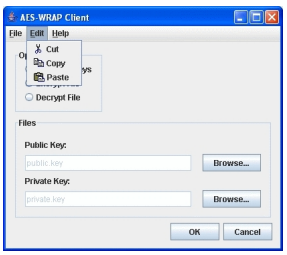
Einführung

Verschlüsselungen wurden von den Menschen schon im frühen Zeitalter verwendet. Sei es im alten Rom der Cäsar-Chiffre, der nach Gaius Julius Cäsar benannt wurde, welcher sie zur verschlüsselten Kommunikation seiner militärischen Korrespondenz verwendete oder die neuen krypotgrafischen Verfahren, wie RSA oder AES. Mit Verschlüsselung bezeichnet man den Vorgang, bei dem ein „Klartext“ mit Hilfe eines Verschlüsselungsverfahrens (Algorithmus) in einen „Geheimtext“ umgewandelt wird.

Verschlüsselungen sind immer dort erforderlich, wo vertrauliche Informationen übertragen bzw. ausgetauscht werden. Dies kann beispielsweise bei der Übertragung von Kreditkartennummer oder E-Mails der Fall sein. Bis vor Kurzem haben Patente und Exportbeschränkungen viele Firmen, einschließlich Sun, davon abgehalten, starke Verschlüsselungsverfahren anzubieten. Glücklicherweise wurden die Exportbeschränkungen mittlerweile gelockert und das Patent für einen wichtigen Algorithmus ist abgelaufen. Seit JDK 1.4 gehört nun eine gute Verschlüsselungsunterstützung zur Java Standardbibliothek.

Ich zeige euch in diesem Tutorial wie man unter Java sehr schnell eine Anwendung schreiben kann die es ermöglicht Daten sicher untereinander auszutauschen. Wir verwenden dazu eine besondere Methode in der ein gängiges symmetrisches Verschlüsselungsverfahren namens AES, mit einem Verfahren kombiniert wird das mit öffentlichen Schlüsseln arbeitet. Das folgende Bild zeigt das kleine Verschlüsselungsprogramm. Es wurde mit dem GUI Builder von NetBeans
erstellt und verwendet das JDK 1.5.



Verschlüsselungsverfahren

In der Regel unterscheidet man Verschlüsselungssysteme in zwei Kategorien. In symmetrische und asymmetrische Verschlüsselung. Je nach verwendetem System entscheidet sich wie der kryptografische Schlüssel an die am Verfahren Beteiligten vermittelt werden. Bei symmetrischen Systemen besitzen beide Kommunikationspartner denselben Schlüssel und müssen diesen vor Beginn der Kommunikation sicher ausgetauscht haben. Bekannte klassische symmetrische Verfahren sind die Cäsar-Chiffre, der AES und das One-Time-Pad. Zu

den modernen und derzeit als sicher angesehenen Verfahren gehören der Rijndael, Twofish sowie 3DES, wobei dem Rijndael durch seine Erhebung zum Advanced Encryption Standard (AES) und aufgrund seiner Bevorzugung durch staatliche US-amerikanische Stellen eine herausragende Rolle zukommt.

Asymmetrische Systeme zeichnen sich dadurch aus, dass für jeden Teilnehmer ein Schlüsselpaar generiert wird. Ein Schlüssel jedes Paars wird veröffentlicht, der andere bleibt geheim. Die Asymmetrie ergibt sich, weil ein Schlüssel eines Paars immer nur ver- und der andere immer nur entschlüsseln kann. Das bekannteste dieser Verfahren ist das RSA-Kryptosystem.

