Das TLS-Protokoll
Transport Layer Security

Version 1.0

Seminar: Internet Dienste SS 2004

Christoph Moll

02.07.2004

Inhaltsverzeichnis

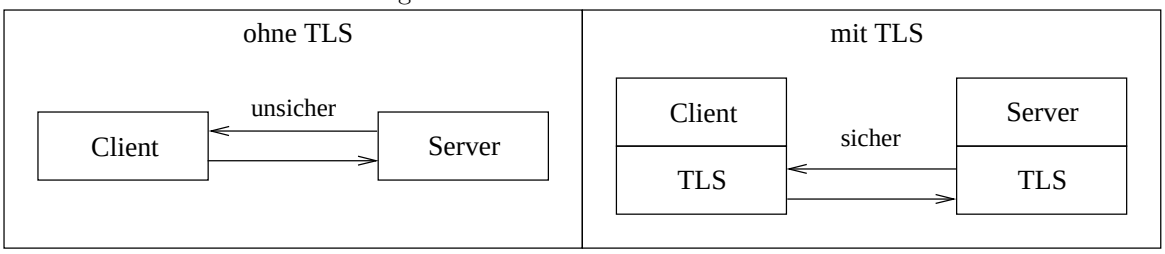
1 Einleitung

1.1 Allgemeines und Geschichte von TLS

Das TLS-Protokoll ist ein Angebot, für aktuelle und zukünftige Anwendungen/Protokolle, die bisher über einen unsicheren Kanal miteinander kommuniziert haben, diese Verbindung nun mit Hilfe von TLS privat aufzubauen. Das TLS-Protokoll wurde von einer Arbeitsgruppe der IETF (Internet Engineering Task Force) spezifiziert und 1999 publiziert [1]. Die IETF ist eine internationale, für jeden zugängliche Gruppe, die sich mit der Entwicklung und dem reibungslosen Betrieb des Internet beschäftig. Nähere Informationen zur IETF sind unter [2] zu finden.

TLS basiert auf dem SSL 3.0 Protocol (Secure Socket Layer), das von Netscape spezifiziert und veröffentlicht wurde. Der Name TLS läßt ein völlig neues Protokoll erwarten, doch die Veränderungen gegenüber SSL 3.0 sind so gering, dass sich diese Version von TLS intern als Version 3.1 zu erkennen gibt. Dennoch kann sich das TLS 1.0 Protokoll nicht direkt mit SSL 3.0 unterhalten (TLS enthält einen Mechanismus, sich auf SSL 3.0 zurückzustufen).

Abbildung 1: Client-Server-Kommunikation mit TLS



Über TLS können beliebige Anwendungen Daten austauschen. In neuen Anwendungen kann TLS mit relativ geringem Aufwand von Beginn an integriert werden. Insbesondere für ältere Protokolle müssen jedoch Ergänzungen definiert werden, die einen reibungslosen Einsatz von Applikationen mit und ohne Update gewährleisten. Für das HTTP/1.1 Protokoll ist so eine Erweiterung in [3] spezifiziert. Zur Entwicklung eigener Applikationen mit TLS stehen bereits einige Bibliotheken zur Verfügung, so z.B. die GNUTLS-Bibliothek [4] oder die OpenSSL-Bibliothek [5].

1.2 Ziele des TLS-Protokoll

Die folgenden Ziele, der Priorität nach aufgelistet, sollten mit der Entwicklung von TLS realisiert werden:

1. *kryptographische Sicherheit*: TLS soll dazu benutzt werden, zwischen zwei Parteien eine sichere Verbindung aufzubauen.
2. *Kompatibilität*: Es soll für unabhängige Programmierer möglich sein, Anwendungen zu entwickeln, die erfolgreich Verschlüsselungsparameter austauschen, ohne den genauen Code des anderen zu kennen.
3. *Erweiterbarkeit*: TLS versucht Rahmenbedingungen zu schaffen, in denen je nach Bedarf neue kryptographische Verfahren eingebettet werden können.

1.3 Aufbau und Eigenschaften von TLS

TLS bietet eine *private* wie auch *verlässliche* Verbindung zwischen zwei Anwendungen. Dazu wurde TLS in 2 Komponenten aufgeteilt, dem *TLS record protocol* und dem *TLS handshake protocol* (siehe Abb. 2).

Das TLS record protocol dient der Übertragung und dem Empfang aller Daten zwischen den Kommunikationspartnern gemäß vereinbarter Sicherheitsparameter. Die Sicherheitsparameter betreffen (De-)Komprimierung, MAC[1], Ver-/Entschlüsselung. Festzuhalten sind folgende Eigenschaften des re-

[1] nähere Informationen dazu in Appendix A

cord protocols:

- Die Verbindung ist *privat*, dafür wird ein symmetrisches Verschlüsselungsverfahren benutzt, bei dem mit demselben Schlüssel die Daten sowohl verschlüsselt wie auch entschlüsselt werden. Der Schlüssel wird für jede Verbindung einmalig mit einem anderen Protokoll bestimmt, z.B. dem TLS Handshake Protocol. Es ist allerdings auch möglich, das TLS Protokoll ohne Verschlüsselung zu benutzen.
- Die Verbindung ist *verlässlich*. Das Protokoll enthält ein Prüfverfahren, das die Daten auf ihre Integrität überprüft. Dazu wird ein verschlüsselter MAC (message authentication code) benutzt. Weitere Informationen über MACs befinden sich im Appendix A. Es ist aber auch möglich, das TLS-Protokoll ohne MAC zu benutzen.

