
Using JSSE for secure socket communication

Skill Level: Introductory

Greg Travis (mito@panix.com)
Programmer

30 Apr 2002

The JDK 1.4 Java Secure Socket Extension is easy enough to use -- once you understand all the steps necessary for the initial setup and configuration. In this tutorial Greg Travis offers cookbook-style instructions for creating and installing JSSE encryption keys in a client/server application environment. Upon completion of this tutorial, you will know how to easily convert any existing client/server application to use encryption, as well as how to create a secure application from scratch.

Section 1. About this tutorial

What is this tutorial about?

This tutorial explains the use of the Java Secure Socket Extension (JSSE) packages included in JDK 1.4. The complexity of using JSSE is not in the communication itself, but rather in the configuration. Before you can run your client/server software, you must create the keys needed by the encryption algorithms, and these keys must be properly loaded by your software before it can create secure sockets.

This tutorial provides cookbook-style instructions for creating and installing JSSE encryption keys in a client/server application environment. When you have completed this tutorial, you will know how to easily convert any existing client/server application to use encryption, as well as how to create a secure application from scratch.

Prerequisites

To follow the discussion in this tutorial, you need to know a few things about the Java language, starting with the basics of Java programming in a client/server environment (that is, working with classes, objects, threads, and so on). Because stream and socket communication is central to our discussion, you need to know how to use streams and sockets. In particular, you should know what a stream is and what it is used for. You should know how to create a `Socket` and a `ServerSocket`, how to get streams from each, and how to communicate using those streams. You also should know how to create and compile a `.java` file using the JDK or an IDE.

Two *developerWorks* tutorials, "Java sockets 101" and "Introduction to Java I/O," both accessible from [Resources](#), provide useful background information.

You do not need to know anything about encryption technology to complete this tutorial. While a comprehensive overview of encryption is beyond the scope of the discussion here, you will be given the information you need to create secure client/server connections within your applications.

Installation requirements

To run the examples in this tutorial, you need the following tools and components:

- JDK 1.4 from Sun Microsystems
- A development environment -- either an IDE or a suitable command-shell
- A computer on which you can run an Internet server

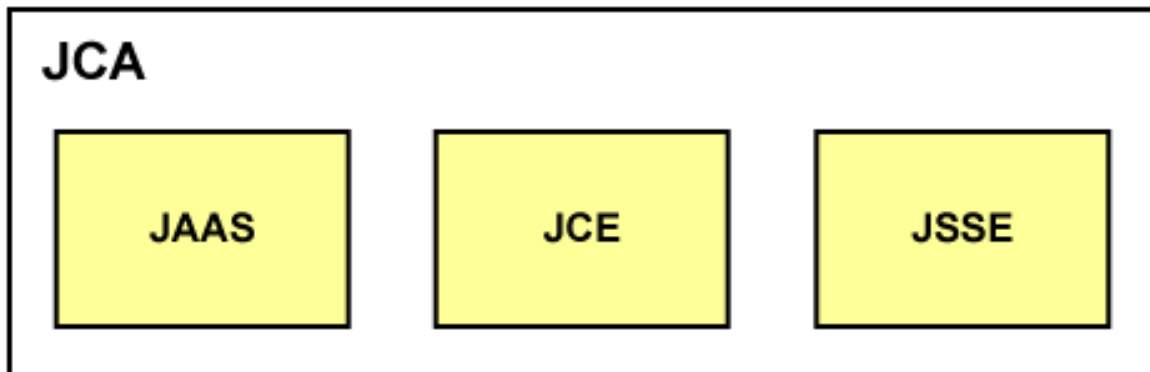
See [Resources](#) to download the JDK 1.4 and complete source files necessary for the completion of this tutorial.

Section 2. The Java Cryptography Architecture

Overview

The Java platform's security and encryption features have grown tremendously over the last few years. The JDK 1.4 (a.k.a. Merlin) release now comes bundled with many security-related packages, including the Java Cryptography Extension (JCE), the Java Secure Socket Extension (JSSE), and the Java Authentication and Authorization Service (JAAS). All of these components are pieces of the Java

Cryptography Architecture (JCA), as illustrated in the figure below:



In this tutorial we'll mostly be working with the JSSE component of the JCA.

