Seminararbeit: Programmierung von SSL-Verbindungen

Rainer Leisen, Bastian Ramm

17.April 2002

Zusammenfassung

Diese Seminararbeit behandelt das Thema “Programmierung von SSL-Verbindungen”. Es wird eine kurze Einführung in das SSL-Protokoll und die nötigen Hintergründe gegeben, gefolgt von einem Überblick über das Einsatzgebiet und die Programmierbibliotheken von SSL. Anhand einer einfachen Beispiel-Implementation von SSL in Java soll die Programmierung von SSL-Verbindungen dargestellt werden.

Inhaltsverzeichnis

Kapitel 1

Einführung in SSL

Dieses Kapitel gibt eine kurze Einführung in den Themen-komplex SSL. Die Grundlagen und Hintergründe werden kurz erläutert, damit das nötige Wissen vorhanden ist um SSL oder den Ablauf einer SSL Session zu verstehen und eine Implementierung nachvollziehen zu können.

1.1 Kurzbeschreibung von SSL

SSL steht für Secure Sockets Layer und ist ein Protokoll, das verschlüsselte Verbindungen über TCP/IP bereit stellt. Es wurde seit 1993 als propietäre Technik von Netscape entwickelt. Die aktuelle (und letzte) Version von SSL ist die Version 3.0. Nach der Version 3.0 wurde eine Version 3.1 mit minimalen Änderungen entwickelt. Diese Version ist jedoch nicht mehr Eigentum von Netscape, sondern wird als offener Internet Standard von der Internet Engineering Task Force (IETF) weiterentwickelt und gepflegt. Mit diesem Versionswechsel fand eine Umbenennung in TLS für Transport Layer Security statt. Der aktuellste Stand ist also TLS 3.1. TLS 3.1 ist abwärts-kompatibel zu SSL 3.0 und diese beiden Versionen sind abwärts-kompatibel zu SSL 2.0. Am verbreitetsten ist dennoch noch SSL 3.0, was hauptsächlich an den Implementierungen der grossen Browser und Server-Hersteller liegt. Ferner hat sich SSL als Bezeichnung durchgesetzt. Im folgenden wird nur noch die Rede von SSL sein, die Aussagen gelten in der Regel jedoch sowohl für SSL als auch für TLS. SSL zeichnet sich durch einige bestimmte Vorteile gegenüber anderen Lösungen wie IPSEC, S-HTTP, oder Kerberos aus. Es ist relativ einfach zu implementieren und bietet dennoch eine grosse Sicherheit. Der grösste Vorteil liegt darin, das beliebige Anwendungen auf SSL zurückgreifen können. SSL bildet eine spezielle Protokollschicht, die sich nur mit der Sicherung von Daten befasst. Diese Schicht ist oberhalb des TCP-Protokolls einzuordnen. Alle Anwendungen/Protokolle die über der TCP-Ebene liegen, wie HTTP, NNTP oder FTP, können also SSL nutzen, SSL ist in dieser Sicht nicht anwendungsspezifisch. Für nicht sicherheitskritische Kommunikation kann das SSL Protokoll aber auch ohne Probleme ausgelassen werden, was Vorteile für die Performance bringt.

SSL unterteilt sich selbst wieder in verschieden Schichten. Derer gibt es fünf: Application, Handshake, Alert, ChangeCipherSpec und Record Layer. In der Handshake Layer finden sich die Funktionen, mit denen SSL eine Session zwischen zwei Teilnehmern aushandelt. Mit der Alert Layer können Fehler oder Alarme behandelt werden. Application enthält die Behandlung der Nutzdaten der über dem SSL Protokoll liegenden Schichten, während ChangeCipherSpec nur ein Hilfsgerüst ist, das den Beginn verschlüsselter Kommunikation markiert. In der Record Layer zuguterletzt, liegt die Funktionalität mehrere SSL Messages in einem Frame zu kapseln.

1.2 Kryptographischer Hintergrund

Die Theorie hinter der Kryptographie und deren Prinzipien sind nicht Thema dieser Ausarbeitung. Es sei hier nur kurz erwähnt auf welche Theorien SSL sich stützt, um sie noch einmal ins Gedächtnis zu rufen. Wer sich mehr für die Hintergründe und das Thema Kryptographie interessiert, dem seien die folgenden Referenzen empfohlen:

Buchtipps

- Professional Java Security; J.Garm, D.Somerfield - wrox
- SSL and TLS Essentials - Securing the Web; Stephen Thomas - Wiley

1.2.1 Verschlüsselung

SSL bedient sich des Public-Key Verfahrens um die Verschlüsselung der Daten zu realisieren. Hierbei ist ein Anwender, der verschlüsselte Daten empfangen möchte, im Besitz zweier Schlüssel. Der erste ist sein privater Schlüssel, den er geheim hält und der dazu dient eine verschlüsselte Nachricht zu entschlüsseln. Der zweite ist ein öffentlicher Schlüssel, der jedermann zugänglich ist. Ein zweiter Anwender kann diesen Schlüssel nutzen um Daten zu verschlüsseln. Diese Daten lassen sich nur mit dem passenden privaten Schlüssel wider entschlüsseln. Damit ist dann sichergestellt, das nur der gedachte Empfänger die Daten zu Gesicht bekommt.Aus Performancegründen wird das Public Key Verfahren oftmals nur eingesetzt um einen Schlüssel für symmetrische Kryptographie auszutauschen. Hier müssen beide Anwender den Schlüssel kennen. Es besteht das Problem, diesen Schlüssel auszutauschen, ohne das ein Dritter in den Besitz dieses Schlüssels gelangen kann. Dieser Austausch wird durch das asymmetrische Verfahren mittels Public Key vorgenommen. Somit kann der Schlüssel sicher ausgetauscht werden. Die Verschlüsselung der eigentlichen Daten findet dann unter Verwendung des symmetrischen Schlüssels statt.