Das TLS handshake protocol ist in erster Linie dazu da, die Sicherheitsparameter für das record protocol zu vereinbaren. Dazu bedient es sich asymetrischer Kryptograhpiemethoden (siehe Appendix C). Zusätzlich beinhaltet es noch ein Fehlerprotokoll. Das TLS Handshake Protocol unterützt folgende Eigenschaften:

- Die Identität eines Teilnehmers kann durch asymetrische bzw. "public key" Kryptographie geprüft werden. Diese Prüfung ist nicht notwendig, wird allerdings im allgemeinen für mindestens einen Teilnehmer gefordert.
- Die Vereinbarung des gemeinsamen Schlüssels ist sicher
- Die Vereinbarung ist verlässlich: Ein Angreifer kann die Vereinbarung nicht verändern, ohne von den beiden Parteien bemerkt zu werden.

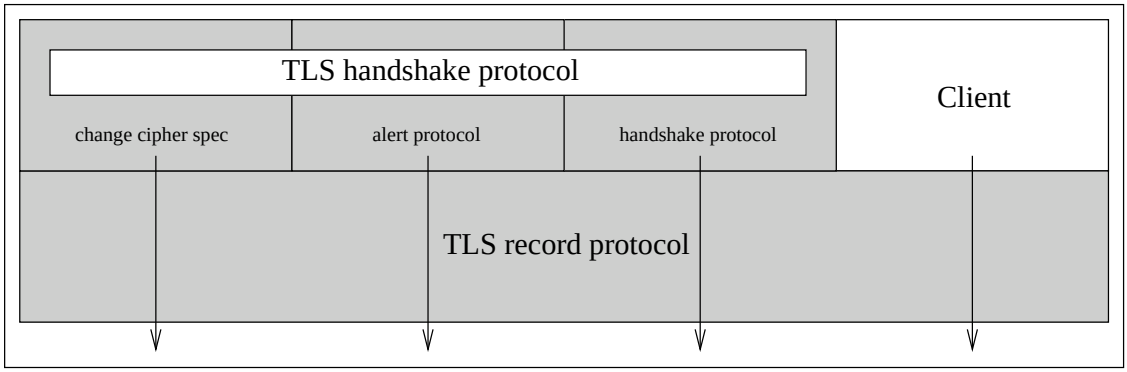


Abbildung 2: Schichten des TLS-Protokolls

1.4 Anmerkungen zur Beschreibungssprache

In Kapitel 2 wird der Aufbau und die Funktionsweise des record protocols auch mittels der in [2] benutzten Beschreibungssprache erläutert, um einen Eindruck zu vermitteln, wie das TLS Protokoll spezifiziert wurde. Die Implementierung des Protokolls bleibt dem Entwickler überlassen und wird nicht von der Beschreibungssprache festgelegt. Sie ist "C" sehr ähnlich, wobei die folgenden folgenden Besonderheiten erwähnt werden sollten:

- Die basic block size ist 1 Byte, multi-Byte-Werte werden in der normalen network byte order (oder big endian format) aneinandergehängt. Uninterpretierte Werte der Größe 1 Byte sind vom Typ opaque.
- Vektoren variabler Länge werden durch ihre minimale und maximale Länge definiert, genauer $T\ T\prime < min..max >$;

- Es gibt Strukturen, die sich zur Laufzeit verändern.

```
struct {
        T1 f1;
        ⋮
        select (E) {
                case e1: Te1;
                ⋮
                case en: Ten;
        };
};
```

