Hypertext Transport Protocol
(Fortsetzung)

Aufbau von Nachrichten

HTTP implementiert ein Request-Response Modell.
Eine Nachricht hat den Aufbau:

```
Anfragezeile/Statuszeile
Headerzeilen
Leerzeile
optionaler Datenblock
```

Eine Anfragezeile hat die Form

```
Methode ' ' URI ' ' HTTP-Version
```

(' ' Leerzeichen, URI, Uniform Resource Identifier)

Eine Statuszeile hat die Form

```
HTTP-Version Status-Code Reason-Phrase
```

HTTP 1.1 Methoden

Folgende Methoden sind standardisiert:

Name	Zweck
OPTIONS	Lesen von Transportoptionen, z.B. `Allow`
GET	Lesen von Daten vom Server
HEAD	Wie GET, Server sendet keinen Datenblock
POST	Senden von Daten zum Server
PUT	Speichern von Daten unter dem URI
DELETE	Löschen von Daten unter dem URI
TRACE	Server sendet den Request als Response
CONNECT	Nur für SSL/TLS Proxies

Ein Server muß nicht alle diese Methoden implementieren, Erweiterungen sind möglich.

Uniform Resource Identifier (vgl. RFC2396)

Die HTTP URL (Uniform Resource Locator) hat die Form:
`"http://" host [ ":" port ] [ path [ "?" query ]]`
Dabei sind Teile in eckigen Klammern "[]" optional.

- **Host**: Hostname des Dienstes.
- **Port**: Portnummer, 80, falls weggelassen.
- **path**: Pfad zur Resource, jeder Pfad startet mit "/"
- **query**: Serverspezifische Parameter, üblicherweise Name "=" Wert Paare, getrennt durch "&" (sog. Formkeys)

IP Adresse statt Hostname ist in URIs zu vermeiden. Falls eine IPv6 Adresse angegeben werden muß, wird sie in eckige Klammern geschrieben.

Schreibweise für URIs

Da URIs auch über andere Transportwege als Computernetze weitergegeben werden, dürfen sie nur druckbare Zeichen enthalten.

Im Pfad einer URI sind mindestens die folgenden Zeichen US-ASCII Zeichen zu ersetzen:

- Codes 0-31 und 127: Control Character
- Code 32: Space
- < > # % ¨: Begrenzer
- { } | \ ˆ [] ‘: Diese Zeichen können von bestimmten Transportmechanismen verändert werden.

Escape Sequenz in URIs

Einige Zeichen haben an bestimmten Stellen der URI eine spezielle Bedeutung und müssen dann ebenfalls ersetzt werden (z.B. im Query Anteil):

- ; / ? : @ & = + $,

Spezielle Zeichen und nicht druckbare Zeichen werden durch ihre Hexadezimaldarstellung, eingeleitet durch ein %-Zeichen, angegeben.

Beispiel: Host `a.b.c`, Port 80, Path `/a_b/c` und Formkeys M=a+b, N="a=b" wird geschrieben als:
`http://a.b.c/a_b/c?M=a%2Bb&N=%22a%3Db%22`

Minimaler HTTP 1.1 Request

Die URL wird vom Client in einen HTTP 1.1 Request umgesetzt. Dazu wird die Serverinformation, d.h. Hostname und Port in den `Host` Header kopiert. Die Anfragezeile enthält im Normalfall als URI nur Path, Query und HTTP-Version. Ausnahme ist der Proxy Request.
Beispiel: Ein `GET` Request auf
`http://localhost:8080/test`
führt zu

```
GET /test HTTP/1.1
Host: localhost:8080
```

Der Header mit Namen `Host` ist bei HTTP 1.1 verbindlich.

HTTP 1.1 Header

In Request und Response sind unterschiedliche Header üblich. Prinzipiell können beiden Nachrichten beliebige Header hinzugefügt werden (sog. extension-header).

Für Requests sind die folgenden Headergruppen standardisiert:

- General Header
- Request Header
- Entity Header

Für Responses entsprechend:

- General Header
- Response Header
- Entity Header

General Header

Ein Auszug aus der Liste der standardisierten General Header:

- **Cache-Control**: Legt fest, was beim Caching der Daten zu beachten ist. (z.B. no-cache, max-age)
- **Connection**: Zeigt an, ob die TCP Verbindung für weitere Anfragen verwendet werden kann (z.B. keep, close)
- **Date**: Zeitpunkt, zu dem die Nachricht erzeugt worden ist.
- **Pragma**: Implementationsspezifische Parameter (z.B. no-cache)
- **Trailer**: Bei Chunked-Encoding können die angegebenen Header am Ende der Nachricht im Datenblock auftauchen.
- **Transfer-Encoding**: Legt fest, wie der Datenblock übertragen wird (z.B. chunked).
- **Via**: Wird von Systemen zwischen Quelle und Ziel eingesetzt, um Schleifen zu entdecken.

Transfer-Encodings

RFC2616 erwähnt folgende Kodierungen:

- **identity**: Die Nachricht wird nicht speziell kodiert.
- **gzip**: Transparente Komprimierung mit Lempel-Ziv Algorithmus.
- **compress**: Unix `compress` LZW Format.
- **deflate**: zlib Format (RFC1950 + RFC1951)
- **chunked**: Der Datenblock besteht aus Länge-Wert kodierten Segmenten.
  - Jedes Segment beginnt mit einer Zeile mit Länge in Hexadezimaldarstellung und optionalem Kommentar
  - Es folgt die angegebene Anzahl Bytes.
  - Das letzte Segment hat die Länge 0
  - Auf das letzte Segment können HTTP Header folgen.

