Universität Hamburg

Fachbereich Informatik

Entwurf vom
29. Juli 2015

Bachelorarbeit

Funktionsweise, Angriffe und Abwehrmechanismen von SSL/TLS

vorgelegt von

Tom Petersen

geb. am 13. Dezember 1990 in Hannover

Matrikelnummer 6359640

Studiengang Informatik

eingereicht am 29. Juli 2015

Betreuer: Dipl.-Inf. Ephraim Zimmer

Erstgutachter: Prof. Dr.-Ing. Hannes Federrath

Zweitgutachter: Dr. Dominik Herrmann

Aufgabenstellung

Die Protokollfamilie SSL/TLS umfasst Techniken zum Schutz von Kommunikationsdaten in IP-basierten Netzen. Ihre weite Verbreitung und Wichtigkeit für die IT-Sicherheit ist historisch gewachsen, und ihr Einsatz erstreckt sich über mittlerweile weit mehr Protokolle der Anwendungsschicht, als nur das ursprünglich anvisierte HTTP. Diese weite Verbreitung hat zwei wesentliche Konsequenzen. Zum einen wurden sowohl die Spezifikation der Protokollfamilie als auch praktische Implementierungen von SSL/TLS Gegenstand zahlreicher Angriffe. Zum zweiten sind ein grundlegendes Verständnis der Funktionsweise von SSL/TLS und der erwähnten Angriffe obligatorisch bei der Entwicklung und Implementierung von verteilter Software, Internetdiensten und Protokollimplementierungen auf der Anwendungsschicht, die mittels SSL/TLS abgesichert werden sollen.

In dieser Bachelorarbeit soll unter Einbeziehung aktueller Entwicklungen und Forschungsergebnisse die Funktionsweise von SSL/TLS, bedeutende Angriffe auf diese Protokollfamilie sowie daraus erarbeitete Anpassungen der Protokollspezifikation und Abwehrmechanismen erläutert und speziell für den Einsatz in der Hochschullehre aufbereitet werden. Darüber hinaus soll ein modular aufgebautes Tool zur Veranschaulichung der SSL/TLS-Funktionsweise sowie deren Angriffe und Abwehrmechanismen entwickelt und prototypisch umgesetzt werden. Der Fokus des Tools liegt in der Demonstration von SSL/TLS und dessen Schwächen mit beliebiger Verständnisvertiefung, sollte allerdings auch um weitere IT-Sicherheitsprotokolle erweiterbar sein.

Zusammenfassung

Abstract schreiben

Für den eiligen Leser sollen auf etwa einer halben, maximal einer Seite die wichtigsten Inhalte, Erkenntnisse, Neuerungen bzw. Ergebnisse der Arbeit beschrieben werden.

Durch eine solche Zusammenfassung (im engl. auch Abstract genannt) am Anfang der Arbeit wird die Arbeit deutlich aufgewertet. Hier sollte vermittelt werden, warum der Leser die Arbeit lesen sollte.

Inhaltsverzeichnis

1 Einführung

In diesem Exposé werde ich darauf eingehen, warum ich mich in meiner Bachelorarbeit gerne mit dem TLS-Protokoll befassen würde und eine kurze Einführung in das Thema geben. Zuerst werde ich kurz erklären, was mich dazu motiviert hat, mich mit dem Thema zu befassen, und in welche Richtung eine mögliche Bachelorarbeit gehen könnte.

Danach folgt eine Übersicht über die Funktionsweise des Protokolls und bisherige Angriffe gegen aktuelle und frühere Versionen des TLS- bzw. SSL-Protokolls.

Auf geeignete Literatur bzw. Veröffentlichungen wird an den entsprechenden Stellen der Ausarbeitung eingegangen.

1.1 Motivation

TLS ist das wohl am meisten genutzte Sicherheitsprotokoll im Internet. Aus diesem Grund wurde es im Laufe seiner Entwicklung oft untersucht und angegriffen. Dabei sind viele einfache und elegante Angriffe gefunden worden, die zeigen, wie wirksam die kleinsten Schwächen in Protokollen ausgenutzt werden können, und das auch Entscheidungen in scheinbar unbedenklichen Bereichen zu Sicherheitslücken führen können.

In den verschiedenen SSL- und TLS-Versionen wurden viele Änderungen vorgenommen, um diese Angriffe zu verhindern. Daher bietet TLS auch gute Beispiele für Dinge, die bei der Erstellung eines Protokolls bedacht werden müssen, und für wirksame Gegenmaßnahmen gegen bestimmte Angriffe.

1.2 Richtung der Bachelorarbeit

Eine Bachelorarbeit, die sich mit TLS befasst, könnte neben der grundsätzlichen Funktionsweise und einer Übersicht über bisherige Angriffe auf die Änderungen in TLS 1.3 eingehen. Für diese Version liegt ein Draft in 5. Version vom 9. März 2015 vor ([Res15]).

Auch ein konstruktiver Teil wäre denkbar, der sich mit der Überprüfung von TLS-gesicherten Servern oder der Implementation von Angriffen (oder verwundbaren Systemen) zum Beispiel für die Lehre befassen könnte.

2 SSL und TLS - ein Überblick

SSL (Secure Socket Layer) bzw. TLS[1] (Transport Layer Security) ist ein zustandsbehaftetes Protokoll, das (meist) auf dem TCP-Protokoll[2] der Transportschicht des TCP/IP-Protokollstapels aufbaut.

Hauptaufgaben von TLS sind Authentifikation der Kommunikationspartner, Verschlüsselung der Kommunikation sowie die Sicherstellung der Integrität der übertragenen Nachrichten ([Mey14]). Dazu läuft die Kommunikation über TLS in zwei Phasen ab: Zu Beginn wird eine sichere Verbindung durch Festlegung der verwendeten kryptographischen Verfahren und des Schlüsselmaterials hergestellt. Danach können Daten transparent für Anwendungen und auf TLS aufbauende Protokolle über diese Verbindung gesendet werden. Einige Beispiele für solche Protokolle und Anwendungen der Anwendungsschicht, die TLS nutzen, sind:

HTTPS für die Datenübertragung, zumeist für die Auslieferung von Webseiten genutzt.

FTPS für die Dateiübertragung.

SMTP für das Senden und Weiterleiten von E-Mails (als SMTPS oder per STARTTLS[3]).

IMAP für den Zugriff auf E-Mails auf Mailservern (als IMAPS oder per STARTTLS[3]).

POP3 für den Abruf von E-Mails von Mailservern (als POP3S oder per STARTTLS[3]).

OpenVPN, eine verbreitete VPN-Software.

SSL wurde von der Firma Netscape entwickelt und zuerst in ihrem Browser, dem Netscape Navigator, verwendet. Nach mehreren neuen Protokollversionen und nachdem es starke Verbreitung gefunden hatte, wurde es durch die IETF als TLS 1.0 standardisiert (TLS 1.0 entspricht SSL 3.1). Aktuell ist die TLS-Version 1.2 und an Version 1.3 wird gearbeitet.
Inzwischen ist TLS laut [Sch09] das „gegenwärtig meistverwendete Verschlüsselungsprotokoll im Internet“. Gründe hierfür sind dem Autor zufolge insbesondere die leichte Integrierbarkeit in bestehende Strukturen, die „[im Gegensatz zu IPSec] deutlich schnörkelloser[e] und einfacher[e]“ Protokollspezifikation und auch die marktreife Verfügbarkeit in den frühen 90er Jahren.

1. Im weiteren Verlauf dieser Arbeit wird der Einfachheit halber lediglich von TLS gesprochen. Bei etwaigen Unterschieden wird explizit auf diese eingegangen.
2. DTLS (Datagram Transport Layer Security) ist ein auf TLS basierendes Protokoll, dass auf UDP aufsetzt.
3. SMTPS/IMAPS/POP3S beginnen die TLS-Verbindung bereits direkt nach dem Verbindungsaufbau und laufen, um dieses Verhalten zu erzwingen, über einen anderen Serverport. STARTTLS ist ein Kommando, das nach Verbindungsaufbau gesendet werden kann, um eine TLS-Verbindung zu initiieren.