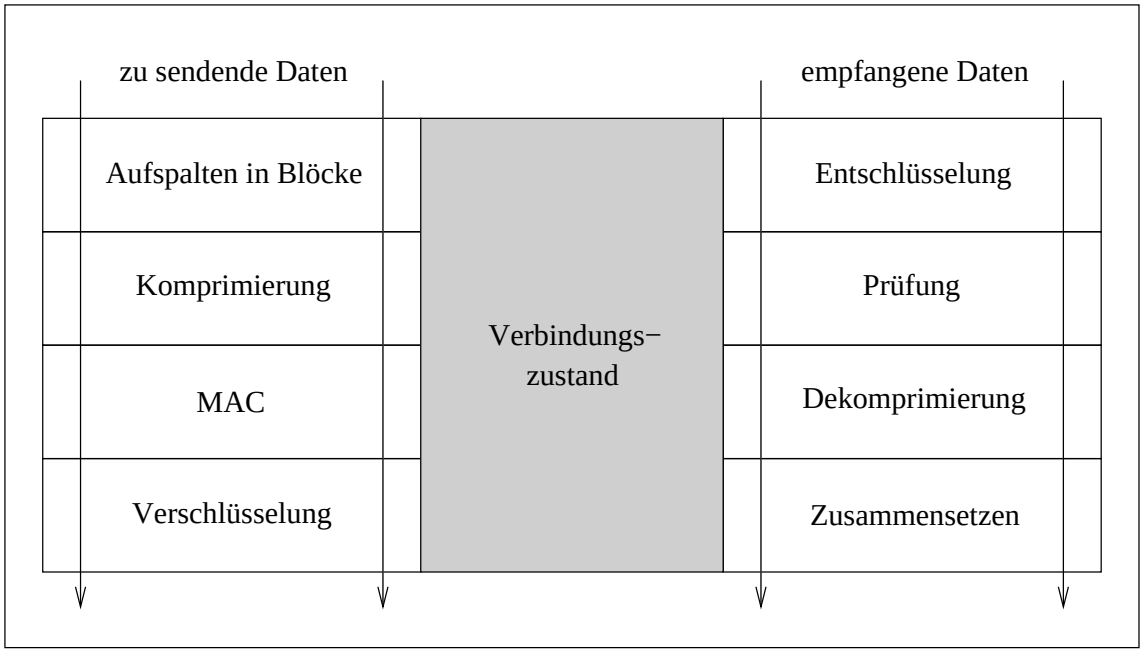
Die Implementierung solcher variabler Strukturen bleibt dem Programmierer überlassen.

2 Das TLS record protocol

2.1 Überblick

Das *TLS record protocol* dient einzig und allein dem Versenden und dem Empfang von Nachrichten. Um eine sichere und verlässliche Verbindung zu gewährleisten, muss eine Nachricht mehrere Stufen durchlaufen (siehe Abb. 3)

Abbildung 3: Stufen des record protocols



Die einzelnen Elemente in Abbildung 3 werden nun im folgenden näher erläutert, wobei der Datenfluss nur für zu versendene Nachrichten betrachtet wird, was aber keine große Einschränkung darstellt.

2.2 Verbindungszustand

Der *TLS Verbindungszustand* ist die *operierende Umgebung* des Protokolls. Natürlich müssen die aktuellen Lese- und Schreibzustände gespeichert werden, aber es muss auch möglich sein, zukünftige Lese- und Schreibzustände vorzubereiten. Daher müssen 4 Zustände gespeichert werden. Ein zukünftiger Zustand darf nur dann zu einem aktuellen Zustand gemacht werden, wenn alle seine Werte initialisiert wurden.

Die *Sicherheitsparameter* eines Zustandes werden durch die folgenden Werte abgebildet:

- enum {server, client} ConnectionEnd;
- enum {null, rc4, rc2, des, 3des, des40 } BulkCipherAlgorithm;
- enum {stream, block} CipherType;
- enum {true, false} IsExportable;
- enum {null, md5, sha} MACAlgorithm;
- enum { null(0), (255) } CompressionMethod;

und zu einer Struktur zusammengefasst:

```
struct {
    ConnectionEnd entity;
    BulkCipherAlgorithm bulk_ciper_algorithm;
    CipherType ciphertype;
    uint8 key_size;
    uint8 key_material_length;
    IsExportable is_exportable;
    MACAlgorithm mac_algorithm;
    uint8 hash_size;
    CompressionMethod compression_algorithm;
    opaque master_secret[48];
    opaque client_random[32];
    opaque server_random[32];
} SecurityParameters;
```

Die Werte dieser Struktur werden vom *TLS handshake protocol* vereinbart. Mit ihr besitzen wir alle nötigen Informationen, um die folgenden wichtigen Schlüssel zu berechnen.

Tabelle 1: benötigte Schlüssel

client write MAC secret	server write MAC secret
client write key	server write key
client write IV	server write IV

Die Berechnung dieser Schlüssel wird in Abschnitt 2.4 beschrieben. Die Bezeichnungen sind selbsterklärend, wobei die Initialisierungswerte (IVs) nur für Block Verschlüsselungsalgorithmen benötigt werden. Doch damit ist ein Verbindungszustand noch nicht ausreichend beschrieben. Zusätzlich beinhaltet er noch folgende weiteren Informationen:

- *compression state*: beinhaltet alle zur Kompression nötigen Informationen
- *cipher state*: Der aktuelle Zustand des Verschlüsselungsalgorithmus: Für Blockverschlüsselung beinhaltet es den Initialisierungswert und den vorgesehenen Schlüssel, für Stream-Verschlüsselung beinhaltet es alle Daten, die benötigt werden, um weiterhin Daten zu ent- bzw. verschlüsseln.
- *MAC secret*: Der MAC-Schlüssel für diese Verbindung
- *sequence number*: Jeder Verbindungszustand enthält eine sequence number $< 2^{64}$, diese wird für jeden bearbeiteten Datensatz um 1 erhöht. Somit kann sichergestellt werden, dass die Daten beim Empfänger wieder in der richtigen Reihenfolge zusammengesetzt werden können.

2.3 Record layer

Der *record layer* erhält uninterpretierte Daten vom *TLS handshake protocol* und von der TLS benutzenden Anwendung, und verarbeitet diese Daten gemäß Abb. 3.