Request Header

Ein Auszug aus der Liste der Request Header:

- **Accept, Accept-Charset, ...**: Liste der Mediatypen, Kodierungen und Sprachen, die der Client akzeptiert
- **Authorization, Proxy-Authorization**: Übergabe von Daten und Anforderung der Authentifizierung.
- **Host**: Servername (eventuell mit Port)
- **If-Match, If-Modified-Since, ...**: Anforderung von Daten nur unter gegebener Bedingung.
- **Max-Forwards**: Maximale Anzahl von Proxies in der Kette.
- **Referer**: URI, von der aus die aktuelle URI angewählt worden ist.
- **TE**: Liste der möglichen Transfer-Encodings
- **User-Agent**: Identifikation des Clients

Response Header

Ein Auszug der Liste der Response-Header:

- **Age**: Alter eines Dokumentes (im Cache)
- **ETag**: Dokumentversion
- **Location**: URI des Dokumentes (benutzt im `201 Created` und z.B. im `302 Redirect`)
- **Proxy-Authenticate, WWW-Authenticate**: Authentifizierungsdaten des Clients
- **Retry-After**: Im `503 Service Unavailable` und bei `3xx Redirect`, wann die Daten verfügbar sein werden.
- **Server**: Identifikation (Typ, Version) des Servers
- **Vary**: Liste der Request-Header, die die Antwort festlegen, nötig für Proxies, um zu entscheiden, ob die Antwort aus dem Cache benutzt wird.

Entity Header

- **Allow**: Methoden, die der Server erlaubt
- **Content-Encoding**: Kodierung, die für die Daten verwendet worden ist (z.B. gzip)
- **Content-Language**: ausgewählte Sprache
- **Content-Length**: Anzahl Bytes im Datenblock, falls nicht `Transfer-Encoding: chunked`.
- **Content-Location**: URI des Dokuments
- **Content-MD5**: Hash der Daten, Prüfsumme des (dekodierten) Datenblocks
- **Content-Range**: gelieferter Bereich in der Antwort `206 Partial Content`
- **Content-Type**: Medientyp
- **Expires**: Wann Daten im Cache veraltet sind
- **Last-Modified**: Zeitpunkt der letzten Änderung

Content-Type Header

- Der Content-Type gibt den Medientyp im Datenblock an.
- Format ist stets `Type "/" Subtype*(";" Parameter)`
- Die standardisierten Medientypen finden sich unter `http://www.iana.org/assignments/media-types/`
- Parameter dienen zur weiteren Festlegung der Interpretation, z.B. `text/plain; charset=utf8`.
- Der Typ `multipart/form-data` (vgl. RFC1867) dient der Übertragung von Formkeys mit Zusatzinformationen wie Zeichensatz oder Medientyp bei Dateiübertragung.

Viele Clients (z.B. MS Windows, Mobiltelefone) benutzen zur Feststellung des Medientyps nicht den Content-Type Header, sondern eventuell vorhandene Dateiheader.

HTTP Proxies

Proxies sind integraler Bestandteil einer HTTP Infrastruktur. Haupteinsatzzwecke sind:

- **Effizienzsteigerung**: Zwischenspeichern von statischen Daten reduziert den Netzwerkverkehr
- **Zugriffskontrolle**: Netzwerkbereiche können nur über Proxies erreicht werden, die Authentifizierung vorschreiben.
- **Protokollierung/Abrechnung**: Proxies können Datenvolumen und Zugriffszeiten protokollieren.
- **Routing**: Zugriff aus privaten Netzen kann über die öffentliche Adresse eines Proxies ermöglicht werden.
- **Sicherheit**: Komplexe Webserver werden durch einfache Proxies vom Internet isoliert.

HTTP Proxies

Man unterscheidet folgende Typen von Proxies:

- **Vorwärtsproxies**: Der Client muß konfiguriert werden, um den Proxy zu benutzen.
- **Rückwärtsproxies**: Dem Client gegenüber verhalten sie sich wie Server. Sie blenden Pfade fremder Server in den eigenen Bereich ein.
- **Transparente Proxies**: Die TCP Verbindung von Clients wird abgefangen und auf den Proxy umgeleitet. Dieser baut bei Bedarf eine eigene Verbindung zum Server auf.

Die Anfragezeile bei Vorwärtsproxies enthält nicht nur den Pfad, sondern die komplette URL.

HTTP Authentifizierung

Soll mittels HTTP auf geschützte Bereiche sowohl auf einem Server als auch hinter einem Proxy zugegriffen werden, erlaubt der Standard, daß Authentifizierungsdaten abgefragt werden.

Wird der Zugriff verweigert, sendet ein Server eine Response mit Statuscode 401, ein Proxy eine Response mit Statuscode 407.

In der Response findet sich der `WWW-Authenticate` oder `Proxy-Authenticate` Header, der festlegt, wie der Client auf den geschützten Bereich zugreifen kann.

Der Client muß dann die Authentifizierungsdaten im `Authorization` bzw. `Proxy-Authorization` Header liefern.

