

[Skip to Content](#)

- [Oracle Technology Network](#)
- [Software Downloads](#)
- [Documentation](#)

[Search](#)

---

# Java™ Secure Socket Extension (JSSE) Reference Guide

## for Java Platform Standard Edition 7

### [Introduction](#)

- [Features and Benefits](#)
- [JSSE Standard API](#)
- [SunJSSE Provider](#)
- [Related Documentation](#)

### [Terms and Definitions](#)

### [Secure Sockets Layer \(SSL\) Protocol Overview](#)

- [Why Use SSL?](#)
- [How SSL Works](#)

### [Key Classes](#)

- [Relationship Between Classes](#)
- [Core Classes and Interfaces](#)

- [SocketFactory and ServerSocketFactory Classes](#)
- [SSLSocketFactory and SSLServerSocketFactory Classes](#)
- [SSLSocket and SSLServerSocket Classes](#)
- [Nonblocking I/O with `SSL`Engine](#)
- [SSLSession and ExtendedSSLSession Interfaces](#)
- [HttpsURLConnection Class](#)

### [Support Classes and Interfaces](#)

- [SSLContext Class](#)
- [TrustManager Interface](#)
- [TrustManagerFactory Class](#)
- [X509TrustManager Interface](#)
- [X509ExtendedTrustManager Class](#)
- [KeyManager Interface](#)
- [KeyManagerFactory Class](#)

[X509KeyManager Interface](#)  
[X509ExtendedKeyManager Class](#)  
[Relationships between TrustManagers and KeyManagers](#)

## **[Secondary Support Classes and Interfaces](#)**

[SSLParameters Class](#)  
[SSLSessionContext Interface](#)  
[SSLSessionBindingListener Interface](#)  
[SSLSessionBindingEvent Class](#)  
[HandShakeCompletedListener Interface](#)  
[HandShakeCompletedEvent Class](#)  
[HostnameVerifier Interface](#)  
[X509Certificate Class](#)  
[AlgorithmConstraints Interface](#)

## **[Previous \(JSSE 1.0.x\) Implementation Classes and Interfaces](#)**

### **[Customizing JSSE](#)**

[The Installation Directory <java-home>](#)  
[Customization](#)

### **[Transport Layer Security \(TLS\) Renegotiation Issue](#)**

[Introduction](#)  
[Phased Approach to Fixing This Issue](#)  
[Description of Phase 2 Fix](#)  
[Workarounds/Alternatives to SSL/TLS Renegotiation](#)  
[Implementation Details](#)  
[Description of the Phase 1 Fix](#)

### **[JCE and Hardware Acceleration/Smartcard Support](#)**

[Use of JCE](#)  
[Hardware Accelerators](#)  
[Configure JSSE to use Smartcards as Keystore and Trust Stores](#)  
[Multiple and Dynamic Keystores](#)

### **[Kerberos Cipher Suites](#)**

[Kerberos Requirements](#)  
[Peer Identity Information](#)  
[Security Manager](#)

### **[Additional Keystore Formats \(PKCS12\)](#)**

### **[Troubleshooting](#)**

[Configuration Problems](#)  
[Debugging Utilities](#)

### **[Code Examples](#)**

[Converting an Unsecure Socket to a Secure Socket](#)  
[Running the JSSE Sample Code](#)  
[Creating a Keystore to Use with JSSE](#)

### **[Appendix A: Standard Names](#)**

### **[Appendix B: Provider Pluggability](#)**

# 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 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 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 Hypertext Transfer Protocol (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 which developers can integrate directly into their applications.

JSSE was previously an optional package to the Java 2 SDK, Standard Edition (J2SDK), v 1.3. JSSE was integrated into the Java Standard Edition Development Kit starting with J2SDK 1.4.

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, SSLContexts, and a socket factory framework for encapsulating socket creation behavior. Because the socket APIs were based on a blocking I/O model, in JDK 5.0, a nonblocking `SSLContextSpi` was introduced to allow implementations to choose their own I/O methods.

The JSSE API is capable of supporting SSL versions 2.0 and 3.0 and Transport Layer Security (TLS) 1.0. These security protocols encapsulate a normal bidirectional stream socket and the JSSE API adds transparent support for authentication, encryption, and integrity protection. The JSSE implementation shipped with Oracle's JRE supports SSL 3.0 and TLS 1.0. It does not implement SSL 2.0.

As mentioned above, JSSE is a security component of the Java SE 6 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 same "[provider](#)" architecture defined in the JCA.

Other security components in the Java SE 6 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 JCE but automatically applies them underneath a simple stream socket API.

The JSSE APIs were designed to allow other SSL/TLS protocol and Public Key Infrastructure (PKI) implementations to be plugged in seamlessly. Developers can also provide alternate logic for determining 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 JRE 1.4 and later
- Extensible, provider based architecture
- Implemented in 100% Pure Java
- Provides API support for SSL versions 2.0 and 3.0, TLS 1.0 and later; and an implementation of SSL 3.0 and TLS 1.0
- 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 Hypertext Transfer Protocol (HTTP) encapsulated in the SSL protocol (HTTPS), 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 several cryptographic algorithms commonly used in cipher suites, including those listed in the following table:

Cryptographic Functionality Available With JSSE

Cryptographic Algorithm <sup>1</sup>	Cryptographic Process	Key Lengths (Bits)
RSA	Authentication and key exchange	512 and larger
RC4	Bulk encryption	128 128 (40 effective)
DES	Bulk encryption	64 (56 effective) 64 (40 effective)
Triple DES	Bulk encryption	192 (112 effective)
AES	Bulk encryption	256 <sup>2</sup> 128
Diffie-Hellman	Key agreement	1024 512
DSA	Authentication	1024

<sup>1</sup> Note: The SunJSSE implementation uses the [Java Cryptography Extension \(JCE\)](#) for all its cryptographic algorithms.

<sup>2</sup> Cipher suites that use AES\_256 require installation of the JCE Unlimited Strength Jurisdiction Policy Files. See [Java SE Download Page](#).

## JSSE Standard API

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

- Secure (SSL) sockets and server sockets.
- A nonblocking engine for producing and consuming streams of SSL/TLS data (`SSLEngine`).
- Factories for creating sockets, server sockets, SSL sockets, and SSL server sockets. 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 pre-installed and pre-registered with the JCA. This provider supplies the following cryptographic services:

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

More information about this provider is available in the [SunJSSE](#) section.

## Related Documentation

### Java Secure Socket Extension Documentation

- Archive of API-related questions and answers posted to Sun's Java Security team through [java-security@sun.com](mailto:java-security@sun.com): <http://archives.java.sun.com/archives/java-security.html>

Note: The above mailing list is not a subscription list or a support mechanism. It is simply a one-way channel that you can use to send comments to the Java SE 6 Standard Edition security team.

- JSSE API documentation:
  - [javax.net.package](#)
  - [javax.net.ssl.package](#)
  - [javax.security.cert.package](#)

### Java Platform Security Documentation

- The Java Security home page has links to White Papers, Books, Secure Coding guidelines, etc: [Java SE Security](#)
- The Java Certification Path API Programmer's Guide: [CertPath Programmer's Guide](#)
- Links to more Java SE 6 platform security documents: [Security Guides page](#)
- Tutorial for Java platform security: [Security Features in Java SE](#)
- Book on Java SE platform security: [Inside Java 2 Platform Security: Architecture, API Design, and Implementation](#) by Li Gong. Addison Wesley Longman, Inc., 1999. ISBN: 0201310007.

## Export Issues Related to Cryptography

For information on U.S. encryption policies, refer to these Web sites:

- U.S. Department of Commerce:

<http://www.commerce.gov>

- Export Policy Resource Page:  
<http://www.crypto.com/>
- Computer Systems Public Policy:  
<http://www.cspp.org/>
- Federal Information Processing Standards Publications (FIPS PUBS) homepage, which has links to the Data Encryption Standard (DES):  
<http://www.itl.nist.gov/fipspubs/>
- Revised U.S. Encryption Export Control Regulations:  
[http://www.epic.org/crypto/export\\_controls/regs\\_1\\_00.html](http://www.epic.org/crypto/export_controls/regs_1_00.html)

## Cryptography Documentation

Online resources:

- Dr. Rivest's Cryptography and Security page:  
<http://theory.lcs.mit.edu/~rivest/crypto-security.html>

Books:

- *Applied Cryptography, Second Edition* by Bruce Schneier. John Wiley and Sons, Inc., 1996.
- *Cryptography Theory and Practice* by Doug Stinson. CRC Press, Inc., 1995.
- *Cryptography & Network Security: Principles & Practice* by William Stallings. Prentice Hall, 1998.

## Secure Sockets Layer Documentation

Online resources:

- Introduction to SSL from Sun ONE Software:  
<http://docs.sun.com/source/816-6156-10/contents.htm>
- The SSL Protocol version 3.0 Internet Draft:  
<http://wp.netscape.com/eng/ssl3/ssl-toc.html>
- The TLS Protocol version 1.0 RFC:  
<http://www.ietf.org/rfc/rfc2246.txt>
- "HTTP Over TLS" Information RFC:  
<http://www.ietf.org/rfc/rfc2818.txt>

Books:

- *SSL and TLS: Designing and Building Secure Systems* by Eric Rescorla. Addison Wesley Professional, 2000.
- *SSL and TLS Essentials: Securing the Web* by Stephen Thomas. John Wiley and Sons, Inc., 2000.
- *Java 2 Network Security, Second Edition*, by Marco Pistoia, Duane F Reller, Deepak Gupta, Milind Nagnur, and Ashok K Ramani. Prentice Hall, 1999. Copyright 1999 International Business Machines.

# Terms and Definitions

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

## Authentication

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

## Cipher Suite

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

## Certificate

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

## Cryptographic Hash Function

A *cryptographic hash function* is similar to a checksum. Data is processed with an algorithm that produces a relatively small string of bits called a hash. A cryptographic hash function 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

In the JCA, implementations for various cryptographic algorithms are provided by *cryptographic service providers*, or "[providers](#)" for short. Providers are essentially packages that implement one or more engine classes for specific algorithms. An engine class defines a cryptographic service in an abstract fashion without a concrete implementation.

## Digital Signature

A *digital signature* is the 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 and Decryption

*Encryption* is the process of using a complex algorithm to convert an original message, or *cleartext*, to an encoded message, called *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

*Key agreement* is 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 will be able to obtain the final key. Diffie-Hellman (DH) is the most common example of a key agreement algorithm.

## Key Exchange

One side generates a symmetric key and encrypts it using the peer's public key (typically RSA). The data is then transmitted to the peer, who then decrypts the key using its corresponding private key.

## Key Managers and Trust Managers

*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, e.g., for use in authenticating the user to others. A trust manager makes decisions about who to trust based on information in the truststore it manages.

## Keystores and Truststores

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

Generally speaking, keystore information can be grouped into two different 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 only contains a public key in addition to the entity's identity. Thus, a trusted certificate entry can not 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 which is used when making decisions about what to trust. If you receive some data from an entity that you already trust, and if you can verify that the entity is the one 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 makes a decision to trust that entity. By either generating a keypair or by importing a certificate, the user has given trust to that entry, and thus any entry in the keystore 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 Certification Authority (CA) certificates. The former contains private information, while the latter does not. Using two different files instead of a single keystore file provides for a cleaner separation of the logical distinction between your own certificates (and corresponding private keys) and others' certificates. You could provide more protection for your private keys if you store them in a keystore with restricted access, while providing the trusted certificates in a more publicly accessible keystore if needed.

## Message Authentication Code

A *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 Message Digest 5 (MD5) and Secure Hash Algorithm (SHA), in combination with a secret shared key. HMAC is specified in RFC 2104.

## Public Key Cryptography

*Public key cryptography* uses an encryption algorithm in which two keys are produced. One key is made public while 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

The *record protocol* 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

*Secret key cryptography* 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 *session* is a named collection of state information including authenticated peer identity, cipher suite, and key agreement secrets which are negotiated through a secure socket handshake and which can be shared among multiple secure socket instances.

## Trust Managers

See [Key Managers and Trust Managers](#).

## Truststore

See [Keystores and Truststores](#).

# Secure Sockets Layer (SSL) Protocol Overview

Secure Sockets Layer (SSL) is the most widely used protocol for implementing cryptography on the Web. SSL uses a combination of cryptographic processes to provide secure communication over a network. This section provides an introduction to SSL and the cryptographic processes it uses.

