Unter vier Augen

Sicherheitsarchitektur mit Java und SSL

Christian Kücherer

Moderne Webanwendungen haben ein hohes Sicherheitsbedürfnis, da sensible Daten über öffentliche Netzwerke ausgetauscht werden. Die Sicherstellung von Datenintegrität und -geheimhaltung während der Übertragung sind daher grundlegende Anforderungen internetbasierter Softwarearchitekturen. Diese Anforderungen lassen sich durch den Einsatz des SSL-Protokolls realisieren. Immer wichtiger wird aber auch die verlässliche Authentifizierung der Kommunikationspartner. Angreifer können ohne passenden Schlüssel nicht auf Server zugreifen und Server können nicht durch böswillige Implementierungen ausgetauscht werden. Der folgende Artikel beschreibt die Technologie und zeigt anhand eines Beispiels, wie sie in Java-Applikationen verwendet werden kann.

Einleitung

Sicherheit spielt eine immer größere Rolle für Softwarearchitekturen. Internetbasierte Webapplikationen übertragen sensible Daten über öffentliche Netzwerke und sind dort verschiedensten Angriffen ausgesetzt. Daher muss gewährleistet werden, dass Daten während der Übertragung nicht verändert (Datenintegrität) oder von dritten Personen gelesen (Vertraulichkeit) werden können.

Moderne Softwarearchitekturen müssen diesen sicherheitsrelevanten Anforderungen gerecht werden. Eine mögliche Technologie, die zur Realisierung verwendet werden kann, ist das SSL-Protokoll. SSL wurde konzipiert, um Geheimhaltung und Integrität von Datenverkehr zwischen zwei Kommunikationspartnern in unsicheren Netzwerken mittels Verschlüsselung zu gewährleisten. Darüber hinaus bietet es die Client- und Server-Authentifizierung, deren Einsatz weitergehende Angriffe verhindert.

Im folgenden Abschnitt werden zunächst die Grundlagen für SSL beschrieben und Angriffszenarien vorgestellt. Anschließend wird eine Architektur vorgestellt, die den wichtigsten Angriffen standhält. Als Abschluss wird die Implementierung einer solchen Architektur in Java gezeigt.

Grundlagen

Die wichtigsten Grundlagen zu den Themenbereichen Sicherheit und SSL werden im Folgenden erläutert. Viele der Referenzen verweisen auf weiterführende Literatur zur Vertiefung der Themengebiete.

Zertifikate

Zertifikate bestätigen die Zugehörigkeit eines öffentlichen Schlüssels zu einer Identität wie einer Person oder Organisation. Dies wird durch Zertifizierungsbehörden (Certification Authority, CA) wie z. B. VeriSign oder Thawte gewährleistet, die die Korrektheit der Daten prüfen und das Zertifikat mit einer digitalen Signatur versehen.

Zertifikate, auch die der Zertifizierungsbehörden selbst, werden in Repositorys gespeichert. Zertifikate mit der Signatur einer vertrauenswürdigen CA gelten dann automatisch auch als vertrauenswürdig [Polk99,Schn95,ITU].

Unter Verwendung des öffentlichen Schlüssels eines Zertifikates lassen sich Daten verschlüsseln, die sich nur mit dem dazugehörigen privaten Schlüssel wieder dechiffrieren lassen. Abbildung 1 zeigt die Felder eines Zertifikates.

Trust- und Keystores

Trust- bzw. Keystores sind persistente Datenstrukturen, die sowohl auf Clients als auch auf Servern existieren können. Truststores enthalten ein oder mehrere Zertifikate vertrauenswürdiger Organisationen und sind typischerweise passwortgeschützt. Keystores können auch ein oder mehrere Schlüsselpaare speichern, wozu auch der private Schlüssel gehört. Mit diesem lassen sich Nachrichten signieren und verschlüsselte Nachrichten dechiffrieren. Schlüsselpaare können optional neben dem Keystore-Passwort zusätzlich mit einem weiteren Passwort verschlüsselt werden.