Unter SSL ist es möglich aus einer Reihe unterstützter Verschlüsselungsalgorithmen auszuwählen.

1.2.2 Authentifizierung

Ein weiteres Problem neben der eigentlichen Sicherung der Daten ist die Authentifizierung der kommunizierenden Teilnehmer. Ein Anwender muss sich sicher sein können, das der öffentliche Schlüssel auch wirklich dem anderen Kommunikationspartner gehört und nicht von einer dritten Partei vorgetäuscht wird. SSL vertraut hierbei auf Zertifikate und eine Trust Infrastructure. Ein Zertifikat identifiziert einen Kommunikationspartner. Es enthält einen mittels des privaten Schlüssels generierten Hashwert den man mit dem öffentlichen Schlüssel überprüfen kann.Ein Zertifikat wird von einer Certificate Authority (CA) ausgestellt, die dieses Zertifikat wiederum mit ihrem Zertifikat verifiziert. Da nicht jeder Nutzer jeder CA vertrauen muss, besteht eine Hierarchie aus CA´s, die sich gegenseitig verifizieren. Oberster Punkt dieser Hierarchie ist die Root Authority. Spätestens dieser Root Authority kann jeder Nutzer vertrauen.

1.3 SSL Basics

In SSL gibt es zwei Rollen, den Client und den Server. Client und Server kommunizieren auf SSL Ebene über SSL Messages. Der Client eröffnet die noch unverschlüsselte Session. Daraufhin handeln Server und Client die Parameter der Session miteinander aus, nachdem sie sich gegebenenfalls authentifiziert haben. Wenn sie sich einigen konnten, testen sie die Vereinbarungen und beginnen dann die verschlüsselte Kommunikation. Andernfalls wird die Verbindung wieder abgebrochen.Während einer aktiven Sitzung besteht die Möglichkeit, das sich Client und Server Alarme senden, wenn die Möglichkeit besteht, das eine sicher Kommunikation nicht mehr gewährleistet werden kann. Es gibt Alarme verschiedener Stufe, nicht alle führen zu einem sofortigen Abbruch der Verbindung.SSL erlaubt es eine Session ID zu vergeben. Mittels dieser lässt sich eine Session wieder aufnehmen, ohne das die Verbindungsparameter neu ausgehandelt werden müssen.Dies ist eine Komfortfunktion die die Effizienz steigern kann, jedoch mit Vorsicht zu geniessen ist. Unter bestimmten Umständen nimmt man hiermit einen Verlust von Sicherheit in Kauf.

1.4 SSL Messages

Wie bereits erwähnt kommunizieren der Client und der Server über Messages miteinander. Es gibt einen festgelegten Satz von Messages im SSL Protokoll. Client und Server teilen sich diesen Satz, wenngleich es einige Server- bzw. Client-spezifische Messages gibt. Es folgt eine Kurzübersicht der Messages in SSL:

Alert - Dient dazu die Gegenseite über mögliche Fehler oder Sicherheitsprobleme zu informieren. Es gibt verschieden Arten von Alerts. Einige davon sind “fatal” und beenden eine SSL Session immer.

ApplicationData - Beinhaltet die zu sichernden Daten

Certificate - Enthält das öffentliche Zertifikat des Absenders

CertificateRequest - Eine Message vom Server mit der ein Zertifikat vom Client angefordert wird (optional)

CertificateVerify - Beinhaltet einen Hashwert mit dem der Client den Besitz des zu seinem öffentlichen Zertifikat passenden privaten Schlüssels nachweist

ChangeCipherSpec - Markiert den Beginn der Nutzung der vereinbarten Sicherheitseinstellungen.

ClientHello - Beginnt eine SSL Session (Clientseitig), enthält Vorschlagsliste mit Parametern

ClientKeyExchange - Enthält Schlüssel des Clients

Finished - Beendet die Verhandlung der Parameter, und bedeutet den Beginn einer gesicherten Verbindung

HelloRequest - Beginnt (oder wiederholt) eine SSL Session (Serverseitig)

ServerHello - Antwort des Servers mit den ausgewählten Parametern

ServerHelloDone - Schliesst die serverseitige Bearbeitung eines Client Requests ab

ServerKeyExchange - Enthält Schlüssel des Servers

1.5 SSL Session

Eine SSL Session folgt einem bestimmten Muster. Es gibt einzelne Variation, je nachdem ob eine Authentifizierung des Clients durch SSL verlangt wird, oder eine vorherige Session wiederaufgenommen wird. Ersteres ist selten der Fall und letzteres ist aus Sicherheitsgründen wenig empfehlenswert. Folgend sei ein Beispiel für den Ablauf einer normale SSL Session gegeben:

Der Client beginnt die Session mit einem ClientHello. Darin enthalten ist eine Vorschlagsliste mit den vom Client unterstützten Parametern.

Der Server antwortet mit einem Block aus drei Messages. Der Block beginnt mit einem ServerHello. Diese enthält die aus der Vorschlagsliste ausgewählten Verbindungsparameter. Als nächstes sendet er in der Message ServerKeyExchange seinen öffentlichen Schlüssel. Danch bendet er den Block mit der Message ServerHelloDone.

Nun ist wiederum der Client am Zuge. Von hier an sind bereits alle Messages mit dem öffentlichen Schlüssel der Servers verschlüsselt. Er sendet seinerseits seinen öffentlichen Schlüssel in der Message Client KeyExchange. Mit der folgenden Message ChangeCipherSpec aktiveiert er die anfangs verhandelten Verschlüsselungsparameter für alle folgenden Messages die er senden wird. Er sendet noch eine nach diesen Parametern verschlüsselte Finished Message, damit der Server die Einstellungen überprüfen kann.