SSL provides a secure enhancement to the standard TCP/IP sockets protocol used for Internet communications. As

shown in the following table, "TCP/IP Protocol Stack with SSL," the secure sockets layer is added between the transport layer and the application layer in the standard TCP/IP protocol stack. The application most commonly used with SSL is Hypertext Transfer Protocol (HTTP), the protocol for Internet Web pages. Other applications, such as Net News Transfer Protocol (NNTP), Telnet, Lightweight Directory Access Protocol (LDAP), Interactive Message Access Protocol (IMAP), and File Transfer Protocol (FTP), can be used with SSL as well.

Note: There is currently no standard for secure FTP.

TCP/IP Protocol Stack with SSL

TCP/IP Layer	Protocol
Application Layer	HTTP, NNTP, Telnet, FTP, etc.
Secure Sockets Layer	SSL
Transport Layer	TCP
Internet Layer	IP

SSL was developed by Netscape in 1994, and with input from the Internet community, has evolved to become a standard. It is now under the control of the international standards organization, the Internet Engineering Task Force (IETF). The IETF has renamed SSL to Transport Layer Security (TLS), and released the first specification, version 1.0, in January 1999. TLS 1.0 is a modest upgrade to the most recent version of SSL, version 3.0. The differences between SSL 3.0 and TLS 1.0 are minor.

## Why Use SSL?

Transferring sensitive information over a network can be risky due to the following three issues:

- You cannot always be sure that the entity with whom you are communicating is really who you think it is.
- Network data can be intercepted, so it is possible that it can be read by an unauthorized third party, sometimes known as an attacker.
- If an attacker can intercept the data, the attacker may be able to modify the data before sending it on to the receiver.

SSL addresses each of these issues. It addresses the first issue by optionally allowing each of two communicating parties to ensure the identity of the other party in a process called authentication. Once the parties are authenticated, SSL provides an encrypted connection between the two parties for secure message transmission. Encrypting the communication between the two parties provides privacy and therefore addresses the second issue. The encryption algorithms used with SSL include a secure hash function, which is similar to a checksum. This ensures that data is not modified in transit. The secure hash function addresses the third issue of data integrity.

Note, both authentication and encryption are optional, and depend on the the negotiated cipher suites between the two entities.

The most obvious example of when you would use SSL is in an e-commerce transaction. In an e-commerce transaction, it would be foolish to assume that you can guarantee the identity of the server with whom you are communicating. It would be easy enough for someone to create a phony Web site promising great services if only you enter your credit card number. SSL allows you, the client, to authenticate the identity of the server. It also allows the server to authenticate the identity of the client, although in Internet transactions, this is seldom done.

Once the client and the server are comfortable with each other's identity, SSL provides privacy and data integrity through the encryption algorithms it uses. This allows sensitive information, such as credit card numbers, to be transmitted securely over the Internet.

While SSL provides authentication, privacy, and data integrity, it does not provide non-repudiation services. Non-

repudiation means that an entity that sends a message cannot later deny that they sent it. When the digital equivalent of a signature is associated with a message, the communication can later be proved. SSL alone does not provide non-repudiation.

## How SSL Works

One of the reasons SSL is effective is that it uses several different cryptographic processes. SSL uses public key cryptography to provide authentication, and secret key cryptography and digital signatures to provide for privacy and data integrity. Before you can understand SSL, it is helpful to understand these cryptographic processes.

### Cryptographic Processes

The primary purpose of cryptography is to make it difficult for an unauthorized third party to access and understand private communication between two parties. It is not always possible to restrict all unauthorized access to data, but private data can be made unintelligible to unauthorized parties through the process of encryption. Encryption uses complex algorithms to convert the original message, or cleartext, to an encoded message, called ciphertext. The algorithms used to encrypt and decrypt data that is transferred over a network typically come in two categories: secret key cryptography and public key cryptography. These forms of cryptography are explained in the following subsections.

Both secret key cryptography and public key cryptography depend on the use of an agreed-upon cryptographic key or pair of keys. A key is a string of bits that is used by the cryptographic algorithm or algorithms during the process of encrypting and decrypting the data. A cryptographic key is like a key for a lock: only with the right key can you open the lock.

Safely transmitting a key between two communicating parties is not a trivial matter. A public key certificate allows a party to safely transmit its public key, while ensuring the receiver of the authenticity of the public key. Public key certificates are described in a later section.

In the descriptions of the cryptographic processes that follow, we use the conventions used by the security community: we label the two communicating parties with the names Alice and Bob. We call the unauthorized third party, also known as the attacker, Charlie.

### Secret Key Cryptography

With secret key cryptography, both communicating parties, Alice and Bob, use the same key to encrypt and decrypt the messages. Before any encrypted data can be sent over the network, both Alice and Bob must have the key and must agree on the cryptographic algorithm that they will use for encryption and decryption.

One of the major problems with secret key cryptography is the logistical issue of how to get the key from one party to the other without allowing access to an attacker. If Alice and Bob are securing their data with secret key cryptography, and if Charlie gains access to their key, Charlie can understand any secret messages he intercepts between Alice and Bob. Not only can Charlie decrypt Alice's and Bob's messages, but he can also pretend that he is Alice and send encrypted data to Bob. Bob will not know that the message came from Charlie, not Alice.

Once the problem of secret key distribution is solved, secret key cryptography can be a valuable tool. The algorithms provide excellent security and encrypt data relatively quickly. The majority of the sensitive data sent in an SSL session is sent using secret key cryptography.

Secret key cryptography is also called *symmetric cryptography* because the same key is used to both encrypt and decrypt the data. Well-known secret key cryptographic algorithms include the Data Encryption Standard (DES), triple-strength DES (3DES), Rivest Cipher 2 (RC2), and Rivest Cipher 4 (RC4).

### Public Key Cryptography

Public key cryptography solves the logistical problem of key distribution by using both a public key and a private key. The public key can be sent openly through the network while the private key is kept private by one of the communicating parties. The public and the private keys are cryptographic inverses of each other; what one key encrypts, the other key will decrypt.

Assume that Bob wants to send a secret message to Alice using public key cryptography. Alice has both a public key and a private key, so she keeps her private key in a safe place and sends her public key to Bob. Bob encrypts the secret message to Alice using Alice's public key. Alice can later decrypt the message with her private key.

If Alice encrypts a message using her private key and sends the encrypted message to Bob, Bob can be sure that the data he receives comes from Alice; if Bob can decrypt the data with Alice's public key, the message must have been encrypted by Alice with her private key, and only Alice has Alice's private key. The problem is that anybody else can read the message as well because Alice's public key is public. While this scenario does not allow for secure data communication, it does provide the basis for digital signatures. A digital signature is one of the components of a public key certificate, and is used in SSL to authenticate a client or a server. Public key certificates and digital signatures are described in later sections.

Public key cryptography is also called *asymmetric cryptography* because different keys are used to encrypt and decrypt the data. A well known public key cryptographic algorithm often used with SSL is the Rivest Shamir Adleman (RSA) algorithm. Another public key algorithm used with SSL that is designed specifically for secret key exchange is the Diffie-Hellman (DH) algorithm. Public key cryptography requires extensive computations, making it very slow. It is therefore typically used only for encrypting small pieces of data, such as secret keys, rather than for the bulk of encrypted data communications.

## A Comparison Between Secret Key and Public Key Cryptography

Both secret key cryptography and public key cryptography have strengths and weaknesses. With secret key cryptography, data can be encrypted and decrypted quickly, but because both communicating parties must share the same secret key information, the logistics of exchanging the key can be a problem. With public key cryptography, key exchange is not a problem because the public key does not need to be kept secret, but the algorithms used to encrypt and decrypt data require extensive computations, and are therefore very slow.

## Public Key Certificates

A public key certificate provides a safe way for an entity to pass on its public key to be used in asymmetric cryptography. The public key certificate avoids the following situation: if Charlie creates his own public key and private key, he can claim that he is Alice and send his public key to Bob. Bob will be able to communicate with Charlie, but Bob will think that he is sending his data to Alice.

A public key certificate can be thought of as the digital equivalent of a passport. It is issued by a trusted organization and provides identification for the bearer. A trusted organization that issues public key certificates is known as a certificate authority (CA). The CA can be likened to a notary public. To obtain a certificate from a CA, one must provide proof of identity. Once the CA is confident that the applicant represents the organization it says it represents, the CA signs the certificate attesting to the validity of the information contained within the certificate.

A public key certificate contains several fields, including:

- **Issuer** - The issuer is the CA that issued the certificate. If a user trusts the CA that issues a certificate, and if the certificate is valid, the user can trust the certificate.
- **Period of validity** - A certificate has an expiration date, and this date is one piece of information that should be checked when verifying the validity of a certificate.
- **Subject** - The subject field includes information about the entity that the certificate represents.
- **Subject's public key** - The primary piece of information that the certificate provides is the subject's public key. All

the other fields are provided to ensure the validity of this key.

- **Signature** - The certificate is digitally signed by the CA that issued the certificate. The signature is created using the CA's private key and ensures the validity of the certificate. Because only the certificate is signed, not the data sent in the SSL transaction, SSL does not provide for non-repudiation.

If Bob only accepts Alice's public key as valid when she sends it in a public key certificate, Bob will not be fooled into sending secret information to Charlie when Charlie masquerades as Alice.

Multiple certificates may be linked in a certificate chain. When a certificate chain is used, the first certificate is always that of the sender. The next is the certificate of the entity that issued the sender's certificate. If there are more certificates in the chain, each is that of the authority that issued the previous certificate. The final certificate in the chain is the certificate for a root CA. A root CA is a public certificate authority that is widely trusted. Information for several root CAs is typically stored in the client's Internet browser. This information includes the CA's public key. Well-known CAs include VeriSign, Entrust, and GTE CyberTrust.

## Cryptographic Hash Functions

When sending encrypted data, SSL typically uses a cryptographic hash function to ensure data integrity. The hash function prevents Charlie from tampering with data that Alice sends to Bob.

A cryptographic hash function is similar to a checksum. The main difference is that while a checksum is designed to detect accidental alterations in data, a cryptographic hash function is designed to detect deliberate alterations. When data is processed by a cryptographic hash function, a small string of bits, known as a hash, is generated. The slightest change to the message typically makes a large change in the resulting hash. A cryptographic hash function does not require a cryptographic key. Two hash functions often used with SSL are Message Digest 5 (MD5) and Secure Hash Algorithm (SHA). SHA was proposed by the [U.S. National Institute of Science and Technology \(NIST\)](http://www.nist.gov).

## Message Authentication Code

A message authentication code (MAC) is similar to a cryptographic hash, except that it is based on a secret key. When secret key information is included with the data that is processed by a cryptographic hash function, the resulting hash is known as an HMAC.

If Alice wants to be sure that Charlie does not tamper with her message to Bob, she can calculate an HMAC for her message and append the HMAC to her original message. She can then encrypt the message plus the HMAC using a secret key she shares with Bob. When Bob decrypts the message and calculates the HMAC, he will be able to tell if the message was modified in transit. With SSL, an HMAC is used with the transmission of secure data.

## Digital Signatures

Once a cryptographic hash is created for a message, the hash is encrypted with the sender's private key. This encrypted hash is called a digital signature.

## The SSL Process

Communication using SSL begins with an exchange of information between the client and the server. This exchange of information is called the SSL handshake.

The three main purposes of the SSL handshake are:

- Negotiate the cipher suite
- Authenticate identity (optional)
- Establish information security by agreeing on encryption mechanisms

## Negotiating the Cipher Suite

The SSL session begins with a negotiation between the client and the server as to which cipher suite they will use. A cipher suite is a set of cryptographic algorithms and key sizes that a computer can use to encrypt data. The cipher suite includes information about the public key exchange algorithms or key agreement algorithms, and cryptographic hash functions. The client tells the server which cipher suites it has available, and the server chooses the best mutually acceptable cipher suite.

## Authenticating the Server

In SSL, the authentication step is optional, but in the example of an e-commerce transaction over the Web, the client will generally want to authenticate the server. Authenticating the server allows the client to be sure that the server represents the entity that the client believes the server represents.

To prove that a server belongs to the organization that it claims to represent, the server presents its public key certificate to the client. If this certificate is valid, the client can be sure of the identity of the server.

The client and server exchange information that allows them to agree on the same secret key. For example, with RSA, the client uses the server's public key, obtained from the public key certificate, to encrypt the secret key information. The client sends the encrypted secret key information to the server. Only the server can decrypt this message because the server's private key is required for this decryption.

