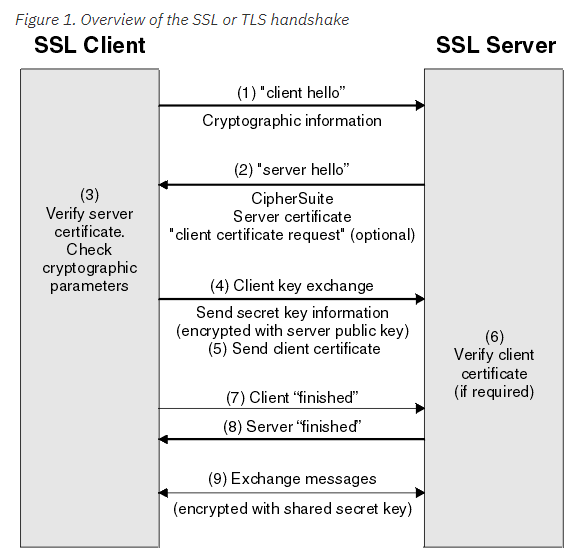
SSL Handshake:



* Client initiates handshake by sending “ClientHello” message which includes
  + Compatible SSL/TLS version.
  + List of [cipher suits](#CipherSuittes) supported
  + The message also contains a random byte string that is used in subsequent computations.
* The SSL/TLS server responds with a server hello message that contains
  + The CipherSuite chosen by the server.
  + Server's digital-certificate.
  + The session ID, and another random byte string.
  + In case of two way authentication (Optional), the server sends a client certificate request that includes
    - A list of the types of certificates supported and
    - Acceptable Certification Authorities (CAs).
* Client: verifies server's certificate. How?
  + Verifies server's private key. how? Generates and encrypt a pre-master (shared secret key) key using server's public key.
  + In two way SSL (optional), the client sends a random byte string encrypted with the client's private key, together with the client's digital certificate.
  + or a no digital certificate alert if client doesn't have it's certificate. The handshake fails if client authentication is mandatory.
* Server : Decrypt pre-master using it's private key.
* Client: sends sample message to verify that master-key works.
* Server : Sends back decrypted version of this message.

With this SSL handshake process is completed.

SSL Log analysis

**Happy Flow** during handshake when we use client certificates:

**Client sends a ClientHello message**

* Using the TLS protocol version he supports,
* A random number and
* A list of suggested cipher suites and compression methods.

\*\*\*

ClientHello, TLSv1

RandomCookie: GMT: 1331663143 bytes = { 141, 219, 18, 140, 148, 60, 33, 241, 10, 21, 31, 90, 88, 145, 34, 153, 238, 105, 148, 72, 163, 210, 233, 49, 99, 224, 226, 64 }

Session ID: {}

Cipher Suites: [SSL\_RSA\_WITH\_RC4\_128\_MD5, SSL\_RSA\_WITH\_RC4\_128\_SHA, TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA, TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA, TLS\_DHE\_RSA\_WITH\_AES\_128\_CBC\_SHA, TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA, TLS\_DHE\_DSS\_WITH\_AES\_128\_CBC\_SHA, TLS\_DHE\_DSS\_WITH\_AES\_256\_CBC\_SHA, SSL\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA, SSL\_DHE\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA, SSL\_DHE\_DSS\_WITH\_3DES\_EDE\_CBC\_SHA, SSL\_RSA\_WITH\_DES\_CBC\_SHA, SSL\_DHE\_RSA\_WITH\_DES\_CBC\_SHA, SSL\_DHE\_DSS\_WITH\_DES\_CBC\_SHA, SSL\_RSA\_EXPORT\_WITH\_RC4\_40\_MD5, SSL\_RSA\_EXPORT\_WITH\_DES40\_CBC\_SHA, SSL\_DHE\_RSA\_EXPORT\_WITH\_DES40\_CBC\_SHA, SSL\_DHE\_DSS\_EXPORT\_WITH\_DES40\_CBC\_SHA, TLS\_EMPTY\_RENEGOTIATION\_INFO\_SCSV]

Compression Methods: { 0 }

\*\*\*

**The server responds, with a ServerHello message**, that contains

* the choices made based on the information provided by the client
* another random number and
* a session id(optionally)

\*\*\*

**ServerHello, TLSv1**

RandomCookie: GMT: 1331663143 bytes = { 172, 233, 79, 197, 14, 21, 187, 161, 114, 206, 7, 38, 188, 228, 120, 102, 115, 214, 155, 86, 211, 41, 156, 179, 138, 2, 230, 81 }

Session ID: {79, 96, 145, 39, 203, 136, 206, 69, 170, 46, 194, 17, 154, 175, 13, 138, 143, 199, 162, 193, 110, 86, 113, 109, 248, 187, 220, 169, 47, 180, 44, 68}

Cipher Suite: SSL\_RSA\_WITH\_RC4\_128\_MD5

Compression Method: 0

Extension renegotiation\_info, renegotiated\_connection: <empty>

\*\*\*

So in this case we're going to use SSL\_RSA\_WITH\_RC4\_128\_MD5 as Cipher Suite.