Basic Authentication (RFC2617)

Basic Authentication funktioniert dadurch, daß beim Zugriff auf den geschützten Bereich Benutzername und Kennwort im Klartext übertragen werden.

Findet ein Zugriff ohne gültige Authentifizierung statt, folgt vom Server eine Antwort mit Status 401 und `WWW-Authenticate` Header mit Parametern `basic` und dem Namen des Bereiches.

Der Client wiederholt den Request und sendet den Header `Authorization: Credentials`, wobei Credentials Benutzername ":" Kennwort in Base64 Kodierung ist.

Beispiel Basic Authentication, erster Versuch

```
GET //scripts/Literaturliste.pdf HTTP/1.0
Host: www.comnets.rwth-aachen.de


HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic
realm=ßcript-downloads"
Content-Type: text/html; charset=iso-8859-1

HTML Fehlertext
```

Beispiel Basic Authentication, erfolgreich

```
GET //scripts/Literaturliste.pdf HTTP/1.0
Host: www.comnets.rwth-aachen.de
Authorization: Basic dXNlcjpwYXNzd29yZA==


HTTP/1.1 200 OK
Content-Length: 34544
Content-Type: application/pdf

PDF Dokument
```

Digest Access Authentication (RFC2617)

Der entscheidende Nachteil der Basic-Authentication ist, daß jeder, der die Datenübertragung abhört, Benutzernamen und Kennwort erfährt. Dies wird bei der Digest-Authentication vermieden.

Der `WWW-Authenticate` oder `Proxy-Authenticate` Header hat die Form: `Digest Challenge`, wobei Challenge eine Folge von Name "=" Wert Paaren ist, die durch "," getrennt werden.

Die Antwort hat die Form `Digest Response`, wobei Response dasselbe Format wie Challenge hat.

Challenge Parameter

- **realm**: Name des Sicherheitsbereiches
- **domain**: Liste von URI Präfixen, für die die Credentials gelten
- **nonce**: Eindeutige Challenge
- **opaque**: Wert, der vom Client zum Server zurückgeschickt wird
- **stale**: Zeigt an, daß der vorherige Nonce abgelaufen ist.
- **algorithm**: Zu verwendender Hash Algorithmus, z.B. MD5, SHA1
- **qop**: angebotene Sicherheitsstufen, z.B. Authentizität (auth), Integrität (auth-int)
- **auth-param**: z.Zt nicht benutzt

Response Parameter

- **username**: Name, unter dem der Client sich anmeldet
- **realm, nonce, algorithm, opaque**: die Werte der Challenge.
- **uri**: URI, die zur Challenge geführt hat (die Anfragezeile könnte von einem Proxy verändert worden sein)
- **response**: Hash, der aus Nonce, Benutzername, Kennwort und möglicherweise weiteren Bestandteilen der Nachricht berechnet wurde.
- **cnonce**: Zufälliger Text des Clients, der in den Hash einbezogen wird und Chosen Plaintext Angriffe verhindert.
- **qop**: gewählte Sicherheitsstufe
- **nonce-count**: Zähler, wie oft der nonce in Requests verwendet worden ist
- **auth-param**: z.Zt. nicht benutzt

Digest Auth Beispiel

```
GET /apache2-default/ HTTP/1.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US)
Host: localhost
Accept: text/html;q=0.9,text/plain;q=0.8,*/*;q=0.5
Accept-Language: en
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: close
If-Modified-Since: Sat, 20 Nov 2004 20:16:24 GMT
Cache-Control: max-age=0
```

Digest Auth Beispiel

```
HTTP/1.1 401 Authorization Required
Date: Sun, 09 Dec 2007 15:07:31 GMT
Server: Apache/2.2.6 (Debian)
WWW-Authenticate: Digest realm="my-realm",
  nonce="6by31ttABAA=1b869666ab4d0c05c785475d376797a
  algorithm=MD5, qop="auth"
Content-Length: 475
Connection: close
Content-Type: text/html; charset=iso-8859-1

475 Bytes Fehlertext
```

```
GET /apache2-default/ HTTP/1.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US)
Host: localhost
Accept: text/html;q=0.9,text/plain;q=0.8,*/*;q=0.5
Accept-Language: en
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: close
If-Modified-Since: Sat, 20 Nov 2004 20:16:24 GMT
Cache-Control: max-age=0
Authorization: Digest username="username",
  realm="my-realm",
  nonce="6by31ttABAA=1b869666ab4d0c05c785475d376797ac126
  uri="/apache2-default/", algorithm=MD5,
  response="6f8e48f7d00453d12f63405b53984537",
  qop=auth, nc=00000001, cnonce="de371d3080a00b58"
```

Cookies

HTTP bietet im Protokoll keine Möglichkeit, Zustandsinformationen zwischen verschiedenen Requests zu transportieren. Zwar gibt es eine Vielzahl von Ansätzen, dieses Problem zu lösen, die jedoch alle anwendungsabhängig sind oder nur eingeschränkte Möglichkeiten bieten (z.B. HTML Hidden Felder oder Session ID in der URL).

Netscape hat daher HTTP um sogenannte Cookies erweitert:

- Die originale Spezifikation findet sich unter `http://wp.netscape.com/newsref/std/cookie_spec.h`
- RFC2109 erweitert den Vorschlag von Netscape so, daß bisherige Server und Clients interoperieren können.
- RFC2965 führt einen neuen Header ein, um die Zustandsinformation zu speichern.