2.1 Implementierungen

Kurz sollen hier auch noch bestehende Implementierungen von SSL/TLS erwähnt werden, auch wenn der Fokus der Arbeit auf der Protokollspezifikation selber liegt.

Die verbreitetste Implementierung, die unter anderem auch im häufig verwendeten Apache-Webserver genutzt wird, ist OpenSSL, eine Open-Source-Implementierung in C. In Produkten von Microsoft wird die Bibliothek SChannel, in Apple-Anwendungen Secure Transport und in Google Chrome und Mozilla-Produkten NSS verwendet. Auch manche Programmiersprachen bringen eigene Implementierungen mit. Ein Beispiel hierfür ist die Java Secure Socket Extension(JSSE) in Java.

Zusätzlich gibt es viele weitere seltener genutzte Implementierungen wie GnuTLS, PolarSSL, LibreSSL oder Amazon s2n, die vollständig neu entwickelt wurden oder als Forks bestehender Implementierungen entstanden sind.

Viele Angriffe auf TLS-gesicherte Verbindungen, die bekannt geworden sind, waren Angriffe auf Implementierungen und nicht auf die Protokollspezifikation selbst (auf diese Angriffe wird in Kapitel 4 eingegangen). Ein Beispiel ist der Heartbleed[4] getaufte Bug in OpenSSL, der es wegen eines Programmierfehlers ermöglichte, Speicherinhalte des Servers auszulesen. Auf solche Angriffe, die lediglich einzelne Implementierungen betreffen, soll im Rahmen dieser Arbeit nicht weiter eingegangen werden.

Eine gelungene Übersicht über bestehende Implementierungen, die tiefer ins Detail geht, ist in [Mey14] zu finden.

4. http://heartbleed.com/

3 Funktionsweise und Teilprotokolle

Die Informationen in diesem Abschnitt stammen überwiegend aus der TLS 1.2-Spezifikation ([DR08]). Für einen ersten Überblick wurde [Eck13] genutzt.

Zum Einstieg grobe Funktionsbeschreibung, evtl. schon mit Grafik über Verbindungsaufbau?, Grafik der TLS-Protokolle

TLS besteht aus zwei Schichten. In der oberen Schicht sind vier Teilprotokolle spezifiziert: Handshake Protocol, Change Cipher Spec Protocol, Alert Protocol und Application Data Protocol, auf die später eingegangen wird. In der unteren Schicht befindet sich das Record Protocol, das die Daten von den Teilprotokollen der oberen Schicht entgegennimmt, verarbeitet und dann an tiefere Netzwerkschichten weitergibt. Eine Übersicht bietet Abbildung 3.1.

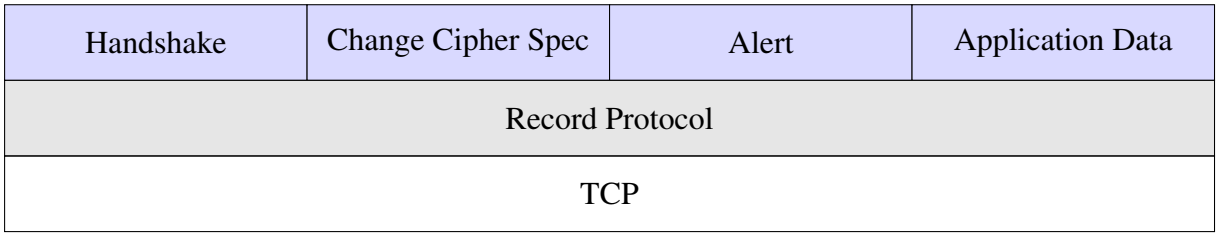

Abbildung 3.1: Überblick über die TLS-Protokollhierarchie

3.1 Record Protocol

Die zu sendenden Protokolldaten werden von dem Record Protocol fragmentiert (in maximal 2^{14} Byte große Pakete) und optional komprimiert. Danach wird je nach während des Handshakes verhandelten kryptographischen Funktionen (vgl. Abschnitt 3.3) die Integrität der Daten durch Berechnen und Anhängen eines MACs gesichert und die Nachricht danach zusammen mit dem MAC verschlüsselt[1]. Auf diese Schritte wird im Folgenden genauer eingegangen.

Kapitel evtl. hinter den Handshake? Und vorher auf Schlüsselberechnung eingehen?

Daten, die von einer höheren Schicht entgegengenommen werden, werden zu Beginn in ein **`TLSPlaintext`**-Objekt verpackt.

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

1. Diese Reihenfolge der Operationen wird MAC-then-Encrypt genannt. Laut [BN00] ist Encrypt-then-MAC vorzuziehen. In [Kra01] wird dieses Ergebnis bestätigt, aber auch die Sicherheit von MAC-then-Encrypt unter bestimmten Voraussetzungen gezeigt.
In [FSK10] fügen die Autoren hinzu, dass das Erkennen und Verwerfen ungültiger Nachrichten durch Encrypt-then-MAC vereinfacht wird. Auf der anderen Seite erwähnen sie aber auch Vorteile der MAC-then-Encrypt-Reihenfolge: Die Ein- und Ausgabe der MAC-Funktion sind einem Angreifer verborgen, so dass Angriffe gegen den MAC erschwert werden und die Integrität der Nachricht damit besser geschützt ist. Außerdem führen sie das Horton-Prinzip an (siehe [WS96]), wonach eine Authentifikation dessen, was gemeint ist und nicht was gesagt wird, stattfinden sollte. Bei der Erstellung eines Systems muss also abgewogen werden, welche Methode genutzt wird.

```
} TLSPlaintext;
```

Der **Contenttype** steht für den Protokolltyp der Nachricht: Change Cipher Spec Protocol (20), Alert Protocol (21), Handshake Protocol (22) oder Application Data Protocol (23). Die **ProtocolVersion** besteht aus zwei Bytes für die über- und untergeordnete Protokollnummer (z.B. (3,3) für TLS 1.2). Im **fragment** werden die zu übertragenden Daten gespeichert.

Danach werden die Daten optional durch den während des Handshakes vereinbarten Kompressionsalgorithmus komprimiert und in ein **TLSCompressed**-Objekt überführt.

```
struct {
   ContentType type;       /* same as TLSPlaintext.type */
   ProtocolVersion version;/* same as TLSPlaintext.version */
   uint16 length;
   opaque fragment[TLSCompressed.length];
} TLSCompressed;
```

Anschließend wird die Integrität der Daten mit einem MAC geschützt, der folgendermaßen berechnet wird:

```
MAC(MAC_write_key, seq_num +
                   TLSCompressed.type +
                   TLSCompressed.version +
                   TLSCompressed.length +
                   TLSCompressed.fragment);
```

Die fortlaufende Sequenznummer **seq_num** dient hierbei zur Verhinderung von Replay-Angriffen, also der erneuten Sendung von mitgelesenen Paketen durch einen Angreifer.
Bei den in TLS verwendeten Cipher-Suites wird das HMAC-Verfahren zur Berechnung des MACs genutzt. Details zu diesem Verfahren sind in [KBC97] zu finden. Die hierbei verwendete Hashfunktion wird in der Cipher-Suite angegeben.

Danach wird der Klartext zusammen mit dem MAC verschlüsselt, in ein **TLSCiphertext**-Objekt überführt und dann verschickt.

```
struct {
    ContentType type;       /* same as TLSCompressed.type */
    ProtocolVersion version;/* same as TLSCompressed.version */
    uint16 length;
    select (SecurityParameters.cipher_type) {
        case stream: GenericStreamCipher;
        case block:  GenericBlockCipher;
        case aead:   GenericAEADCipher;
    } fragment;
} TLSCiphertext;
```

Abhängig vom verwendeten Verschlüsselungsverfahren sehen diese Nachrichten unterschiedlich aus. Bei Stromchiffren wird der MAC zusammen mit den Daten (**TLSCompressed.fragment**) verschlüsselt und übertragen.
Bei Blockchiffren werden die Daten zusammen mit dem MAC zuerst mit Padding versehen, um ein Vielfaches der Blocklänge als Länge zu erhalten. Jedes Padding-Byte enthält die Paddinglänge als Wert. Dann werden MAC und Daten zusammen verschlüsselt und mit dem für jede Nachricht zufällig generierten Initialisierungsvektor (IV) versendet.