2.3.1 Fragmentation

Diese Schicht fragmentiert die ankommenden Daten in Blöcke der Größe $\leq 2^{14}$ Bytes und speichert sie dann in der Struktur *TLSPlaintest*:

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

Dabei beinhaltet TLSPlaintext.type Informationen über welche Art von Daten es sich handelt, die übertragen werden. Dabei sind folgende Werte möglich:

```
enum {
    change_ciper_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;
```

In TSLPlaintext.version ist die implementierte Protokoll-Version gespeichert, bei TLS 1.0 würde es sich hierbei um Version 3.1 handeln. In TLSPlaintext.fragment befinden sich letztendlich noch die zu übertragenden Daten.

2.3.2 Record compression

Diese Stufe komprimiert die Daten nach dem im Verbindungszustand spezifizierten Verfahren. Die Struktur zur Speicherung der Daten bleibt im wesentlichen die gleiche wie in 2.3.1, nur dass diese nun komprimierte Daten beinhaltet. Auf die Komprimierung soll hier nicht näher eingegangen werden, da sie weder zu einer sicheren noch verlässlichen Verbindung, den Haupteigenschaften des Protokolls, beiträgt.

2.3.3 Record payload protection

Diese Schicht des *record protocols* sorgt dafür, dass die Verbindung sicher und verlässlich ist. Um die Verlässlichkeit zu gewährleisten, wird für die Daten ihr *MAC-Wert*[2] berechnet. Nach Anwendung des *MAC* werden die Daten in folgender Struktur gespeichert.

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (CipherSpec.cipher_type) {
        case stream: GenericStreamCipher;
        case block: GenericBlockCipher;
    } fragment;
} TLSCipherText;
```

Angenommen, wir würden einen Stream Verschlüsselungsalgorithmus benutzen, dann wäre fragment vom Typ GenericStreamCipher. Dieser Typ ist folgendermaßen definiert:

```
stream-ciphered struct {
    opaque content[TLSCompressed.length];
    opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

Die einzige Veränderung gegenüber unserer Struktur *TLSPlaintext* ist, dass wir nun den *MAC-Wert* hinzugefügt haben. Der MAC-Wert selbst berechnet sich nach folgender Formel:

```
HMAC_hash(MAC_write_secret, seq_num + TLSCypherText.type +
    TLSCypherText.version + TLSCypherText.length + TLSCypherText.fragment));
```

Nun müssen wir noch dafür sorgen, dass unsere Verbindung sicher ist, also die Daten verschlüsseln. Dazu wird die komplette Struktur gemäß unseres gewählten Verschlüsselungsverfahrens verschlüsselt.

Damit sind wir am Ende des record protocols. Die Daten haben alle Stufen durchlaufen und können nun an unseren Kommunikationspartner verschickt werden.

2.4 Schlüsselberechnung

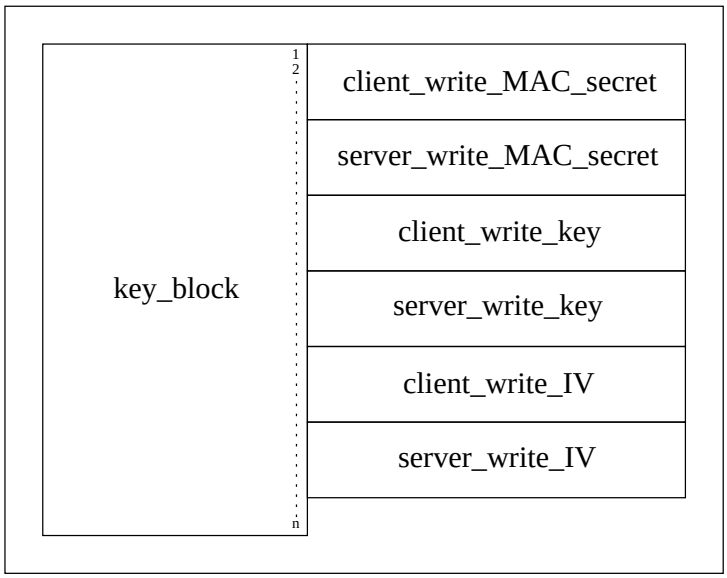
Was uns noch fehlt, ist eine Funktion zur Berechnung der Schlüssel aus Tabelle 1. Dazu benutzen wir die im Appendix D beschriebene *Pseudozufallsfunktion*. Wir berechnen

```
key_block = PRF(SecurityParameters.master_secret, "key expansion",
    SecurityParameters.server_random + SecurityParameters.client_random).
```

[2] nähere Informationen zu MACs und der von TLS verwendeten MAC-Funktion finden sich in Appendix A

Dies machen wir solange, bis wir für alle zu generierenden Schlüssel genügend Daten zusammen haben. Aus diesem Datenblock weisen wir nun verschiedene Bereiche unseren Schlüsseln gemäß Abbildung 4 zu.

Abbildung 4: Schlüsselberechnung - Blockaufteilung



3 Das TLS-Handshake-Protokoll

Das TLS-Handshake-Protokoll besteht aus 3 Unterprotokollen (vgl. Abb. 2). Diese dienen der Vereinbarung von Sicherheitsparametern, der Authentifizierung, der Instanzierung der Sicherheitsparameter und der Fehlerberichterstattung.