## Sending the Encrypted Data

Both the client and the server now have access to the same secret key. With each message, they use the cryptographic hash function, chosen in the first step of this process, and shared secret information, to compute an HMAC that they append to the message. They then use the secret key and the secret key algorithm negotiated in the first step of this process to encrypt the secure data and the HMAC. The client and server can now communicate securely using their encrypted and hashed data.

## The SSL Protocol

The previous section provides a high-level description of the SSL handshake, which is the exchange of information between the client and the server prior to sending the encrypted message. This section provides more detail.

The "SSL Messages" figure that follows shows the sequence of messages that are exchanged in the SSL handshake. Messages that are sent only in certain situations are noted as optional. Each of the SSL messages is described in the following figure:

Sequence of messages exchanged in SSL handshake.

The SSL messages are sent in the following order:

1. **Client hello** - The client sends the server information including the highest version of SSL it supports and a list of the cipher suites it supports. (TLS 1.0 is indicated as SSL 3.1.) The cipher suite information includes cryptographic algorithms and key sizes.
2. **Server hello** - The server chooses the highest version of SSL and the best cipher suite that both the client and server support and sends this information to the client.
3. **Certificate** - The server sends the client a certificate or a certificate chain. A certificate chain typically begins with the server's public key certificate and ends with the certificate authority's root certificate. This message is optional, but is used whenever server authentication is required.
4. **Certificate request** - If the server needs to authenticate the client, it sends the client a certificate request. In Internet applications, this message is rarely sent.
5. **Server key exchange** - The server sends the client a server key exchange message when the public key information sent in message 3 above is not sufficient for key exchange. For example, in ciphersuites based on Diffie-Hellman, this message contains the server's DH public key.
6. **Server hello done** - The server tells the client that it is finished with its initial negotiation messages.
7. **Certificate** - If the server requests a certificate from the client in message 4, the client sends its certificate chain, just as the server did in message 3.

Note: Only a few Internet server applications ask for a certificate from the client.

8. **Client key exchange** - The client generates information used to create a key to use for symmetric encryption. For RSA, the client then encrypts this key information with the server's public key and sends it to the server. For ciphersuites based on Diffie-Hellman, this message contains the client's DH public key.
9. **Certificate verify** - This message is sent when a client presents a certificate as previously explained. Its purpose is to allow the server to complete the process of authenticating the client. When this message is used, the client sends information that it digitally signs using a cryptographic hash function. When the server decrypts this information with the client's public key, the server is able to authenticate the client.

10. **Change cipher spec** - The client sends a message telling the server to change to encrypted mode.
11. **Finished** - The client tells the server that it is ready for secure data communication to begin.
12. **Change cipher spec** - The server sends a message telling the client to change to encrypted mode.
13. **Finished** - The server tells the client that it is ready for secure data communication to begin. This is the end of the SSL handshake.
14. **Encrypted data** - The client and the server communicate using the symmetric encryption algorithm and the cryptographic hash function negotiated in messages 1 and 2, and using the secret key that the client sent to the server in Message 8. The handshake can be renegotiated at this time. See the next section for details.
15. **Close Messages** - At the end of the connection, each side will send a `close_notify` message to inform the peer that the connection is closed.

If the parameters generated during an SSL session are saved, these parameters can sometimes be reused for future SSL sessions. Saving SSL session parameters allows encrypted communication to begin much more quickly.

### Handshaking Again (Renegotiation)

Once the initial handshake has been finished and application data is flowing, either side is free to initiate a new handshake at any time. An application might like to use a stronger cipher suite for especially critical operations, or a server application might want to require client authentication.

Regardless of the reason, the new handshake takes place over the existing encrypted session, and application data and handshake messages are interleaved until a new session is established.

Your application can initiate a new handshake using one of the following methods:

- `SSLSocket.startHandshake()`
- `SSLEngine.beginHandshake()`

Note that a protocol flaw related to renegotiation was found in 2009. The protocol and the Java SE implementation have both been fixed. For more information, see [Transport Layer Security \(TLS\) Renegotiation Issue](#).

### 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 will directly impact the type of security the connection enjoys. For example, if an anonymous cipher suite is selected, the application will have no way to verify the remote peer's identity. If a suite with no encryption is selected, then the privacy of the data can not 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 presented were acceptable based on the current trust material, the connection would be considered valid.

When using raw `SSLsockets/SSLEngines` you should always check the peer's credentials before sending any data. The `SSLSocket` and `SSLEngine` classes do not automatically verify that the hostname in a URL matches the hostname in the peer's credentials. An application could be exploited with URL spoofing if the hostname is not verified.

Protocols such as [https](#) do require hostname verification. Applications can use [HostnameVerifier](#) to override the default HTTPS hostname rules. See [HttpsURLConnection](#) for more information.

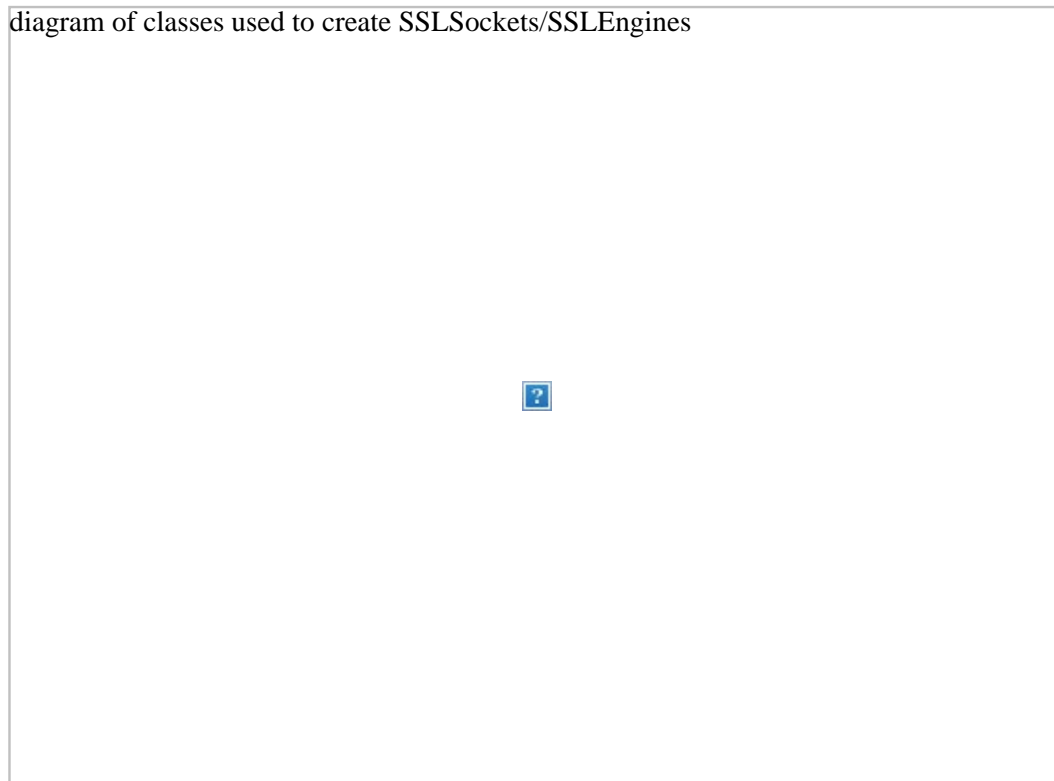
## SSL and TLS References

For a list of resources containing more information about SSL, see [Secure Sockets Layer Documentation](#).

# Key Classes

## Relationship Between Classes

To communicate securely, both sides of the connection must be SSL-enabled. In the JSSE API, the endpoint classes of the connection is the `SSLSocket` and `SSLEngine`. In the diagram below, the major classes used to create `SSLSocket`/`SSLEngines` are laid out in a logical ordering.



An `SSLSocket` is created either by an `SSLSocketFactory` or by an `SSLServerSocket` accepting an in-bound 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 the `SSLContext`, and relies on the application to handle all I/O.

---

**IMPORTANT NOTE:** When using raw `SSLockets/SSLEngines` you should always check the peer's credentials before sending any data. The `SSLSocket/SSLEngine` classes do not automatically verify, for example, that the hostname in a URL matches the hostname in the peer's credentials. An application could be exploited with URL spoofing if the hostname is not verified.

---

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

- The simplest is to call the static `getDefault` method on either the `SSLSocketFactory` or `SSLServerSocketFactory` class. These methods create a default `SSLContext` with a default `KeyManager`, `TrustManager`, and 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/truststore, as determined by system properties described in [Customizing the Default Key and Trust Stores, 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, 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` random number generator. The `KeyManager` and `TrustManager` objects are created by either implementing the appropriate interface(s) 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 `TrustManagerFactory/KeyManagerFactory` `init` method. Finally, the `getTrustManagers` method (in `TrustManagerFactory`) and `getKeyManagers` method (in `KeyManagerFactory`) can be called to obtain the array of trust 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, cipher suite used, etc. 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.

## 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. It must be subclassed by other factories, which create particular subclasses of sockets and thus provide a general framework for the addition of public socket-level functionality. (See, for example, [SSLSocketFactory](#).)

The `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 which does not require special configuration of the code which 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. So 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

A `javax.net.ssl.SSLSocketFactory` 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`

There are three primary ways of obtaining an `SSLSocketFactory`:

- Get the default factory by calling the `SSLSocketFactory.getDefault` static method.
- Receive a factory as an API parameter. That is, code which needs to create sockets but which doesn't 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.HttpURLConnection`.)
- 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 which 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 which 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 additional 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 a `SSLSocket`.

Implementation 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 two ways. First, an `SSLSocket` can be created by an instance of [SSLSocketFactory](#) via one of the several `createSocket` methods on that class. The second way to obtain `SSLSocket`s is through the `accept` method on the `SSLServerSocket` class.

### Nonblocking I/O with `SSLEngine`

SSL/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 comes demands to use it with different I/O and threading models in order to satisfy the applications' performance, scalability, footprint, and other requirements. There are

demands to use it 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.

Prior to Java SE 5, the JSSE API supported only a single transport abstraction: stream-based sockets via `SSLSocket`. While this was adequate for many applications, it did not meet the needs of applications that need to use different I/O or threading models. In 1.6.0, a new abstraction was introduced to allow applications to use the SSL/TLS protocols in a transport independent way, and thus freeing applications to choose transport and computing models that best meet their needs. Not only does this new 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. Because of this flexibility, the application must now manage I/O and threading (complex topics in and of themselves), as well as have some understanding of the SSL/TLS protocols. The new abstraction is therefore an advanced API: beginners should continue to use `SSLSocket`.

Newcomers to the API may wonder "Why not just have an `SSLSocketChannel` which extends `java.nio.channels.SocketChannel`?" There are two main reasons:

- There were a lot of very difficult questions about what a `SSLSocketChannel` should be, including its class hierarchy and how it should interoperate with `Selectors` and other types of `SocketChannels`. Each proposal brought up more questions than answers. It was noted that any new API abstraction extended to work with SSL/TLS would require the same significant analysis and could result in large and complex APIs.
- Any JSSE implementation of a new API would be free to choose the "best" I/O & compute strategy, but hiding any of these details is inappropriate for those applications needing full control. Any specific implementation would be inappropriate for some application segment.

By abstracting the I/O and treating data as streams of bytes, these issues are resolved and the new API could be used with any existing or future I/O model. While this solution makes I/O and CPU handling the developers' responsibility, JSSE implementations are prevented from being unusable due to some unconfigurable and/or unchangeable internal detail.

Users of other Java programming language APIs such as JGSS and SASL will notice similarities in that the application is also responsible for transporting data.

### SSLEngine

The core class in this new abstraction is [javax.net.ssl.SSLEngine](#). It encapsulates an SSL/TLS state machine and operates on inbound and outbound byte buffers supplied by the user of the `SSLEngine`. The following diagram illustrates the flow of data from the application, to the `SSLEngine`, to the transport mechanism, and back.



The application, shown on the left, supplies application (plaintext) data in an application buffer and passes it to the `SSLEngine`. The `SSLEngine` processes the data contained in the buffer, or any handshaking data, to produce SSL/TLS encoded data and places it 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 SSL/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` processes the network buffer's contents to produce handshaking data or application data.

In all, `SSLEngine` can be in one of five states.

