Internet

Applicaties maken steeds vaker gebruik van informatiebronnen via het internet. Het wordt dan ook steeds belangrijker om de uitwisseling van informatie goed te beveiligen. In dit artikel wordt de vraag onderzocht of Java geschikt is voor beveiliging van informatiestromen.

Secure Sockets in Java

Arjan Lamers
First Eight

Aan de hand van enkele praktijkvoorbeelden wordt uitgelegd hoe in Java een beveiligde verbinding tot stand gebracht kan worden met behulp van SSL (Secure Sockets Layer) en welke begrippen en classes daarbij een rol spelen. Om een SSL-verbinding echt veilig te kunnen opbouwen en eventuele foutmeldingen te begrijpen, is een goed begrip nodig van de onderliggende concepten. Deze worden in het artikel uitgelegd. De verzameling van technieken die hierbij een grote rol heeft, wordt cryptografie genoemd.

Cryptografische begrippen

Om te voorkomen dat een bericht door iedereen kan worden gelezen, kunnen berichten geëncrypt worden. *Encryptie* is een wiskundige bewerking (een zogenaamde *cipher*) waarbij aan de hand van een key (of sleutel) het bericht zodanig wordt vervormd dat niemand het kan lezen als hij niet de goede key in bezit heeft. Als met diezelfde key het bericht terugvertaald kan worden, is er sprake van een *symmetrische cipher*. Er zijn ook ciphers die gebruik maken van een sleutelpaar. Wanneer een bericht gëencrypt wordt met de ene sleutel, kan het alleen ontcijferd of ge*decrypt* worden met de andere sleutel van het paar. Dit is aan de orde bij *asymmetrische ciphers*. Hierbij is het gebruikelijk dat één sleutel altijd geheim gehouden wordt (de *private key*) en de andere openbaar is (de *public key*). Om een bericht te sturen naar een persoon, bijvoorbeeld Bob, wordt zijn publieke sleutel gebruikt om te encrypten. Als Bob als enige beschikt over de bijbehorende private key kan hij ook als enige dat bericht lezen.

Een laatste begrip is de *one way hash*. Een hash is een wiskundige bewerking die een samenvatting van een bericht maakt. De in de cryptografie gebruikte hashes hebben de eigenschap dat, gegeven een hash, het lastig is om een bijpassend bericht te maken. Vandaar de term one way in de naam. De hash maakt een digitale handtekening mogelijk voor een bericht. Deze handtekening bestaat uit een hash van het bericht dat met de private key geëncrypt wordt. Door de handtekening mee te sturen, kan gecontroleerd worden van wie het bericht is en of de inhoud ongewijzigd is. Dit kan door de handtekening te decrypten met de publieke key van diegene die het bericht gestuurd zou hebben. De gedecrypte hash kan dan vergeleken worden met een nieuwe hash van het bericht. Als de hashes gelijk zijn moet de handtekening wel gezet zijn door iemand die de private key bezit.

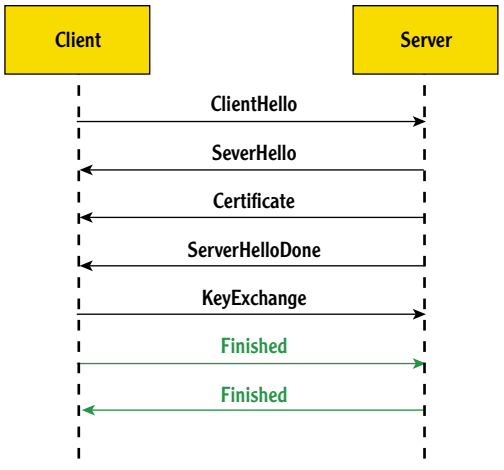
Public Key Infrastructure (PKI)

Om berichten uit te wisselen met een symmetrische cipher, moet met de andere partij een sleutel afgesproken worden, zonder dat derden deze informatie onderscheppen. Op internet is dit nog wel eens lastig. Dit probleem bestaat niet bij een asymmetrische cipher: daar wordt immers iemands openbare sleutel gebruikt. Dit verlegt echter het probleem: hoe weet men zeker dat men de juiste publieke sleutel heeft? Iemand kan immers zijn eigen sleutel aanbieden uit naam van bijvoorbeeld Bob. Om deze reden worden publieke sleutels vaak aangeboden in de vorm van een *certificaat* waarin ook Bob's personalia staan. Het bedrijf waar Bob werkt kan bijvoorbeeld garant staan voor

