# Les échanges SSL

Les messages SSL sont envoyés dans l'ordre suivant:

1. **Client: Client hello** : Le client envoie des informations au serveur, y compris la version la plus élevée de SSL qu'il prend en charge et une liste des suites de chiffrement qu'il prend en charge. (TLS 1.0 est indiqué comme SSL 3.1.) Les informations de la suite de chiffrement comprennent des algorithmes cryptographiques et des tailles de clés.
2. **Server: Server hello** : Le serveur choisit la version la plus élevée de SSL et la meilleure suite de chiffrement que le client et lui-même prennent en charge et envoie ces informations au client.
3. **Server: Certificate** : Le serveur envoie au client un certificat ou une chaîne de certificats. Une chaîne de certificats commence généralement par le certificat de clé publique du serveur et se termine par le certificat racine de l'autorité de certification. Ce message est facultatif, mais est utilisé chaque fois que l'authentification du serveur est requise.
4. **Server: Certificate request** : Si le serveur doit authentifier le client, il envoie au client une demande de certificat. Dans les applications Internet, ce message est rarement envoyé.
5. **Server: Server key exchange** : Le serveur envoie au client un message d'échange de clé de serveur lorsque les informations de clé publique envoyées dans le message 3 ci-dessus ne sont pas suffisantes pour l'échange de clé. Par exemple, dans les suites de chiffrement basées sur Diffie-Hellman, ce message contient la clé publique DH du serveur.
6. **Server: Server hello done** : Le serveur indique au client qu'il a terminé ses messages de négociation initiaux.
7. **Client: Certificate** : Si le serveur demande un certificat au client dans le message 4, le client envoie sa chaîne de certificats, tout comme le serveur l'a fait dans le message 3. Remarque: Seules quelques applications de serveur Internet demandent un certificat au client.
8. **Client: Client key exchange** : Le client génère des informations utilisées pour créer une clé à utiliser pour le chiffrement symétrique. Pour RSA, le client crypte ensuite ces informations de clé avec la clé publique du serveur et les envoie au serveur. Pour les suites de chiffrement basées sur Diffie-Hellman, ce message contient la clé publique DH du client.
9. **Client: Certificate verify** : Ce message est envoyé lorsqu'un client présente un certificat comme expliqué précédemment. Son objectif est de permettre au serveur de terminer le processus d'authentification du client. Lorsque ce message est utilisé, le client envoie des informations qu'il signe numériquement à l'aide d'une fonction de hachage cryptographique. Lorsque le serveur déchiffre ces informations avec la clé publique du client, le serveur est en mesure d'authentifier le client.
10. **Client: Change cipher spec** : Le client envoie un message indiquant au serveur de passer en mode crypté.
11. **Client: Finished** : Le client indique au serveur qu'il est prêt à commencer la communication sécurisée des données.
12. **Server: Change cipher spec** : Le serveur envoie un message indiquant au client de passer en mode crypté.
13. **Server: Finished** : Le serveur indique au client qu'il est prêt à commencer la communication sécurisée des données. C'est la fin de la négociation SSL.
14. **Client: Encrypted data** : Le client et le serveur communiquent en utilisant l'algorithme de cryptage symétrique et la fonction de hachage cryptographique négociés dans les messages 1 et 2, et en utilisant la clé secrète que le client a envoyée au serveur dans le message 8. La prise de contact peut être renégociée à ce moment. Voir la section suivante pour plus de détails.
15. **Client: Close Messages** : À la fin de la connexion, chaque côté enverra un message close\_notify pour informer l'homologue que la connexion est fermée.

