Zusammenfassung PKCS Dokumente 1, 5 & 8

1 PKCS #1

Das PKCS Dokument #1 gibt Empfehlungen zur Implementierung von Public Key Kryptografie auf Basis des RSA Algorithmus. Dabei werden hauptsächlich die folgenden Aspekte abgedeckt, welche nachfolgend genauer erläutert werden:

- Kryptografische Primitive
- Verschlüsselungs Schemas (Encryption schemes)
- Signierungs Schemas (Signature schemes)

1.1 Key Typen

Bei RSA existieren 2 unterschiedliche Key Typen, welche zusammen als *RSA Key Pair* bezeichnet werden:

- *RSA Public Key* wird zum verschlüsseln bzw . verifizieren benutzt.
- *RSA Private Key* wird zum entschlüsseln bzw. signieren benutzt.

1.2 RSA Public Key

Der Public Key besteht aus zwei Komponenten, nämlich einem Modulo und einem öffentlichen Exponenten (Public exponent). Beides sind positive Integer.

n = RSA Modulo. Der Modulo ist ein Produkt von mindestens 2 Primzahlen. In Lehrbüchern und Beispielen werden häufig nur 2 Primzahlen multipliziert, es können aber auch mehr verwendet werden. $(r_1, r_2, \ldots, r_n)$ r_1 und r_2 werden häufig auch als p und q bezeichnet

e = Public exponent. Zahl zwischen 3 und n-1, welche relativ Prim (teilerfremd) zu $(r_1 - 1) \times (r_2 - 1) \times (r_n - 1)$ ist

Der Einfachheit halber wird in dieser Zusammenfassung immer davon ausgegangen, dass bei der Generierung von n nur 2 Primzahlen verwendet wurden.

1.3 RSA Private Key

Der Private Key besteht aus dem selben Modulo n wie der Public Key. Zusätzlich existiert ein privater Exponent d:

n = RSA Modulo (siehe RSA Public Key)

d = Private Exponent. $e \times d \equiv 1 \, mod(p-1)(q-1)$. d ist also das Inverse von e $mod(p-1)(q-1$

Bemerkung: Würden mehr als 2 Primzahlen verwendet werden, würde der Private Key noch einige Komponenten mehr enthalten.

1.4 Kryptografische Primitive

Kryptografische Primitive bezeichnen grundlegende mathematische Operationen auf denen die Kryptografischen Schemas aufbauen. Es werden 4 Typen von Primitiven spezifiziert:

- Verschlüsselung & Entschlüsselung
- Signierung und Verifikation

1.4.1 Verschlüsselungs und Entschlüsselungs Primitive

Ein Verschlüsselungsprimitiv erstellt unter Verwendung des Public Keys aus einer Nachricht ein Chiffrat. Das Entschlüsselungsprimitiv gewinnt unter Verwendung des Private Keys wieder die Nachricht aus dem Chiffrat.
RSA definiert die beiden Schemes RSAEP und RSADP (RSA Encryption/Decryption Primitiv). Beide verwenden die selben mathematischen Operationen, wobei einfach unterschiedlicher Input verwendet wird

RSAEP

Input	(n,e)	Public Key
	m	Nachricht
Output	c	Chiffrat

RSADP

Input	(n,d)	Private Key
	c	Chiffrat
Output	m	Nachricht

Für die genaue Berechnung sei auf PKCS #1 Kapitel 5.1[1] verwiesen

1.4.2 Signatur und Verifikation Primitive

Das Signatur Primitiv erstellt unter Verwendung des Private Keys aus einer Nachricht[2] eine Signatur. Das Verifikations Primitiv gewinnt unter Verwendung des Public Keys aus der Signatur wieder die Nachricht. Die Primitiven RSASP1 und RSAVP1 (RSA Signature/Verification Primitives) funktionieren gleich wie RSADP und RSAEP, mit dem einzigen Unterschied, dass die Input und Output Argumente unterschiedlich sind.