Kryptografie in Java

Die Java Cryptographic Extensions enthalten eine Klasse *Cipher*, die die Superklasse für alle Verschlüsselungsalgorithmen bildet. Um ein Verschlüsselungsobjekt zu erhalten, ruft man die Methode *getInstance* über die zwei möglichen Wege auf:

[code xml:lang="java"]Cipher cipher = Cipher.getInstance(algorithmName); Cipher cipher = Cipher.getInstance(algorithmName, providerName);[/code]

Zum JDK gehören Verschlüsselungsverfahren von einem Provider namens "SunJCE". Dieser Standardprovider wird verwendet, wenn Sie keinen anderen Providernamen spezifizieren. Ein anderer Provider ist dann erforderlich, wenn ein spezieller Algorithmus verwendet werden soll, der von Sun nicht unterstützt wird. Der Algorithmusname wird durch einen String wie zum Beispiel "AES" oder "RSA" festgelegt.

Der DES wurde aufgrund seiner unzureichenden Sicherheit und des in die Jahre gekommenen Algorithmuses mittlerweile vom AES abgelöst. Beide Algorithmen sind Blockverschlüsselungsverfahren. Die Blockgröße des AES beträgt 128 Bit und ist im Gegensatz zum normalen Rijndael fest implementiert, kann also nicht verändert werden.

Nachdem das Verschlüsselungsobjekt in Java erzeugt wurde, kann es initialisiert werden. Dazu muss der Modus und der Schlüssel festgelegt werden.

[code xml:lang="java"]int mode = ...; Key key = ...; cipher.init(mode, key);[/code]

Java stellt verschiedene Konstanten für den Modus zur Verfügung. Dazu gehören Cipher.ENCRYPT_MODE, Cipher.DECRYPT, Cipher.UNWRAP_MODE und

Cipher.WRAP_MODE. Die letzten beiden Modi werden verwendet um einen Schlüssel mit einem anderen Schlüssel zu verschlüsseln. Man spricht auch von umhüllen oder wrappen. Nachdem der Modus festgelegt wurde, wird damit begonnen die Datenblöcke zu verschlüsseln. Dies erfolgt über den Aufruf der Methode *update*.

[code xml:lang="java"]int blockSize = cipher.getBlockSize(); byte[] inBytes = new byte[blockSize]; ... // inBytes lesen int outputSize = cipher.getOutputSize(inLength); byte[] outBytes = new byte[outputSize]; int outLength = cipher.update(inBytes, 0, outputSize, outBytes); ... // write outBytes[/code]

Abschließend wird die Methode *doFinal* aufgerufen.

[code xml:lang="java"]outBytes = cipher.doFinal(); // If there is more data in this block use outBytes = cipher.doFinal(inBytes, 0, inLength);[/code]

Der Aufruf von doFinal ist erforderlich, um den letzten Block aufzufüllen. Um zu verstehen warum das notwendig ist, muss man wissen wie Blockverschlüsselungsverfahren funktionieren. Sehen wir und dazu die DES-Verschlüsselung an. DES verschlüsselt die Eingabedaten in Byteblöcken einer fest definierten Blockgröße. Die Blockgröße beträgt 8 Bytes also 64 Bit. Wir gehen davon aus, das der letzte Block mit Eingabedaten weniger als 8 Bytes umfasst. Eine Möglichkeit den Rest des Blocks voll zu bekommen, wäre es diesen einfach mit 0 zu füllen und diesen anschließend zu verschlüsseln. Jedoch würde man nach dem Entschlüsseln mehrere nachgestellte 0-Bytes erhalten, wodurch sich die Eingabedatei und Ausgabedatei leicht unterscheiden würden. Dieses Problem wird mithilfe eines bestimmten Auffüllschemas gelöst. Beim so genannten PKCS wird der Block mit einem Füllwert aufgefüllt der gleich der Anzahl der Füllbytes ist. Auf diese Weise kann später nachvollzogen werden mit wieviel Bytes der Block aufgefüllt wurde.

Neben der Verschlüsselung der Eingabedaten ist der eigentliche Schlüssel von entscheidender Bedeutung. Bei symmetrischen Verfahren ist ein Schlüssel sowohl für die Verschlüsselung als auch für die Entschlüsselung zuständig. Um einen Schlüssel zu generieren muss per KeyGenerator eine neue Schlüsselinstanz erzeugt werden. Der Schlüssel selbst wird mithilfe des Zufallsgenarators erzeugt. Alternativ kann auch bei variablen Verschlüsselungsalgorithmen einfach die Blocklänge übergeben werden. Anschließend wird die Methode *generateKey* aufgerufen.