# Un serveur SSL

Création d'un keystore server avec un certificat auto-signé:

keytool -genkey -keyalg RSA -keypass password -storepass password -keystore jks.server.keystore -alias serveur

Vérification du contenu du keystore server :

keytool -list -storepass password -v -keystore jks.server.keystore

*Keystore type: JKS*

*Keystore provider: SUN*

*Your keystore contains 1 entry*

*Alias name: serveur*

*Creation date: 12 févr. 2020*

*Entry type: PrivateKeyEntry*

*Certificate chain length: 1*

*Certificate[1]:*

*Owner: CN=localhost, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Issuer: CN=localhost, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Serial number: df87008*

*Valid from: Wed Feb 12 23:42:58 CET 2020 until: Sat Feb 06 23:42:58 CET 2021*

*Certificate fingerprints:*

*SHA1: ED:42:B5:06:53:73:01:54:70:BC:FA:B5:FD:76:EA:BC:EB:9D:9F:31*

*SHA256: 8F:71:C9:69:42:C1:52:55:CA:54:D0:EE:C2:02:6A:7B:3A:9D:EA:A6:08:8B:1E:E6:CD:66:CA:10:D2:65:C5:4B*

*Signature algorithm name: SHA256withRSA*

*Subject Public Key Algorithm: 2048-bit RSA key*

*Version: 3*

*Extensions:*

*#1: ObjectId: 2.5.29.14 Criticality=false*

*SubjectKeyIdentifier [*

*KeyIdentifier [*

*0000: 39 6C 66 CA 11 12 23 CB 57 60 FB F0 22 7B 37 C0 9lf...#.W`..".7.*

*0010: 68 68 13 24 hh.$*

*]*

*]*

Extraction du certificat du keystore server:

keytool -export -storepass password -keystore jks.server.keystore -file serveur.cer -alias serveur

Ajout du certificat server dans le truststore client:

keytool -import -v -trustcacerts -keypass password -storepass password -file serveur.cer -keystore jks.client.truststore -alias serveur

Vérification du truststore client jks.client.truststore :

keytool -list -storepass password -v -keystore jks.client.truststore

*Keystore type: JKS*

*Keystore provider: SUN*

*Your keystore contains 1 entry*

*Alias name: serveur*

*Creation date: 12 févr. 2020*

*Entry type: trustedCertEntry*

*Owner: CN=localhost, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Issuer: CN=localhost, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Serial number: df87008*

*Valid from: Wed Feb 12 23:42:58 CET 2020 until: Sat Feb 06 23:42:58 CET 2021*

*Certificate fingerprints:*

*SHA1: ED:42:B5:06:53:73:01:54:70:BC:FA:B5:FD:76:EA:BC:EB:9D:9F:31*

*SHA256: 8F:71:C9:69:42:C1:52:55:CA:54:D0:EE:C2:02:6A:7B:3A:9D:EA:A6:08:8B:1E:E6:CD:66:CA:10:D2:65:C5:4B*

*Signature algorithm name: SHA256withRSA*

*Subject Public Key Algorithm: 2048-bit RSA key*

*Version: 3*

*Extensions:*

*#1: ObjectId: 2.5.29.14 Criticality=false*

*SubjectKeyIdentifier [*

*KeyIdentifier [*

*0000: 39 6C 66 CA 11 12 23 CB 57 60 FB F0 22 7B 37 C0 9lf...#.W`..".7.*

*0010: 68 68 13 24 hh.$*

*]*

*]*

**package** ssl;

**import** java.io.File;

**import** java.io.IOException;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.security.NoSuchAlgorithmException;

**import** java.util.Scanner;

**import** javax.net.ServerSocketFactory;

**import** javax.net.ssl.SSLServerSocket;

**import** javax.net.ssl.SSLServerSocketFactory;

**public** **class** SimpleServer {

**public** **static** **final** **int** ***SERVER\_PORT*** = 8443;

**private** **static** **boolean** *IS\_DEBUG\_ENABLE* = **false**;

**public** **static** **boolean** *CLIENT\_AUTHENTICATION\_REQUIERED* = **false**;

**public** **static** **void** main(String[] args) **throws** IOException, NoSuchAlgorithmException {

*setSslProperties*(args);

*startServer*(***SERVER\_PORT***);

}

**private** **static** **void** setSslProperties(String[] args) {

**if** (args.length < 2) {

*usage*();

}

// clientAuthentication

**try** {

*CLIENT\_AUTHENTICATION\_REQUIERED* = *getBoolean*(args[0]);

System.***out***.println("clientAuthentication : "+ *CLIENT\_AUTHENTICATION\_REQUIERED*);

} **catch** (Exception e) {

System.***err***.println("Erreur sur clientAuthentication");

