 illustrates this.

Example 6: Shutting Down an SSL/TLS Connection

```
// Indicate that application is done with engine
engine.closeOutbound();

while (!engine.isOutboundDone()) {
// Get close message
SSLEngineResult res = engine.wrap(empty, myNetData);

// Check res statuses

// Send close message to peer
while(myNetData.hasRemaining()) {
int num = socketChannel.write(myNetData);
if (num == 0) {
// no bytes written; try again later
}
myNetData().compact();
}
}

// Close transport
socketChannel.close();
```

In addition to an application explicitly closing the `SSLEngine`, the `SSLEngine` might be closed by the peer (via receipt of a close message while it is processing handshake data), or by the `SSLEngine` encountering an error while processing application or handshake data, indicated by throwing an `SSLException`. In such cases, the application should invoke `SSLEngine.wrap()` to get the close message and send it to the peer until `SSLEngine.isOutboundDone()` returns `true` (as shown in Example 6), or until the `SSLEngineResult.getStatus()` returns `CLOSED`.

In addition to orderly shutdowns, there can also be unexpected shutdowns when the transport link is severed before close messages are exchanged. In the previous examples, the application might get `-1` or `IOException` when trying to read from the nonblocking `SocketChannel`, or get `IOException` when trying to write to the non-blocking `SocketChannel`. When you get to the end of your input data, you should call `engine.closeInbound()`, which will verify with the `SSLEngine` that the remote peer has closed cleanly from the SSL/TLS perspective. Then the application should still try to shut down cleanly by using the procedure in Example 6. Obviously, unlike `SSLSocket`, the application using `SSLEngine` must deal with more state transitions, statuses, and programming. For more information about writing an `SSLEngine`-based application, see Sample Code Illustrating the Use of an SSLEngine.

SSLSession and ExtendedSSLSession

The `javax.net.ssl.SSLSession` interface represents a security context negotiated between the two peers of an `SSLSocket` or `SSLEngine` connection. After a session has been arranged, it can be shared by future `SSLSocket` or `SSLEngine` objects connected between the same two peers.

In some cases, parameters negotiated during the handshake are needed later in the handshake to make decisions about trust. For example, the list of valid signature algorithms might restrict the certificate types that can be used for authentication. The `SSLSession` can be retrieved during the handshake by calling `getHandshakeSession()` on an `SSLSocket` or `SSLEngine`. Implementations of `TrustManager` or `KeyManager` can use the `getHandshakeSession()` method to get information about session parameters to help them make decisions.

A fully initialized `SSLSession` contains the cipher suite that will be used for communications over a secure socket as well as a nonauthoritative hint as to the network address of the remote peer, and management information such as the time of creation and last use. A session also contains a shared master secret negotiated between the peers that is used to create cryptographic keys for encrypting and guaranteeing the integrity of the communications over an `SSLSocket` or `SSLEngine` connection. The value of this master secret is known only to the underlying secure socket implementation and is not exposed through the `SSLSession` API.

In Java SE, a TLS 1.2 session is represented by `ExtendedSSLSession`, an implementation of `SSLSession`. The `ExtendedSSLSession` class adds methods that describe the signature algorithms that are supported by the local implementation and the peer. The `getRequestedServerNames()` method called on an `ExtendedSSLSession` instance is used to obtain a list of `SNIServerName` objects in the requested Server Name Indication (SNI) extension. The server should use the requested server names to guide its selection of an appropriate authentication certificate, and/or other aspects of the security policy. The client should use the requested server names to guide its endpoint identification of the peer's identity, and/or other aspects of the security policy.

Calls to the `getPacketBufferSize()` and `getApplicationBufferSize()` methods on `SSLSession` are used to determine the appropriate buffer sizes used by `SSLEngine`.

Note: The SSL/TLS protocols specify that implementations are to produce packets containing at most 16 kilobytes (KB) of plain text. However, some implementations violate the specification and generate large records up to 32 KB. If the `SSLEngine.unwrap()` code detects large inbound packets, then the buffer sizes returned by `SSLSession` will be updated dynamically. Applications should always check the `BUFFER_OVERFLOW` and `BUFFER_UNDERFLOW` statuses and enlarge the corresponding buffers if necessary. `SunJSSE` will always send standard compliant 16 KB records and allow incoming 32 KB records. For a workaround, see the system property `jsse.SSLEngine.acceptLargeFragments` in Customizing JSSE.

HttpsURLConnection Class

The HTTPS protocol is similar to HTTP, but HTTPS first establishes a secure channel via SSL/TLS sockets and then verifies the identity of the peer before requesting or receiving data. The `javax.net.ssl.HttpsURLConnection` class extends the `java.net.HttpsURLConnection` class and adds support for HTTPS-specific features. For more information about how HTTPS URLs are constructed and used, see the API specification sections about the `java.net.URL`, `java.net.URLConnection`, `java.net.HttpURLConnection`, and `javax.net.ssl.HttpURLConnection` classes.

Upon obtaining an `HttpsURLConnection` instance, you can configure a number of HTTP and HTTPS parameters before actually initiating the network connection via the `URLConnection.connect()` method. Of particular interest are:

- Setting the Assigned SSLSocketFactory
- Setting the Assigned HostnameVerifier

Setting the Assigned SSLSocketFactory

In some situations, it is desirable to specify the `SSLSocketFactory` that an `HttpsURLConnection` instance uses. For example, you might want to tunnel through a proxy type that is not supported by the default implementation. The new `SSLSocketFactory` could return sockets that have already performed all necessary tunneling, thus allowing `HttpsURLConnection` to use additional proxies.

The `HttpsURLConnection` class has a default `SSLSocketFactory` that is assigned when the class is loaded (this is the factory returned by the `SSLSocketFactory.getDefault()` method). Future instances of `HttpsURLConnection` will inherit the current default `SSLSocketFactory` until a new default `SSLSocketFactory` is assigned to the class via the static `HttpsURLConnection.setDefaultSSLSocketFactory()` method. Once an instance of `HttpsURLConnection` has been created, the inherited `SSLSocketFactory` on this instance can be overridden with a call to the `setSSLSocketFactory()` method.

Note: Changing the default static `SSLSocketFactory` has no effect on existing instances of `HttpsURLConnection`. A call to the `setSSLSocketFactory()` method is necessary to change the existing instances.

You can obtain the per-instance or per-class `SSLSocketFactory` by making a call to the `getSSLSocketFactory()` or `getDefaultSSLSocketFactory()` method, respectively.

Setting the Assigned HostnameVerifier

If the host name of the URL does not match the host name in the credentials received as part of the SSL/TLS handshake, then it is possible that URL spoofing has occurred. If the implementation cannot determine a host name match with reasonable certainty, then the SSL implementation performs a callback to the instance's assigned `HostnameVerifier` for further checking. The host name verifier can take whatever steps are necessary to make the determination, such as performing host name pattern matching or perhaps opening an interactive dialog box. An unsuccessful verification by the host name verifier closes the connection. For more information regarding host name verification, see RFC 2818.

The `setHostnameVerifier()` and `setDefaultHostnameVerifier()` methods operate in a similar manner to the `setSSLSocketFactory()` and `setDefaultSSLSocketFactory()` methods, in that `HostnameVerifier` objects are assigned on a per-instance and per-class basis, and the current values can be obtained by a call to the `getHostnameVerifier()` or `getDefaultHostnameVerifier()` method.

Support Classes and Interfaces

The classes and interfaces in this section are provided to support the creation and initialization of `SSLContext` objects, which are used to create `SSLSocketFactory`, `SSLServerSocketFactory`, and `SSLEngine` objects. The support classes and interfaces are part of the `javax.net.ssl` package.

Three of the classes described in this section (`SSLContext`, `KeyManagerFactory`, and `TrustManagerFactory`) are engine classes. An engine class is an API class for specific algorithms (or protocols, in the case of `SSLContext`), for which implementations may be provided in one or more Cryptographic Service Provider (provider) packages. For more information about providers and engine classes, see the "Design Principles" and "Concepts" sections of the Java Cryptography Architecture Reference Guide.

The `SunJSSE` provider that comes standard with JSSE provides `SSLContext`, `KeyManagerFactory`, and `TrustManagerFactory` implementations, as well as implementations for engine classes in the standard `java.security` API. The following table lists implementations supplied by `SunJSSE`.

Implementations Supplied by SunJSEE

Engine Class Implemented	Algorithm or Protocol
`KeyStore`	PKCS12

Engine Class Implemented	Algorithm or Protocol
`KeyManagerFactory`	PKIX, SunX509
`TrustManagerFactory`	PKIX (X509 or SunPKIX), SunX509
`SSLContext`	SSLv3([1]), TLSv1, TLSv1.1, TLSv1.2, TLSv1.3 (since JDK 8u261)

The SSLContext Class

The `javax.net.ssl.SSLContext` class is an engine class for an implementation of a secure socket protocol. An instance of this class acts as a factory for SSL socket factories and SSL engines. An `SSLContext` object holds all of the state information shared across all objects created under that context. For example, session state is associated with the `SSLContext` when it is negotiated through the handshake protocol by sockets created by socket factories provided by the context. These cached sessions can be reused and shared by other sockets created under the same context.

Each instance is configured through its `init` method with the keys, certificate chains, and trusted root CA certificates that it needs to perform authentication. This configuration is provided in the form of key and trust managers. These managers provide support for the authentication and key agreement aspects of the cipher suites supported by the context.

Currently, only X.509-based managers are supported.

Obtaining and Initializing the SSLContext Class

There are two ways to obtain and initialize an `SSLContext`:

- The simplest way is to call the static `getDefault()` method on either the `SSLSocketFactory` or `SSLServerSocketFactory` class. This method creates a default `SSLContext` with a default `KeyManager`, `TrustManager`, and `SecureRandom` (a secure random number generator). A default `KeyManagerFactory` and `TrustManagerFactory` are used to create the `KeyManager` and `TrustManager`, respectively. The key material used is found in the default keystore and truststore, as determined by system properties described in Customizing the Default Keystores and Truststores, Store Types, and Store Passwords.
- The approach that gives the caller the most control over the behavior of the created context is to call the static method `getInstance()` on the `SSLContext` class, and then initialize the context by calling the instance's proper `init()` method. One variant of the `init()` method takes three arguments: an array of `KeyManager` objects, an array of `TrustManager` objects, and a `SecureRandom` object. The `KeyManager` and `TrustManager` objects are created by either implementing the appropriate interfaces or using the `KeyManagerFactory` and `TrustManagerFactory` classes to generate implementations. The `KeyManagerFactory` and `TrustManagerFactory` can then each be initialized with key material contained in the `KeyStore` passed as an argument to the `init()` method of the `TrustManagerFactory` or `KeyManagerFactory` classes. Finally, the `getTrustManagers()` method (in `TrustManagerFactory`) and `getKeyManagers()` method (in `KeyManagerFactory`) can be called to obtain the array of trust managers or key managers, one for each type of trust or key material.

Once an SSL connection is established, an `SSLSession` is created which contains various information, such as identities established and cipher suite used. The `SSLSession` is then used to describe an ongoing relationship and state information between two entities. Each SSL connection involves one session at a time, but that session may be used on many connections between those entities, simultaneously or sequentially.

Creating an SSLContext Object

Like other JCA provider-based engine classes, `SSLContext` objects are created using the `getInstance()` factory methods of the `SSLContext` class. These static methods each return an instance that implements at least the requested secure socket protocol. The returned instance may implement other protocols, too. For example, `getInstance("TLSv1.2")` may return an instance that implements TLSv1, TLSv1.1, and TLSv1.2. The `getSupportedProtocols()` method returns a list of supported protocols when an `SSLSocket`, `SSLServerSocket`, or `SSLEngine` is created from this context. You can control which protocols are actually enabled for an SSL connection by using the `setEnabledProtocols(String[] protocols)` method.

Note: An `SSLContext` object is automatically created, initialized, and statically assigned to the `SSLSocketFactory` class when you call the `SSLSocketFactory.getDefault()` method. Therefore, you do not have to directly create and initialize an `SSLContext` object (unless you want to override the default behavior).

To create an `SSLContext` object by calling the `getInstance()` factory method, you must specify the protocol name. You may also specify which provider you want to supply the implementation of the requested protocol:

- `public static SSLContext getInstance(String protocol);`
- `public static SSLContext getInstance(String protocol, String provider);`
- `public static SSLContext getInstance(String protocol, Provider provider);`

If just a protocol name is specified, then the system will determine whether an implementation of the requested protocol is available in the environment. If there is more than one implementation, then it will determine whether there is a preferred one.

If both a protocol name and a provider are specified, then the system will determine whether an implementation of the requested protocol is in the provider requested. If there is no implementation, an exception will be thrown.

A protocol is a string (such as "TLS") that describes the secure socket protocol desired. Common protocol names for SSLContext objects are defined in Appendix A.

An SSLContext can be obtained as follows:

```
SSLContext sc = SSLContext.getInstance("TLS");
```

A newly created SSLContext should be initialized by calling the init method:

```
public void init(KeyManager[] km, TrustManager[] tm, SecureRandom random);
```

If the KeyManager[] parameter is null, then an empty KeyManager will be defined for this context. If the TrustManager[] parameter is null, then the installed security providers will be searched for the highest-priority implementation of the TrustManagerFactory, from which an appropriate TrustManager will be obtained. Likewise, the SecureRandom parameter may be null, in which case a default implementation will be used.

If the internal default context is used, (for example, an SSLContext is created by SSLSocketFactory.getDefault() or SSLServerSocketFactory.getDefault()), then a default KeyManager and TrustManager are created. The default SecureRandom implementation is also chosen.

The TrustManager Interface

The primary responsibility of the TrustManager is to determine whether the presented authentication credentials should be trusted. If the credentials are not trusted, then the connection will be terminated. To authenticate the remote identity of a secure socket peer, you must initialize an SSLContext object with one or more TrustManager objects. You must pass one TrustManager for each authentication mechanism that is supported. If null is passed into the SSLContext initialization, then a trust manager will be created for you. Typically, a single trust manager supports authentication based on X.509 public key certificates (for example, X509TrustManager). Some secure socket implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

TrustManager objects are created either by a TrustManagerFactory, or by providing a concrete implementation of the interface.

The TrustManagerFactory Class

The javax.net.ssl.TrustManagerFactory is an engine class for a provider-based service that acts as a factory for one or more types of TrustManager objects. Because it is provider-based, additional factories can be implemented and configured to provide additional or alternative trust managers that provide more sophisticated services or that implement installation-specific authentication policies.

Creating a TrustManagerFactory

You create an instance of this class in a similar manner to SSLContext, except for passing an algorithm name string instead of a protocol name to the getInstance() method:

```
TrustManagerFactory tmf =
TrustManagerFactory.getInstance(String algorithm);

TrustManagerFactory tmf =
TrustManagerFactory.getInstance(String algorithm, String provider);

TrustManagerFactory tmf =
TrustManagerFactory.getInstance(String algorithm, Provider provider);
```

A sample call is as follows:

```
TrustManagerFactory tmf =
TrustManagerFactory.getInstance("PKIX", "SunJSSE");
```

The preceding call creates an instance of the SunJSSE provider's PKIX trust manager factory. This factory can be used to create trust managers that provide X.509 PKIX-based certification path validity checking.

When initializing an SSLContext, you can use trust managers created from a trust manager factory, or you can write your own trust manager, for example, using the CertPath API. For details, see the Java PKI Programmer's Guide. You do not need to use a trust manager factory if you implement a trust manager using the X509TrustManager interface.

A newly created factory should be initialized by calling one of the init() methods:

```
public void init(KeyStore ks);
public void init(ManagerFactoryParameters spec);
```

Call whichever init() method is appropriate for the TrustManagerFactory you are using. If you are not sure, then ask the provider vendor.

For many factories, such as the SunX509 TrustManagerFactory from the SunJSSE provider, the KeyStore is the only information required to initialize the TrustManagerFactory and thus the first init method is the appropriate one to call. The

`TrustManagerFactory` will query the `KeyStore` for information about which remote certificates should be trusted during authorization checks.

Sometimes, initialization parameters other than a `KeyStore` are needed by a provider. Users of that provider are expected to pass an implementation of the appropriate `ManagerFactoryParameters` as defined by the provider. The provider can then call the specified methods in the `ManagerFactoryParameters` implementation to obtain the needed information.

For example, suppose the `TrustManagerFactory` provider requires initialization parameters B, R, and S from any application that wants to use that provider. Like all providers that require initialization parameters other than a `KeyStore`, the provider requires the application to provide an instance of a class that implements a particular `ManagerFactoryParameters` subinterface. In the example, suppose that the provider requires the calling application to implement and create an instance of `MyTrustManagerFactoryParams` and pass it to the second `init()` method. The following example illustrates what `MyTrustManagerFactoryParams` can look like:

```
public interface MyTrustManagerFactoryParams extends ManagerFactoryParameters {
public boolean getBValue();
public float getRValue();
public String getSValue():
}
```

Some trust managers can make trust decisions without being explicitly initialized with a `KeyStore` object or any other parameters. For example, they may access trust material from a local directory service via LDAP, use a remote online certificate status checking server, or access default trust material from a standard local location.

PKIX TrustManager Support

The default trust manager algorithm is PKIX. It can be changed by editing the `ssl.TrustManagerFactory.algorithm` property in the `java.security` file.

The PKIX trust manager factory uses the CertPath PKIX implementation from an installed security provider. The trust manager factory can be initialized using the normal `init(KeyStore ks)` method, or by passing CertPath parameters to the the PKIX trust manager using the `javax.net.ssl.CertPathTrustManagerParameters` class.

The following example illustrates how to get the trust manager to use a particular LDAP certificate store and enable revocation checking:

```
import javax.net.ssl.*;
import java.security.cert.*;
import java.security.KeyStore;
import java.io.FileInputStream;
...

// Obtain Keystore password
char[] pass = System.console().readPassword("Password: ");

// Create PKIX parameters
KeyStore anchors = KeyStore.getInstance("PKCS12");
anchors.load(new FileInputStream(anchorsFile, pass));
PKIXBuilderParameters pkixParams = new PKIXBuilderParameters(anchors, new X509CertSelector());

// Specify LDAP certificate store to use
LDAPCertStoreParameters lcsp = new LDAPCertStoreParameters("ldap.imc.org", 389);
pkixParams.addCertStore(CertStore.getInstance("LDAP", lcsp));

// Specify that revocation checking is to be enabled
pkixParams.setRevocationEnabled(true);

// Wrap PKIX parameters as trust manager parameters
ManagerFactoryParameters trustParams = new CertPathTrustManagerParameters(pkixParams);

// Create TrustManagerFactory for PKIX-compliant trust managers
TrustManagerFactory factory = TrustManagerFactory.getInstance("PKIX");

// Pass parameters to factory to be passed to CertPath implementation
factory.init(trustParams);

// Use factory
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(null, factory.getTrustManagers(), null);
```

If the `init(KeyStore ks)` method is used, then default PKIX parameters are used with the exception that revocation checking is disabled. It can be enabled by setting the `com.sun.net.ssl.checkRevocation` system property to `true`. This setting requires that the CertPath implementation can locate revocation information by itself. The PKIX implementation in the provider can do this in many cases but requires that the system property `com.sun.security.enableCRLDP` be set to `true`.