[code xml:lang="java"]KeyGenerator keygen = KeyGenerator.getInstance("AES"); SecureRandom random = new SecureRandom(); keygen.init(random); Key key = keygen.generateKey();[/code]

Der Schlüssel lässt sich aus einem beliebigen Seed erzeugen. So kann auch ein eigenes Kennwort verwendet werden, oder ein Schlüssel aus einer feststehenden Menge von Rohdaten.

[code xml:lang="java"]SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("AES");

byte[] keyData = ...; // 16 bytes for AES SecretKeySpec = new SecretKeySpec(keyData, "AES"); Key key = keyFactory.generateSecret(keySpec);[/code]

Symmetrische Verfahren mit asymmetrischen kombinieren

Das Problem mit symmetrischen Verschlüsselungsverfahren ist die Tatsache das der generierte Schlüssel für die Verschlüsselung selbst, als auch für die anschließende Entschlüsselung zuständig ist. Tauschen zwei Personen verschlüsselte Nachrichten untereinander aus, so müssen beide über denselben Schlüssel verfügen. Sobald eine Person den Schlüssel verändert, muss auch die zweite Person eine Kopie des veränderten Schlüssels erhalten. Verschlüsselte Nachrichten werden verwendet um Daten vor dem Zugriff Dritter zu schützen. Beim Austausch der Schlüssel muss demnach ein geschützter Kanal vorhanden sein. Doch eben dies ist meist nicht der Fall, weshalb die Daten bzw. Nachrichten auch verschlüsselt werden müssen.

Dieses Problem lässt sich mit asymmetrischen Verschlüsselungsverfahren beheben. Man spricht auch von Systemen die mit öffentlichen Schlüsseln arbeiten. Hier stehen zwei Schlüssel zur Verfügung. Ein öffentlicher (engl. public key) und ein privater (engl. private key) Schlüssel. Beide Schlüssel stehen zueinander in einer bestimmten mathematischen Beziehung, die es erlaubt, ein Chiffrat, das mit dem öffentlichen Schlüssel verschlüsselt wurde mit dem privaten Schlüssel zu entschlüsseln. Die Definition, ob ein Schlüssel privat oder öffentlich ist, hängt dabei nicht vom Schlüssel selbst ab, sondern nur davon welche Rolle man dem Schlüssel zuweist. Der wesentliche Punkt bei einem asymmetrischen Kryptosystem ist, dass eine verschlüsselte Nachricht nur mit dem anderen Schlüssel des Schlüsselpaares wieder entschlüsselt werden kann. Möchte man zum Beispiel eine geheime Nachricht per E-Mail empfangen kann man ein Schlüsselpaar generieren. Anschließend versendet man den öffentlichen Schlüssel an alle Personen, die einem eine Nachricht schreiben wollen. Den privaten Schlüssel hält man geheim, denn nur mit ihm lassen sich die mit dem öffentlichen Schlüssel verschlüsselten Daten wieder entschlüsseln. Auf diese Weise ist sichergestellt das die Daten vor dem Zugriff Dritter geschützt sind. Für einen Angreifer ist es nahezu unmöglich in einer vertretbaren Zeitspanne aus einem hinreichend langen Schlüssel den korrespondierenden Zweitschlüssel zu generieren. Die mathematische Grundlage für diesen Umstand bildet die Faktorisierung von Primzahlen. Während die Multiplikation von zwei Primzahlen relativ wenig Rechenzeit in Anspruch nimmt, ist die Faktorisierung einer Primzahl deutlich komplexer und kann nicht in polynomieller Zeit gelöst werden. Die meisten Kryptosysteme, die mit öffentlichen Schlüsseln arbeiten, basieren auf der Faktorisierung von Ganzzahlen.

