SSL Handshake Failures

**1. Overview**

Secured Socket Layer (SSL) is a cryptographic protocol which provides security in communication over the network. **In this tutorial, we'll discuss various scenarios that can result in an SSL handshake failure and how to it.**

Note that our [Introduction to SSL using JSSE](https://www.baeldung.com/java-ssl) covers the basics of SSL in more detail.

**2. Terminology**

It's important to note that, due to security vulnerabilities, SSL as a standard is superseded by Transport Layer Security (TLS). Most programming languages, including Java, have libraries to support both SSL and TLS.

Since the inception of SSL, many products and languages like OpenSSL and Java had references to SSL which they kept even after TLS took over. For this reason, in the remainder of this tutorial, we will use the term SSL to refer generally to cryptographic protocols.

**3. Setup**

For the purpose of this tutorial, we'll create a simple server and client applications using [the Java Socket API](https://www.baeldung.com/a-guide-to-java-sockets) to simulate a network connection.

**3.1. Creating a Client and a Server**

In Java, we can use **sockets to establish a communication channel between a server and client over the network**. Sockets are a part of the Java Secure Socket Extension (JSSE) in Java.

Let's begin by defining a simple server:

|  |
| --- |
| int port = 8443;  ServerSocketFactory factory = SSLServerSocketFactory.getDefault();  try (ServerSocket listener = factory.createServerSocket(port)) {      SSLServerSocket sslListener = (SSLServerSocket) listener;      sslListener.setNeedClientAuth(true);      sslListener.setEnabledCipherSuites(        new String[] { "TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256" });      sslListener.setEnabledProtocols(        new String[] { "TLSv1.2" });      while (true) {          try (Socket socket = sslListener.accept()) {              PrintWriter out = new PrintWriter(socket.getOutputStream(), true);              out.println("Hello World!");          }      }  } |

The server defined above returns the message “Hello World!” to a connected client.

Next, let's define a basic client, which we'll connect to our *SimpleServer:*

|  |
| --- |
| String host = "localhost";  int port = 8443;  SocketFactory factory = SSLSocketFactory.getDefault();  try (Socket connection = factory.createSocket(host, port)) {      ((SSLSocket) connection).setEnabledCipherSuites(        new String[] { "TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA256" });      ((SSLSocket) connection).setEnabledProtocols(        new String[] { "TLSv1.2" });        SSLParameters sslParams = new SSLParameters();      sslParams.setEndpointIdentificationAlgorithm("HTTPS");      ((SSLSocket) connection).setSSLParameters(sslParams);        BufferedReader input = new BufferedReader(        new InputStreamReader(connection.getInputStream()));      return input.readLine();  } |

Our client prints the message returned by the server.

**3.2. Creating Certificates in Java**

SSL provides secrecy, integrity, and authenticity in network communications. Certificates play an important role as far as establishing authenticity.

Typically, these certificates are purchased and signed by a Certificate Authority, but for this tutorial, we'll use self-signed certificates.

**To achieve this, we can use *keytool,*which ships with the JDK:**

|  |
| --- |
| $ keytool -genkey -keypass password \                    -storepass password \                    -keystore serverkeystore.jks |

The above command starts an interactive shell to gather information for the certificate like Common Name (CN) and Distinguished Name (DN). When we provide all relevant details, it generates the file *serverkeystore.jks*, which contains the private key of the server and its public certificate.

Note that *serverkeystore.jks*is stored in the Java Key Store (JKS) format, which is proprietary to Java.**These days, *keytool*will remind us that we ought to consider using PKCS#12, which it also supports.**

We can further use *keytool*to extract the public certificate from the generated keystore file:

|  |
| --- |
| $ keytool -export -storepass password \                    -file server.cer \                    -keystore serverkeystore.jks |
|  |

The above command exports the public certificate from keystore as a file *server.cer*. Let's use the exported certificate for the client by adding it to its truststore:

|  |
| --- |
| $ keytool -import -v -trustcacerts \                       -file server.cer \                       -keypass password \                       -storepass password \                       -keystore clienttruststore.jks |

We have now generated a keystore for the server and corresponding truststore for the client. We will go over the use of these generated files when we discuss possible handshake failures.

And more details around the usage of Java's keystore can be found in our [previous tutorial](https://www.baeldung.com/java-keystore).

**4. SSL Handshake**

SSL handshakes are **a mechanism by which a client and server establish the trust and logistics required to secure their connection over the network**.

This is a very orchestrated procedure and understanding the details of this can help understand why it often fails, which we intend to cover in the next section.