SSL

Das SSL(Secure Socket Layer)-Protokoll wurde 1996 von Netscape entwickelt und in der Version 3.0 als Transport Layer Security (TLS) 1.0 von der IETF im RFC 2246 im Jahre 1999 standardisiert [AlDie99]. Unabhängig davon, ob SSL oder TLS eingesetzt wird, spricht man meist von SSL. Das Protokoll gewährleistet Datenintegrität, Vertraulichkeit und Endpunkt-Authentifizierung zwischen zwei Kommunikationspartnern [AlDie99,Tan02]. SSL/TLS kann mit jedem zuverlässigen Transportprotokoll, wie z.B. TCP/IP, transportiert werden und ist als Sicherheitsschicht für die verschiedensten Protokolle der Anwendungsschicht be-

version	
serial number	
signature	algorithm
issuer	**C**ountry
	Organisation
	Organisational**U**nit
vailidity	not before
	not after
subject	**C**ountry
	STate
	Land
	Organisation
	Organisational**U**nit
	Canonical**N**ame
subject public key	algorithm
	modulus
	public exponent
signature	value

Abb. 1: Felder eines X.509-Zertifikates

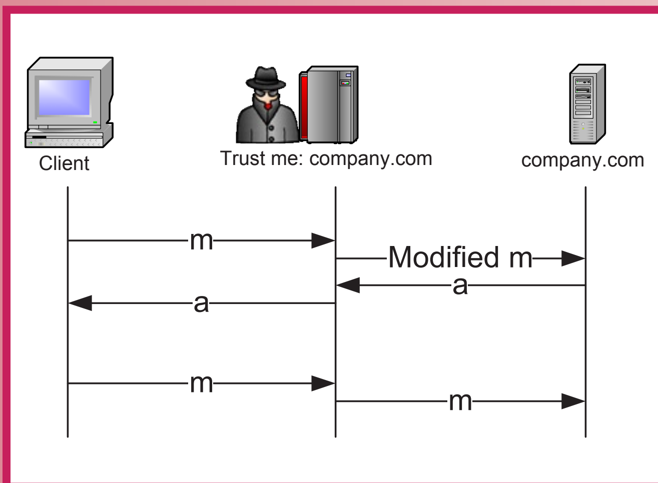

Abb 2a: Das Schema einer Man-in-the-Middle-Attacke

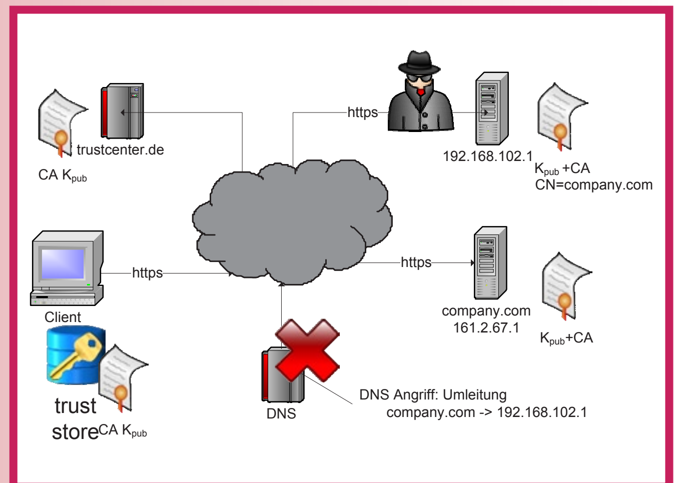

Abb 2b: und das dafür eingesetzte DNS-Spoofing

reits weit verbreitet, z. B. HTTPS, SSH, SMTPS, SFTP, IMAPS usw. [Resc00,Resc01].

Bei dem Verbindungsaufbau wird die Endpunkt-Authentifizierung durchgeführt, bestehend aus der Server- und Client-Authentifizierung. Bei der Server-Authentifizierung prüft der Client die Gültigkeit des Server-Zertifikates und die Übereinstimmung des angegebenen Domainnamens mit der angeforderten URL. Bei der Client-Authentifizierung überprüft der Server, ob ihm der öffentliche Schlüssel des Clients bekannt und vertrauenswürdig ist.

Angriffe

Es gibt eine große Anzahl möglicher Angriffe auf die Sicherheit eines Softwaresystems. Alle davon zielen jedoch darauf ab, unerlaubt an Informationen zu kommen oder die Funktion zu stören. Die wichtigsten davon sind das Abhören, die Man-in-the-Middle-Attacke und Denial-of-Service(DoS)-Angriffe.

Der naheliegendste Angriff auf ein System ist das Abhören der Kommunikation. Dabei hört ein Angreifer den ausgetauschten Datenverkehr unbemerkt ab und kommt in den Besitz von vertraulichen Informationen. Bei einer Man-in-the-Middle-Attacke, wie in Abbildung 2a dargestellt, platziert sich ein Angreifer zwischen Sender und Empfänger, nimmt eine Nachricht `m` vom Sender entgegen und leitet sie evtl. verändert als `modified m` an den eigentlichen Empfänger weiter [ViVils98,Tan02]. Die dazugehörige Antwort vom Server `a` kann unverändert weitergereicht werden. Eine Man-in-the-Middle-Attacke kann durch den Einsatz von DNS-Spoofing durchgeführt werden. Dabei wird die Namensauflösung des DNS Servers dahingehend manipuliert, dass die vom Client gesendeten Daten auf den Server des Angreifers umgeleitet werden [ViVils98,Tan02]. Abbildung 2b zeigt diesen Angriff. Unter einer Denial-of-Service-Attacke schließlich versteht man einen Angriff auf einen Dienst mit dem Ziel, diesen durch Überlastung lahm zu legen [Tan02].