Bob's publieke sleutel en Bob's certificaat voorzien van een digitale handtekening. Met behulp van het certificaat (en dus de public key) van Bob's bedrijf kan geverifieerd worden of Bob's certificaat echt is; althans volgens Bob's bedrijf. Het certificaat van het bedrijf is weer getekend door een andere partij en zo ontstaat een keten van partijen die garant staan voor elkaar: een zogenaamde *chain of trust*. Aan de top van zo'n keten staat een *Certificate Authority (CA)*. Deze bedrijven, bijvoorbeeld Verisign en Thawte, tekenen hun eigen certificaat. Deze architectuur noemt men public key infrastructure (PKI). Iedereen die hieraan wil deelnemen, kiest zelf welke CA's hij wil vertrouwen. De verzameling van vertrouwde CA's wordt ook wel de *trust* genoemd. Elke internetbrowser wordt met een standaard lijst CA's geleverd. Ook de Java virtual machine heeft zo'n lijst. Deze is te vinden in jre\lib\security\cacerts.

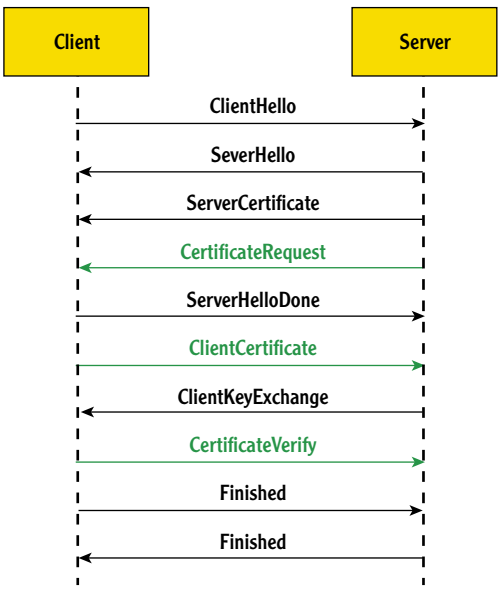
Secure Sockets Layer (SSL) en HTTPS

Secure Sockets Layer (SSL) is een protocol dat boven het op internet gebruikte transportprotocol TCP ligt. Er zijn nu drie versies waarvan de laatste versie ook wel TLS (Transport Layer Security) genoemd wordt. Dit protocol dient drie doelen: zorgen dat de berichten correct blijven (integriteit), het voorkomen van afluisteren (confidentialiteit), en het controleren van de identiteit van de server en optioneel ook die van de client (authentificatie). Applicatieprotocollen zoals HTTP kunnen, in plaats van TCP, SSL gebruiken om het bijvoorbeeld mogelijk te maken om bij webwinkels veilig creditcardgegevens over te sturen. De combinatie van HTTP en SSL wordt HTTPS genoemd.



Figuur 1: Basic SSL

Om een SSL-verbinding te maken wordt eerst een TCP-socket geopend. Vervolgens vindt de SSL-handshake plaats. In figuur 1 is vereenvoudigd weergegeven hoe dit plaatsvindt. Nadat de Hello's zijn uitgewisseld, reageert de server met zijn certificaat en de bijbehorende onderliggende certificaten. De client zoekt in zijn eigen verzameling CA's of het certificaat getekend is door een vertrouwde partij. In het geval van HTTPS wordt bovendien nog de common name uit het certificaat vergeleken met het opgevraagde domein. Als dat allemaal klopt wordt de verbinding verder opgebouwd. De client beschikt nu over de gecontroleerde publieke key van de server. De berichten die de client verstuurt kunnen daarmee dus geëncrypt worden, zodat ze alleen voor de server leesbaar zijn.
Voor de communicatie na de handshake wil men in plaats van een asymmetrische cipher een symmetrische cipher, omdat deze laatste veel sneller is. Vandaar dat door de client een sleutel voorgesteld wordt, geëncrypt met de publieke key van de server, die voor de rest van de sessie gebruikt gaat worden. De blauwe pijl in figuur 1 is dan ook asymmetrisch geëncrypt en de groene pijlen zijn symmetrisch geëncrypt.
Dit is de handshake in vogelvlucht. In detail zijn er veel meer stappen. Zo kan er onderhandeld worden over welke ciphers gebruikt mogen worden en hoe de key-exchange gaat plaatsvinden. Een optie is de client-authenticatie. In de hierboven beschreven handshake wordt alleen de identiteit van de server gecontroleerd. De server kan echter ook aan de client vragen om zich te authentificeren; zie figuur 2. De server stuurt dan een certificaatverzoek met daarin alle door de server geaccepteerde CA's. De client zoekt daarbij een certificaat dat de client identificeert, getekend door een van die CA's.