For more information about PKIX and the CertPath API, see the Java PKI Programmer's Guide.

The X509TrustManager Interface

The `javax.net.ssl.X509TrustManager` interface extends the general `TrustManager` interface. It must be implemented by a trust manager when using X.509-based authentication.

To support X.509 authentication of remote socket peers through JSSE, an instance of this interface must be passed to the `init` method of an `SSLContext` object.

Creating an X509TrustManager

You can either implement this interface directly yourself or obtain one from a provider-based `TrustManagerFactory` (such as that supplied by the `SunJSSE` provider). You could also implement your own interface that delegates to a factory-generated trust manager. For example, you might do this to filter the resulting trust decisions and query an end-user through a graphical user interface.

Note: If a null KeyStore parameter is passed to the `SunJSSE` PKIX or SunX509 `TrustManagerFactory`, then the factory uses the following process to try to find trust material:

---

1. If the `javax.net.ssl.trustStore` property is defined, then the `TrustManagerFactory` attempts to find a file using the file name specified by that system property, and uses that file for the KeyStore parameter. If the `javax.net.ssl.trustStorePassword` system property is also defined, then its value is used to check the integrity of the data in the truststore before opening it.

   If the `javax.net.ssl.trustStore` property is defined but the specified file does not exist, then a default `TrustManager` using an empty keystore is created.

2. If the `javax.net.ssl.trustStore` system property was not specified, then:
   - if the file `java-home/lib/security/jssecacerts` exists, that file is used;
   - if the file `java-home/lib/security/cacerts` exists, that file is used;
   - if neither of these files exists, then the SSL cipher suite is anonymous, does not perform any authentication, and thus does not need a truststore.

The factory looks for a file specified via the `javax.net.ssl.trustStore` Security Property or for the `jssecacerts` file before checking for a `cacerts` file. Therefore, you can provide a JSSE-specific set of trusted root certificates separate from ones that might be present in cacerts for code-signing purposes.

Creating Your Own X509TrustManager

If the supplied `X509TrustManager` behavior is not suitable for your situation, then you can create your own `X509TrustManager` by either creating and registering your own `TrustManagerFactory` or by implementing the `X509TrustManager` interface directly.

The following example illustrates a `MyX509TrustManager` class that enhances the default `SunJSSE X509TrustManager` behavior by providing alternative authentication logic when the default `X509TrustManager` fails:

```
class MyX509TrustManager implements X509TrustManager {

/*
* The default PKIX X509TrustManager9.  Decisions are delegated
* to it, and a fall back to the logic in this class is performed
* if the default X509TrustManager does not trust it.
*/
X509TrustManager pkixTrustManager;

MyX509TrustManager() throws Exception {
// create a "default" JSSE X509TrustManager.

KeyStore ks = KeyStore.getInstance("PKCS12");
ks.load(new FileInputStream("trustedCerts"), "passphrase".toCharArray());

TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ks);

TrustManager tms [] = tmf.getTrustManagers();

/*
* Iterate over the returned trust managers, looking
* for an instance of X509TrustManager.  If found,
* use that as the default trust manager.
*/
for (int i = 0; i < tms.length; i++) {
if (tms[i] instanceof X509TrustManager) {
pkixTrustManager = (X509TrustManager) tms[i];
return;
}
}

/*
* Find some other way to initialize, or else the
* constructor fails.
*/
throw new Exception("Couldn't initialize");
}

/*
* Delegate to the default trust manager.
*/
public void checkClientTrusted(X509Certificate[] chain, String authType)
throws CertificateException {
try {
pkixTrustManager.checkClientTrusted(chain, authType);
} catch (CertificateException excep) {
// do any special handling here, or rethrow exception.
}
}

/*
* Delegate to the default trust manager.
*/
public void checkServerTrusted(X509Certificate[] chain, String authType)
throws CertificateException {
try {
pkixTrustManager.checkServerTrusted(chain, authType);
} catch (CertificateException excep) {
/*
* Possibly pop up a dialog box asking whether to trust the
* cert chain.
*/
}
}

/*
* Merely pass this through.
*/
public X509Certificate[] getAcceptedIssuers() {
return pkixTrustManager.getAcceptedIssuers();
}
}
```

Once you have created such a trust manager, assign it to an `SSLContext` via the `init()` method, as in the following example. Future `SocketFactories` created from this `SSLContext` will use your new `TrustManager` when making trust decisions.

```
TrustManager[] myTMs = new TrustManager[] { new MyX509TrustManager() };
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(null, myTMs, null);
```

Updating the Keystore Dynamically

You can enhance `MyX509TrustManager` to handle dynamic keystore updates. When a `checkClientTrusted` or `checkServerTrusted` test fails and does not establish a trusted certificate chain, you can add the required trusted certificate to the keystore. You must create a new `pkixTrustManager` from the `TrustManagerFactory` initialized with the updated keystore. When you establish a new connection (using the previously initialized `SSLContext`), the newly added certificate will be used when making trust decisions.

X509ExtendedTrustManager Class

The `X509ExtendedTrustManager` class is an abstract implementation of the `X509TrustManager` interface. It adds methods for connection-sensitive trust management. In addition, it enables endpoint verification at the TLS layer.

In TLS 1.2 and later, both client and server can specify which hash and signature algorithms they will accept. To authenticate the remote side, authentication decisions must be based on both X509 certificates and the local accepted hash and signature algorithms. The local accepted hash and signature algorithms can be obtained using the `ExtendedSSLSession.getLocalSupportedSignatureAlgorithms()` method.

The `ExtendedSSLSession` object can be retrieved by calling the `SSLSocket.getHandshakeSession()` method or the `SSLEngine.getHandshakeSession()` method.

The `X509TrustManager` interface is not connection-sensitive. It provides no way to access `SSLSocket` or `SSLEngine` session properties.

Besides TLS 1.2 support, the `X509ExtendedTrustManager` class also supports algorithm constraints and SSL layer host name verification. For JSSE providers and trust manager implementations, the `X509ExtendedTrustManager` class is highly recommended over the legacy `X509TrustManager` interface.

Creating an X509ExtendedTrustManager

You can either create an `X509ExtendedTrustManager` subclass yourself (which is outlined in the following section) or obtain one from a provider-based `TrustManagerFactory` (such as that supplied by the `SunJSSE` provider). In Java SE 7, the PKIX or SunX509 `TrustManagerFactory` returns an `X509ExtendedTrustManager` instance.

Creating Your Own X509ExtendedTrustManager

This section outlines how to create a subclass of `X509ExtendedTrustManager` in nearly the same way as described for `X509TrustManager`.

The following example illustrates how to create a class that uses the PKIX `TrustManagerFactory` to locate a default `X509ExtendedTrustManager` that will be used to make decisions about trust. If the default trust manager fails for any reason, then the subclass is can add other behavior. In the example, these locations are indicated by comments in the `catch` clauses.

```
import java.io.*;
import java.net.*;

import java.security.*;
import java.security.cert.*;
import javax.net.ssl.*;

public class MyX509ExtendedTrustManager extends X509ExtendedTrustManager {

/*
* The default PKIX X509ExtendedTrustManager.  Decisions are
* delegated to it, and a fall back to the logic in this class is
* performed if the default X509ExtendedTrustManager does not
* trust it.
*/
X509ExtendedTrustManager pkixTrustManager;

MyX509ExtendedTrustManager() throws Exception {
// create a "default" JSSE X509ExtendedTrustManager.

KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream("trustedCerts"), "passphrase".toCharArray());

TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");
tmf.init(ks);

TrustManager tms [] = tmf.getTrustManagers();

/*
* Iterate over the returned trust managers, looking
* for an instance of X509ExtendedTrustManager. If found,
* use that as the default trust manager.
*/
for (int i = 0; i < tms.length; i++) {
if (tms[i] instanceof X509ExtendedTrustManager) {
pkixTrustManager = (X509ExtendedTrustManager) tms[i];
return;
}
}

/*
* Find some other way to initialize, or else we have to fail the
* constructor.
*/
throw new Exception("Couldn't initialize");
}

/*
* Delegate to the default trust manager.
*/
public void checkClientTrusted(X509Certificate[] chain, String authType)
throws CertificateException {
try {
pkixTrustManager.checkClientTrusted(chain, authType);
} catch (CertificateException excep) {
// do any special handling here, or rethrow exception.
}
}

/*
* Delegate to the default trust manager.
*/
public void checkServerTrusted(X509Certificate[] chain, String authType)
throws CertificateException {
try {
pkixTrustManager.checkServerTrusted(chain, authType);
} catch (CertificateException excep) {
/*
* Possibly pop up a dialog box asking whether to trust the
* cert chain.
*/
}
}

/*
* Connection-sensitive verification.
*/
public void checkClientTrusted(X509Certificate[] chain, String authType, Socket socket)
throws CertificateException {
try {
pkixTrustManager.checkClientTrusted(chain, authType, socket);
} catch (CertificateException excep) {
// do any special handling here, or rethrow exception.
}
}

public void checkClientTrusted(X509Certificate[] chain, String authType, SSLEngine engine)
throws CertificateException {
try {
pkixTrustManager.checkClientTrusted(chain, authType, engine);
} catch (CertificateException excep) {
// do any special handling here, or rethrow exception.
}
}

public void checkServerTrusted(X509Certificate[] chain, String authType, Socket socket)
throws CertificateException {
try {
pkixTrustManager.checkServerTrusted(chain, authType, socket);
} catch (CertificateException excep) {
// do any special handling here, or rethrow exception.
}
}

public void checkServerTrusted(X509Certificate[] chain, String authType, SSLEngine engine)
throws CertificateException {
try {
pkixTrustManager.checkServerTrusted(chain, authType, engine);
} catch (CertificateException excep) {
// do any special handling here, or rethrow exception.
}
}

/*
* Merely pass this through.
*/
public X509Certificate[] getAcceptedIssuers() {
return pkixTrustManager.getAcceptedIssuers();
}
}
```

The KeyManager Interface

The primary responsibility of the `KeyManager` is to select the authentication credentials that will eventually be sent to the remote host. To authenticate yourself (a local secure socket peer) to a remote peer, you must initialize an `SSLContext` object with one or more `KeyManager` objects. You must pass one `KeyManager` for each different authentication mechanism that will be supported. If null is passed into the `SSLContext` initialization, then an empty `KeyManager` will be created. If the internal default context is used (for example, an `SSLContext` created by `SSLSocketFactory.getDefault()` or `SSLServerSocketFactory.getDefault()`), then a default `KeyManager` is created. Typically, a single key manager supports authentication based on X.509 public key certificates. Some secure socket implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

`KeyManager` objects are created either by a `KeyManagerFactory`, or by providing a concrete implementation of the interface.

The KeyManagerFactory Class

The `javax.net.ssl.KeyManagerFactory` class is an engine class for a provider-based service that acts as a factory for one or more types of `KeyManager` objects. The `SunJSSE` provider implements a factory that can return a basic X.509 key manager. Because it is provider-based, additional factories can be implemented and configured to provide additional or alternative key managers.

Creating a KeyManagerFactory

You create an instance of this class in a similar manner to `SSLContext`, except for passing an algorithm name string instead of a protocol name to the `getInstance()` method:

```
KeyManagerFactory kmf = getInstance(String algorithm);

KeyManagerFactory kmf = getInstance(String algorithm, String provider);

KeyManagerFactory kmf = getInstance(String algorithm, Provider provider);
```

A sample call as follows:

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509", "SunJSSE");
```

The preceding call creates an instance of the `SunJSSE` provider's default key manager factory, which provides basic X.509-based authentication keys.

A newly created factory should be initialized by calling one of the `init` methods:

```
public void init(KeyStore ks, char[] password);
public void init(ManagerFactoryParameters spec);
```

Call whichever `init` method is appropriate for the `KeyManagerFactory` you are using. If you are not sure, then ask the provider vendor.

For many factories, such as the default SunX509 `KeyManagerFactory` from the `SunJSSE` provider, the `KeyStore` and password are the only information required to initialize the `KeyManagerFactory` and thus the first `init` method is the appropriate one to call. The `KeyManagerFactory` will query the `KeyStore` for information about which private key and matching public key certificates should be used for authenticating to a remote socket peer. The password parameter specifies the password that will be used with the methods for accessing keys from the `KeyStore`. All keys in the `KeyStore` must be protected by the same password.

Sometimes initialization parameters other than a `KeyStore` and password are needed by a provider. Users of that provider are expected to pass an implementation of the appropriate `ManagerFactoryParameters` as defined by the provider. The provider can then call the specified methods in the `ManagerFactoryParameters` implementation to obtain the needed information.

Some factories can provide access to authentication material without being initialized with a `KeyStore` object or any other parameters. For example, they may access key material as part of a login mechanism such as one based on JAAS, the Java Authentication and Authorization Service.

As previously indicated, the `SunJSSE` provider supports a SunX509 factory that must be initialized with a `KeyStore` parameter.

The X509KeyManager Interface

The `javax.net.ssl.X509KeyManager` interface extends the general `KeyManager` interface. It must be implemented by a key manager for X.509-based authentication. To support X.509 authentication to remote socket peers through JSSE, an instance of this interface must be passed to the `init()` method of an `SSLContext` object.

Creating an X509KeyManager

You can either implement this interface directly yourself or obtain one from a provider-based `KeyManagerFactory` (such as that supplied by the `SunJSSE` provider). You could also implement your own interface that delegates to a factory-generated key manager. For example, you might do this to filter the resulting keys and query an end-user through a graphical user interface.

Creating Your Own X509KeyManager

If the default `X509KeyManager` behavior is not suitable for your situation, then you can create your own `X509KeyManager` in a way similar to that shown in Creating Your Own X509TrustManager.

The X509ExtendedKeyManager Class

The `X509ExtendedKeyManager` abstract class is an implementation of the `X509KeyManager` interface that allows for connection-specific key selection. It adds two methods that select a key alias for client or server based on the key type, allowed issuers, and current `SSLEngine`:

- `public String chooseEngineClientAlias(String[] keyType, Principal[] issuers, SSLEngine engine)`
- `public String chooseEngineServerAlias(String keyType, Principal[] issuers, SSLEngine engine)`

If a key manager is not an instance of the `X509ExtendedKeyManager` class, then it will not work with the `SSLEngine` class.

For JSSE providers and key manager implementations, the `X509ExtendedKeyManager` class is highly recommended over the legacy `X509KeyManager` interface.

In TLS 1.2 and later, both client and server can specify which hash and signature algorithms they will accept. To pass the authentication required by the remote side, local key selection decisions must be based on both X509 certificates and the remote accepted hash and signature algorithms. The remote accepted hash and signature algorithms can be retrieved using the `ExtendedSSLSession.getPeerSupportedSignatureAlgorithms()` method.

You can create your own `X509ExtendedKeyManager` subclass in a way similar to that shown in Creating Your Own X509ExtendedTrustManager.

Support for the Server Name Indication (SNI) extension on the server side enables the key manager to check the server name and select the appropriate key accordingly. For example, suppose there are three key entries with certificates in the keystore:

- `cn=www.example.com`
- `cn=www.example.org`
- `cn=www.example.net`

If the ClientHello message requests to connect to `www.example.net` in the SNI extension, then the server should be able to select the certificate with subject `cn=www.example.net`.

Relationship Between a TrustManager and a KeyManager

Historically, there has been confusion regarding the functionality of a `TrustManager` and a `KeyManager`.

A `TrustManager` determines whether the remote authentication credentials (and thus the connection) should be trusted.

A `KeyManager` determines which authentication credentials to send to the remote host.

Secondary Support Classes and Interfaces

These classes are provided as part of the JSSE API to support the creation, use, and management of secure sockets. They are less likely to be used by secure socket applications than are the core and support classes. The secondary support classes and interfaces are part of the `javax.net.ssl` and `javax.security.cert` packages.

The SSLParameters Class

The `SSLParameters` class encapsulates the following parameters that affect a TLS connection:

- The list of cipher suites to be accepted in an SSL/TLS handshake
- The list of protocols to be allowed
- The endpoint identification algorithm during SSL/TLS handshaking
- The server names and server name matchers (see the Server Name Indication (SNI) Extension)
- The algorithm constraints
- Whether SSL/TLS servers should request or require client authentication
- The cipher suite preference to be used in an SSL/TLS handshake

You can retrieve the current `SSLParameters` for an `SSLSocket` or `SSLEngine` by using the following methods:

- `getSSLParameters()` in an `SSLSocket`, `SSLServerSocket`, and `SSLEngine`
- `getDefaultSSLParameters()` and `getSupportedSSLParamters()` in an `SSLContext`

You can assign `SSLParameters` with the `setSSLParameters()` method in an `SSLSocket`, `SSLServerSocket` and `SSLEngine`.

You can explicitly set the server name indication with the `SSLParameters.setServerNames()` method. The server name indication in client mode also affects endpoint identification. In the implementation of `X509ExtendedTrustManager`, it uses the server name indication retrieved by the `ExtendedSSLSession.getRequestedServerNames()` method. The following example illustrates this functionality:

```
SSLSocketFactory factory = ...
SSLSocket sslSocket = factory.createSocket("172.16.10.6", 443);
// SSLEngine sslEngine = sslContext.createSSLEngine("172.16.10.6", 443);

SNIHostName serverName = new SNIHostName("www.example.com");
List<SNIServerName> serverNames = new ArrayList<>(1);
serverNames.add(serverName);

SSLParameters params = sslSocket.getSSLParameters();
params.setServerNames(serverNames);
sslSocket.setSSLParameters(params);
// sslEngine.setSSLParameters(params);
```

In the preceding example, the host name in the server name indication (`www.example.com`) will be used to make endpoint identification against the peer's identity presented in the end-entity's X.509 certificate.

Cipher Suite Preference

During TLS handshaking, the client requests to negotiate a cipher suite from a list of cryptographic options that it supports, starting with its first preference. Then, the server selects a single cipher suite from the list of cipher suites requested by the client. Normally, the selection honors the client's preference. However, to mitigate the risks of using weak cipher suites, the server may select cipher suites based on its own preference rather than the client's preference, by invoking the method `SSLParameters.setUseCipherSuitesOrder(true)`.

The SSLSessionContext Interface

The `javax.net.ssl.SSLSessionContext` interface is a grouping of `SSLSession` objects associated with a single entity. For example, it could be associated with a server or client that participates in many sessions concurrently. The methods in this interface enable the enumeration of all sessions in a context and allow lookup of specific sessions via their session IDs.

An `SSLSessionContext` may optionally be obtained from an `SSLSession` by calling the SSLSession `getSessionContext()` method. The context may be unavailable in some environments, in which case the `getSessionContext()` method returns null.

The SSLSessionBindingListener Interface

The `javax.net.ssl.SSLSessionBindingListener` interface is implemented by objects that are notified when they are being bound or unbound from an `SSLSession`.

The SSLSessionBindingEvent Class

The `javax.net.ssl.SSLSessionBindingEvent` class defines the event communicated to an `SSLSessionBindingListener` when it is bound or unbound from an `SSLSession`.

The HandShakeCompletedListener Interface

The `javax.net.ssl.HandShakeCompletedListener` interface is an interface implemented by any class that is notified of the completion of an SSL protocol handshake on a given `SSLSocket` connection.

The HandShakeCompletedEvent Class

The `javax.net.ssl.HandShakeCompletedEvent` class define the event communicated to a [`HandShakeCompletedListener`]() upon completion of an SSL protocol handshake on a given `SSLSocket` connection.

The HostnameVerifier Interface

If the SSL/TLS implementation's standard host name verification logic fails, then the implementation calls the `verify()` method of the class that implements this interface and is assigned to this `HttpsURLConnection` instance. If the callback class can determine that the host name is acceptable given the parameters, it reports that the connection should be allowed. An unacceptable response causes the connection to be terminated.

For example:

```
public class MyHostnameVerifier implements HostnameVerifier {

public boolean verify(String hostname, SSLSession session) {
// pop up an interactive dialog box
// or insert additional matching logic
if (good_address) {
return true;
} else {
return false;
}
}
}

//...deleted...