Tabelle 1 zeigt, welche Kombination von Vorkehrungen getroffen werden muss, um die bereits geschilderten Angriffsarten abzuwehren.

Daher sollte eine gute Architektur die folgenden Anforderungen umsetzen:

- Gewährleistung von Datenintegrität, Vertraulichkeit und Endpunkt-Authentifizierung.
- Verhinderung von Man-in-the-Middle-Attacken.
- Abwehr von Denial-of-Service-Attacken.
- Verbindungsaufbau zum Server nur durch autorisierte Clients.

Eine Architektur, die diesen Anforderungen gerecht wird, zeigt Abbildung 3, die in den Grundzügen dem *Known-Partners*- und *Secure Channel*-Muster [Schu06] entspricht. Dabei liegen folgende Designprinzipien zugrunde: Durch den Einsatz von SSL zur Kommunikation zwischen Client und Server ist Datenintegrität und Vertraulichkeit gewährleistet. Um eine Man-in-the-Middle-Attacke zu verhindern, wird die Client-Software mit einem eigens konfigurierten Truststore ausgeliefert, der als einziges Zertifikat das Server-Zertifikat enthält. Dadurch vertraut der Client nicht einer bestimmten CA, sondern nur dem Server, der über den dazugehörigen privaten Schlüssel verfügt. Dahinter kann sich auch ein Lastverteiler verbergen, der die eingehenden Anfragen an eine Serverfarm im dahinterliegenden und gesicherten Firmennetz weiterleitet. Läuft das Serverzertifikat aus, muss rechtzeitig durch Softwareupdates das neue Zertifikat auf die Clients verteilt werden. Bei Online-Systemen wird in den meisten Fällen bei jedem Start die Aktualität des Clients überprüft und gegebenenfalls ein Update durchgeführt.

Um DoS-Attacken abzuwehren, kann Client-Authentifizierung beim Verbindungsaufbau eingesetzt werden, um Software-Authentifizierung durchzuführen. Dabei gibt es zwei mögliche Szenarien: Alle Clients verwenden das gleiche Schlüsselpaar oder jeder Client besitzt ein eigenes Schlüssel-

Architektur

Beim Design und der späteren Realisierung von IT-Systemen dürfen sicherheitsrelevante Anforderungen nicht vernachlässigt werden. Der Verlust von Geschäftsgeheimnissen, hohe Ausfallzeiten oder Datenmanipulation führen zu finanziellen Verlusten und können Firmenimages schädigen.

Angriff	Techniken zur Abwehr		
	Authentifizierung	Vertraulichkeit	Integrität
DoS	x		
Abhören		x	
Man-in-the-Middle	x	x	x