Austausch von Cookies

Cookies werden in der Response vom Server im “Set-Cookie” oder “Set-Cookie2” (RFC2965) Header gesetzt. Sie bestehen aus Segmenten, die durch Semikolon voneinander getrennt sind. Erstes Segment ist ein Name “=” Wert Paar, das den Namen des Cookies festlegt.

In weiteren Requests sendet der Client das Cookie im “Cookie” Header an den Server zurück, sofern im Cookie enthaltene Bedingungen erfüllt sind.

Der Server kann über das Cookie verschiedene Requests demselben Client zuordnen.

Felder im Cookie

- **Comment**: Für Menschen lesbarer Kommentar zum Cookie
- **Domain**: Domain (oder ein Suffix beginnend mit .), für den das Cookie gilt.
- **Max-Age**: Nach dieser Zeit in Sekunden soll der Client das Cookie nicht mehr benutzen.
- **Path**: Pfad (oder ein Präfix), für den das Cookie gilt. Nur für URIs mit einem Pfad unterhalb des Attributes wird das Cookie benutzt.
- **Secure**: Versende das Cookie nur bei ausreichend hoher Sicherheit (z.B. bei verschlüsselter Verbindung).
- **Version**: Version des Cookies (verbindlich = 1 für RFC2109, nicht benutzt für Netscape)

RFC2965 benutzt einige weitere Felder. Version ist ebenfalls 1.

Verarbeitung auf Clientseite

- Ist **Domain** nicht angegeben, wird der FQDN der URI benutzt.
- Ist **Max-Age** nicht angegeben, wird das Cookie bei Beenden des Clients verworfen.
- Der Standardwert für **Path** ist der Path der aktuellen URI bis zum letzten “/”.
- Ein fehlendes **Secure** wird als nicht gesetzt angenommen.

Cookies werden vom Client verworfen, wenn

- **Path** kein Präfix des aktuellen Path ist.
- der Host nicht zur **Domain** gehört.
- **Domain** keinen eingeschlossenen Punkt enthält.
- Falls das Präfix, das zusammen mit **Domain** den FQDN des Hosts ergibt, einen Punkt enthält.

Beispiel

```
GET http://www.google.de/ HTTP/1.0
Host: www.google.de


HTTP/1.0 200 OK
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie:
PREF=ID=112381d8e7dc6b1e:TM=1197301885:...;
  path=/; domain=.google.de
Via: 1.0 localhost.localdomain:3128
(squid/2.6.STABLE17)
Proxy-Connection: close
Connection: close

...HTML Text...
```

Transport Layer Security

Entstehung von RSA

- Veröffentlicht 1977 von Ron Rivest, Adi Shamir und Leonard Adleman
- US Patent von 1983 bis 2000
- 1982 haben die Autoren RSA Data Security gegründet
- 1995 wurde Digital Certificates International (heute Verisign) von RSA Mitarbeitern gegründet.
- RSA wurde 2006 von EMC für ca. 2 Milliarden Dollar übernommen

Verschlüsselung und Entschlüsselung

- Das Verfahren benutzt drei Zahlen N, E, D.
- Das Paar (E, N) bildet den öffentlichen Schlüssel.
- Das Paar (D, N) bildet den privaten Schlüssel.
- Wer den öffentlichen Schlüssel kennt, kann eine Zahl m zwischen 0 und $N - 1$ verschlüsseln, indem er

$$C = m^E \mod N$$

berechnet.

- Wer den privaten Schlüssel kennt, kann C entschlüsseln durch

$$m = C^D \mod N.$$

Implementation

Beispielimplementation in Python:

```
def pot(a,b,n):
  """compute a^b mod n"""
  if b == 0: return 1
  return (1,a)[b&1]*pot(a*a % n, b » 1, n) % n


def crypt(v, (k, N)):
  """RSA crypt v using key (k,N)"""
  return pot(v, k, N)
```

Schlüsselerzeugung

- Bestimme zwei Primzahlen P und Q.
  Große Primzahlen lassen sich z.B. mit dem Miller-Rabin Primzahltest finden.
- Setze $N := P \cdot Q$ und $\Phi(N) := (P-1)(Q-1)$
- Bestimme ein E, so daß $\gcd(E, \Phi(N)) = 1$.
  Oft werden für E die Werte 3, 17 oder 65537 verwendet. Das sind sog. Fermat Primzahlen ($2^k + 1$ für ein k), da das die Verschlüsselung beschleunigt.
- Bestimme D, so daß $D \cdot E \mod \Phi(N) = 1$.
  Der erweiterte euklidische Algorithmus bestimmt zu zwei natürlichen Zahlen a, b zwei ganze Zahlen x, y, so daß $xa + yb = \gcd(a, b)$ gilt. Mit $a := E$ und $b := \Phi(N)$ liefert x den Wert für D, da $\gcd(E, \Phi(N)) = 1$.