Ron Rivest, einer der Erfinder des bekannten asymmetrischen Kryptosystems RSA stellte 1977 die These auf, dass die Faktorisierung von 125 Bit eine Billiarde Jahre in Anspruch nehmen würde. Die sehr schnell wachsende Rechenleistung führte allerdings dazu das bereits 1994 ein 129 Bit langer Schlüssel erfolgreich faktorisiert werden konnte. 2003 gelang es mit

Supercomputern 576 Bit in wenigen Minuten zu faktorisieren. Aus diesem Grunde musste man die Schlüsselgröße kontinuierlich vergrößern. Derzeit (Stand 2011) gilt ein 2048 Bit langer RSA-Schlüssel als sicher. Nachfolgend ist das Prinzip von asymmetrischen Kryptosystemen abgebildet.

Ein asymmetrisches Kryptosystem hat offensichtlich einige Vorteile. Doch warum verwendet man dann nicht immer asymmetrische Verschlüsselungsverfahren? Ein Grund liegt im unpraktischen Umgang asymmetrischer Verfahren mit großen Datensätzen, da der Algorithmus deutlich langsamer arbeitet als der von symmetrischen Verfahren. Weiterhin spezifizieren viele bekannte asymmetrische Kryptosysteme nur die Verschlüsselung von wenigen Byte.

Abhilfe schafft eine Kombination von symmetrischen und asymmetrischen Kryptosystemen. Während die Daten selbst mit einem symmetrischen Verfahren verschlüsselt werden, wird der symmetrische Schlüssel mit dem öffentlichen Schlüssel der Gegenstelle verschlüsselt bzw. eingehüllt. Die Gegenstelle kann den symmetrischen Schlüssel anschließend mit dem eigenen privaten Schlüssel entschlüsseln und erhält somit den symmetrischen Schlüssel, um die eigentliche Nachricht zu entschlüsseln. Man spricht in diesem Fall auch von Hybridverfahren . Das hier vorgestellte Java Programm nutzt eine hybride Verschlüsselung, wie sie auch in IPsec und TLS/SSL verwendet wird. Die Methoden zur Verschlüsselung befinden sich in der Klasse Crypt der Datei Crypt.java. Mithilfe der Methode
generateKey
wird zunächt ein RSA Schlüsselpaar erzeugt.

[code xml:lang="java"]/** * Generate a RSA pair key (public.key, private.key) and stores * it in files. * * @param privateKeyFileName name of the private key * @param publicKeyFileName name of the public key */ public static void generateKey(String privateKeyFileName, String publicKeyFileName) { try { KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA"); SecureRandom random = new SecureRandom(); pairgen.initialize(KEYSIZE, random); KeyPair keyPair = pairgen.generateKeyPair(); ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(publicKeyFileName)); out.writeObject(keyPair.getPublic()); out.close(); out = new ObjectOutputStream(new FileOutputStream(privateKeyFileName)); out.writeObject(keyPair.getPrivate()); out.close(); } catch (IOException e) { new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage()); } catch (GeneralSecurityException e) { new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage()); } }[/code]

Das Tool stellt hierfür die Option Generate Keys zur Verfügung, die sowohl einen public.key als auch einen private.key erzeugt.

