Sichere Programmierung
Projekt 4

David Pierre Sugar
(76050)
Julian Sobott
(76511)

Inhaltsverzeichnis

1 Einleitung **1**

2 Aufgabe 1: Kryptographische Hashfunktion **2**

3 Aufgabe 2: Symmetrische Verschlüsselung **2**

3.1 Cipher 2

3.2 Implementierung 2

3.2.1 Key erstellen 3

3.2.2 Initialisierungs-Vektor 3

3.2.3 Cipher 3

3.3 Verschlüsselung 3

3.3.1 Ausgabe Cipher Text 3

3.3.2 Entschlüsselung 4

3.3.3 Ausgabe Plain Text 4

3.4 Beispielausgabe 4

4 Aufgabe 3 Asymmetrische Verschlüsselung **4**

5 Aufgabe 4: Digitale Signatur **5**

5.1 Verfahren 5

5.2 Implementierung 5

1 Einleitung

In diesem Praktikum geht es darum Konzepte, die in der Vorlesung *Einführung in die IT-Sicherheit* erklärt wurden, in Java zu implementieren. Hierzu wird die Kryptographie-API von Java verwendet.

2 Aufgabe 1: Kryptographische Hashfunktion

In dieser Aufgabe soll die Erzeugung und Verifikation einer Prüfsumme implementiert werden. Als Hashalgorithmus soll hier `SHA-256` verwendet werden. Für die Aufgabe wurden zwei statische Methoden implementiert. `public static byte[] createChecksum(String message)` und `public static boolean verifyChecksum(byte[] checksum, String message)`. Die erste erzeugt eine Prüfsumme und die zweite verifiziert ob ein String dieselbe Prüfsumme erzeugt.

Fürs Erzeugen wird eine `MessageDigest` Instanz, welche mit `SHA-256` arbeitet, geholt. Mit dieser wird dann, der String in einen entsprechenden Hash Wert umgewandelt.

Fürs Verifizieren, wird auf dieselbe Weise von dem neuen String die Checksum erstellt. Ob die beiden gleich sind, wird mit der statischen Methode `MessageGigest.isEqual()` überprüft.

Zur demosntration der beiden Methoden, wurden in der `main` Methode 2 Strings erstellt. Die Verifikation gibt true zurück, wenn beide Strings gleich sind und false, wenn sie unterschiedlich sind.

3 Aufgabe 2: Symmetrische Verschlüsselung

Symmetrische Verschlüsselung ist ein Weg, Nachrichten vor unerwünschten Einblicken zu schützen. Symmetrisch heißt dabei, dass der selbe Key zum ver-/ und wieder endschlüsseln verwendet wird. Ein Beispiel hierfür wäre der `AES` (Advanced Encryption Standard) Algorithmus, der auch `Rijndael` Algorithmus genannt wird, in Anlehnung an die Erfinder Vincent Rijmen und Joan Daemen. Symmetrische Verschlüsselungen bieten, einen angemessen großen Key vorrausgesetzt, guten Schutz und eine schnelle Ver-/ Entschlüsselung, jedoch muss zur sicheren Übermittlung auf andere Verschlüsselungsverfahren wie RSA zurückgegriffen werden, da diese sonst leicht von Dritten abgegriffen werden können.

3.1 Cipher

Der AES ist ein Block Cipher, d.h. es werden jeweils Blöcke zu je `128 Bit` verschlüsselt, dabei werden die `Key-Längen 128, 192 und 256 Bit` unterstützt. Die Länge des Keys gibt dabei die Anzahl an Transformationsrunden vor, die für die Verschlüsselung durchlaufen werden. Um den Text wieder zu entschlüsseln, werden die Runden rückwärts durchlaufen. Die Transformationen finden dabei auf einer 4x4 Byte Matrix statt.

3.2 Implementierung

Das Beispiel besteht darin, einen String zu verschlüsseln, den Cipher Text auf Standard Out auszugeben, den Cipher Text wieder zu entschlüsseln und auch den entschlüsselten Text auf der Kommandozeile auszugeben. Sollte schlussendlich der anfangs verschlüsselte Text richtig ausgegeben werden, so kann davon ausgeganen werden, dass die symmetrische Verschlüsselung mittels AES richtig ausgeführt wurde.

3.2.1 Key erstellen

Als erstes wird ein zufälliger Key erstellt, der dann zusammen mit dem AES Cipher verwendet werden soll. Dafür wird die Methode `getAESKey()` aufgerufen, die einen `AES Key Generator` instanziert und danach mittels `init()` diesen Key Generator mit einer Key Größe von 256 initialisiert. Danach wird mit `generateKey()` der zufällige Key erstellt und zurück gegeben.

```
1 SecretKey key = getAESKey();
```

3.2.2 Initialisierungs-Vektor

Ein Initialisierungsvektor ist eine Zufallszahl, die als weitere Sicherheit in die Verschlüsselung mit eingebracht wird. Um das Brechen des Ciphers, z.B. mittels Dictionary Attack, zu erschweren wird der vorrausgegangene Cipher Block mit genutzt, um den derzeitigen Plaintext Block zu verschlüsseln. Da für den ersten Plaintext Block noch kein Cipher Block zur verfügung steht, der verwendet werden kann wird eine Zufallszahl, der IV, zur Verschlüsselung hinzugezogen.

Dieser IV wird von `getIvSpec()` zurückgegeben.

```
1 IvParameterSpec ivSpec = getIvSpec();
```

Dabei nutzt die Methode einen Pseudo-Zufallszahl-Generator um ein Byte Array zu befüllen. Dieses Array wird dann als Argument genutzt, um mit `IvParameterSpec()` ein IV Objekt zu erzeugen.