Bei der Nutzung von AEAD-Chiffren[2] werden bei der Verschlüsselung zusätzlich zum Klartext und Schlüssel zwei zusätzliche Parameter verwendet: eine sogenannte nonce (eine einmalig verwendete, zufällige Eingabe) und zusätzliche Daten, in die die Sequenznummer der Nachricht, ihr Typ, ihre Version und ihre Länge einfließen. Der explizite Teil der nonce wird neben den verschlüsselten Daten übertragen. Der implizite Teil wird durch server bzw. client write IV gebildet (vgl. Abschnitt 3.2). Die Notwendigkeit einer MAC-Berechnung entfällt in diesem Fall.

3.2 Berechnung des Schlüsselmaterials

Bei der Berechnung von Schlüsseln verwendet TLS eine eigene Konstruktion namens PRF (Pseudo Random Function), die standardmäßig für alle Cipher-Suites verwendet wird und auf dem HMAC-Verfahren aufbaut:

```
PRF(secret, label, seed) = P_hash(secret, label + seed)

P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
mit
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

Nach der ClientKeyExchange-Nachricht sind Client und Server im Besitz des pre-master-secret. Aus diesem und den in den Hello-Nachrichten übertragenen random-Werten wird nun auf beiden Seiten das master-secret folgendermaßen generiert:

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

Aus diesem master-secret werden je nach verwendeten kryptographischen Verfahren Schlüssel für die Erstellung des MACs, für die Verschlüsselung zwischen Client und Server und für eine zusätzliche Eingabe bei AEAD-Chiffren berechnet:

```
client write MAC key
server write MAC key
client write encryption key
server write encryption key
client write IV
server write IV
```

Dazu werden solange Schlüsselblöcke nach dem folgenden Verfahren erstellt, bis genug Daten vorhanden sind um alle benötigten Schlüssel konstruieren zu können:

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

2. Authenticated Encryption with Associated Data: Betriebsmodi für Blockchiffren, die ohne zusätzlichen MAC Authentizität und Integrität bereitstellen. Beispiele für solche Modi sind CCM (Counter with CBC-MAC) oder GCM (Galois/Counter Mode).

3.3 TLS-Handshake

Das Handshake Protocol dient zur Herstellung einer gesicherten Verbindung. Hierbei werden verwendete TLS-Version und kryptographische Verfahren zwischen den Kommunikationspartnern vereinbart, optional ihre Identitäten authentifiziert und ein gemeinsames Geheimnis (das sogenannte pre-master-secret) für die bereits beschriebene Generierung der während der eigentlichen Kommunikation verwendeten Schlüssel übertragen oder berechnet.

Eine Übersicht über die während eines vollständigen Handshakes ausgetauschten Nachrichten bietet Abbildung 3.2. Im Folgenden werden diese Nachrichten und ihr Aufbau im Detail betrachtet. Nachrichten, die - je nach gewünschten Eigenschaften der Verbindung - optional gesendet werden können, sind mit einem Stern(*) gekennzeichnet.

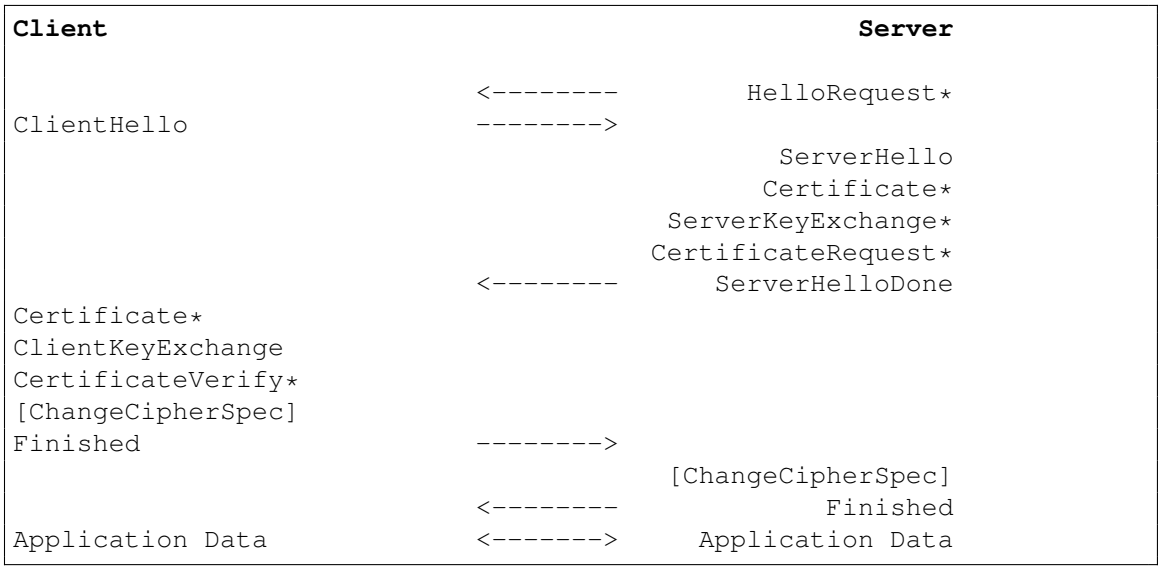

Abbildung 3.2: Nachrichtenverlauf beim vollständigen TLS-Handshake. Entnommen aus [DR08].

HelloRequest*

```
struct { } HelloRequest;
```

Diese Nachricht kann vom Server gesendet werden, wenn ein neuer Handshake gewünscht wird (beispielsweise um Schlüssel und kryptographische Funktionen während einer Verbindung neu auszuhandeln).

ClientHello

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
      case false:
```

```
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Hierbei enthält **client_version** die neueste vom Client unterstützte TLS-/SSL-Version (z.B. 3.3 für TLS 1.2). **random** besteht aus einem 4-Byte großen Zeitstempel (UNIX-Format) und 28 zufälligen Bytes. Die **session_id** dient zur Identifikation einer Sitzung. Sie ist bei dem ersten Handshake leer und kann später dazu verwendet werden, bestehende Sitzungen wieder aufzunehmen (vgl. Abschnitt 3.6). Die Cipher-Suite-Liste enthält alle vom Client unterstützten Cipher-Suites in Reihenfolge seiner Präferenz. Ebenso wird eine Liste von unterstützten Kompressionsalgorithmen übertragen. Optional kann auch eine Liste von gewünschten TLS-Extensions angegeben werden (vgl. 3.8).

ServerHello

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
      case false:
          struct {};
      case true:
          Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

In **server_version** steht die höchste Version, die Server und Client unterstützen und die damit für die Kommunikation verwendet wird. **random** besteht äquivalent zur ClientHello-Nachricht aus einem 4-Byte Zeitstempel und 28 zufälligen Bytes. Die **session_id** enthält entweder eine neu generierte ID, die ID einer wieder aufgenommen Sitzung oder kann auch leer sein, um anzugeben, dass die Sitzung nicht wieder aufgenommen werden kann. In **cipher_suite** und **compression_method** überträgt der Server die von ihm aus den vom Client übertragenen Listen ausgewählte Cipher-Suite bzw. den Kompressionsalgorithmus. In der Extensionliste gibt der Server alle vom Client gewünschten Extensions an, die er unterstützt.

ServerCertificate*

```
struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

In dieser Nachricht sendet der Server seine Zertifikatskette zur Überprüfung seiner Identität. Das erste Zertifikat in der Liste bildet das Serverzertifikat, folgende Zertifikate müssen das jeweils vorhergehende zertifizieren. Der im Zertifikat enthaltene Public-Key muss zum ausgehandelten Schlüsselaustausch-Algorithmus passen. Wenn nicht anders ausgehandelt wird für die Zertifikate das X.509v3-Format verwendet.