The JCA and JSSE

One of the most important features of JCA is that it doesn't rely on any one particular encryption algorithm. Each well-known encryption algorithm has its advantages and disadvantages, and new ones are being developed all the time. The JCA allows new algorithms to be plugged in as they are developed. It uses the concept of the *cryptographic service provider* (CSP), which is something like a security plug-in. A CSP supplies the implementation of a particular algorithm. JDK 1.4 comes bundled with CSPs, including the SunJSSE, that provide many standard algorithms; altogether, these are sufficient for most uses.

JSSE provides secure socket communication for the Java 2 platform. More precisely, it implements Secure Socket Layer (SSL) and Transport Layer Security (TLS), two standardized protocols for implementing secure communications over the Internet. Both SSL and TLS rely on *public-key cryptography*, which is described in the next panel.

Public-key cryptography

One problem with many cryptographic algorithms is that they require the distribution of *shared keys*. A password is a good example of a shared key. The problem with shared keys is that they must be shared between communicating entities before secure communication can start. The sharing process, however, can be vulnerable to eavesdropping, which leads to a chicken-and-egg problem: before we can exchange data securely, we must first exchange secret keys securely.

This problem was solved in 1976 by Whitfield Diffie and Martin Hellman, with the creation of *public key cryptography*. In the Diffie-Hellman public-key system each

communicating party holds a pair of keys -- one public and one private. The private key is known only to the communicating party, while the public key can be given to anyone. Data encrypted using one of the keys can only be decrypted with the other.

Thus, if you want to create a message to be read only by a particular party, you use their public key to make the encryption, and they then use their private key to decrypt the message.

Likewise, if you encrypt a message with your private key, then anyone who has a copy of your public key can use it to decrypt the message. This assures the person on the receiving end that the message came from you and not someone else, since only you have your private key. A message that you have encrypted in this way bears your *digital signature*.

Certificates and certificate authority

A *certificate* is a public key that has been digitally signed by a trusted party in order to prove that it is a valid public key. This trusted party is called a *certification authority (CA)*. In a sense, the CA provides a testimonial that the public key really does belong to the person who owns it.

You can use commercial CAs for a fee, or you can create your own -- it all depends on how much authority you want to wield when proving your identity in the digital realm. If an entity signs its own public key, it's called a *self-signed certificate*. We use self-signed certificates throughout this tutorial.

SSL and TLS

As previously mentioned, the JSSE framework, along with the SunJSSE provider, implements the SSL and TLS protocol suites -- TLS being really just the newest version of SSL.

SSL uses public-key cryptography to exchange a set of shared keys, and then uses standard shared-key encryption to exchange data. The shared keys are used both for encrypting the data (making it unreadable by others) and for authenticating the data (ensuring that it hasn't come from an impostor).

The SSL handshake protocol

Before data can be sent across an SSL connection, the two ends must negotiate and exchange key information. This is called the *handshake protocol*. We won't go into much detail about the handshake protocol here, because it isn't necessary for our purposes. For our purposes, you need to know that the handshake involves the

following steps:

1. The server sends its certificate to the client and the client verifies the server certificate.
2. The client sends its certificate to the server and the server verifies the client certificate.
3. The client encrypts password information with the server's public key and sends it to the server. This password information is used by each end of the connection to generate identical secret keys, which will then be used to transmit data.

Client authentication (Step 2) is optional: the server *can* request that the client provide its certificate, but it is not required to make such a request. We will be using client authentication in our example.

Now that you have a basic understanding of the infrastructure of JCA and JSSE under your belt, we can move on to the more active part of the tutorial: working with a live example.

Section 3. Whiteboard: An example application

Overview

To demonstrate how JSSE works in the real world, we're going to employ a simple *distributed whiteboard system*. The whiteboard is a program that lets users write text messages onto a blank canvas. These text messages then appear on the canvases of other users who are connected to the same whiteboard server. This allows people in different locations to communicate.

Because we're more interested in the security aspects of this program than in the whiteboard functionality itself, we'll keep the application very simple. Our whiteboard will allow users to create text messages, but it will not allow them to delete messages. Mostly, we'll just pay attention to how the whiteboard implements JSSE, thus ensuring that messages can be securely sent between users.

In this section we discuss the basic structure of the whiteboard and how it implements JSSE. In the next section, we'll begin working directly with the code for the whiteboard application.