RSASP1

Input	(n,d)	Private Key
	m	Nachricht
Output	s	Signatur

RSAVP1

Input	(n,e)	Public Key
	s	Signatur
Output	m	Nachricht

1.5 Schemas

Ein Schema kombiniert kryptografische Primitive mit anderen Techniken um ein bestimmtes Schutzziel zu erreichen. In PKCS #1 werden Verschlüsselungs und Signatur Schemas spezifiziert. Die Schemas definieren nur die Schritte welche unternommen werden um die Daten zu verarbeiten. Es wird z.B. keine Schlüssel generiert oder validiert.

---

1 ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf

2 Wie später noch gezeigt wird, muss es sich bei der Nachricht nicht um den gleichen Text handeln wie bei der Verschlüsselung. Es kann z.B. auch der Hashwert der verschlüsselten Nachricht sein.

1.6 Verschlüsselungs Schemas

Ein Verschlüsselungs Schema besteht aus einer Verschlüsselungs- und einer Entschlüsselungs Operation. Ein solches Schema kann für verschiedene Zwecke verwendet werden. Häufig wird es verwendet um z.B. einen symmetrischen Schlüssel auszutauschen. In PKCS #1 werden zwei unterschiedliche Schemas spezifiziert. RSAES-OAEP und RSAES-PKCS1-v1_5. Für neuere Applikationen sollte nur noch ersteres verwendet werden, deshalb wird das letztere hier nicht weiter berücksichtigt.

Grundsätzlich wird bei den Schemas zuerst ein Encoding der Nachricht durchgeführt. Die codierte Nachricht wird dann in eine Integerversion der Nachricht konvertiert. Auf diese Integerversion der Nachricht wird anschliessend die Verschlüsselungs Primitve angewendet um das Chiffrat zu erstellen. Genau umgekehrt funktioniert das Verfahren um wieder den Klartext zu erhalten.

1.6.1 RSAES-OAEP

RSAES-OAEP kombiniert die RSAEP/RSADP Primitive mit der EME-OAEP Encoding Methode. OAEP steht für Optimal Asymmetric Encryption Padding und dient dazu das Kryptosystem gegen Chosen-Plaintext Attacken zu schützen in dem bei jeder Nachricht ein Zufalls-Padding vorgenommen wird. Dieses sehr wichtige Padding wird im Schulbuch RSA selten erwähnt.
Um den Rahmen dieses Dokumentes nicht zu sprengen, wird im folgenden nur die Verschlüsselung beschrieben. Die Entschlüsselung funktioniert nach einem ähnlichen Prinzip.

Grobes Verfahren:

1. Länge der Nachricht prüfen
2. Encoding vornehmen (Details siehe EME-OAEP)
3. Konvertieren in Integer Version, mithilfe von OS2IP (Octet String to Integer Primitive)
4. Anwenden der RSAEP Primitive

1.6.1.1 EME-OAEP

1. Generieren des Hashwertes[3] lHash des Labels L (welches in dieser Version der Spezifikation leer ist. Somit wird hier immer der gleiche Hashwert verwendet, welcher jedoch natürlich abhängig von der verwendeten Hashfunktion ist)
2. Generieren eines Null Oktet Strings PS der Länge *k - mLen - 2hLen*
3. Zusammenfügen von lHash, PS, einem Oktet 0x01 und der Nachricht M. Wird zusammen als data block DB bezeichnet.
4. Generieren eines Random Strings seed
5. Anwenden einer MGF[4] auf dem seed. Output = dbMask
6. maskedDB = DB xor dbMask
7. Anwenden einer MGF auf maskedDB. Output = seedMask
8. maskedSeed = seed xor seedMask
9. Das Oktet 0x00, maskedSeed und maskedDB bilden zusammen die codierte Nachricht.

---

[3] In PKCS #1 werden für RSAES-OAEP lediglich die Hashfunktionen SHA-1 und SHA-256/384/512 empfohlen.