RFC als Quelle

ServerKeyExchange*

```
struct {
    select (KeyExchangeAlgorithm) {
      case dh_anon:
          ServerDHParams params;
      case dhe_dss:
      case dhe_rsa:
          ServerDHParams params;
          digitally-signed struct {
              opaque client_random[32];
              opaque server_random[32];
              ServerDHParams params;
          } signed_params;
      case rsa:
      case dh_dss:
      case dh_rsa:
          struct {} ; /* message is omitted for rsa, dh_dss, and
             dh_rsa */
    };
} ServerKeyExchange;
```

Diese Nachricht wird nur für bestimmte Schlüsselaustausch-Verfahren gesendet, wenn die ServerCertificate-Nachricht nicht genügend Informationen zum Austausch des pre-master-secret bietet (vgl. ClientKeyExchange-Nachricht).

`ServerDHParams` enthält dabei die öffentlichen Diffie-Hellman-Parameter p (die prime Ordnung der gewählten Gruppe), g (einen Erzeuger der Gruppe) und den öffentlichen Schlüssel $Y_s = g^{X_s} \mod p$.

Im Fall von nicht anonymen Diffie-Hellman-Schlüsselaustausch werden diese Parameter mit gewähltem asymmetrischen Verfahren (DSS oder RSA) signiert.

CertificateRequest*

```
struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    SignatureAndHashAlgorithm supported_signature_algorithms<2^16-1>;
    DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

TLS unterstützt optionale Clientauthentifizierung. Mit dieser Nachricht kann der Client vom Server aufgefordert werden ebenfalls ein Zertifikat zu senden. Die **`certificate_types`**-Liste enthält alle Zertifikatarten (z.B. Zertifikat mit RSA-Schlüssel, ...), die vom Server unterstützt werden, und **`supported_signature_algorithms`** die unterstützten Signaturalgorithmen. In **`certificate_authorities`** kann eine Liste von erwarteten CAs übertragen werden.

ServerHelloDone

```
struct { } ServerHelloDone;
```

Mit dieser Nachricht signalisiert der Server das Ende des ServerHello und zugehöriger Nachrichten.

ClientCertificate*

Wenn von dem Server eine Clientauthentifizierung gefordert wurde, kann der Client in dieser Nachricht seine Zertifikatskette senden. Das Format entspricht dem der ServerCertificate-Nachricht.

ClientKeyExchange

```
struct {
    select (KeyExchangeAlgorithm) {
      case rsa:
          EncryptedPreMasterSecret;
      case dhe_dss:
      case dhe_rsa:
      case dh_dss:
      case dh_rsa:
      case dh_anon:
          ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

Wenn RSA als Schlüsselaustausch-Algorithmus vereinbart wurde, so wird das vom Client generierte pre-master-secret mit dem Public Key des Servers verschlüsselt und gesendet. Es besteht aus der größten vom Client unterstützten Protokollversion (2 Bytes), um Version-Rollback-Angriffe zu verhindern (siehe Abschnitt 4.1), und 46 zufällig generierten Bytes.

```
struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

Wenn der Schlüsselaustausch per Diffie-Hellman-Verfahren geschieht und der Public-Key des Client nicht in seinem Zertifikat enthalten ist (ClientCertificate-Nachricht, entspricht dem implicit-Fall), sendet der Client in dieser Nachricht seinen öffentlichen DH-Schlüssel $Y_c = g^{X_c} \mod p$.

```
struct {
    select (PublicValueEncoding) {
      case implicit: struct { };
      case explicit: opaque dh_Yc<1..2^16-1>;
    } dh_public;
} ClientDiffieHellmanPublic;
```

Das pre-master-secret wird dann als $Z = (Y_c)^{X_s} \mod p$ auf der Serverseite bzw. $Z = (Y_s)^{X_c} \mod p$ auf der Clientseite berechnet.

CertificateVerify*

```
struct {
    digitally-signed struct {
       opaque handshake_messages[handshake_messages_length];
    }
} CertificateVerify;
```

Diese Nachricht wird gesendet, falls ein Clientzertifikat vom Server angefordert wurde. Sie besteht aus einem mit dem Private Key des Clients signierten Hash der bisherigen Handshake-Nachrichten und dient dazu den Client zu authentifizieren.

ChangeCipherSpec

Diese Nachricht gehört zum Change Cipher Spec Protocol. Es enthält lediglich diese eine Nachricht mit dem Wert 1. Das Empfangen signalisiert dem Empfänger, dass alle nachfolgend gesendeten Nachrichten mit den ausgehandelten kryptographischen Verfahren und Schlüsseln geschützt werden. Dazu werden read pending state (bei Empfang der Nachricht) und write pending state (nach Senden der Nachricht) in die current states kopiert.

Irgendwo schon erwähnt oder bei der Einführung?

Finished

```
struct {
    opaque verify_data[verify_data_length];
} Finished;
```

Die Finished-Nachricht dient zur Verifikation von erfolgreichem Schlüsselaustausch und Authentifikation. Diese Nachricht ist wie erwähnt die erste, die von den ausgehandelten Verfahren und Schlüsseln geschützt wird. Daher kann hier überprüft werden, ob der Handshake erfolgreich verlaufen ist und beiden Kommunikationspartnern die gleichen Informationen vorliegen.

Die Nachricht besteht aus einem Hash über die bisher gesendeten bzw. empfangenen Nachrichten des Handshake Protocols zusammen mit dem master-secret:

```
verify_data = PRF(master_secret,
                  finished_label,
                  Hash(handshake_messages)) [0..verify_data_length-1];
```

Das **`finished_label`** wird durch "client finished" auf der Client- bzw. "server finished" auf der Serverseite gebildet. Der Hash wird durch die in der PRF verwendete Hashfunktion berechnet (vgl. Abschnitt 3.2). **`verify_data_length`** entspricht, wenn durch die Cipher-Suite nicht anders vorgegeben, 12 Bytes Länge.

Nachdem **`verify_data`** vom Server und Client jeweils mit der selbst gesendeten Nachricht verglichen wurde, ist die Verbindung im Erfolgsfall aufgebaut. Nun sind die **`SecurityParameters`** vereinbart und bilden zusammen mit den berechneten Schlüsseln (vgl. Abschnitt 3.2) den Verbindungszustand.

```
struct {
    ConnectionEnd          entity;
    PRFAlgorithm           prf_algorithm;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  enc_key_length;
    uint8                  block_length;
    uint8                  fixed_iv_length;
    uint8                  record_iv_length;
    MACAlgorithm           mac_algorithm;
    uint8                  mac_length;
    uint8                  mac_key_length;
    CompressionMethod      compression_algorithm;
    opaque                 master_secret[48];
```

