JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

Java Security

Systemtechnik Labor
5BHIT 2016/17, Gruppe B

Daniel May

Version 1.0
Note:
Begonnen am 07. Oktober 2016
Betreuer: Thomas Micheler
Beendet am 17. Oktober 2016

Inhaltsverzeichnis

1 Einführung

Diese Übung zeigt die Anwendung von Verschlüsselung in Java.

1.1 Ziele

Das Ziel dieser Übung ist die symmetrische und asymmetrische Verschlüsselung in Java umzusetzen. Dabei soll ein Service mit einem Client einen sicheren Kommunikationskanal aufbauen und im Anschluss verschlüsselte Nachrichten austauschen. Ebenso soll die Verwendung eines Namensdienstes zum Speichern von Informationen (hier PublicKey) verwendet werden.

Die Kommunikation zwischen Client und Service soll mit Hilfe einer Übertragungsmethode (IPC, RPC, Java RMI, JMS, etc) aus dem letzten Jahr umgesetzt werden.

1.2 Voraussetzungen

- Grundlagen Verzeichnisdienst
- Administration eines LDAP Dienstes
- Grundlagen der JNDI API für eine JAVA Implementierung
- Grundlagen Verschlüsselung (symmetrisch, asymmetrisch)
- Einführung in Java Security JCA (Cipher, KeyPairGenerator, KeyFactory)
- Kommunikation in Java (IPC, RPC, Java RMI, JMS)
- Verwendung einer virtuellen Instanz für den Betrieb des Verzeichnisdienstes

1.3 Aufgabenstellung

Mit Hilfe der zur Verfügung gestellten VM wird ein vorkonfiguriertes LDAP Service zur Verfügung gestellt. Dieser Verzeichnisdienst soll verwendet werden, um den PublicKey von einem Service zu veröffentlichen. Der PublicKey wird beim Start des Services erzeugt und im LDAP Verzeichnis abgespeichert. Wenn der Client das Service nutzen will, so muss zunächst der PublicKey des Services aus dem Verzeichnis gelesen werden. Dieser PublicKey wird dazu verwendet, um den symmetrischen Schlüssel des Clients zu verschlüsseln und im Anschluss an das Service zu senden.

Das Service empfängt den verschlüsselten symmetrischen Schlüssel und entschlüsselt diesen mit dem PrivateKey. Nun kann eine Nachricht verschlüsselt mit dem symmetrischen Schlüssel vom Service zum Client gesendet werden.

Der Client empfängt die verschlüsselte Nachricht und entschlüsselt diese mit dem symmetrischen Schlüssel. Die Nachricht wird zuletzt zur Kontrolle ausgegeben.

Die folgende Grafik soll den Vorgang verdeutlichen:

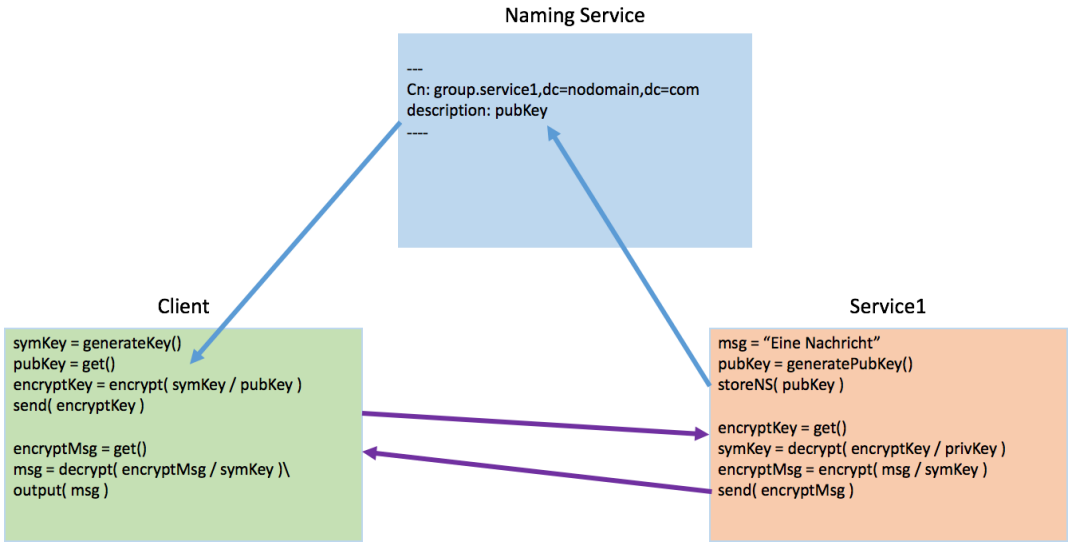

Abbildung 1: Kombinierte Verschlüsselung

Gruppengröße: 1 Person

Bewertung: 16 Punkte

- asymmetrische Verschlüsselung (4 Punkte)
- symmetrische Verschlüsselung (4 Punkte)
- Kommunikation in Java (3 Punkte)
- Verwendung eines Naming Service, JNDI (3 Punkte)
- Protokoll (2 Punkte)

Links:

- Java Security Overview
  `https://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html`
- Security Architecture
  `https://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc.html`
- Java Cryptography Architecture (JCA) Reference Guide
  `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`

Read the Java Security Documentation and focus on following Classes: KeyPairGenerator, SecureRandom, KeyFactory, X509EncodedKeySpec, Cipher

2 Ergebnisse

2.1 Designüberlegungen

Grundsätzlich wurden beim Server und Client die Aufgaben in Kryptographie und Übertragung geteilt. Somit gibt es insgesamt 4 Klassen, wobei die Netzwerkobjekte schlussendlich die `main`-Funktion enthalten und auf die Verschlüsselungsfunktionalitäten der `SecureXXX` Klassen zurückgreifen. Das Klassendiagramm sieht letztendlich wie folgt aus:

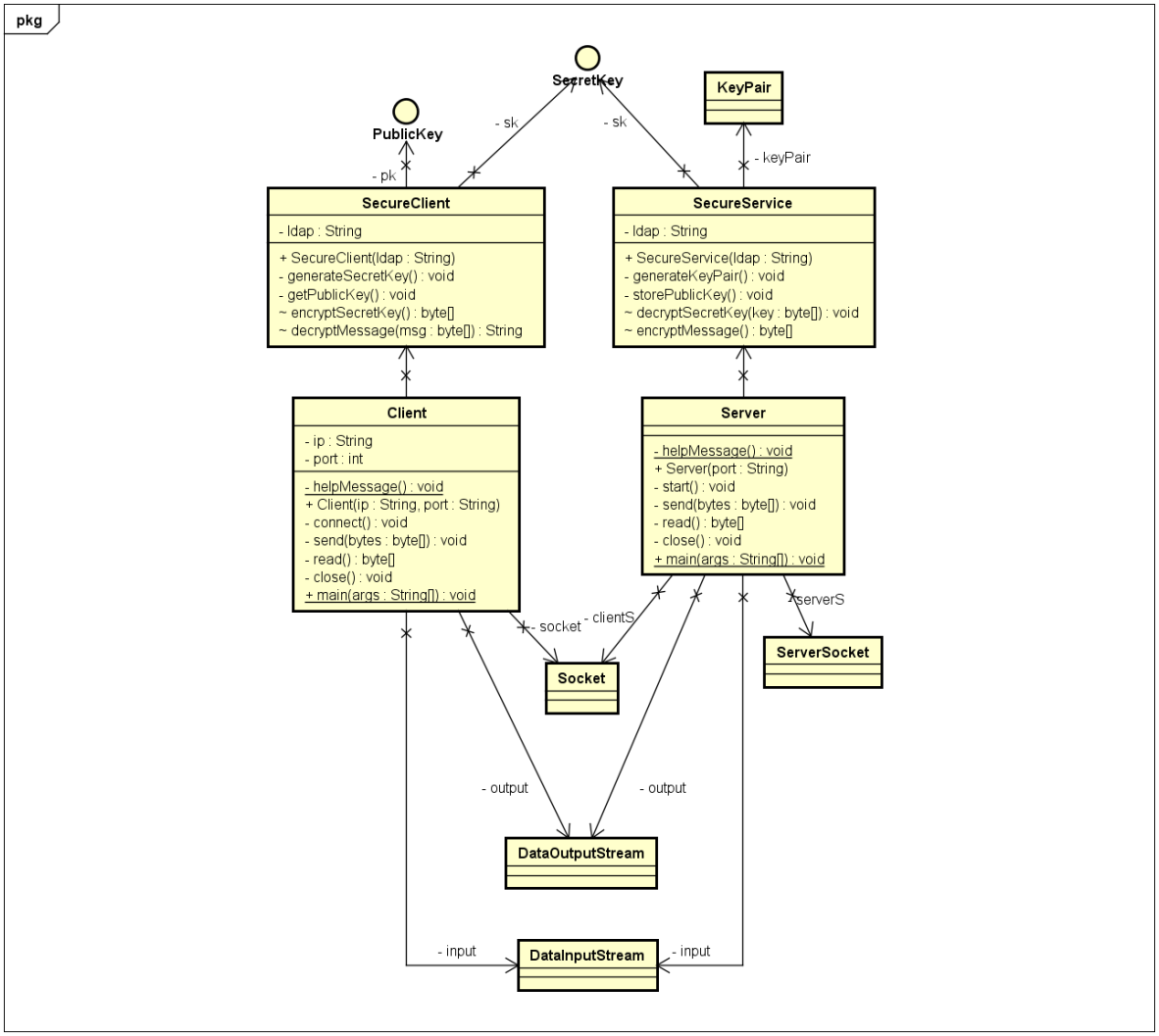

Abbildung 2: UML - Klassendiagramm

Die Methoden in den `SecureXXX` Klassen wurden unter der Vorgabe des Ablaufdiagramms (Abbildung 1) auf Seite 2 entworfen.

Der Ablauf der Methoden ist selbsterklärend:

- `SecureService:` `generateKeyPair()`
- `SecureService:` `storePublicKey()`
- `SecureClient:` `generateSecretKey()`
- `SecureClient:` `getPublicKey()`
- `SecureClient:` `encryptSecretKey()`
- `SecureService:` `decryptSecretKey()`
- `SecureService:` `encryptMessage()`
- `SecureClient:` `decryptMessage()`

Die Kommunikationsklassen sorgen nur noch für das Senden und Empfangen, sowie das Verwalten des `ServerSockets` und `Sockets`.

2.2 Kommunikation

Als Kommunikationsart wurde IPC gewählt, da diese einfach zu realisieren ist.

Da die kryptographischen Java-Libraries direkt mit `byte`-Arrays arbeiten, wurde auf `PrintWriter` und sonstige Klassen, die bei der IPC Laborübung des Vorjahres verwendet wurden, verzichtet.

Die Datenübertragung funktioniert wie folgt:

```
1  private void send(byte[] bytes) {
       try {
3          output.writeInt(bytes.length);
           output.write(bytes);
5      } catch (IOException e) {
           System.err.println("Couldn't send data: " + e.getMessage());
7          System.exit(1);
       }
9  }
```

Listing 1: Senden per `DataOutputStream`

Hierbei wird zuerst die Länge der zu schreibenden Daten festgelegt und danach werden die Daten geschrieben.