Implementation

Erweiterter euklidischer Algorithmus in Python:

```
def ext_gcd(a,b):
  """extended greatest common divisor"""
  q, rem = divmod(a,b)
  if rem == 0: return (0,1)
  x,y = ext_gcd(b, rem)
  return (y, x-y*q)

>>> ext_gcd(17, 3120)
(-367, 2)
```

Beispiel 1

- Seien $P = 61$, $Q = 53$, damit ist $N = 3233$ und $\Phi(N) = 3120$.
- Wähle $E = 17$, dann ist $D = -367 \mod 3120 = 2753$
- Damit Public Key $(17, 3233)$ und Private Key $(2753, 3233)$.
- Verschlüsselung der Nachricht 15 ergibt dann $15^{17} \mod 3233 = 3031$.
- Entschlüsselung von 3031 ergibt wieder $3031^{2753} \mod 3233 = 15$.

Beispiel 2

- Seien $P = 3$, $Q = 11$, damit $N = 33$ und $\Phi(N) = 20$.
- Wähle $E = 3$, dann ist $D = 7$
- Damit Public Key $(3, 33)$ und Private Key $(7, 33)$.
- Da $N = 33$ sehr klein, muß hier die Nachricht (hier P, nicht P) sehr klein (<33) sein: Kodierung A-Z nach 1-26

Plaintext (P)		Ciphertext (C)			After decryption	
Symbolic	Numeric	P^3	P^3 (mod 33)	C^7	C^7 (mod 33)	Symbolic
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	01	1	1	1	01	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	05	125	26	8031810176	05	E

Sender's computation

Receiver's computation

(c) Tanenbaum, Computer Networks

Sicherheit von RSA

- Gelingt es einem Angreifer, N im Public Key zu faktorisieren, kann er den Private Key berechnen.
- Bei ungeschickter Wahl von P und Q ist es einfach, N zu faktorisieren.
- Es ist nicht bekannt, ob es keine effizienten Algorithmen zur Faktorisierung gibt.
- Es ist nicht bekannt, ob RSA nur durch Faktorisieren von N angegriffen werden kann.

Digitale Signatur

Um Integrität und Authentizität eines Dokumentes nachweisen zu können, kann es digital signiert werden.

Derjenige, der die Echtheit beurkunden soll, ergänzt das Dokument um seine eigene Identität (beglaubigt durch).

Aus den Daten berechnet er einen Hashwert nach einem vorher festgelegten und veröffentlichten Verfahren (z.B. mit MD5, SHA1, SHA256,...).

Diesen Wert verschlüsselt er mit seinem Private Key und hängt das Ergebnis als Signatur an das Dokument an.

Jeder kann nun die Echtheit des Dokumentes überprüfen, indem er mit dem Public Key die Signatur verschlüsselt und das Ergebnis mit dem Hash vergleicht.

Schlüsselaustausch

Alice und Bob wollen Daten so austauschen, daß ein Angreifer (Mallory) den Inhalt nicht mitlesen kann.

- Symmetrische Verschlüsselungsverfahren erfordern, daß Alice und Bob im voraus Schlüssel ausgetauscht haben.

TLS benutzt zum Schlüsselaustausch ein asymmetrisches Verschlüsselungsverfahren wie z.B. RSA

- Alice sendet ihren Public Key an Bob
- Bob verschlüsselt mit dem Public Key von Alice einen temporären Schlüssel
- Jetzt kennen Alice und Bob einen gemeinsamen Schlüssel, nicht aber Mallory.

Zertifikate

Problem beim Schlüsselaustausch ist, daß die Identität der beteiligten Parteien sichergestellt werden muß. Im Beispiel heißt das, daß Bob feststellen können muß, daß der gesendete Public Key tatsächlich Alice gehört.

Da sich in einem Rechnernetz die Teilnehmer nicht gegenseitig ausweisen können, basiert die Authentifizierung bei TLS auf vertrauenswürdigen Parteien.

Eine vertrauenswürdige Partei erhält eine Datei mit dem Public Key des Servers und dessen Identifikationsmerkmalen (z.B. FQDN). Diese Daten werden geprüft und die Datei digital signiert. Dadurch entsteht ein Zertifikat.

Chain of Trust, vgl. RFC2693

Die Zertifikate bilden eine hierarchische Struktur von Vertrauensbeziehungen.

- An der Spitze der Hierarchie stehen die Root Certificate Authorities. Deren Zertifikate sind von ihnen selbst signiert. Eine Liste der Zertifikate ist in vielen TLS Clients und Servern als vertrauenswürdig vorkonfiguriert.
- Erhält ein Knoten ein Zertifikat, prüft er
  1. die Identifikationsinformationen der Gegenseite.
  2. die Signatur des Zertifikates.
- Um die Signatur prüfen zu können, benötigt er das Zertifikat der Instanz, die signiert hat.
- Dies wird fortgeführt, bis ein vertrauenswürdiges Zertifikat erreicht ist.

Kommentar zu SSL/TLS

Bruce Schneier schreibt in “Secrets and Lies”, S. 239:

> *As it is used, with the average user not bothering to verify certificates exchanged and no revocation mechanism, SSL is just simply a (very slow) Diffie-Hellman key-exchange method. Digital certificates provide no actual security for electronic commerce; it’s a complete sham.*

Schneier, Bruce: Secrets and Lies(2000): Digital Security in a Networked World, Wiley Computer Publishing, ISBN 0-471-25311-1.