Figuur 2: SSL met clientauthenticatie

Keystore

Om de verschillende sleutels en certificaten centraal te beheren bestaat er in Java een `Keystore`.

Deze class bewaart de gegevens in-memory, maar kan deze ook wegschrijven naar en lezen van een wachtwoordbeveiligde stream. De al eerder genoemde cacerts-file is zo'n keystore. Bij de JRE wordt een command-line tool geleverd, genaamd `keytool`. Met deze tool is het mogelijk de gegevens in de keystore uit te breiden en aan te passen via de command-line. Om een SSL-verbinding te maken met een partij wier certificaat getekend is door een CA die Java niet standaard herkent, dan kan overwogen worden het certificaat van die CA toe te voegen aan de keystore:

```
$ keytool -import -alias mijn_ca -file certfile.cer
```

Het toegevoegde certificaat is nu geldig voor alle Java-applicaties. Dat kan te grof zijn voor bepaalde toepassingen. Het is ook mogelijk om per toepassing aparte keystores te maken (met de `-keystore` parameter bijvoorbeeld) of specifieker aan te geven welke CA's gebruikt mogen worden. Hoe dat voor SSL werkt, wordt verderop besproken.

SSL in Java

Java heeft SSL en HTTPS sinds versie 1.4 opgenomen in de standaard API. Voor oudere versies zijn de libraries apart te installeren. De belangrijkste zijn Java Cryptography Architecture (JCA) en Java Secure Socket Extension (JSSE).
Het gebruik van SSL is zo transparant mogelijk gemaakt. Een normale TCP-socket wordt in Java als volgt geopend:

```
Socket s = new Socket( "www.first8.nl", 80 );
s.getOutputStream();
s.getInputStream();
...
s.close();
```

Voor een SSL Socket gaat het op vrijwel dezelfde manier:

```
SSLSocketFactory ssf = SSLSocketFactory.getDe-
fault()
SSLSocket s = ssf.createSocket( "www.first8.nl",
443);
s.startHandshake();
s.getOutputStream();
s.getInputStream();
....
....
s.close();
```

Ook HTTPS wordt transparant ondersteund in Java:

```
HttpsUrlConnection con =
    (new Url( “https://www.first8.nl” )).openConnec-
tion();
con.connect();
…
```

Deze functies maken gebruik van de standaard aanwezige certificaten in de virtual machine (het al eerder genoemde `cacerts`-bestand).

SSLContext

Voor sommige toepassingen zijn bovenstaande voorbeelden voldoende. Wanneer meer controle noodzakelijk is, of andere partijen een slechte SSL-implementatie hebben, biedt Java ook uitgebreide mogelijkheden om in te grijpen. Dit kan door een zogenaamde `SSLContext` op te bouwen. Via deze class kunnen de randvoorwaarden gedefinieerd worden voor een SSL-verbinding. In plaats van de standaard SSLSocketFactory te gebruiken, wordt eerst een SSL Context opgebouwd waaraan een SSLSocketFactory gevraagd kan worden:

```
SSLContext con = SSLContext.createContext(“SSLv3”);
con.init(   key_managers,   trust_managers,   new
SecureRandom() );
SSLSocketFactory ssf = con.getSSLSocketFactory();
SSLSocket  s  =  ssf.createSocket(  “www.first8.nl”,
443);
s.startHandshake()
s.getOutputStream();
s.getInputStream();
....
```

Aan de init()-functie worden drie parameters meegegeven. De eerste – key_managers – is een lijst van keymanagers die de applicatie mogen vertegenwoordigen. Via de keymanagers wordt de identiteit van de applicatie beschreven. Via de tweede parameter – trust_managers – wordt beschreven wat de applicatie vertrouwt. De laatste – random – is minder relevant. Deze geeft aan welke random generator gebruikt moet worden om willekeurige sleutels te kunnen genereren. Voor echt zware beveiligingen kan het bijvoorbeeld noodzakelijk zijn om hardware random generators toe te passen. Aan de hand van enkele voorbeelden wordt het nut van de SSLContext duidelijk gemaakt.