```
    opaque                  client_random[32];
    opaque                  server_random[32];
} SecurityParameters;
```

Diese Informationen enthalten Angaben zu verwendeter Cipher-Suite, zum Kompressionsalgorithmus und das master-secret. Sie werden vom Record Protocol für die Verschlüsselung und Authentifizierung von Nachrichten verwendet.

3.4 Alert Protocol

Das Alert Protocol dient dazu, auftretende Fehler zu versenden, die während der Kommunikation auftreten. Hierbei kann es sich zum Beispiel um fehlgeschlagene Überprüfung von entschlüsselten Nachrichten (bad_record_mac) oder fehlerhafte Zertifikatsüberprüfung (bad_certificate) handeln. Unterschieden wird zwischen Fehlern (fatal alert), die sofort zum Schließen der Sitzung führen, und Warnungen (warning alert). Eine Übersicht über alle Fehler findet sich in Abschnitt 7.2 von [DR08].

3.5 Application Data Protocol

Das Application Data Protocol ist zuständig für das Durchreichen von Anwendungsdaten, die von der Anwendungsschicht gesendet werden sollen. Die Daten werden durch das Record Protocol übertragen und damit durch die während des Handshakes ausgehandelten Verfahren und Schlüssel geschützt.

3.6 Sitzungen, Verbindungen und der verkürzte Handshake

Wo passt das Kapitel hier denn am Besten hin?

TLS erstellt beim ersten Handshake eine Sitzung zwischen Client und Server. Hierbei wird ein Sitzungsidentifikator erstellt, der beim ServerHello mitgesendet wird.

Ein Client kann nun später, wenn er den erhaltenen Sitzungsidentifikator in einer ClientHello-Nachricht mitschickt, eine alte Sitzung in Form einer neuen Verbindung wiederaufnehmen oder mehrere Verbindungen parallel aufbauen. Dabei werden die in den **`SecurityParameters`** hinterlegten Verfahren genutzt und aus dem ebenfalls hinterlegten master-secret sowie den in den Hello-Nachrichten übertragenen random-Werten neue Schlüssel berechnet wie in Abschnitt 3.2 beschrieben. Dadurch kommt der erneute Verbindungsaufbau mit weniger gesendeten Nachrichten aus, als ein neuer Verbindungsaufbau, wie in Abbildung 3.3 ersichtlich ist. So kann auf Neuberechnung des master-secret, Server- und Client-Validierung und Aushandlung der Cipher-Suite verzichtet werden. Durch die Finished-Nachricht können sich Client und Server durch das gleiche Schlüsselmaterial, das auf dem master-secret beruht, trotzdem sicher sein, mit dem optional authentifizierten Gegenüber zu kommunizieren.

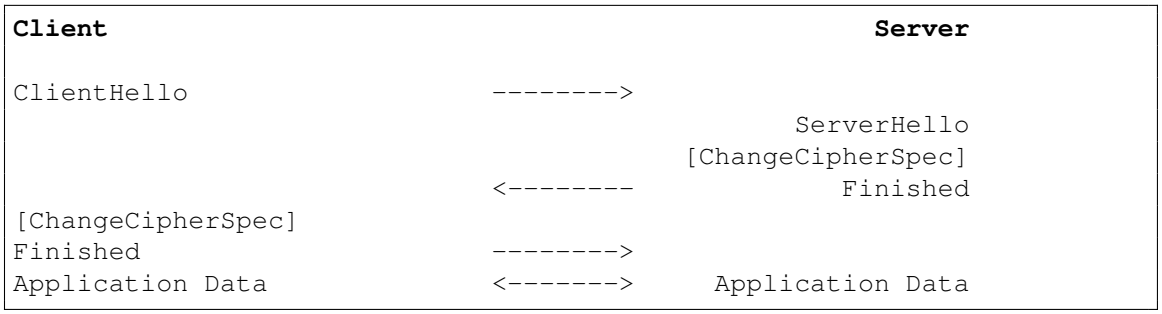

Abbildung 3.3: Nachrichtenverlauf beim abgekürzten TLS-Handshake. Entnommen aus [DR08].

3.7 Cipher-Suites

Eine Cipher-Suite legt fest, welche Algorithmen zum Schlüsselaustausch, zur Verschlüsselung und zur Berechnung des MACs verwendet werden, sowie welche Eigenschaften(Schlüssellänge, Blocklänge, ...) diese besitzen. In der TLS 1.2-Spezifikation ([DR08]) sind 37 Cipher-Suites festgelegt.

Zum Schlüsselaustausch stehen `RSA` sowie verschiedene Varianten[3] des Diffie-Hellman-Verfahrens zur Verfügung. Zur Verschlüsselung sind die Stromchiffre `RC4_128` sowie die Blockchiffren `3DES_EDE_CBC`, `AES_128_CBC` und `AES_256_CBC` festgelegt. Zur Berechnung des MACs können `MD5`, `SHA1` und `SHA256` verwendet werden (jeweils wie erwähnt unter Nutzung von HMAC).

Laut Spezifikation muss zumindest `TLS_RSA_WITH_AES_128_CBC_SHA` von jeder konformen Implementation angeboten werden.

Einen Sonderfall bildet die Cipher-Suite `TLS_NULL_WITH_NULL_NULL`, die vor der Festlegung der Cipher-Suite während des Handshakes als Standard festgelegt ist und weder Verschlüsselung noch MAC bietet.

Kurz sei hier auch noch erwähnt, dass frühere Versionen (bis TLS 1.1) aufgrund von gesetzlichen Vorschriften zum Export von Kryptographie in den USA Cipher-Suites enthielten, die durch Nutzung kürzerer Schlüssel leichter zu brechen sein sollten (so genannte export-geschwächte Cipher-Suites). Diese teilweise noch unterstützten Verfahren führen noch heute zu Angriffe auf SSL/TLS (siehe z.B. den FREAK-Angriff in Abschnitt 4.11).

3.8 TLS-Extensions

TLS-Extensions werden dazu genutzt, das Protokoll um zusätzliche Funktionalität zu erweitern. Das Konzept wurde parallel zu TLS entwickelt und mit TLS 1.2 in den Standard aufgenommen.

3. `DH`: Zertifikat mit festen Diffie-Hellman-Parametern
`DHE`: Temporäre Generierung von DH-Parametern für jede Sitzung
`DH_anon`: Nicht-authentifizierte DH-Parameter

In der ClientHello- und ServerHello-Nachricht können sich die Kommunikationspartner auf Extensions einigen, die von beiden Seiten unterstützt werden und im Verlauf der Sitzung genutzt werden können. Jeder Extension-Eintrag wird dabei durch ihren Typ und Extension-spezifische Daten gebildet.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

Beispiele für solche Extensions sind Server Name Indication, die es einem Server erlaubt abhängig vom geforderten Host verschiedene Zertifikate auszuliefern, oder Encrypt-then-MAC, die es erlaubt, die Reihenfolge von Verschlüsselung und Authentifizierung einer Nachricht im Record Protocol zu tauschen. Eine aktuelle Liste registrierter TLS-Extensions wird durch die IANA (Internet Assigned Numbers Authority) bereitgestellt[4].

3.9 Frühere SSL-/TLS-Versionen und TLS 1.3

Im Folgenden soll kurz auf frühere Versionen von TLS/SSL und die entscheidendsten Unterschiede zwischen diesen Versionen eingegangen, sowie ein kurzer Blick auf das noch nicht veröffentlichte TLS 1.3 geworfen werden.

SSL 1.0 wurde nie veröffentlicht.

SSL 2.0 war die erste Version, die öffentlich gemacht und auch patentiert wurde. In dieser Version bestanden einige große Schwachstellen: Der Handshake wurde noch nicht authentifiziert, so dass Angreifer beispielsweise die Cipher-Suite-Liste unbemerkt verändern konnten, viele schwache kryptographische Algorithmen wurden unterstützt und für Verschlüsselung und MAC-Berechnung wurden die gleichen Schlüssel verwendet. Auf diese Schwachstellen wird teilweise genauer in Kapitel 4 eingegangen. Da die Spezifikation nicht veröffentlicht wurde stammen diese Informationen aus [Mey14]. Die Unterstützung von SSL 2.0 wird für TLS-Implementierungen durch RFC 6176 ([TP11]) verboten.

In **SSL 3.0** wurden verschiedene Schlüssel zur Verschlüsselung und MAC-Berechnung eingeführt und die während der MAC-Berechnung genutzte Hashfunktion konfigurierbar gemacht (aber noch kein HMAC verwendet). Der Handshake wurde jetzt authentifiziert (durch den Inhalt der Finished-Nachricht) und in die Generierung des pre-master-secret floss jetzt die Versionsnummer ein, um Version-Rollback-Angriffe zu verhindern. Außerdem wurden neue kryptographische Algorithmen eingeführt und weitere kleine Änderungen vorgenommen (vgl. [FKK11]). Von der Abwärtskompabilität von TLS-Implementierungen zu SSL 3.0 wird in RFC 7568 ([BTPL15]) abgeraten.

TLS 1.0 ist großenteils äquivalent zu SSL 3.0. Es wurde die Pseudo Random Function eingeführt, die allerdings noch anders spezifiziert war als in der aktuellen Version. Die Berechnung des MACs erfolgte nun durch eine HMAC-Konstruktion. Außerdem wurde eine ChangeCipherSpec-Nachricht vor der Finished-Nachricht vorgeschrieben, um den DropChangeCipherSpec-Angriff zu verhindern (vgl. [DA99]).