HttpsURLConnection urlc = (HttpsURLConnection)
(new URL("https://www.example.com/")).openConnection();
urlc.setHostnameVerifier(new MyHostnameVerifier());
```

See [The HttpsURLConnection Class]() for more information about how to assign the `HostnameVerifier` to the `HttpsURLConnection`.

The X509Certificate Class

Many secure socket protocols perform authentication using public key certificates, also called X.509 certificates. This is the default authentication mechanism for the SSL/TLS protocols.

The `java.security.cert.X509Certificate` abstract class provides a standard way to access the attributes of X.509 certificates.

Note: The `javax.security.cert.X509Certificate` class is supported only for backward compatibility with previous (1.0.x and 1.1.x) versions of JSSE. New applications should use the `java.security.cert.X509Certificate` class instead.

The AlgorithmConstraints Interface

The `java.security.AlgorithmConstraints` interface is used for controlling allowed cryptographic algorithms. `AlgorithmConstraints` defines three `permits()` methods. These methods tell whether an algorithm name or a key is permitted for certain cryptographic functions. Cryptographic functions are represented by a set of `CryptoPrimitive`, which is an enumeration containing fields like `STREAM_CIPHER`, `MESSAGE_DIGEST`, and `SIGNATURE`.

Thus, an `AlgorithmConstraints` implementation can answer questions like: Can I use this key with this algorithm for the purpose of a cryptographic operation?

An `AlgorithmConstraints` object can be associated with an `SSLParameters` object by using the new `setAlgorithmConstraints()` method. The current `AlgorithmConstraints` object for an `SSLParameters` object is retrieved using the `getAlgorithmConstraints()` method.

The StandardConstants Class

The `StandardConstants` class is used to represent standard constants definitions in JSSE.

`StandardConstants.SNI_HOST_NAME` represents a domain name server (DNS) host name in a [Server Name Indication (SNI)]() extension, which can be used when instantiating an `SNIServerName` or `SNIMatcher` object.

The SNIServerName Class

An instance of the abstract `SNIServerName` class represents a server name in the [Server Name Indication (SNI)]() extension. It is instantiated using the type and encoded value of the specified server name.

You can use the `getType()` and `getEncoded()` methods to return the server name type and a copy of the encoded server name value, respectively. The `equals()` method can be used to check if some other object is "equal" to this server name. The `hashCode()` method returns a hash code value for this server name. To get a string representation of the server name (including the server name type and encoded server name value), use the `toString()` method.

The SNIMatcher Class

An instance of the abstract `SNIMatcher` class performs match operations on an `SNIServerName` object. Servers can use information from the Server Name Indication (SNI) extension to decide if a specific `SSLSocket` or `SSLEngine` should accept a connection. For example, when multiple "virtual" or "name-based" servers are hosted on a single underlying network address, the server application can use SNI information to determine whether this server is the exact server that the client wants to access. Instances of this class can be used by a server to verify the acceptable server names of a particular type, such as host names.

The `SNIMatcher` class is instantiated using the specified server name type on which match operations will be performed. To match a given `SNIServerName`, use the `matches()` method. To return the server name type of the given `SNIMatcher` object, use the `getType()` method.

The SNIHostName Class

An instance of the `SNIHostName` class (which extends the `SNIServerName` class) represents a server name of type "host_name" (see The StandardConstants Class) in the Server Name Indication (SNI) extension. To instantiate an `SNIHostName`, specify the fully qualified DNS host name of the server (as understood by the client) as a `String` argument. The argument is illegal in the following cases:

- The argument is empty.
- The argument ends with a trailing period.
- The argument is not a valid Internationalized Domain Name (IDN) compliant with the RFC 3490 specification.

You can also instantiate an `SNIHostName` by specifying the encoded host name value as a byte array. This method is typically used to parse the encoded name value in a requested SNI extension. Otherwise, use the `SNIHostName(String hostname)` constructor. The `encoded` argument is illegal in the following cases:

- The argument is empty.
- The argument ends with a trailing period.
- The argument is not a valid Internationalized Domain Name (IDN) compliant with the RFC 3490 specification.
- The argument is not encoded in UTF-8 or US-ASCII.

Note: The `encoded` byte array passed in as an argument is cloned to protect against subsequent modification.

To return the host name of an `SNIHostName` object in US-ASCII encoding, use the `getAsciiName()` method. To compare a server name to another object, use the `equals()` method (comparison is not case-sensitive). To return a hash code value of an `SNIHostName`, use the `hashCode()` method. To return a string representation of an `SNIHostName`, including the DNS host name, use the `toString()` method.

You can create an `SNIMatcher` object for an `SNIHostName` object by passing a regular expression representing one or more host names to match to the `createSNIMatcher()` method.

Customizing JSSE

JSSE includes a standard implementation that can be customized by plugging in different implementations or specifying the default keystore, and so on. The following tables summarize which aspects can be customized, what the defaults are, and which mechanisms are used to provide customization.

Some of the customizations are done by setting Security Property or system property values. Sections following the table explain how to set such property values.

The following table shows items that are customized by setting a `java.security.Security` property:

Security Property	Customized Item	Default Value	Notes
`cert.provider.x509v1`	Customizing the X509Certificate Implementation	X509Certificate implementation from Oracle	None
JCE encryption algorithms used by the `SunJSSE` provider	Give alternative JCE algorithm providers a higher preference order than the SunJCE provider; see Customizing the Encryption Algorithm Providers.	SunJCE implementations	None
`jdk.certpath.disabledAlgorithms`[1]	Disabled certificate verification cryptographic algorithm (see Disabled and Restricted Cryptographic Algorithms)	MD2, MD5, SHA1 jdkCA & usage TLSServer, RSA keySize < 1024, DSA keySize < 1024, EC keySize < 224, include jdk.disabled.namedCurves[2]	None

`jdk.tls.disabledAlgorithms`[1]	Disabled and Restricted Cryptographic Algorithms	SSLv3, TLSv1, TLSv1.1, RC4, DES, MD5withRSA, DH keySize < 1024, EC keySize < 224, 3DES_EDE_CBC, anon, NULL, include jdk.disabled.namedCurves[2]	Disables specific algorithms (protocols versions, cipher suites, key exchange mechanisms, etc.) that will not be negotiated for TLS connections, even if they are enabled explicitly in an application
`jdk.tls.keyLimits`[1] (since JDK 8u261)	Limiting Amount of Data Algorithms May Encrypt with a Set of Keys	AES/GCM/NoPadding KeyUpdate 2^37	Limits the amount of data an algorithm may encrypt with a specific set of keys; once this limit is reached, a KeyUpdate post-handshake message is sent, which requests that the current set of keys be updated.
`jdk.tls.legacyAlgorithms`[1]	Legacy Cryptographic Algorithms	K_NULL, C_NULL, M_NULL, DH_anon, ECDH_anon, RC4_128, RC4_40, DES_CBC, DES40_CBC, 3DES_EDE_CBC[2]	Specifies which algorithms are considered legacy algorithms, which are not negotiated during TLS security parameters negotiation unless there are no other candidates.
`jdk.tls.maxCertificateChainLength`[1]	Certificate chain handling	10	Specifies the maximum allowed length of the certificate chain in TLS/DTLS handshaking.
`jdk.tls.maxHandshakeMessageSize`[1]	Certificate chain handling	32768 (32 kilobytes)	Specifies the maximum allowed size, in bytes, for the handshake message in TLS/DTLS handshaking.
`jdk.tls.server.defaultDHEParameters`[1]	Diffie-Hellman groups	Safe prime Diffie-Hellman groups in JDK TLS implementation	Defines default finite field Diffie-Hellman ephemeral (DHE) parameters for Transport Layer Security (TLS) processing
`ocsp.enable`[1]	Client-Driven OCSP and OCSP Stapling	false	Enables client-driven Online Certificate Status Protocol (OCSP). You must also enable revocation checking; see Setting up a Java Client to use Client-Driven OCSP.
`security.provider.n`	Cryptographic service provider; see Customizing the Provider Implementation and Customizing the Encryption Algorithm Providers	Differs per platform; check the `java.security` Security Properties file.	Specify the provider in the `security.provider.n=` line in the Security Properties file, where `n` is an integer whose value is equal or greater than 1.
`ssl.KeyManagerFactory.algorithm`	Default key manager factory algorithm name (see Customizing the Default Key Managers and Trust Managers)	SunX509	None
`ssl.ServerSocketFactory.provider`[1]	Default `SSLServerSocketFactory` implementation	`SSLServerSocketFactory` implementation from Oracle	None
`ssl.SocketFactory.provider`[1]	Default `SSLSocketFactory` implementation	`SSLSocketFactory` implementation from Oracle	None
`ssl.TrustManagerFactory.algorithm`	Default trust manager factory algorithm name (see Customizing the Default Key Managers and Trust Managers)	PKIX	None

[1]This Security Property is currently used by the JSSE implementation, but it is not guaranteed to be examined and used by other implementations. If it is examined by another implementation, then that implementation should handle it in the same manner as the JSSE implementation does. There is no guarantee the property will continue to exist or be of the same type (system or Security) in future releases.

[2]The list of restricted, disabled, and legacy algorithms specified in these Security Properties may change; see the `java.security` file in your JDK installation for the latest values.

The following table shows items that are customized by setting a `java.lang.System` property.

System Property	Customized Item	Default	Notes
`com.sun.net.ssl.checkRevocation`[1]	Revocation checking	false	You must enable revocation checking to enable client-driven OCSP; see Client-Driven OCSP and OCSP Stapling.
Customize via port field in the HTTPS URL*	Default HTTPS port	443	None
`https.cipherSuites`[1]	Default cipher suites for HTTPS connections[1]	Determined by the socket factory.	This contains a comma-separated list of cipher suite names specifying which cipher suites to enable for use on this `HttpsURLConnection`. See the `SSLSocket.setEnabledCipherSuites(String[])` method. Note that this method sets the preference order of the ClientHello cipher suites directly from the String array passed to it.
`https.protocols`[1]	Default handshaking protocols for HTTPS connections. See also Enabling TLS 1.3	Determined by the socket factory.	This contains a comma-separated list of protocol suite names specifying which protocol suites to enable on this `HttpsURLConnection`. See `SSLSocket.setEnabledProtocols(String[])`.
`https.proxyHost`[1]	Default proxy host	None	None
`https.proxyPort`[1]	Default proxy port	80	None
`java.protocol.handler.pkgs`	Specifying an Alternative HTTPS Protocol Implementation	Implementation from Oracle	None
`javax.net.ssl.keyStore`[1]	Default keystore; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`NONE`	The value `NONE` may be specified. This setting is appropriate if the keystore is not file-based (for example, it resides in a hardware token)
`javax.net.ssl.keyStorePassword`[1]	Default keystore password; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	It is inadvisable to specify the password in a way that exposes it to discovery by other users. For example, specifying the password on the command line. To keep the password secure, have the application prompt for the password, or specify the password in a properly protected option file.
`javax.net.ssl.keyStoreProvider`[1]	Default keystore provider; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	None
`javax.net.ssl.keyStoreType`[1]	Default keystore type; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`KeyStore.getDefaultType()`	None
`javax.net.ssl.sessionCacheSize` (since JDK 8u261)	Default value for the maximum number of entries in the	20480	The session cache size can be set by calling the `SSLSessionContext.setSessionCacheSize` method or by setting the `javax.net.ssl.sessionCachSize` system property. If the cache size is not set, the default value is used.

	SSL session cache		
`javax.net.ssl.trustStore`[1]	Default truststore; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`jssecacerts`, if it exists; otherwise, `cacerts`	None
`javax.net.ssl.trustStorePassword`[1]	Default truststore password; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	It is inadvisable to specify the password in a way that exposes it to discovery by other users. For example, specifying the password on the command line. To keep the password secure, have the application prompt for the password, or specify the password in a properly protected option file.
`javax.net.ssl.trustStoreProvider`[1]	Default truststore provider; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	None	None
`javax.net.ssl.trustStoreType`[1]	Default truststore type; see Customizing the Default Keystores and Truststores, Store Types, and Store Passwords	`KeyStore.getDefaultType()`	The value `NONE` may be specified. This setting is appropriate if the truststore is not file-based (for example, it resides in a hardware token).
`jdk.tls.acknowledgeCloseNotify`[1] (since JDK 8u261)	Specifying that close_notify Alert Is Sent When One Is Received	false	If the system property is set to true, then when the client or server receives a close_notify alert, it sends a corresponding close_notify alert and the connection is duplex closed.
`jdk.tls.client.cipherSuites`[1]	Client-side default enabled cipher suites; see Specifying Default Enabled Cipher Suites.	See SunJSSE Cipher Suites for a list of currently implemented `SunJSSE` cipher suites for this JDK release, sorted by order of preference.	**Caution**: These system properties can be used to configure weak cipher suites, or the configured cipher suites may be weak in the future. It is not recommended that you use these system properties without understanding the risks.
`jdk.tls.client.protocols`[1]	Default handshaking protocols for TLS clients. See The SunJSSE Provider and Enabling TLS 1.3.	None	To enable specific `SunJSSE` protocols on the client, specify them in a comma-separated list within quotation marks; all other supported protocols are not enabled on the client. For example, if `jdk.tls.client.protocols="TLSv1,TLSv1.1"`, then the default protocol settings on the client for TLSv1 and TLSv1.1 are enabled, while SSLv3, TLSv1.2, TLSv1.3, and SSLv2Hello are not enabled.
`jdk.tls.client.SignatureSchemes`[1]	Contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used	None	Unrecognized or unsupported signature scheme names specified in the property are ignored. If this system property is not defined or empty, then the provider-specific default is used. The names are not case sensitive. For a list of signature scheme names, see the section Signature Schemes in Java Cryptography Architecture Standard Algorithm Name Documentation.

	for TLS connections on the client side.		
`jdk.tls.ephemeralDHKeySize`1	Customizing Size of Ephemeral Diffie-Hellman Keys	1024 bits	None
`jdk.tls.namedGroups`1	Customizing the supported named groups for TLS key exchange	If this system property is not defined or the value is empty, then the implementation default groups and preferences will be used.	This contains a comma-separated list within quotation marks of enabled named groups in preference order. For example: `jdk.tls.namedGroups="secp521r1,secp256r1,ffdhe2048"`
`jdk.tls.server.cipherSuites`1	Server-side default enabled cipher suites. See Specifying Default Enabled Cipher Suites	See SunJSSE Cipher Suites to determine which cipher suites are enabled by default	**Caution**: These system properties can be used to configure weak cipher suites, or the configured cipher suites may be weak in the future. It is not recommended that you use these system properties without understanding the risks.
`jdk.tls.server.protocols`1 (since JDK 8u261)	Default handshaking protocols for TLS servers. See The SunJSSE Provider and Enabling TLS 1.3.	None	To configure the default enabled protocol suite in the server side of a `SunJSSE` provider, specify the protocols in a comma-separated list within quotation marks. The protocols in this list are standard SSL protocol names as described in Java Security Standard Algorithms. Note that this system property impacts only the default protocol suite (SSLContext of the algorithms SSL and TLS). If an application uses a version-specific `SSLContext` (SSLv3, TLSv1, TLSv1.1, TLSv1.2, or TLSv1.3), or sets the enabled protocol version explicitly, this system property has no impact.
`jdk.tls.server.SignatureSchemes`1	Contains a comma-separated list of supported signature scheme names that specifies the signature schemes that could be used for TLS connections on the server side.	None	Unrecognized or unsupported signature scheme names specified in the property are ignored. If this system property is not defined or empty, then the provider-specific default is used. The names are not case sensitive. For a list of signature scheme names, see the section Signature Schemes in Java Cryptography Architecture Standard Algorithm Name Documentation.
`jsse.enableFFDHEExtension`1 (since JDK 8u261)	Enables or disables Finite Field Diffie-Hellman Ephemeral (FFDHE) parameters for TLS key exchange	true	FFDHE is a TLS extension defined in RFC 7919. It enables TLS connections to use known finite field Diffie-Hellman groups. Some very old TLS vendors may not be able handle TLS extensions. In this case, set this property to false to disable the FFDHE extension.
`jsse.enableMFLNExtension`1 (since JDK 8u261)	Customizing Maximum Fragment Length Negotiation (MFLN) Extension	false	None
`jsse.enableSNIExtension`1	Server Name Indication option	true	Server Name Indication (SNI) is a TLS extension, defined in RFC 6066. It enables TLS connections to virtual servers, in which multiple servers for different network names are hosted at a single underlying network

			address. Some very old TLS vendors may not be able handle TLS extensions. In this case, set this property to false to disable the SNI extension
`jsse.SSLEngine.acceptLargeFragments`[1]	Default sizing buffers for large TLS packets	None	Setting this system property to true, `SSLSession` will size buffers to handle large data packets by default (see the note in SSLSession and ExtendedSSLSession. This may cause applications to allocate unnecessarily large `SSLEngine` buffers. Instead, applications should dynamically check for buffer overflow conditions and resize buffers as appropriate (see Understanding SSLEngine Operation Statuses).
`jdk.tls.client.enableStatusRequestExtension`[1]	Setting up a Java Client to use Client-Driven OCSP	false	If true, then the status_request and status_request_v2 extensions are enabled, and processing for CertificateStatus messages sent by the server is enabled.
`jdk.tls.server.enableStatusRequestExtension`[1]	Setting Up a Java Server to Use OCSP Stapling	false	If true, then server-side support for OCSP stapling is enabled
`sun.security.ssl.allowLegacyHelloMessages`	Allow Legacy Hello Messages (Renegotiations)	true	If true, then allow the peer to handshake without requiring the proper RFC 5746 messages.
`sun.security.ssl.allowUnsafeRenegotiation`	Allow Unsafe SSL/TLS Renegotiations	false	If true, then permit full (unsafe) legacy negotiation.

[1]This system property is currently used by the JSSE implementation, but it is not guaranteed to be examined and used by other implementations. If it is examined by another implementation, then that implementation should handle it in the same manner as the JSSE implementation does. There is no guarantee the property will continue to exist or be of the same type (system or Security) in future releases.

How to Specify a java.security.Security Property

You can customize some aspects of JSSE by setting security properties. You can set a Security Property either statically or dynamically:

- To set a Security Property statically, add a line to the Security Properties file. The Security Properties file is located at `java-home/lib/security/java.security`.

  `java-home` refers to the directory where the JRE is installed.

  To specify a Security Property value in the Security Properties file, you add a line of the following form:

  ```
  propertyName=propertyValue
  ```

  For example, suppose that you want to specify a different key manager factory algorithm name than the default SunX509. You do this by specifying the algorithm name as the value of a security property named `ssl.KeyManagerFactory.algorithm`. For example, to set the value to MyX509, add the following line to the Security Properties file:

  ```
  ssl.KeyManagerFactory.algorithm=MyX509
  ```

- To set a Security Property dynamically, call the `java.security.Security.setProperty` method in your code:

  ```
  Security.setProperty("propertyName," "propertyValue");
  ```

  For example, a call to the `setProperty()` method corresponding to the previous example for specifying the key manager factory algorithm name would be:

  ```
  Security.setProperty("ssl.KeyManagerFactory.algorithm", "MyX509");
  ```

How to Specify a java.lang.System Property

You can customize some aspects of JSSE by setting system properties. There are several ways to set these properties:

- To set a system property statically, use the `-D` option of the `java` command. For example, to run an application named MyApp and set the `javax.net.ssl.trustStore` system property to specify a truststore named MyCacertsFile, enter the following:

```
java -Djavax.net.ssl.trustStore=MyCacertsFile MyApp
```

- To set a system property dynamically, call the `java.lang.System.setProperty()` method in your code:

  ```
  System.setProperty("propertyName", "propertyValue");
  ```

  For example, a `setProperty()` call corresponding to the previous example for setting the `javax.net.ssl.trustStore` system property to specify a truststore named "`MyCacertsFile`" would be:

  ```
  System.setProperty("javax.net.ssl.trustStore", "MyCacertsFile");
  ```

- In the Java Deployment environment (Plug-In/Web Start), there are several ways to set the system properties. For more information, see Java Platform, Standard Edition Deployment Guide.
  - Use the Java Control Panel to set the Runtime Environment Property on a local or per-VM basis. This creates a local `deployment.properties` file. Deployers can also distribute an enterprise wide `deployment.properties` file by using the `deployment.config` mechanism. For more information, see Deployment Configuration File and Properties.
  - To set a property for a specific applet, use the HTML subtag `<PARAM>` "java_arguments" within the `<APPLET>` tag. For more information, see the Command-line Arguments section of the Java Platform, Standard Edition Deployment Guide.
  - To set the property in a specific Java Web Start application or applet using Java Plug-in, use the JNLP `property` sub-element of the `resources` element. For more information, see the resources Element section of the Java Web Start Guide.

Enabling TLS 1.3

JDK 8u261 and later includes an implementation of the Transport Layer Security (TLS) 1.3 specification (RFC 8446).

TLS 1.3 is disabled for the default `SSLContext` (`SSL` or `TLS`) at the client endpoint.

Examples of how to enable the TLS 1.3 protocol at the client endpoint include the following:

- Specify the supported protocols of an existing connection with the `SSLSocket.setEnabledProtocols` method:

  ```
  sslSocket.setEnabledProtocols(new String[] { "TLSv1.3", "TLSv1.2"});
  ```

- Create a TLS 1.3-based `SSLContext`:

  ```
  SSLContext ctx = SSLContext.getInstance("TLSv1.3");
  ```

- Specify the supported protocols with the `SSLParameters.setProtocols` method:

  ```
  sslParameters.setProtocols(new String[] {"TLSv1.3", "TLSv1.2"});
  ```

- Specify the supported protocols for client `SSLSocket`s with the `jdk.tls.client.protocols` system property:

  ```
  java -Djdk.tls.client.protocols="TLSv1.3,TLSv1.2" MyApplication
  ```

- Specify the supported protocols for connections obtained through `HttpsURLConnection` or the method `URL.openStream` with the `https.protocols` system property:

  ```
  java -Dhttps.protocols="TLSv1.3,TLSv1.2" MyApplication
  ```

Note: Although TLS 1.3 is enabled on the server by default, you can specify the default enabled protocol suite in the server side of a SunJSSE provider with the `jdk.tls.server.protocols` system property.

TLS 1.3 Not Directly Compatible with Previous Versions

TLS 1.3 is not directly compatible with previous versions. Although TLS 1.3 can be implemented with a backward-compatibility mode, there are still several compatibility risks to consider when upgrading to TLS 1.3:

- TLS 1.3 uses a half-close policy, while TLS 1.2 and earlier use a duplex-close policy. For applications that depend on the duplex-close policy, there may be compatibility issues when upgrading to TLS 1.3.
- The signature_algorithms_cert extension requires that pre-defined signature algorithms are used for certificate authentication. In practice, however, an application may use unsupported signature algorithms.
- The DSA signature algorithm is not supported in TLS 1.3. If a server is configured to only use DSA certificates, it cannot negotiate a TLS 1.3 connection.

- The supported cipher suites for TLS 1.3 are not the same as TLS 1.2 and earlier. If an application hardcodes cipher suites that are no longer supported, it may not be able to use TLS 1.3 without modifications to its code, for example TLS_AES_128_GCM_SHA256 (1.3 and later) versus TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (1.2 and earlier).
- The TLS 1.3 session resumption and key update behaviors are different from TLS 1.2 and earlier. The compatibility impact should be minimal, but it could be a risk if an application depends on the handshake details of the TLS protocols.

Customizing the X509Certificate Implementation

The X509Certificate implementation returned by the `X509Certificate.getInstance()` method is by default the implementation from the JSSE implementation.

You can optionally cause a different implementation to be returned. To do so, specify the name (and package) of the other implementation's class as the value of a Security Property named `cert.provider.x509v1`. For example, if the class is called `MyX509CertificateImpl` and it appears in the `com.cryptox` package, then you should add the following line to the Security Properties file:

```
cert.provider.x509v1=com.cryptox.MyX509CertificateImpl
```

Specifying Default Enabled Cipher Suites

You can specify the default enabled cipher suites in your application or with the system properties `jdk.tls.client.cipherSuites` and `jdk.tls.server.cipherSuites`.

Note: The actual use of enabled cipher suites is restricted by algorithm constraints.

The set of cipher suites to enable by default is determined by one of the following ways in this order of preference:

1. Explicitly set by application
2. Specified by system property
3. Specified by JSSE provider defaults

For example, explicitly setting the default enabled cipher suites in your application overrides settings specified in `jdk.tls.client.cipherSuites` or `jdk.tls.server.cipherSuites` as well as JSSE provider defaults.

Explicitly Set by Application

You can set which cipher suites are enabled with one of the following methods:

- `SSLSocket.setEnabledCipherSuites(String[])`
- `SSLEngine.setEnabledCipherSuites(String[])`
- `SSLServerSocket.setEnabledCipherSuites(String[])`
- `SSLParameters(String[] cipherSuites)`
- `SSLParameters(String[] cipherSuites, String[] protocols)`
- `SSLParameters.setCipherSuites(String[])`
- `https.cipherSuites` system property for `HttpsURLConnection`

Specified by System Property

The system property `jdk.tls.client.cipherSuites` specifies the default enabled cipher suites on the client side; `jdk.tls.server.cipherSuites` specifies those on the server side.

The syntax of the value of these two system properties is a comma-separated list of supported cipher suite names. Unrecognized or unsupported cipher suite names that are specified in these properties are ignored. See Java Security Standard Algorithms for standard JSSE cipher suite names.

Note: These system properties are currently supported by Oracle JDK. They are not guaranteed to be supported by other JDK implementations.

Caution: These system properties can be used to configure weak cipher suites, or the configured cipher suites may be weak in the future. It is not recommended that you use these system properties without understanding the risks.

Specified by JSSE Provider Defaults

Each JSSE provider has its own default enabled cipher suites. See The SunJSSE Provider in Java Cryptography Architecture Oracle Providers Documentation for JDK 8 for the cipher suite names supported by the `SunJSSE` provider and which ones that are enabled by default.

Specifying an Alternative HTTPS Protocol Implementation

You can communicate securely with an SSL-enabled web server by using the HTTPS URL scheme for the `java.net.URL` class. The JDK provides a default HTTPS URL implementation.

If you want an alternative HTTPS protocol implementation to be used, set the `java.protocol.handler.pkgs` system property to include the new class name. This action causes the specified classes to be found and loaded before the JDK default classes. See the `java.net.URL` class documentation for details.

Note: In past JSSE releases, you had to set the `java.protocol.handler.pkgs` system property during JSSE installation. This step is no longer required unless you want to obtain an instance of `com.sun.net.ssl.HttpsURLConnection`. For more information, see Code Using the HttpsURLConnection Class in the "Troubleshooting" section.

Customizing the Provider Implementation

JDK 1.4 and later releases come standard with a JSSE Cryptographic Service Provider, or provider for short, named SunJSSE. Providers are essentially packages that implement one or more engine classes for specific cryptographic algorithms. The JSSE engine classes are `SSLContext`, `KeyManagerFactory`, and `TrustManagerFactory`. For more information about providers and engine classes, see the Java Cryptography Architecture Reference Guide.

Note: The transformation strings used when SunJSSE calls `Cipher.getInstance()` are "RSA/ECB/PKCS1Padding", "RC4", "DES/CBC/NoPadding", and "DESede/CBC/NoPadding". For further information about the `Cipher` class and transformation strings see the Java Cryptography Architecture Reference Guide.

Before it can be used, a provider must be registered, either statically or dynamically. You do not need to register the SunJSSE provider because it is preregistered. If you want to use other providers, read the following sections to see how to register them.

Registering the Cryptographic Service Provider Statically

You register a provider statically by adding a line of the following form to the Security Properties file:

```
security.provider.n=providerClassName
```

This declares a provider, and specifies its preference order `n`. The preference order is the order in which providers are searched for requested algorithms (when no specific provider is requested). "1" is the most preferred, followed by "2", and so on.

The providerClassName is the fully qualified name of the provider class. You obtain this name from the provider vendor.

The standard security provider and the SunJSSE provider shipped with JDK 6 are automatically registered for you; the following lines appear in the `java.security` Security Properties file to register the `SunJCE` security provider with preference order 5 and the SunJSSE provider with preference order 4:

```
security.provider.1=sun.security.pkcs11.SunPKCS11 \
${java.home}/lib/security/sunpkcs11-solaris.cfg
security.provider.2=sun.security.provider.Sun
security.provider.3=sun.security.rsa.SunRsaSign
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
```

To use another JSSE provider, add a line registering the other provider, giving it whatever preference order you prefer.

You can have more than one JSSE provider registered at the same time. The registered providers may include different implementations for different algorithms for different engine classes, or they may have support for some or all of the same types of algorithms and engine classes. When a particular engine class implementation for a particular algorithm is searched for, if no specific provider is specified for the search, then the providers are searched in preference order and the implementation from the first provider that supplies an implementation for the specified algorithm is used.

Registering the Cryptographic Service Provider Dynamically

Instead of registering a provider statically, you can add the provider dynamically at runtime by calling the `Security.addProvider()` method at the beginning of your program. For example, to dynamically add a provider whose provider class name is `MyProvider` and whose `MyProvider` class resides in the `com.ABC` package, you would call:

```
Security.addProvider(new com.ABC.MyProvider());
```

The `Security.addProvider()` method adds the specified provider to the next available preference position.

This type of registration is not persistent and can only be done by a program with sufficient permissions.

Customizing the Default Keystores and Truststores, Store Types, and Store Passwords

Whenever a default `SSLSocketFactory` or `SSLServerSocketFactory` is created (via a call to `SSLSocketFactory.getDefault` or `SSLServerSocketFactory.getDefault`), and this default `SSLSocketFactory` (or `SSLServerSocketFactory`) comes from the JSSE reference implementation, a default `SSLContext` is associated with the socket factory. (The default socket factory will come from the JSSE implementation.)

This default `SSLContext` is initialized with a default `KeyManager` and a default `TrustManager`. If a keystore is specified by the `javax.net.ssl.keyStore` system property and an appropriate `javax.net.ssl.keyStorePassword` system property, then the `KeyManager` created by the default `SSLContext` will be a `KeyManager` implementation for managing the specified keystore. (The actual implementation will be as specified in Customizing the Default Key and Trust Managers.) If no such system property is specified, then the keystore managed by the `KeyManager` will be a new empty keystore.

Generally, the peer acting as the server in the handshake will need a keystore for its KeyManager in order to obtain credentials for authentication to the client. However, if one of the anonymous cipher suites is selected, then the server's `KeyManager` keystore is not necessary. And, unless the server requires client authentication, the peer acting as the client does not need a `KeyManager` keystore. Thus, in these situations it may be OK if no `javax.net.ssl.keyStore` system property value is defined.

Similarly, if a truststore is specified by the `javax.net.ssl.trustStore` system property, then the `TrustManager` created by the default `SSLContext` will be a `TrustManager` implementation for managing the specified truststore. In this case, if such a property exists but the file it specifies does not, then no truststore is used. If no `javax.net.ssl.trustStore` property exists, then a default truststore is searched for. If a truststore named `java-home/lib/security/jssecacerts` is found, it is used. If not, then a truststore named `java-home/lib/security/cacerts` is searched for and used (if it exists). Finally, if a truststore is still not found, then the truststore managed by the `TrustManager` will be a new empty truststore.

Note: The JDK ships with a limited number of trusted root certificates in the `java-home/lib/security/cacerts` file. As documented in keytool reference pages, it is your responsibility to maintain (that is, add and remove) the certificates contained in this file if you use this file as a truststore.

Depending on the certificate configuration of the servers that you contact, you may need to add additional root certificates. Obtain the needed specific root certificates from the appropriate vendor.

If the `javax.net.ssl.keyStoreType` and/or `javax.net.ssl.keyStorePassword` system properties are also specified, then they are treated as the default `KeyManager` keystore type and password, respectively. If no type is specified, then the default type is that returned by the `KeyStore.getDefaultType()` method, which is the value of the `keystore.type` Security Property, or "jks" if no such Security Property is specified. If no keystore password is specified, then it is assumed to be a blank string "".

Similarly, if the `javax.net.ssl.trustStoreType` and/or `javax.net.ssl.trustStorePassword` system properties are also specified, then they are treated as the default truststore type and password, respectively. If no type is specified, then the default type is that returned by the `KeyStore.getDefaultType()` method. If no truststore password is specified, then it is assumed to be a blank string "".

Note: This section describes the current JSSE reference implementation behavior. The system properties described in this section are not guaranteed to continue to have the same names and types (system or security) or even to exist at all in future releases. They are also not guaranteed to be examined and used by any other JSSE implementations. If they are examined by an implementation, then that implementation should handle them in the same manner as the JSSE reference implementation does, as described herein.

Customizing the Default Key Managers and Trust Managers

As noted in Customizing the Default Keystores and Truststores, Store Types, and Store Passwords, whenever a default `SSLSocketFactory` or `SSLServerSocketFactory` is created, and this default `SSLSocketFactory` (or `SSLServerSocketFactory`) comes from the JSSE reference implementation, a default `SSLContext` is associated with the socket factory.

This default `SSLContext` is initialized with a `KeyManager` and a `TrustManager`. The `KeyManager` and/or `TrustManager` supplied to the default `SSLContext` will be an implementation for managing the specified keystore or truststore, as described in the aforementioned section.

The `KeyManager` implementation chosen is determined by first examining the `ssl.KeyManagerFactory.algorithm` Security Property. If such a property value is specified, then a `KeyManagerFactory` implementation for the specified algorithm is searched for. The implementation from the first provider that supplies an implementation is used. Its `getKeyManagers()` method is called to determine the `KeyManager` to supply to the default `SSLContext`. Technically, `getKeyManagers()` returns an array of `KeyManager` objects, one `KeyManager` for each type of key material. If no such Security Property value is specified, then the default value of SunX509 is used to perform the search.

Note: A `KeyManagerFactory` implementation for the SunX509 algorithm is supplied by the SunJSSE provider. The `KeyManager` that it specifies is a `javax.net.ssl.X509KeyManager` implementation.

Similarly, the `TrustManager` implementation chosen is determined by first examining the `ssl.TrustManagerFactory.algorithm` Security Property. If such a property value is specified, then a `TrustManagerFactory` implementation for the specified algorithm is searched for. The implementation from the first provider that supplies an implementation is used. Its `getTrustManagers()` method is called to determine the `TrustManager` to supply to the default `SSLContext`. Technically, `getTrustManagers()` returns an array of `TrustManager` objects, one `TrustManager` for each type of trust material. If no such Security Property value is specified, then the default value of PKIX is used to perform the search.

Note: A `TrustManagerFactory` implementation for the PKIX algorithm is supplied by the SunJSSE provider. The `TrustManager` that it specifies is a `javax.net.ssl.X509TrustManager` implementation.

Note: This section describes the current JSSE reference implementation behavior. The system properties described in this section are not guaranteed to continue to have the same names and types (system or security) or even to exist at all in future releases. They are also not guaranteed to be examined and used by any other JSSE implementations. If they are examined by an implementation, then that implementation should handle them in the same manner as the JSSE reference implementation does, as described herein.

Disabled and Restricted Cryptographic Algorithms

In some environments, certain algorithms or key lengths may be undesirable when using TLS. The Oracle JDK uses the `jdk.certpath.disabledAlgorithms` and `jdk.tls.disabledAlgorithm` Security Properties to disable algorithms during TLS protocol negotiation, including version negotiation, cipher suites selection, peer authentication, and key exchange mechanisms. Note that these Security Properties are not guaranteed to be used by other JDK implementations. See the `<java-home>/lib/security/java.security` file for information about the syntax of these Security Properties and their current active values.

- `jdk.certpath.disabledAlgorithms` Property: CertPath code uses this Security Property to determine which algorithms should not be allowed during CertPath checking. For example, when a TLS server sends an identifying certificate chain, a client TrustManager that uses a CertPath implementation to verify the received chain will not allow the stated conditions. For example, the following line blocks any MD2-based certificate, as well as SHA1 TLSServer certificates that chain to trust anchors that are pre-installed in the cacaerts keystore. Likewise, this line blocks any RSA key less than 1024 bits.