The next step is also done by the server. **Server sends a Certificate message that contains its complete certificate chai**n:

\*\*\*\*\*\*

**Certificate chain**

chain [0] = [

[

Version: V1

Subject: CN=server, C=NL

Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

Key: Sun RSA public key, 1024 bits

modulus: 143864428144045085986129639694300995179398936575198896494655652087658861594939489453166811774109137006267822033915476680673848164790815913192075840268069822357600376998775923266017630332239546722181180383155088413406178660120548292599278819762883993031950564327152510982887716901499177102158407884939613382007

public exponent: 65537

Validity: [From: Wed Mar 14 13:32:04 CET 2012,

To: Thu Mar 14 13:32:04 CET 2013]

Issuer: CN=Application CA, OU=GKD, O=Smartjava, L=Maasland, ST=ZH, C=NL

SerialNumber: [ a881d144 5e631f21]

]

Algorithm: [SHA1withRSA]

Signature:

0000: C3 56 81 7F 33 91 8A FF 84 5E 0B BA 7A 01 D8 41 .V..3....^..z..A

0010: 6B 47 B2 F7 8F FB B5 77 23 D8 FB B2 35 19 6E C4 kG.....w#...5.n.

0020: A4 6A BC 23 BB 69 92 F6 85 5A 1E CB FE 23 C6 98 .j.#.i...Z...#..

0030: A0 57 F8 FB E9 DB B0 40 BD 8E F8 35 F8 77 E1 09 .W.....@...5.w..

0040: 5A 2E 45 71 80 F6 89 E7 0B 93 E2 48 EB 40 92 13 Z.Eq.......H.@..

0050: 14 AA 1F 59 AA 98 67 46 9B 52 33 49 9A 3C 91 9B ...Y..gF.R3I.<..

0060: F1 CB 8A BD 7D D4 DD 76 C4 15 00 36 A3 B2 87 A7 .......v...6....

0070: D5 FF 52 E3 68 D4 F0 E0 32 86 74 02 DD 92 EC 1D ..R.h...2.t.....

]

chain [1] = [

[

\*\*\*\*\*\*\*\*\*\*\*

In this message you can see that the issuer of this certificate is our example CA.

Our client checks to see if this certificate is trusted, which it is in this case. Since we require the client to authenticate itself the server requests a certificate from the **client and after that sends a helloDone.**

\*\*\*

CertificateRequest

Cert Types: RSA, DSS

Cert Authorities:

<CN=Application CA, OU=CA, O=Blaat, L=Waalwijk, ST=ZH, C=NL>

ServerHelloDone

\*\*\*\*\*

Here server provides a list of Cert Authorities it trusts. The client will use this information to determine if it has a keypair that matches this CA.

Client sends:

\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Certificate chain

chain [0] = [

[

Version: V1

Subject: CN=Application 3, OU=Smartjava, O=Smartjava, L=NL, ST=ZH, C=NL

Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

Key: Sun RSA public key, 1024 bits

modulus: 90655907749318585147523875906892969031300830816947226352221659107570169820452561428696751943383590982109524990627182456571533992582229229163232831159652561902456847954385746762477844009336466314872376131553489447601649924116778337873632641536164462534398137791450495316700015095054427027256393580022887087767

public exponent: 65537

Validity: [From: Mon Mar 12 15:13:24 CET 2012,

To: Tue Mar 12 15:13:24 CET 2013]

Issuer: CN=Application CA, OU=Smartjava, O=Smartjava, L=Maasland, ST=ZH, C=NL

SerialNumber: [ b247ffb2 ce060768]

]

Algorithm: [SHA1withRSA]

Signature:

0000: 97 58 36 C5 28 87 B3 16 9B DD 31 0C E0 C6 23 76 .X6.(.....1...#v

0010: 72 82 5B 13 4D 23 B6 0E A9 2F 9F 0C 3F 97 15 6E r.[.M#.../..?..n

0020: 7B 38 EC DE E2 57 D7 AA 07 12 E3 98 B7 86 A7 CE .8...W..........