Typical steps in an SSL handshake are:

1. Client provides a list of possible SSL version and cipher suites to use
2. Server agrees on a particular SSL version and cipher suite, responding back with its certificate
3. Client extracts the public key from the certificate responds back with an encrypted “pre-master key”
4. Server decrypts the “pre-master key” using its private key
5. Client and server compute a “shared secret” using the exchanged “pre-master key”
6. Client and server exchange messages confirming the successful encryption and decryption using the “shared secret”

While most of the steps are the same for any SSL handshake, there is a subtle difference between one-way and two-way SSL. Let's quickly review these differences.

**4.1. The Handshake in One-way SSL**

If we refer to the steps mentioned above, step two mentions the certificate exchange. One-way SSL requires that a client can trust the server through its public certificate. This **leaves the server to trust all clients** that request a connection. There is no way for a server to request and validate the public certificate from clients which can pose a security risk.

**4.2. The Handshake in Two-way SSL**

With one-way SSL, the server must trust all clients. But, two-way SSL adds the ability for the server to be able to establish trusted clients as well. During a two-way handshake, **both the client and server must present and accept each other's public certificates** before a successful connection can be established.

**5. Handshake Failure Scenarios**

**Having done that quick review, we can look at failure scenarios with greater clarity.**

An SSL handshake, in one-way or two-way communication, can fail for multiple reasons. We will go through each of these reasons, simulate the failure and understand how can we avoid such scenarios.

In each of these scenarios, we will use the *SimpleClient* and *SimpleServer* we created earlier.

**5.1. Missing Server Certificate**

Let's try to run the *SimpleServer* and connect it through the *SimpleClient*. While we expect to see the message “Hello World!”, we are presented with an exception:

|  |
| --- |
| Exception in thread "main" javax.net.ssl.SSLHandshakeException:    Received fatal alert: handshake\_failure |

Now, this indicates something went wrong. The *SSLHandshakeException* above, in an abstract manner, **is stating that the client when connecting to the server did not receive any certificate.**

To address this issue, we will use the keystore we generated earlier by passing them as system properties to the server:

|  |
| --- |
| -Djavax.net.ssl.keyStore=clientkeystore.jks -Djavax.net.ssl.keyStorePassword=password |

It's important to note that the system property for the keystore file path should either be an absolute path or the keystore file should be placed in the same directory from where the Java command is invoked to start the server. **Java system property for keystore does not support relative paths.**

Does this help us get the output we are expecting? Let's find out in the next sub-section.

**5.2. Untrusted Server Certificate**

As we run the *SimpleServer* and the *SimpleClient* again with the changes in the previous sub-section, what do we get as output:

|  |
| --- |
| Exception in thread "main" javax.net.ssl.SSLHandshakeException:    sun.security.validator.ValidatorException:    PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException:    unable to find valid certification path to requested target |

Well, it did not work exactly as we expected, but looks like it has failed for a different reason.

This particular failure is caused by the fact that our server is using a *self-signed* certificate which is not signed by a Certificate Authority (CA).

**Really, any time the certificate is signed by something other than what is in the default truststore, we'll see this error.** The default truststore in JDK typically ships with information about common CAs in use.

To solve this issue here, we will have to force *SimpleClient* to trust the certificate presented by *SimpleServer*. Let's use the truststore we generated earlier by passing them as system properties to the client:

|  |
| --- |
| -Djavax.net.ssl.trustStore=clienttruststore.jks -Djavax.net.ssl.trustStorePassword=password |

Please note that this is not an ideal solution.**In an ideal scenario, we should not use a self-signed certificate but a certificate which has been certified by a Certificate Authority (CA) which clients can trust by default.**

Let's go to the next sub-section to find out if we get our expected output now.

**5.3. Missing Client Certificate**

Let's try one more time running the SimpleServer and the SimpleClient, having applied the changes from previous sub-sections:

|  |
| --- |
| Exception in thread "main" java.net.SocketException:    Software caused connection abort: recv failed |

Again, not something we expected. The *SocketException*here tells us that the server could not trust the client. This is because we have set up a two-way SSL. In our *SimpleServer*we have:

|  |
| --- |
| ((SSLServerSocket) listener).setNeedClientAuth(true); |

**The above code indicates an *SSLServerSocket*is required for client authentication through their public certificate.**

We can create a keystore for the client and a corresponding truststore for the server in a way similar to the one that we used when creating the previous keystore and truststore.