  ```
  jdk.certpath.disabledAlgorithms=MD2, SHA1 jdkCA & usage TLSServer, RSA keySize < 1024
  ```

- `jdk.tls.disabledAlgorithms` Property: SunJSSE code uses this Security Property to disable TLS protocols, cipher suites, keys, and so on. The syntax is similar to the `jdk.certpath.disabledAlgorithms` Security Property. For example, the following line disables the SSLv3 algorithm and all of the TLS_*_RC4_* cipher suites:

  ```
  jdk.tls.disabledAlgorithms=SSLv3, RC4
  ```

If you require a particular condition, you can reactivate it by either removing the associated value in the Security Property in the `java.security` file or dynamically setting the proper Security Property before JSSE is initialized.

Note that these Security Properties effectively create a third set of cipher suites, Disabled. The following list describes these three sets:

- **Disabled**: If a cipher suite contains any components (for example, RC4) on the disabled list (for example, RC4 is specified in the `jdk.tls.disabledAlgorithms` Security Property), then that cipher suite is disabled and will not be considered for a connection handshake.
- **Enabled**: A list of specific cipher suites that will be considered for a connection.
- **Not Enabled**: A list of non-disabled cipher suites that will not be considered for a connection. To re-enable these cipher suites, call the appropriate `setEnabledCipherSuites` or `setSSLParameters` methods.

Legacy Cryptographic Algorithms

In some environments, a certain algorithm may be undesirable but it cannot be disabled because of its use in legacy applications. Legacy algorithms may still be supported, but applications should not use them as the security strength of legacy algorithms is usually not strong enough. During TLS security parameters negotiation, legacy algorithms are not negotiated unless there are no other candidates. The Security Property `jdk.tls.legacyAlgorithms` specifies which algorithms the Oracle JDK considers as legacy algorithms. See the `<java-home>/lib/security/java.security` file for the syntax of this Security Property.

Note:

- If a legacy algorithm is also restricted through the `jdk.tls.disabledAlgorithms` property or the `java.security.AlgorithmConstraints` API (see the method `javax.net.ssl.SSLParameters.setAlgorithmConstraints`), then the algorithm is completely disabled and will not be negotiated.

- If your application uses an algorithm specified in the Security Property `jdk.tls.legacyAlgorithms`, use an alternative algorithm as soon as possible; a future JDK release may specify a legacy algorithm as a restricted algorithm.

Customizing the Encryption Algorithm Providers

The SunJSSE provider uses the SunJCE implementation for all its cryptographic needs. Although it is recommended that you leave the provider at its regular position, you can use implementations from other JCA or JCE providers by registering them before the SunJCE provider. The standard JCA mechanism can be used to configure providers, either statically via the Security Properties file `java-home/lib/security/java.security`, or dynamically via the `addProvider()` or `insertProviderAt()` method in the `java.security.Security` class.

Customizing Size of Ephemeral Diffie-Hellman Keys

Diffie-Hellman (DH) keys of sizes less than 1024 bits have been deprecated because of their insufficient strength. You can customize the ephemeral DH key size with the system property `jdk.tls.ephemeralDHKeySize`. This system property does not impact DH key sizes in ServerKeyExchange messages for exportable cipher suites. It impacts only the DHE_RSA, DHE_DSS, and DH_anon-based cipher suites in the JSSE Oracle provider.

You can specify one of the following values for this property:

- Undefined: A DH key of size 1024 bits will be used always for non-exportable cipher suites. This is the default value for this property.
- `legacy`: The JSSE Oracle provider preserves the legacy behavior (for example, using ephemeral DH keys of sizes 512 bits and 768 bits) of JDK 7 and earlier releases.
- `matched`: For non-exportable anonymous cipher suites, the DH key size in ServerKeyExchange messages is 1024 bits. For X.509 certificate based authentication (of non-exportable cipher suites), the DH key size matching the corresponding authentication key is used, except that the size must be between 1024 bits and 2048 bits. For example, if the public key size of an authentication certificate is 2048 bits, then the ephemeral DH key size should be 2048 bits unless the cipher suite is exportable. This key sizing scheme keeps the cryptographic strength consistent between authentication keys and key-exchange keys.
- A valid integer between 1024 and 2048, inclusively: A fixed ephemeral DH key size of the specified value, in bits, will be used for non-exportable cipher suites.

The following table summaries the minimum and maximum acceptable DH key sizes for each of the possible values for the system property `jdk.tls.ephemeralDHKeySize`:

Value of `jdk.tls.ephemeralDHKeySize`	**Undefined**	`legacy`	`matched`	**Integer value (fixed)**
Exportable DH key size	512	512	512	512
Non-exportable anonymous cipher suites	1024	768	1024	The fixed key size is specified by a valid integer property value, which must be between 1024 and 2048, inclusively.
Authentication certificate	1024	768	The key size is the same as the authentication certificate, but must be between 1024 bits and 2048 bits, inclusively. However, the SunJCE provider only supports 2048-bit DH keys larger than 1024 bits. Consequently, you may use the values 1024 or 2048 only.	The fixed key size is specified by a valid integer property value, which must be between 1024 and 2048, inclusively.

Customizing Maximum Fragment Length Negotiation (MFLN) Extension

In order to negotiate smaller maximum fragment lengths, clients have an option to include an extension of type max_fragment_length in the ClientHello message. A system property, `jsse.enableMFLNExtension`, can be used to enable or disable the MFLN extension for TLS.

Maximum Fragment Length Negotiation

It may be desirable for constrained TLS clients to negotiate a smaller maximum fragment length due to memory limitations or bandwidth limitations. In order to negotiate smaller maximum fragment lengths, clients have an option to include an extension of type max_fragment_length in the (extended) ClientHello message. See RFC 6066.

Once a maximum fragment length has been successfully negotiated, the TLS client and server can immediately begin fragmenting messages (including handshake messages) to ensure that no fragment larger than the negotiated length is sent.

System Property jsse.enableMFLNExtension

A system property `jsse.enableMFLNExtension` is defined to enable or disable the MFLN extension. The `jsse.enableMFLNExtension` is disabled by default.

The value of the system property can be set as follows:

System Property	Description
`jsse.enableMFLNExtension=true`	Enable the MFLN extension. If the returned value of `SSLParameters.getMaximumPacketSize()` is less than (2^{12} + header-size) the maximum fragment length negotiation extension would be enabled.
`jsse.enableMFLNExtension=false`	Disable the MFLN extension.

Limiting Amount of Data Algorithms May Encrypt with a Set of Keys

You can specify a limit on the amount of data an algorithm may encrypt with a specific set of keys with the `jdk.tls.keyLimits` Security Property. Once this limit is reached, a KeyUpdate post-handshake message is sent, which requests that the current set of keys be updated. This Security Property is only for symmetrical ciphers with TLS 1.3.

The syntax for this property is as follows:

```
jdk.tls.keyLimits=KeyLimit { , KeyLimit }

KeyLimit:
AlgorithmName KeyUpdate Length
```

- `AlgorithmName`: A full algorithm transformation
- `Length`: The amount of encrypted data in a session before a KeyUpdate message is sent. This value may be an integer value in bytes or as a power of two, for example, 2^37.

For example, the following specifies that a KeyUpdate message is sent once the algorithm AES/GCM/NoPadding has encrypted 237 bytes:

```
jdk.tls.keyLimits=AES/GCM/NoPadding KeyUpdate 2^37
```

Specifying that close_notify Alert Is Sent When One Is Received

If the `jdk.tls.acknowledgeCloseNotify` system property is set to true, then when the client or server receives a close_notify alert, it sends a corresponding close_notify alert and the connection is duplex closed.

TLS 1.2 and earlier versions use a duplex-close policy. However, TLS 1.3 uses a half-close policy, which means that the inbound and the outbound close_notify alerts are independent. When upgrading to TLS 1.3, unexpected behavior can occur if your application shuts down the TLS connection by using only one of the `SSLEngine.closeInbound()` or `SSLEngine.closeOutbound()` methods but not both on each side of the connection. If your application unexpectedly hangs or times out when the underlying TLS transportation is not duplex closed, you may need to set this property to true.

Note that when a TLS connection is no longer needed, the client and server applications should each close both sides of their respective connection.

Determine X.509 Certificate Revocation Status with OCSP

Use the Online Certificate Status Protocol (OCSP) to determine the X.509 certificate revocation status during the Transport Layer Security (TLS) handshake. See Client-Driven OCSP and OCSP Stapling.

Hardware Acceleration and Smartcard Support

The Java Cryptography Architecture (JCA) is a set of packages that provides a framework and implementations for encryption, key generation and key agreement, and message authentication code (MAC) algorithms. The SunJSSE provider uses JCA exclusively for all of its cryptographic operations and can automatically take advantage of JCE features and enhancements, including JCA's support for PKCS#11. This support enables the SunJSSE provider to use hardware cryptographic accelerators for significant performance improvements and to use smartcards as keystores for greater flexibility in key and trust management.

Use of hardware cryptographic accelerators is automatic if JCA has been configured to use the Oracle PKCS#11 provider, which in turn has been configured to use the underlying accelerator hardware. The provider must be configured before any other JCA providers in the provider list. For details on how to configure the Oracle PKCS#11 provider, see the PKCS#11 Guide.

Configuring JSSE to Use Smartcards as Keystores and Truststores

Support for PKCS#11 in JCA also enables access to smartcards as a keystore. For details on how to configure the type and location of the keystores to be used by JSSE, see the Customizing JSSE section. To use a smartcard as a keystore or truststore, set the `javax.net.ssl.keyStoreType` and `javax.net.ssl.trustStoreType` system properties, respectively, to `pkcs11`, and set the `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` system properties, respectively, to `NONE`. To specify the use of a specific provider, use the `javax.net.ssl.keyStoreProvider` and `javax.net.ssl.trustStoreProvider` system properties (for example, set them to SunPKCS11-joe). By using these properties, you can configure an application that previously depended on these properties to access a file-based keystore to use a smartcard keystore with no changes to the application.

Some applications request the use of keystores programmatically. These applications can continue to use the existing APIs to instantiate a Keystore and pass it to its key manager and trust manager. If the Keystore instance refers to a PKCS#11 keystore backed by a Smartcard, then the JSSE application will have access to the keys on the smartcard.

Multiple and Dynamic Keystores

smartcards (and other removable tokens) have additional requirements for an X509KeyManager. Different smartcards can be present in a smartcard reader during the lifetime of a Java application, and they can be protected using different passwords.

The java.security.KeyStore.Builder class abstracts the construction and initialization of a KeyStore object. It supports the use of CallbackHandler for password prompting, and its subclasses can be used to support additional features as desired by an application. For example, it is possible to implement a Builder that allows individual KeyStore entries to be protected with different passwords. The javax.net.ssl.KeyStoreBuilderParameters class then can be used to initialize a KeyManagerFactory using one or more of these Builder objects.

A X509KeyManager implementation in the SunJSSE provider called NewSunX509 supports these parameters. If multiple certificates are available, it attempts to pick a certificate with the appropriate key usage and prefers valid to expired certificates.

The following example illustrates how to tell JSSE to use both a PKCS#11 keystore (which might in turn use a smartcard) and a PKCS#12 file-based keystore.

```
import javax.net.ssl.*;
import java.security.KeyStore.*;
...

// Specify keystore builder parameters for PKCS#11 keystores
Builder scBuilder = Builder.newInstance("PKCS11", null,
new CallbackHandlerProtection(myGuiCallbackHandler));

// Specify keystore builder parameters for a specific PKCS#12 keystore
Builder fsBuilder = Builder.newInstance("PKCS12", null,
new File(pkcsFileName), new PasswordProtection(pkcsKsPassword));

// Wrap them as key manager parameters
ManagerFactoryParameters ksParams = new KeyStoreBuilderParameters(
Arrays.asList(new Builder[] { scBuilder, fsBuilder }) );

// Create KeyManagerFactory
KeyManagerFactory factory = KeyManagerFactory.getInstance("NewSunX509");

// Pass builder parameters to factory
factory.init(ksParams);

// Use factory
SSLContext ctx = SSLContext.getInstance("TLS");
ctx.init(factory.getKeyManagers(), null, null);
```

Kerberos Cipher Suites

The SunJSSE provider has support for Kerberos cipher suites, as described in RFC 2712. The following cipher suites are supported but not enabled by default:

- TLS_KRB5_WITH_RC4_128_SHA
- TLS_KRB5_WITH_RC4_128_MD5
- TLS_KRB5_WITH_3DES_EDE_CBC_SHA
- TLS_KRB5_WITH_3DES_EDE_CBC_MD5
- TLS_KRB5_WITH_DES_CBC_SHA
- TLS_KRB5_WITH_DES_CBC_MD5
- TLS_KRB5_EXPORT_WITH_RC4_40_SHA
- TLS_KRB5_EXPORT_WITH_RC4_40_MD5
- TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
- TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5

To enable the use of these cipher suites, you must do so explicitly. For more information, see the API documentation for the SSLEngine.setEnabledCipherSuites() and SSLSocket.setEnabledCipherSuites() methods. As with all other SSL/TLS cipher suites, if a cipher suite is not supported by the peer, then it will not be selected during cipher negotiation. Furthermore, if the application and/or server cannot acquire the necessary Kerberos credentials, then the Kerberos cipher suites also will not be selected.

The following is an example of a TLS client that will only use the TLS_KRB5_WITH_DES_CBC_SHA cipher suite:

```
// Create socket
SSLSocketFactory sslsf = (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket sslSocket = (SSLSocket) sslsf.createSocket(tlsServer, serverPort);
```

```
// Enable only one cipher suite
String enabledSuites[] = { "TLS_KRB5_WITH_DES_CBC_SHA" };
sslSocket.setEnabledCipherSuites(enabledSuites);
```

Kerberos Requirements

You must have the Kerberos infrastructure set up in your deployment environment before you can use the Kerberos cipher suites with JSSE. In particular, both the TLS client and server must have accounts set up with the Kerberos Key Distribution Center (KDC). At runtime, if one or more of the Kerberos cipher suites have been enabled, then the TLS client and server will acquire their Kerberos credentials associated with their respective account from the KDC. For example, a TLS server running on the machine `mach1.imc.org` in the Kerberos realm `IMC.ORG` must have an account with the name `host/mach1.imc.org@IMC.ORG` and be configured to use the KDC for `IMC.ORG`. For information about using Kerberos with Java SE, see the Kerberos Requirements document.

An application can acquire its Kerberos credentials by using the Java Authentication and Authorization Service (JAAS) and a Kerberos login module. The JDK comes with a Kerberos login module. You can use the Kerberos cipher suites with JSSE with or without JAAS programming, similar to how you can use the Java Generic Security Services (Java GSS) with or without JAAS programming.

To use the Kerberos cipher suites with JSSE without JAAS programming, you must use the index names `com.sun.net.ssl.server` or `other` for the TLS server JAAS configuration entry, and `com.sun.net.ssl.client` or `other` for the TLS client, and set the `javax.security.auth.useSubjectCredsOnly` system property to false. For example, a TLS server that is not using JAAS programming might have the following JAAS configuration file:

```
com.sun.net.ssl.server {
com.sun.security.auth.module.Krb5LoginModule required
principal="host/mach1.imc.org@IMC.ORG"
useKeyTab=true
keyTab=mach1.keytab
storeKey=true;
};
```

An example of how to use Java GSS and Kerberos without JAAS programming is described in the Java GSS Tutorial. You can adapt it to use JSSE by replacing Java GSS calls with JSSE calls.

To use the Kerberos cipher suites with JAAS programming, you can use any index name because your application is responsible for creating the JAAS `LoginContext` using the index name, and then wrapping the JSSE calls inside of a `Subject.doAs()` or `Subject.doAsPrivileged()` call. An example of how to use JAAS with Java GSS and Kerberos is described in the Java GSS Tutorial. You can adapt it to use JSSE by replacing Java GSS calls with JSSE calls.

If you have trouble using or configuring the JSSE application to use Kerberos, see the Troubleshooting section of the Java GSS Tutorial.

Peer Identity Information

To determine the identity of the peer of an SSL connection, use the `getPeerPrincipal()` method in the following classes:

- `javax.net.ssl.SSLSession`
- `javax.net.ssl.HttpsURLConnection`
- `javax.net.HandshakeCompletedEvent`

Similarly, to get the identity that was sent to the peer (to identify the local entity), use the `getLocalPrincipal()` method in these classes. For X509-based cipher suites, these methods will return an instance of `javax.security.auth.x500.X500Principal`; for Kerberos cipher suites, these methods will return an instance of `javax.security.auth.kerberos.KerberosPrincipal`.

JSSE applications use `getPeerCertificates()` and similar methods in `javax.net.ssl.SSLSession`, `javax.net.ssl.HttpsURLConnection`, and `javax.net.HandshakeCompletedEvent` classes to obtain information about the peer. When the peer does not have any certificates, `SSLPeerUnverifiedException` is thrown.

If the application must determine only the identity of the peer or identity sent to the peer, then it should use the `getPeerPrincipal()` and `getLocalPrincipal()` methods, respectively. It should use `getPeerCertificates()` and `getLocalCertificates()` methods only if it must examine the contents of those certificates. Furthermore, the application must be prepared to handle the case where an authenticated peer might not have any certificate.

Security Manager

When the security manager has been enabled, in addition to the `SocketPermission` needed to communicate with the peer, a TLS client application that uses the Kerberos cipher suites also needs the following permission:

```
javax.security.auth.kerberos.ServicePermission(serverPrincipal, "initiate");
```

In the preceding code, serverPrincipal is the Kerberos principal name of the TLS server that the TLS client will be communicating with (such as `host/mach1.imc.org@IMC.ORG`). A TLS server application needs the following permission:

```
javax.security.auth.kerberos.ServicePermission(serverPrincipal, "accept");
```

In the preceding code, serverPrincipal is the Kerberos principal name of the TLS server (such as `host/mach1.imc.org@IMC.ORG`). If the server or client must contact the KDC (for example, if its credentials are not cached locally), then it also needs the following permission:

```
javax.security.auth.kerberos.ServicePermission(tgtPrincipal, "initiate");
```

In the preceding code, tgtPrincipal is the principal name of the KDC (such as `krbtgt/IMC.ORG@IMC.ORG`).

Additional Keystore Formats (PKCS12)

The PKCS#12 (Personal Information Exchange Syntax Standard) specifies a portable format for storage and/or transport of a user's private keys, certificates, miscellaneous secrets, and other items. The SunJSSE provider supplies a complete implementation of the PKCS12 `java.security.KeyStore` format for reading and writing PKCS12 files. This format is also supported by other toolkits and applications for importing and exporting keys and certificates, such as Netscape/Mozilla, Microsoft's Internet Explorer, and OpenSSL. For example, these implementations can export client certificates and keys into a file using the `.p12` file name extension.

With the SunJSSE provider, you can access PKCS12 keys through the `KeyStore` API with a keystore type of PKCS12. In addition, you can list the installed keys and associated certificates by using the `keytool` command with the `-storetype` option set to `pkcs12`. For more information about `keytool`, see Security Tools.

Server Name Indication (SNI) Extension

The SNI extension is a feature that extends the SSL/TLS protocols to indicate what server name the client is attempting to connect to during handshaking. Servers can use server name indication information to decide if specific `SSLSocket` or `SSLEngine` instances should accept a connection. For example, when multiple virtual or name-based servers are hosted on a single underlying network address, the server application can use SNI information to determine whether this server is the exact server that the client wants to access. Instances of this class can be used by a server to verify the acceptable server names of a particular type, such as host names. For more information, see section 3 of TLS Extensions (RFC 6066).

Developers of client applications can explicitly set the server name indication using the `SSLParameters.setServerNames(List<SNIServerName> serverNames)` method. The following example illustrates this functionality:

```
SSLSocketFactory factory = ...
SSLSocket sslSocket = factory.createSocket("172.16.10.6", 443);
// SSLEngine sslEngine = sslContext.createSSLEngine("172.16.10.6", 443);

SNIHostName serverName = new SNIHostName("www.example.com");
List<SNIServerName> serverNames = new ArrayList<>(1);
serverNames.add(serverName);

SSLParameters params = sslSocket.getSSLParameters();
params.setServerNames(serverNames);
sslSocket.setSSLParameters(params);
// sslEngine.setSSLParameters(params);
```

Developers of server applications can use the `SNIMatcher` class to decide how to recognize server name indication. The following two examples illustrate this functionality:

Example 1

```
SSLSocket sslSocket = sslServerSocket.accept();

SNIMatcher matcher = SNIHostName.createSNIMatcher("www\\.example\\.(com|org)");
Collection<SNIMatcher> matchers = new ArrayList<>(1);
matchers.add(matcher);

SSLParameters params = sslSocket.getSSLParameters();
params.setSNIMatchers(matchers);
sslSocket.setSSLParameters(params);
```

Example 2

```
SSLServerSocket sslServerSocket = ...;

SNIMatcher matcher = SNIHostName.createSNIMatcher("www\\.example\\.(com|org)");
Collection<SNIMatcher> matchers = new ArrayList<>(1);
matchers.add(matcher);

SSLParameters params = sslServerSocket.getSSLParameters();
params.setSNIMatchers(matchers);
sslServerSocket.setSSLParameters(params);

SSLSocket sslSocket = sslServerSocket.accept();
```

The following list provides examples for the behavior of the `SNIMatcher` when receiving various server name indication requests in the ClientHello message:

- Matcher configured to `www\\.example\\.com`:
  - If the requested host name is `www.example.com`, then it will be accepted and a confirmation will be sent in the ServerHello message.
  - If the requested host name is `www.example.org`, then it will be rejected with an `unrecognized_name` fatal error.
  - If there is no requested host name or it is empty, then the request will be accepted but no confirmation will be sent in the ServerHello message.
- Matcher configured to `www\\.invalid\\.com`:
  - If the requested host name is `www.example.com`, then it will be rejected with an `unrecognized_name` fatal error.
  - If the requested host name is `www.example.org`, then it will be accepted and a confirmation will be sent in the ServerHello message.
  - If there is no requested host name or it is empty, then the request will be accepted but no confirmation will be sent in the ServerHello message.
- Matcher is not configured:

  Any requested host name will be accepted but no confirmation will be sent in the ServerHello message.

For descriptions of new classes that implement the SNI extension, see:

- StandardConstants Class
- SNIServerName Class
- SNIMatcher Class
- SNIHostName Class

For examples, see Using the Server Name Indication (SNI) Extension.

TLS Application Layer Protocol Negotiation

In JDK 8u251 and later, Application Layer Protocol Negotiation (ALPN) enables you to negotiate an application protocol for a TLS connection; see TLS Application Layer Protocol Negotiation.

Troubleshooting

This section contains information for troubleshooting JSSE. First, it provides some common configuration problems and ways to solve them, and then it describes helpful debugging utilities.

Configuration Problems

This section describes some common configuration problems that might arise when you use JSSE.

CertificateException While Handshaking

Problem: When negotiating an SSL connection, the client or server throws a `CertificateException`.

Cause 1: This is generally caused by the remote side sending a certificate that is unknown to the local side.

Solution 1: The best way to debug this type of problem is to turn on debugging (see Debugging Utilities) and watch as certificates are loaded and when certificates are received via the network connection. Most likely, the received certificate is unknown to the trust mechanism because the wrong trust file was loaded. Refer to the following sections for more information:

- JSSE Classes and Interfaces
- The TrustManager Interface
- The KeyManager Interface

Cause 2: The system clock is not set correctly. In this case, the perceived time may be outside the validity period on one of the certificates, and unless the certificate can be replaced with a valid one from a truststore, the system must assume that the certificate is invalid, and therefore throw the exception.

Solution 2: Correct the system clock time.

java.security.KeyStoreException: TrustedCertEntry Not Supported

Problem: Attempt to store trusted certificates in PKCS12 keystore throws `java.security.KeyStoreException: TrustedCertEntry not supported`.

Cause: Storing trusted certificates in a PKCS12 keystore is not supported. PKCS12 is mainly used to deliver private keys with the associated certificate chains. It does not have any notion of "trusted" certificates. In terms of interoperability, other PKCS12 vendors have the same restriction. Browsers such as Mozilla and Internet Explorer do not accept a PKCS12 file with only trusted certificates.

Solution: Use the JKS keystore for storing trusted certificates.

Runtime Exception: SSL Service Not Available

Problem: When running a program that uses JSSE, an exception occurs indicating that an SSL service is not available. For example, an exception similar to one of the following is thrown:

```
Exception in thread "main" java.net.SocketException:
no SSL Server Sockets

Exception in thread "main":
SSL implementation not available
```

Cause: There was a problem with `SSLContext` initialization, for example, due to an incorrect password on a keystore or a corrupted keystore (a JDK vendor once shipped a keystore in an unknown format, and that caused this type of error).

Solution: Check initialization parameters. Ensure that any keystores specified are valid and that the passwords specified are correct. One way that you can check this is by trying to use the `keytool` command-line utility to examine the keystores and the relevant contents.

Runtime Exception: "No available certificate corresponding to the SSL cipher suites which are enabled"

Problem: When trying to run a simple SSL server program, the following exception is thrown:

```
Exception in thread "main" javax.net.ssl.SSLException:
No available certificate corresponding to the SSL cipher suites which are enabled...
```

Cause: Various cipher suites require certain types of key material. For example, if an RSA cipher suite is enabled, then an RSA `keyEntry` must be available in the keystore. If no such key is available, then this cipher suite cannot be used. This exception is thrown if there are no available key entries for all of the cipher suites enabled.

Solution: Create key entries for the various cipher suite types, or use an anonymous suite. Anonymous cipher suites are inherently dangerous because they are vulnerable to MITM (man-in-the-middle) attacks. For more information, see RFC 2246.

Refer to the following sections to learn how to pass the correct keystore and certificates:

- JSSE Classes and Interfaces
- Customizing the Default Keystores and Truststores, Store Types, and Store Passwords
- Additional Keystore Formats

Runtime Exception: No Cipher Suites in Common

Problem 1: When handshaking, the client and/or server throw this exception.

Cause 1: Both sides of an SSL connection must agree on a common cipher suite. If the intersection of the client's cipher suite set with the server's cipher suite set is empty, then you will see this exception.

Solution 1: Configure the enabled cipher suites to include common cipher suites, and be sure to provide an appropriate `keyEntry` for asymmetric cipher suites. Also see Runtime Exception: "No available certificate..." in this section.)

Problem 2: When using Netscape Navigator or Microsoft Internet Explorer to access files on a server that only has DSA-based certificates, a runtime exception occurs indicating that there are no cipher suites in common.

Cause 2: By default, `keyEntries` created with `keytool` use DSA public keys. If only DSA `keyEntries` exist in the keystore, then only DSA-based cipher suites can be used. By default, Navigator and Internet Explorer send only RSA-based cipher suites. Because the intersection of client and server cipher suite sets is empty, this exception is thrown.

Solution 2: To interact with Navigator or Internet Explorer, you should create certificates that use RSA-based keys. To do this, specify the `-keyalg` RSA option when using keytool. For example:

```
keytool -genkeypair -alias duke -keystore testkeys -keyalg rsa
```

Slowness of the First JSSE Access

Problem: JSSE seems to stall on first access.

Cause: JSSE must have a secure source of random numbers. The initialization takes a while.

Solution: Provide an alternative generator of random numbers, or initialize ahead of time when the overhead will not be noticed:

```
SecureRandom sr = new SecureRandom();
sr.nextInt();
SSLContext.init(..., ..., sr);
```

The `java-home/lib/security/java.security` file also provides a way to specify the source of seed data for `SecureRandom`. See the contents of the file for more information.

Code Using HttpsURLConnection Class Throws ClassCastException in JSSE 1.0.x

Problem: The following code snippet was written using `com.sun.net.ssl.HttpsURLConnection` in JSSE 1.0.x:

```
import com.sun.net.ssl.*;
...deleted...
HttpsURLConnection urlc = new URL("https://example.com/").openConnection();
```

When running under JSSE 1.0.x, this code returns a `javax.net.ssl.HttpsURLConnection` object and throws a `ClassCastException`.

Cause: By default, opening an HTTPS URL will create a `javax.net.ssl.HttpsURLConnection`.

Solution: Previous releases of the JDK (release 6 and earlier) did not ship with an HTTPS URL implementation. The JSSE 1.0.x implementation did provide such an HTTPS URL handler, and the installation guide described how to set the URL handler search path to obtain a JSSE 1.0.x `com.sun.net.ssl.HttpsURLConnection` implementation.

In the JDK, there is an HTTPS handler in the default URL handler search path. It returns an instance of `javax.net.ssl.HttpsURLConnection`. By prepending the old JSSE 1.0.x implementation path to the URL search path via the `java.protocol.handler.pkgs` variable, you can still obtain a `com.sun.net.ssl.HttpsURLConnection`, and the code will no longer throw cast exceptions.

See the following examples:

```
% java -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol YourClass

System.setProperty("java.protocol.handler.pkgs", "com.sun.net.ssl.internal.www.protocol");
```

Socket Disconnected After Sending ClientHello Message

Problem: A socket attempts to connect, sends a ClientHello message, and is immediately disconnected.

Cause: Some SSL/TLS servers will disconnect if a ClientHello message is received in a format they do not understand or with a protocol version number that they do not support.

Solution: Try adjusting the enabled protocols on the client side. This involves modifying or invoking some of the following system properties and methods:

- System property `https.protocols` for the `HttpsURLConnection` class
- System property `jdk.tls.client.protocols`
- `SSLContext.getInstance` method
- `SSLEngine.setEnabledProtocols` method
- `SSLSocket.setEnabledProtocols` method
- `SSLParameters.setProtocols` and `SSLEngine.setSSLParameters` methods
- `SSLParameters.setProtocols` and `SSLSocket.setSSLParameters` methods

For backwards compatibility, some SSL/TLS implementations (such as `SunJSSE`) can send SSL/TLS ClientHello messages encapsulated in the SSLv2 ClientHello format. The `SunJSSE` provider supports this feature. If you want to use this feature, add the "SSLv2Hello" protocol to the enabled protocol list, if necessary. (Also see the Protocols section, which lists the protocols that are enabled by default for the `SunJSSE` provider.)

The SSL/TLS RFC standards require that implementations negotiate to the latest version both sides speak, but some non-conforming implementation simply hang up if presented with a version they don't understand. For example, some older server implementations that speak only SSLv3 will shutdown if TLSv1.2 is requested. In this situation, consider using a SSL/TLS version fallback scheme:

1. Fall back from TLSv1.2 to TLSv1.1 if the server does not understand TLSv1.2.
2. Fall back from TLSv1.1 to TLSv1.0 if the previous step does not work.

For example, if the enabled protocol list on the client is TLSv1, TLSv1.1, and TLSv1.2, a typical SSL/TLS version fallback scheme may look like:

1. Try to connect to server. If server rejects the SSL/TLS connection request immediately, go to step 2.
2. Try the version fallback scheme by removing the highest protocol version (for example, TLSv1.2 for the first failure) in the enabled protocol list.
3. Try to connect to the server again. If server rejects the connection, go to step 2 unless there is no version to which the server can fall back.
4. If the connection fails and SSLv2Hello is not on the enabled protocol list, restore the enable protocol list and enable SSLv2Hello. (For example, the enable protocol list should be SSLv2Hello, SSLv3, TLSv1, TLSv1.1, and TLSv1.2.) Start again from step 1.

Note: A fallback to a previous version normally means security strength downgrading to a weaker protocol. It is not suggested to use a fallback scheme unless it is really necessary, and you clearly know that the server does not support a higher protocol version.

Note: As part of disabling SSLv3, some servers have also disabled SSLv2Hello, which means communications with SSLv2Hello-active clients (e.g. JDK 1.5/6) will fail. Starting with JDK 7, SSLv2Hello default to disabled on clients, enabled on servers.

SunJSSE Cannot Find a JCA Provider That Supports a Required Algorithm and Causes a NoSuchAlgorithmException

Problem: A handshake is attempted and fails when it cannot find a required algorithm. Examples might include:

```
Exception in thread ...deleted...
...deleted...
Caused by java.security.NoSuchAlgorithmException: Cannot find any
provider supporting RSA/ECB/PKCS1Padding
```

or

```
Caused by java.security.NoSuchAlgorithmException: Cannot find any
provider supporting AES/CBC/NoPadding
```

Cause: `SunJSSE` uses JCE for all its cryptographic algorithms. By default, the Oracle JDK will use the Standard Extension ClassLoader to load the SunJCE provider located in `java-home/lib/ext/sunjce_provider.jar`. If the file cannot be found or loaded, or if the SunJCE provider has been deregistered from the `Provider` mechanism and an alternative implementation from JCE is not available, then this exception will be thrown.

Solution: Ensure that the SunJCE is available by checking that the file is loadable and that the provider is registered with the `Provider` interface. Try to run the following code in the context of your SSL connection:

```
import javax.crypto.*;

System.out.println("=====Where did you get AES=====");
Cipher c = Cipher.getInstance("AES/CBC/NoPadding");
System.out.println(c.getProvider());
```

FailedDownloadException Thrown When Trying to Obtain Application Resources from Web Server over SSL

Problem: If you receive a `com.sun.deploy.net.FailedDownloadException` when trying to obtain application resources from your web server over SSL, and your web server uses the virtual host with Server Name Indication (SNI) extension (such as Apache HTTP Server), then you may have not configured your web server correctly.

Cause: Because Java SE 7 supports the SNI extension in the JSSE client, the requested host name of the virtual server is included in the first message sent from the client to the server during the SSL handshake. The server may deny the client's request for a connection if the requested host name (the server name indication) does not match the expected server name, which should be specified in the virtual host's configuration. This triggers an SSL handshake unrecognized name alert, which results in a `FailedDownloadException` being thrown.

Solution: To better diagnose the problem, enable tracing through the Java Console. See Java Console, Tracing, and Logging for more information. If the cause of the problem is `javax.net.ssl.SSLProtocolException: handshake alert: unrecognized_name`, it is likely that the virtual host configuration for SNI is incorrect. If you are using Apache HTTP Server, see Name-based Virtual Host Support for information about configuring virtual hosts. In particular, ensure that the `ServerName` directive is configured properly in a `<VirtualHost>` block.

For more information, see the following:

- SSL with Virtual Hosts Using SNI from Apache HTTP Server Wiki
- SSL/TLS Strong Encryption: FAQ from Apache HTTP Server Documentation
- RFC 3546, Transport Layer Security (TLS) Extensions
- Bug 7194590: SSL handshaking error caused by virtual server misconfiguration

Debugging Utilities

JSSE provides dynamic debug tracing support. This is similar to the support used for debugging access control failures in the Java SE platform. The generic Java dynamic debug tracing support is accessed with the `java.security.debug` system property, whereas the JSSE-specific dynamic debug tracing support is accessed with the `javax.net.debug` system property.

Note: The `debug` utility is not an officially supported feature of JSSE.

To view the options of the JSSE dynamic debug utility, use the following command-line option on the `java` command:

```
-Djavax.net.debug=help
```

Note: If you specify the value `help` with either dynamic debug utility when running a program that does not use any classes that the utility was designed to debug, you will not get the debugging options.

The following complete example shows how to get a list of the debug options for an application named `MyApp` that uses some of the JSSE classes:

```
java -Djavax.net.debug=help MyApp
```

The `MyApp` application will not run after the debug help information is printed, as the help code causes the application to exit.

Current options are:

- `all`: Turn on all debugging
- `ssl`: Turn on SSL debugging

The following can be used with the `ssl` option:

- `record`: Enable per-record tracing
- `handshake`: Print each handshake message
- `keygen`: Print key generation data
- `session`: Print session activity
- `defaultctx`: Print default SSL initialization
- `sslctx`: Print `SSLContext` tracing
- `sessioncache`: Print session cache tracing
- `keymanager`: Print key manager tracing
- `trustmanager`: Print trust manager tracing

Messages generated from the `handshake` option can be widened with these options:

- `data`: Hex dump of each handshake message
- `verbose`: Verbose handshake message printing

Messages generated from the `record` option can be widened with these options:

- `plaintext`: Hex dump of record plaintext
- `packet`: Print raw SSL/TLS packets

The `javax.net.debug` property value must be either `all` or `ssl`, optionally followed by debug specifiers. You can use one or more options. You do not have to have a separator between options, although a separator such as a colon (:) or a comma (,) helps readability. It does not matter what separators you use, and the ordering of the option keywords is also not important.

For an introduction to reading this debug information, see the guide, Debugging SSL/TLS Connections.

The following are examples of using the `javax.net.debug` property:

- To view all debugging messages:

  ```
  java -Djavax.net.debug=all MyApp
  ```

- To view the hexadecimal dumps of each handshake message, enter the following (the colons are optional):

  ```
  java -Djavax.net.debug=ssl:handshake:data MyApp
  ```

- To view the hexadecimal dumps of each handshake message, and to print trust manager tracing, enter the following (the commas are optional):

  ```
  java -Djavax.net.debug=SSL,handshake,data,trustmanager MyApp
  ```

Code Examples

The following code examples are included in this section:

- Converting an Unsecure Socket to a Secure Socket
  - Socket Example Without SSL
  - Socket Example With SSL
- Running the JSSE Sample Code
  - Sample Code Illustrating a Secure Socket Connection Between a Client and a Server

Converting an Unsecure Socket to a Secure Socket

This section provides examples of source code that illustrate how to use JSSE to convert an unsecure socket connection to a secure socket connection. The code in this section is excerpted from the book Java SE 6 Network Security by Marco Pistoia, et. al.

First, "Socket Example Without SSL" shows sample code that can be used to set up communication between a client and a server using unsecure sockets. This code is then modified in "Socket Example with SSL" to use JSSE to set up secure socket communication.

Socket Example Without SSL

The following examples demonstrates server-side and client-side code for setting up an unsecure socket connection.

In a Java program that acts as a server and communicates with a client using sockets, the socket communication is set up with code similar to the following:

```
import java.io.*;
import java.net.*;

. . .

int port = availablePortNumber;

ServerSocket s;

try {
s = new ServerSocket(port);
Socket c = s.accept();

OutputStream out = c.getOutputStream();
InputStream in = c.getInputStream();

// Send messages to the client through
// the OutputStream
// Receive messages from the client
// through the InputStream
} catch (IOException e) { }
```

The client code to set up communication with a server using sockets is similar to the following:

```
import java.io.*;
import java.net.*;

. . .

int port = availablePortNumber;
String host = "hostname";

try {
s = new Socket(host, port);

OutputStream out = s.getOutputStream();
InputStream in = s.getInputStream();

// Send messages to the server through
// the OutputStream
// Receive messages from the server
```

```
    // through the InputStream
    } catch (IOException e) { }
```

Socket Example with SSL

The following examples demonstrate server-side and client-side code for setting up a secure socket connection.

In a Java program that acts as a server and communicates with a client using secure sockets, the socket communication is set up with code similar to the following. Differences between this program and the one for communication using unsecure sockets are highlighted in bold.

```
import java.io.*;
import javax.net.ssl.*;

. . .

int port = availablePortNumber;

SSLServerSocket s;

try {
SSLServerSocketFactory sslSrvFact =
(SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
s = (SSLServerSocket)sslSrvFact.createServerSocket(port);

SSLSocket c = (SSLSocket)s.accept();

OutputStream out = c.getOutputStream();
InputStream in = c.getInputStream();

// Send messages to the client through
// the OutputStream
// Receive messages from the client
// through the InputStream
}

catch (IOException e) {
}
```

The client code to set up communication with a server using secure sockets is similar to the following, where differences with the unsecure version are highlighted in bold:

```
import java.io.*;
import javax.net.ssl.*;

. . .

int port = availablePortNumber;
String host = "hostname";

try {
SSLSocketFactory sslFact =
(SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket s = (SSLSocket)sslFact.createSocket(host, port);

OutputStream out = s.getOutputStream();
InputStream in = s.getInputStream();

// Send messages to the server through
// the OutputStream
// Receive messages from the server
// through the InputStream
}

catch (IOException e) {
}
```

Running the JSSE Sample Code

The JSSE sample programs illustrate how to use JSSE to:

- Create a secure socket connection between a client and a server
- Create a secure connection to an HTTPS website
- Use secure communications with RMI
- Illustrate SSLEngine usage

When you use the sample code, be aware that the sample programs are designed to illustrate how to use JSSE. They are not designed to be robust applications.

Note: Setting up secure communications involves complex algorithms. The sample programs provide no feedback during the setup process. When you run the programs, be patient: you may not see any output for a while. If you run the programs with the `javax.net.debug` system property set to `all`, you will see more feedback. For an introduction to reading this debug information, see the guide, Debugging SSL/TLS Connections.

Where to Find the Sample Code

Most of the sample code is located in the samples subdirectory of the same directory as that containing the document you are reading. Follow that link to see a listing of all the sample code files and text files. That page also provides a link to a ZIP file that you can download to obtain all the sample code files, which is helpful if you are viewing this documentation from the web.

The following sections describe the samples. For more information, see README.txt.

Sample Code Illustrating a Secure Socket Connection Between a Client and a Server

The sample programs in the `samples/sockets` directory illustrate how to set up a secure socket connection between a client and a server.

When running the sample client programs, you can communicate with an existing server, such as a commercial web server, or you can communicate with the sample server program, `ClassFileServer`. You can run the sample client and the sample server programs on different machines connected to the same network, or you can run them both on one machine but from different terminal windows.

All the sample `SSLSocketClient*` programs in the samples/sockets/client directory (and `URLReader*` programs described in Sample Code Illustrating HTTPS Connections) can be run with the `ClassFileServer` sample server program. An example of how to do this is shown in Running SSLSocketClientWithClientAuth with ClassFileServer. You can make similar changes to run `URLReader`, `SSLSocketClient`, or `SSLSocketClientWithTunneling` with `ClassFileServer`.

If an authentication error occurs during communication between the client and the server (whether using a web server or `ClassFileServer`), it is most likely because the necessary keys are not in the truststore (trust key database). For example, the `ClassFileServer` uses a keystore called `testkeys` containing the private key for `localhost` as needed during the SSL handshake. The `testkeys` keystore is included in the same samples/sockets/server directory as the `ClassFileServer` source. If the client cannot find a certificate for the corresponding public key of `localhost` in the truststore it consults, then an authentication error will occur. Be sure to use the `samplecacerts` truststore (which contains the public key and certificate of the `localhost`), as described in the next section.

Configuration Requirements

When running the sample programs that create a secure socket connection between a client and a server, you will need to make the appropriate certificates file (truststore) available. For both the client and the server programs, you should use the certificates file `samplecacerts` from the `samples` directory. Using this certificates file will allow the client to authenticate the server. The file contains all the common Certificate Authority (CA) certificates shipped with the JDK (in the cacerts file), plus a certificate for `localhost` needed by the client to authenticate `localhost` when communicating with the sample server `ClassFileServer`. The `ClassFileServer` uses a keystore containing the private key for `localhost` that corresponds to the public key in `samplecacerts`.

To make the `samplecacerts` file available to both the client and the server, you can either copy it to the file `java-home/lib/security/jssecacerts`, rename it to cacerts, and use it to replace the `java-home/lib/security/cacerts` file, or add the following option to the command line when running the `java` command for both the client and the server:

```
-Djavax.net.ssl.trustStore=path_to_samplecacerts_file
```

The password for the `samplecacerts` truststore is `changeit`. You can substitute your own certificates in the samples by using the `keytool` utility.

If you use a browser, such as Netscape Navigator or Microsoft's Internet Explorer, to access the sample SSL server provided in the `ClassFileServer` example, then a dialog box may pop up with the message that it does not recognize the certificate. This is normal because the certificate used with the sample programs is self-signed and is for testing only. You can accept the certificate for the current session. After testing the SSL server, you should exit the browser, which deletes the test certificate from the browser's namespace.

For client authentication, a separate `duke` certificate is available in the appropriate directories. The public key and certificate is also stored in the `samplecacerts` file.

Running SSLSocketClient

The SSLSocketClient.java program demonstrates how to create a client that uses an `SSLSocket` to send an HTTP request and to get a response from an HTTPS server. The output of this program is the HTML source for `https://www.verisign.com/index.html`.

You must not be behind a firewall to run this program as provided. If you run it from behind a firewall, you will get an `UnknownHostException` because JSSE cannot find a path through your firewall to `www.verisign.com`. To create an equivalent client that can run from behind a firewall, set up proxy tunneling as illustrated in the sample program `SSLSocketClientWithTunneling`.

Running SSLSocketClientWithTunneling

The SSLSocketClientWithTunneling.java program illustrates how to do proxy tunneling to access a secure web server from behind a firewall. To run this program, you must set the following Java system properties to the appropriate values:

```
java -Dhttps.proxyHost=webproxy
-Dhttps.proxyPort=ProxyPortNumber
SSLSocketClientWithTunneling
```

Note: Proxy specifications with the `-D` options are optional. Replace webproxy with the name of your proxy host and ProxyPortNumber with the appropriate port number.

The program will return the HTML source file from `https://www.verisign.com/index.html`.

Running SSLSocketClientWithClientAuth

The SSLSocketClientWithClientAuth.java program shows how to set up a key manager to do client authentication if required by a server. This program also assumes that the client is not outside a firewall. You can modify the program to connect from inside a firewall by following the example in `SSLSocketClientWithTunneling`.

To run this program, you must specify three parameters: host, port, and requested file path. To mirror the previous examples, you can run this program without client authentication by setting the host to `www.verisign.com`, the port to `443`, and the requested file path to `https://www.verisign.com/`. The output when using these parameters is the HTML for the website `https://www.verisign.com/`.

To run `SSLSocketClientWithClientAuth` to do client authentication, you must access a server that requests client authentication. You can use the sample program `ClassFileServer` as this server. This is described in the following sections.

Running ClassFileServer

The program referred to herein as `ClassFileServer` is made up of two files: ClassFileServer.java and ClassServer.java.

To execute them, run `ClassFileServer.class`, which requires the following parameters:

- `port` can be any available unused port number, for example, you can use the number `2001`.
- `docroot` indicates the directory on the server that contains the file you want to retrieve. For example, on Solaris, you can use /home/userid/ (where userid refers to your particular UID), whereas on Microsoft Windows systems, you can use c:\.
- `TLS` is an optional parameter that indicates that the server is to use SSL or TLS.
- `true` is an optional parameter that indicates that client authentication is required. This parameter is only consulted if the TLS parameter is set.

Note: The `TLS` and `true` parameters are optional. If you omit them, indicating that an ordinary (not TLS) file server should be used, without authentication, then nothing happens. This is because one side (the client) is trying to negotiate with TLS, while the other (the server) is not, so they cannot communicate.

Note: The server expects GET requests in the form `GET /path_to_file`.

Running SSLSocketClientWithClientAuth with ClassFileServer

You can use the sample programs SSLSocketClientWithClientAuth and `ClassFileServer` to set up authenticated communication, where the client and server are authenticated to each other. You can run both sample programs on different machines connected to the same network, or you can run them both on one machine but from different terminal windows or command prompt windows. To set up both the client and the server, do the following:

1. Run the program `ClassFileServer` from one machine or terminal window, as described in Running ClassFileServer.
2. Run the program `SSLSocketClientWithClientAuth` on another machine or terminal window. `SSLSocketClientWithClientAuth` requires the following parameters:
   - `host` is the host name of the machine that you are using to run `ClassFileServer`.
   - `port` is the same port that you specified for `ClassFileServer`.
   - `requestedfilepath` indicates the path to the file that you want to retrieve from the server. You must give this parameter as `/filepath`. Forward slashes are required in the file path because it is used as part of a GET statement, which requires forward slashes regardless of what type of operating system you are running. The statement is formed as follows:

     ```
     "GET " + requestedfilepath + " HTTP/1.0"
     ```

Note: You can modify the other `SSLClient*` applications' `GET` commands to connect to a local machine running `ClassFileServer`.

Sample Code Illustrating HTTPS Connections

There are two primary APIs for accessing secure communications through JSSE. One way is through a socket-level API that can be used for arbitrary secure communications, as illustrated by the `SSLSocketClient`, `SSLSocketClientWithTunneling`, and `SSLSocketClientWithClientAuth` (with and without `ClassFileServer`) sample programs.

A second, and often simpler, way is through the standard Java URL API. You can communicate securely with an SSL-enabled web server by using the HTTPS URL protocol or scheme using the `java.net.URL` class.

Support for HTTPS URL schemes is implemented in many of the common browsers, which allows access to secured communications without requiring the socket-level API provided with JSSE.

An example URL is `https://www.verisign.com`.

The trust and key management for the HTTPS URL implementation is environment-specific. The JSSE implementation provides an HTTPS URL implementation. To use a different HTTPS protocol implementation, set the `java.protocol.handler.pkgs` system property to the package name. See the `java.net.URL` class documentation for details.

The samples that you can download with JSSE include two sample programs that illustrate how to create an HTTPS connection. Both of these sample programs (`URLReader.java` and `URLReaderWithOptions.java`) are in the samples/urls directory.

Running URLReader

The URLReader.java program illustrates using the URL class to access a secure site. The output of this program is the HTML source for `https://www.verisign.com/`. By default, the HTTPS protocol implementation included with JSSE is used. To use a different implementation, set the system property `java.protocol.handler.pkgs` value to be the name of the package containing the implementation.

If you are running the sample code behind a firewall, then you must set the `https.proxyHost` and `https.proxyPort` system properties. For example, to use the proxy host "webproxy" on port 8080, you can use the following options for the `java` command:

```
-Dhttps.proxyHost=webproxy
-Dhttps.proxyPort=8080
```

Alternatively, you can set the system properties within the source code with the `java.lang.System` method `setProperty()`. For example, instead of using the command-line options, you can include the following lines in your program:

```
System.setProperty("java.protocol.handler.pkgs", "com.ABC.myhttpsprotocol");

System.setProperty("https.proxyHost", "webproxy");

System.setProperty("https.proxyPort", "8080");
```

Running URLReaderWithOptions

The URLReaderWithOptions.java program is essentially the same as the URLReader.java program, except that it allows you to optionally input any or all of the following system properties as arguments to the program when you run it:

- `java.protocol.handler.pkgs`
- `https.proxyHost`
- `https.proxyPort`
- `https.cipherSuites`

To run `URLReaderWithOptions`, enter the following command:

```
java URLReaderWithOptions [-h proxyhost -p proxyport] [-k protocolhandlerpkgs] [-c ciphersarray]
```

Note: Multiple protocol handlers can be included in the `protocolhandlerpkgs` argument as a list with items separated by vertical bars. Multiple SSL cipher suite names can be included in the `ciphersarray` argument as a list with items separated by commas. The possible cipher suite names are the same as those returned by the `SSLSocket.getSupportedCipherSuites()` method. The suite names are taken from the SSL and TLS protocol specifications.

You need a `protocolhandlerpkgs` argument only if you want to use an HTTPS protocol handler implementation other than the default one provided by Oracle.

If you are running the sample code behind a firewall, then you must include arguments for the proxy host and the proxy port. Additionally, you can include a list of cipher suites to enable.

Here is an example of running `URLReaderWithOptions` and specifying the proxy host "webproxy" on port 8080:

```
java URLReaderWithOptions -h webproxy -p 8080
```

Sample Code Illustrating a Secure RMI Connection

The sample code in the samples/rmi directory illustrates how to create a secure Java Remote Method Invocation (RMI) connection. The sample code is based on an RMI example that is basically a "Hello World" example modified to install and use a custom RMI socket factory.

For more information about Java RMI, see the Java RMI documentation. This web page points to Java RMI tutorials and other information about Java RMI.

Sample Code Illustrating the Use of an SSLEngine

`SSLEngine` gives application developers flexibility when choosing I/O and compute strategies. Rather than tie the SSL/TLS implementation to a specific I/O abstraction (such as single-threaded `SSLSockets`), `SSLEngine` removes the I/O and compute constraints from the SSL/TLS implementation.

As mentioned earlier, `SSLEngine` is an advanced API, and is not appropriate for casual use. Some introductory sample code is provided here that helps illustrate its use. The first demo removes most of the I/O and threading issues, and focuses on many of the SSLEngine methods. The second demo is a more realistic example showing how `SSLEngine` might be combined with Java NIO to create a rudimentary HTTP/HTTPS server.

Running SSLEngineSimpleDemo

The SSLEngineSimpleDemo is a very simple application that focuses on the operation of the `SSLEngine` while simplifying the I/O and threading issues. This application creates two `SSLEngine` objects that exchange SSL/TLS messages via common `ByteBuffer` objects. A single loop serially performs all of the engine operations and demonstrates how a secure connection is established (handshaking), how application data is transferred, and how the engine is closed.

The `SSLEngineResult` provides a great deal of information about the current state of the `SSLEngine`. This example does not examine all of the states. It simplifies the I/O and threading issues to the point that this is not a good example for a production environment; nonetheless, it is useful to demonstrate the overall function of the `SSLEngine`.

Creating a Keystore to Use with JSSE

This section demonstrates how you can use the `keytool` utility to create a simple JKS keystore suitable for use with JSSE. First you make a `keyEntry` (with public and private keys) in the keystore, and then you make a corresponding `trustedCertEntry` (public keys only) in a truststore. For client authentication, you follow a similar process for the client's certificates.

Note: Storing trust anchors and secret keys in PKCS12 is supported since JDK 8.

Note: It is beyond the scope of this example to explain each step in detail. For more information, see the `keytool` documentation for Solaris, Linux, or macOS or Microsoft Windows.

User input is shown in bold.

1. Create a new keystore and self-signed certificate with corresponding public and private keys.