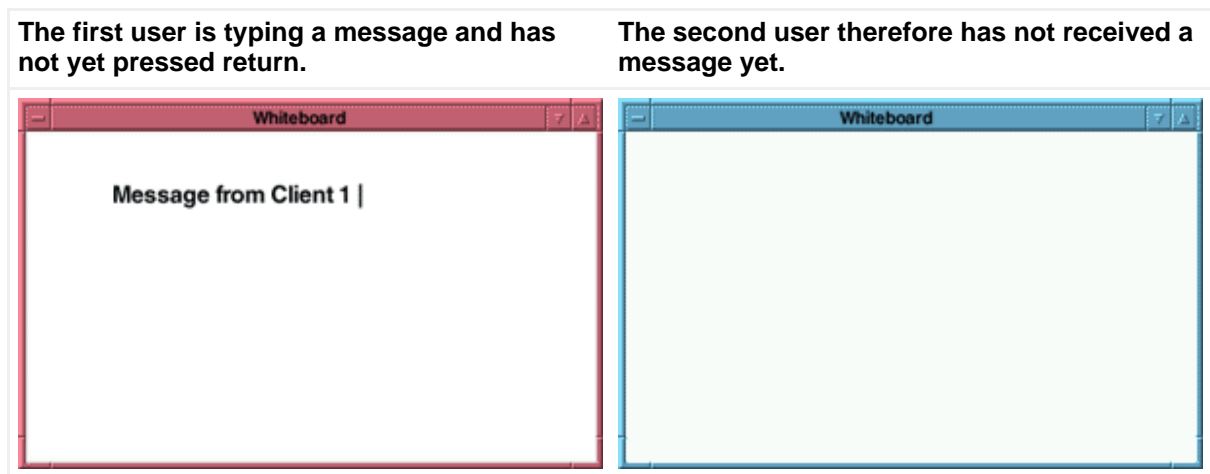
Using the whiteboard

The figure below illustrates the simple whiteboard we'll be working with. Each client's messages are shown in a different color.

The distributed whiteboard showing messages sent by different users



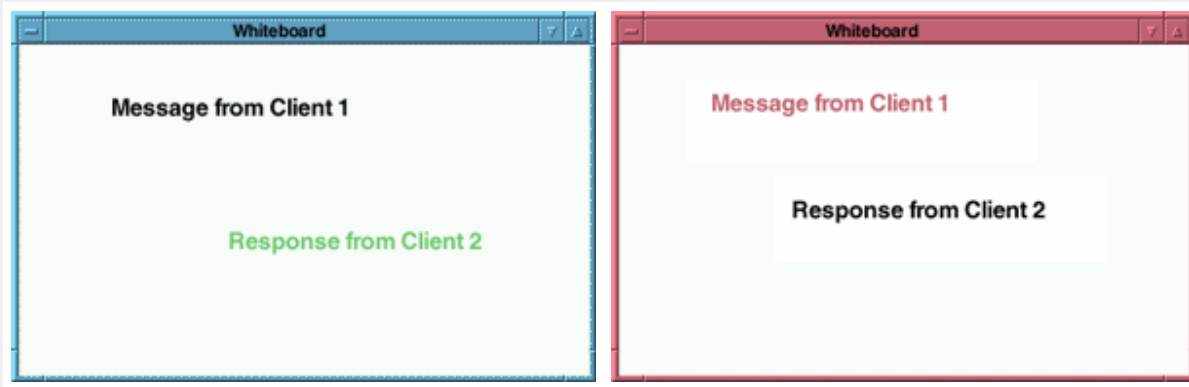
In the following two figures, you see two whiteboard client windows. Imagine that each of these windows is running on a different machine. The first user has clicked somewhere in his window and typed a message, but he has not yet pressed Return.



Now, imagine the first user presses Return, which causes his message to show up in the window of the second user. The second user then sends a response, as illustrated in the two figures below.



the second user.