Tabelle 1: Angriffe und Techniken zur Abwehr

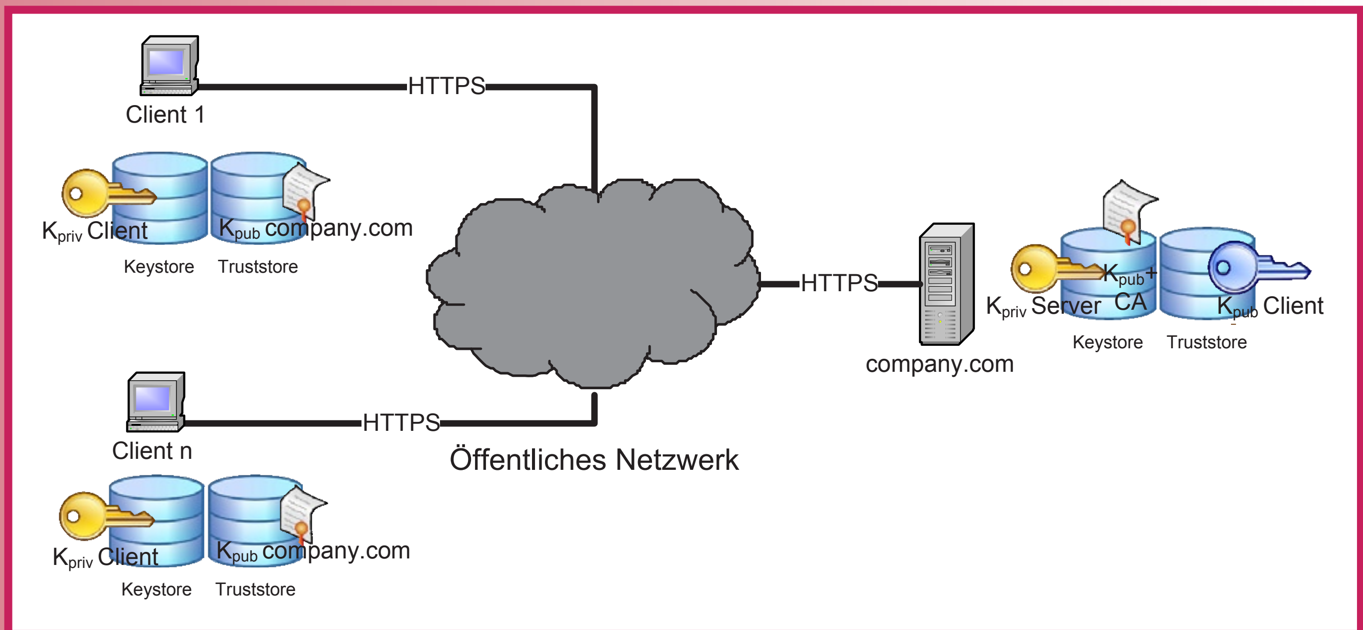

Abb. 3: Die Architektur einer gesicherten Webanwendung.

paar. Dieses Schlüsselpaar kann entweder vom Server vorgeneriert oder direkt auf dem Client erzeugt werden. Die Verwendung dedizierter Schlüsselpaare bietet höhere Sicherheit, da kompromittierte Schlüssel gesperrt werden können. Diese Technik zur Verhinderung von DoS-Attacken beruht darauf, dass der SSL-Handshake von spezieller Hardware erledigt werden kann und dadurch keine Systemressourcen auf dem Server verbraucht.

Besitzen alle Clients das gleiche Schlüsselpaar, enthält der Server den öffentlichen Schlüssel der Clients in seinem Truststore. Beim SSL-Verbindungsaufbau werden Clients abgewiesen, deren privater Schlüssel nicht zu den registrierten, öffentlichen Schlüsseln auf dem Server passt.

Allerdings hat dieses Design die Auswirkung, dass der private Schlüssel (das Geheimnis) Bestandteil der Client-Software ist. Erlangt ein Angreifer Zugriff auf einen Client, kann er den Schlüssel extrahieren. Dadurch sind wieder DoS-Attacken durchführbar. Allerdings gibt es auch hier geeignete Gegenmaßnahmen, die diese Lücke zwar nicht beseitigen, die Hürden jedoch sehr hoch stecken. Verfügt jeder Client über ein eigenes Schlüsselpaar, kann im Falle eines Angriffs genau dieser Schlüssel gesperrt werden. Darüber hinaus kommt beispielsweise Obfuscation in Betracht, um die Client-Distribution schwerer lesbar zu machen [Wiki]. Das Auffinden des passwortgesicherten Keystores ist dann erschwert. Ein weiteres Problem ist die Auslieferung des Keystore-Passworts zusammen mit der Software. Eine Lösung wäre, das Passwort durch den Benutzer eingeben zu lassen, und das Keystore-Passwort mit dem Benutzer-Passwort synchron zu halten. Weitere Sicherheit bietet ein Signieren der JAR-Archive in Verbindung mit der Validierung beim Programmstart gegenüber dem Server, um eine böswillige Änderung der Client-Software zu erkennen [ExFiles].

Eine weitere Hürde ist die eigentliche Verteilung der Client-Software. Ein Angreifer müsste zunächst in den Besitz der Client-Software gelangen. Um dies zu vermeiden, kann die Auslieferung der Software beispielsweise über eine gesicherte Website mit Passwort-Authentifizierung oder durch Postversand erfolgen.

Implementierung

Die vorgestellte Architektur lässt sich mit wenig Aufwand mit Hilfe von Java-Bordmitteln und Tomcat oder anderen gängigen Applikationsservern implementieren. Für die Entwicklung eines Testszenarios sollte zuerst ein funktionsfähiges Server-Backend vorhanden sein. Dann lässt sich ein Client erstellen. Für das Beispiel genügt es zunächst, Client und Server auf derselben Maschine (z. B. localhost) laufen zu lassen. Der im Folgenden dargestellte Quellcode zeigt keine Ausnahmebehandlung, was in produktivem Code jedoch notwendig ist.