Abstrakte Syntax Notation (ASN)

Kodierung von Zertifikaten

Ausschnitt eines Zertifikates im PEM Format (RFC 1422)

```
---BEGIN CERTIFICATE---
MIIDEzCCAnygAwIBAgIBATANBgkqhkiG9w0BAQQFADCBxDELMAk
WkExFTATBgNVBAgTDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBxMJQ2F
...
---END CERTIFICATE---
```

Das Zertifikat ist gemäß X.509 DER kodiert, dann zum Transport Base64 kodiert.

Inhalt des Zertifikates

```
#openssl x509 -in Thawte_Server_CA.pem -noout -text
Certificate:
  Data:
     Version: 3 (0x2)
     Serial Number: 1 (0x1)
     Signature Algorithm: md5WithRSAEncryption
     Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte ...
     Validity
        Not Before: Aug 1 00:00:00 1996 GMT
        Not After : Dec 31 23:59:59 2020 GMT
        Subject: C=ZA, ST=Western Cape, L=Cape Town, ...
     Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
        RSA Public Key: (1024 bit)
           Modulus (1024 bit):
              00:d3:a4:50:6e:c8:ff:56:6b:e6:cf:5d:b6:ea:0c:
              ...
           Exponent: 65537 (0x10001)
     X509v3 extensions:
        X509v3 Basic Constraints: critical
           CA:TRUE
  Signature Algorithm: md5WithRSAEncryption
     07:fa:4c:69:5c:fb:95:cc:46:ee:85:83:4d:21:30:8e:ca:d9:

     ...
```

ASN.1 Basistypen

ASN.1 (Abstrakte Syntax Notation, X.680ff, vgl. `http://www.larmouth.demon.co.uk/books/ASN.1complete.PDF`) ist eine Beschreibungssprache für Datentypen und Daten. Sie heißt “abstrakt”, da die Beschreibung der Datentypen keine Angaben über deren Kodierung macht. Komplexe Datentypen können aus Basistypen konstruiert werden.

Basistypen (Simple Types) sind

- Strings (BIT STRING, OCTET STRING, IA5 String, ...)
- Aufzählungen (ENUMERATED)
- Boolsche Werte (BOOLEAN)
- Zahlen (INTEGER, REAL)
- Objekbezeichner (OBJECT IDENTIFIER)
- Nulltyp (NULL)

Zusammengesetzte Datentypen

Um komplexe Datentypen aus den Basistypen bilden zu können, werden folgende zusammengesetzten Datentypen (Component Types) angeboten:

- **CHOICE**: einer der folgenden Typen wird für Daten benutzt (varianter Record / union)
- **SET**: (ungeordnete) Menge von Daten möglicherweise verschiedener Typen
- **SET OF**: (ungeordnete) Menge von Daten eines Typs
- **SEQUENCE**: (Geordnete) Folge von Daten möglicherweise verschiedener Typen
- **SEQUENCE OF**: (Geordnete) Folge von Daten eines Typs

Tagging

Einem Datentyp kann zur Identifikation ein Tag zugewiesen werden. Es werden folgende Klassen unterschieden:

- **Universal**: Standarddatentypen, die in jedem Protokoll verwendet werden dürfen.
- **Application**: Tag wird innerhalb einer Applikation verwendet.
- **Context Specific**: Die Bedeutung hängt ab vom umschließenden Datentyp.
- **Private**: Datentypen innerhalb z.B. einer Organisation.

Tagging erfolgt in der Regel automatisch mit Context Specific Tags. Es können bei Bedarf eigene Tags in der Form [tag] festgelegt werden. Soll ein Datentyp an anderer Stelle mit anderem Tag wiederverwendet werden, kann das Tag mit IMPLICIT überschrieben werden.

Syntax der ASN.1 Definition

Die vollständige Grammatik von ASN.1 findet sich in der X.680 Spezifikation.

- Bezeichner beginnen mit einem Kleinbuchstaben und enden nicht mit einem Minus
- Typen beginnen mit einem Großbuchstaben und enden nicht in einem Minus
- Kommentare beginnen mit “–” und enden mit Zeilenende oder weiterem “–”. Für mehrzeilige Kommentare wird C Notation benutzt “/*”, “*/”
- Formatierung spielt keine Rolle, Leerzeichen, horizontale Tabs und Zeilenumbrüche können durch ein einzelnes Leerzeichen ersetzt werden.
- Zuweisungen und Definitionen erfolgen mit “::=”

Übersicht

Für ASN.1 Datentypen sind verschiedene konkrete Kodierungen standardisiert.

- **BER**: Basic Encoding Rules (X.690)
- **DER**: Distinguished Encoding Rules (X.690), wie BER, aber die Kodierung ist eindeutig.
- **PER**: Packed Encoding Rules (X.691), extrem kompakte Darstellung
- **XER**: XML Encoding Rules (X.693), Darstellung der Daten als XML Dokument
- **GSER**: Generic String Encoding Rules (RFC3641), in LDAP verwendet.

Ferner gibt es die Möglichkeit, in ASN.1 spezifizierte Datentypen in andere Spezifikationen zu übersetzen (z.B. CORBA IDL mittels `snacc`).