Anschließend kann eine Datei mit der Option [thumb src="images/tutorials/java/aes-crypto-pic1.gif" arg=";popup;link"]Encrypt[/thumb] verschlüsselt werden. Dazu ist der öffentliche Schlüssel erforderlich. Die Methode produziert eine Datei, die sowohl
die verschlüsselten Daten als auch den symmetrischen Schlüssel beinhaltet. Dazu schreibt die Methode die Länge des eingehüllten Schlüssels und zusätzlich die eingehüllten Schlüsselbytes in die Ausgabedatei. Auf diese Weise muss man nicht zwei einzelne Dateien verteilen, da die Nachricht und der symmetrische Schlüssel sich in derselben Datei befinden. Die ursprüngliche Datei, z.B. [thumb src="images/tutorials/java/aes-crypto-pic2.gif" arg=";popup;link"]Plain.txt[/thumb] wird verschlüsselt und eine [thumb src="images/tutorials/java/aes-crypto-pic3.gif" arg=";popup;link"]geheime Datei[/thumb] wird erzeugt.

```
[code xml:lang="java"]/**   * Encrypt a file with AES using the public RSA key.   *   * @param publicKeyFile     name of the public key   * @param inputFile         name of the input file   * @param outputFile        name of the output file   */      public static void encrypt(String publicKeyFile, String inputFile, String outputFile)  {      try {          KeyGenerator keygen = KeyGenerator.getInstance("AES");          SecureRandom random = new SecureRandom(); keygen.init(random);          SecretKey key = keygen.generateKey();           // Wrap with public key          ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(publicKeyFile));          Key publicKey = (Key) keyIn.readObject(); keyIn.close();           Cipher cipher = Cipher.getInstance("RSA"); cipher.init(Cipher.WRAP_MODE, publicKey);          byte[] wrappedKey = cipher.wrap(key); DataOutputStream out = new DataOutputStream(new FileOutputStream(outputFile)); out.writeInt(wrappedKey.length);          out.write(wrappedKey);          InputStream in = new FileInputStream(inputFile);          cipher = Cipher.getInstance("AES"); cipher.init(Cipher.ENCRYPT_MODE, key);          crypt(in, out, cipher);          in.close(); out.close();      } catch (IOException e) {          new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage());      } catch (GeneralSecurityException e) {          new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage());      } catch (ClassNotFoundException e) {          new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage());      } }[/code]
```

Zuletzt muss es noch eine Methode geben die die verschlüsselte Datei auch wieder entschlüsselt. Die Option [thumb src="images/tutorials/java/aes-crypto-pic4.gif" arg=";popup;link"]Decrypt File[/thumb] lokalisiert und enthüllt den symmetrischen Schlüssel mittels cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY); in der Datei. Dazu wird der private Schlüssel benötigt. Die Methode heißt sinngemäß *decrypt*.

```
[code xml:lang="java"]/**   * Decrypt a file with the private key.   *   * @param privateKeyFile
```

* @param inputFile * @param outputFile */ public static void decrypt(String privateKeyFile, String inputFile, String outputFile) { try { DataInputStream in = new DataInputStream(new FileInputStream(inputFile)); int length = in.readInt(); byte[] wrappedKey = new byte[length]; in.read(wrappedKey, 0, length); // Open with private key ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(privateKeyFile)); Key privateKey = (Key) keyIn.readObject(); keyIn.close(); Cipher cipher = Cipher.getInstance("RSA"); cipher.init(Cipher.UNWRAP_MODE, privateKey); Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY); OutputStream out = new FileOutputStream(outputFile); cipher = Cipher.getInstance("AES"); cipher.init(Cipher.DECRYPT_MODE, key); crypt(in, out, cipher); in.close(); out.close(); } catch (IOException e) { new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage()); } catch (GeneralSecurityException e) { new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage()); } catch (ClassNotFoundException e) { new MessageBox().doOkCancelDialog("Exception", "Message:nn" + e.getMessage()); } } [/code]

Abschließend wird die Nachricht mit dem enthüllten symmetrischen Schlüssel entschlüsselt und [thumb src="images/tutorials/java/aes-crypto-pic5.gif" arg=";popup;link"]ausgegeben[/thumb].

Das war eine kleine Einführung in die Anwendung von Verschlüsselungen in Java. Im Downloadbereich findet sich neben dem Code zum Verschlüsseln von Dateien auch Code zum Verschlüsseln von gewöhnlichen Byte-Arrays in Java. Ich hoffe ich konnte ein wenig zum Verständnis der beiden derzeit gängigsten symmetrischen und asymmetrischen Kryptoalgorithmen beitragen. Bei Unklarheiten kann man im Forum nachfragen. Ein kurzer Blick ins untere AUSWAHLMENÜ reicht.