0030: 57 8E A1 29 96 C9 F0 30 57 67 C7 F1 F2 98 90 64 W..)...0Wg.....d

0040: 6C B9 6C 05 24 8B 56 3F B1 FF 03 62 3D 81 DB 45 l.l.$.V?...b=..E

0050: D3 1F C1 B2 DD 77 CF 74 54 EB 9D 82 23 89 1A 70 .....w.tT...#..p

0060: F8 C4 68 6A B7 41 C7 DE 7B B6 3A 0C 17 E7 FA 98 ..hj.A....:.....

0070: 19 0C D8 91 FB 5E FE D2 B3 92 FD 2D 2A 6B 51 10 .....^.....-\*kQ.

]

chain [1] = [

[

Version: V3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This certificate is checked on the server side and if all is well, the final steps in the handshake are executed to setup the secured connection.

Note that there is a CertificateVerify step. In this step the client signs a message with its private key.

This is done so the server can verify the client has access to its private key. This might seem a step where things can go wrong in an incorrectly configured environment.

In the default java implementation this won't happen.

In the phase where the client has to determine which certificate to present to the server, the java implementation already checks if the privatekey is available.

So what could possibly go wrong in this handshake?

**Passwords**

Exception in thread "main" java.security.UnrecoverableKeyException: Cannot recover key

at sun.security.provider.KeyProtector.recover(KeyProtector.java:311)

at sun.security.provider.JavaKeyStore.engineGetKey(JavaKeyStore.java:121)

This very helpful message is thrown when (from the javadoc) " .. a key in the keystore cannot be recovered".

There are a couple of reasons this can happen, but normally this occurs when the **key in the keystore** is accessed with the **wrong password**.

When you use the keytool to create and manage your keys, the keystore password is usually the same as the key password.

However, if you import keys from a PKCS#12 type keystore, the password of the keystore can be easily set to a different value.

Not all the SSL client allow you to specify a different password for the key and the keystore. If that is the case you can use the following command, to change the password of the key:

keytool -keypasswd -alias <keyalias> -keystore <keystore>

It is also possible to set an incorrect password for the keystore. Luckily in that case the error message that is thrown is much more helpful:

Exception in thread "main" java.io.IOException: Keystore was tampered with, or password was incorrect

at sun.security.provider.JavaKeyStore.engineLoad(JavaKeyStore.java:771)

at sun.security.provider.JavaKeyStore$JKS.engineLoad(JavaKeyStore.java:38)

at java.security.KeyStore.load(KeyStore.java:1185)

...

Caused by: java.security.UnrecoverableKeyException: Password verification failed

at sun.security.provider.JavaKeyStore.engineLoad(JavaKeyStore.java:769)

... 3 more

If this occurs at the server side, we can see the same message when the SSL listener is being set up.

**Incomplete CA Chains**: We see "peer not authenticated" exceptions.

javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated

at com.sun.net.ssl.internal.ssl.SSLSessionImpl.getPeerCertificates(SSLSessionImpl.java:352)

at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:128)

at org.apache.http.conn.ssl.SSLSocketFactory.connectSocket(SSLSocketFactory.java:397)

We'll start by looking from the client side. If we look through the logging we find the following CertificateRequest message from the server and the ServerHelloDone.

\*\*\*

CertificateRequest

Cert Types: RSA, DSS

Cert Authorities:

ServerHelloDone

\*\*\*

The server has already sent its certificate, and since our client doesn't throw an error on that part, we can assume it is trusted by the client.

So something seems to be wrong with the steps that come after this message from the server.

You can see that the server doesn't specify a set of Cert Authorities it trusts.

* This could be a misconfiguration at the server side, or
* It could just be that the server expects one of the trusted Root CAs.

In any case, the client is free to send any certificate he wants. So the client sends the following certificate:

\*\*\*

Certificate chain

chain [0] = [

[

Version: V1

Subject: CN=Application4, OU=Smartjava, O=Smartjava, L=NL, ST=NB, C=NL

Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

...

\*\*\*\*\*\*\*\*\*

According to the specification the client now continues with the key exchange and generates secrets to exchange. Somewhere along the lines we can see the following:

\*\*\*\*\*\*\*\*

pool-1-thread-1, WRITE: TLSv1 Handshake, length = 32

pool-1-thread-1, READ: TLSv1 Alert, length = 2

pool-1-thread-1, RECV TLSv1 ALERT: fatal, internal\_error

pool-1-thread-1, called closeSocket()

\*\*\*\*\*\*\*\*\*\*

This means we've received an internal error. So something at the server side went wrong. Looking at the server we see the following in the SSL dump:

\*\*\*

Certificate chain

chain [0] = [

[

Version: V1

Subject: CN=Application4, OU=Smartjava, O=Smartjava, L=NL, ST=NB, C=NL

Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

...