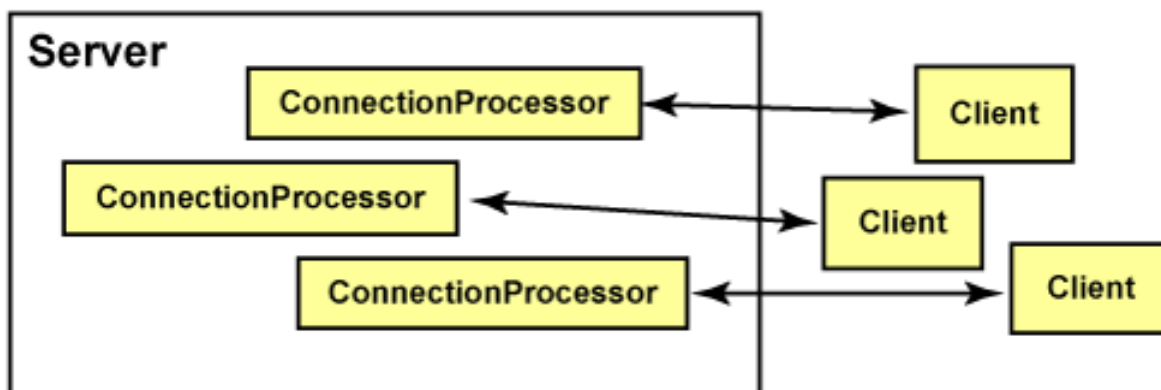
Each user sees his own text in black; other users' text appears in different colors assigned by the server.

The client/server structure

On the server side of the whiteboard application, we have a class called `Server`. This class listens for incoming connections on a specified port. Each time a connection comes in, the `Server` creates a `ConnectionProcessor` to process the connection. Processing a connection means receiving text messages and sending them back out to other clients. The `Server` gives each client one `ConnectionProcessor`.

When a client first starts up, it initiates a connection to the server. The client keeps this connection open throughout the whiteboard session. Each text message is sent along this connection.

The figure below illustrates the client/server processes for the whiteboard application.



Key files

As we already discussed, the whiteboard uses JSSE, and JSSE implements the SSL and TLS protocols. These protocols use public-key encryption to ensure the privacy of messages sent over the Internet. In a public-key encryption system, both clients and server must have a pair of keys, one public and one private. Before our whiteboard can even begin to mediate messages between users, we must generate these keys.

Once the keys have been generated, we'll provide the client side a file containing its public and private keys. It will also have a copy of the server's public key certificate. Keys are stored in a specially formatted file called a keystore.

The following table describes the keystore files we'll be using.

Keystore file	What it contains	Where it goes
client.private	The client's public/private key pair	Client side
server.public	The server's public key certificate	Client side
server.private	The server's public/private key pair	Server side
client.public	The client public key certificate	Server side

Key protection

The server also has a file containing its own public and private keys, as well as the client's public key certificate. Recall that public keys can be given out freely -- there's no need to hide them from any other party.

It is important that each end of the client/server connection has only the key files it needs to work properly. In particular, it's important that only the server has a copy of its own private key. In the wrong hands, this key could do much damage, since it would essentially allow a malicious entity to cloak itself under the server's identity.

Now that you have an idea of how the whiteboard is put together, we can begin working more directly with each of its components. In the next section, you'll learn how to generate and manage public/private key pairs in a client/server system.

Section 4. Key management

Overview

Key generation and manipulation is performed with the `keytool` program, which is included with the JSSE packages in JDK 1.4. `keytool` can be used for a variety of purposes. Here, we'll be using it to create public/private key pairs, and to extract the public key certificates from these pairs and place them in their own files.

Generating a key pair

The following `keytool` command is used to generate a new public/private key pair:

```
keytool -genkey -keystore [filename]
```

When you run this command, you will be asked a series of questions. These questions concern you as an entity (your name, organization, and the like). The information you provide will be used to create a self-signed certificate that associates the information with a public key and testifies to the authenticity of the association. You will also be asked to enter passwords for the keystore and, optionally, passwords for the key pair you are creating.

Working from the command line

Below is a complete command that generates a public/private key pair and specifies all the required entity information without asking you any questions about your identity; that information is provided directly on the command line. The table that follows explains each option in the command.

```
keytool -genkey -alias clientprivate -keystore client.private
-storetype JKS -keyalg rsa -dname "CN=Your Name, OU=Your
Organizational Unit, O=Your Organization, L=Your City, S=Your State,
C=Your Country" -storepass clientpw -keypass clientpw
```

Option	What it means
<code>-genkey</code>	Tells <code>keytool</code> to generate a key pair.
<code>-alias clientprivate</code>	Identifies the new key pair within the keystore.
<code>-keystore client.private</code>	Uses the file <code>client.private</code> as the keystore.

-storetype JKS	Declares the type of the keystore. JKS is the default.
-keyalg rsa	Declares the algorithm to be used; we're using the RSA public key algorithm, which is the default.
-dname "CN=Your Name..."	Provides information about the entity owning the key pair.
-storepass clientpw	Specifies the password for the entire keystore.
-keypass clientpw	Specifies the password for the new key pair.

