**Assignment Report for IK 2001: Java and E-Commerce Security**

**Group Member:** Mahboobeh AbdalMahmoodAbadi, 840706-6862

Mehran Nasseri, 800921-8192

**Assignment Number and name:** Assignment 4 (Client/Server Message Security)

1. **Program description**

An AC module is developed, so the user can register/ login and server accept their (typed) message. The module protects the channel by the SSL (through JSSE) and protects messages by the PKCS#7standard according to the RFC2315.

* 1. **How to run**

Run the SSLServerV2 and ClientUI. The program uses JDBC to connect to the database and to store/retrieve login information. Now through the client user interface, it is possible to register or login. After logging in, the client can send the typed message to the server and the server will display the received message along with the signer’s certificate info, if the signature is verified.

* + 1. **Run options**

Server side application can be run with the following options:

-port <port of server> -ts <truststore> -tspass <truststore password> -ks <keystore> -kspass <keystore password>

Client side application can be run with the following options:

-host <host of server> -port <port of server> -ks <keystore> -kspass <keystore password> -ts <truststore > -tspass <truststore password> –alias <alias to use>

If you ignore any of these options the application will use the default values.

* 1. **SSL channel protection**

JSSE makes it possible for applications to communicate securely via SSL. In order to allow transport security, a key-store and trust-store is created using keytool.

* + 1. **SSLServerV2**

The constructor of the SSLServerV2 obtains the SSLServerSocketFactory, by calling the getSSLServerSocketFactory(). getSSLServerSocketFactory() sets up various SSLSocketFactory, with different KeyStore and TrustStore settings. Without such a method, every connection in a JVM must use the same Key and Trust store. This method is necessary for large application, in which it may need to connect to several peers on behalf of various users. It also enabled us to get rid of long command line to run the application.

getSSLServerSocketFactory calls getKeyManagers(), which returns KeyManager[]. The getSSLServerSocketFactory then calls getTrustManagers(), which returns TrustManager[].

getKeyManagers() and getTrustManagers() overrides the Key and trust store dynamically. These methods facilitate to establish a TrustManagerFactory and a KeyManagerFactory and use these to initialize SSLContext. getSSLServerSocketFactory then uses the SSLContext to create an SSLServerSocketFactory. In the SSLContext "SSL" protocol is stated to allow JSSE to the maximum level it can support. When initializing SSLContext it gets KeyManager array, TrustManager array and the last one the source of randomness for this generator. The random number generator is a critical part of SSL, and the wrongly use of it could cause connections to be insecure. Therefore, it is set to be null to let JSSE to employ default random number generator.

Server runs on a different thread, so main() exits when the server is running. Now the run() method creates an SSLServerSocket, to accept the incoming connection and necessitates client authentication:

SSLServerSocket sss= (SSLServerSocket)serverSocketFactory.createServerSocket(port);

sss.setNeedClientAuth(true);

Now server starts new thread for each accepted incoming connection.

Each socket use HandshakeCompletedListener, to present the peer's name indicated in certificate. The socket's InputStream and OutputStream are wrapped in an InputHandler. It runs on different thread and displays coming messages to System.out.

while(true) {

String ids=String.valueOf(id++);

SSLSocket ss= (SSLSocket)sss.accept();

HandshakeCompletedListener x=new SimpleHandshakeListener(ids);

ss.addHandshakeCompletedListener(x);

InputStream ins=ss.getInputStream();

OutputStream outs = ss.getOutputStream();

InputHandler inputHandler = new InputHandler (ids, ins, outs);

}

* + 1. **SSLClient**

Similar to the getSSLServerSocketFactory() method in server side, getSSLSocketFactory() is used in client side. SSLSocketFactory is used to create JSSE secure socket. getSSLSocketFactory() allows to adapt the actions of JSSE. This method returns a SSLSocketFactory object, which is used to construct the client side SSLSocket. When the SSL connection established the client can send messages to the server.

However getSSLSocketFactory() is slightly different from its version in server side. It uses custom KeyManager to pick an alias. The client KeyStore may have a number of recognized certificates (in our case Alice and Bob). Therefore, JSSE will decide either certificate. In order to tell the application to choose specific certificate –alias command option should be used.