```
% keytool -genkeypair -alias duke -keyalg RSA -validity 7 -keystore keystore

Enter keystore password:  password
What is your first and last name?
[Unknown]:  Duke
What is the name of your organizational unit?
[Unknown]:  Java Software
What is the name of your organization?
[Unknown]:  Oracle, Inc.
What is the name of your City or Locality?
[Unknown]:  Palo Alto
What is the name of your State or Province?
[Unknown]:  CA
What is the two-letter country code for this unit?
[Unknown]:  US
Is CN=Duke, OU=Java Software, O="Oracle, Inc.",
L=Palo Alto, ST=CA, C=US correct?
[no]:  yes

Enter key password for <duke>
(RETURN if same as keystore password):  <CR>
```

2. Examine the keystore. Notice that the entry type is `keyEntry`, which means that this entry has a private key associated with it).

```
% keytool -list -v -keystore keystore

Enter keystore password:  password

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry
```

```
Alias name: duke
Creation date: Dec 20, 2001
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Duke, OU=Java Software, O="Oracle, Inc.",
L=Palo Alto, ST=CA, C=US
Issuer: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto, ST=CA, C=US
Serial number: 3c22adc1
Valid from: Thu Dec 20 19:34:25 PST 2001 until: Thu Dec 27 19:34:25 PST 2001
Certificate fingerprints:
MD5: F1:5B:9B:A1:F7:16:CF:25:CF:F4:FF:35:3F:4C:9C:F0
SHA1: B2:00:50:DD:B6:CC:35:66:21:45:0F:96:AA:AF:6A:3D:E4:03:7C:74
```

3. Export and examine the self-signed certificate.

```
% keytool -export -alias duke -keystore keystore -rfc -file duke.cer
Enter keystore password:  password
Certificate stored in file <duke.cer>
% cat duke.cer
-----BEGIN CERTIFICATE-----
MIICXjCCAccCBDwircEwDQYJKoZIhvcNAQEEBQAwdjELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAkNB
MRIwEAYDVQQHEwlQYWxvIEFsdG8xHzAdBgNVBAoTFlN1biBNaWNyb3N5c3RlbXMsIEluYy4xFjAU
BgNVBAsTDUphdmEgU29mdHdhcmUxDTALBgNVBAMTBER1a2UwHhcNMDExMjIxMDMzNDI1WhcNMDEx
MjI4MDMzNDI1WjB2MQswCQYDVQQGEwJVUzELMAkGA1UECBMCQ0ExEjAQBgNVBAcTCVBhbG8gQWx0
bzEfMB0GA1UEChMWU3VuIE1pY3Jvc3lzdGVtcywgSW5jLjEWMBQGA1UECxMNSmF2YSBTb2Z0d2Fy
ZTENMAsGA1UEAxMERHVrZTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA1loObJzNXsi5aSr8
N4XzDksD6GjTHFeqG9DUFXKEOQetfYXvA8F9uWtz8WInrqskLTNzwXgmNeWkoM7mrPpK6Rf5M3G1
NXtYzvxyi473Gh1h9k7tjJvqSVKO7E1oFkQYeUPYifxmjbSMVirWZgvo2UmA1c76oNK+NhoHJ4qj
eCUCAwEAATANBgkqhkiG9w0BAQQFAAOBgQCRPoQYw9rWWvfLPQuPXowvFmuebsTc28qI7iFWm6BJ
TT/qdmzti7B5MHOt9BeVEft3mMeBUOCS2guaBjDpGlf+zsK/UUi1w9C4mnwGDZzqY/NKKWtLxabZ
5M+4MAKLZ92ePPKGpobM2CPLfM8ap4IgAzCbBKd8+CMp8yFmifze9Q==
-----END CERTIFICATE-----
```

Alternatively, you could generate a Certificate Signing Request (CSR) with `-certreq` and send that to a Certificate Authority (CA) for signing, but that is beyond the scope of this example.

4. Import the certificate into a new truststore.

```
% keytool -import -alias dukecert -file duke.cer -keystore truststore
Enter keystore password:  trustword
Owner: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto, ST=CA, C=US
Issuer: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto, ST=CA, C=US
Serial number: 3c22adc1
Valid from: Thu Dec 20 19:34:25 PST 2001 until: Thu Dec 27 19:34:25 PST 2001
Certificate fingerprints:
MD5: F1:5B:9B:A1:F7:16:CF:25:CF:F4:FF:35:3F:4C:9C:F0
SHA1: B2:00:50:DD:B6:CC:35:66:21:45:0F:96:AA:AF:6A:3D:E4:03:7C:74
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

5. Examine the truststore. Note that the entry type is `trustedCertEntry`, which means that a private key is not available for this entry. It also means that this file is not suitable as a keystore of the `KeyManager`.

```
% keytool -list -v -keystore truststore
Enter keystore password:  trustword

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: dukecert
Creation date: Dec 20, 2001
Entry type: trustedCertEntry

Owner: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto, ST=CA, C=US
Issuer: CN=Duke, OU=Java Software, O="Oracle, Inc.", L=Palo Alto, ST=CA, C=US
Serial number: 3c22adc1
Valid from: Thu Dec 20 19:34:25 PST 2001 until: Thu Dec 27 19:34:25 PST 2001
Certificate fingerprints:
MD5: F1:5B:9B:A1:F7:16:CF:25:CF:F4:FF:35:3F:4C:9C:F0
SHA1: B2:00:50:DD:B6:CC:35:66:21:45:0F:96:AA:AF:6A:3D:E4:03:7C:74
```

6. Now run your applications with the appropriate keystores. Because this example assumes that the default `X509KeyManager` and `X509TrustManager` are used, you select the keystores using the system properties described in Customizing JSSE.

```
% java -Djavax.net.ssl.keyStore=keystore -Djavax.net.ssl.keyStorePassword=password Server

% java -Djavax.net.ssl.trustStore=truststore -Djavax.net.ssl.trustStorePassword=trustword Client
```

Note: This example authenticated the server only. For client authentication, provide a similar keystore for the client's keys and an appropriate truststore for the server.

Using the Server Name Indication (SNI) Extension

This section provides code examples that illustrate how you can use the Server Name Indication (SNI) extension for client-side and server-side applications, and how it can be applied to a virtual infrastructure.

For all examples in this section, to apply the parameters after you set them, call the `setSSLParameters(SSLParameters)` method on the corresponding `SSLSocket`, `SSLEngine`, or `SSLServerSocket` object.

Typical Client-Side Usage Examples

The following is a list of use cases that require understanding of the SNI extension for developing a client application:

- **Case 1. The client wants to access `www.example.com`.**

  Set the host name explicitly:

  ```
  SNIHostName serverName = new SNIHostName("www.example.com");
  List<SNIServerName> serverNames = new ArrayList<>(1);
  serverNames.add(serverName);
  sslParameters.setServerNames(serverNames);
  ```

  The client should always specify the host name explicitly.

- **Case 2. The client does not want to use SNI because the server does not support it.**

  Disable SNI with an empty server name list:

  ```
  List<SNIServerName> serverNames = new ArrayList<>(1);
  sslParameters.setServerNames(serverNames);
  ```

- **Case 3. The client wants to access URL `https://www.example.com`.**

  Oracle providers will set the host name in the SNI extension by default, but third-party providers may not support the default server name indication. To keep your application provider-independent, always set the host name explicitly.

- **Case 4. The client wants to switch a socket from server mode to client mode.**

  First switch the mode with the following method: `sslSocket.setUseClientMode(true)`. Then reset the server name indication parameters on the socket.

Typical Server-Side Usage Examples

The following is a list of use cases that require understanding of the SNI extension for developing a server application:

- **Case 1. The server wants to accept all server name indication types.**

  If you do not have any code dealing with the SNI extension, then the server ignores all server name indication types.

- **Case 2. The server wants to deny all server name indications of type `host_name`.**

  Set an invalid server name pattern for `host_name`:

  ```
  SNIMatcher matcher = SNIHostName.createSNIMatcher("");
  Collection<SNIMatcher> matchers = new ArrayList<>(1);
  matchers.add(matcher);
  sslParameters.setSNIMatchers(matchers);
  ```

  Another way is to create an `SNIMatcher` subclass with a `matches()` method that always returns `false`:

  ```
  class DenialSNIMatcher extends SNIMatcher {
  DenialSNIMatcher() {
  super(StandardConstants.SNI_HOST_NAME);
  }

  @Override
  public boolean matches(SNIServerName serverName) {
  return false;
  }
  }

  SNIMatcher matcher = new DenialSNIMatcher();
  Collection<SNIMatcher> matchers = new ArrayList<>(1);
  matchers.add(matcher);
  ```

```
sslParameters.setSNIMatchers(matchers);
```

- **Case 3. The server wants to accept connections to any host names in the `example.com` domain.**

  Set the recognizable server name for `host_name` as a pattern that includes all `*.example.com` addresses:

  ```
  SNIMatcher matcher = SNIHostName.createSNIMatcher("(.*\\.)*example\\.com");
  Collection<SNIMatcher> matchers = new ArrayList<>(1);
  matchers.add(matcher);
  sslParameters.setSNIMatchers(matchers);
  ```

- **Case 4. The server wants to switch a socket from client mode to server mode.**

  First switch the mode with the following method: `sslSocket.setUseClientMode(false)`. Then reset the server name indication parameters on the socket.

Working with Virtual Infrastructures

This section describes how to use the Server Name Indication (SNI) extension from within a virtual infrastructure. It illustrates how to create a parser for ClientHello messages from a socket, provides examples of virtual server dispatchers using `SSLSocket` and `SSLEngine`, describes what happens when the SNI extension is not available, and demonstrates how to create a failover `SSLContext`.

Preparing the ClientHello Parser

Applications must implement an API to parse the ClientHello messages from a socket. The following examples illustrate the `SSLCapabilities` and `SSLExplorer` classes that can perform these functions.

SSLCapabilities.java encapsulates the SSL/TLS security capabilities during handshaking (that is, the list of cipher suites to be accepted in an SSL/TLS handshake, the record version, the hello version, and the server name indication). It can be retrieved by exploring the network data of an SSL/TLS connection via the `SSLExplorer.explore()` method.

SSLExplorer.java explores the initial ClientHello message from a TLS client, but it does not initiate handshaking or consume network data. The `SSLExplorer.explore()` method parses the ClientHello message, and retrieves the security parameters into `SSLCapabilities`. The method must be called before handshaking occurs on any TLS connections.

Virtual Server Dispatcher Based on SSLSocket

This section describes the procedure for using a virtual server dispatcher based on `SSLSocket`.

1. **Register the server name handler.**

   At this step, the application may create different `SSLContext` objects for different server name indications, or link a certain server name indication to a specified virtual machine or distributed system.

   For example, if the server name is `www.example.org`, then the registered server name handler may be for a local virtual hosting web service. The local virtual hosting web service will use the specified `SSLContext`. If the server name is `www.example.com`, then the registered server name handler may be for a virtual machine hosting on `10.0.0.36`. The handler may map this connection to the virtual machine.

2. **Create a `ServerSocket` and accept the new connection.**

   ```
   ServerSocket serverSocket = new ServerSocket(serverPort);
   Socket socket = serverSocket.accept();
   ```

3. **Read and buffer bytes from the socket input stream, and then explore the buffered bytes.**

   ```
   InputStream ins = socket.getInputStream();

   byte[] buffer = new byte[0xFF];
   int position = 0;
   SSLCapabilities capabilities = null;

   // Read the header of TLS record
   while (position < SSLExplorer.RECORD_HEADER_SIZE) {
   int count = SSLExplorer.RECORD_HEADER_SIZE - position;
   int n = ins.read(buffer, position, count);
   if (n < 0) {
   throw new Exception("unexpected end of stream!");
   }
   position += n;
   }

   // Get the required size to explore the SSL capabilities
   int recordLength = SSLExplorer.getRequiredSize(buffer, 0, position);
   if (buffer.length < recordLength) {
   buffer = Arrays.copyOf(buffer, recordLength);
   ```

```
}

while (position < recordLength) {
int count = recordLength - position;
int n = ins.read(buffer, position, count);
if (n < 0) {
throw new Exception("unexpected end of stream!");
}
position += n;
}

// Explore
capabilities = SSLExplorer.explore(buffer, 0, recordLength);
if (capabilities != null) {
System.out.println("Record version: " + capabilities.getRecordVersion());
System.out.println("Hello version: " + capabilities.getHelloVersion());
}
```

4. **Get the requested server name from the explored capabilities.**

```
List<SNIServerName> serverNames = capabilities.getServerNames();
```

5. **Look for the registered server name handler for this server name indication.**

   If the service of the host name is resident in a virtual machine or another distributed system, then the application must forward the connection to the destination. The application will need to read and write the raw internet data, rather then the SSL application from the socket stream.

```
Socket destinationSocket = new Socket(serverName, 443);

// Forward buffered bytes and network data from the current socket to the destinationSocket.
```

   If the service of the host name is resident in the same process, and the host name service can use the `SSLSocket` directly, then the application will need to set the `SSLSocket` instance to the server:

```
// Get service context from registered handler
// or create the context
SSLContext serviceContext = ...

SSLSocketFactory serviceSocketFac = serviceContext.getSSLSocketFactory();

// wrap the buffered bytes
ByteArrayInputStream bais = new ByteArrayInputStream(buffer, 0, position);
SSLSocket serviceSocket = (SSLSocket)serviceSocketFac.createSocket(socket, bais, true);

// Now the service can use serviceSocket as usual.
```

Virtual Server Dispatcher Based on SSLEngine

This section describes the procedure for using a virtual server dispatcher based on `SSLEngine`.

1. **Register the server name handler.**

   At this step, the application may create different `SSLContext` objects for different server name indications, or link a certain server name indication to a specified virtual machine or distributed system.

   For example, if the server name is `www.example.org`, then the registered server name handler may be for a local virtual hosting web service. The local virtual hosting web service will use the specified `SSLContext`. If the server name is `www.example.com`, then the registered server name handler may be for a virtual machine hosting on `10.0.0.36`. The handler may map this connection to the virtual machine.

2. **Create a `ServerSocket` or `ServerSocketChannel` and accept the new connection.**

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.bind(...);
...
SocketChannel socketChannel = serverSocketChannel.accept();
```

3. **Read and buffer bytes from the socket input stream, and then explore the buffered bytes.**

```
ByteBuffer buffer = ByteBuffer.allocate(0xFF);
SSLCapabilities capabilities = null;
while (true) {
// ensure the capacity
if (buffer.remaining() == 0) {
ByteBuffer oldBuffer = buffer;
buffer = ByteBuffer.allocate(buffer.capacity() + 0xFF);
buffer.put(oldBuffer);
}
int n = sc.read(buffer);
if (n < 0) {
throw new Exception("unexpected end of stream!");
}

int position = buffer.position();
buffer.flip();
capabilities = explorer.explore(buffer);
buffer.rewind();
buffer.position(position);
buffer.limit(buffer.capacity());
if (capabilities != null) {
System.out.println("Record version: " +
capabilities.getRecordVersion());
System.out.println("Hello version: " +
capabilities.getHelloVersion());
break;
}
}
buffer.flip();  // reset the buffer position and limitation
```

4. **Get the requested server name from the explored capabilities.**

```
List<SNIServerName> serverNames = capabilities.getServerNames();
```

5. **Look for the registered server name handler for this server name indication.**

   If the service of the host name is resident in a virtual machine or another distributed system, then the application must forward the connection to the destination. The application will need to read and write the raw internet data, rather then the SSL application from the socket stream.

```
Socket destinationSocket = new Socket(serverName, 443);

// Forward buffered bytes and network data from the current socket to the destinationSocket.
```

   If the service of the host name is resident in the same process, and the host name service can use the `SSLEngine` directly, then the application will simply feed the net data to the `SSLEngine` instance:

```
// Get service context from registered handler
// or create the context
SSLContext serviceContext = ...

SSLEngine serviceEngine = serviceContext.createSSLEngine();

// Now the service can use the buffered bytes and other byte buffer as usual.
```

No SNI Extension Available

If there is no server name indication in a ClientHello message, then there is no way to select the proper service according to SNI. For such cases, the application may need to specify a default service, so that the connection can be delegated to it if there is no server name indication.

Failover SSLContext

The `SSLExplorer.explore()` method does not check the validity of SSL/TLS contents. If the record format does not comply with SSL/TLS specification, or the `explore()` method is invoked after handshaking has started, then the method may throw an `IOException` and be unable to produce network data. In such cases, handle the exception thrown by `SSLExplorer.explore()` by using a failover `SSLContext`, which is not used to negotiate an SSL/TLS connection, but to close the connection with the proper alert message. The following example illustrates a failover `SSLContext`. You can find an example of the `DenialSNIMatcher` class in Case 2 of the Typical Server-Side Usage Examples.

```
byte[] buffer = ...       // buffered network data
boolean failed = true;    // SSLExplorer.explore() throws an exception

SSLContext context = SSLContext.getInstance("TLS");
// the failover SSLContext

context.init(null, null, null);
SSLSocketFactory sslsf = context.getSocketFactory();
ByteArrayInputStream bais = new ByteArrayInputStream(buffer, 0, position);
SSLSocket sslSocket = (SSLSocket)sslsf.createSocket(socket, bais, true);

SNIMatcher matcher = new DenialSNIMatcher();
Collection<SNIMatcher> matchers = new ArrayList<>(1);
matchers.add(matcher);
SSLParameters params = sslSocket.getSSLParameters();
params.setSNIMatchers(matchers);    // no recognizable server name
sslSocket.setSSLParameters(params);

try {
InputStream sslIS = sslSocket.getInputStream();
sslIS.read();
} catch (Exception e) {
System.out.println("Server exception " + e);
} finally {
sslSocket.close();
}
```

Appendix A: Standard Names

The JDK Security API requires and uses a set of standard names for algorithms, certificates and keystore types. The specification names previously found here in Appendix A and in the other security specifications (JCA, CertPath) have been combined in the Standard Names document. Specific provider information can be found in the Oracle Provider Documentation.

Appendix B: Provider Pluggability

JSSE is fully pluggable and does not restrict the use of third-party JSSE providers in any way.

Appendix C: TLS Renegotiation Issue

In the fall of 2009, a flaw was discovered in the SSL/TLS protocols. A fix to the protocol was developed by the IETF TLS Working Group, and current versions of the JDK contain this fix. The page Transport Layer Security (TLS) Renegotiation Issue describes the situation in much more detail, along with interoperability issues when communicating with older implementations that do not contain this protocol fix