1. Creation - ready to be configured.
2. Initial handshaking - perform authentication and negotiate communication parameters.
3. Application data - ready for application exchange.
4. Rehandshaking - renegotiate communications parameters/authentication; handshaking data may be mixed with

application data.

#### 5. Closure - ready to shut down connection.

These five states are described in more detail in the [SSL`Engine`](#) class documentation.

## Getting Started

To create an `SSLEngine`, you use the `SSLContext.createSSLEngine()` methods. You must then configure the engine to act as a client or a server, as well as set other configuration parameters such as which cipher suites to use and whether to require client authentication.

Here is an example that creates an `SSLEngine`. Note that 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.

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

// Create/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);

// KeyManager's decide which key material to use.
KeyManagerFactory kmf =
    KeyManagerFactory.getInstance("SunX509");
kmf.init(ksKeys, passphrase);

// TrustManager's decide whether to allow connections.
TrustManagerFactory tmf =
    TrustManagerFactory.getInstance("SunX509");
tmf.init(ksTrust);

sslContext = SSLContext.getInstance("TLS");
sslContext.init(
    kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// We're ready for the engine.
SSLEngine engine = sslContext.createSSLEngine(hostname, port);

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

## Generating and Processing SSL/TLS data

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

Each `SSLEngine` has several phases during its lifetime. Before application data can be sent/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`. [The SSL Process](#) can provide more details about the handshake itself.

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

A typical handshake might look like this:

client	SSL/TLS message	HSStatus
wrap()	ClientHello	NEED_UNWRAP
unwrap()	ServerHello/Cert/ServerHelloDone	NEED_WRAP
wrap()	ClientKeyExchange	NEED_WRAP
wrap()	ChangeCipherSpec	NEED_WRAP
wrap()	Finished	NEED_UNWRAP
unwrap()	ChangeCipherSpec	NEED_UNWRAP
unwrap()	Finished	FINISHED

Now that handshaking is complete, further calls to `wrap()` will attempt to consume application data and packages it for transport. `unwrap()` attempts the opposite.

To send data to the peer, the application first supplies the data that it wants to send to `SSLSession.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 `SSLSession` via `SSLSession.unwrap()` to obtain the plaintext data sent by the peer.

Here is an example of an SSL application that is using a nonblocking `SocketChannel` to communicate with its peer. (It can be made more robust and scalable by using a `Selector` with the nonblocking `SocketChannel`.) The following sample code sends the string "hello" to its peer, by encoding it using the `SSLSession` created in the previous example. It uses information from the `SSLSession` to determine how large to make the byte buffers.

```
// 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)
    SSLSessionResult res = engine.wrap(myAppData, myNetData);

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

        // Send SSL/TLS encoded data to peer
        while(myNetData.hasRemaining()) {
            int num = socketChannel.write(myNetData);
            if (num == -1) {
                // handle closed channel
            } else if (num == 0) {
                // no bytes written; try again later
            }
        }
    }

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

The following code reads data from the same nonblocking `SocketChannel` and extracts the plaintext data from it by

using the `SSLEngine` created previously. Each iteration of this code may or may not produce any plaintext data, depending on whether handshaking is in progress.

```
// Read SSL/TLS encoded data from peer
int num = socketChannel.read(peerNetData);
if (num == -1) {
    // Handle closed channel
} 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
    ...
}
```

## Status of Operations

To indicate the status of the engine and what action(s) the application should take, the `SSLEngine.wrap()` and `SSLEngine.unwrap()` methods return an [SSL`Engine``Result`](#) instance, as shown in the previous examples. The `SSLEngineResult` 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. Some examples of this status include `OK`, which means that there was no error, and `BUFFER_UNDERFLOW`, which means that the input buffer had insufficient data, indicating that the application needs to obtain more data from the peer (for example, by reading more data from the network), and `BUFFER_OVERFLOW`, which means that the output buffer had insufficient space to hold the result, indicating that the application needs to clear or enlarge the destination buffer.

Here is an example of how to handle `BUFFER_UNDERFLOW` and `BUFFER_OVERFLOW` statuses of `SSLEngine.unwrap()`. It uses `SSLSession.getApplicationBufferSize()` and `SSLSession.getPacketBufferSize()` to determine how large to make the byte buffers.

```
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 needs to obtain more handshaking data from the peer, send more handshaking data to the peer, and so on.

Having two statuses per result allows the engine to indicate that the application must take two actions: one in response to the handshaking and one representing the overall status of the `wrap()/unwrap()` method. 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 SSL/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.

Here is an example of how to process handshaking data by checking handshaking status and the overall status of the `wrap()/unwrap()` method.

```
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) {
                    // Handle closed channel
                }

                // 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()) {
                            if (socketChannel.write(myNetData) < 0) {
                                // Handle closed channel
                            }
                        }
                }
            }
        }
    }
}
```

```

        }
        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
    ...
}

```

## Blocking Tasks

During handshaking, the `SSL`Engine might encounter tasks that might 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 `SSL`Engine, when the engine encounters such a task, it will return `SSL`EngineResult.HandshakeStatus.NEED\_TASK. Upon receiving this status, the application should invoke `SSL`Engine.getDelegatedTask() to get the task, and then, using the threading model appropriate for its requirements, process the task. The application might, for example, obtain thread(s) from a thread pool to process the task(s), while the main thread goes about handling other I/O.

Here is an example that 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 engine will block future `wrap/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 `SSL`Engine, then transmit them to the peer using its transport mechanism, and finally shut down the transport mechanism. Here is an example.

```

// 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 == -1) {
            // handle closed channel
        } else if (num == 0) {
            // no bytes written; try again later
        }
        myNetData().compact();
    }
}

```

```

}
// Close transport
socketChannel.close();

```

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

In addition to orderly shutdowns, there can also be unorderly shutdowns in which the transport link is severed before close messages are exchanged. In the previous examples, the application might get `-1` when trying to read or write to the nonblocking `SocketChannel`. When you get to the end of your input data, you should call `engine.closeInbound()`, which will verify with the `SSL`Engine that the remote peer has closed cleanly from the `SSL/TLS` perspective, and then the application should still try to shutdown cleanly by using the procedure above. Obviously, unlike `SSL`Socket, the application using `SSL`Engine must deal with more state transitions, statuses and programming than when using `SSL`Engine. Please see the [NIO-based HTTPS server](#) for more information on writing a `SSL`Engine-based application.

## SSLSession and ExtendedSSLSession Interfaces

A `javax.net.ssl.SSLSession` represents a security context negotiated between the two peers of an `SSL`Socket or `SSL`Engine connection. After a session has been arranged, it can be shared by future `SSL`Socket or `SSL`Engine 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. In the Java SE 7 release, the `SSLSession` can be retrieved *during* the handshake by calling `getHandshakeSession()` on an `SSL`Socket or `SSL`Engine. Implementations of `TrustManager` or `KeyManager` can use `getHandshakeSession()` to get information about session parameters to help them make decisions.

A fully initialized `SSLSession` contains the cipher suite which will be used for communications over a secure socket as well as a non-authoritative 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 `SSL`Socket or `SSL`Engine 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 the Java SE 7 release, a `TLS 1.2` session is represented by `ExtendedSSLSession`, an implementation of `SSLSession`. `ExtendedSSLSession` adds methods that describe the signature algorithms that are supported by the local implementation and the peer.

Calls to `SSLSession.getPacketBufferSize()` and `SSLSession.getApplicationBufferSize()` also are used to determine the appropriate buffer sizes used by `SSL`Engine.

**Note:** The `SSL/TLS` protocols specify that implementations are to produce packets containing at most 16 KB of plaintext. However, some implementations violate the specification and generate large records up to 32 KB. If the `SSL`Engine.unwrap() code detects large inbound packets, the buffer sizes returned by `SSLSession` will be updated dynamically. Applications should always [check the BUFFER\\_OVERFLOW/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. (Also see the System property `jsse.SSL`Engine.acceptLargeFragments in [Customization](#) for a workaround.)

## 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/receiving data. `javax.net.ssl.HttpURLConnection` extends the `java.net.HttpURLConnection` class, and adds support for https-specific features. See the [java.net.URL](#), [java.net.URLConnection](#), [java.net.HttpURLConnection](#), and [javax.net.ssl.HttpURLConnection](#) classes for more information about how https URLs are constructed and used.

Upon obtaining a `HttpURLConnection`, you can configure a number of http/https parameters before actually initiating the network connection via the method `URLConnection.connect`. 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 `HttpURLConnection` instance uses. For example, you may wish to tunnel through a proxy type that isn't supported by the default implementation. The new `SSLSocketFactory` could return sockets that have already performed all necessary tunneling, thus allowing `HttpURLConnection` to use additional proxies.

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

Note that changing the default static `SSLSocketFactory` has no effect on existing instances of `HttpURLConnections`, a call to the `setSSLSocketFactory` method is necessary to change the existing instance.

One can obtain the per-instance or per-class `SSLSocketFactory` by making a call to the `getSSLSocketFactory/getDefaultSSLSocketFactory` methods, 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, it is possible that URL spoofing has occurred. If the implementation cannot determine a host name match with reasonable certainty, the SSL implementation will perform a callback to the instance's assigned `HostnameVerifier` for further checking. The host name verifier can perform whatever steps are necessary to make the determination, such as performing alternate host name pattern matching or perhaps popping up an interactive dialog box. An unsuccessful verification by the host name verifier will close the connection. (See [RFC 2818](#) for more information regarding host name verification.)

The `setHostnameVerifier/setDefaultHostnameVerifier` methods operate in a similar manner to the `setSSLSocketFactory/setDefaultSSLSocketFactory` methods, in that there are `HostnameVerifiers` assigned on a per-instance and per-class basis, and the current values can be obtained by a call to the `getHostnameVerifier/getDefaultHostnameVerifier` methods.

## 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 `SSLContext` 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 on 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 (`java.security`) API. The implementations supplied by `SunJSSE` are:

Engine Class Implemented	Algorithm or Protocol
<code>KeyStore</code>	PKCS12
<code>KeyManagerFactory</code>	PKIX, SunX509
<code>TrustManagerFactory</code>	PKIX (a.k.a. X509 or SunPKIX), SunX509
<code>SSLContext</code>	SSLv3 (a.k.a. SSL), TLSv1 (a.k.a. TLS), TLSv1.1, TLSv1.2

### SSLContext Class

`javax.net.ssl.SSLContext` 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` 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.

### 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")` may return a instance which implements "TLSv1", "TLSv1.1" and "TLSv1.2". The `getSupportedProtocols` method returns a list of supported protocols when an `SSLSocket`, `SSLServerSocket` or `SSLContext` is created from this context. You can control which protocols are actually enabled for an SSL connection by using the method `setEnabledProtocols(String[] protocols)`.

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

To create an `SSLContext` object by calling a `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, the system will determine if there is an implementation of the requested protocol available in the environment, and if there is more than one, if there is a preferred one.

If both a protocol name and a provider are specified, the system will determine if there is an implementation of the requested protocol in the provider requested, and throw an exception if there is not.

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

Here is an example of obtaining an `SSLContext`:

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

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, 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, (e.g. a `SSLContext` is created by `SSLContext.getDefault()` or `SSLContext.getDefault()`), [a default KeyManager and a TrustManager](#) are created. The default `SecureRandom` implementation is also chosen.

## 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, the connection will be terminated. To authenticate the remote identity of a secure socket peer, you need to initialize an `SSLContext` object with one or more `TrustManager`s. You need to pass one `TrustManager` for each authentication mechanism that is supported. If null is passed into the `SSLContext` initialization, a trust manager will be created for you. Typically, there is a single trust manager that supports authentication based on X.509 public key certificates (e.g. `X509TrustManager`). Some secure socket implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

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

## 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 that provide additional or alternate 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:

```
public static TrustManagerFactory
```

```

        getInstance(String algorithm);

public static TrustManagerFactory
        getInstance(String algorithm,
                   String provider);

public static TrustManagerFactory
        getInstance(String algorithm,
                   Provider provider);

```

A sample algorithm name string is:

`"PKIX"`

A sample call is the following:

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

The above call will create an instance of the `SunJSSE` provider's PKIX trust manager factory. This factory can then be used to create trust managers which provide X.509 PKIX-based certification path validity checking.