Analog dazu funktioniert das Empfangen von Daten:

```
1  private byte[] read() {
       try {
3          byte[] message = new byte[input.readInt()];
           input.readFully(message, 0, message.length);
5          return message;
       } catch (IOException e) {
7          System.err.println("Couldn't receive data: " + e.getMessage());
           System.exit(1);
9          return null;
       }
11 }
```

Listing 2: Empfangen per `DataInputStream`

Auf das Speichern und Lesen bezüglich LDAP Verzeichnisdienste wird nicht näher eingegangen, da dies nicht Thema der Übung ist und bereits im letzten Jahr behandelt wurde.

Wichtig für diese Übung war nur, dass man sich für das Manipulieren eines Attributs authentifizieren muss, für das Auslesen jedoch nicht.

2.3 Schlüsselpaar erstellen

Der erste Schritt des hybriden Verschlüsselungsverfahren ist, ein Schlüsselpaar zu generieren.

```
// setting up key pair generator
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
// generating randomness securely
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
// initialize generator with 2048 bit key size
generator.initialize(2048, random);
// generating key pair
keyPair = generator.generateKeyPair();
```

Listing 3: Generieren eines Schlüsselpaars

Dazu wird zuerst ein `KeyPairGenerator` mit dem gewünschten Algorithmus erstellt. Mögliche Werte wären hier auch EC (elliptische Kurven), DSA (für Signaturen) oder Diffie-Hellman.

Danach wird ein "Zufallsgenerator" erstellt, der so konfiguriert ist, dass er eine plattformunabhängige Implementierung verwendet. Andere mögliche Algorithmen greifen auf Funktionalitäten des OS zurück.

Anschließend wird der Schlüsselgenerator mit einer Key-Größe von 2048-bit und dem "Zufallsgenerator" initialisiert.

Zuletzt wird das eigentliche Schlüsselpaar erzeugt.

2.4 Übertragung des `PublicKey`

Der Schlüssel wird in dem LDAP Verzeichnisdienst als hexadezimaler Wert gespeichert.

```
DatatypeConverter.printHexBinary(keyPair.getPublic().getEncoded());
```

Listing 4: Hexadezimale String-Darstellung des `PublicKey`

Der Client muss diesen String auslesen und zu einem `PublicKey` Objekt umwandeln.

```
// get binary array from hex string
byte[] key = DatatypeConverter.parseHexBinary(response);
// key specifications
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(key);
// key factory for RSA
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
// generate public key from the specification
pk = keyFactory.generatePublic(pubKeySpec);
```

Listing 5: Umwandlung eines Strings zu einem gültigen `PublicKey`

Zuerst wird der String wieder in ein `byte`-Array umgewandelt.

Danach wird die "Codierungsspezifikation" erstellt, welche die Codierungsinformationen nach dem X.509 Standard enthält.

Nun wird eine `KeyFactory` für den RSA Algorithmus erstellt.

Zuletzt decodiert die `KeyFactory` mit Hilfe der Codierungsspezifikationen den `PublicKey`.

2.5 `SecretKey` erstellen

Das Erstellen des `SecretKey` ist simpel.

```
// setting the algorithm
KeyGenerator keygen = KeyGenerator.getInstance("AES");
// actual generating
sk = keygen.generateKey();
```

Listing 6: Erstellen des `SecretKey`

Zuerst wird wieder der Algorithmus gewählt und danach wird der Schlüssel generiert. Mögliche Werte wären zum Beispiel Blowfish, DES oder HmacSHA512.

2.6 `SecretKey` mit dem `PublicKey` verschlüsseln

```
// setting up the cipher
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, pk);
// encrypt the secret key
return cipher.doFinal(sk.getEncoded());
```

Listing 7: `SecretKey` mit dem `PublicKey` verschlüsseln

Der Cipher transformiert (verschlüsselt) Information mit Hilfe eines bestimmten Algorithmus (hier RSA) und eines Keys (`PublicKey`). Dazu wird der Cipher in den Verschlüsselungsmodus gesetzt und schon kann verschlüsselt werden.