Kodierregeln für BER / DER

BER ist eine Tag-Length-Value Kodierung. Das Tag legt fest, wie der Wert interpretiert werden muß. Die Klasse des Tags wird in die höchstwertigen Bits des Tags kodiert:

Bit 8	Bit 7	Klasse
0	0	Universal
0	1	Application
1	0	Context Specific, Default
1	1	Private

Typ der Kodierung

Man unterscheidet zwei Typen der Kodierung, primitive und zusammengesetzte Kodierung.

- **Primitive Kodierung** kann benutzt werden, wenn Basistypen oder Typen, die aus Basistypen durch implizite Tags erzeugt wurden, kodiert werden. Die Anzahl Byte des kodierten Wertes muß bekannt sein.
- **Zusammengesetzte Kodierung** muß benutzt werden, wenn primitive Kodierung nicht möglich ist. Bei Strings sind beide Kodierungen möglich. Zusammengesetzte Kodierung ermöglicht, einen String als Folge von Teilen zu übertragen.

Kodierung Tags

Das Tag eines BER kodierten Wertes ist aus Bit 7 und Bit 8 für die TAG Klasse, Bit 6 für den Type der Kodierung (primitiv = 0, zusammengesetzt = 1) und Bit 1 bis 5 für Tags mit den Werten 0 bis 30.
Für Tags mit Wert > 30 sind Bit 1-5 des ersten Bytes alle 1.
Der Wert des Tags wird mit 7 Bit pro Byte kodiert, wobei das höchstwertige Bit nur für das letzte Byte 0 ist.

Beispiel: (Context Specific) INTEGER
mit Tag $1234567 = 75 \cdot 128^2 + 45 \cdot 128 + 7$
Wegen $75 + 128 = CB_{16}, 45 + 128 = AD_{16}$
und $10011111_2 = 9F_{16}$
ist der Kode: `9F CB AD 07`.

Länge

Die Länge eines Datentyps gibt die Anzahl der Bytes des kodierten Wertes an.

BER: Für Werte ≤ 127 kann die Länge in einem Byte kodiert werden, dessen höchstwertiges Bit 0 ist, die übrigen 7 Bit geben die Länge des Wertes an.
Für größere Längen und alternativ für alle Längen kann eine Länge Wert Darstellung verwendet werden. Dabei ist das höchstwertige Bit des ersten Bytes 1, die anderen Bits geben die Anzahl Längenbytes an. Dann folgen die Längenbytes.

DER: Ist die Länge zwischen 0 und 127 einschließlich, muß die Länge in einem Byte kodiert werden.

Unbestimmte Länge

Ist die Länge bei Start der Kodierung des Wertes nicht bekannt, kann sie als unbestimmt angegeben werden. Dazu wird eine Länge ohne Längenbytes verwendet. Das Ende einer solchen Sektion wird durch 2 Bytes mit Wert 0 als Tag und Länge angezeigt.

Beispiel:

Wert	Kodierregel	Kode
12	BER / DER	`0C`
12	BER	`81 0C`, `82 00 0C`
129	BER/DER	`81 81`
?	unbestimmt	`80` TLV TLV TLV `00 00`

Universal Tags ausgewählter Datentypen

Tag	Type	Tag	Type
1	BOOLEAN	18	NumericString
2	INTEGER	19	PrintableString
3	BIT STRING	20	T61String
4	OCTET STRING	22	IA5String
5	NULL	23	UTCTime
6	OBJECT IDENTIFIER	24	GeneralizedTime
9	REAL	27	GeneralString
10	ENUMERATED	28	UniversalString
12	UTF8String	29	CHAR STRING
13	RELATIVE-OID	30	BMPString
14	TIME (1)	31	DATE (1)
16	SEQUENCE [OF]	32	TIME-OF-DAY (1)
17	SET [OF]	33	DATE-TIME (1)

BIT STRING

ASN.1: BIT STRING
String Typ, der beliebige Bitfolgen halten kann. Der String hat variable Länge, Länge 0 ist möglich.

BER: Primitive oder zusammengesetzte Kodierung
Primitive Kodierung: Der Bitstring wird auf eine Bytegrenze vervollständigt. Das erste Byte des Inhalts gibt die Anzahl Füllbits an, danach folgen die Bytes des Bitstrings.
Zusammengesetzte Kodierung: Kodierung als Folge von Substrings

DER: Ausschließlich primitive Kodierung und Füllbit 0 ist erlaubt.

Beispiel BIT STRING

Der String `01101110 11110111 11` wird kodiert:

- BER primitiv: `03 04 06 6E F7 EA`, Füller `101010`
- BER primitiv mit Länge in Langform: `03 81 04 06 6E F7 EA`
- DER primitiv: `03 04 06 6E F7 C0`, Füller `000000`
- BER zusammengesetzt: 16 Bit + 2 Bit, Füller `101010`

  `23 09`

  `03 03 00 6E F7`

  `03 02 06 EA`

CHOICE

ASN.1: CHOICE { [id_1] $Type_1$, . . ., [id_n] $Type_n$ }
Die optionalen Bezeichner id_1, . . ., id_n haben nur Kommentarfunktion.

BER: Werte werden kodiert, wie der gewählte Typ. Die Unterscheidung, welcher Eintrag genutzt wird, erfolgt über das Tag.

DER: Werte werden kodiert, wie der gewählte Typ. Die Unterscheidung, welcher Eintrag genutzt wird, erfolgt über das Tag.