Folgende Variablen werden im nachstehenden Text referenziert und können für das jeweilige Zielsystem angepasst werden:

```
$HOMEPATH       c:/Documents and Settings/user
$CATALINA_BASE  c:/apache-tomcat-5.5.23
$CLIENT_BASE    c:/projects/client
$JAVA_HOME      c:/Program Files/Java/jdk1.5.0_03
```

Serverseite

Die notwendige Konfiguration der Serverseite hängt vom eingesetzten Applikationsserver ab. Als Beispiel verwenden wir Apache Tomcat in der Version 5.5.23 [Tomcat].

Tomcat-Installation

Tomcat wird ohne vorgeneriertes Schlüsselpaar ausgeliefert. Daher muss zunächst ein privater und öffentlicher Schlüssel generiert werden. Dazu verwenden wir das Java-Werkzeug **keytool** (s. Listing 1 und 2), das im JDK enthalten ist.

```
keytool -genkey -alias tomcat -keystore $CATALINA_BASE/keystore
```

Listing 1: Erzeugen des Schlüsselpaares

```
keytool -list -keystore $CATALINA_BASE/keystore
```

Listing 2: Überprüfen des erzeugten Schlüsselpaares

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443 -->
  <Connector port="8443" maxHttpHeaderSize="8192"
  maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
  enableLookups="false" disableUploadTimeout="true"
  acceptCount="100" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS" />
```

Listing 3: Tomcat HTTPS Connector

Anschließend muss der HTTPS-Connector in der Tomcat-Konfigurationsdatei **conf/server.xml** aktiviert werden (s. Listing 3). Tomcat erwartet den Keystore zunächst in der Datei **$HOMEPATH/.keystore**. Dieser Pfad kann durch die Angabe von:

```
keystoreFile="$CATALINA_BASE\conf\keystore"
keystorePass="changeit"
```

im Tag **Connector** der **server.xml** geändert werden.

Nach erfolgreichem Serverstart kann man die HTTPS-Verbindung mit einem Browser und der Adresse **https://localhost:8443/** testen. Hat alles geklappt, erscheint eine Sicherheitsmeldung, dass die aufgerufene Seite nicht vertrauenswürdig ist. Hier sollte man die Option – sinngemäß, je nach Browser – *Accept this certificate temporarily for this session* auswählen. Anschließend erscheint die Warnung *Domain Name Mismatch*, da das erstellte Zertifikat im Feld **Common Name** nicht *localhost* enthält. Ignoriert man diese Warnung, erscheint die Tomcat-Startseite.

Tomcat Web-App

Als einfaches Beispiel verwenden wir im Folgenden die mit Tomcat mitgelieferte Web-App **HelloWorldExample**, die den Text **Hello World** als HTML-Code erzeugt. Die URL ist **https://localhost:8443/servlets-examples/servlet/HelloWorldExample**. Anstelle dieser einfachen Web-App kann aber genauso gut ein Webservice laufen.

Export des öffentlichen Schlüssels für den Client

Um einen Truststore für den Client zu erzeugen, der lediglich den öffentlichen Schlüssel des Servers beinhaltet, muss zunächst der Schlüssel exportiert werden (s. Listing 4). In einem zweiten Schritt wird der öffentliche Schlüssel in einen neuen Truststore importiert (s. Listing 5).

```
keytool -export -alias tomcat -file $CLIENT_BASE/tomcat-key.pub
  -keystore $CATALINA_BASE/keystore
```

Listing 4: Export des öffentlichen Schlüssels

```
keytool -import -alias tomcat
  -keystore $CLIENT_BASE/truststore -file $CLIENT_BASE/tomcat-key.pub
```

Listing 5: Import des öffentlichen Schlüssels in den Truststore

HTTPS-Verbindung

Die einfachste Möglichkeit, auf dem Client eine HTTPS-Verbindung aufzubauen, ist die Verwendung der Klasse **URLConnection**. Listing 6 zeigt ein Beispiel, in dem ein Client den Inhalt der lokalen URL **https://localhost:8443** ausliest. Auch wenn auf dem Client Kommunikationsframeworks eingesetzt werden, geht der eigentliche Kommunikationsvorgang auf **url.openStream()** zurück.