Zum Abschluss beginnt der Server mit der Message ChangeCipherSpec ebenfalls die Verschlüsselung nach den verhandelten Parametern für alle folgenden Messages. Er schickt dem Client eine Message Finished, damit auch dieser die Einstellungen überprüfen kann.

Einen störungsfreien Ablauf ohne Alerts vorausgesetzt, würden nach diesem Handshake nur noch Messages vom Typ ApplicationData versendet, die die verschlüsselten daten der über der SSL-Schicht liegenden Schichten enthalten.

Nicht behandelte Messages

Alert - Diese Message enthält einen Fehlercode aus einer Liste mit möglichen Alerts. Unter den für einen Alert gegebenen Umständen (z.B. Warnung über bevorstehende Beendigung der Session, inkompatible Verbindungsparameter) wird diese Message von einem der Kommunikationspartner abgesandt. Alerts bilden eine eigene Schicht im SSL Protokoll, sollen hier aber nicht detaillierter behandelt werden.

Certificate - Diese Message sendet der Server anstelle der Message ServerKeyExchange, wenn er über ein Zertifikat verfügt, mit dem er sich authentifizieren möchte. Verlangt der Server seinerseits ein Zertifikat, sendet der Client diese Message direkt als Antwort auf ServerHelloDone.

CertificateRequest - Wird vom Server nach der Message Certificate abgeschickt, wenn der vom Client ein Zertifikat verlangt

CertificateVerify - Hat der Client ein verlangtes Zertifikat an den Server geschickt, so bestätigt er mit dieser Message den Besitz des zugehörigen privaten Schlüssels nach der Message ClientKeyExchange.

HelloRequest - Wird nur sehr selten eingesetzt, beispielsweise, wenn der Server nach einer langen Zeit eine laufende SSL Session nicht mehr als sicher erachtet, fordert er hiermit den Client auf, eine neue Session zu initiieren.

1.5.1 Record Layer

Einzelne SSL Messages sind sehr klein haben also kein hohes Datenaufkommen. Deswegen gibt es in SSL einen Mechanismus, der es erlaubt mehrere Messages zu bündeln. Hierfür zuständig ist eine eigene Schicht innerhalb des SSL Protokolls. Es handelt sich um die sogenannte “Record Layer”. Diese Record Layer nimmt mehrere aufeinander folgende SSL Messages und packt sie in einen Frame. Die Gegenseite ist in der Lage aus diesem Frame wieder die einzelnen SSL Messages zu extrahieren. Unabhängig von Art und Anzahl der SSL Messages, findet die Record Layer immer Anwendung. Sie hat nur einen geringen Overhead, und da in einer normalen SSL Session eine Seite meist mehrere Messages hintereinander schickt, reduziert sie den Overhead den das darunter liegende Transport-Protokoll (TCP) bei vielen einzelnen Messages verursachen würde.

Kapitel 2

SSL in der Praxis

2.1 Einsatzgebiete

Die SSL Technologie wird meistens bei Anwendungen die über das Internet kommunizieren eingesetzt. Ein Einsatzgebiet ist aber auch die Verschlüsselung im Intranet, bei denen ein vertrauenswürdigen Kanal aufgebaut werden muß.

2.1.1 Beispiel HTTPS

Eines der bekanntesten Anwendung ist die Verschlüsselung von http Anfragen von einem Browser zu einem Webserver. Diese gesicherte Verbindung ist heutzutage ein muß - Kriterium für zum Beispiel Onlinebanking oder für das versenden von persönlichen Daten. Wie bei den meisten Client Server Modell, läuft die Kommunikation wie folgt ab:

1) Zu Beginn stellt der Client eine Anfrage an den Server und schickt ihm die Verschlüsselungsverfahren mit, die er unterstützt. Bei den aktuellen Versionen der Browser sind folgende Konfigurationen möglich:

- Netscape Version 6: SSL3 wird unterstützt
- Microsoft Explorer 5: SSL2, SSL3, TLS können aktiviert werden
- Opera Version 6: SSL2, SSL3, TLS können aktiviert werden

2) Der Server wählt ein Verfahren aus und übermittelt dem Client sein Zertifikat mit dem öffentlichen Schlüssel der Servers. Der Browser überprüft die Signatur des Zertifikates. Ist diese von eine der ihm bekannten Trust-Center signiert, so prüft es diese mit dem öffentlichen Schlüssel des jeweiligen Centers. Ist dem Client die Signatur nicht bekannt, wird der Anwender gefragt, ob er diese Dokument für vertrauenswürdig hält.

3) Danach werden alle http - Requests und Daten ohne das Protokoll zu Verändern von dem Client mit dem Public Key verschlüsselt und diese vom Server mit dem Private Key entschlüsselt. Diese Anfragen liegen dann beim Webserver in der bekannten Syntax vor und können beantwortet werden. Es müssen also keine Veränderungen am http - Protokoll vorgenommen werden.

4) Da die Verschlüsselung (siehe Abbildung) zwischen TCP und Anwendungsschicht realisiert wird, können viele Server Client Architekturen um dieses Feautere erweitert werden. Verschieden Produkte die SSL - Kommunikation anbieten: Borland Enterptice Server, Apache usw.

2.2 Programmierschnittstellen

Im diesen Abschnitt soll kurz darauf eingegangen werden, was es in den heutigen “gängigen” höheren Programmiersprachen wie C++ und .NET an Klassen und Methode zu Realisierung einer SSL Verbindung zu Verfügung gestellt werden.

2.2.1 C++

Borland bietet kein Paket zur Unterstüzung von SSL an. Im Internet findet man freie Implementiere Packte wie zum Beispiel SSLeay mit SSLv2 und SSLv3, die eine Reihe von Methoden anbieten.