When initializing a `SSLContext`, you can use trust managers created from a trust manager factory, or you can write your own trust manager, perhaps using the [CertPath](#) API. (See the [Java Certification Path API Programmer's Guide](#) for details.) You don't need to use a trust manager factory at all 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);
```

You should call whichever `init` method is appropriate for the `TrustManagerFactory` you are using. (Ask the provider vendor.)

For many factories, such as the `"SunX509"` `TrustManagerFactory` from the `SunJSSE` provider, the `KeyStore` is the only information required in order to initialize the `TrustManagerFactory` and thus the first `init` method is the appropriate one to call. The `TrustManagerFactory` will query the `KeyStore` for information on which remote certificates should be trusted during authorization checks.

In some cases, initialization parameters other than a `KeyStore` may be needed by a provider. Users of that particular 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 wishes to use that provider. Like all providers that require initialization parameters other than a `KeyStore`, the provider will require that the application provide an instance of a class that implements a particular `ManagerFactoryParameters` sub-interface. In our example, suppose the provider requires that the calling application implement and create an instance of `MyTrustManagerFactoryParams` and pass it to the second `init`. Here is what `MyTrustManagerFactoryParams` may look like:

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

Some trustmanagers are capable of making trust decisions without having to be explicitly initialized with a `KeyStore` object or any other parameters. For example, they may access trust material from a local directory service via LDAP,

may use a remote online certificate status checking server, or may access default trust material from a standard local location.

## PKIX TrustManager Support

The default trust manager algorithm is "PKIX". The default 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; a "SUN" CertPath provider is supplied with the Java SE Development Kit 6. 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 newly introduced class [javax.net.ssl.CertPathTrustManagerParameters](#).

Here is an example of 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;
...

// Create PKIX parameters
KeyStore anchors = KeyStore.getInstance("JKS");
anchors.load(new FileInputStream(anchorsFile));
CertPathParameters 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 them 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, default PKIXParameters are used with the exception that revocation checking is disabled. It can be enabled by setting the system property `com.sun.net.ssl.checkRevocation` to `true`. Note that this setting requires that the CertPath implementation can locate revocation information by itself. The PKIX implementation in the SUN provider can do this in many cases but requires that the system property `com.sun.security.enableCRLDP` be set to `true`.

More information about PKIX and the CertPath API can be found in the [Java Certificate Path API Programming Guide](#).

## X509TrustManager Interface

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

In order 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 that delegates to a factory-generated trust manager. For example, you might do this in order 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`, the factory uses the following steps to try to find trust material:

1. If the [system property](#):

```
javax.net.ssl.trustStore
```

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

If `javax.net.ssl.trustStore` 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. (See [The Installation Directory <java-home>](#) for information about what `<java-home>` refers to.) Otherwise,

3. If the file

```
<java-home>/lib/security/cacerts
```

exists, that file is used.

(If none of these files exists, that may be okay because there are SSL cipher suites which are anonymous, that is, which don't do any authentication and thus don't need a truststore.)

The factory looks for a file specified via the security property `javax.net.ssl.trustStore` or for the `jssecacerts` file before checking for a `cacerts` file so that 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 isn't suitable for your situation, you can create your own `X509TrustManager` by either creating and registering your own `TrustManagerFactory` or by implementing the `X509TrustManager` interface directly.

The following `MyX509TrustManager` class enhances the default `SunJSSE X509 TrustManager` behavior by providing alternative authentication logic when the default `SunJSSE X509 TrustManager` fails.

```
class MyX509TrustManager implements X509TrustManager {
    /*
     * The default PKIX X509TrustManager9. We'll delegate
     * decisions to it, and fall back to the logic in this class if the
     * default X509TrustManager doesn't trust it.
     */
    X509TrustManager pkixTrustManager;

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

```

    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 trustmanagers, look
     * for an instance of X509TrustManager. If found,
     * use that as our "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 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.
         */
    }
}

/*
 * 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. 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 need to 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**

In the Java SE 7 release, 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 are able to specify which hash and signature algorithms they will accept. In order 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 got from 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 support algorithm constraints and SSL layer hostname verification. For JSSE providers and trust manager implementations, the `X509ExtendedTrustManager` class is highly recommended rather than 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 the Java SE 7 release, the `PKIX` or `SunX509` `TrustManagerFactory` returns an `X509ExtendedTrustManager` instance.

### **Creating Your Own X509ExtendedTrustManager**

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

The following class 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, the subclass is able to 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. We'll delegate
     * decisions to it, and fall back to the logic in this class if the
     * default X509ExtendedTrustManager doesn't 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 trustmanagers, look
     * for an instance of X509TrustManager. If found,
     * use that as our "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();
}
}

```

## 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 need to initialize an `SSLContext` object with one or more `KeyManagers`. You need to pass one `KeyManager` for each different authentication mechanism that will be supported. If null is passed into the `SSLContext` initialization, an empty `KeyManager` will be created. If the internal default context is used (e.g. a `SSLContext` created by `SSLContext.getDefault()` or `SSLContext.getDefault()`), a [default KeyManager](#) is created. Typically, there is a single key manager that 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.

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

## KeyManagerFactory Class

`javax.net.ssl.KeyManagerFactory` 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 which can return a basic X.509 key manager. Because it is provider-based, additional factories can be implemented and configured to provide additional or alternate 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:

```

public static KeyManagerFactory
    getInstance(String algorithm);

public static KeyManagerFactory
    getInstance(String algorithm,
               String provider);

public static KeyManagerFactory
    getInstance(String algorithm,
               Provider provider);

```

A sample algorithm name string is:

```
"SunX509"
```

A sample call is the following:

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

The above call will create 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);
```

You should call whichever `init` method is appropriate for the `KeyManagerFactory` you are using. (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 in order to initialize the `KeyManagerFactory` and thus the first `init` method is the appropriate one to call. The `KeyManagerFactory` will query the `KeyStore` for information on 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.

In some cases, initialization parameters other than a `KeyStore` and password may be needed by a provider. Users of that particular 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 are capable of providing access to authentication material without having to be 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 indicated above, the `SunJSSE` provider supports a "SunX509" factory that must be initialized with a `KeyStore` parameter.

## **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. In order 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 those supplied by the `SunJSSE` provider). You could also implement your own that delegates to a factory-generated key manager. For example, you might do this in order to filter the resulting keys and query an end-user through a graphical user interface.

### **Creating Your Own x509KeyManager**

If the default `x509KeyManager` behavior isn't suitable for your situation, you can create your own `x509KeyManager` in a way similar to that shown in [Creating Your Own x509TrustManager](#).

### **x509ExtendedKeyManager Class**

The `X509ExtendedKeyManager` abstract class is an implementation of the `X509KeyManager` interface which 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 a instance of the `X509ExtendedKeyManager` class, it will not work with the `SSLEngine` class.

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

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

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

## Relationships Between TrustManagers and KeyManagers

Historically, there has been confusion regarding the jobs of `TrustManagers` and `KeyManagers`. In summary, here are the primary responsibilities of each manager type:

Type	Function
<code>TrustManager</code>	Determines whether the remote authentication credentials (and thus the connection) should be trusted.
<code>KeyManager</code>	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.

### SSLParameters Class

`SSLParameters` encapsulates things that affect a TLS connection:

- The list of ciphersuites to be accepted in an SSL/TLS handshake
- The list of protocols to be allowed
- The endpoint identification algorithm during SSL/TLS handshaking
- The algorithm constraints
- Whether SSL/TLS servers should request or require client authentication

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

- `getSSLParameters()` in `SSLSocket`, `SSLServerSocket` and `SSLSEngine`
- `getDefaultSSLParameters()` and `getSupportedSSLParameters()` in `SSLContext`

Assign `SSLParameters` with the `setSSLParameters()` method in `SSLSocket`, `SSLServerSocket`, or `SSLSEngine`.

## SSLSessionContext Interface

A `javax.net.ssl.SSLSessionContext` is a grouping of [SSLSessions](#) associated with a single entity. For example, it could be associated with a server or client that participates in many sessions concurrently. The methods on 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.

## SSLSessionBindingListener Interface

`javax.net.ssl.SSLSessionBindingListener` is an interface implemented by objects which want to be notified when they are being bound or unbound from an [SSLSession](#).

## SSLSessionBindingEvent Class

A `javax.net.ssl.SSLSessionBindingEvent` is the event communicated to an [SSLSessionBindingListener](#) when it is bound or unbound from an [SSLSession](#).

## HandShakeCompletedListener Interface

`javax.net.ssl.HandShakeCompletedListener` is an interface implemented by any class which wants to receive notification of the completion of an SSL protocol handshake on a given `SSLSocket` connection.

## HandShakeCompletedEvent Class

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

## HostnameVerifier Interface

If the SSL/TLS implementation's standard hostname verification logic fails, the implementation will call the `verify` method of the class which implements this interface and is assigned to this `HttpsURLConnection` instance. If the callback class can determine that the hostname is acceptable given the parameters, it should report that the connection should be allowed. An unacceptable response will cause 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.sun.com/").openConnection());
urlc.setHostnameVerifier(new MyHostnameVerifier());
```

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

## 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 and 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 `java.security.cert.X509Certificate`, not `javax.security.cert.X509Certificate`.

## AlgorithmConstraints Interface

The Java SE 7 release includes an interface, `java.security.AlgorithmConstraints`, 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`, `SIGNATURE`, and more.

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 using a new method, `setAlgorithmConstraints()`. The current `AlgorithmConstraints` object for an `SSLParameters` object is retrieved with `getAlgorithmConstraints()`.

## Previous (JSSE 1.0.x) Implementation Classes and Interfaces

In previous (1.0.x) versions of JSSE, there was a reference implementation whose classes and interfaces were provided in the `com.sun.net.ssl` package.

As of v1.4, JSSE has been integrated into the J2SDK. The classes formerly in `com.sun.net.ssl` have been promoted to the `javax.net.ssl` package and are now a part of the standard JSSE API.

For compatibility purposes the `com.sun.net.ssl` classes and interfaces still exist, but have been deprecated. Applications written using them can run in the J2SDK v1.4 and later without being recompiled. This may change in a future release; these classes/interfaces may be removed. Thus, all new applications should be written using the `javax` classes/interfaces.

For now, applications written using the `com.sun.net.ssl` API can utilize *either* JSSE 1.0.2 providers (ones using `com.sun.net.ssl`) *or* JSSE providers written for the J2SDK v1.4 and later (ones using the `javax` API). However, applications written using the JSSE API in the J2SDK 1.4 and later can only utilize JSSE providers written for the J2SDK 1.4 and later. There more recent releases contain some new functionality and attempting to access such functionality on a provider that doesn't supply it wouldn't work. `SunJSSE`, provided with the JDK from Oracle, is a provider written using the `javax` API.

You can still obtain a `com.sun.net.ssl.HttpURLConnection` if you update the URL search path by setting the `java.protocol.handler.pkgs` System property as you did when using JSSE 1.0.2. For more information, see [Code Using HttpURLConnection Class...](#) in the Troubleshooting section.

# Customizing JSSE

## The Installation Directory <java-home>

The term <java-home> is used throughout this document to refer to the directory where the Java SE 6 Runtime Environment (JRE) is installed. It is determined based on whether you are running JSSE on a JRE with or without the Java SDK installed. Java SE 6 SDK includes the JRE, but it is located in a different level in the file hierarchy.

The following are some examples of which directories <java-home> refers to:

- On Solaris, if the Java SE 6 SDK is installed in `/home/user1/jdk1.6.0`, then <java-home> is `/home/user1/jdk1.6.0/jre`
- On Solaris, if JRE is installed in `/home/user1/jre1.6.0` and the Java 2 SDK is *not* installed, then <java-home> is `/home/user1/jre1.6.0`
- On Microsoft Windows platforms, if the Java SE 6 SDK is installed in `C:\jdk1.6.0`, then <java-home> is `C:\j2k1.6.0\jre`
- On Microsoft Windows platforms, if the JRE is installed in `C:\jre1.6.0` and the Java SE 6 SDK is *not* installed, then <java-home> is `C:\jre1.6.0`

## Customization

JSSE includes an implementation that all users can utilize. If desired, it is also possible to customize a number of aspects of JSSE, plugging in different implementations or specifying the default keystore, and so on. The table that follows summarizes which aspects can be customized, what the defaults are, and which mechanisms are used to provide customization. The first column of the table provides links to more detailed descriptions of each designated aspect and how to customize it.

