

Entstehung

- ▶ 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

$$C = m^E \pmod{N}$$

berechnet.

- ▶ Wer den privaten Schlüssel kennt, kann C entschlüsseln durch

$$m = C^D \pmod{N}.$$

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 $\text{gcd}(E, \phi(N)) = 1$.
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 $D \cdot E = 1 \pmod{\phi(N)}$.
Der erweiterte euklidische Algorithmus bestimmt zu zwei natürlichen Zahlen a, b zwei Zahlen x, y , so daß $xa + yb = \text{gcd}(a, b)$ gilt. Mit $a := E$ und $b := \phi(N)$ liefert x den Wert für D , da $\text{gcd}(E, \phi(N)) = 1$.

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

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

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.

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, OCTETSTRING, IA5AString, ...)
- ▶ Aufzählungen (ENUMERATED)
- ▶ Boolesche 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 eine Länge ohne Längenbytes verwendet. Das Ende einer solchen Sektion wird durch 2 Bytes mit Wert 0 als Tag und Länge angezeigt.

	Wert	Kodierregel	Kode
	12	BER / DER	0C
Beispiel:	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 { [id₁] Type₁, . . . , [id_n] Type_n }

Die optionalen Bezeichner id₁, . . . , 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 $\{ \{ id_1(value_1), \dots, id_n(value_n) \} \}$

Zahlen beliebiger Größe, id_1, \dots, id_n erzeugen symbolische Namen für $value_1, \dots, value_n$.

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

$\{ a_1, a_2, \dots, a_n \}$ (ITU) oder $a_1.a_2.\dots.a_n$ (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 {
[id₁] Type₁ [{ OPTIONAL | DEFAULT value₁ }], ... ,
[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 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 {
[id₁] Type₁ [{ OPTIONAL | DEFAULT value₁ }], ... ,
[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.