Beispiel:

Enc - entschlüsseln

Dgst - verschlüsseln

Rsa, Dsa - Manipulieren der Schhlüssel

Dh, dsaparam - verändern der Diffie-Hellman und der DSA Parameter Files

X509 - Manipuliert das Zertifikat

Req - Zertifikat Anfrage

Genrsa - Generien eines privaten Schlüssels

Usw.

Doch sind diese Klassen nur unter Vorbehalt zu benutzen, da die man sich erst selbst um die Korrektheit der Sourcen kümmern muß. Zudem werden bei diesen Projekt nicht mehr kontinuierlich Sicherheitspatches bei bekannten Sicherheitslöchern zu Verfügung gestellt.

Dieses machen sich Firmen wie RSA BSAFE zu nutzen, die für SSL in C eine Kommerzielle Software anbieten und diese dann auch succesive Pflegen.

2.2.2 .NET

Microsoft bietet mit seiner neuen Programmierumgebung .Net auch fertige Klassen und Methoden in dem Securtiy-Package an, mit denen eine SSL Verbindung aufgebaut werden kann. Leider waren diese Klassen bei der Erstellungung diese Dokuments (Stand Mai 2002) noch nicht vollständig im Internet [?] dokumentiert und konnten so nicht mit hier Aufgenommen werden.

2.2.3 Java

Mit dem Paket JSSE bietet SUN [?] in Java Methoden an, die die Implementierungen von SSL erleichtern. Bei den Versionen 1.2 und 1.3 von Java 2 SDK, Standard Edition (J2SDK) war diese Paket noch optional. In der Version 1.4 J2SDK werden die in der API beschriebenen Funktionen direkt mitgeliefert.

Überblick:

- Unterstützt werde SSL Versionen 2.0 und 3.0, Implementierung von SSL Version 3.0
- Unterstützt werde TLS Version 1.0, Implementierung von TLS Version 1.0 Klassen und Methoden zum Aufbau eines Secure Channels (SSLSocket und SSLServerSocket)
- Unterstützung für das Hypertext Transfer Protocol (HTTP) mit SSL (HTTPS)

Mögliche Krytographische Funktionen

Cryptographic Algorithm	Cryptographic Process	Key Lengths (Bits)
RSA public key	Authentication and key agreement	2048 (authentication)
		2048 (key agreement)
		512 (key agreement)
RC4	Bulk encryption	128
		128 (40 effective)
DES	Bulk encryption	64 (56 effective)
		64 (40 effective)
Triple DES	Bulk encryption	192 (112 effective)
Diffie-Hellman public key	Key agreement	1024
		512
DSA	public key Authentication	2048

Auszg aus der API fuer javax.net.ssl:

Interface Summary

HandshakeCompletedListener: This interface is implemented by any class which wants to receive notifications about the completion of an SSL protocol handshake on a given SSL connection.

SSLSession: In SSL, sessions are used to describe an ongoing relationship between two entities.

SSLSessionBindingListener: This is interface is implemented by objects which want to know when they are being bound or unbound from a SSLSession.

SSLSessionContext: A SSLSessionContext is a grouping of SSLSessions associated with a single entity.

Class Summary

HandshakeCompletedEvent: This event indicates that an SSL handshake completed on a given SSL connection.

SSLServerSocket: This class is extended by server sockets which return connections which are protected using the Secure Sockets Layer (SSL) protocol, and which extend the SSLSocket class.

SSLServerSocketFactory: This class creates SSL server sockets.

SSLSessionBindingEvent: This event is communicated to a SSLSessionBindingListener whenever such a listener is bound to or unbound from a SSLSession value.

SSLSocket: SSLSocket is a class extended by sockets which support the "Secure Sockets Layer" (SSL) or IETF "Transport Layer Security" (TLS) protocols.

SSLSocketFactory Instances of this kind of socket factory return SSL sockets.

Exception Summary

SSLException: Indicates some kind of error detected by an SSL subsystem.

SSLHandshakeException: Indicates that the client and server could not negotiate the desired level of security.

SSLKeyException: Reports a bad SSL key.

SSLPeerUnverifiedException: Indicates that the peer’s identity has not been verified.

SSLProtocolException: Reports an error in the operation of the SSL protocol.

Auszug aus der API fuer javax.security.cert:

Class Summary

Certificate: Abstract class for managing a variety of identity certificates.

X509Certificate: Abstract class for X.509 v1 certificates.

Exception Summary

CertificateEncodingException: Certificate Encoding Exception.

CertificateException: This exception indicates one of a variety of certificate problems.

CertificateExpiredException: Certificate Expired Exception.

CertificateNotYetValidException: Certificate is not yet valid exception.

CertificateParsingException: Certificate Parsing Exception.

Kapitel 3

Beispiel-Implementation

3.1 Aufgabenstellung

Ziel ist es, einen Client sowie einen Server zu programmieren. Die Kommunikation zwischen Server und Client soll durch SSL verschlüsselt werden. Anhand des Quellcodes der beiden Programme soll die Programmierung beispielhaft erläutert werden. Als Programmiersprache wurde hier Java gewählt, da SSL bereits vollständig im Lieferumfang des JDK 1.4 enthalten ist.

Damit die Möglichkeit besteht, den SSL Handshake und die Verschlüsselung betrachten zu können, wird das Tool Ethereal eingesetzt. Ethereal ist in der Lage den gesamten Traffic auf einer Netzwerkschnittstelle zu protokollieren. Ferner ist es mit Ethereal möglich sich die Parameter und die Daten der Pakete bis ins Detail anzuschauen. Ethereal kann hierbei intelligent erkennen welches Protokoll gerade übertragen wird und auch Pakete die in anderen Protokollen gekapselt sind entschlüsseln.