4. http://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

In **TLS 1.1** wurden explizite Initialisierungsvektoren für Blockchiffren im CBC-Modus vorgeschrieben, um den in Abschnitt 4.7 vorgestellten Angriff zu verhindern. Das Verhalten bei Padding-Fehlern wurde verändert, um den Bleichenbacher-Angriff zu verhindern. Außerdem wurden die exportgeschwächten Cipher-Suites aus der Spezifikation entfernt (vgl. [DR06]).

In **TLS 1.2** wurde der Gebrauch von SHA1 und MD5 in der Pseudo Random Function durch eine Cipher-Suite-abhängige Hashfunktion ersetzt. Zusätzlich wurde die Unterstützung von AEAD-Cipher-Suites eingefügt. TLS-Extensions und der Gebrauch von AES als Blockchiffre wurden ergänzt. Außerdem wurden DES und IDEA als Blockchiffren aus der Spezifikation entfernt, die Unterstützung von SSL 2.0 nicht mehr empfohlen und weitere kleine Änderungen vorgenommen (vgl. [DR08]).

TLS 1.3 liegt momentan lediglich als Draft vor ([Res15], Version von Juli 2015]). Die hier dargestellten Informationen stellen also lediglich den aktuellen Entwicklungsstand dar und sind daher nicht als endgültig und nur mit Vorsicht zu betrachten. In dem aktuellen Draft wird Elliptic Curve Cryptography für das Diffie-Hellman-Verfahren hinzugefügt, die Unterstützung für alle SSL-Versionen und auch Kompression komplett entfernt (wahrscheinlich als Maßnahme gegen den Angriff aus Abschnitt 4.9). Die größte Veränderung ist die komplette Entfernung von Strom- und Blockchiffren im CBC-Modus zur Verschlüsselung. Es werden nur noch AEAD-Chiffren unterstützt, womit auch die separate Berechnung eines MACs entfällt. Außerdem wurde die PRF-Konstruktion durch die Verwendung von HKDF (HMAC-based Key Derivation Function, siehe [Kra10]) ersetzt.

https://www.trustworthpulse/ - SSL-Versionsverbreitung ergänzen?

TLS 1.0 RFC 2246 - http://tools.ietf.org/html/rfc2246

TLS 1.1 RFC 4346 - http://tools.ietf.org/html/rfc4346

TLS 1.2 RFC 5246 - http://tools.ietf.org/html/rfc5246

TLS 1.3 Draft -https://tools.ietf.org/html/draft-ietf-tls-tls13-07

TLS Extensions Z.B.
RFC 3546 - http://tools.ietf.org/html/rfc3546,
RFC 3466 - http://tools.ietf.org/html/rfc4366,
RFC 6066 - http://tools.ietf.org/html/rfc6066

4 Angriffe gegen SSL und TLS

Eine gute Übersicht zu bisherigen Angriffen auf TLS findet sich in [MS13]. Viele Schwächen früherer Protokollversionen bis SSL 3 sind in [WS96] zu finden.

JEDEN Angriff auf angreifbare Version und Änderungen in neuen Versionen überprüfen.

4.1 Version Rollback

Ein Angreifer kann eine SSL 3.0-konforme ClientHello-Nachricht so modifizieren, dass der Server eine SSL 2.0-Verbindung aufbaut. So kann der Angreifer alle Schwächen der älteren Protokollversion ausnutzen. In der SSL 3.0-Spezifikation ([FKK11]) wurde vorgeschrieben, dass bestimmte Bytes des PCKS #1-Paddings einen festen Wert erhalten sollten, falls der Client SSL 3.0 unterstützt. So bleibt Kompatibilität mit der älteren Version erhalten, aber Version-Rollback-Angriffe werden trotzdem erkannt.
Ein Schwachpunkt könnte laut [WS96] immer noch die Wiederaufnahme einer SSL 3.0-Sitzung durch eine SSL 2.0-ClientHello-Nachricht sein. Dieses sollte in Implementierungen verhindert werden.

Auch in neueren Browsern kann dieser Angriff noch zum Problem werden, wenn Fallback-Lösungen auf ältere Versionen implementiert sind, wenn ein Verbindungsversuch scheitert.

Durch die Finished-Nachricht ab SSL 3.0, die alle Handshake-Nachrichten authentifiziert, wird dieser Angriff verhindert.

4.2 Ciphersuite Rollback

Ein für SSL 2.0 bestehender Angriff ermöglichte aktiven Angreifern die während des Handshake-Protokolls übertragenen Listen von unterstützten Cipher Suites zu verändern, so dass schwache kryptographische Verfahren erzwungen werden konnten (oftmals exportgeschwächte Verfahren mit kürzeren Schlüsselängen).

Ab SSL 3.0 wird dieser Angriff dadurch verhindert, dass die Finished-Nachrichten von Client und Server jeweils einen mit dem master-secret berechneten MAC über die Nachrichten des *Handshake*-Protokolls enthalten, der die Integrität dieser Nachrichten bestätigt.

Eine detaillierte Übersicht ist in [WS96] zu finden.

4.3 Verhindern der ChangeCipherSpec-Nachricht

Im Sonderfall einer SSL-Verbindung, die lediglich die Integrität der Nachrichten schützen soll, aber nicht verschlüsselt, lässt sich ausnutzen, dass der in der Finished-Nachricht gesendete MAC die ChangeCipherSpec-Nachricht nicht mit einschließt. Dadurch kann ein aktiver Angreifer diese Nachrichten abfangen und nicht weiterleiten, sodass die Verbindungspartner die Integritätsprüfung nicht einsetzen. Ein Angreifer ist so in der Lage, gesendete Nachrichten zu verändern.

Theoretisch wäre der Angriff unter bestimmten Voraussetzungen und schwacher Kryptographie auch bei verschlüsselten Verbindungen möglich. Dazu müsste die empfangene Finished-Nachricht vor dem Weiterleiten entschlüsselt werden. In bestimmten Fällen wäre zwar genug bekannter Klartext vorhanden, um einen Brute-Force-Angriff auf den Schlüssel zu erlauben, aber selbst bei exportgeschwächten 40-Bit Schlüsseln ist der Rechenaufwand hierfür sehr groß und unter praktischen Gesichtspunkten kaum unbemerkt ausführbar.

Alle TLS-Versionen verhindern diesen Angriff dadurch, dass sie eine ChangeCipherSpec-Nachricht vor der Finished-Nachricht explizit vorschreiben.

Details zu diesem Angriff sind in [WS96] zu finden.

4.4 Bleichenbacher-Angriff

Daniel Bleichenbacher stellte 1998 in [Ble98] einen Adaptive-Chosen-Ciphertext-Angriff gegen RSA-basierte Protokolle vor. Im Fall von SSL wird versucht, das pre-master-secret, das während des Handshakes RSA-verschlüsselt gesendet werden kann, zu erhalten.

Der Angriff basiert auf dem festen Format nach PKCS #1 (siehe [JK03]) formatierter Nachrichten, wie in Abbildung 4.1 dargestellt. Die ersten beiden Bytes haben immer den gleichen Wert, danach folgen Padding (bestehend aus zufälligen Bytes ungleich null) und die Daten, getrennt durch ein Nullbyte. Nachrichten in diesem Format werden als Integer interpretiert, per RSA verschlüsselt und versendet. Der Empfänger entschlüsselt die Nachricht, überprüft das Format des als Bytekette interpretierten Ergebnisses und kann dann die Daten wieder extrahieren.

Abbildung 4.1: PCKS #1-Format

Voraussetzung für diesen Angriff ist der Zugriff auf ein Orakel, das dem Angreifer für eine verschlüsselte Nachricht lediglich mitteilt, ob das Padding der entschlüsselten Nachricht korrektes Format besitzt.

Im Folgenden sei (n,e) ein RSA-Public-Key und (n,d) der zugehörige Private-Key. Der Angreifer möchte eine Nachricht $m \equiv c^d \mod n$ erhalten, für die er im Besitz von c ist.
Dazu wählt er eine Zahl s, berechnet $c' \equiv cs^e \mod n$ und sendet c' an das Orakel. Wenn das Orakel korrektes Format signalisiert, dann weiß der Angreifer, dass die ersten zwei Bytes von

$$(c')^d \equiv (cs^e)^d \equiv c^d s \equiv ms \mod n$$

$0x00$ und $0x02$ sind. Mit diesem Wissen lässt sich ein neuer Wert s wählen, der weitere Informationen über m enthüllt. Details zu diesem iterativen Verfahren sind in [Ble98] zu finden. Der Autor schätzt die Anzahl an nötigen Orakelanfragen mit etwa 2^{20} ab.