2.7 `SecretKey` mit dem `PrivateKey` entschlüsseln

```
// setting up cipher for decryption
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
// decrypting
byte[] ready = cipher.doFinal(key);
sk = new SecretKeySpec(ready, 0, ready.length, "AES");
```

Listing 8: `SecretKey` mit dem `PrivateKey` entschlüsseln

Diese Methode funktioniert analog zur vorherigen. Zusätzlich muss beim Konvertieren des `byte`-Arrays angegeben werden, um welche Schlüsselart (AES) es sich handelt.

2.8 Nachricht mit dem `SecretKey` verschlüsseln

```
// setting up cipher for encryption
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, sk);
// encrypt
return cipher.doFinal(message.getBytes());
```

Listing 9: symmetrische Verschlüsselung

Wie man sieht, gibt es keinen programmiertechnischen Unterschied beim Verwenden eines Ciphers mit asymmetrischer oder symmetrischer Verschlüsselung.

2.9 Nachricht mit dem `SecretKey` entschlüsseln

```
// setting up the decrypting cipher
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, sk);
// decrypting
byte[] ready = cipher.doFinal(msg);
return new String(ready);
```

Listing 10: symmetrische Entschlüsselung

Zusätzlich wird die Nachricht am Schluss wieder in einen String verwandelt.

2.10 Testing

Der Server wurde wie folgt auf der Xubuntu-VM gestartet:

`secureservice <ldap-ip:ldap-port> <service-port>`

```
user@vmxubuntu:~/Desktop$ java -jar server.jar localhost:389 1234
Generating KeyPair ...
Storing public key ...
Waiting for client ...
Press [ENTER] to terminate.
```

Abbildung 3: Server wartet auf den Client

Zur Kontrolle wird geprüft, ob der `PublicKey` in das LDAP Verzeichnis geschrieben wurde.

description

```
30820122300D06092A864886F70D0101010500038201 0F0030820
10A0282010100BDDDFA69CD9D86174D240ACDAFEFF6ABF56F5BDE
BBC513AA74DF08E754336F6DA0B9F50E584FCDF08872AC77DDF5A
4679061CF12CC4A1D39A9D3D80D3E7E998B00C28E985F586723F2
137C76CC0E2EDF4D7E641C41B9652BA33A87E4CB86791D9879874
96FD9AF0085FB8D95B81307D27A9C29586BCB9467FA7AAF97C3EB
208247D83BA04F80E34465626E4EFAEB4B2BD5D0B428F3A352C90
DC22BF419B3CE1F82D8E9B24928841681475CF271495E1EA9C511
68E07D84D2A3A3003CEAE182F6C339A73226F9A67DB1157DF8B69
8871773C6C0E4E2D9B84C7F77DDDCEC387D24BAEC213E6897A1F5
4AB9468E2CC2EAAA93FE8B6B22FB1B8103293255770BE05302030
10001
```

Abbildung 4: `PublicKey` in hexadezimaler Darstellung

Der Client wurde wie folgt auf dem Host-System gestartet:

`secureclient <ldap-ip:ldap-port> <service-ip> <service-port>`

```
C:\Users\danie\Desktop>java -jar client.jar 192.168.1.85:389 192.168.1.85 1234
Generating secret key ...
Receiving public key ...
Encrypting secret key ...
Decrypting message ...
This super secret message was generated on 17-10-2016 at 22:57:24
Terminated.
```

Abbildung 5: Sicherer Nachrichtenaustausch aus Client Sicht

Nachdem der Client den `SecretKey` verschlüsselt weitergegeben hat, sendet der Server die verschlüsselte Nachricht und beendet danach die Kommunikation.

```
user@vmxubuntu:~/Desktop$ java -jar server.jar localhost:389 1234
Generating KeyPair ...
Storing public key ...
Waiting for client ...
Press [ENTER] to terminate.
Decrypting secret key ...
Enrypting message ...
Terminated.
```

Abbildung 6: Server schickt sichere Nachricht

Damit ist das Beispiel komplett.

Listings

Abbildungsverzeichnis