Sikring af netværkskommunikation

Som udgangspunkt kan sikring af en netværkskommunikation foretages på et vilkårligt lag i netværksprotokolstakken.

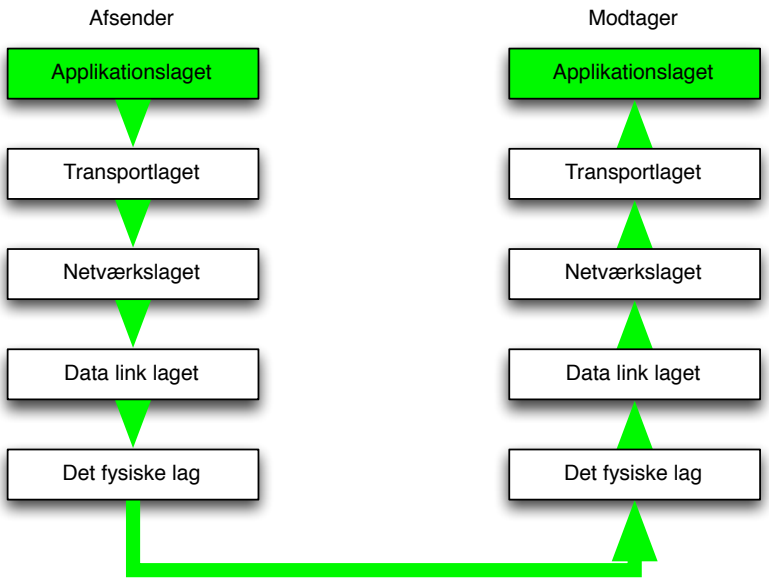
Hvis vi ser på TCP/IP protokolstakken vil det sige at en kommunikation over Internettet kan sikres ved:

- Sikring af det fysiske lag
- Sikring af data link laget
- Sikring af netværkslaget
- Sikring af transportlaget
- Sikring af applikationslaget

Disse 5 sikringsmetoder har forskellige formål, hvor det kun i begrænset omfang giver mening at anvende metoderne til andre formål.

I forbindelse med Java er det kun applikationslaget og transportlaget der er relevante.

Sikring på applikationslaget



Rent sikkerhedsmæssigt er den bedste måde at kommunikere sikkert på at lade meddelelserne være krypterede hele vejen fra den afsendende til den modtagende proces.
Det vil i praksis sige at brugeren styrer processen, og at sikringen af kommunikationen foregår på applikationsniveauet.

Sikring på applikationslaget

Formål:

At give mulighed for at information kan overføres sikkert mellem to processer (her typisk identisk med brugerne, da disse skal identificere sig for at få adgang til informationen).

Fordele:

- Maksimal sikring af data.
- Ingen krav til det underliggende netværk.
- Mulighed for at opbevare data krypteret/signeret.
- Autentificerer brugeren.

Ulemper:

- Alle klienter der skal kunne indgå i kommunikationen skal anvende samme sikkerhedsstandard og certifikater.
- Kræver nøglehåndtering, da der skal kunne kommunikeres sikkert med ukendte personer/maskiner.
- Hver anvendelse af internettet implementerer sin egen sikring.
- Bruger skal aktivt involveres.
- Netværksoplysninger (IP og port numre) står i klartekst.
- Autentificerer ikke afsendermaskinen.