3.2 JSSL-Server

Im folgenden sind an Quellcodeauszügen die wesentlichen Schritte zur Programmierung eines Servers mit SSL-Funktionalität erläutert. Die Präsentation des Programms ist Teil des Vortrags. Die vollständigen Quellen zu dem Programm finden sich im Anhang dieser Arbeit.

3.2.1 Pakete importieren

Zunächst müssen für unsere Beispielimplementation einige Pakete importiert werden:

- java.io.* - nicht direkt benötigt für SSL. Allerdings für Ein- und Ausgabeoperationen (z.B. Buffer) bei der Datenübertragung
- java.net.* - grundsätzliche Netzwerkfunktionen
- javax.net.ssl.* - SSL-spezifische Netzwerkfunktionen
- java.security.* - Sicherheitsspezifische Funktionen wie Schlüsselverwaltung

3.2.2 Schlüssel-/Zertifikatverwaltung

Um erfolgreich über SSL kommunizieren zu können, braucht der Server ein Schlüsselpaar und ein Zertifikat. Diese müssen zunächst generiert werden (näheres hierzu unter: 3.4.2 auf Seite 14). Diese sind dann für gewöhnlich in einer Datei abgelegt. Aus dieser wird die Information in eine Keystore-Klasse geladen. Solche Keystores werden in KeyManagern verwaltet, die man aus einer mit dem Keystore initialisierten KeyManagagerFactory erhält.

```
// zuvor erstellten Keystore aus Datei laden
```

```
KeyStore keystore =
KeyStore.getInstance(KeyStore.getDefaultType());
char[] passphrase = "passwort".toCharArray();
keystore.load(new FileInputStream("keystore"), passphrase);
// KeyManager-Liste mittels Default-KeyManagerFactory erstellen und
mit KeyStrore initialisieren
KeyManagerFactory kmf =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
kmf.init(keystore, passphrase);
KeyManager[] keyManagers = kmf.getKeyManagers();
```

3.2.3 SSLSocket erstellen

Die eigentliche Kommunikation läuft über eine Tochterklasse von ServerSocket, der Klasse SSLServerSocket. Um ein solches Socket zu generieren muss man zunächst einen SSLContext erstellen und mit den KeyManagern initialisieren. Der SSLContext liefert einem eine SSLServerSocketFactory, welche wiederum ein SSLServerSocket generiert. Bei der Erstellung des SSLServerSockets wird diesem noch ein Port zugewiesen (in diesem Beispiel 4711). Über dieses SSLServerSocket kann man dann eine Verbindung akzeptieren und eine Datenübertragung abwickeln.

```
   // SSLContext erstellen und mit KeyManager-Liste initialisieren
SSLContext context = SSLContext.getInstance("TLS");
context.init(keyManagers, null, null);
// aus SSLContext SSLServerSocketFactory erstellen und
SSLServerSocket generieren lassen
SSLServerSocketFactory ssf = (SSLServerSocketFactory)
context.getServerSocketFactory();
SSLServerSocket socket = (SSLServerSocket)
ssf.createServerSocket(SSL_PORT_SERVER);
// über SSLSocket auf eingehende Verbindung warten und akzeptieren
Socket acceptedSocket = socket.accept();
```

3.2.4 Beispielhafte Datenübertragung

In unserer Implementation empfängt der SSLServer Strings, sendet sie wieder zurück und gibt sie auf dem Bildschirm aus. Empfängt er den String “endetext”, so erkennt er dies als Zeichen, die Verbindung zu beenden. Die Kommunikation ist SSL-unspezifisch, das bedeutet, man bemerkt keine Interna des SSL-Protokolls. Um den Kommunikationsvorgang auf Ebene des SSL Protokolls betrachten zu können und insbesondere um sichtbar zu machen, das die Übertragung tatsächlich verschlüsselt ist, wird bei der Demonstration des Programms das Tool Ethereal einesetzt. Mehr hierzu unter: 3.4.5 auf Seite 15.

```
// Datenübertragung
DataInputStream in = new DataInputStream(acceptedSocket.getInputStream());
DataOutputStream out = new DataOutputStream(acceptedSocket.getOutputStream()
String received;
do{
   received = in.readUTF();
   System.out.print("\n\nReceived: "+ received);
   out.writeUTF("Confirming: " + received);
}while(! received.equals("endetext"));
```

3.2.5 Verbindung beenden

Nachdem der Server das Signal die Verbindung zu beenden erhalten hat, müssen nur noch die Buffer und das Socket geschlossen werden:

```
// Verbindungen beenden
in.close();
acceptedSocket.close();
socket.close();
System.out.print("\nConnection closed, now leaving SSLServer!");
System.out.print("\n\n");
```

3.3 JSSL-Client

Im folgenden sind an Quellcodeauszügen die wesentlichen Schritte zur Programmierung eines zu dem Server passenden Clients mit SSL-Funktionalität erläutert. Die Präsentation des Programms ist Teil des Vortrags. Die vollständigen Quellen zu dem Programm finden sich im Anhang dieser Arbeit.