3.1 Change cipher spec protocol

Dieses Protokoll dient einzig und allein dem Zweck, Veränderungen der Schlüsselspezifikationen dem Gesprächspartner mitzuteilen. Die Nachricht besteht aus einem einzigen Byte mit dem Wert 1 und wird unter dem aktuellen Verbindungszustand verschickt. Die Nachricht wird sowohl vom Client wie auch vom Server verschickt, um anzuzeigen, dass die nun folgenden Daten mit den neu vereinbarten Sicherheitsparametern und Schlüsseln bearbeitet werden. Der Sender (Empfänger) macht nach dem Versand (Empfang) der Nachricht, den nachfolgenden, vollständig initialisierten Zustand zu seinem aktuellen Zustand.

3.2 Das Alarmprotokoll

Das Alarmprotokoll dient der Mitteilung von aufgetretenen Fehlern. Es wird hier nicht näher vorgestellt. Die genaue Spezifikation kann in RFC 2246 [1] nachgelesen werden.

3.3 Das Handshake-Protokoll

3.3.1 Überblick

Das Handshake-Protokoll legt die Sicherheitsparameter der Verbindung fest. Dafür benutzt es *public key* Verfahren, um gemeinsame Schlüssel zu generieren. Da der erste Verbindungszustand alle Sicherheitsparameter mit 0 initialisiert, steht der *"handshake"* i. a. ganz am Anfang jedes Gesprächs. Im einzelnen werden folgende Schritte ausgeführt:

1. Austausch von *"hello messages"* zur Festlegung des Algorithmus und zum Austausch von Zufallszahlen
2. Austausch der nötigen Sicherheitsparameter zur Erstellung eines *premaster Schlüssels*
3. Austausch von *Zertifikaten* und *Verschlüsselungsinformationen*
4. Erzeugung eines *master Schlüssels* und Austausch von neuen Zufallszahlen
5. Sicherheitsparameter im *record layer* ändern
6. Überprüfung, ob Client und Server dieselben Sicherheitsparameter benutzen und ob der "Handschlag" ohne Datenmanipulation durch einen Angreifer stattgefunden hat

Der Protokollablauf zwischen Client und Server kann Abb. 5 entnommen werden. Im folgenden wird ein solcher "handshake" einmal Schritt für Schritt nachvollzogen, wobei auf eine so ausführliche Beschreibung wie in Kapitel 2 verzichtet wird.

Anstatt neue Sicherheitsparameter mit dem Server zu vereinbaren, kann der Client auch versuchen, eine frühere Sitzung mit dem Server mit bereits bekannten Parametern wieder aufzunehmen. Es ist dem Server überlassen, ob er diese Anfrage akzeptiert oder nicht. Der Ablauf des Protokolls für diesen Spezialfall wird im folgenden aber nicht näher beschrieben.

3.3.2 Client hello

Diese Nachricht ist die erste Nachricht beim Aufsetzen einer neuen Verbindung. Sie beinhaltet die folgenden Attribute:

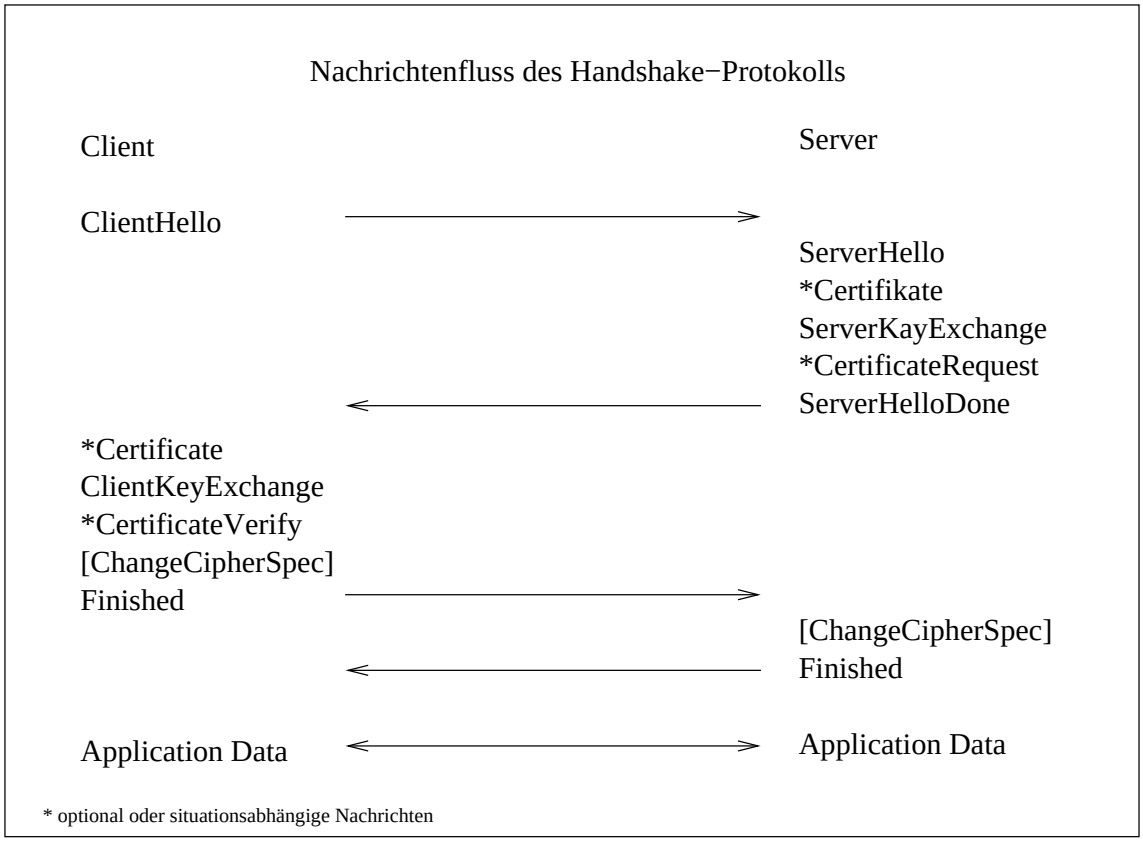


Abbildung 5: Protokollfluss - TLS Handshake

- Eine 28 Byte lange Zufallszahl und die aktuelle Zeit. Die Zufallszahl wird zur Signierung beim public key Verfahren und zur Berechnung des master-Schlüssels verwendet. Anmerkung: Diese TLS-Version überprüft nicht, ob die Uhrzeit richtig gesetzt ist.
- SessionID: Trägt für eine neue Verbindung den Wert Null, ansonsten den Wert der Verbindung, die der Client erneut aufsetzten will.
- Protokoll-Version: spezifiziert die Protokollversion, in der sich der Client in dieser Sitzung unterhalten will.
- eine Liste der gewollten und unterstützen Sicherheitsalgorithmen des Clients. Jedes Element der Liste definiert einen Schlüsselaustauschalgorithmus, einen Blockverschlüsselungsalgorithmus und einen MAC Algorithmus.
- Eine Liste der möglichen Komprimierungsverfahren. Diese Liste muss die Möglichkeit beinhalten, Daten nicht zu komprimieren.

Um im folgenden den Ablauf möglichst einfach zu halten, nehmen wir an, wir haben dem Server als einzige Möglichkeit des Schlüsselaustauschs einen anonymen *Diffie-Hellmann* Austausch (DH-Austausch, siehe Appendix C.1) vorgeschlagen. Dadurch entfallen alle *Certificate* messages. Allerdings wird bereits in der Spezifikation des TLS-Protokolls [1] darauf hingewiesen, dass dieser Schlüsselaustauschalgorithmus anfällig für *"Man-in-the-middle"* Attacken ist.

3.3.3 Server hello

Diese Nachricht wird als Antwort auf ein *Client hello* gesendet und legt aus der Liste der vom Client gesendeten Sicherheitsalgorithmen und Komprimierungsverfahren ein Paar fest. Außerdem definiert sie

noch die Protokollversion sowie SessionID und beinhaltet eine vom Server generierte Zufallszahl, die sich natürlich von der des Client unterscheiden und unabhängig sein sollte.

In unserem Fall gehen wir also davon aus, dass der Server unserem anonymen DH-Austausch zugestimmt hat.

3.3.4 Server key exchange message

Diese Nachricht ist dazu da, dem Client die nötigen Informationen des *public key* Verfahrens zu übergeben, mit denen ein *premaster-Schlüssel* generiert werden kann. Zur Verifizierung wird zusätzlich mit einer Hash-Funktion eine Signatur erstellt und mitgeschickt. Für die Signatur werden der Hash-Funktion die vom Client und vom Server generierten Zufallszahlen und die Informationen für das *public key* Verfahren übergeben.

In unserem Fall erhalten wir also vom Server die zahlen p, g und y_{Server} (vlg. Appendix C.1), mit denen wir nun unseren Schlüssel gemäß

$$\text{premaster\_secret} = y_{Server}^{x_{Client}}$$

berechnen können, wobei x_{Client} eine von uns frei gewählte, geheime Zahl ist, sowie einen Hashwert zur Verifizierung.

3.3.5 Server hello done

Diese Nachricht bedeutet, dass der Server dem Client nun genügend Information zum Schlüsselaustausch zur Verfügung gestellt hat. Der Client kann nun mit dem Schlüsselaustausch fortfahren.

3.3.6 Client key exchange message

Mit dieser Nachricht wird der *premaster-Schlüssel* gesetzt. Der Client ist zu diesem Zeitpunkt bereits in Besitz des premaster Schlüssels[3]. Da unser Austauschverfahren DH ist, benötigt der Server von uns noch den Wert y_{Client}. Deshalb wird mit dieser Nachricht y_{Client} an den Server übermittelt.