]

chain [1] = [

[

...

]

qtp1735121130-17, handling exception: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty

qtp1735121130-17, SEND TLSv1 ALERT: fatal, description = internal\_error

qtp1735121130-17, WRITE: TLSv1 Alert, length = 2

\*\*\*\*\*

You can see that we received the certificate from the client, and directly after that we get this error.

This error however doesn't really tell us anything. We do however have enough information to at least limit the possible errors.

We know that the server didn't send a list of CAs, we can see that the client sent a valid certificate, and that server somehow isn't able to process it.

It looks like a problem with the server truststore.

In this case the best approach is to look at the certificates the server trusts. Either in the cacerts file or in it's own truststore.

Validate whether the CA certificate our client sends is in the server's truststore, and the server actually loads the stores we expect.

It's also possible that the client has an incomplete chain of trust for the certificate received from the server.

In that case we once again get the "peer not authenticated" error at the client side. If we look at the SSL debug logging, we see the following exception occuring at the client side:

\*\*\*\*

pool-1-thread-1, handling exception: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty

pool-1-thread-1, SEND TLSv1 ALERT: fatal, description = internal\_error

pool-1-thread-1, WRITE: TLSv1 Alert, length = 2

\*\*\*\*\*

This exception occurred directly after the server has sent its certificate using a "Certificate message":

\*\*\*\*\*\*\*\*

Certificate chain

chain [0] = [

[

Version: V1

Subject: CN=server, C=NL

Signature Algorithm: SHA1withRSA, OID = 1.2.840.113549.1.1.5

\*\*\*\*\*

Following the same reasoning as for the server we can conclude that there is something wrong with the client side truststore. For completeness sake, the server receives this error message when this situation occurs at the client:

qtp1500389297-17, READ: TLSv1 Alert, length = 2

qtp1500389297-17, RECV TLSv1 ALERT: fatal, internal\_error

qtp1500389297-17, called closeSocket()

qtp1500389297-17, handling exception: javax.net.ssl.SSLException: Received fatal alert: internal\_error

qtp1500389297-17, called close()

qtp1500389297-17, called closeInternal(true)

**Invalid keys :**

Look at the following error that occurs during this handshake. In the logging at the client side we see the following error message in the SSL output:

\*\*\*\*

ool-1-thread-1, WRITE: TLSv1 Handshake, length = 32

pool-1-thread-1, READ: TLSv1 Alert, length = 2

pool-1-thread-1, RECV TLSv1 ALERT: fatal, internal\_error

pool-1-thread-1, called closeSocket()

pool-1-thread-1, handling exception: javax.net.ssl.SSLException: Received fatal alert: internal\_error

pool-1-thread-1, IOException in getSession(): javax.net.ssl.SSLException: Received fatal alert: internal\_error

\*\*\*\*

Which results in the very unhelpful:

\*\*\*\*

javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated

at com.sun.net.ssl.internal.ssl.SSLSessionImpl.getPeerCertificates(SSLSessionImpl.java:352)

at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:128)

at org.apache.http.conn.ssl.SSLSocketFactory.connectSocket(SSLSocketFactory.java:397)

\*\*\*\*

When you receive an internal error, there is usually something wrong at the server side. So looking at the serverside, let’s see what caused this error.

\*\*\*

qtp2044601711-16, handling exception: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty

qtp2044601711-16, SEND TLSv1 ALERT: fatal, description = internal\_error

\*\*\*\*\*

Somewhat useful. It seems that there is something wrong with the algorithm we used, the client seems to have provided an incorrect certificate.

But what is wrong? If you look back at the happy flow, you can send that at a certain time the server asks the client for a certificate using a "Certificate" message. Lets look a bit closer at this message and the response:

\*\*\*\*\* CertificateRequest

Cert Types: RSA, DSS

Cert Authorities:

<EMAILADDRESS=jos.dirksen@gmail.com, CN=CA2, OU=Smartjava, O=Smartjava, L=Waalwijk, ST=NB, C=NL>

\*\*\* ServerHelloDone

matching alias: application4

\*\*\* Certificate chain

chain [0] = [

[

Version: V1

Subject: CN=Application4, OU=Smartjava, O=Smartjava, L=NL, ST=NB, C=NL

Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

Key: Sun DSA Public Key

...

\*\*\*\*\*\*\*\*

What you can see here is that the server specifies the cert types it accepts, and the authorities it accepts.