3.3.1 Pakete importieren

Zunächst müssen für unsere Beispielimplementation einige Pakete importiert werden. Dieses sind dieselben, die auch der Server importiert:

- java.io.* - nicht direkt benötigt für SSL. Allerdings für Ein- und Ausgabeoperationen (z.B. Buffer) bei der Datenübertragung
- java.net.* - grundsätzliche Netzwerkfunktionen
- javax.net.ssl.* - SSL-spezifische Netzwerkfunktionen
- java.security.* - Sicherheitsspezifische Funktionen wie Schlüsselverwaltung

3.3.2 Zertifikatverwaltung/Trustmanager

Dem Client muss bekannt sein welchem Server bzw. welchem Zertifikat er vertrauen kann. Die vertrauenswürdigen Zertifikate werden auch hier in einem KeyStore gespeichert und in einem TrustManager verwaltet. Den TrustManager erhält man analog zum KeyManager aus einer mit dem KeyStore initialisierten TrustManagerFactory.

```
  // zuvor erstellten Keystore aus Datei laden
KeyStore keystore =
KeyStore.getInstance(KeyStore.getDefaultType());
char[] passphrase = "passwort".toCharArray();
keystore.load(new FileInputStream("keystore"), passphrase);
// TrustManager-Liste mittels Default-TrustManagerFactory erstellen
und mit KeyStrore initialisieren
TrustManagerFactory tmf =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init(keystore);
TrustManager[] trustManagers = tmf.getTrustManagers();
```

3.3.3 SSLSocket/Tunnel über Socket

Nun kann man daran gehen ein SSLSocket zu generieren. SSLSocket ist eine Tochterklasse von Socket, über die die Verbindung abgewickelt wird. Man benötigt wieder einen SSLContext, den man mit dem zuvor erstellten TrustManager initialisiert. Aus diesem generiert man eine SSLSocketFactory. Dieser SSLSocketFactory muss man ein normales Socket, den Servernamen und -port übergeben, um ein SSLSocket zu erhalten. Mit erfolgreichem Erstellen des SSLSockets ist die SSL-Verbindung bereits hergestellt.

```
   // SSLContext erstellen und mit TrustManager-Liste initialisieren
   SSLContext context = SSLContext.getInstance("TLS");
   context.init(null, trustManagers, null);
   // aus SSLContext SSLSocketFactory erstellen und SSLSocket (über Socket)
generieren lassen
   SSLSocketFactory ssf = (SSLSocketFactory) context.getSocketFactory();
   Socket defaultSocket = new Socket(SERVER_NAME, SSL_PORT);
   SSLSocket sslSocket = (SSLSocket) ssf.createSocket(defaultSocket,
SERVER_NAME, SSL_PORT, true);
   System.out.print("\nSSL Connection successfully established!");
```

3.3.4 Beispielhafte Datenübertragung

In unserer Implementation werden einige Strings an den Server übertragen. Als letztes wird der String “endetext” übertragen, der dem Server das Ende der Verbindung anzeigt. Die Kommunikation ist SSL-unspezifisch, das bedeutet, man bemerkt keine Interna des SSL-Protokolls. Um den Kommunikationsvorgang auf Ebene des SSL Protokolls betrachten zu können und insbesondere um sichtbar zu machen, das die Übertragung tatsächlich verschlüsselt ist, wird bei der Demonstration des Programms das Tool Ethereal einesetzt. Mehr hierzu unter: 3.4.5 auf Seite 15.

```
   // Datenübertragung
DataOutputStream out = new
DataOutputStream(sslSocket.getOutputStream());
DataInputStream in = new
DataInputStream(sslSocket.getInputStream());
String[] transmit = {"Das Pferd frisst keinen Gurkensalat.",
                     "Ein kleiner Schritt fuer SSL, aber ein grosser
Schritt fuer die TB.",
                     "Denken Sie, wie schön der Krieger, der die
Botschaft, die den Sieg, den die Athener bei Marathon, obwohl sie
in der Minderheit waren, nach Athen, das in grosser Sorge, ob es
die Perser nicht zerstören würden, schwebte, erfochten hatten,
verkündete, brachte, starb!",
                     "endetext"};
for(int i=0; i<transmit.length; i++){
   System.out.print("\n\nTransmitting: " + transmit[i]);
   out.writeUTF(transmit[i]);
   out.flush();
   System.out.print("\n --> Server answered: " + in.readUTF() );
   System.out.print("\n----------------------------------");
}//for
```

3.3.5 Verbindung beenden

Nach erfolgreicher Übertragung, bleibt nur noch die Buffer und die Sockets zu schliessen.

```
// Verbindungen beenden
out.close();
sslSocket.close();
defaultSocket.close();
System.out.print("\n\nConnection closed, now leaving SSLClient!");
System.out.print("\n\n");
```

3.4 Beispiel ausführen

3.4.1 Vorraussetzungen

Vorraussetzung für dieses Beispielprogramm ist das JDK 1.4 . Die Standard(sicherheits)einstellungen reichen aus um den Quellcode zu kompilieren und das Programm auszuführen.

3.4.2 Schlüsselpaare/Zertifikate erstellen

Mit dem Programm keytool, was bei der openssl Installation mit generiert wurde, kann man Zertifikate erstellen und verwalten. Für die Schlüssel Generierung muss das Programm mit dem Parameter “-genkey” aufgerufen werden. Danach werden die Parameter wie Passwort, Name, Organisation, usw die ein Zertifikat enthält, über die Kommandozeile abgefragt. Zudem stehen noch die Optionen wie zum Beispiel speichern unter einem selbst gewählten Namen oder die Auswahl der mathematischen Parameter zur Verfügung. Weitere Informationen findet man in der Manpages zu keytool.

3.4.3 Quellcode kompilieren und Programme starten

Ab dem JDK 1.4 sind keine besonderen Einstellungen und Parameter nötig um das Programm zu kompilieren. Es ist ausreichend die beiden *.java Dateien mit javac zu kompilieren. Der Servername (localhost) und Port (4711) sind in diesem Beispiel fest einkompiliert und müssten im Quellcode angepasst werden. Das Programm läuft auch wenn Server und Client auf verschiedenen Rechnern gestartet werden.

Man kann nach erfolgreichem kompilieren Server und Client ohne weiter Parameterübergabe von der Konsole aus starten (mittels java).