Mit Hilfe des premaster Schlüssels können nun Client und Server den master Schlüssel berechnen:

```
master_secret = PRF(premaster_secret, "master secret",
    ClientHello.random + ServerHello.random)[0..47];
```

Der master Schlüssel ist immer 48 Bytes lang. Die Zahlen ClientHello.random und ServerHello.random sind die in den *hello messages* ausgetauschten Zufallszahlen.

Direkt nach dieser Nachricht wird mit der Nachricht *change cipher spec* der *record layer* angewiesen, die gerade vereinbarten Sicherheitseinstellungen zum aktuellen Zustand zu machen.

3.3.7 Finished

Mit dieser Nachricht wird der Erfolg des Schlüsselaustauschs und der Authentifizierung geprüft. Ihr geht immer eine *change cipher spec* Nachricht voraus. Die *finished* Nachricht ist also die erste Nachricht unter den neu vereinbarten Sicherheitsparametern. Deshalb muß der Empfänger die Korrektheit der Daten überprüfen.

Zur Verifizierung wird wieder die Pseudozufallsfunktion aus Appendix D benutzt. Mit ihrer Hilfe wird

```
verify = PRF(master_secret, finished_label, MD5(handshake_messages)
        + SHA-1(handshake_messages) )[0..11]
```

mit folgenden Argumenten berechnet:

- *finished_label*: Wird die Nachricht vom Client verschickt, wird finished_label durch "client finished" ersetzt, analog für den Server durch "server finished".

[3] Bei RSA-Austausch wird vom Client eine Zufallszahl als premaster Schlüssel generiert

- *handshake_messages*: Das sind alle Daten, die bisher während des "handshakes" ausgetauscht wurden.

3.3.8 Hello request

Die *hello request* Nachricht gehört auch zum *handshake protocol*. Sie kann jederzeit vom Server verschickt werden und ist eine Aufforderung an den Client, einen *"handshake"* mit *client hello* einzuleiten. Somit können jederzeit während der Verbindung die Sicherheitsparameter verändert werden.

A MACs und HMACs

MACs (message authentication codes) dienen der Authentifizierung von Daten in Netzwerken. Dabei wird der gesendeten Nachricht ein Authentifizierungswert angehängt, mit dessen Hilfe der Empfänger feststellen kann, ob die Daten verändert wurden. Genauer: A will eine Nachricht N an B senden und beide besitzen einen gemeinsamen, nur Ihnen bekannten Schlüssel K. Dazu berechnet A anhand der Nachricht und des gemeinsamen Schlüssels einen Authentifizierungswert V = MAC(N,K). Nun verschickt A das Paar (N,V) an B. B kann damit prüfen, ob seine MAC-Funktion denselben Authentifizierungswert berechnet. Stimmen die Werte überein, hat er die Nachricht unverändert erhalten. Voraussetzung dafür ist, dass die MAC-Funktion so "Kollisionsfrei" wie möglich arbeitet, das bedeutet, es soll möglichst schwer sein, zu einem String x einen zweiten String $x\prime \neq x$ zu finden mit $MAC(x, K) = MAC(x\prime, K)$.

Bedient sich die MAC-Funktion kryptographischer Hash-Funktionen H, so bezeichnet man sie als HMAC-Funktion. TLS benutzt eine solche HMAC-Funktion. Ihre Definition lautet

HMAC = H(K XOR opad, H(K XOR ipad, Nachricht)),

wobei K der gemeinsame Schlüssel ist und mit

ipad = 64mal das Byte 0x36 wiederholt,
opad = 64mal das Byte 0x5C wiederholt

bezeichnet wird. Sie wurde RFC2104 [6] entnommen. Weitere Informationen und eine Sicherheitsanalyse befinden sich in [7].

B symmetrische Verschlüsselungsverfahren

Symmetrische Verschlüsselungsverfahren benutzen zum Ver- und Entschlüsseln denselben Schlüssel. Man kann hier zwischen block cipher und stream cipher unterscheiden.
block cipher: Sammelt Daten, bis er eine bestimmte Blockgröße beisammen hat. Dieser Block wird dann verschlüsselt. TLS hat folgende block cipher vorgesehen: RC2, DES, 3DES, DES40, IDEA.
stream cipher: Entschlüsselt ankommende Daten von gewünschter Größe (i.a. 1 Byte oder 1 Bit) direkt. Der Schlüssel wird üblicherweise nach jeder Benutzung mit einem Zufallszahlengenerator verändert. Der einzige im Protokoll unterstützte stream cipher ist RC4.

C asymetrische Verschlüsselungsverfahren

Asymetrische Verschlüsselungsverfahren werden auch public key Verfahren genannt. TLS benutzt diese Verfahren zur Generierung eines gemeinsamen Hauptschlüssels, mit dem dann die Daten symmetrisch verschlüsselt werden.

Public key Verfahren besitzen eine private Komponente, ohne deren Kenntnis es nahezu unmöglich ist, hinter das "Geheimnis" der Verbindung zu kommen.

TLS unterstützt 2 public key Verfahren. Das Verfahren von Diffie-Hellmann und das RSA-Verfahren.