3.2.3 Cipher

Um den Cipher zu initialisieren, wurde die Methode `initCipher()` implementiert. Dies erhällt `Key` und `IV`, sowie den Modus (ENCRYPT, DECRYPT) als Parameter und instanziert ein AES Cipher Objekt, das dann zur verschlüsselung verwendet werden kann.

```
1 Cipher cipher = initCipher(key, Cipher.ENCRYPT_MODE, ivSpec);
```

3.3 Verschlüsselung

Mittels `encrypt()` kann nun der Plain Text verschlüsselt werden. Dazu werden Cipher sowie Text and die Methode übergeben, zurück kommt der Cipher Text als Byte Array. Intern wird einfach nur `doFinal()` auf dem Cipher Objekt aufgerufen.

```
1 byte cipherText[] = encrypt(cipher, pt);
```

3.3.1 Ausgabe Cipher Text

Die Methode `printCipherText()` erhällte das Byte Array als Argument und gibt es in `Base 64` codiert auf der Kommandozeile aus.

```
1 printCipherText(cipherText);
```

3.3.2 Entschlüsselung

Nun muss der Text wieder entschlüsselt werden. Dazu wird das Cipher Objekt nun mit dem `DECRYPT` Mode initialisiert, dabei bleiben Key und IV die selben. Danach wird `decrypt()` aufgerufen. Die Methode ruft erneut `doFinal()` auf und entschlüsselt damit den übergebenen Cipher Text, zurück in den Ausgangstext.

```
1 cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);
2 String new_pt = decrypt(cipher, cipherText);
```

3.3.3 Ausgabe Plain Text

Schlussendlich wird nun auch der entschlüsselte Plain Text auf der Kommandozeile ausgegeben.

```
1 System.out.println("Plain Text: " + new_pt);
```

3.4 Beispielausgabe

```
1 [sugar@jellyfish Code]$ java SymEncrypt
2 Cipher Text: lPSGD5vFeGS7KlTBZjX2RQ==
3 Plain Text: Neo... Wake up!
4 [sugar@jellyfish Code]$ java SymEncrypt
5 Cipher Text: OG5fVtza0u6P5RybQLLMGA==
6 Plain Text: Neo... Wake up!
7 [sugar@jellyfish Code]$ java SymEncrypt
8 Cipher Text: +wCZzQ9bfxIA9r1aGFjB8Q==
9 Plain Text: Neo... Wake up!
```

4 Aufgabe 3 Asymmetrische Verschlüsselung

Diese Aufgabe soll die Nutzung des `RSA` Algorithmus für die asymmetrische Verschlüsselung zeigen. Ein Text soll mit einem public key verschlüsselt werden, und mit dem entsprechendem private key wieder entschlüssselt werden.

Hierzu wird als erstes mithilfe eines `KeyPairGenerator` ein Schlüsselpaar für die Verschlüsselung mit RSA erstellt. Auf die beiden Schlüssel kann dann mithilfe von gettern zugegriffen werden.

Als nächstes wird ein Objekt, welches für die Ver- und Entschlüsselung verantwortlich ist, mithilfe der factory Methode `getInstance` geholt. Dieses kann mehrmal verwendet werden, muss aber vor jeder Verwendung initialisiert werden. Hierbei muss übergeben werden, ob ver- oder entschlüsselt werden soll, und der jeweilige passende key. Mit der `doFinal` Methode, kann dann der ver-/entschlüsselungs Prozess abgeschlossen werden. Hierbei wird immer ein `byte[]` zurückgegebn, welches den Text enthält. Um den verschlüsselten Text schöner auf der Konsole anzuzeigen, wird er noch in Base64 umgewandelt. Außerdem wird das `byte[]` an den Konstruktor der Klasse `String` übergeben, um den Text als zusammenhängenden Text und nicht als Array auszugeben.

5 Aufgabe 4: Digitale Signatur

Digitale Signaturen werden genutzt, um die Authentizität von digitalen Dokumenten sicher zu stellen, gewährleisten aber in keinem Fall die Vertraulichkeit eines Dokuments. Zur digitalen Signatur eignen sich dabei die meisten asymmetrischen Verschlüsselungssysteme, wie z.B. RSA. RSA ist dabei aber anfällig für Attacken, sollte der selbe Text mehrmals mit dem Selben Schlüssel signiert werden.

5.1 Verfahren

Anfangs wird eine zufälliges Schlüsselpaar, bestehend aus Public-Key und Private-Key, erstellt. Mittels RSA und unter Verwendung des Private-Key wird nun das gewünschte Dokument verschlüsselt. Jeder mit Zugang zum Public-Key kann nun diesen verwenden um die Nachricht wieder zu entschlüsseln. Sollte dabei ein sinnvoller und lesbarer Text herauskommen, kann davon ausgegangen werden, dass die Nachricht tatsächlich vom Author verfasst wurde, da dieser der Einzige mit Kenntniss vom Private-Key ist. Besonders für große Dateien ist dieses Verfahren jedoch ineffizient, in diesem Fall kann auf Hashing zurückgegriffen werden. Dabei wird das Dokument gehashed und der Hash verschlüsselt. Zur Verifikation kann der Hashwert wieder entschlüsselt und mit dem eigens, über dem Dokument erzeugten, Hashwert verglichen werden.

5.2 Implementierung

Das Beispiel besteht darin, einen Text mittels RSA digital zu signieren. Dieser wird dann auf der Konsole ausgegeben. Danach wird er wieder entschlüsselt um zu überprüfen ob das Dokument authentisch ist. Das Ergebnis wird ebenfalls wieder auf der Kommandozeile ausgegeben. Erwarte wird, dass der Text identisch zu dem ursprünglichen Plain-Text ist.