[4] MGF = Mask Generation Functions werden hier nicht näher erläutert. Für Details sei auf das Kapitel B.2 in PKCS #1 verwiesen

k	Länge des Modulo n
mLen	Länge der Nachricht
hLen	Länge des Hashwertes

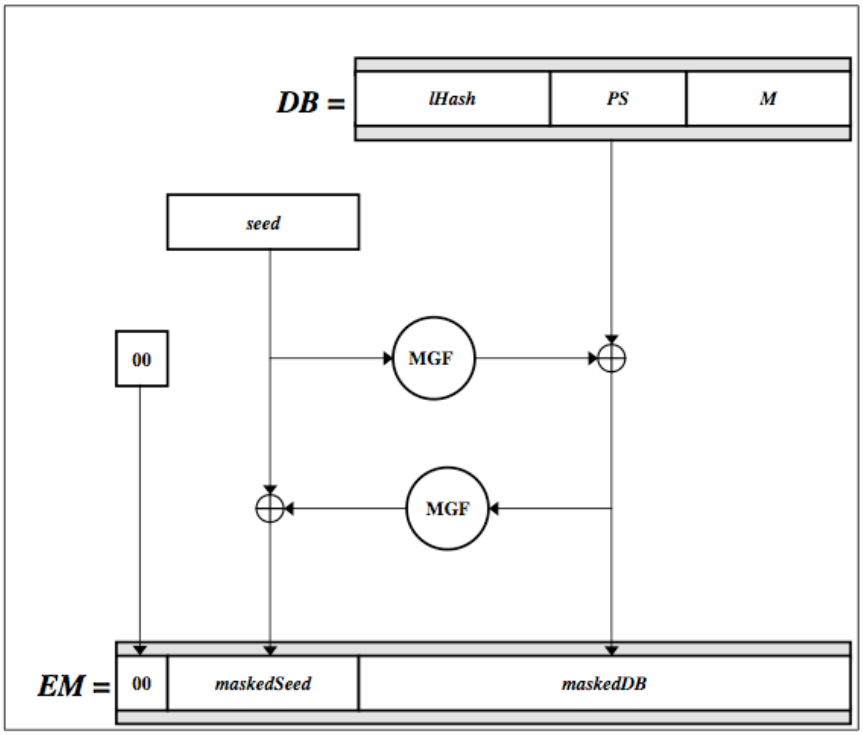

Abbildung 1 Grafische Ansicht des EME-OAEP. *Quelle: PKCS #1*

1.7 Signatur Schemas

Ein Signatur Schema besteht aus einer *signature generation operation* und einer *signature verifcation operation*. Die Erstere wird von der unterzeichnenden Partie mit Hilfe seines privaten Schlüssels verwendet um eine Signatur zu erstellen. Letztere wird von der prüfenden Partie mit Hilfe des öffentlichen Schlüssels der unterzeichnenden Partie verwendet um die Signatur zu überprüfen. Die in PKCS #1 spezifizierten Schemas werden auch als Signatur Schemas mit Anhang bezeichnet, da sie die Nachricht selbst ebenfalls benötigen um die Signatur zu verifizieren. Im Gegensatz dazu existieren auch Signatur Schemas mit Möglichkeit zur Nachricht-Wiederherstellung.

In PKCS#1 werden zwei Schemas spezifiziert. RSASSA-PSS und RSASSA-PKCS1-v1_5. Für neuere Applikationen wird RSASSA-PSS empfohlen, weshalb in diesem Dokument nur auf dieses eingegangen wird.

1.7.1 RSASSA-PSS

RSASSA-PSS kombiniert die Primitive RSASP1/RSAVP1 mit dem EMSA-PSS Encoding (Auf EMSA-PSS wird in diesem Dokument nicht näher eingegangen). RSASSA-PSS ist probabilistisch da ein zufallsgeneriertes Salt eingefügt wird (PSS = Probabilistic Signature Scheme). Die Sicherheit beruht jedoch nicht auf diesem Salt und kann immer noch als gewährleistet betrachtet werden, wenn ein solcher Salt nicht generiert werden kann.