C.1 Diffie-Hellmann

Das Verfahren von Diffie-Hellmann (benannt nach ihren Erfindern) war das erste public key Verfahren (1976). Die Schlüsselerzeugung läuft wie folgt ab:

1. Die Partner A und B einigen sich auf eine große Primzahl p, für die $(p-1)$ einen großen Primfaktor besitzt (optimal: $(p-1)/2$ ist Primzahl), und eine beliebige Zahl g mit $g < p$.

2. Beide Partner überlegen sich für sich Zahlen x_A bzw. x_B, den privaten Teil des Schlüssels, den sie streng geheim halten, und berechnen damit den Wert $y_A = g^{x_A} \ mod \ p$ bzw. $y_B = g^{x_B} \ mod \ p$. A teilt nun B den Wert y_A mit und umgekehrt.

3. A und B können nun den gemeinsamen Schlüssel berechnet. Dazu berechnen beide

$$K = y_B^{x_A} \ mod \ p = y_A^{x_B} \ mod \ p.$$

Der Schlüssel K ist bei beiden Partnern identisch und mit ihm kann nun symmetrisch verschlüsselt werden. Eine ausführliche Beschreibung ist in [8] zu finden.

C.2 RSA

Der RSA-Algorithmus wurde 1978 von Ron Rivest, Adi Shamir und Leonard Adleman erfunden. Die Vorgehensweise bei RSA ist folgende:

1. Suche zwei große Primzahlen P und Q.
2. Wähle eine Zahl E mit:
   - $1 < E < P * Q$
   - E hat mit $(P-1)(Q-1)$ keine gemeinsamen Primfaktoren

   E muss folglich ungerade sein.
3. Berechne nun eine Zahl D, so dass $(DE-1)$ ganzzahlig durch $(P-1)(Q-1)$ teilbar ist.

Der *public key* besteht nun aus dem Paar (PQ, E). Verschlüsselt wird der Text T durch

$$C = T^E \ mod \ PQ.$$

PQ wird deshalb auch oft mit *modulus*, E als *public exponent* bezeichnet. Der *private key* ist die Zahl D. Mit ihr kann die verschüsselte Nachricht wieder entschlüsselt werden:

$$T = C^D \ mod \ PQ.$$

D wird deshalb auch gerne als *private exponent* bezeichnet. Diese Beschreibung des RSA-Algorithmus wurde [9] entnommen.

D Pseudozufallsfunktion (PRF)

Die PRF wird vom TLS-Protokoll an mehreren Stellen benutzt. Um sie definieren zu können, benötigen wir zuerst eine Datenexpansionsfunktion P_hash(Schlüssel, Daten), die eine HMAC-Funktion (mit MD5 oder SHA-1) benützt, um aus dem Schlüssel und den Daten eine gewünschte Anzahl an Ausgabe zu erzeugen:

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ... ,
```

wobei mit + Aneinanderfügung bezeichnet wird und A() als

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1) )
```

rekursiv definiert wird. P_hash kann so oft iteriert werden, bis man die benötigte Anzahl an Ausgabe produziert hat.

Die PRF bekommt als Argumente einen Schüssel, eine Bezeichnung und einen "Samen", und erzeugt daraus mit Hilfe der Funktion P_hash eine Ausgabe beliebiger Länge. Um die PRF so sicher wie möglich zu machen, werden dazu zwei verschiedene Hash-Algorithmen so miteinander kombiniert, dass die PRF sicher sein sollte, falls beide Algorithmen sicher sind. Der Schlüssel wird in 2 Hälften S1 und S2 gespalten, wobei eventuell ein Byte geteilt wird. Mit S1 werden nun Daten mit Hilfe von P_MD5() generiert, analog mit S2 Daten mit P_SHA-1(). Die gewonnenen Daten werden dann mit exklusivem oder wieder miteinander verknüpft. Das Schema ist also folgendes:

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) XOR
                           P_SHA-1(S2, label + seed);
```

Eine ausführliche Beschreibung und Sicherheitsanalyse findet der Leser in [10].

Literatur

[1] The TLS-Protocol Version 1.0, T. Dierks, C. Allen - http://www.ietf.org/rfc2246.txt

[2] IETF - http://www.ietf.org

[3] Upgrading to TLS within HTTP/1.1, R. Khare, S. Lawrence - http://www.ietf.orf/rfc2817.txt

[4] GNU TLS-Bibliothek - http://www.gnu.org/software/gnutls/

[5] OpenSSL-Bibliothek - http://www.openssl.org

[6] HMAC: Keyed-Hashing for Message Authentication, H. Krawczyk, M. Bellare, R. Canetti - http://www.ietf.org/rfc/rfc2104.txt

[7] Keying Hash Functions for Message Authentication, H. Krawczyk, M. Bellare, R. Canetti - http://www.research.ibm.com/security/bck2.ps

[8] Kryptographie, Annegret Weng - http://www.mathematik.uni-mainz.de/∼ weng/skript.pdf

[9] The Mathematical Guts of RSA Encryption - http://world.std.com/∼ franl/crypto/rsa-guts.html

[10] Pseudorandom Functions Revisited: The Cascade Construction and its Concrete Security, H. Krawczyk, M. Bellare, R. Canetti - http://www.research.ibm.com/security/bck1.ps

Abbildungsverzeichnis