Das Orakel lässt sich auf zwei Weisen erhalten. Entweder gibt die Implementierung detaillierte Fehlermeldungen über ungültiges PKCS-Format zurück oder ermöglicht durch Zeitunterschiede bei der Verarbeitung gültiger und ungültiger Nachrichten einen Timing-Angriff.

In TLS ab Version 1.0 wird der Angriff dadurch verhindert, dass bei ungültigem PKCS-Format ein zufälliges pre-master-secret erzeugt wird, mit dem der Handshake fortgesetzt wird. Dadurch scheitert der Handshake erst bei der Überprüfung der Finished-Nachricht und enthüllt keine Informationen über gültiges oder ungültiges Format.

4.5 Padding-Oracle-Angriff

In [Vau02] beschreibt der Autor einen Angriff zur Erlangung des Klartextes, bei dem das für Blockchiffren nötige Padding im CBC-Modus ausgenutzt wird. Durch das vorgegebene Format des Paddings und da das Padding bei TLS nicht durch den MAC geschützt ist (MAC - then PAD - then Encrypt) ermöglicht es theoretisch in einer relativ kleinen Zahl von Anfragen die Berechnung des Klartextes.
Praktisch konnte das Verfahren nicht eingesetzt werden, da SSL 3.0 für Paddingfehler und MAC-Überprüfung gleiche Fehlermeldungen (bad_record_mac) ausgibt. In TLS 1.0 und 1.1 gibt es getrennte Fehlermeldungen (bad_record_mac und decryption_failed), so dass der Angriff theoretisch möglich wäre. Allerdings werden die Fehler über das Alert Protocol verschlüsselt gesendet, so dass ein Angreifer die Fehlerart anders (beispielsweise über Log-Einträge) erhalten muss. TLS 1.2 verbietet das Senden von decryption_failed-Fehlern aus diesem Grund.
Ein weiterer Nachteil ist, dass es sich bei bad_record_mac- und decryption_failed-Fehlern um fatal alerts handelt, die zum Abbruch der Sitzung führen.

Das verwendete Padding besteht immer aus Bytes mit dem Wert n, wobei n die nötige Anzahl an Paddingbytes bis zum Erreichen eines Vielfachens der Blocklänge bezeichnet. Das Padding kann also folgende Werte annehmen: $1, 22, 333, 4444, \ldots$. Voraussetzung für diesen Angriff ist der Zugriff auf ein Orakel, das dem Angreifer für eine verschlüsselte Nachricht lediglich mitteilt, ob das Padding der entschlüsselten Nachricht korrektes Format besitzt. Im Folgenden bezeichnet $C(x)$ die Verschlüsselung des Blockes x und $C^{-1}(y)$ die Entschlüsselung von y.
Wenn der Angreifer nun das letzte Byte eines Chiffretextblocks y erhalten möchte, so sendet er $r|y$ mit $r = r_1, \ldots, r_b$ als zufällige Bytes und b als Blocklänge (in Byte) an das Orakel. Bei der Entschlüsselung im CBC-Modus wird der letzte Chiffretextblock (hier y) entschlüsselt und mit dem vorletzten Block XOR-verknüpft, um den Klartextblock zu erhalten. Dieser Block (hier x) wird dann auf gültiges Padding überprüft:

$$x = C^{-1}(y) \oplus r$$

Wenn das Orakel gültiges Padding signalisiert, dann ist am wahrscheinlichsten, dass x auf 1 endet und somit das letzte Byte von $C^{-1}(y) = r_b \oplus 1$ ist. Bei ungültigem Padding wird ein neuer Wert r_b gewählt und das Orakel neu befragt.
In [Vau02] wird ein Algorithmus, mit dem auch die unwahrscheinlicheren Fälle von längerem Padding abgedeckt werden, und ein Verfahren, um aus dem letzten Byte einen kompletten Block zu erhalten, angegeben.

In [CHVV03] beschreiben die Autoren eine Umsetzung des Angriffs auf TLS-gesicherte IMAP-Verbindungen zur Erlangung von Passwörtern. Hierbei wird das Problem ununterscheidbarer und verschlüsselter Fehlermeldungen durch einen Timing-Angriff umgangen. Außerdem bedenken die Autoren das Abbrechen der Sitzung durch Nutzung vieler paralleler Sitzungen mit dem gleichen verschlüsselten Aufruf (wie es bei der Authentifizierung im IMAP-Protokoll der Fall ist).

4.6 Lucky Thirteen

In [AFP13] stellen die Autoren weitere auf [Vau02] basierende Angriffe vor, die ebenfalls auf Timing-Angriffen zur Erkennung falschen Paddings und mehrere Verbindungen setzen.

4.7 Chosen-Plaintext-Angriff gegen bekannte IVs

In [Bar04] stellt der Autor einen Angriff vor, der die Art ausnutzt, wie die für den CBC-Modus nötigen Initialisierungsvektoren (IV) von TLS bereitgestellt werden. Durch die Nutzung des letzten Ciphertextblocks der letzten Nachricht als IV der neuen Nachricht lässt sich unter bestimmten Voraussetzungen ein Chosen-Plaintext-Angriff durchführen. Der Autor beschreibt eine Möglichkeit unter Nutzung von Browser-Plugins über HTTPS übertragene Passwörter oder PINs herauszufinden. In [Bar06] verbessert der Autor seinen Angriff durch die Nutzung von Java-Applets anstelle von Browser-Plugins.

Der eigentliche Angriff folgt dem folgenden Prinzip: Wenn eine Nachricht $C = C_0, \ldots, C_l$ gesendet wurde, wird für die nächste Nachricht C_l als IV verwendet werden. Wenn der Angreifer überprüfen möchte, ob ein Klartextblock $P^* = P_j$, also zu C_j verschlüsselt wurde, so bringt er einen Sender dazu eine Nachricht P' mit dem ersten Block $P'_1 = C_{j-1} \oplus C_l \oplus P*$ zu verschlüsseln und erhält als ersten Chiffretextblock:

$$\begin{aligned} C'_1 &= C_K(P'_1 \oplus \mathrm{IV}) \\ &= C_K(P'_1 \oplus C_l) \\ &= C_K(C_{j-1} \oplus C_l \oplus P^* \oplus C_l) \\ &= C_K(C_{j-1} \oplus P^*) \end{aligned}$$

Außerdem gilt auch $C_j = C_K(P_j \oplus C_{j-1})$. Der Angreifer kann nun überprüfen, ob $C'_1 = C_j$ und damit $P^* = P_j$ gilt, ob also seine Wahl für den gesuchten Klartextblock stimmt.

Seit TLS 1.1 werden explizite IV vorgeschrieben. Hierzu besteht jede Nachricht aus einem Ciphertextblock mehr als Klartextblöcken. Dieser erste Block bildet den IV für die restliche Verschlüsselung. Da dieser IV nicht vor dem Senden der Nachricht bekannt ist, wird der hier beschriebene Chosen-Plaintext-Angriff verhindert.

4.8 BEAST

In [DR11] und in einem Konferenzbeitrag auf der ekoparty Security Conference 2011 wurde von den Autoren das Tool BEAST vorgestellt, das die Ideen aus [Bar04] aufgreift. Die Autoren

erweiterten den Angriff jedoch auf einen sogenannten block-wise chosen-boundary Angriff, bei dem der Angreifer die Lage der Nachricht in den verschlüsselten Blöcken verändern kann. Die Autoren zeigten auch die praktische Umsetzbarkeit am Beispiel des Entschlüsselns einer über HTTPS gesendeten Session-ID.

4.9 CRIME

Auf der ekoparty Security Conference 2012 stellten die Entdecker des BEAST-Angriff einen weiteren Angriff vor, der die (optionale) Kompression in TLS nutzt, um beispielsweise Cookiedaten zu stehlen.

Dabei wird ausgenutzt, dass Kompressionsalgorithmen bereits verwendete Zeichenketten beim erneuten Auftreten verkürzen. Wird nun beispielsweise ein HTTP-Header wie `Cookie: twid= secret` gesendet, kann ein Angreifer durch das Einbringen von `Cookie: twid= a...` und `Cookie: twid= s...` einen Längenunterschied der Nachrichten feststellen und so das Geheimnis zeichenweise erhalten.

Als Folge wurde Kompressionsunterstützung in Firefox und Chrome deaktiviert. Kompression ist im aktuellen Draft von TLS 1.3 ([Res15]) nicht mehr enthalten.

4.10 Poodle

