Sicurezza dell'informazione M

Descrizione del progetto

Il progetto ha come scopo quello creare un servizio di registrazione e autenticazione remoto. Un utente può registrarsi ad un server fornendo il proprio nome e la chiave pubblica a lui associata, in seguito potrà poi autenticarsi presso il server fornendo prova della sua identità tramite un protocollo di sfida/risposta.

In questo modo i due host saranno in grado di comunicare in maniera riservata, senza che un intruso possa capire quali informazioni si stanno scambiando sul canale: pur potendo intercettare le chiavi pubbliche di entrambi ed i messaggi che stanno trasmettendo l'intruso non sarà in grado di decifrare nessuna informazione.

Lo scopo di questo programma è fornire un esempio delle API crittografiche fornite dalle librerie standard di Java.

Protocollo - Registrazione

In questa sezione verrà descritto il protocollo per la registrazione dell'utente: ogni nuovo utente che voglia iscriversi al sistema dovrà fornire un username con il quale registrarsi e la propria chiave pubblica. Il server per completare la registrazione dovrà tenere in memoria queste informazioni in modo da poter identificare l'utente durante la fase di autenticazione.

Indichiamo dunque con U il nostro utente e come S il server al quale vogliamo fare riferimento, la coppia di chiavi asimmetriche del server è definita da S_{PUB} e S_{PRV}, mentre la coppia di chiavi dell'utente è indicata come U_{PUB} e U_{PRV}.

U $\rightarrow$ S : "REG user"
S $\rightarrow$ U : "S_{PUB}"
U $\rightarrow$ S : "U_{PUB}"
S $\rightarrow$ U : "OK"

Generazione delle chiavi

Il codice qui riportato genera una coppia di chiavi asimmetriche dell'algoritmo e della lunghezza indicata (in questo caso è stato utilizzato RSA a 1024 bit).

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(1024);
KeyPair key = keyGen.generateKeyPair();
```

Partendo dalla prima riga possiamo vedere come la classe KeyPairGenerator sia in realtà statica e quindi per ottenere una sua istanza è necessario utilizzare il metodo *getInstance()*, specificando quale algoritmo si intende utilizzare. Il secodo paramentro che viene definito all'interno del metodo *initialize()* specifica quale dimensione deve avere la chiave che vogliamo generare: è importante tener conto che aumentando la lunghezza della chiave aumentiamo la sicurezza ma le operazioni di generazione, criptazione e decriptazione risulteranno più costose in termini di risorse.
Altri algoritmi disponibili sono DiffieHellman, RSA e DSA. Dopo aver effettuato questa operazione è possibile accedere alla chiave pubblica e privata utilizzando i metodi *key.getPublic()* e *key.getPrivate()*.

Invio della chiave

Per inviare la chiave pubblica da un host all'altro è stato necessario incapsulare il suo valore all'interno di una classe accessoria di nome *Frame*. Definendo questa classe come *Serializable* ed inserendo al suo interno in campo *byte[] data* ci è stato possibile inviare in maniera molto più agile la chiave tramite socket.

```
Frame frame = new Frame();
ObjectInputStream inputStream = new ObjectInputStream(new
    FileInputStream(kh.PUBLIC_KEY));
PublicKey publicKey = (PublicKey) inputStream.readObject();
frame.data = publicKey.getEncoded();
outSocket.writeObject(frame); outSocket.flush();
```

Come è possibile osservare dal codice è stato necessario convertire la chiave e portarla sotto forma di array di byte, che è stato poi incapsulato all'interno dell'oggetto *Frame*. Dopo aver preparato l'oggetto abbiamo inviato il tutto al destinatario tramite la primitiva *writeObject()* sulla socket di invio, seguita da un *flush()* per assicurarci che tutto il buffer di invio venga svuotato.

Ricezione della chiave

```
frame = (Frame) inSocket.readObject();
byte[] pubKey = frame.data;
X509EncodedKeySpec ks = new X509EncodedKeySpec(pubKey);
kh.keyDb.put(userName, KeyFactory.getInstance("RSA").generatePublic(ks));
```

Come è possibile vedere anche in ricezione necessitiamo di utilizzare la classe *Frame* per incapsulare la chiave in arrivo, questa viene poi riportata alla sua classe originale (*PublicKey*) tramite altri due classi accessorie: *X509EncodedKeySpec* e *KeyFactory*. Nel Server er mantenere in memoria le associazioni tra username e chiave abbiamo utilizzato una *HashMap*$\prec$*String,PublicKey*$\succ$. Ricordiamo che il client memorizza la chiave pubblica del server durante questa fase.

Protocollo - Autenticazione

Dopo essersi registrato presso il server è possibile per l'utente autenticarsi presso il servizio tramite il protocollo che segue: all'inizio l'utente invia al server il nome utente con il quale intende identificarsi, nel caso in cui l'utente risulti effettivamente registrato il server risponde con un messaggio criptato con la chiave pubblica dell'utente specificato nel primo passaggio.
A questo punto se U è veramente chi dichiara di essere dovrà essere in grado di decriptare il messaggio tramite la chiave privata e quindi di ottenere il *nonce* inviatogli dal server.
Una volta avvenuto ciò l'utente cripta nuovamente il *nonce* con la chiave pubblica del server e lo invia, se il server verifica che il *nonce* inviato è lo stesso che ha ricevuto allora l'autenticazione è completata: solo l'utente U era in possesso della chiave privata corrispondente e quindi è l'unico in grado di decifrare il messaggio.

U $\rightarrow$ S : "AUTH user"
S $\rightarrow$ U : "$E_{U_{PUB}}$(nonce)"
U $\rightarrow$ S : "$E_{S_{PUB}}$(nonce)"
S $\rightarrow$ U : "OK"

Criptazione e Decriptazione

```
byte[] cipherText = null;
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, key);
cipherText = cipher.doFinal(text);
```

```
byte[] dectyptedText = null;
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.DECRYPT_MODE, key);
dectyptedText = cipher.doFinal(text);
```

Come è possibile notare una volta ottenuta la chiave criptare e decriptare è molto semplice, l'unica classe di cui abbiamo bisogno è *Cipher* che, a tutti gli effetti, si comporta come un cifrario diverso a seconda di quale algoritmo viene specificato.
Il metodo *init()* è necessario inserire quale operazione si desidera fare (Criptazione / Decriptazione) e la chiave che si desidera utilizzare.
Infine il metodo *doFinal()* prende come in input il testo da convertire ed effettua la trasformazione.