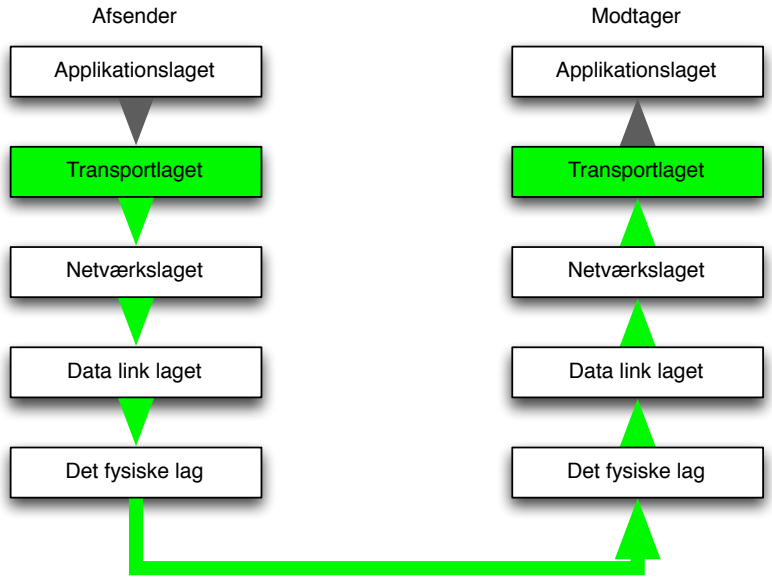
Sikring på applikationslaget

Problemet er at dette kræver at alle programmer der skal kunne afsende eller modtage sådanne meddelelser skal anvende den samme standard. For at dette skal være praktisk muligt skal for eksempel alle e-mail klienter anvende den samme standard og nøglecertifikater.

I praksis er dette næppe muligt at opnå, da der på nuværende tidspunkt er for mange e-mail klienter i brug.

Der er imidlertid ikke noget til hinder for at alle klienter anvender den samme krypteringsstandard, enten ved at alle implementerer den samme standard, eller ved at alle anvender de samme krypteringsbiblioteker.

Sikring på transportlaget



Sikring på transportlaget

Formål:

At kunne sikre på sessionsniveau, således at den samme proces både kan bruge sikre og usikre kommunikationer.

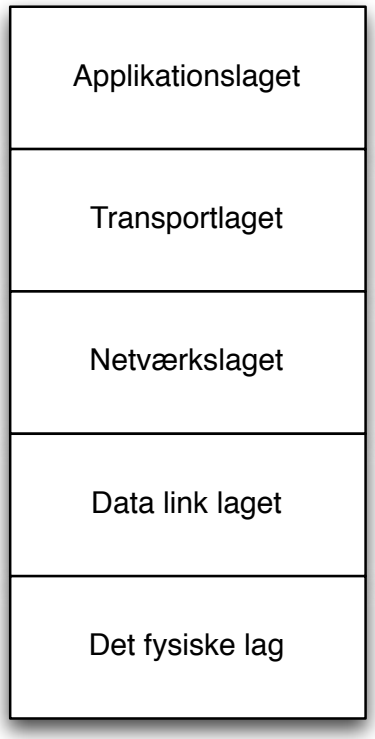
Fordele:

- Alle klienter der ønsker at anvende en sikker kommunikation kan anvende den samme API til kommunikationen, hvorfor denne ikke skal implementeres på applikationsniveau.
- Intet behov for en infrastruktur ud over internettet.
- Normalt transparent for brugere.
- Kan til en vis grad bruges til at autentificere brugeren af programmet, på bekostning af transparens.

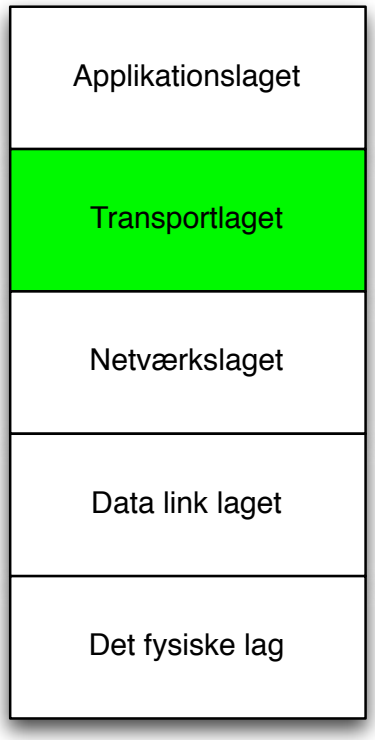
Ulemper:

- Data opbevares usikret på afsender/modtager.
- Der er behov for håndtering af certifikater og nøgler da der skal kommunikeres med et stort antal forskellige servere og services.
- Netværksoplysninger (IP og port numre) står i klartekst.
- Kan ikke autentificere afsendermaskinen.