*usage*();

}

// knownClient

**boolean** knownClient = **false**;

**try** {

knownClient = *getBoolean*(args[1]);

System.***out***.println("knownClient : "+ knownClient);

} **catch** (Exception e) {

System.***err***.println("Erreur sur knownClient");

*usage*();

}

// certificatPath

String rootResources = **null**;

**if** (args.length > 2) {

File certificatsPath = **new** File(args[2]);

**if** (certificatsPath.exists()) {

rootResources = certificatsPath.getAbsolutePath();

} **else** {

System.***err***.println("Le repertoire "+ args[2] + " n'existe pas");

*usage*();

}

} **else** {

rootResources = **new** File(".").getAbsolutePath();

}

rootResources = rootResources.endsWith(File.***separator***) ? rootResources : rootResources+File.***separator***;

System.***out***.println("Current certificats path is : "+ rootResources);

// debugEnable

**if** (args.length > 3) {

**try** {

*IS\_DEBUG\_ENABLE* = *getBoolean*(args[3]);

} **catch** (Exception e) {

System.***err***.println("ERROR : The seconde argument, debugEnable, must be true or false.");

}

}

System.***out***.println("Debug enable is " + *IS\_DEBUG\_ENABLE*);

System.*setProperty*("javax.net.ssl.keyStore", rootResources + "jks.server.keystore");

System.*setProperty*("javax.net.ssl.keyStorePassword", "password");

**if** (knownClient) {

System.*setProperty*("javax.net.ssl.trustStore", rootResources + "jks.server.truststore");

System.*setProperty*("javax.net.ssl.trustStorePassword", "password");

}

**if** (*IS\_DEBUG\_ENABLE*) {

System.*setProperty*("javax.net.debug", "ssl:all");

System.*setProperty*("java.security.debug", "access:stack");

}

}

**private** **static** **void** usage() {

System.***err***.println("Usage : java SimpleServer.class clientAuthentication knownClient certificatPath debugEnable");

System.***err***.println(" clientAuthentication (true/false) : Handshake SSL bidirectionnel");

System.***err***.println(" knownClient (true/false) : Est-ce que le server connait le client ?");

System.***err***.println(" certificatPath (String) : Chemin des keystores (default : .");

System.***err***.println(" debugEnable (true/false) : activer le debug (default : false");

System.*exit*(1);

}

**private** **static** **boolean** getBoolean(String arg) **throws** Exception {

**boolean** b;

**if** (arg != **null** && ("true".equals(arg.toLowerCase()) || "false".equals(arg.toLowerCase()))) {

b = Boolean.*valueOf*(arg);

} **else** {

**throw** **new** Exception();

}

**return** b;

}

**static** **void** startServer(**int** port) **throws** IOException, NoSuchAlgorithmException {

ServerSocketFactory factory = SSLServerSocketFactory.*getDefault*();

**try** (ServerSocket listener = factory.createServerSocket(port)) {

**if** (*IS\_DEBUG\_ENABLE*) {

*sslInfos*(listener);

}

((SSLServerSocket) listener).setNeedClientAuth(*CLIENT\_AUTHENTICATION\_REQUIERED*);

((SSLServerSocket) listener).setEnabledCipherSuites(**new** String[] { "TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA384" });

((SSLServerSocket) listener).setEnabledProtocols(**new** String[] { "TLSv1.2" });

// while (true) {

**try** (Socket socket = listener.accept()) {

**try** (Scanner response = **new** Scanner(socket.getInputStream())) {

**if** (response.hasNextLine()) {

String clientName = response.next();

System.***out***.println("Hello " + clientName);

}

}

}

}

}

**private** **static** **void** sslInfos(ServerSocket listener) {

System.***out***.println();

System.***out***.println("Ciphers suite enable");

String[] enabledCipherSuites = ((SSLServerSocket) listener).getEnabledCipherSuites();

**for** (String cipher : enabledCipherSuites) {

System.***out***.println(cipher);

}

System.***out***.println();

System.***out***.println("Protocols enable");

String[] enabledProtocols = ((SSLServerSocket) listener).getEnabledProtocols();