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

---

**IMPORTANT NOTE: Many of the properties shown in this table are currently utilized by the JSSE implementation, but there is no guarantee that they will continue to have the same names and types (system or security) or even that they will exist at all in future releases. All such properties are flagged with an "\*" . They are documented here for your convenience for use with the JSSE implementation.**

---

### JSSE Customization

--	--	--

Customizable Item	Default	How To Customize
<a href="#">X509Certificate implementation</a>	X509Certificate implementation from Oracle	cert.provider.x509v1 security property
<a href="#">HTTPS protocol implementation</a>	Implementation from Oracle	java.protocol.handler.pkgs system property
<a href="#">provider implementation</a>	SunJSSE	A security.provider.n= line in security properties file. See description.
<b>default SSLSocketFactory implementation</b>	SSLSocketFactory implementation from Sun Microsystems.	** ssl.SocketFactory.provider security property
<b>default SSLServerSocketFactory implementation</b>	SSLServerSocketFactory implementation from Sun Microsystems.	** ssl.ServerSocketFactory.provider security property
<a href="#">default keystore</a>	No default.	* javax.net.ssl.keyStore system property Note that 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).
<a href="#">default keystore password</a>	No default.	* javax.net.ssl.keyStorePassword system property
<a href="#">default keystore provider</a>	No default.	* javax.net.ssl.keyStoreProvider system property
<a href="#">default keystore type</a>	KeyStore.getDefaultType()	* javax.net.ssl.keyStoreType system property
<a href="#">default truststore</a>	jssecacerts, if it exists. Otherwise, cacerts	* javax.net.ssl.trustStore system property

<a href="#"><u>default truststore password</u></a>	No default.	* <code>javax.net.ssl.trustStorePassword</code> system property
<a href="#"><u>default truststore provider</u></a>	No default.	* <code>javax.net.ssl.trustStoreProvider</code> system property
<a href="#"><u>default truststore type</u></a>	<code>KeyStore.getDefaultType()</code>	* <code>javax.net.ssl.trustStoreType</code> system property Note that the value <code>NONE</code> may be specified. This setting is appropriate if the truststore is not file-based (for example, it resides in a hardware token.)
<a href="#"><u>default key manager factory algorithm name</u></a>	<code>SunX509</code>	<code>ssl.KeyManagerFactory.algorithm</code> security property
<a href="#"><u>default trust manager factory algorithm name</u></a>	<code>PKIX</code>	<code>ssl.TrustManagerFactory.algorithm</code> security property
<a href="#"><u>disabled certificate verification cryptographic algorithms</u></a>	<code>MD2</code>	<code>jdk.certpath.disabledAlgorithms</code> security property
<a href="#"><u>disabled cipher suite cryptographic algorithms</u></a>	No default.	<code>jdk.tls.disabledAlgorithms</code> security property
<b>default proxy host</b>	No default.	* <code>https.proxyHost</code> system property
<b>default proxy port</b>	80	* <code>https.proxyPort</code> system property
<b>Server Name Indication option</b>	<code>true</code>	* <code>jsse.enableSNIExtension</code> system property. Server Name Indication (SNI) is a TLS extension, defined in <a href="#">RFC 4366</a> . 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 SSL/TLS vendors may not be able handle SSL/TLS extensions. In this case, set this property to <code>false</code> to disable the SNI extension.
<b>default ciphersuites</b>	Determined by the socket factory.	* <code>https.cipherSuites</code> system property. This contains a comma-separated list of cipher suite names specifying which cipher suites to enable

		for use on this <code>HttpsURLConnection</code> . See the <a href="#">SSLSocket.setEnabledCipherSuites(String[])</a> method.
<b>default handshaking protocols</b>	Determined by the socket factory	* <code>https.protocols</code> system property. This contains a comma-separated list of protocol suite names specifying which protocol suites to enable on this <code>HttpsURLConnection</code> . See the <a href="#">SSLSocket.setEnabledProtocols(String[])</a> method.
<b>default https port</b>	443	* Customize via <code>port</code> field in the https URL.
<b><a href="#">JCE encryption algorithms used by SunJSSE provider</a></b>	SunJCE implementations	Give alternate JCE algorithm provider(s) a higher preference order than the SunJCE provider
<b>defaultly sizing buffers for large SSL/TLS packets</b>	No default.	* <code>jsse.SSLEngine.acceptLargeFragments</code> system property By setting this system property to <code>true</code> , <code>SSLSession</code> will size buffers to handle <a href="#">large data packets</a> by default. This may cause applications to allocate unnecessarily large <code>SSLEngine</code> buffers. Instead, applications should <a href="#">dynamically check for buffer overflow conditions</a> and resize buffers as appropriate.
<b><a href="#">Allow Unsafe SSL/TLS Renegotiations</a></b>	<code>false</code>	* <code>sun.security.ssl.allowUnsafeRenegotiation</code> system property. Setting this system property to <code>true</code> permits full (unsafe) legacy renegotiation.
<b><a href="#">Allow Legacy Hello Messages (Renegotiations)</a></b>	<code>true</code>	* <code>sun.security.ssl.allowLegacyHelloMessages</code> system property. Setting this system property to <code>true</code> allows the peer to handshake without requiring the proper RFC 5746 messages.

\* This property is currently used by the JSSE implementation. It is not guaranteed to be examined and used by other implementations. If it *is* examined by another implementation, 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.

Note that some items are customized by setting `java.lang.System` properties while others are customized by setting `java.security.Security` properties. The following sections explain how to set values for both types of properties.

### How to Specify a `java.lang.System` Property

Some aspects of JSSE may be customized 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", type 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");
```

substituting the appropriate property name and value. 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. (See [Java Rich Internet Applications Development and Deployment](#) for more information.)
  - Use the Java Control Panel to set the Runtime Environment Property on a local/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. (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. (See [java arguments](#).)
  - To set the property in a specific Java Web Start application or applet using the new Plugin2 (6u10+), use the JNLP "property" sub-element of the "resources" element. (See [resources Element](#).)

## How to Specify a `java.security.Security` Property

Some aspects of JSSE may be customized 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
```

where `<java-home>` refers to the directory where the JRE runtime software is installed, as described in [The Installation Directory <java-home>](#).

To specify a security property value in the security properties file, you add a line of the following form:

```
propertyName=propertyValue
```

For example, suppose you want to specify a different key manager factory algorithm name than the "SunX509" default. You do this by specifying the algorithm name as the value of a security property named `ssl.KeyManagerFactory.algorithm`. Suppose you want to set the value to "MyX509". To do so, place the following in 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");
```

substituting the appropriate property name and value. For example, a `setProperty` call corresponding to the previous example for specifying the key manager factory algorithm name would be:

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

## 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 alternate 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, you should place the following in the security properties file:

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

## Specifying an Alternate 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 alternate 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 to previous JSSE users:** In past Sun JSSE releases, you had to set the `java.protocol.handler.pkgs` `system` property during JSSE installation. This step is no longer required unless you wish to obtain an instance of `com.sun.net.ssl.HttpsURLConnection`. For more information, see [Code Using HttpsURLConnection Class...](#) in the Troubleshooting section.

## Customizing the Provider Implementation

The J2SDK 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 on providers and engine classes, see the "Design Principles" and "Concepts" sections of the [Java Cryptography Architecture Reference Guide](#).

In order to be used, a provider must be registered, either statically or dynamically. You do not need to register the "SunJSSE" provider because it is pre-registered. 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). The order is 1-based; 1 is the most preferred, followed by 2, and so on.

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

To register a provider, add the above line to the security properties file, replacing *providerClassName* with the fully qualified name of the provider class and substituting *n* with the priority that you would like to assign to the provider.

The standard security provider and the SunJSSE provider shipped with the Java SE 6 platform 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 utilize another JSSE provider, add a line registering the alternate provider, giving it whatever preference order you prefer.

You can have more than one JSSE provider registered at the same time. They 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, 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 Key and Trust Stores, Store Types, and Store Passwords

Whenever a default `SSLConnectionFactory` or `SSLServerConnectionFactory` is created (via a call to `SSLConnectionFactory.getDefault` or `SSLServerConnectionFactory.getDefault`), and this default `SSLConnectionFactory` (or `SSLServerConnectionFactory`) 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 `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, the server's `KeyManager` keystore is not necessary. And, unless the server requires client authentication, the peer acting as the client will not need a `KeyManager` keystore. Thus, in these situations it may be okay if there is no `javax.net.ssl.keyStore` system property value 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 doesn't, then no truststore is utilized. 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). See [The Installation Directory <java-home>](#) for information as to what `<java-home>` refers to. Finally, if a truststore is still not found, then the truststore managed by the `TrustManager` will be a new empty truststore.

---

**IMPORTANT NOTE: The JDK ships with a limited number of trusted root certificates in the `<java-home>/lib/security/cacerts` file. As documented in [keytool](#), it is your responsibility to maintain (that is, add/remove) the certificates contained in this file if you use this file as a truststore.**

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

---

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

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

**Important 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, that implementation should handle them in the same manner as the JSSE reference implementation does, as described herein.

## Customizing the Default Key and Trust Managers

As noted in [Customizing the Default Key and Trust Stores, Store Types, and Store Passwords](#), whenever a default `SSLConnectionFactory` or `SSLServerConnectionFactory` is created, and this default `SSLConnectionFactory` (or `SSLServerConnectionFactory`) 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 a `KeyManager/TrustManager` implementation for managing the specified keystore/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, 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 `KeyManagers`, one `KeyManager` for each type of key material.) If there is no such security property value specified, 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` 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, 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 `TrustManagers`, one `TrustManager` for each type of trust material.) If there is no such security property value specified, 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` it specifies is a `javax.net.ssl.X509TrustManager` implementation.

**Important 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, that implementation should handle them in the same manner as the JSSE reference implementation does, as described herein.

## Disabled Cryptographic Algorithms

The cryptographic hash algorithm MD2 is no longer considered secure. The Java SE 7 release includes two new security properties and a new API that support disabling specific cryptographic algorithms.

The `jdk.tls.disabledAlgorithms` property applies to TLS handshaking, while the `jdk.certpath.disabledAlgorithms` property applies to certification path processing.

For example, the default value of `jdk.certpath.disabledAlgorithms` is MD2. This means that any certificate signed with MD2 is not acceptable.

Each security property contains a list of cryptographic algorithms that will not be used during certification path processing. The exact syntax of the properties is described in the `jre/lib/security/java.security` file, but is briefly summarized here.

The security property contains a list of cryptographic algorithms that must not be used. The algorithm names are separated by commas. Furthermore, you can also specify certain key sizes that cannot be used.

For example, the following line in `java.security` specifies that the MD2 and DSA algorithms must not be used for certification path processing. Furthermore, RSA is disabled for key sizes less than 2048 bits.