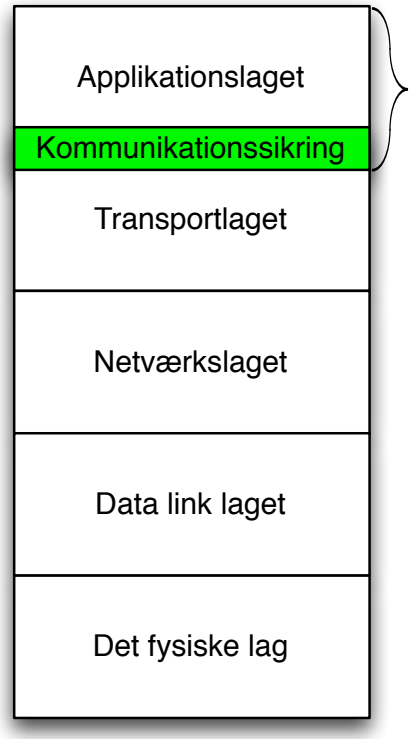
Usikret kommunikation



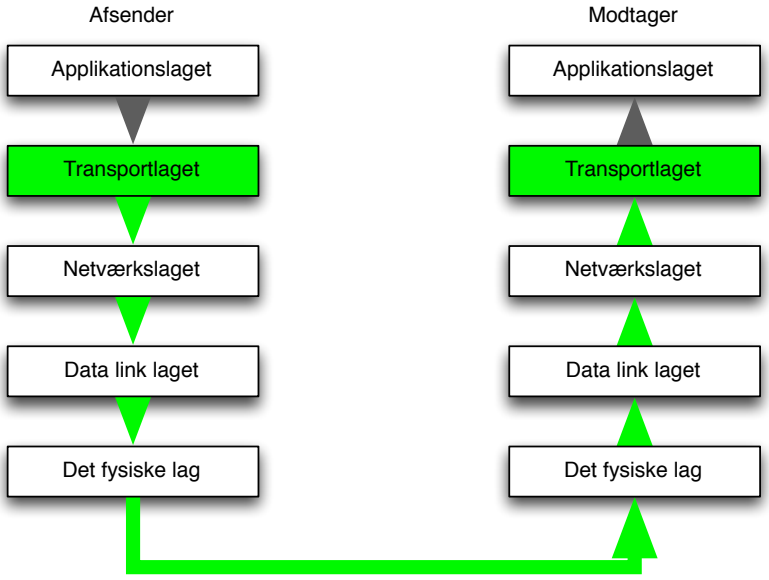
Sikret transportlag



Sikring via bibliotek



Sikring på transportlaget



SSL/TLS

En familie af protokoller:

SSLv2

SSLv3

TLS

Ligger ovenpå TCP således at det implementeres i de individuelle brugerprocesser. En anvendelse af SSL/TLS kræver derfor ikke ændring af det underliggende OS eller den anvendte TCP/IP stak.

Ikke bare en overførselsprotokol, men giver mulighed for at definere en API, således at den samme implementation kan anvendes fra mange forskellige brugerprocesser.

Typiske anvendelsesområder:

SSL server autentificering
SSL klient autentificering
En krypteret SSL session

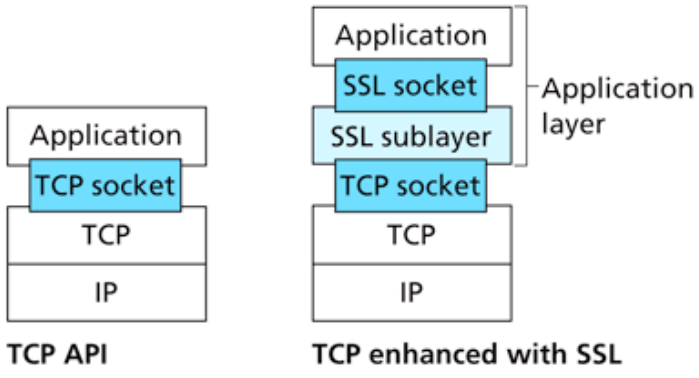


Figure 8.27 ♦ Although SSL technically resides in the application layer, from the developer's perspective it is a transport-layer protocol.