**for** (String protocol : enabledProtocols) {

System.***out***.println(protocol);

}

System.***out***.println();

}

}

# Un client SSL

Création d'un keystore client avec un certificat auto-signé:

keytool -genkey -keyalg RSA -keypass password -storepass password -keystore jks.client.keystore -alias client

Vérification du contenu du keystore client :

keytool -list -storepass password -v -keystore jks.client.keystore

*Keystore type: JKS*

*Keystore provider: SUN*

*Your keystore contains 2 entries*

*Alias name: client*

*Creation date: 12 févr. 2020*

*Entry type: PrivateKeyEntry*

*Certificate chain length: 1*

*Certificate[1]:*

*Owner: CN=client2, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Issuer: CN=client2, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Serial number: 6e4aa199*

*Valid from: Wed Feb 12 23:42:59 CET 2020 until: Sat Feb 06 23:42:59 CET 2021*

*Certificate fingerprints:*

*SHA1: 4E:4B:B0:6B:4A:C8:C0:7A:23:CE:A9:C6:39:53:8F:CB:69:7D:87:9E*

*SHA256: A9:B0:50:67:1D:CD:3F:69:35:60:69:0E:DF:D3:21:68:01:96:29:C2:D4:9A:25:66:BE:9F:47:F5:78:70:74:AE*

*Signature algorithm name: SHA256withRSA*

*Subject Public Key Algorithm: 2048-bit RSA key*

*Version: 3*

*Extensions:*

*#1: ObjectId: 2.5.29.14 Criticality=false*

*SubjectKeyIdentifier [*

*KeyIdentifier [*

*0000: 52 B5 7B AF D0 15 ED 68 5C 57 44 E2 54 D0 62 86 R......h\WD.T.b.*

*0010: 4E 56 E2 B2 NV..*

*]*

*]*

Extraction du certificat du keystore client:

keytool -export -storepass password -keystore jks.client.keystore -file client.cer -alias client

Ajout du certificat client dans le truststore server:

keytool -import -v -trustcacerts -keypass password -storepass password -file client.cer -keystore jks.server.truststore -alias client

Vérification du contenu du truststore server

keytool -list -storepass password -v -keystore jks.server.truststore

*Keystore type: JKS*

*Keystore provider: SUN*

*Your keystore contains 2 entries*

*Alias name: client*

*Creation date: 12 févr. 2020*

*Entry type: trustedCertEntry*

*Owner: CN=client2, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Issuer: CN=client2, OU=Bureau Web, O=PenPen, L=Paris, ST=IDF, C=FR*

*Serial number: 6e4aa199*

*Valid from: Wed Feb 12 23:42:59 CET 2020 until: Sat Feb 06 23:42:59 CET 2021*

*Certificate fingerprints:*

*SHA1: 4E:4B:B0:6B:4A:C8:C0:7A:23:CE:A9:C6:39:53:8F:CB:69:7D:87:9E*

*SHA256: A9:B0:50:67:1D:CD:3F:69:35:60:69:0E:DF:D3:21:68:01:96:29:C2:D4:9A:25:66:BE:9F:47:F5:78:70:74:AE*

*Signature algorithm name: SHA256withRSA*

*Subject Public Key Algorithm: 2048-bit RSA key*

*Version: 3*

*Extensions:*

*#1: ObjectId: 2.5.29.14 Criticality=false*

*SubjectKeyIdentifier [*

*KeyIdentifier [*

*0000: 52 B5 7B AF D0 15 ED 68 5C 57 44 E2 54 D0 62 86 R......h\WD.T.b.*

*0010: 4E 56 E2 B2 NV..*

*]*

*]*

**package** ssl;

**import** java.io.File;

**import** java.io.IOException;

**import** java.io.OutputStream;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**import** javax.net.SocketFactory;

**import** javax.net.ssl.SSLParameters;

**import** javax.net.ssl.SSLSocket;

**import** javax.net.ssl.SSLSocketFactory;

**public** **class** SimpleClient {

**private** **static** **final** String ***HOSTNAME*** = "localhost";

**private** **static** **boolean** *IS\_DEBUG\_ENABLE* = **false**;

**public** **static** **void** main(String[] args) **throws** IOException {

*setSslProperties*(args);