```
jdk.certpath.disabledAlgorithms=MD2, DSA, RSA keySize < 2048
```

## Customizing the Encryption Algorithm Providers

As of the Java SE 5 release, the `SunJSSE` provider uses the `SunJCE` implementation for all its cryptographic needs. While it is recommended that you leave the Sun provider at its regular position, you can use implementations from other JCA/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. (See [The Installation Directory <java-home>](#) for information about what `<java-home>` refers to.)

## Note for People Implementing Providers

The transformation strings used when SunJSSE calls `Cipher.getInstance()` are "RSA/ECB/PKCS1Padding", "RC4", "DES/CBC/NoPadding", and "DESede/CBC/NoPadding". For further information on the Cipher class and transformation strings see the [Cryptography Specification](#).

# Transport Layer Security (TLS) Renegotiation Issue

## Introduction

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. This section describes the situation in much more detail, along with interoperability issues when communicating with the older implementations which do not contain this protocol fix.

The vulnerability allowed for Man-In-The-Middle (MITM) type attacks where chosen plain text could be injected as a prefix to a TLS connection. This vulnerability does not allow an attacker to decrypt or modify the intercepted network communication once the client and server have successfully negotiated a session between themselves. This vulnerability has been disclosed at:

- [Authentication Gap in TLS Renegotiation](#) - posted on Marsh Ray's blog, [Extended Subset](#), November 5th, 2009.

and additional information is available at:

- [CVE-2009-3555](#) - posted on Mitre's [Common Vulnerabilities and Exposures List](#), 2009.
- [Understanding the TLS Renegotiation Attack](#) - posted on Eric Rescorla's blog, [Educated Guesswork](#), November 5th, 2009.

## Phased Approach To Fixing This Issue

The fix for this issue was handled in two phases:

- Phase 1: Until a protocol fix could be developed, an interim fix which disabled SSL/TLS renegotiations by default, was made available in the [March 30, 2010 Java SE and Java for Business Critical Patch Update](#).
- Phase 2: The IETF issued [RFC 5746](#) which addresses the renegotiation protocol flaw. A fix which implements RFC 5746 and supports secure renegotiation is included in the following releases:

JDK Family	Vulnerable Releases	Phase 1 Fix (Disable Reneg.)	Phase 2 Fix (RFC 5746)
JDK and JRE 6	Update 18 and earlier	Updates 19-21	Update 22
JDK and JRE 5.0	Update 23 and earlier	Updates 24-25	Update 26
SDK and JRE 1.4.2	Update 25 and earlier	Updates 26-27	Update 28

**NOTE:** In the Phase 2 default configuration, there is no impact to applications that do not require renegotiations. Applications that require a renegotiation (e.g. web servers that initially allow for anonymous client browsing, but later require SSL/TLS authenticated clients):

- will not be impacted if the peer is also RFC 5746-compliant.
- will be impacted if the peer has not been upgraded to RFC 5746 (see next section for details).

## Description of Phase 2 Fix

The SunJSSE implementation reenables renegotiations by default for connections to RFC 5746 compliant peers. That is, both the client and server **must support RFC 5746** in order to securely renegotiate. SunJSSE provides some interoperability modes for connections with peers that have not been upgraded, but users are **strongly encouraged to update both their client and server implementations as soon as possible**.

With the Phase 2 fix, SunJSSE now has three "renegotiation interoperability modes." Each mode fully supports RFC 5746's secure renegotiation, but has these added semantics when communicating with an unupgraded peer:

1. **Strict mode:** Requires both client and server be upgraded to RFC 5746 and send the proper RFC 5746 messages. If not, the initial (or subsequent) handshaking will fail and the connection will be terminated.
2. **Interoperable mode (default) :** Use of the proper RFC 5746 messages is optional, however legacy (original SSL/TLS specifications) renegotiations are disabled if the proper messages are not used. Initial legacy connections are still allowed, but legacy renegotiations are disabled. This is the best mix of security and interoperability, and is the default setting.
3. **Insecure mode:** Permits full legacy renegotiation. Most interoperable with legacy peers but vulnerable to the original MITM attack.

The mode distinctions above only affect a connection with an unupgraded peer. Ideally, strict (full RFC 5746) mode should be used for all clients/servers, however it will take some time for all deployed SSL/TLS implementations to support RFC 5746, thus the interoperable mode will be the default for now.

Here is some additional interoperability information:

Client	Server	Mode
Updated	Updated	Secure Renegotiation in all modes.
Legacy[1]	Updated	<ul style="list-style-type: none"> <li>• <b>Strict:</b> If clients do not send the proper RFC 5746 messages, initial connections will immediately be terminated by the server (<code>SSLHandshakeException/handshake_failure</code>).</li> <li>• <b>Interoperable:</b> Initial connections from legacy clients allowed (missing RFC 5746 messages), but renegotiations will not be allowed by the server. [2][3]</li> <li>• <b>Insecure:</b> Connections and renegotiations with legacy clients are allowed, but are vulnerable to the original MITM attack.</li> </ul>
Updated	Legacy[1]	<ul style="list-style-type: none"> <li>• <b>Strict:</b> If the server does not respond with the proper RFC 5746 messages, the client will immediately terminate the connection (<code>SSLHandshakeException/handshake_failure</code>).</li> <li>• <b>Interoperable:</b> The client will not require the proper initial RFC 5746 message from the server, but renegotiations will not be allowed by the client. [2][3]</li> <li>• <b>Insecure:</b> Connections and renegotiations with legacy clients are allowed, but are vulnerable to the original MITM attack.</li> </ul>
Legacy[1]	Legacy[1]	Existing SSL/TLS behavior, vulnerable to the MITM attack.

[1] Legacy means the original SSL/TLS specifications (i.e. non-RFC 5746).

[2] SunJSSE Phase 1 implementations (see above) reject renegotiations unless specifically reenabled. If renegotiations are reenabled, they will be treated as Legacy by the RFC 5746-compliant peer since they do not send the proper RFC 5746 messages.

[3] In SSL/TLS, renegotiations can be initiated by either side. Like the Phase 1 fix, applications communicating with an unupgraded peer in Interoperable mode and that attempt to initiate renegotiation (via `SSLSocket.startHandshake()` or `SSLEngine.beginHandshake()`) will receive a `SSLHandshakeException` (`IOException`) and the connection will be shutdown (`handshake_failure`). Applications that receive a renegotiation request from a non-upgraded peer will respond according to the type of connection in place:

- **TLV1:** A warning Alert message of type "no\_renegotiation(100)" will be sent to the peer and the connection will remain open. Older versions of SunJSSE will shutdown the connection when a "no\_renegotiation" Alert is received.
- **SSLv3:** The application will receive a `SSLHandshakeException`, and the connection will be closed (`handshake_failure`). ("no\_renegotiation" is not defined in the SSLv3 spec.)

To set these modes, two system properties are used:

- **`sun.security.ssl.allowUnsafeRenegotiation`** Introduced in Phase 1, this controls whether legacy (unsafe) renegotiations are permitted.
- **`sun.security.ssl.allowLegacyHelloMessages`** Introduced in Phase 2, this allows the peer to handshake without requiring the proper RFC 5746 messages.

mode	<code>allowLegacyHelloMessages</code>	<code>allowUnsafeRenegotiation</code>
Strict	false	false
Interoperable (default)	true	false
Insecure	true	true

**WARNING:** It is not recommended to re-enable the insecure SSL/TLS renegotiation, as the vulnerability is once again present.

For information on how to configure a specific mode by setting a system property, see [How to Specify a java.lang.System Property](#).

## Workarounds/Alternatives to SSL/TLS Renegotiation

All peers should be updated to RFC 5746-compliant implementation as soon as possible. Even with this RFC 5746 fix, communications with unupgraded peers will be impacted if a renegotiation is necessary. Here are a few suggested options:

- Restructure the peer to not require renegotiation.

Renegotiations are typically used by web servers that initially allow for anonymous client browsing but later require SSL/TLS authenticated clients, or which may initially allow weak ciphersuites but later need stronger ones. The alternative is to require client authentication/strong ciphersuites during the **initial** negotiation. There are a couple of options for doing so:

- If an application has a "browse mode" until a certain point is reached and a renegotiation is required, one can restructure the server to eliminate the "browse mode" and require all initial connections be strong.
- Another alternative is to break the server into two entities, with the "browse mode" occurring on server, and a second for the more secure mode. When the point is reached, transfer any relevant information between the servers and.

Both of these options couple require a fair amount of work, but will not reopen the original hole.

- Set renegotiation interoperability mode to "insecure" using the system properties (see above for information and

warnings).

## Implementation Details

RFC 5746 defines two new data structures which are mentioned here for advanced users:

- a new pseudo-ciphersuite called the Signaling Cipher Suite Value (SCSV), "TLS\_EMPTY\_RENEGOTIATION\_INFO\_SCSV", and
- a new TLS extension called the "Renegotiation Info" (RI).

Either of these can be used to signal that an implementation is RFC 5746-compliant and can perform secure renegotiations. Please see the [IETF email discussion](#) from November 2009 to February 2010 for the relevant technical discussions.

RFC 5746 allows for clients to send either a SCSV or RI in the first ClientHello. For maximum interoperability, SunJSSE will use the SCSV by default, as a few TLS and SSL servers do not handle unknown extensions correctly. The presence of the SCSV in the enabled Cipher Suites (i.e.

`SSLSocket.setEnabledCipherSuites()/SSLEngine.setEnabledCipherSuites()` will determine whether the SCSV is sent in the initial ClientHello, or if an RI should be sent instead.

SSLv2 does not support SSL/TLS extensions. If the `SSLv2Hello` protocol is enabled, SCSV will be sent in the initial ClientHello.

## Description of the Phase 1 Fix

As mentioned above, the Phase 1 Fix was to disable renegotiations by default until a RFC 5746-compliant fix could be developed. Renegotiations could be reenabled by setting the `sun.security.ssl.allowUnsafeRenegotiation` system property. The Phase 2 fix uses the same system property, with the addition of the `sun.security.ssl.allowUnsafeRenegotiation` system property to require the use of RFC 5746 messages.

All applications should upgrade to the Phase 2 RFC 5746 fix as soon as possible.

# JCE and Hardware Acceleration/Smartcard Support

## Use of JCE

The [Java Cryptography Extension \(JCE\)](#) is a set of packages that provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Prior to Java SE 5, the SunJSSE provider could make use of JCE providers when configured to do so, but it still contained internal cryptographic code that did not use JCE. In Java SE 6, the SunJSSE provider uses JCE exclusively for all of its cryptographic operations and hence, is able to automatically take advantage of JCE features and enhancements, including JCE's newly added support for [PKCS#11](#). This allows the SunJSSE provider in Java SE 6 to be able to use hardware cryptographic accelerators for significant performance improvements and to use Smartcards as keystores for greater flexibility in key and trust management.

## Hardware Accelerators

Use of hardware cryptographic accelerators is automatic if JCE 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 JCE/JCA providers in the provider list. See the [PKCS#11 Guide](#) for details on how to configure the Oracle PKCS#11 provider.

## Configuring JSSE to use Smartcards as Keystores and Trust Stores

Support in JCE for PKCS#11 also enables access to Smartcards as a keystore. See the [Customization](#) section for details on how to configure the type and location of the keystores to be used by JSSE. To use a Smartcard as a keystore or trust store, 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 (e.g., "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 may be present in a Smartcard reader during the lifetime of a Java application and they may be protected using different passwords. The pre-J2SE 5 APIs and the `SunX509` key manager do not accommodate these requirements well. As a result, in Java SE 5, new APIs were introduced and a new `X509KeyManager` implementation was added to the `SunJSSE` provider.

The [java.security.KeyStore.Builder](#) class abstracts the construction and initialization of a `KeyStore` object. It supports the use of `CallbackHandlers` for password prompting and can be subclassed 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 new `X509KeyManager` implementation in the `SunJSSE` provider called "NewSunX509" supports these parameters. If multiple certificates are available, it also makes the effort to pick a certificate with the appropriate key usage and prefers valid to expired certificates.

Here is an example of 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");
```



```
principal="host/mach1.imc.org@IMC.ORG"
useKeyTab=true
keyTab=mach1.keytab
storeKey=true;
};
```

An example of how to 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.HttpURLConnection`, and `javax.net.HandshakeCompletedEvent`. Similarly, to get the identity that was sent to the peer (to identify the local entity), use `getLocalPrincipal()` 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`.

Prior to Java SE 5, JSSE applications used `getPeerCertificates()` and similar methods in `javax.net.ssl.SSLSession`, `javax.net.ssl.HttpURLConnection`, and `javax.net.HandshakeCompletedEvent` to obtain information about the peer. When the peer does not have any certificates, `SSLPeerUnverifiedException` is thrown. The behavior of these methods remain unchanged in Java SE 6, which means that if the connection was secured using a Kerberos cipher suite, these methods will throw `SSLPeerUnverifiedException`.

If the application needs to determine only the identity of the peer or identity sent to the peer, it should use the `getPeerPrincipal()` and `getLocalPrincipal()` methods, respectively. It should use `getPeerCertificates()` and `getLocalCertificates()` only if it needs to examine the contents of those certificates. Furthermore, it 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 `SocketPermissions` 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");
```

where *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");
```

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

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

where *tgtPrincipal* is 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 write 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" filename extension.

With the `SunJSSE` provider, you can access PKCS12 keys through the KeyStore API with a keystore type of "pkcs12" (or "PKCS12", the name is case-insensitive). In addition, you can list the installed keys and associated certificates using the `keytool` command with the `-storetype` option set to `pkcs12`. (See [Security Tools](#) for information about `keytool`.)

## Troubleshooting

### Configuration Problems

#### 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 the following sections for more information:

- [Relationship Between Classes](#)
- [TrustManager Interface](#)
- [KeyManager Interface](#)

**Cause 2:** The system clock is not set correctly.

**Solution 2:** If the clock is not set correctly, 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.

#### java.security.KeyStoreException: TrustedCertEntry not supported

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

**Cause 1:** We do not support storing trusted certificates in pkcs12 keystore. PKCS12 is mainly used to deliver private keys with the associated cert chains. It does not have any notion of "trusted" certificates. Note that 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 certs.