Kurose & Ross: *Computer Networking*, 4.ed.

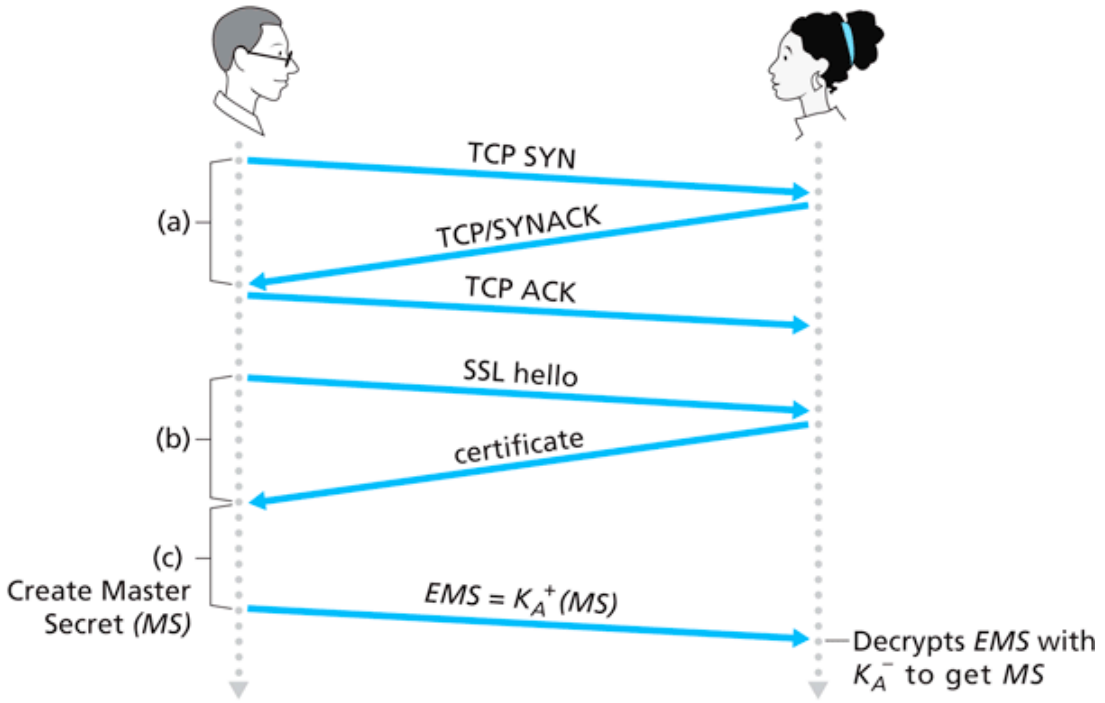


Figure 8.28 ♦ The almost-SSL handshake, beginning with a TCP connection

Kurose & Ross: *Computer Networking*, 4.ed.