1.7.1.1 Signature generation operation

Input	K	Private Key der signierenden Partie
	M	Nachricht
Ouput	S	Signatur

Schritte:

1. Anwenden des Encodings EMSA-PSS-ENCODE
2. Konvertieren der codierten Nachricht in Integer (OS2IP)
3. Anwenden von RSASP1 Primitive mit Hilfe von K und M
4. Konvertieren Integer Version in Oktet-String (I2OSP)
5. Output von Signatur S

1.7.1.2 Signature verification operation

Input	(n,e)	Public Key der signierenden Partie
	M	Nachricht
	S	Signatur
Ouput	valide/invalide Signatur	

Schritte:

1. Prüfung der Signatur Länge (Muss gleich Lang wie der Modulo n sein)
2. Konvertieren der Signatur S in Integer (OS2IP)
3. Anwenden der RSAVP1 Primitive mit Hilfe vom Public Key (n,e) und der Nachricht M.
4. Konvertieren in Oktet-String (I2OSP)
5. Verifikation mittels EMSA-PSS-VERIFY
6. Ouput valide/invalide Signatur

Bei der Verifikation wird zuerst der Salt wiederhergestellt, danach der Hashwert erneut berechnet und mit dem Hashwert welcher in der Signatur enthalten ist verglichen. Bei Übereinstimmung ist die Signatur valide.