Client and server key pairs

The first step in the preparation of our algorithm is to generate a public/private key pair for the client and another pair for the server. The following command will generate the file `client.private`, which is the key pair for the client:

```
keytool -genkey -alias clientprivate -keystore client.private
-storetype JKS -keyalg rsa -dname "CN=Your Name, OU=Your
Organizational Unit, O=Your Organization, L=Your City, S=Your State,
C=Your Country" -storepass clientpw -keypass clientpw
```

And here is the command to generate the file `server.private`, which is the key pair for the server:

```
keytool -genkey -alias serverprivate -keystore server.private
-storetype JKS -keyalg rsa -dname "CN=Your Name, OU=Your
Organizational Unit, O=Your Organization, L=Your City, S=Your State,
C=Your Country" -storepass serverpw -keypass serverpw
```

As mentioned in the section [Key files](#), the private client and server key files need to be installed in specific places. The `client.private` file is installed on the client side; the `server.private` file is installed on the server side.

Extracting public keys

The next step is to extract the public keys so they can be installed with the client and server. Specifically, the client-side software must have the public key of the server side, and vice versa.

To get the public keys, we extract them from the `client.private` and `server.private` files and place them in temporary files. Then, we insert them into their own keystores, called `client.public` and `server.public`, respectively.

The temporary key file, temp.key, is used to hold each of the public keys, temporarily, while it is being copied from the private keystore to the public keystore -- for example, from client.private to client.public. After completing the export/import procedure, you'll want to remove the temp.key from the current directory.

Export/import commands

We'll use the `keytool -export` command to extract the public key into a file, and then use the `keytool -import` command to insert it into a new keystore. Here's the command to extract the client's public key:

```
keytool -export -alias clientprivate -keystore client.private -file
temp.key -storepass clientpw
```

And here's the command to insert the client's private key into its own keystore:

```
keytool -import -noprompt -alias clientpublic -keystore client.public
-file temp.key -storepass public
```

We'll also extract and store the server's public key. Here's the command to extract the key:

```
keytool -export -alias serverprivate -keystore server.private -file
temp.key -storepass serverpw
```

And here's the command to place it in its own keystore:

```
keytool -import -noprompt -alias serverpublic -keystore server.public
-file temp.key -storepass public
```

The script `generatekeys.sh` (for UNIX) and `generatekeys.bat` (for DOS or Microsoft Windows) automatically generates the client and server key files for you and cleans up any temporary files. See [Resources](#) to download the script.

Installing the files

Before we move on to the next section, it's important to make sure that the key files are installed properly on the client and server ends. The following files should be installed on the client side:

- Client.class

- Client\$1.class
- Client\$2.class
- Client\$ClientCanvas.class
- Posting.class
- client.private
- server.public

And these files should be installed on the server side:

- Server.class
- Posting.class
- ConnectionProcessor.class
- client.public
- server.private

With the public/private key pairs properly installed on our client and server systems, we're ready to begin using the whiteboard application to exchange messages. In the next section, we'll examine the code for the whiteboard application itself.

Section 5. Using JSSE sockets

Overview

Our whiteboard is a client/server application that uses sockets to communicate with the server. Because we want our communications to be private, we'll be using JSSE secure sockets instead of regular sockets. Secure sockets are used the same way that regular sockets are, except that they transparently encrypt any data that passes through them.

Before we get into how JSSE creates and manages secure connections, we should review how regular, nonsecure connections are initiated and accepted.

Nonsecure sockets: A review

The following code fragment is typical for initiating a socket connection. This example creates a new `Socket` connection to port `port` at the remote computer `host`:

```
Socket socket = new Socket( host, port );
```

Similarly, the following code demonstrates how we listen for incoming connections. This example creates a `ServerSocket` listening on port `port`, and then enters an infinite loop, accepting and processing incoming connections:

```
ServerSocket serverSocket = new ServerSocket( port );  
while (true) {  
    Socket socket = serverSocket.accept();  
    doSomethingWithNewConnection( socket );  
}
```

Secure sockets work in a very similar manner, but before we can implement them for our example we must complete several steps. We'll go over these steps in the sections that follow.