*startClient*(***HOSTNAME***, SimpleServer.***SERVER\_PORT***);

}

**private** **static** **void** setSslProperties(String[] args) {

**if** (args.length < 1) {

*usage*();

}

**boolean** knownServer = **false**;

// knownServer

**try** {

knownServer = *getBoolean*(args[0]);

System.***out***.println("knownServer : "+ knownServer);

} **catch** (Exception e) {

System.***err***.println("Erreur sur knownServer");

*usage*();

}

// certificatPath

String rootResources = **null**;

**if** (args.length > 1) {

File certificatsPath = **new** File(args[1]);

**if** (certificatsPath.exists()) {

rootResources = certificatsPath.getAbsolutePath();

} **else** {

System.***err***.println("Le repertoire "+ args[2] + " n'existe pas");

*usage*();

}

} **else** {

rootResources = **new** File(".").getAbsolutePath();

}

rootResources = rootResources.endsWith(File.***separator***) ? rootResources : rootResources+File.***separator***;

System.***out***.println("Current certificats path is : "+ rootResources);

// debugEnable

**if** (args.length > 2) {

**try** {

*IS\_DEBUG\_ENABLE* = *getBoolean*(args[2]);

} **catch** (Exception e) {

System.***err***.println("ERROR : The seconde argument, debugEnable, must be true or false.");

}

}

System.***out***.println("Debug enable is " + *IS\_DEBUG\_ENABLE*);

System.*setProperty*("javax.net.ssl.keyStore", rootResources + "jks.client.keystore");

System.*setProperty*("javax.net.ssl.keyStorePassword", "password");

**if** (knownServer) {

System.*setProperty*("javax.net.ssl.trustStore", rootResources + "jks.client.truststore");

System.*setProperty*("javax.net.ssl.trustStorePassword", "password");

}

**if** (*IS\_DEBUG\_ENABLE*) {

System.*setProperty*("javax.net.debug", "ssl:all");

System.*setProperty*("java.security.debug", "access:stack");

}

}

**private** **static** **void** usage() {

System.***err***.println("Usage : java SimpleClient.class clientAuthentication knownClient certificatPath debugEnable");

System.***err***.println(" knownServer (true/false) : Est-ce que le client connait le server ?");

System.***err***.println(" certificatPath (String) : Chemin des keystores (default : .");

System.***err***.println(" debugEnable (true/false) : activer le debug (default : false");

System.*exit*(1);

}

**private** **static** **boolean** getBoolean(String arg) **throws** Exception {

**boolean** b;

**if** (arg != **null** && ("true".equals(arg.toLowerCase()) || "false".equals(arg.toLowerCase()))) {

b = Boolean.*valueOf*(arg);

} **else** {

**throw** **new** Exception();

}

**return** b;

}

**static** **void** startClient(String host, **int** port) **throws** IOException {

SocketFactory factory = SSLSocketFactory.*getDefault*();

**try** (Socket connection = factory.createSocket(host, port)) {

**if** (*IS\_DEBUG\_ENABLE*) {

*sslInfos*(connection);

}

((SSLSocket) connection).setEnabledCipherSuites(**new** String[] { "TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA384"});

((SSLSocket) connection).setEnabledProtocols(**new** String[] { "TLSv1.2" });

SSLParameters sslParams = **new** SSLParameters();

sslParams.setEndpointIdentificationAlgorithm("HTTPS");

((SSLSocket) connection).setSSLParameters(sslParams);

OutputStream outputStream = ((SSLSocket) connection).getOutputStream();

PrintWriter out = **new** PrintWriter(outputStream, **true**);

out.println("Client1");

}

}

**private** **static** **void** sslInfos(Socket connection) {

System.***out***.println();

System.***out***.println("Ciphers suite enable");

String[] enabledCipherSuites = ((SSLSocket) connection).getEnabledCipherSuites();

**for** (String cipher : enabledCipherSuites) {

System.***out***.println(cipher);

}

System.***out***.println();

System.***out***.println("Protocols enable");

String[] enabledProtocols = ((SSLSocket) connection).getEnabledProtocols();

**for** (String protocol : enabledProtocols) {

System.***out***.println(protocol);

}

System.***out***.println();

}

}

# La poignée de main en SSL unidirectionnel (The Handshake in One-way SSL)

Si nous nous référons aux étapes mentionnées ci-dessus, la deuxième étape mentionne l'échange de certificats.