```
URL url = new URL("https", "localhost", 8443, "");
BufferedReader in = new BufferedReader(
  new InputStreamReader(url.openStream()));

String inputLine;
while ((inputLine = in.readLine()) != null)
  System.out.println(inputLine);

in.close();
```

Listing 6: Testen einer SSL-Verbindung

Die Java-Implementierung des SSL-Protokolls verwendet den mitgelieferten Trust- und Keystore aus dem JRE-Verzeichnis **$JAVA_HOME/jre/lib/security/cacerts**. Dieser Pfad kann innerhalb des Dateisystems durch Setzen von Java-Propertys geändert werden [Prop]. Standardmäßig sind sie nicht definiert:

```
javax.net.ssl.trustStore = c:/projects/client/client.ts
javax.net.ssl.keyStore = c:/projects/client/client.ks
```

Wird auf diese Weise auf die Web-App zugegriffen, entsteht zunächst eine **javax.net.ssl.SSLHandshakeException: unable to find valid certification path to requested target**, da weder das Zertifikat von einer CA signiert ist, noch das Zertifikat selbst im Truststore vorhanden ist. Der Server wird als nicht vertrauenswürdig eingestuft.

Modifikationen auf dem Client

In der hier vorgestellten Architektur werden das Server-Zertifikat und der private Schlüssel des Clients als Bestandteil der Client-Software ausgeliefert. Daher ist es notwendig, den Truststore aus einem JAR-Archiv zu lesen. Die Instantiierung des Java-SSL-Stacks muss deshalb vor erstmaliger Benutzung so geändert werden, dass der Keystore aus einer alternativen Quelle geladen wird. Dazu bedient man sich der Java-SSL-API.

Die Java-SSL-API bietet entsprechende Methoden an, die die Definition eines eigenen Key/Truststores – zumindest indirekt – ermöglichen. Die folgenden Aufgaben müssen dazu durchgeführt werden:

- Keystore initialisieren und Inhalt als Ressource laden.
- Keystore über TrustManager in SSLContext einfügen.
- Die DefaultSSLSocketFactory mit Hilfe des modifizierten SSLContextes definieren.
- Einen modifizierten HostnameVerifyer an die Klasse HttpsURLConnection übergeben.

Listing 7 zeigt deren Implementierung. Alle wichtigen Klassen, die von der Konfigurationsänderung betroffen sind, liegen in **javax.net.ssl.*** und **java.security.***. Es existieren Klassen mit gleichem Namen im Paket **com.sun.net.ssl.***, die aber Java-interne Klassen sind und keinesfalls verwendet werden sollten.