Nøgleudledning

Klienten sender en krypteret Pre Master Secret til serveren, hvorefter begge anvender denne til at generere en Master Secret som de derefter anvender til at udlede 4 nøgler:

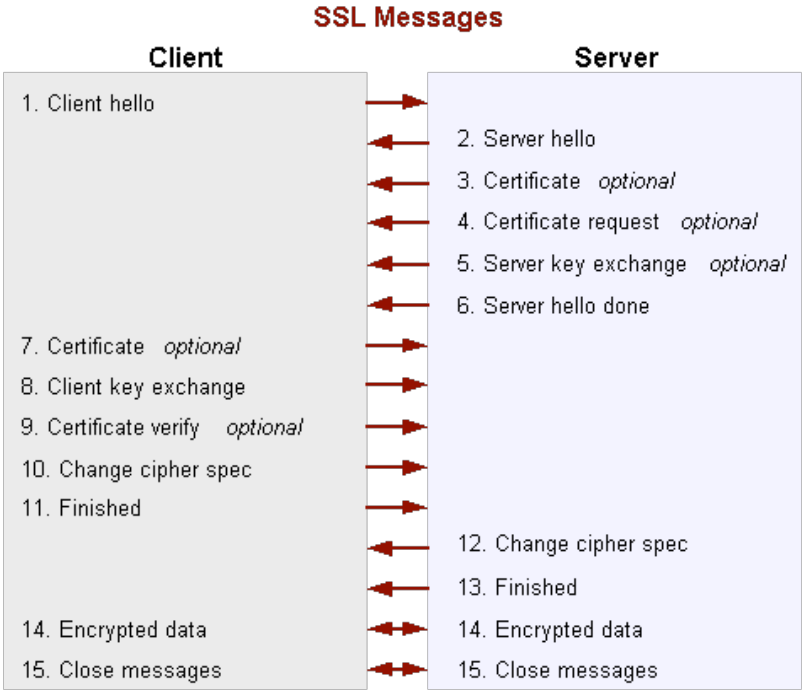
2 symmetriske sessionskrypteringsnøgler - en til hver retning

2 symmetriske sessions MAC nøgler - en til hver retning

Man kunne vælge at bruge premasternøglen som sessionsnøgle til både kryptering og MAC, men da man så anvender den samme nøgle i forbindelse med 2 algoritmer vil dette give ekstra information der kan bruges til at analysere sig frem til nøglen.

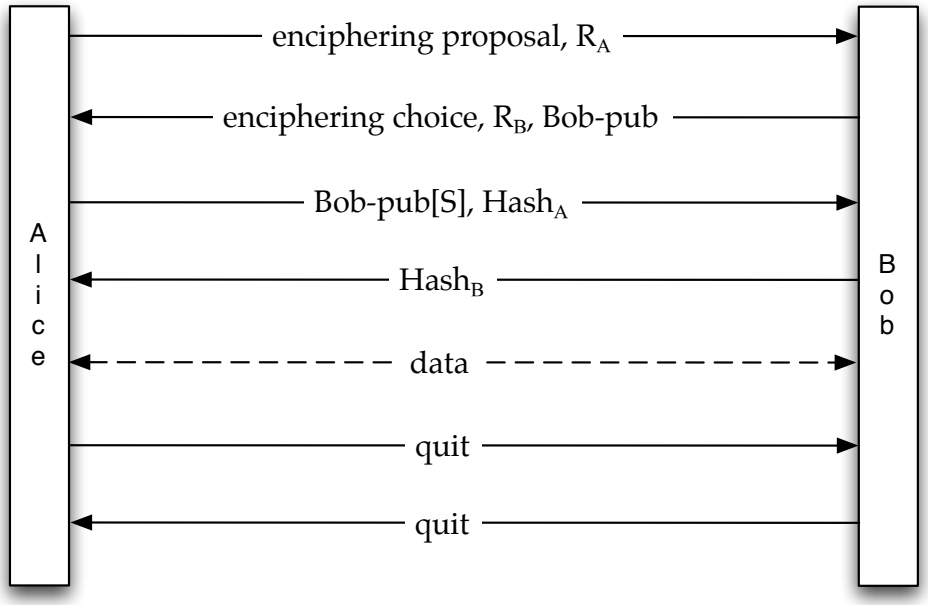
Desto flere meddelelser der bruger den samme nøgle, desto større er sandsynligheden for at man kan analysere sig frem til nøglen — specielt hvis flere meddelelser indeholder den samme information. Da en samtale har en tilbøjelighed til at en meddelelse gentager dele af en foregående meddelelse vil det være en fordel hvis der blev anvendt forskellige nøgler i de to retninger.