**Solution 1:** Use JKS (or JCEKS) 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 occurs:

```
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. (Note: A JDK vendor once shipped a keystore in an unknown format, and that caused this type of error.)

**Solution:** Check initialization parameters. Ensure any keystores specified are valid and that the passwords specified are correct. (One way you can check these things is by trying to use the [keytool](#) to examine the keystore(s) and the relevant contents.)

## Exception, "No available certificate corresponding to the SSL cipher suites which are enabled"

**Problem:** When I try 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, an RSA `keyEntry` must be available in the keystore. If no such key is available, this cipher suite cannot be used. If there are no available key entries for all of the cipher suites enabled, this exception is thrown.

**Solution:** Create key entries for the various cipher suite types, or use an anonymous suite. (Be aware that anonymous ciphersuites are inherently dangerous because they are vulnerable to "man-in-the-middle" attacks, see [RFC 2246](#).) Refer to the following sections to learn how to pass the correct keystore and certificates:

- [Relationship Between Classes](#)
- [Customizing the Default Key and Trust Stores, 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 ciphersuite. If the intersection of the client's ciphersuite set with the server's ciphersuite set is empty, then you will see this exception.

**Solution 1:** Configure the enabled cipher suites to include common ciphersuites, and be sure to provide an appropriate `keyEntry` for asymmetric ciphersuites. (See [Exception, "No available certificate..."](#) in this section.)

**Problem 2:** When using Netscape Navigator or Microsoft Internet Explorer (IE) 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, only DSA-based ciphersuites can be used. By default, Navigator and IE send only RSA-based ciphersuites. Since the intersection of client and server ciphersuite sets is empty, this exception is thrown.

**Solution 2:** To interact with Navigator or IE, you should create certificates that use RSA-based keys. To do this, you need to 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 the first access.

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

**Solution:** Provide an alternate generator of random numbers, or initialize ahead of time when the overhead won't 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 file for more information.

## Code Using `HttpsURLConnection` Class Throws `ClassCastException` in JSSE 1.0.x

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

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

When running under this release, this code returns a `javax.net.ssl.HttpsURLConnection` and throws a `ClassCastException`.

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

**Solution:** Previous releases of the JDK (now known as the Java SE 6 SDK) 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 this release, there is now 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.

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

or

```
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 it doesn't understand or with a protocol version number that it doesn't support.

**Solution:** Try adjusting the protocols in `SSLSocket.setEnabledProtocols`. For example, some older server implementations speak only SSLv3 and do not understand TLS. Ideally, these implementations should negotiate to

SSLv3, but some simply hangup. For backwards compatibility, some server implementations (such as SunJSSE) can send SSLv3/TLS `ClientHello`s encapsulated in a SSLv2 `ClientHello` packet. The SunJSSE provider supports this feature, but it is not enabled by default. If you wish to use this feature, call `setEnabledProtocols` to enable `SSLv2Hello`, which is the sending of encapsulated SSLv2 `ClientHello`s.

## SunJSSE can not find a JCA/JCE provider which supports a required algorithm and causes `NoSuchAlgorithmException`

**Problem:** A handshake is attempted, and fails when it can not 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 can't be found or loaded, or if the SunJCE provider has been deregistered from the `Provider` mechanism and an alternate implementation from JCE isn't available, this exception will be seen.

**Solution:** Ensure the SunJCE is available by checking 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());
```

## Debugging Utilities

JSSE provides dynamic debug tracing support. This is similar to the support used for debugging access control failures in the Java SE 6 platform. The generic Java dynamic debug tracing support is accessed with the system property `java.security.debug`, while the JSSE-specific dynamic debug tracing support is accessed with the system property `javax.net.debug`.

**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.

Here is a complete example of how to get a list of the debug options:

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

where `MyApp` is an application that uses some of the JSSE classes. `MyApp` will not run after the debug help information is printed, as the help code causes the application to exit.

Here are the current options:

```
all          turn on all debugging
```

`ssl`            turn on ssl debugging

The following can be used with `ssl`:

<code>record</code>	enable per-record tracing
<code>handshake</code>	print each handshake message
<code>keygen</code>	print key generation data
<code>session</code>	print session activity
<code>defaultctx</code>	print default SSL initialization
<code>sslctx</code>	print SSLContext tracing
<code>sessioncache</code>	print session cache tracing
<code>keymanager</code>	print key manager tracing
<code>trustmanager</code>	print trust manager tracing

`handshake debugging` can be widened with:

<code>data</code>	hex dump of each handshake message
<code>verbose</code>	verbose handshake message printing

`record debugging` can be widened with:

<code>plaintext</code>	hex dump of record plaintext
<code>packet</code>	print raw SSL/TLS packets

The `javax.net.debug` property value must specify 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 ":" or "," helps readability. It doesn't matter what separators you use, and the ordering of the option keywords is also not important.

For an introduction on reading this debug information, please refer to the guide, [Debugging SSL/TLS Connections](#).

## Examples

- To view all debugging messages:

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

- To view the hexadecimal dumps of each handshake message, you can type the following, where 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, you can type the following, where the commas are optional:

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

## Code Examples

The sections below describe the following code examples:

- [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](#)
    - [Configuration Requirements](#)
    - [Running SSLSocketClient](#)
    - [Running SSLSocketClientWithTunneling](#)
    - [Running SSLSocketClientWithClientAuth](#)
    - [Running ClassFileServer](#)
    - [Running SSLSocketClientWithClientAuth With ClassFileServer](#)
  - [Sample Code Illustrating HTTPS Connections](#)
    - [Running URLReader](#)

- [Running URLReaderWithOptions](#)
  - [Sample Code Illustrating a Secure RMI Connection](#)
  - [Sample Code Illustrating the Use of an SSL Engine](#)
    - [Running SSL Engine Simple Demo](#)
    - [Running the NIO-based server](#)
- [Creating a Keystore to Use with JSSE](#)
  - [Creating a Simple Keystore and Truststore](#)

## 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

#### Server Code for Unsecure Socket Communications

When writing 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) {
}
```

#### Client Code for Unsecure Socket Communications

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

### Server Code for Secure Socket Communications

When writing 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) {
}

```

### Client Code for Secure Socket Communications

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 Web site](#)
- [Use secure communications with RMI](#)
- [Illustrate SSLEngine usage](#)

When using 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 running the programs, be patient: you may not see any output for a while. If you run the programs with the system property `javax.net.debug` set to `all`, you will see more feedback. For an introduction on reading this debug information, refer to 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 samples files and to link to the text files. That page also has a zip file you can download to obtain all the samples files, which is helpful if you are viewing this documentation from the web.

The sections below describe the samples. See the [README](#) for further information.

## 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 in order to run `URLReader`, `SSLSocketClient` or `SSLSocketClientWithTunneling` with `ClassFileServer`.

If an authentication error occurs while attempting to send messages 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. ("testkeys" 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, an authentication error will occur. Be sure to use the `samplecacerts` truststore (which contains "localhost"s public key/cert), 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 Certification Authority 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`. (`ClassFileServer` uses a keystore containing the private key for "localhost" which 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 `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
```

(See [The Installation Directory <java-home>](#) for information about what `<java-home>` refers to.)

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

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, 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/certificate is also stored in the `samplecacerts` file.

## Running `SSLSocketClient`

The [SSLSocketClient.java](#) program demonstrates how to create a client to use 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 shipped. If you run it from behind a firewall, you will get an `UnknownHostException` because JSSE can't 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 (shown in blue) are optional. Also, be sure to 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 Web site `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` - The port parameter can be any available unused port number, for example, you can use the number 2001.
- `docroot` - This parameter indicates the directory on the server that contains the file you wish to retrieve. For example, on Solaris, you can use `/home/userid/` (where `userid` refers to your particular user id), while on Microsoft Windows systems, you can use `c:\`.
- `TLS` - This is an optional parameter. When used, it indicates that the server is to use SSL or TLS.
- `true` - This is an optional parameter. When used, client authentication is required. This parameter is only consulted if the `TLS` parameter is set.

Note 1: The `TLS` and `true` parameters are optional. If you leave them off, indicating that just an ordinary (not TLS) file server should be used, without authentication, nothing happens. This is because one side (the client) is trying to negotiate with TLS, while the other (the server) isn't, so they can't communicate.

Note 2: The server expects GET requests in the form "GET /...", where "..." is the path to the 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` - This is the hostname of the machine you are using to run `ClassFileServer`.
- `port` - This is the same port you specified for `ClassFileServer`.
- `requestedfilepath` - This parameter indicates the path to the file 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

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

NOTE: you can modify the other `SSLClient*` application's "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 which 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 might be:

```
"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. If you want a different https protocol implementation to be used, you can 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 `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 will be utilized. If you want to use a different implementation, you must 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, you must set the system properties `https.proxyHost` and `https.proxyPort`. For example, to use the proxy host "webproxy" on port 8080, you can use the following options to 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");
```

Note: When running on Windows 95 or Windows 98, the maximum number of characters allowed in an MS-DOS prompt may not be enough to include all the command-line options. If you encounter this problem, either create a .bat file with the entire command or add the system properties to the source code and recompile the source code.

## Running URLReaderWithOptions

The [URLReaderWithOptions.java](#) program is essentially the same as URLReader, 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, type the following command (all on one line):

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

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

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

If you are running behind a firewall, 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 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 RMI, see the [Java RMI documentation](#). This Web page points to RMI tutorials and other information about RMI.

## Sample Code Illustrating the Use of an SSL Engine

SSL Engine was introduced in the Java SE 5 release of the Java 2 Platform to give 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 `SSLEngines` which exchange SSL/TLS messages via common `ByteBuffer`s. 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 `SSLEngine`'s current state. This example doesn't 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`.

## Running the NIO-based Server

---

**Note:** The server example discussed in this section is included in the Java SE Development Kit 6. You can find the code bundled in the `<jdk-home>/samples/nio/server` directory.

---

To fully exploit the flexibility provided by `SSLEngine`, one must first understand complementary API's such as I/O and threading models.

An I/O model that large-scale application developers find of use is NIO `SocketChannels`. NIO was introduced in part to solve some of the scaling problem inherent in the `java.net.Socket` API. `SocketChannels` have many different modes of operation including:

- blocking
- nonblocking
- nonblocking with Selectors

Sample code for a bare-bones HTTP server is provided that not only demonstrates many of the new NIO APIs, and also shows how `SSLEngine` can be employed to create a secure HTTPS server. The server is not production quality, but does show many of these new APIs in action.

Inside the sample directory is a `README.txt` file which introduces the server, explains how to build and configure, and provides a brief overview of the code layout. The files of most interest for `SSLEngine` users are `ChannelIO.java` and `ChannelIOSecure.java`.

## Creating a Keystore to Use with JSSE

### Creating a Simple Keystore and Truststore

In this section, we'll use `keytool` to create a simple JKS keystore suitable for use with JSSE. We'll make a `keyEntry` (with public/private keys) in the keystore, then make a corresponding `trustedCertEntry` (public keys only) in a truststore. (For client authentication, you'll need to do a similar process for the client's certificates.) Note: Storing trust anchors in PKCS12 is not supported. Users should use JKS for storing trust anchors and PKCS12 for private keys.



```
-----END CERTIFICATE-----
```

Alternatively, you could generate Certificate Signing Request (CSR) with `-certreq` and send that to a Certificate Authority (CA) for signing, but again, that's 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 (shown in red). It also means that this file is not suitable as a `KeyManager`'s keystore.

```
% 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
```

Now run your applications with the appropriate key stores. This example assumes the default `X509KeyManager` and `X509TrustManager` are used, thus we will select the keystores using the system properties described in [Customization](#).

```
% 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:** In this example, we authenticated the server only. If client authentication is desired, you will need to provide a similar keystore for the client's keys, and an appropriate truststore for the server.

---

## Appendix A: Standard Names

The JDK Security API requires and uses a set of standard names for algorithms, certificate and keystore types. The specification names previously found here in Appendix A and in the other security specifications (JCA/CertPath/etc.) have been combined in the [Standard Names document](#). Specific provider information can be found in the [Oracle Provider Documentation](#).

# Appendix B: Provider Pluggability

JSSE in Java SE 6 is fully pluggable and does not restrict the use of third party JSSE providers in any way.

---

[Copyright ©](#) 1993, 2011, Oracle and/or its affiliates. All rights reserved.  
[Contact Us](#)