Laborprotokoll

Java Security

Systemtechnik Labor
5BHITT 2015/16, Gruppe X

Stefan Erceg

Version 1.0

Note:

Betreuer: Prof. Micheler

Begonnen am 27. November 2015

Beendet am 4. Dezember 2015

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 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

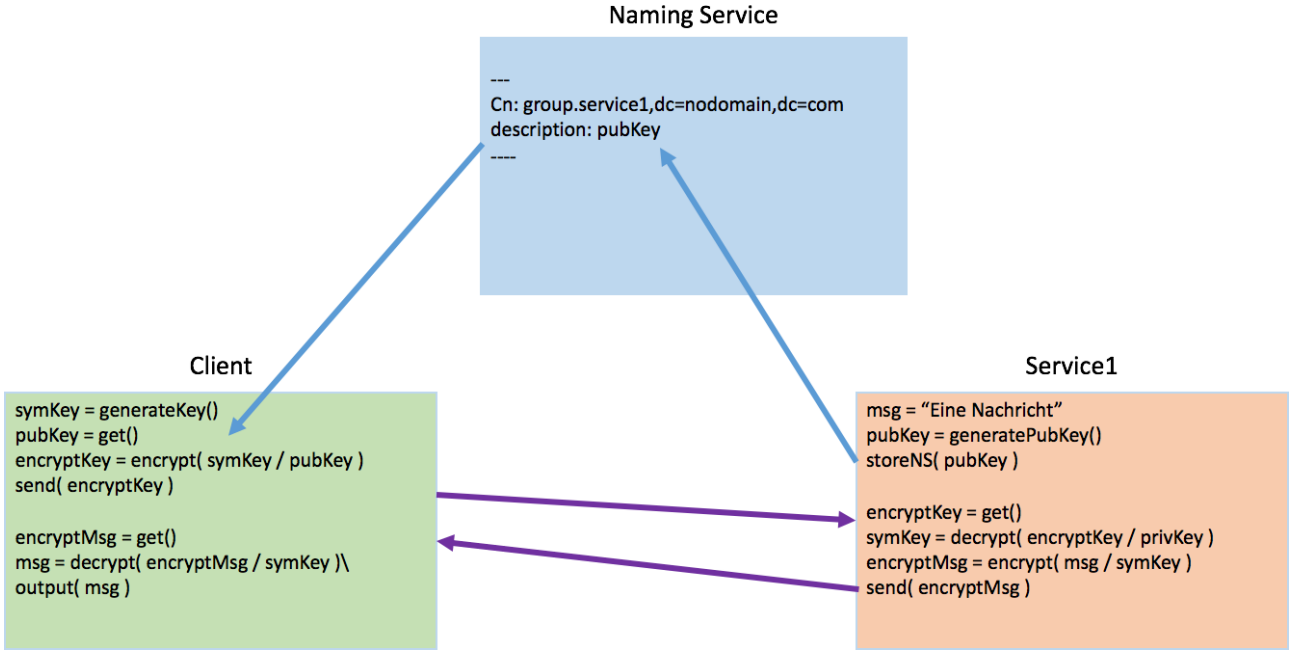
1.3 Aufgabenstellung

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:



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)

2 Ergebnisse

2.1 Ablauf des Programms

In der main-Funktion sieht der Ablauf des Programms folgendermaßen aus:

- Objekte vom Server und Client werden erstellt
- Public Key wird vom Server generiert
- Public Key wird vom Server am LDAP-Server gespeichert
- Client holt sich diesen Public Key vom LDAP-Server
- Verbindung zwischen Server und Client wird über Sockets aufgebaut
- symmetrischer Key wird vom Client generiert
- symmetrischer Key wird vom Client mit dem Public Key verschlüsselt
- Server verschlüsselt mit dem symmetrischen Key eine bestimmte Nachricht
- Client entschlüsselt erneut mit dem symmetrischen Key die Nachricht vom Server

Folgende Ausgaben produziere ich in der Konsole:

```
Server generiert den Public-Key.
Client generiert den symmetrischen Key.
Server speichert den Public-Key in die Description der Gruppe group.service2 am LDAP-Server.
Server wandelt das Byte-Array in folgenden Hex-String um: 30819F300D06092A864886F70D010101050003818D0030818902
Client wandelt folgenden Hex-String erneut in ein Byte-Array um: 30819F300D06092A864886F70D010101050003818D003
Verbindung zwischen Server und Client wurde aufgebaut.
Client verschluesselt den symmetrischen Key mit dem Public-Key.
Server entschluesselt den symmetrischen Key erneut mit dem Private-Key.
Server verschluesselt folgende Nachricht: Mal schauen, ob's funktioniert...
Client entschluesselt folgende Nachricht vom Server: Mal schauen, ob's funktioniert...
Client schliesst die Verbindung.
Server schliesst die Verbindung.
```

2.2 Umsetzung der Socket-Verbindung

Zur Kommunikation zwischen Client und Service habe ich Sockets verwendet. Dazu öffnet der Server-Socket zu Beginn einen Port, welcher im Konstruktor der Service-Klasse als Parameter übergeben wird:

```
public Service1(int port) {
	try {
		this.serverSocket = new ServerSocket(port);
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

Daraufhin gibt der Client die Host-Adresse und den Port des Servers im Konstruktor als Parameter an:

```
public Client(String host, int port) {
	this.host = host;
	this.port = port;
}
```

Der Server wartet daraufhin, bis sich ein Client zu dem Port des Servers verbinden möchte. Daraufhin werden die Attribute „in" und „out" als DataInputStream und DataOutputStream vom Client definiert:

```
clientSocket = serverSocket.accept();
in = new DataInputStream(clientSocket.getInputStream());
out = new DataOutputStream(clientSocket.getOutputStream());
```

Der Client kann danach eine Verbindung zum Server aufbauen, indem er dessen Host-Adresse und Port beim Erstellen eines Sockets angibt. Die Attribute „in" und „out" werden als DataInputStream und DataOutputStream vom Server definiert:

```
socket = new Socket(host, port);
in = new DataInputStream(this.socket.getInputStream());
out = new DataOutputStream(this.socket.getOutputStream());
```

Ebenfalls existieren noch die Methoden "readMsg" und „writeMsg" beim Client, welche zum Lesen und Schreiben eines bestimmten Byte-Arrays dient. Beim Server existiert hingegen die Methode „readByte" zum Lesen eines Byte-Arrays.

[1]

2.3 Umsetzung der asymmetrischen Verschlüsselung

Für die Umsetzung der asymmetrischen Verschlüsselung habe ich zu Beginn bei der Service1-Klasse die Methode „generateKeyPair", bei welcher ein asymmetrischer Key mittels des RSA-Verfahrens generiert wird, entwickelt:

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
generator.initialize(1024, random);
KeyPair keyPair = generator.generateKeyPair();
return keyPair;
```

[2]

Damit der Public Key in die Description von der Gruppe „group.service1" hinzugefügt wird, wird im Service-Konstruktor ein Objekt von der Klasse „LDAPConnector" erstellt und somit eine Verbindung zum LDAP-Server aufgebaut. Der Public Key wird ebenfalls von einem Byte-Array in ein Hex-String umgewandelt:

```
LDAPConnector connector = new LDAPConnector();
this.connector.setDescription(byteArrayToHexString(this.keyPair.getPublic
().getEncoded()));
```

Um das Byte-Array in ein Hex-String umzuwandeln, ist folgende Anweisung notwendig („array" ist ein Byte-Array und wird als Parameter übergeben):

```
String hex = DatatypeConverter.printHexBinary(array);
```

Diesen Public Key holt sich der Client nun vom Naming Service, indem er ebenfalls eine Verbindung zum LDAP-Server aufbaut und die Description ausliest:

```
LDAPConnector connector = new LDAPConnector();
String ldapKey = this.connector.getDescription();
```

Der Key wird daraufhin erneut in ein Byte-Array umgewandelt. Ebenfalls wird eine Key-Spezifikation erstellt, damit Java einen Public Key generieren kann:

```
byte[] key = hexStringToByteArray(ldapKey);
X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(key);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
return keyFactory.generatePublic(pubKeySpec);
```

2.4 Umsetzung der symmetrischen Verschlüsselung

Für die Umsetzung der symmetrischen Verschlüsselung habe ich zu Beginn bei der Client-Klasse die Methode „generateSymKey", bei welcher ein symmetrischer Key mittels des AES-Verfahrens generiert wird, entwickelt:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
return keygen.generateKey();
```

[3]

Dieser generierte Key wurde daraufhin mit dem Public Key folgendermaßen verschlüsselt:

```
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, this.pubKey);
byte[] encoded = cipher.doFinal(this.symKey.getEncoded());
```

Der Server kann den symmetrischen Key mit dem Private Key der asymmetrischen Verschlüsselung erneut entschlüsseln:

```
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
byte[] decrypted = cipher.doFinal(encryptedSymKey);
```

Eine verschlüsselte Nachricht wurde dann mit dem symmetrischen Key gesendet („message" ist ein String und wird als Parameter übergeben):

```
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, this.symKey);
byte[] encrypted = cipher.doFinal((message).getBytes());
```

Der Client kann die verschlüsselte Nachricht vom Server mit dem symmetrischen Key erneut entschlüsseln:

```
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, this.symKey);
byte[] decoded = cipher.doFinal(this.readMsg());
```

3 Zeitaufwand

Für die Umsetzung dieser Übung hatten wir in der zweiten und dritten Laboreinheit Zeit. In dieser Zeit konnte ich die Funktionalität fertig implementieren, allerdings musste ich aufgrund der JavaDoc-Dokumentation und dem Protokoll zusätzliche 2 Stunden in meiner Freizeit investieren.

4 Quellenangaben

[1] Oracle (1995, 2015).
Socket Communications [Online]. Available at:
http://www.oracle.com/technetwork/java/socket-140484.html
[zuletzt abgerufen am 04.12.2015]

[2] Oracle (1995, 2015).
Generate Public and Private Keys [Online]. Available at:
https://docs.oracle.com/javase/tutorial/security/apisign/step2.html
[zuletzt abgerufen am 04.12.2015]

[3] Oracle (1995, 2015).
The KeyGenerator Class [Online]. Available at:
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyGenerator
[zuletzt abgerufen am 04.12.2015]