Folgende Grafik illustriert das EMSA-PSS-ENCODING

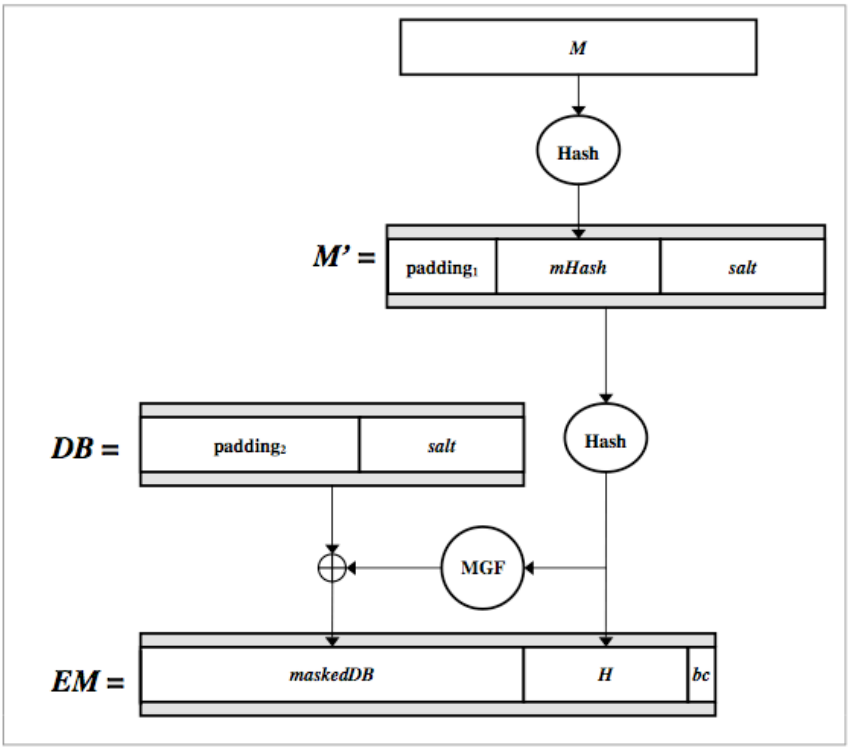

Abbildung 2 EMSA-PSS-ENCODING. *Quelle: PKCS #1*

2 PKCS #5

PKCS Dokument #5 gibt Empfehlungen für die Implementierung von Passwort basierter Kryptografie. Passwörter sind nicht direkt als Schlüssel verwendbar und müssen deshalb angepasst werden, um als solchen verwendet werden zu können. Bei dieser Umwandlung in einen Schlüssel muss auch beachtet werden, dass Passwörter meist Teil von einem kleineren Raum sind. Die Anzahl verschiedener Werte ist also viel kleiner als z.B. bei richtigen Schlüssel.
Ein Ansatz zur Passwort basierten Kryptografie ist die Verwendung eines sogenannten Salt um einen Schlüssel zu produzieren. Der Salt ist ein zufälliger Wert welcher dem Passwort angehängt wird und somit verhindert, dass aus einem sogenannten Dictionary von bereits berechneten Hashwerten das Passwort ausgelesen werden kann.
Ein anderer Ansatz ist die Verwendung einer Funktion zur Ableitung eines Passworts, welche unterschiedlich oft ausgeführt wird (abhängig vom sogenannten Wiederholungszähler)
Die in PKCS #5 beschriebenen Methoden implementieren beide Ansätze, dementsprechend ist die Passwort basierte Schlüssel Ableitung eine Funktion von einem Passwort, einem Salt und dem Wiederholungszähler.
In PKCS #5 wird die Verwendung dieser Ableitung in Zusammenhang mit Verschlüsselung und Message Authentication definiert obwohl auch andere Verwendungszwecke denkbar wären (z.B. bei der Speicherung von Passwörtern).

2.1 Salt und Wiederholungszähler

Der Salt dient dazu für ein einzelnes Passwort eine grosse Menge von möglichen Keys generieren zu können. Bei einem 64 Bit langen Salt gibt es also für jedes Passwort 2^{64} verschiedene Keys. Zudem ist es unwahrscheinlich, dass für das selbe Passwort zwei mal derselbe Key generiert wird.

Der Wiederholungszähler dient dazu die Erstellung des Keys rechenaufwändiger zu machen, und somit die Schwierigkeit einer Attacke zu erhöhen. Es wird eine Anzahl von mindestens 1000 Iterationen empfohlen.

Salt und Wiederholungszähler müssen nicht geheim sein!

2.2 Schlüsselableitungsfunktion PBKDF2

Für neuere Applikationen wird PBKDF2 empfohlen, weshalb hier die PBKDF1 ausser Acht gelassen wird. Grob zusammengefasst funktioniert PBKDF2 folgendermassen.

1. Der abgeleitete Schlüssel wird in einzelne Blöcke unterteilt
2. Für jeden Block wird eine Pseudozufalls Funktion mehrere Male ausgeführt (Anzahl gemäss Wiederholungszähler) und mit dem vorherigen Resultat der Funktion XORed. Input der Funktion bei der ersten Ausführung ist das Passwort, der Salt sowie die Nummer des Blocks (siehe Schritt 1). Bei allen weiteren Ausführungen das Passwort und das Resultat der letzten Iteration.
3. Am Schluss werden die einzelnen Blöcke zusammengesetzt und das ist dann der abgeleitete Schlüssel.

Als Pseudozufallsfunktionen kommen HMAC-SHA-1 oder HMAC-SHA-2 in Frage.

2.3 Verschlüsselungs Schema PBES2

Eine typische Anwendung von einem **P**assword **B**ased **E**ncryption **S**cheme (wie PBES2[5]) ist der Schutz von Private Keys, also von Nachrichten welche Private Key Informationen enthalten.
PBES2 kombiniert die Schlüsselableitungsfunktion PBKDF2 mit einem Verschlüsselungs-Schema (z.B. AES-CBC-Pad)

Etwas vereinfacht werden bei Ver- und Entschlüsselung folgende Schritte durchgeführt:

Verschlüsselung

1. Wählen von Salt und Wiederholungszähler
2. Erstellen des Schlüssels mit Hilfe von PBKDF2
3. Verschlüsseln der Nachricht mit z.B. AES-CBC-Pad

Entschlüsselung

1. Erlangen von Salt und Wiederholungszähler
2. Erstellen des Schlüssels mit Hilfe von PBKDF2
3. Entschlüsseln der Nachricht mithilfe des abgeleiteten Schlüssel aus Schritt 2

Wie man sieht, wird mit dem gleichen Schlüssel ver- und entschlüsselt (schliesslich ist AES ja auch ein symmetrisches Verfahren).