In [MDK14] nutzen die Autoren den erneuten Verbindungsversuch mit älteren Protokollversionen, wenn der Handshake fehlschlägt (SSL 3.0-Fallback), der in vielen TLS-Implementierungen eingesetzt wird. Darauf aufbauend beschreiben sie einen Angriff, der bestehende Schwächen in der RC4-Chiffre bzw. in der Nicht-Prüfung von Padding im CBC-Modus in SSL 3.0 ausnutzt, um Cookiedaten zu stehlen. Von der Unterstützung von SSL 3.0 wird in [BTPL15] abgeraten.

RC4 noch irgendwo unterbringen? Vllt bei den Ciphersuites? http://www.isg.rhul.ac.

4.11 FREAK

Eine Gruppe von Pariser Wissenschaftlern entdeckte eine Möglichkeit, wie ein Angreifer die Kommunikationspartner während des Handshakes zur Nutzung schwacher Kryptographie (RSA export cipher suite) bringen kann. Weiterhin zeigten sie in [BDLF+] die Machbarkeit der Faktorisierung der entsprechenden RSA-Module und die Praxistauglichkeit des Angriffs.

Der Angriff beruht auf Fehlern in TLS-Implementierungen, die schwache RSA-Schlüssel vom Server akzeptieren, selbst wenn sie die Cipher-Suitesnicht anbieten.

4.12 logjam

In [ABD+] beschreiben die Autoren mehrere Angriffe gegen die Nutzung von Diffie-Hellman(DH)-Schlüsselaustausch während des TLS Handshakes. Ein Angriff richtet sich gegen kleine DH-Parameter (DHE-EXPORT), ein weiterer nutzt die weite Verbreitung von standardisierten

DH-Parametern, um mittels Vorberechnung bestimmter Werte schneller diskrete Logarithmen für beim DH-Verfahren gesendete Nachrichten zu berechnen.

4.13 Zertifikate und Verwandtes

Viele Probleme, die in den letzten Jahren aufgetreten sind, betreffen nicht das TLS-Protokoll direkt, sondern die Erstellung und Validierung von (insbesondere) Server-Zertifikaten, und seien deshalb nur am Rande erwähnt. Ein guter Überblick ist in [MS13] zu finden.

Viele dieser Angriffe richteten sich gegen mangelnde Zertifikatvalidierung in TLS-Implementierungen (keine Validierung, keine Überprüfung des Servernamens, Akzeptanz unsignierter oder abgelaufener Zertifikate, ...) oder wenig Sorgfalt bei der Zertifikaterstellung durch Certificate Authorities (Nutzung von MD5, fehlerhafte Validierung von übermittelten Servernamen, fehlerhafte Ausgabe von intermediate-Zertifikaten, mangelhaft abgesicherte Server, ...).

Der Vollständigkeit halber sei hier auch noch die notwendige Sicherheit des privaten Serverschlüssels erwähnt. Gelangt ein Angreifer in seinen Besitz, so kann er den Datenverkehr problemlos mitlesen oder verändern.

5 TLS in der Lehre

Stichworte

- “einfaches“ Protokoll, das gut zur Erklärung einzelner Verfahren dienen kann (Record-Protocol für vertrauliche und authentifizierte Nachrichtenübertragung, Handshake zur Schlüsselaushandlung, ...)
- Schwerpunkte für die Lehre setzen
- Viele Angriffe, die Abwehrmaßnahmen und auch “Prinzip des schwächsten Kettenglieds“ deutlich machen (CBC-IV, Padding, MAC-then-Encrypt)
- weite Verbreitung -> notwendiges Verständnis
- Familien von Angriffen betrachten?

Informatik-Didaktik, Anforderungen an Visualisierungen, ... -> Methodik erläutern

- Ludger Humbert, Didaktik der Informatik, Stuttgart: Teubner, 2005
- Sigried Schubert/Andreas Schwill, Didaktik der Informatik, Heidelberg/Berlin: Spektrum, 2004
- Hilbert Meyer, Unterrichtsmethoden, 2 Bände, Berlin: Cornelsen, 11.Aufl. 2005

Hier die Überleitung zu Kapitel Implementierung.

6 Implementierung

6.1 Architekturentscheidungen

Zugrundeliegend: Gemeinsamkeit von Protokollen, Erweiterbarkeit/Pluginfähigkeit, ...

Warum Java? (UHH als Einstieg-> für alle verständlich und auch erweiterbar, ...)

Welche TLS-Version? Oder SSL 2.0? Aus welchen Gründen?

Welche Dinge werden betrachtet, welche ausgelassen (zB Extensions, Zertifikatvalidierung, ...), aus welchen Gründen?

6.2 Implementation notes

Analyse (Rückgriff auf Didaktik-Kapitel), Entwurf (UML ist toll), Implementierung, Tests, ...

Probleme/Schwierigkeiten bei der Umsetzung, ...

6.3 Tutorial: So schreibe ich ein Plugin, ...

Evtl. auch in den Anhang?

A Fehlermeldungen des Alert Protocols

Literaturverzeichnis

[ABD+] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. https://weakdh.org/imperfect-forward-secrecy.pdf. Zugriff am 22.05.2015.

[AFP13] N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *2014 IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[Bar04] Gregory V. Bard. The Vulnerability of SSL to Chosen Plaintext Attack, 2004.

[Bar06] Gregory V. Bard. A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. In *SECRYPT 2006, Proceedings of the international conference on security and cryptography*, pages 7–10. INSTICC Press, 2006.

[BDLF+] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella-Béguelin, Jean-Karim Zinzindohoué, and Benjamin Beurdouche. FREAK: Factoring RSA Export Keys. https://www.smacktls.com/#freak. Zugriff am 22.05.2015.

[Ble98] Daniel Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pages 1–12, London, UK, 1998. Springer-Verlag.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *Advances in Cryptology - ASIACRYPT 2000*. Springer Berlin Heidelberg, 2000.

[BTPL15] R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0. RFC 7568, 2015.

[CHVV03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password Interception in a SSL/TLS Channel. In *Advances in Cryptology - CRYPTO 2003*, volume 2729, pages 583–599. Springer, 2003.

[DA99] T. Dierks and C. Allen. The TLS Protocol - Version 1.0. RFC 2246, 1999.

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol - Version 1.1. RFC 4346, 2006.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol - Version 1.2. RFC 5246, 2008.

[DR11] Thai Duong and Juliano Rizzo. Here Come The XOR Ninjas. 2011.

[Eck13] Claudia Eckert. *IT-Sicherheit - Konzepte, Verfahren, Protokolle*. Oldenbourg Verlag, München, 2013.

[FKK11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, 2011.

[FSK10] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.

[JK03] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, 2003.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, 1997.

[Kra01] Hugo Krawczyk. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 310–331, London, UK, 2001. Springer-Verlag.

[Kra10] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 631–648, 2010.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting the SSL 3.0 Fallback, 2014.

[Mey14] Christopher Meyer. *20 Years of SSL/TLS Research - An Analysis of the Internet's Security Foundation*. Dissertation, Ruhr-University Bochum, 2014.

[MS13] Christopher Meyer and Jörg Schwenk. SoK: Lessons Learned From SSL/TLS Attacks. In *The 14th International Workshop on Information Security Applications*, 2013.

[Res15] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 - draft-ietf-tls-tls13-07. Technical report, 2015.

[Sch09] Klaus Schmeh. *Kryptographie - Verfahren, Protokolle, Infrastrukturen*. dpunkt.verlag, Heidelberg, 2009.

[TP11] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176, 2011.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology - EUROCRYPT 2002*, pages 534–545. Springer, 2002.

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *The Second USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX Association, 1996.

Todo list