To obligate the picking of an exacting alias, X509KeyManager is implemented (public class AliasChoosingKeyManager implements X509KeyManager). The only method it overrides is chooseClientAlias(). It verifies validity of alias, and returns it if it is valid.

The X509KeyManager interface employs some methods during the SSL handshake to get the key and identify the peer, as follow:

* getClientAliases() returns an array of acceptable aliases for SSLSockets.
* getServerAliases() returns an array of acceptable aliases for SSLServerSockets.
* chooseClientAlias() provides a acceptable alias.
* chooseServerAlias() provides a acceptable alias
* getCertificateChain() gets the alias as an argument, and return an array of X509Certificate[].
* getPrivateKey() gets the alias as an argument and returns corresponding PrivateKey.

*AliasChoosingKeyManager*

JSSE calls chooseClientAlias. chooseClientAlias calls getClientAliases to determine a record of acceptable aliases so to ensure if the desired alias is valid. JSSE calls getPrivateKey and getCertificateChain to bind KeyManager. The chooseClientAlias() method of AliasChoosingKeyManager requires to call getClientAliases() several times, one time for each one of supported key type by the JSSE:

public String chooseClientAlias(String[] keyType, Principal[] issuers, Socket socket){

boolean found=false;

for (int i=0; i<keyType.length && !found; i++) {

String[]valid=baseKM.getClientAliases(keyType[i],issue);

if (valid!=null) {

for (int j=0; j<valid.length && !found; j++) {

if (valid [j].equals(alias)) found=true;

}

}

}

if (found) return alias;

else return null;

}

Now AliasChoosingKeyManager can be used in preference to the normal KeyManager. Unlike getSSLServerSocketFactory, getSSLSocketFactory binds the return result of getKeyManagers in an instance of AliasChoosingKeyManager:

protected SSLSocketFactory getSSLSocketFactory()

throws IOException, GeneralSecurityException

{

KeyManager[] kms=getKeyManagers();

TrustManager[] tms=getTrustManagers();

if (alias!=null) {

for (int i=0; i<kms.length; i++) {

if (kms[i] instanceof X509KeyManager)

kms[i]=new AliasChoosingKeyManager((X509KeyManager)kms[i], alias);

}

}

SSLContext context=SSLContext.getInstance("SSL");

context.init(kms, tms, null);

SSLSocketFactory ssf=context.getSocketFactory();

return ssf; }

* 1. **Message protection PKSC#7 standard**

Each message send from client to the server is signed (using IAIK library) and enveloped (using BouncyCastle library) sequentially using PKCS#7 standard. On the server side the message is decrypted and then the signature is checked to be verified.

Figure 1 represents the summary of the PKCS#7 standard which is used to sign and verify the data.

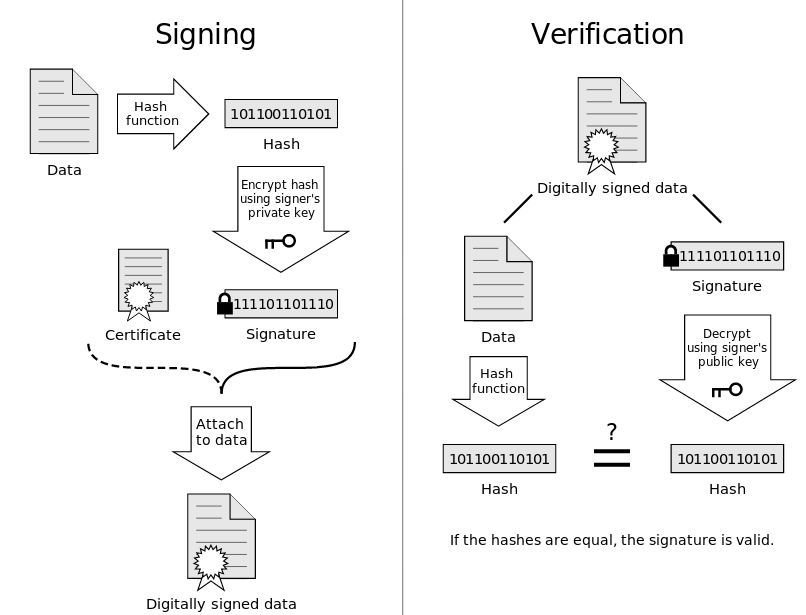


Figure 1.