We will restart the server and pass it the following system properties:

|  |
| --- |
| -Djavax.net.ssl.keyStore=serverkeystore.jks \      -Djavax.net.ssl.keyStorePassword=password \      -Djavax.net.ssl.trustStore=servertruststore.jks \      -Djavax.net.ssl.trustStorePassword=password |

Then, we will restart the client by passing these system properties:

|  |
| --- |
| -Djavax.net.ssl.keyStore=clientkeystore.jks \      -Djavax.net.ssl.keyStorePassword=password \      -Djavax.net.ssl.trustStore=clienttruststore.jks \      -Djavax.net.ssl.trustStorePassword=password |

Finally, we have the output we desired:

|  |
| --- |
| Hello World! |

**5.4. Incorrect Certificates**

Apart from the above errors, a handshake can fail due to a variety of reasons related to how we have created the certificates. One common error is related to an incorrect CN. Let's explore the details of the server keystore we created previously:

|  |
| --- |
| keytool -v -list -keystore serverkeystore.jks |

When we run the above command, we can see the details of the keystore, specifically the owner:

|  |
| --- |
| ...  Owner: CN=localhost, OU=technology, O=baeldung, L=city, ST=state, C=xx  ... |

The CN of the owner of this certificate is set to localhost. The CN of the owner must exactly match the host of the server. If there is any mismatch it will result in an *SSLHandshakeException*.

Let's try to regenerate the server certificate with CN as anything other than localhost. When we use the regenerated certificate now to run the *SimpleServer* and *SimpleClient* it promptly fails:

|  |
| --- |
| Exception in thread "main" javax.net.ssl.SSLHandshakeException:      java.security.cert.CertificateException:      No name matching localhost found |

The exception trace above clearly indicates that the client was expecting a certificate bearing the name as localhost which it did not find.

Please note that **JSSE does not mandate hostname verification by default.** We have enabled hostname verification in the *SimpleClient* through explicit use of HTTPS:

|  |  |
| --- | --- |
| 1  2  3 | SSLParameters sslParams = new SSLParameters();  sslParams.setEndpointIdentificationAlgorithm("HTTPS");  ((SSLSocket) connection).setSSLParameters(sslParams); |

Hostname verification is a common cause of failure and in general and should always be enforced for better security. For details on hostname verification and its importance in security with TLS, please refer to [this article](https://tersesystems.com/blog/2014/03/23/fixing-hostname-verification/).

**5.5. Incompatible SSL Version**

**Currently, there are various cryptographic protocols including different versions of SSL and TLS in operation.**

As mentioned earlier, SSL, in general, has been superseded by TLS for its cryptographic strength. The cryptographic protocol and version are an additional element that a client and a server must agree on during a handshake.

For example, if the server uses a cryptographic protocol of SSL3 and the client uses TLS1.3 they cannot agree on a cryptographic protocol and an *SSLHandshakeException* will be generated.

In our *SimpleClient* let's change the protocol to something that is not compatible with the protocol set for the server:

|  |
| --- |
| ((SSLSocket) connection).setEnabledProtocols(new String[] { "TLSv1.1" }); |

When we run our client again, we will get an *SSLHandshakeException*:

|  |
| --- |
| Exception in thread "main" javax.net.ssl.SSLHandshakeException:    No appropriate protocol (protocol is disabled or cipher suites are inappropriate) |

The exception trace in such cases is abstract and does not tell us the exact problem. **To resolve these types of problems it is necessary to verify that both the client and server are using either the same or compatible cryptographic protocols.**

**5.6. Incompatible Cipher Suite**

The client and server must also agree on the cipher suite they will use to encrypt messages.

During a handshake, the client will present a list of possible ciphers to use and the server will respond with a selected cipher from the list. The server will generate an *SSLHandshakeException*if it cannot select a suitable cipher.

In our *SimpleClient* let's change the cipher suite to something that is not compatible with the cipher suite used by our server:

|  |
| --- |
| ((SSLSocket) connection).setEnabledCipherSuites(    new String[] { "TLS\_RSA\_WITH\_AES\_128\_GCM\_SHA256" }); |

When we restart our client we will get an *SSLHandshakeException*:

|  |
| --- |
| Exception in thread "main" javax.net.ssl.SSLHandshakeException:    Received fatal alert: handshake\_failure |

Again, the exception trace is quite abstract and does not tell us the exact problem. The resolution to such an error is to verify the enabled cipher suites used by both the client and server and ensure that there is at least one common suite available.

Normally, clients and servers are configured to use a wide variety of cipher suites so this error is less likely to happen. **If we encounter this error it is typically because the server has been configured to use a very selective cipher.** A server may choose to enforce a selective set of ciphers for security reasons.

**6. Conclusion**

In this tutorial, we learned about setting up SSL using Java sockets. Then we discussed SSL handshakes with one-way and two-way SSL. Finally, we went through a list of possible reasons that SSL handshakes may fail and discussed the solutions.

As always, the code for the examples is available [over on GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-security).