SSL unidirectionnel requiert qu'un client puisse faire confiance au serveur via son certificat public (client).

Cela oblige le serveur à approuver tous les clients qui demandent une connexion.

Il n'y a aucun moyen pour un serveur de demander et de valider le certificat public auprès des clients, ce qui peut poser un risque pour la sécurité.

# La poignée de main en SSL bidirectionnel (The Handshake in Two-way SSL)

Avec SSL unidirectionnel, le serveur doit approuver tous les clients. Mais, le SSL bidirectionnel ajoute la possibilité pour le serveur de pouvoir également se connecter à des clients de confiance.

Lors d'une négociation bidirectionnelle, le client et le serveur doivent se présenter et accepter les certificats publics de chacun avant qu'une connexion réussie puisse être établie.

# Scénarios d’erreur

|  |  |  |  |
| --- | --- | --- | --- |
| Authentification client demandée | Server connait client  (cert client dans trust server) | Client connait Server  (Cert server dans trust Client) | Résultat |
| Non | Non | Non | KO :   * Server : Received fatal alert: certificate\_unknown * Client : unable to find valid certification path to requested target |
| Non | Non | Oui | OK |
| Non | Oui | Non | KO :   * Server : Received fatal alert: certificate\_unknown * Client : unable to find valid certification path to requested target |
| Non | Oui | Oui | OK |
| Oui | Non | Non | KO :   * Server : Received fatal alert: certificate\_unknown * Client : unable to find valid certification path to requested target |
| Oui | Non | Oui | KO :   * Server : null cert chain * Client : Software caused connection abort: socket write error |
| Oui | Oui | Non | KO :   * Server : Received fatal alert: certificate\_unknown * Client : unable to find valid certification path to requested target |
| Oui | Oui | Oui | OK |

## Certificat de serveur manquant:

Exception in thread "main" javax.net.ssl.SSLHandshakeException:

Received fatal alert: handshake\_failure

-> le client lors de la connexion au serveur n'a reçu aucun certificat.

Solution : -Djavax.net.ssl.keyStore=clientkeystore.jks -Djavax.net.ssl.keyStorePassword=password

Attention : La propriété système Java pour le magasin de clés ne prend pas en charge les chemins relatifs.

## Certificat de serveur non approuvé - authentification server requise

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

-> Le serveur utilise un certificat auto-signé qui n'est pas signé par une autorité de certification (CA).

Le client doit connaitre le serveur

Solution : -Djavax.net.ssl.trustStore=clienttruststore.jks -Djavax.net.ssl.trustStorePassword=password

Dans un scénario idéal, nous ne devrions pas utiliser un certificat auto-signé mais un certificat qui a été certifié par une autorité de certification (CA) à laquelle les clients peuvent faire confiance par défaut.

## Certificat client manquant - authentification client requise

Exception in thread "main" java.net.SocketException:

Software caused connection abort: recv failed

-> L'exception SocketException nous indique ici que le serveur ne pouvait pas faire confiance au client.

Le serveur doit connaitre le client

En effet, nous avons mis en place un SSL bidirectionnel: ((SSLServerSocket) listener).setNeedClientAuth(true);

Ce code indique que SSLServerSocket est requis pour l'authentification du client via leur certificat public.