SSL protokol

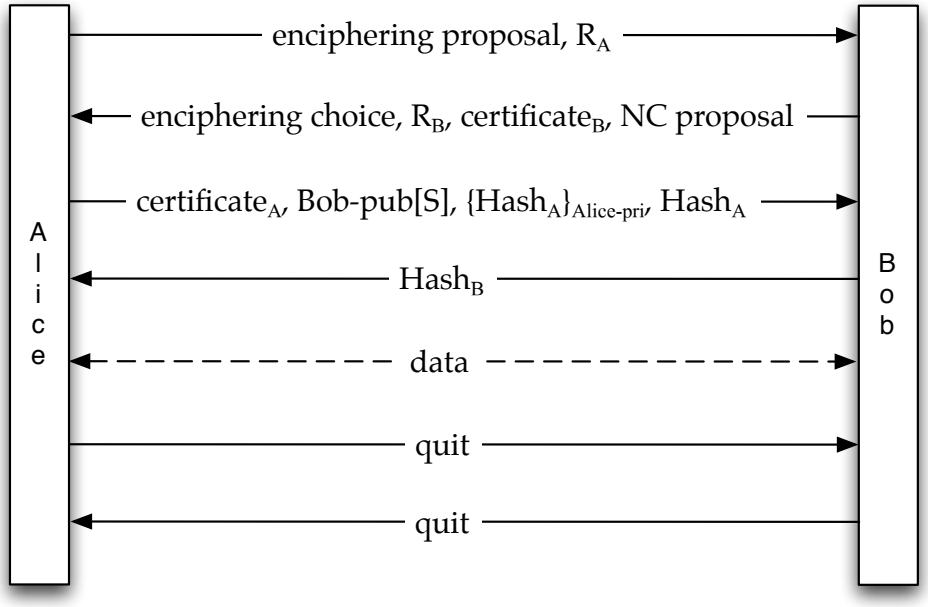


http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

SSL uden certifikater



Tovejs certifikat autentificering



Encryption possibilities

A numeric value defined by strings using "_" as delimiter.

Example:

SSL_RSA_EXPORT_WITH_DES40_CBC_SHA

Meaning:

SSL => SSLv3

RSA => RSA key exchange

EXPORT => weak encryption

WITH => filler!

DES40 => 40 bit DES encryption

CBC => Cipher Block Chaining

SHA => records has HMAC-SHA digest

Protocol

First in the description string.

SSL_: SSLv3

SSL2_: SSLv2

Key exchange

DHE_DSS_: Diffie-Hellman with DSS signatures

DHE_RSA_: Diffie-Hellman with RSA signatures

DH_anon_: Anonymous Diffie-Hellman

DH_DSS_: Diffie-Hellman with DSS certificates

DH_RSA_: Diffie-Hellman with RSA certificates

FORTEZZA_DMS_: Fortezza/DMS

NULL_: No key exchange

RSA_: RSA key exchange

Variants:

If followed by "EXPORT_" weak encryption is used (all but NULL and FORTEZZA_DMS).

Enciphering algorithms

3DES_EDE_CBC_

DES_CBC_

DES40_CBC_

FORTEZZA_CBC_

IDEA_CBC_

RC2_CBC_40_

RC4_128_

RC4_40_

NULL_

Hash algorithms

MD5

SHA

NULL

Java Secure Socket Extension

Udvidbart SSL/TLS bibliotek implementeret i ren Java.

Implementerer SSL 3.0 og TLS 1.0 (bagudkompatibel med SSL 2.0).