Das UML-Klassendiagramm in Abbildung 4 zeigt alle wichtigen Klassen, die für die Konfiguration der **SSLSocketFactory** notwendig sind. Die Klasse **TrustManagerFactory** wird mit einem **KeyStore** initialisiert und liefert mit **getTrustManagers(..)** einen Array vom Typ **X509TrustManager** zurück. Dieser **TrustManager** wird zur Initialisierung des **SSLContext** benötigt. In der Klasse **HTTPSUrlConnection** wird mit Hilfe der Klasse **SSLContext** die **SSLSocketFactory** und der **HostNameVerifier** gesetzt.

Listing 7 zeigt den notwendigen Code, um die Klasse **SSLSocketFactory** so zu konfigurieren, dass der Key- und Truststore über den aktuellen Classloader aus einem JAR-Archiv gelesen wird. Zunächst benötigt man eine Instanz der Klasse **KeyStore** vom Typ **JKS**. Dann besorgt man einen **InputStream**, der über den

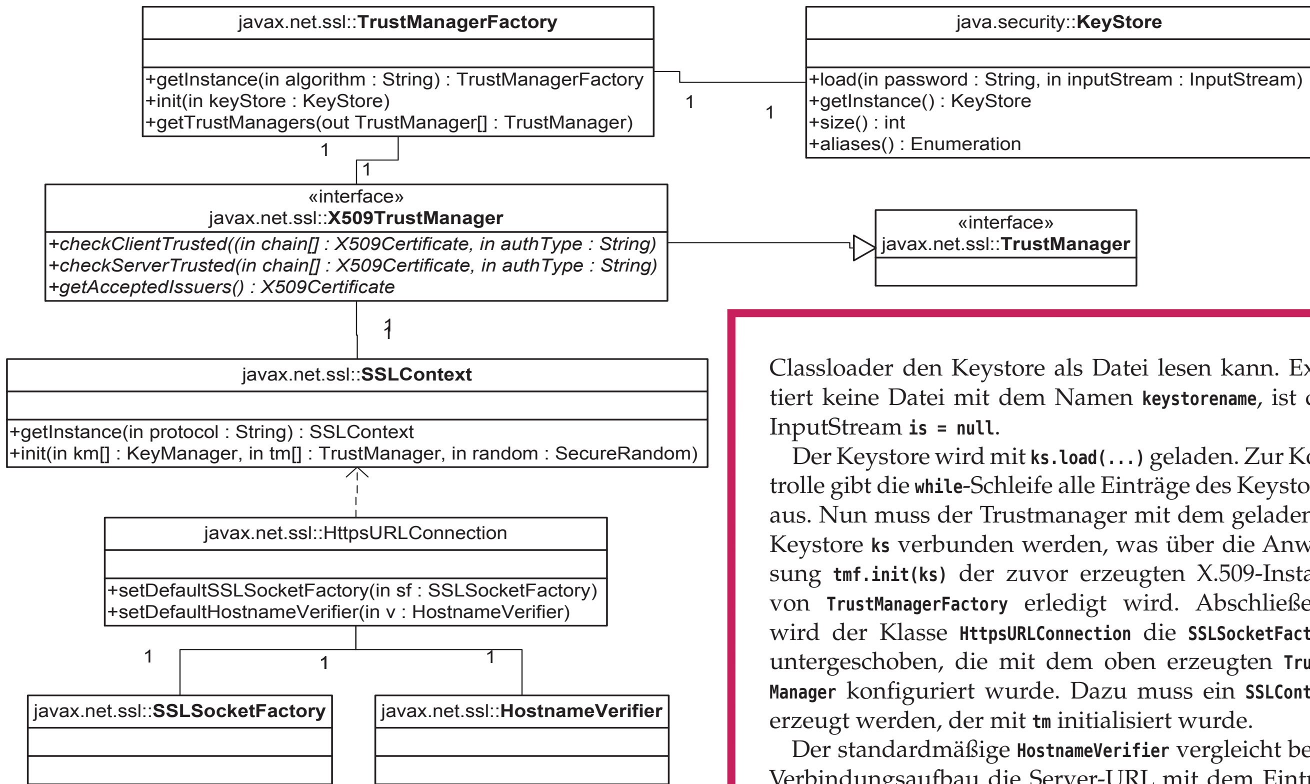

Abb. 4: Das Klassendiagramm der verwendeten Klassen

Classloader den Keystore als Datei lesen kann. Existiert keine Datei mit dem Namen `keystorename`, ist der InputStream `is = null`.

Der Keystore wird mit `ks.load(...)` geladen. Zur Kontrolle gibt die `while`-Schleife alle Einträge des Keystores aus. Nun muss der Trustmanager mit dem geladenen Keystore `ks` verbunden werden, was über die Anweisung `tmf.init(ks)` der zuvor erzeugten X.509-Instanz von `TrustManagerFactory` erledigt wird. Abschließend wird der Klasse `HttpsURLConnection` die `SSLSocketFactory` untergeschoben, die mit dem oben erzeugten `TrustManager` konfiguriert wurde. Dazu muss ein `SSLContext` erzeugt werden, der mit `tm` initialisiert wurde.

Der standardmäßige `HostnameVerifier` vergleicht beim Verbindungsaufbau die Server-URL mit dem Eintrag `CommonName` im Zertifikat, um eine Entscheidung über die Gültigkeit des Zertifikats zu treffen. Ein eigener `HostnameVerifier` kann wie in Listing 8 implementiert werden. Dabei können weiterführende Checks des Zertifikates durchgeführt werden. Denkbar ist beispielsweise eine Prüfung der zeitlichen Gültigkeit des Zertifikates oder des verwendeten Signatur-Algorithmus.

Verwendung von SSLSockets

Wie bereits gezeigt, existiert zum Aufbau von HTTPS-Verbindungen in der Java-API die Klasse `HttpsURLConnection`. Möchte man SSL direkt verwenden, kommt die Klasse `SSLSockets` zum Einsatz. Die Konfiguration findet auch hier wieder mit Hilfe der Klasse `SSLSocketFactory` wie in Listing 7 statt, jedoch muss eine eigene Implementierung dieser Klasse vorhanden sein. Der JRE