Solution : Nous pouvons créer un fichier de clés pour le client et un fichier de clés de confiance correspondant pour le serveur d'une manière similaire à celle que nous avons utilisée lors de la création du fichier de clés et du fichier de clés de confiance précédents. Nous redémarrerons le serveur et lui transmettrons les propriétés système suivantes:

-Djavax.net.ssl.keyStore=serverkeystore.jks \

-Djavax.net.ssl.keyStorePassword=password \

-Djavax.net.ssl.trustStore=servertruststore.jks \

-Djavax.net.ssl.trustStorePassword=password

Ensuite, nous redémarrerons le client en transmettant ces propriétés système:

-Djavax.net.ssl.keyStore=clientkeystore.jks \

-Djavax.net.ssl.keyStorePassword=password \

-Djavax.net.ssl.trustStore=clienttruststore.jks \

-Djavax.net.ssl.trustStorePassword=password

## Certificats incorrects

Exception in thread "main" javax.net.ssl.SSLHandshakeException:

java.security.cert.CertificateException:

No name matching localhost found

-> keytool -v -list -keystore serverkeystore.jks : nous pouvons voir les détails du magasin de clés, en particulier le propriétaire (owner)

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

Le CN du propriétaire de ce certificat est défini sur localhost. Le CN du propriétaire doit correspondre exactement à l'hôte du serveur. S'il y a une incompatibilité, cela entraînera une exception SSLHandshakeException.

Si nous régénérons le certificat de serveur avec CN comme autre chose que localhost alors SimpleServer et SimpleClient plantent !

La trace d'exception ci-dessus indique clairement que le client attendait un certificat portant le nom d'hôte local qu'il n'a pas trouvé.

Veuillez noter que JSSE n'impose pas la vérification du nom d'hôte par défaut.

Nous avons activé la vérification du nom d'hôte dans SimpleClient grâce à l'utilisation explicite de HTTPS:

SSLParameters sslParams = new SSLParameters();

sslParams.setEndpointIdentificationAlgorithm("HTTPS");

((SSLSocket) connection).setSSLParameters(sslParams);

## Incompatible SSL Version

Exception in thread "main" javax.net.ssl.SSLHandshakeException:

No appropriate protocol (protocol is disabled or cipher suites are inappropriate)

-> si le serveur utilise un protocole cryptographique SSL3 et que le client utilise TLS1.3,

ils ne peuvent pas se mettre d'accord sur un protocole cryptographique et une SSLHandshakeException sera générée.

(Par exemple, dans le client : ((SSLSocket) connection).setEnabledProtocols(new String[] { "TLSv1.1" }); )

Solution : Pour résoudre ces types de problèmes, il est nécessaire de vérifier que le client et le serveur utilisent à la fois les mêmes protocoles cryptographiques ou des protocoles compatibles.

## Ciphers imcompatibles

Exception in thread "main" javax.net.ssl.SSLHandshakeException:

Received fatal alert: handshake\_failure

-> Lors d'une poignée de main, le client présentera une liste de chiffrements possibles à utiliser et le serveur répondra avec un chiffrement sélectionné dans la liste.

Le serveur générera une exception SSLHandshakeException s'il ne peut pas sélectionner un chiffre approprié.

(Par exemple, dans le client : ((SSLSocket) connection).setEnabledCipherSuites( new String[] {"TLS\_RSA\_WITH\_AES\_128\_GCM\_SHA256" }); )

Si vous rencontrez cette erreur, c'est généralement parce que le serveur a été configuré pour utiliser un chiffrement très sélectif

Solution : vérifier les suites de chiffrement actives utilisées par le client et le serveur et s'assurer qu'au moins une suite commune est disponible.

## Mauvais mot de passer

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

-> cela se produit lorsque la clé du magasin de clés est accessible avec le mauvais mot de passe

Solution : Changer le password: keytool -keypasswd -alias <keyalias> -keystore <keystore>

## Chaînes CA incomplètes

javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated

Solution: Activez donc les logs SSL, exécutez à nouveau et nous commencerons par analyser la poignée de main.

Nous allons commencer par regarder du côté client. Si nous examinons la log, nous trouvons le message

CertificateRequest suivant du serveur et du ServerHelloDone.

\*\*\* CertificateRequest

Cert Types: RSA, DSS

Cert Authorities:

\*\*\* ServerHelloDone

Jusqu'à présent, tout s'est bien passé. Le serveur a déjà envoyé son certificat, et puisque notre client ne lance pas d'erreur sur cette partie, nous pouvons supposer qu'il est approuvé par le client. Il semble donc que quelque chose ne va pas avec les étapes qui suivent ce message du serveur.