IA5String

ASN.1: IA5String
String mit Zeichen des 7Bit ASCII Alphabets. Die Länge ist variabel, Länge 0 ist erlaubt.

BER: Primitive oder zusammengesetzte Kodierung
Primitive Kodierung: Die Daten des Strings
Zusammengesetzte Kodierung: BER Kodierung der Substrings

DER: Primitive BER Kodierung

Beispiel: Kodierung von `test`
primitive BER oder DER Kodierung: `16 04 74 65 73 74`
zusammengesetzte BER Kodierung:

```
36 0X
  16 02 74 65
  16 02 73 74
```

INTEGER

ASN.1: INTEGER [{ $\mathrm{id}_1(\mathrm{value}_1)$, . . ., $\mathrm{id}_n(\mathrm{value}_n)$ }]
Zahlen beliebiger Größe, id_1, . . ., id_n erzeugen symbolische Namen für value_1, . . ., value_n.

BER/DER: Primitive Kodierung, Big Endian, negative Zahlen im 2er Komplement

Beispiel:

Wert	BER/DER Kodierung
0	02 01 00
127	02 01 7F
128	02 02 00 80
-128	02 01 80
-129	02 02 FF 7F

NULL

ASN.1: NULL
Fester Wert NULL

BER: Primitive Kodierung mit Länge 0:
`05 00` oder `05 81 00`

DER: Nur `05 00` ist erlaubt.

OBJECT IDENTIFIER

ASN.1: OBJECT IDENTIFIER
Die Werte sind Folgen von nicht negativen Zahlen, geschrieben entweder als
$\{ a_1, a_2, \ldots, a_n \}$ (ITU) oder $a_1.a_2.\ldots.a_n$ (IETF). Die Verwaltung und Zuweisung ist in ITU-T Rec. X.660 geregelt.

BER/DER: a_1 und a_2 werden als $40 \cdot a_1 + a_2$ kodiert, was zu höchstens einem Byte führt. Die weiteren Werte werden mit 7 Bit pro Byte kodiert, wobei das höchstwertige Bit des Bytes anzeigt, daß ein weiteres Byte für den Wert folgt.

Beispiel: `1.3.6.1.4.1.3495.1.2` (squid Cache Config MIB für SNMP)
$1 \cdot 40 + 3 = 43 = 2B_{16}, 3495/128 = 27$ Rest 39, damit ist die Kodierung: `06 09 2B 06 01 04 01 9B 27 01 02`

SEQUENCE

ASN.1: SEQUENCE {
[id_1] $Type_1$ [{ OPTIONAL | DEFAULT $value_1$}], . . .,
[id_n] $Type_n$ [{ OPTIONAL | DEFAULT $value_n$}] }

BER: Zusammengesetzte Kodierung, Folge der BER Kodierungen der Elemente. Wird ein Wert, der als OPTIONAL oder mit DEFAULT angegeben ist, nicht geliefert, ist er im Kode nicht enthalten.
Wird für einen Wert der DEFAULT angegeben, kann er im Kode enthalten sein.

DER: Wird für einen Wert der DEFAULT angegeben, wird der Wert nicht kodiert.

SEQUENCE OF

ASN.1: SEQUENCE OF Type

BER/DER: Die Daten enthalten die BER/DER Kodierung der Folge der Daten in der Reihenfolge des Auftretens.

Tags

BER/DER: Bei IMPLICIT Tags wird das angegebene Tag mit dem Wert und der Kodierung (primitiv/zusammengesetzt) des Basistyps verwendet. Andernfalls wird eine zusammengesetzte Kodierung erzeugt mit dem vollständigen TLV (Type-Length-Value) Kode des Basistyps.

Beispiel:
Type1 ::= VisibleString
Type2 ::= [APPLICATION 3] IMPLICIT Type1
Type3 ::= [2] Type2
Type4 ::= [APPLICATION 7] IMPLICIT Type3
Type5 ::= [2] IMPLICIT Type2

Tags: Beispiel

Kodierung des Wortes “Jones” (vgl. X690)

Type	Tag	Länge	Wert
Type1	1A	05	4A6F6E6573
Type2	43	05	4A6F6E6573
Type3	A2	07	43 05 4A6F6E6573
Type4	67	07	43 05 4A6F6E6573
Type5	82	05	4A6F6E6573

SET

ASN.1: SET {
[id$_1$] Type$_1$ [{ OPTIONAL | DEFAULT value$_1$}], . . .,
[id$_n$] Type$_n$ [{ OPTIONAL | DEFAULT value$_n$}] }

BER: Zusammengesetzte Kodierung, Folge der BER Kodierungen der Elemente. Wird ein Wert, der als OPTIONAL oder mit DEFAULT angegeben ist, nicht geliefert, ist er im Kode nicht enthalten.
Wird für einen Wert der DEFAULT angegeben, kann er im Kode enthalten sein.

DER: Wird für einen Wert der DEFAULT angegeben, wird der Wert nicht kodiert. Die Werte erscheinen in der Reihenfolge der Tags.

SET OF

ASN.1: SET OF Type
BER: Zusammengesetzte Kodierung, Folge der BER Kodierungen der Elemente.

DER: Wie BER. Die Werte werden in aufsteigender lexikographischer Sortierung kodiert. Zum Vergleich werden zwei Werte mit 0 Bytes auf gleiche Länge von rechts aufgefüllt.