2.4 Message Authentication Schema PBMAC1

Eine Message Authentication Schema besteht aus einer MAC Erzeugungs-Operation und einer MAC Verifikations-Operation. Dabei wird (anders als bei einer Signatur) die Erzeugung sowie die Verifikation des MACs(Message Authentication Code) mit dem selben Schlüssel vorgenommen.
PBMAC1 kombiniert die Schlüsselableitungsfunktion PBKDF2 mit einem Message Authentication Schema. Namentlich sind das HMAC-SHA-1 oder HMAC-SHA-2, welche beide auf den SHA-1 respektive SHA-224, SHA-256, SHA-384 oder SHA-512 Hash Funktionen basieren.

Die beiden Operationen werden hier etwas vereinfach beschrieben:

PBMAC1 Erzeugungsoperation

1. Wählen von Salt und Wiederholungszähler
2. Erstellen des Schlüssels mit Hilfe von PBKDF2
3. Erstellen des MAC mithilfe der erzeugten Schlüssels und dem zugrunde liegenden Message Authentication Schemas (z.B. HMAC-SHA-2)

PBMAC1 Verifikationsoperation

1. Erlangen von Salt und Wiederholungszähler
2. Erstellen des Schlüssels mit Hilfe von PBKDF2
3. Erstellen des MAC mithilfe der erzeugten Schlüssels und dem zugrunde liegenden Message Authentication Schemas (z.B. HMAC-SHA-2)
4. Vergleichen des eben erzeugten MACs und des zu verifizierenden MACs
5. Bei Übereinstimmung ist der Output "correct", ansonsten "incorrect"

---

[5] Neben PBES2 existiert auch noch PBES1, welches jedoch nicht mehr verwendet werden soll.

3 PKCS #8

Der Standard in PKCS #8 beschreibt die Syntax um Private Key Informationen zu speichern. Dies kann sowohl unverschlüsselt als auch verschlüsselt geschehen (z.B. mit den in PKCS #5 beschriebenen Algorithmen). Wie in allen PKCS Dokumenten wird zur Beschreibung der Syntax die ASN.1[6] verwendet

3.1 Private Key Information Syntax

Die Syntax für den ASN.1 Typ PrivateKeyInfo:

```
PrivateKeyInfo ::= SEQUENCE {
  version Version,
  privateKeyAlgorithm PrivateKeyAlgorithmIdentifier,
  privateKey PrivateKey,
  attributes [0] IMPLICIT Attributes OPTIONAL }
Version ::= INTEGER
PrivateKeyAlgorithmIdentifier ::= AlgorithmIdentifier
PrivateKey ::= OCTET STRING
Attributes ::= SET OF Attribute
```

Bedeutung der Felder:

version	Syntax Versionsnummer dient der Kompatibilität von zukünftigen Überarbeitungen dieses Standards. Für diese Version des Standards ist sie 0
privateKeyAlgorithm	Bezeichnet der verwendete Algorithmus. Z.B rsaEncryption aus PKCS #1
privateKey	Dies ist der Private Key
attributes	Eine Menge von Attributen, wie sie z.B. in PKCS #9 definiert sind.

3.2 Verschlüsselte Private Key Information Syntax

Die Syntax für den ASN.1 Typ EncryptedPrivateKeyInfo:

```
EncryptedPrivateKeyInfo ::= SEQUENCE {
  encryptionAlgorithm EncryptionAlgorithmIdentifier,
  encryptedData EncryptedData }
EncryptionAlgorithmIdentifier ::= AlgorithmIdentifier
EncryptedData ::= OCTET STRING
```

Bedeutung der Felder:

encryptionAlgorithm	Bezeichnet den verwendeten Algorithmus um die Private Key Information zu verschlüsseln (Bsp. aus PKCS #5: pbeWithMD5AndDES-CBC)
encryptedData	Die verschlüsselte Private Key Information (PrivateKeyInfo aus 3.1)

[6] Abstract Syntax Notation One: http://www.itu.int/ITU-T/asn1/