* + 1. **Signing data (signData(String msg1))**

iaik.pkcs.pkcs7.SignedData class is used for implementation of the PKCS#7 digitally signed data. SignData object calculates message digest with a signer’s specific message-digest algorithm. Afterwards, the message digest is encrypted with the signer´s private key and – along with some signer information is added into a SignerInfo. Finally SignerInfo and the content form SignedData.

*The steps for SignedData creation and transmission:*

* Construct the SignedData object by the data that need to be signed and the transmission mode (*SignedData.IMPLICIT* ):

SignedData sd = new SignedData(msg1, SignedData.IMPLICIT);

* Set the signer’s certificates by setCertificates. The certificates are provided as array of X509Certificates:

sd.setCertificates((iaik.x509.X509Certificate[]) (X509Certificate[]) certChain);

* construct a SignerInfo object, and adding it to the SignedData:

IssuerAndSerialNumber iS = new IssuerAndSerialNumber(cert);

SignerInfo sI = new SignerInfo(iS, AlgorithmID.sha1, pkey);

sd.addSignerInfo(sI);

When adding a SignerInfo the digest calculation is carried out on the data with signer´s digest algorithm.

* DER encoding SignedData object and make it ready for transmission.

encoding = sd.getEncoded();

* + 1. **Enveloping data (encryptData(byte[] signedData))**

*The process of enveloping data:*

* A content-encryption key is generated at random.
* The key is encrypted with the recipient's public key.
* The encrypted key and recipient information are gathered into a RecipientInfo.
* The data is encrypted with the content-encryption key.
* The RecipientInfo and encrypted content is gathered into a CMSEnvelopedData object.
* Finally DER encoding CMSEnvelopedData object and make it ready for transmission.

CMSTypedData ctd = new CMSProcessableByteArray(signedData);

CMSEnvelopedDataGenerator edg = new CMSEnvelopedDataGenerator();

edg.addRecipientInfoGenerator(new JceKeyTransRecipientInfoGenerator(cert));

CMSEnvelopedData ed = edg.generate(ctd, new JceCMSContentEncryptorBuilder(CMSAlgorithm.DES\_EDE3\_CBC).setProvider("BC").build());

encoded=ed.getEncoded();

* + 1. **Parsing enveloped data (decryptData(byte[] encryptedData))**
* In order to open the envelope the encrypted content-encryption keys should be decrypted by using private key of the recipient
* Decrypt the encrypted message with the content-encryption key.

CMSEnvelopedDataParser edp = new CMSEnvelopedDataParser(encryptedData);

RecipientInformationStore ris = edp.getRecipientInfos();

Collection envCollection = ris.getRecipients();

Iterator it = envCollection.iterator();

RecipientInformation ris = (RecipientInformation) it.next();

byte[] decoded=ris.getContent(new JceKeyTransEnvelopedRecipient(privateKey).setProvider("BC"));

* + 1. **Verifying the signature (verify(byte [] asn))**

*Parsing SignedData content and signature verification:*

* The encoded content needs to be decoded to acquire an ASN1 Object:

ASN1Object asn\_obj = DerCoder.decode(asn);

* Since ASN1Object is an implicit SignedData object the SignedData(ASN1Object asn\_obj) is used:

SignedData sd = new SignedData(asn\_obj);

* Retrieve the SignerInfos and verifying the signature by calling the verify method:

SignerInfo[] signerInfos = sd.getSignerInfos();

for (int i=0; i < signerInfos.length; i++) {

try {

X509Certificate cert = sd.verify(i);

} catch (SignatureException ex) {

ex.printStackTrace();

}

}

* Finally, Retrieving the raw data:

byte[] buf=signedData.getContent();