The client responses in this case however with a DSA public key.

Depending on the server implementation this can cause this strange message.

Another possible scenario I've seen (especially with self-signed certificates) is that with a "CertificateRequest" message like this:

\*\*\* CertificateRequest

Cert Types: RSA, DSS

Cert Authorities:

\*\*\* ServerHelloDone

This client won't respond with a certificate at all, if you only have DSA based keys in your keystore. It won't throw an error on the client side, but will cause a "null certificate chain" message as the server side. I haven't seen this scenario, though, when you don't use self-signed certificates.

**Certificate expiration** : In this case we see the very cryptic message at the client side:

If we look at the phase of the SSL handshake we're in, we can see that we've already sent our client certificate and finishing up the handshake when we receive this error. The error on the serverside is actually pretty helpful. After receiving the invalid certificate, in the debug logging, it shows us the following:

\*\*\*

qtp1735121130-17, SEND TLSv1 ALERT: fatal, description = certificate\_unknown

qtp1735121130-17, WRITE: TLSv1 Alert, length = 2

[Raw write]: length = 7

0000: 15 03 01 00 02 02 2E .......

qtp1735121130-17, called closeSocket()

qtp1735121130-17, handling exception: javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path validation failed: java.security.cert.CertPathValidatorException: timestamp check failed

qtp1735121130-17, called close()

qtp1735121130-17, called closeInternal(true)

\*\*\*

It tells us that during the validation of the certificate, a timestamp check failed. This tells us that we should look at the validity of the certificates in our certificate chain to see what is happening.

**Server is http**

Unrecognized SSL message, plaintext connection?

In this case we don’t see server hello message. And could see **HTTP/ just before exception.**

main, WRITE: TLSv1.2 Handshake, length = 161

[Raw write]: length = 166

0000: 16 03 03 00 A1 01 00 00 9D 03 03 5D 02 3B 7F 67 ...........].;.g

0010: 36 A2 A0 92 87 B9 06 85 CE 99 B6 CC 67 6B 25 FC 6...........gk%.

0020: C1 36 45 6E 99 98 9F 22 93 FD B1 00 00 3A C0 23 .6En...".....:.#

0030: C0 27 00 3C C0 25 C0 29 00 67 00 40 C0 09 C0 13 .'.<.%.).g.@....

0040: 00 2F C0 04 C0 0E 00 33 00 32 C0 2B C0 2F 00 9C ./.....3.2.+./..

0050: C0 2D C0 31 00 9E 00 A2 C0 08 C0 12 00 0A C0 03 .-.1............

0060: C0 0D 00 16 00 13 00 FF 01 00 00 3A 00 0A 00 16 ...........:....

0070: 00 14 00 17 00 18 00 19 00 09 00 0A 00 0B 00 0C ................

0080: 00 0D 00 0E 00 16 00 0B 00 02 01 00 00 0D 00 16 ................

0090: 00 14 06 03 06 01 05 03 05 01 04 03 04 01 04 02 ................

00A0: 02 03 02 01 02 02 ......

[Raw read]: length = 5

0000: 48 54 54 50 2F **HTTP/**

main, handling exception: javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connection?

main, SEND TLSv1.2 ALERT: fatal, description = unexpected\_message

main, WRITE: TLSv1.2 Alert, length = 2

[Raw write]: length = 7

0000: 15 03 03 00 02 02 0A .......

main, called closeSocket()

main, IOException in getSession(): javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connection?

Exception in thread "main" javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated

at sun.security.ssl.SSLSessionImpl.getPeerCertificates(SSLSessionImpl.java:431)

at SSL.Security.SSLClientSessionMain.main(SSLClientSessionMain.java:32)

**Cipher Suites**

<https://en.wikipedia.org/wiki/Cipher_suite>

A cipher suite is a set of algorithms that help secure a network connection that uses Transport Layer Security (TLS) or its now-deprecated predecessor Secure Socket Layer (SSL).

The set of algorithms that cipher suites (for ex TLS\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA) usually include:

* TLS defines the protocol that this cipher suite is for; it will usually be TLS.
* A key exchange algorithm (RSA): to exchange a key between two devices.
  + Key is used to encrypt and decrypt the messages being sent between two machines.
* A bulk encryption algorithm (3DES\_EDE\_CBC): algorithm used to encrypt the data being sent.
* A message authentication code - MAC0(SHA) algorithm: data integrity checks to ensure that the data sent does not change in transit.

In addition, cipher suites can include signatures and an authentication algorithm to help authenticate the server and or client.