Connection setup

To initiate a secure socket connection to a remote server, we must carry out the following steps:

1. Create a `SecureRandom`, a source of secure random numbers. Secure random numbers are numbers that are random enough that they will not make the encryption vulnerable to attack.
2. Create a `KeyStore` object containing the remote server's public key. This is read from `server.public`.
3. Create a `KeyStore` object containing the client's public/private key pair, including its public key certificate. This is read from `client.private`.
4. Create a `TrustManagerFactory` from the remote server's `KeyStore`. This is used to authenticate the remote server.
5. Create a `KeyManagerFactory` from the client's `KeyStore`. This is used for encrypting and decrypting data.
6. Create an `SSLContext` object, using the `KeyManagerFactory`, the `TrustManagerFactory`, and the `SecureRandom`.

7. Use the `SSLContext` to create an `SSLConnectionFactory`.
8. Use the `SSLConnectionFactory` to create an `SSLSocket`, which acts just like a regular `Socket`, except that it is secure.

Listening setup

To listen for incoming connections, we must carry out a similar set of steps:

1. Create a `SecureRandom`, a source of secure random numbers.
2. Create a `KeyStore` object containing the remote client's public key. This is read from `client.public`.
3. Create a `KeyStore` object containing the server's public/private key pair, including its public key certificate. This is read from `server.private`.
4. Create a `TrustManagerFactory` from the remote client's `KeyStore`. This is used to authenticate the remote client.
5. Create a `KeyManagerFactory` from the server's `KeyStore`. This is used for encrypting and decrypting data.
6. Create an `SSLContext` object, using the `KeyManagerFactory`, the `TrustManagerFactory`, and the `SecureRandom`.
7. Use the `SSLContext` to create an `SSLServerConnectionFactory`.
8. Use the `SSLServerConnectionFactory` to create an `SSLServerSocket`, which acts just like a regular `ServerSocket`, except that it is secure.
9. Call the `accept()` method of the `SSLServerSocket` to wait for an incoming connection.

It's all pretty complicated, but the process is the same each time, so it makes sense to follow along and see how it all works. In the sections that follow, we'll walk through the code that carries out these steps. We'll examine only the client-side process in detail, because the server-side process is nearly the same. Afterwards, we'll note the differences between the two sides.

Creating a `SecureRandom`

Creating a `SecureRandom` object is easy; just use the following lines in your code:

```
secureRandom = new SecureRandom();
secureRandom.nextInt();
```

The first line actually creates the `SecureRandom`. Creating a `SecureRandom` takes a lot of computation, and this computation may not be executed until it's actually used. By calling the `nextInt()` method here, we get the creation process going and ensure the delay will happen at the start of the program, not later when it might inconvenience us.

Creating keystores

Next, we need to create some `KeyStore` objects. We create an empty `KeyStore` using the static method `KeyStore.getInstance()`, and initialize it using its `load()` method, as illustrated below:

```
private void setupServerKeystore()
    throws GeneralSecurityException, IOException {
    serverKeystore = KeyStore.getInstance( "JKS" );
    serverKeystore.load( new FileInputStream( "server.public" ),
        passphrase.toCharArray() );
}
```

Note that we've created a `KeyStore` of type "JKS"; this is the standard keystore format used in JCA.

Reading keystores

In the previous section, we read the key information from `server.public`, which contains the server side's public key. We also need to read the client key pair from `client.private`, as shown here:

```
private void setupClientKeystore()
    throws GeneralSecurityException, IOException {
    clientKeystore = KeyStore.getInstance( "JKS" );
    clientKeystore.load( new FileInputStream( "client.private" ),
        "public".toCharArray() );
}
```

Setting up factories

The next step is to use the `KeyStore` objects we've created to initialize key and trust manager factories. We'll create a `TrustManagerFactory` from the server keystore; this will be used to authenticate (that is, begin to trust) the remote server:

```
TrustManagerFactory tmf = TrustManagerFactory.getInstance( "SunX509" );
tmf.init( serverKeyStore );
```

Note that the `TrustManagerFactory` is of type "SunX509"; 509 is the name of the certification protocol we're using throughout this program. In the second code line, the `TrustManagerFactory` is loaded with the server's keystore.

We must also create a `KeyManagerFactory` from the client's `KeyStore`, as shown below:

```
KeyManagerFactory kmf = KeyManagerFactory.getInstance( "SunX509" );
kmf.init( clientKeyStore, passphrase.toCharArray() );
```

Creating an `SSLContext`

We're getting close to finalizing our secure socket setup, so stay with me! The next step is to create an `SSLContext`. An `SSLContext` contains all the key and certificate information we've mentioned so far, and is used to create an `SSLSocketFactory`, which in turn creates secure sockets.