3.4.4 Bildschirmausgabe

Es folgt die Bildschirmausgabe die bei korrektem Programmablauf zu sehen sein sollte:

Bildschirmausgabe auf Server

```
Welcome to SSLServer!

SSL Connection successfully established!

Received: Das Pferd frisst keinen Gurkensalat.

Received: Ein kleiner Schritt fuer SSL, aber ein grosser Schritt fuer die
TB.

Received: Denken Sie, wie schön der Krieger, der die Botschaft, die den
Sieg, den die Athener bei Marathon, obwohl sie in der Minderheit waren,
```

```
nach Athen, das in grosser Sorge, ob es die Perser nicht zerstören würden,
schwebte, erfochten hatten, verkündete, brachte, starb!

Received: endetext
Connection closed, now leaving SSLServer!
```

Bildschirmausgabe auf Client

```
Welcome to SSLClient! SSL Connection successfully established!

Transmitting: Das Pferd frisst keinen Gurkensalat.
--> Server answered: Confirming: Das Pferd frisst keinen Gurkensalat.
---------------------------------

Transmitting: Ein kleiner Schritt fuer SSL, aber ein grosser Schritt fuer
die TB.
--> Server answered: Confirming: Ein kleiner Schritt fuer SSL, aber ein
grosser Schritt fuer die TB. ---------------------------------

Transmitting: Denken Sie, wie schön der Krieger, der die Botschaft, die den
Sieg, den die Athener bei Marathon, obwohl sie in der Minderheit waren,
nach Athen, das in grosser Sorge, ob es die Perser nicht zerstören würden,
schwebte, erfochten hatten, verkündete, brachte, starb! --> Server
answered: Confirming: Denken Sie, wie schön der Krieger, der die Botschaft,
die den Sieg, den die Athener bei Marathon, obwohl sie in der Minderheit
waren, nach Athen, das in grosser Sorge, ob es die Perser nicht zerstören
würden, schwebte, erfochten hatten, verkündete, brachte, starb!
---------------------------------

Transmitting: endetext
--> Server answered: Confirming: endetext
---------------------------------

Connection closed, now leaving SSLClient!
```

3.4.5 Paketanalyse in Ethereal

Ethereal ist ein Netzwerksniffer, der den gesamten Traffic auf den Netzwerkschnittstellen mithören kann. Bevor man Server und Client startet, startet man Ethereal. Man konfiguriert das Tool auf die gewünschte Netzwerkschnittstelle, in unserem Fall localhost. Danach start man das Mitschneiden. Nach erfolgreichem Ablauf von Client und Server, kann man das Mitschneiden beenden.

Daraufhin zeigt einenm Ethereal alle Pakete an die es mitgeschnitten hat. Zunächst kann man nur TCP-Pakete sehen, da SSL auf TCP aufsetzt. dementsprechen sind auch einige TCP-Pakete mitgeschnitten worden, die kein SSL beiinhalten, sondern reine TCP-Kommunikation.

Wählt man nun unter der Option “decode as...” SSL aus, so werden die TCP-Pakete, die SSL-Pakete enthalten aufgeklappt und man bekommt die SSL-Pakete angezeigt. So ist es nun möglich den SSL-Handshake nachträglich zu verfolgen und sich sogar die einzelnen SSL-Messages anzeigen zu lassen. In der SSL-Message “Application Data” kann man sich im Datenfeld die übertragenen Daten anschauen. Die Beispielsätze aus unserer Implementation sollten hierbei nicht zu erkennen sein, schliesslich sollen sie ja verschlüsselt sein.

3.5 Zusammenfassung

Die Programmierung einer SSL-Verbindung unter Java verhält sich ganz ähnlich wie die Programmierung einer normalen Verbindung über Sockets. Die Verbindung ans Laufen zu kriegen birgt zwar einige Hürden, aber die verschlüsselte Kommuniktion dann zu nutzen ist gewohnt einfach. Das SSL-Protokoll ist gekapselt und von dem Handshake und den eim Versand einzelner Messages oder anderen Protokoll-Interna bekommt der Anwender wenig bzw. gar nichts zu Gesicht. Demzufolge muss er sich wenig um das SSL-Protokoll kümmern und kann sich auf die Anwendung konzentrieren, die die Verbindung nutzen soll.

Hier wurden zwar nur die Standardeinstellungen verwendet, doch bietet Java komfortable Möglichkeiten die Sicherheitseinstellungen den eigenen Bedürfnissen anzupassen. Die Klasse des SSLSockets beinhaltet zahlreiche “set” Methoden um alle Einstellungen, wie etwa Verschlüsselungsalgorithmus, erforderliches Zertifikat, etc., die SSL bietet, auszuwählen.

Kapitel 4

Zusammenfassung

4.1 Fazit

Man wird heutzutage nicht nur bei Verbindungen im World Wide Web, sondern auch bei Diensten die von einem Server im Intranet angeboten werden, aus Sicherheitsgründen kaum noch an SSL-Verbindungen vorbei kommen. Sind diese Funktionen in den Programmen und Programmiersprachen implementiert, so können sie ohne viel Aufwand genutzt werden. So können auch vorhandene Programme die in Java programmiert worden sind, mit ein paar Zeilen mehr die SSL-Verbindung unterstützen.

4.2 Zukunft

In Zukunft werden alle Programmiersprachen und Programme, die Verbindungen über ein Netzwerk anbieten oder benötigen, diese Fetaure anbieten. Als Problematisch sehen wir das Schlüsselmanagement an. Den viele Anbieter scheuen die Registrierung der Schlüssel bei den zuständigen Zertifizierungsstellen. Die Erstellung eines eigenen Trustcenter bedeutet wider einen Mehraufwand, in folge wird der Schlüssel wohl auf einen nicht sicher Weg übermittelt.