Specifieke trust

Als het toevoegen van een CA aan de algemene keystore geen optie is, kan worden gebruikgemaakt van een aparte keystore voor een specifieke verbinding. Het volgende voorbeeld geeft aan hoe de trust aangepast kan worden.

```
KeyStore  ks  =  KeyStore.getInstance(KeyStore.getDe-
faultType());
ks.load( new FileInputStream( keystore_file ),
keystore_password.toCharArray() );
TrustManagerFactory tmf = TrustManagerFactory.getIn-
stance(  TrustManagerFactory.getDefaultAlgorithm()
);
tmf.init( ks );
SSLContext con = SSLContext.createContext();
con.init( key_managers, tmf.getTrustManagers(), new
SecureRandom());
```

Specifiek client-certificaat

Zoals eerder besproken geeft de server bij client-authenticatie aan welke CA's hij ondersteunt. Bij veel beveiligde toepassingen heeft de serverpartij een eigen CA en wordt een client-certificaat toegewezen. Bij de SSL-handshake wordt dan ook verwacht dat de client het juiste certificaat gebruikt. Het kan echter voorkomen dat de server niet volgens de richtlijnen is geconfigureerd of dat er verschillende client-certificaten passen. Een eigen KeyManager is dan de oplossing. Het volgende voorbeeld beschrijft een keymanager-wrapper die altijd een specifiek certificaat-alias teruggeeft, zodat precies dat certificaat uit de keystore kan worden gebruikt. De standaard voor certificaten binnen PKI is X.509. In dit voorbeeld wordt dan ook van dat formaat uitgegaan.

```
public   class   ForcedAliasKeyManager   implements
X509KeyManager {
  private X509KeyManager keymanager;
  private String certname;

  public ForcedAliasKeyManager(X509KeyManager _mana-
ger, String _certname ) {
        keymanager = _manager;
        certname = _certname;
  }

  public String chooseClientAlias(String[] keyType,
Principal[] ca, Socket s ) {
        return certname;
  }

  // ... implement other functions by calling super
}
```

Met deze wrapper kan de default X509KeyManager vervangen worden.

```
KeyManagerFactory  kmf  =  KeyManagerFactory.getIn-
stance(  KeyManagerFactory.getDefaultAlgorithm() );
KeyManager key_managers[] = kmf.getKeyManagers();
for( int i = 0;i < key_managers.length;i++ ) {
  KeyManager m = key_managers[ i ];
  if( m instanceof X509KeyManager ) {
        ForcedAliasKeyManager    man    =    new
ForcedAliasKeyManager(
 (X509KeyManager) m, client_cert_alias );
        key_managers[ i ] = man;
  }
}
```

Hostname verifier

In de praktijk komen geregeld test- of acceptatieomgevingen voor die gebruikmaken van de productiecertificaten. Hoewel het certificaat zelf geldig is, wordt de HTTPS-verbinding alsnog geweigerd, omdat de hostname niet overeenkomt met de common name op het certificaat. Ook hier is een oplossing voor. Het volgende voorbeeld accepteert elke hostname voor de HTTPS-verbinding, maar het mag duidelijk zijn dat hier nog eigen businessrules geïmplementeerd moeten worden.

```
HttpsURLConnection con = (HttpsURLConnection) (new
URL(     "https://www.first8.nl"    )).openConnec-
tion();
HostnameVerifier bogusVerifier = new HostnameVerifier()
{
   public boolean verify(String hostname, SSLSession
session) {
          return true;
   }
};
con.setHostnameVerifier( bogusVerifier );
```

Eenvoudig op te lossen

Java maakt het eenvoudig een SSL-verbinding op te bouwen. Met enige achtergrondkennis van SSL is het mogelijk de meest voorkomende problemen te begrijpen en op een veilige manier op te lossen. «