Once you've created an `SSLContext` at the start of an application, you can use it for each connection you need to make, as long as each connection uses the same keys.

To create an `SSLContext`, we use our factories and the `SecureRandom`, as shown here:

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

Note that we've created an `SSLContext` of type "TLS". As you suspect, this stands for Transport Layer Security, which is the new name for the Secure Sockets Layer, or SSL. We then initialize it with the `TrustManagerFactory` and `KeyManagerFactory` objects we created a few steps back.

Establishing a connection

We're at the last step. All the work we've done so far has given us an `SSLContext`, which we're going to use to make a connection to a remote machine, as shown in the code sample below:

```
SSLSocketFactory sf = sslContext.getSocketFactory();  
SSLSocket socket = (SSLSocket)sf.createSocket( host, port );
```

We've just made a secure connection to port *port* on the machine *host*.

Server-side setup

Setting up the server side is more or less the same as setting up the client side, so we won't go over it in detail. Of course, the server reads its key information from `client.public` and `server.private`, rather than from `server.public` and `client.private`.

In addition, the code to carry out the final step (establishing a connection) is a little different for the server side, as shown here:

```
SSLServerSocketFactory sf = sslContext.getServerSocketFactory();  
SSLServerSocket ss = (SSLServerSocket)sf.createServerSocket( port );  
ss.setNeedClientAuth( true );
```

Note that we called `SSLServerSocket.setNeedClientAuth()`. This is the server call indicating that the client should authenticate itself. Client applications do not authenticate themselves by default, so you must make this call if you want client authentication to be part of the handshaking process.

Section 6. Wrapup

Tying it all together

In this tutorial we've focused on the most complex aspect of using JSSE: properly configuring and installing the client and server key pairs. We talked briefly about the new Java Cryptography Architecture, but mostly focused on the use of key pairs for secure encryption, as implemented by JSSE. We used a very simple whiteboard example to make the discussion in the tutorial more concrete, going over each step necessary to generate key pairs, properly install them, and initiate a secure, socket-based client/server connection.

You can find the complete code source for the whiteboard example in [Resources](#). Build on what you've learned here by studying that code source and paying close attention to the steps outlined in the previous section. You may also want to practice what you've learned, and expand upon it, with the suggested exercise in the next

section. Whatever you do, be sure to use what you've learned here when it comes time to develop your next secure socket-based application.

Further exercises: Using a CA

In our whiteboard example, we've assumed that you have control over both the client-side and server-side installations. In other configurations, this isn't the case. For example, Web servers can be used with just about any Web browser, and the people who install the Web server software have nothing to do with those who install browser software. For this reason, Web servers and other public-protocol servers use third-party certification authorities to provide key certificates.

Instruction for using these systems is beyond the scope of this tutorial, but each one generally provides its own detailed instructions. In many cases, you can request a free certificate which can be used for testing purposes -- this lets you "try before you buy." See [Resources](#) for links to third-party certification authorities.

Resources

Learn

- Read the installation instructions for [JSSE 1.0.2](#), an extension to the Java 2 platform.
- The IEEE maintains a page documenting [specifications for public-key cryptography](#).
- Verisign, an IBM affiliate, is a major third-party provider of [SSL certificates](#).
- Another good resource for learning about socket programming and security is [Java Network Programming](#), discusses the new joint IBM-Microsoft proposal for integrated, standardized security on Web services architectures; the proposal includes a provision for the use of public-key technology.
- The IBM [Zurich Research Lab](#) has a long history in the study and implementation of security systems, especially in the area of cryptography.
- [The Java technology zone](#): Hundreds of articles about every aspect of Java programming, including all the concepts covered in this tutorial.

Get products and technologies

- Download [jsse-source.zip](#), the code source for the examples and scripts used in this tutorial.
- Download the [JDK 1.4](#) from Sun Microsystems.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Greg Travis

Greg Travis is a freelance programmer living in New York City. His interest in computers can probably be traced back to that episode of *The Bionic Woman* where Jamie runs around trying to escape a building whose lights and doors are controlled by an evil artificial intelligence that mocks her through loudspeakers. Greg is a devout believer in the idea that, when a computer program works, it's a complete coincidence.