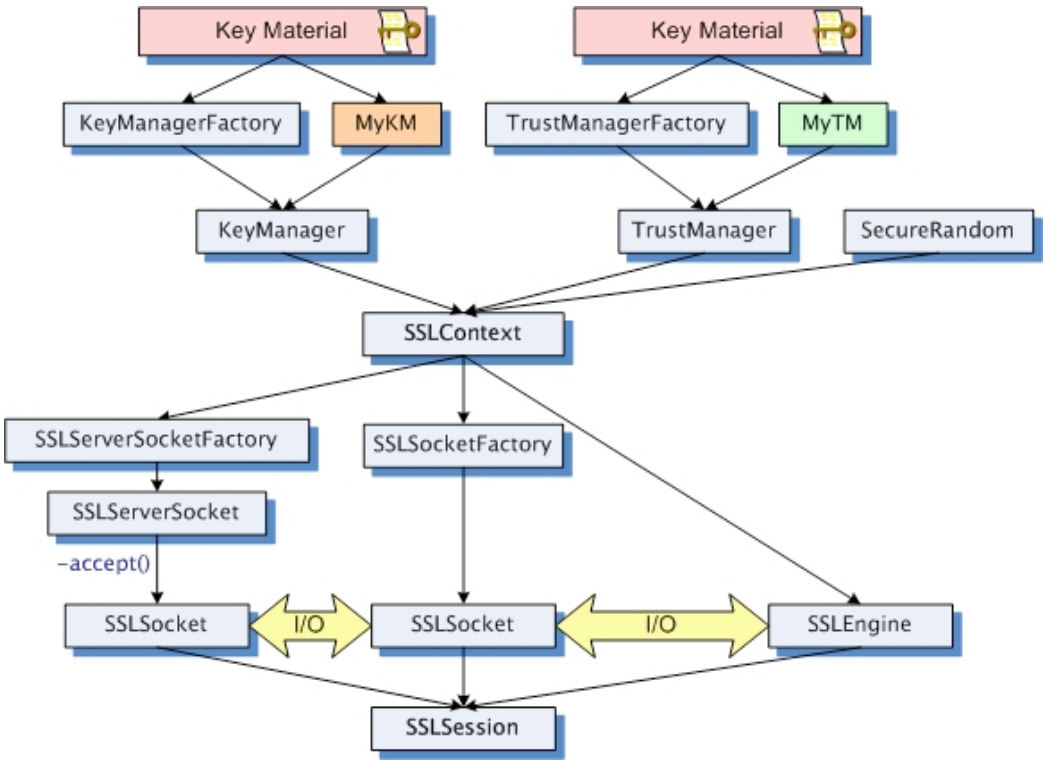
Indeholder klasser der leverer sikrede SSL forbindelser — både til server- og klient-siden.

Kan anvende:

- RSA (512 og større nøgler) til autentificering og nøgleudveksling.
- RC4 (40 og 128 bit nøgler), DES (56 og 40 bit nøgler), 3DES (112 bit nøgler) samt AES (128 og 256 bit nøgler) til datakryptering.
- Diffie-Hellman (512 og 1024 bit nøgler) til nøgleudveksling.
- DSA (1024 bit nøgler) til autentificering.