Anhang A: Quellcode SSLServer

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

public class SSLServer{

    private static final int SSL_PORT_SERVER = 4711;

    public static void main(String[] args){
        try{
            System.out.print("\nWelcome to SSLServer!");

           // zuvor erstellten Keystore aus Datei laden
            KeyStore keystore =
KeyStore.getInstance(KeyStore.getDefaultType());
            char[] passphrase = "passwort".toCharArray();
            keystore.load(new FileInputStream("keystore"),
passphrase);

           // KeyManager-Liste mittels Default-KeyManagerFactory
erstellen und mit KeyStrore initialisieren
            KeyManagerFactory kmf =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
            kmf.init(keystore, passphrase);
            KeyManager[] keyManagers = kmf.getKeyManagers();

           // SSLContext erstellen und mit KeyManager-Liste
initialisieren
            SSLContext context = SSLContext.getInstance("TLS");
            context.init(keyManagers, null, null);

           // aus SSLContext SSLServerSocketFactory erstellen und
SSLServerSocket generieren lassen
            SSLServerSocketFactory ssf = (SSLServerSocketFactory)
context.getServerSocketFactory();
            SSLServerSocket socket = (SSLServerSocket)
ssf.createServerSocket(SSL_PORT_SERVER);

           // über SSLSocket auf eingehende Verbindung warten und
akzeptieren
            Socket acceptedSocket = socket.accept();
```

```
            System.out.print("\nSSL Connection successfully
established!");

            // Datenübertragung
            DataInputStream in = new
DataInputStream(acceptedSocket.getInputStream());
            DataOutputStream out = new
DataOutputStream(acceptedSocket.getOutputStream());
            String received;

            do{
                received = in.readUTF();
                System.out.print("\n\nReceived: "+ received);
                out.writeUTF("Confirming: " + received);
            }while(! received.equals("endetext"));

            // Verbindungen beenden
            in.close();
            acceptedSocket.close();
            socket.close();
            System.out.print("\nConnection closed, now leaving
SSLServer!");
            System.out.print("\n\n");
        }//try

        catch(Exception e){// Ausnahmebehandlung
            System.out.print("\n\nAn Exception occured!");
            System.out.print("\nException Class: "        +
e.getClass() );
            System.out.print("\nException Cause: "        +
e.getCause() );
            System.out.print("\nException Message: "      +
e.getMessage() );
             System.out.print("\nException Stacktrace: \n");
            e.printStackTrace();
            System.out.print("\nEnd of Stacktrace: \n\n");
        }//catch
    }// main()
}// class
```

Anhang B: Quellcode SSLCLient

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

public class SSLClient {

    private static final int SSL_PORT =4711;
    private static final String SERVER_NAME = "localhost";

    public static void main (String[] args){
        try{
                System.out.print("\nWelcome to SSLClient!");

                // zuvor erstellten Keystore aus Datei laden
                KeyStore keystore =
KeyStore.getInstance(KeyStore.getDefaultType());
                char[] passphrase = "passwort".toCharArray();
                keystore.load(new FileInputStream("keystore"),
passphrase);

                // TrustManager-Liste mittels
Default-TrustManagerFactory erstellen und mit KeyStrore
initialisieren
                TrustManagerFactory tmf =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                tmf.init(keystore);
                TrustManager[] trustManagers =
tmf.getTrustManagers();

                // SSLContext erstellen und mit TrustManager-Liste
initialisieren
                SSLContext context = SSLContext.getInstance("TLS");
                context.init(null, trustManagers, null);

                // aus SSLContext SSLSocketFactory erstellen und
SSLSocket (über Socket) generieren lassen
                SSLSocketFactory ssf = (SSLSocketFactory)
context.getSocketFactory();
                Socket defaultSocket = new Socket(SERVER_NAME,
SSL_PORT);
```

```
                SSLSocket sslSocket = (SSLSocket)
ssf.createSocket(defaultSocket, SERVER_NAME, SSL_PORT, true);

                System.out.print("\nSSL Connection successfully
established!");

                // Datenübertragung
                DataOutputStream out = new
DataOutputStream(sslSocket.getOutputStream());
                DataInputStream in = new
DataInputStream(sslSocket.getInputStream());
                String[] transmit = {"Das Pferd frisst keinen
Gurkensalat.",
                                     "Ein kleiner Schritt fuer SSL,
aber ein grosser Schritt fuer die TB.",
                                     "Denken Sie, wie schön der
Krieger, der die Botschaft, die den Sieg, den die Athener bei
Marathon, obwohl sie in der Minderheit waren, nach Athen, das in
grosser Sorge, ob es die Perser nicht zerstören würden, schwebte,
erfochten hatten, verkündete, brachte, starb!",
                                     "endetext"};

                for(int i=0; i<transmit.length; i++){
                  System.out.print("\n\nTransmitting: " +
transmit[i]);
                  out.writeUTF(transmit[i]);
                  out.flush();
                  System.out.print("\n --> Server answered: " +
in.readUTF() );
                  System.out.print("\n----------------------------------");
                }//for

                // Verbindungen beenden
                out.close();
                sslSocket.close();
                defaultSocket.close();

                System.out.print("\n\nConnection closed, now
leaving SSLClient!");
                System.out.print("\n\n");
        }//try

        catch(Exception e){// Ausnahmebehandlung
            System.out.print("\n\nAn Exception occured!");
            System.out.print("\nException Class: "        +
e.getClass() );
            System.out.print("\nException Cause: "        +
e.getCause() );
            System.out.print("\nException Message: "      +
e.getMessage() );
            System.out.print("\nException Stacktrace: \n");
            e.printStackTrace();
```

```
            System.out.print("\nEnd of Stacktrace: \n\n");
        }//catch
    }//main()
}// class
```

Literaturverzeichnis

[1] Professional Java Security; J.Garm, D.Somerfield; wrox; pp.261-305

[2] SSL and TLS Essentials - Securing the Web; Stephen Thomas; Wiley