```
String keystorename = "truststore";
String password = "changeit";
...
try {
  // 1. Keystore initialisieren und Inhalt von Resource laden
  KeyStore ks = KeyStore.getInstance("JKS");
  InputStream is = this.getClass().getResourceAsStream(keystorename);
  if (is == null) {
    throw new IOException("Keystore " + keystorename + " not
    found.");
  }

  ks.load(is, password.toCharArray());

  System.out.println("Keystore contains " + ks.size() + " entries");
  Enumeration e = ks.aliases();
  while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
  }

  // 2. Keystore über TrustManager in SSLContext einfügen.
  TrustManagerFactory tmf =
    TrustManagerFactory.getInstance("SunX509");
  tmf.init(ks);
  TrustManager[] tm = tmf.getTrustManagers();

  SSLContext sc = SSLContext.getInstance("SSL");
  sc.init(null, tm, null);

  // 3. Die DefaultSSLSocketFactory mit Hilfe des
  // modifizierten SSLContextes definieren.
  javax.net.ssl.HttpsURLConnection.setDefaultSSLSocketFactory(
    sc.getSocketFactory());
} catch (...) {
...
}
```

Listing 7: Konfiguration der SSLSocketFactory

```
// 4. Einen modifizierten HostnameVerifyer an
// die Klasse HttpsURLConnection übergeben.
HostnameVerifier hv = new HostnameVerifier() {
  public boolean verify(String hostName, SSLSession session) {
    Date notAfter = session.getPeerCertificateChain()[0].
      getNotAfter();
    Date notBefore = session.getPeerCertificateChain()[0].
      getNotBefore();
    if (notAfter.before(new Date())) {
      return false; // certificate not accepted
    }
    if (session.getPeerHost().equals("127.0.0.1")) {
       return true; // certificate is ok
    }
    return false;
  }
};
javax.net.ssl.HttpsURLConnection.setDefaultHostnameVerifier(hv);
```

Listing 8: Eigener HostnameVerifier

wird diese Implementierung durch Anpassung der Eigenschaft `ssl.SocketFactory.provider` in der Security-Property-Datei `$JAVA_HOME/lib/security/java.security` bekannt gegeben.

Weiterführende Themen

Anstelle der Auslieferung des öffentlichen Schlüssels des Servers kann eine Organisation eine eigene Zertifizierungsbehörde einrichten. Daraus ergibt sich, dass jedes Server-Backend ein eigenes, von der firmeninternen CA signiertes Zertifikat besitzt. Der Vorteil dabei ist, dass ohne client-seitige Änderungen neue Server-Zertifikate verwendet werden können.

Zusammenfassung

Internetbasierte Webapplikationen kommunizieren über öffentliche Netzwerke. In ungeschützten Netzwerken gibt es jedoch die Bedrohung durch Datenspionage, -manipulation und Attacken auf Dienste, die finanzielle Verluste zur Folge haben können. Mit relativ einfachen Mitteln lässt sich das Sicherheitsniveau von Java-Anwendungen deutlich steigern und gegen diese Gefahren schützen.

Dieser Artikel hat Grundzüge einer für diese Umgebung geeigneten Sicherheitsarchitektur vorgestellt und aufgezeigt, wie diese in Java mit Hilfe von SSL umgesetzt werden kann.

Literatur und Links

[AllDie99] C. Allen, T. Dierks, Transport Layer Security, `http://www.ietf.org/rfc/rfc2246.txt`, Januar 1999, Internet Engineering Task Force (IETF), RFC 2818

[ExFiles] Sun Microsystems Inc., The Java Tutorials – Exchanging Files, `http://java.sun.com/docs/books/tutorial/security/toolfilex/index.html`, 2006

[ITU] ITU International Telecommunication Union, ITU-T X.509, `http://www.itu.int/rec/T-REC-X.509-199708-S/e`, August 1997

[Polk99] W. Polk, D. Solo, R. Housley, W. Ford, Internet X.509 Public Key Infrastructure Certificate and CRL Profile, `http://www.ietf.org/rfc/rfc2459.txt`, Januar 1999, Internet Engineering Task Force (IETF), RFC 2459

[Prop] Sun Microsystems Inc., The Java Tutorials – Properties, `http://java.sun.com/docs/books/tutorial/essential/environment/properties.html`, 2006

[Resc00] E. Rescorla, HTTP Over TLS, `http://www.ietf.org/rfc/rfc2818.txt`, Mai 2000, Internet Engineering Task Force (IETF), RFC 2818

[Resc01] E. Rescorla, SSL and TLS: designing and building secure systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001

[Schn95] B. Schneier, Applied cryptography (2nd ed.): protocols, algorithms, and source code in C, John Wiley & Sons, Inc., New York, NY, USA, 1995

[Schu06] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad, Security Patterns – Integrating Security and Systems Engineering, John Wiley & Sons Ltd., West Sussex, England, 2006

[Tan02] A. Tanenbaum, Computer Networks, Prentice Hall Professional Technical Reference, 2002

[Tomcat] The Apache Software Foundation, Apache Tomcat, `http://tomcat.apache.org/`, April 2007

[ViVils98] M. de Vivo, G. O. de Vivo, G. Isern, Internet security attacks at the basic levels, in: SIGOPS Oper. Syst. Rev., 32(2):4–15, 1998

[Wiki] Wikipedia, Obfuscation, `http://en.wikipedia.org/wiki/Obfuscation`

Christian Kücherer arbeitet seit 2004 als Softwareingenieur bei Zühlke in Eschborn. Er befasst sich seit über neun Jahren mit Java und hat seine Erfahrungsschwerpunkte in Enterprise-Architekturen, SOA und netzwerknahen Themen.
E-Mail: kue@zuehlke.com.