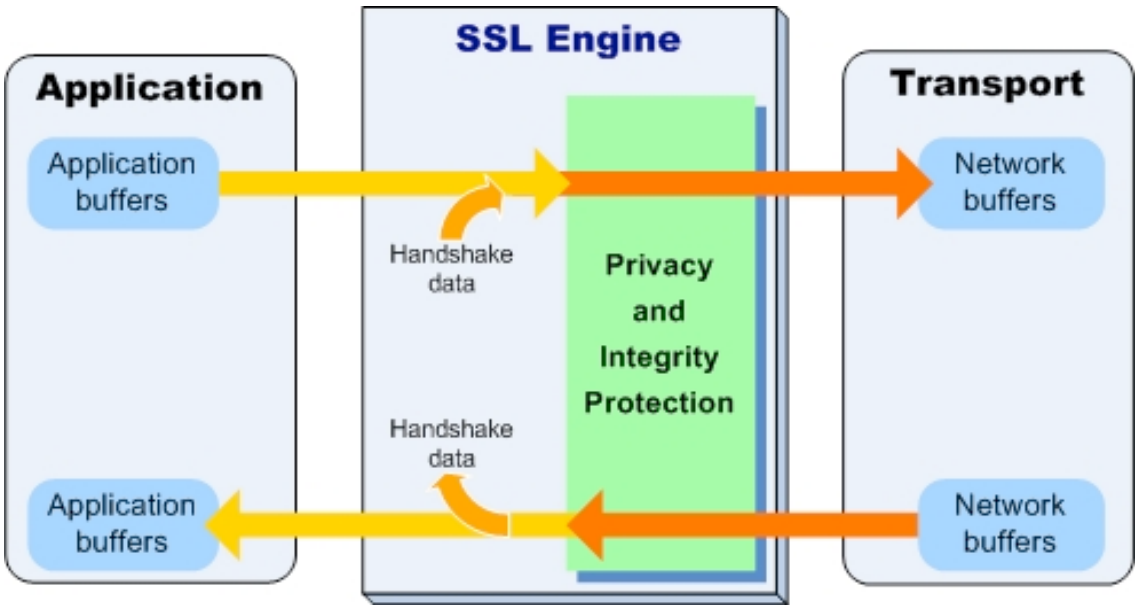
Baseret på JCA, og leveres som standard med en SSL provider, SunJSSE, der implementerer SSL 3.0 og TLS 1.0, samt anvendelse af X.509 certifikater der opbevares i et JCA keystore, med en simpel implementering af PKCS12

Overblik over JSSE klasser



http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

SSL Engine



http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html

Eksempel: HTTPS klient

```
import java.io.*;
import java.util.*;
import javax.net.ssl.*;

public class WebTest {
   public static void main(String argv[]) throws Exception {
      String inputLine;
      SSLSocketFactory sslFact =
                      (SSLSocketFactory)SSLSocketFactory.getDefault();
      SSLSocket s = (SSLSocket)sslFact.createSocket("www.thawte.com", 443);
      BufferedReader inFromServer;
      DataOutputStream outToServer;
      inFromServer = new BufferedReader(
                     new InputStreamReader(s.getInputStream()));
      outToServer = new DataOutputStream(s.getOutputStream());
      outToServer.writeBytes("GET / HTTP/1.1\r\n"+
                             "Host: www.thawte.com\r\n"+
                             "Connection: close\r\n\r\n");
      inputLine = inFromServer.readLine();
      while ( inputLine != null ) {
         System.out.println(inputLine);
        inputLine = inFromServer.readLine();
      }
      s.close();
   }
}
```

Eksempel: HTTPS server

```
. . .
import javax.net.ssl.*;
. . .
public class WebServer {
   public static void main(String argv[]) throws Exception
      . . .
      SSLServerSocketFactory sslSrvFact =
               (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
      SSLServerSocket listenSocket =
               (SSLServerSocket)sslSrvFact.createServerSocket(443);

      . . .
      while (true) {
         try {
            connectionSocket = (SSLSocket)listenSocket.accept();
            . . .
            // som normalt
            . . .
         } catch ( . . . ) {
            . . .
         }
      }
   }
}
```


Certifikater

Javas `keytool` værktøj kan bruges til at oprette og manipulere keystores og certifikater.

For at oprette et keystore (server.jks) og et *selfsigned* server certifikat (localhost) kaldes `keytool` med følgende argumenter:

- -genkey -alias server -keyalg RSA -keypass 123456 -storepass 123456 -keystore server.jks

Ved udførsel bedes om et navn, og her angives navnet på serveren (for eksempel localhost) hvor der bedes om for- og efternavn.

Indholdet af denne keystore kan ses med:

- keytool -list -storepass 123456 -keystore server.jks

For at bruge denne keystore kan den angives som argument når en SSL server startes:

- java -Djavax.net.ssl.keyStorePassword=123456
  -Djavax.net.ssl.keyStore=server.jks HTTPServer