

Pràctica 5: *Protocols Criptogràfics.*

Objectiu

L'objectiu d'aquesta pràctica és veure com protegir les comunicacions assegurant integritat, autenticitat i confidencialitat mitjançant l'ús de protocols criptogràfics. Concretament ens centrarem en protegir una aplicació que utilitza comunicacions en clar assegurant les propietats abans esmentades.

Protocols Criptogràfics: SSL i TLS

El protocol criptogràfic SSL (*Secure Socket Layer*) fou proposat per Netscape i s'utilitza actualment per a protegir tot tipus de comunicacions en l'àmbit de Internet, especialment les connexions HTTPS. Una evolució d'aquest protocol que incorpora algunes millores és el TLS (*Transport Layer Security*), proposat per l'IETF. Aquest protocol, tot i incorporar millores és totalment compatible amb l'SSL de Netscape.

Tant SSL com TLS estan pensats per protegir les comunicacions entre un client i un servidor. Els protocols ofereixen la possibilitat de xifrar les dades que es transmeten per assegurar la confidencialitat. En quan a l'autenticitat, podem tenir autenticació simple (El servidor s'autentica davant el client) o mútua (Client i servidor s'autentiquen un davant l'altre).

Protocols Criptogràfics a JAVA

Com en altres llenguatges les comunicacions a Java s'implementen mitjançant l'ús de *Sockets*. En Java concretament disposem de les classes *Socket* i *ServerSocket* que permeten establir o esperar connexions cap a altres màquines mitjançant la xarxa. Utilitzar comunicacions xifrades és tan senzill com deixar de utilitzar aquestes dues classes, i canviar-les per *SSLSocket* i *SSLServerSocket*, que són sub-classes de les anteriors.

Aquest esquema és molt útil a l'hora de protegir les comunicacions d'una aplicació existent que no protegeix les connexions. Només canviant les declaracions dels sockets, inicialitzant-los com sockets SSL en comptes de sockets normals proporcionem seguretat a les comunicacions de l'aplicació. La resta de línies del

programa no han de ser modificades per res, el que fa molt senzill la protecció de programes sense introduir errors de programació.

Factories de Sockets

Si donem un cop d'ull a la API de Java veurem que les classes *SSLSocket* i *SSLServerSocket* són classes abstractes, i per tant, no en podem crear objectes directament. Per crear objectes d'aquestes classes o farem mitjançant factories de sockets.

Les factories de sockets SSL venen representades per les classes *SSLSocketFactory* i *SSLServerSocketFactory*. Aquestes dues classes, tot i ser abstractes, tenen un mètode estàtic anomenat *getDefault()* que ens crea una factoria de sockets estàndard. Així doncs, invocant per exemple:

```
SSLSocketFactory ssf =  
    (SSLSocketFactory)SSLSocketFactory.getDefault();  
SSLSocket s = ssf.createSocket(host, port);
```

podríem crear una connexió SSL contra el host indicat, a través del port també indicat per a paràmetres. De forma semblant es faria per els sockets de servidor.

KeyStores i TrustStores

El problema principal una vegada tenim els sockets creats és com es fa l'autenticació. El protocol SSL utilitza certificats X.509 per a dur a terme l'autenticació. En la versió simple del protocol, el servidor mostra el seu certificat al client, mentre que quan s'utilitza autenticació mútua tant el client com el servidor es mostren els seus certificats. Quan una de les parts rep un certificat ha de decidir si aquell certificat autentica o no a la part amb la que s'està connectant.

Quan utilitzem factories de sockets "per defecte" com en l'apartat anterior, els certificats utilitzats per dur a terme l'autenticació es van a buscar a disc, en arxius en format KeyStore.

Una aplicació de xarxa que utilitzi SSL haurà de disposar d'una KeyStore a disc on hi hagi el seu certificat i la seva clau privada. Si aquesta aplicació s'ha d'autenticar davant les parts amb les que vulgui connectar, el motor SSL anirà a buscar el certificat a mostrar dins d'aquesta KeyStore.

De la mateixa manera, si l'aplicació estableix una comunicació i la part remota li entrega un certificat, aquesta ha de decidir si el certificat rebut és "confiable" o no. En les versions "per defecte" de SSL aquesta comprovació també es du a terme amb un KeyStore anomenat TrustStore. Si el certificat rebut es troba dins

el TrustStore, llavors s'accepta la comunicació, en cas contrari es dona un error d'autenticació.

Per especificar on es troben el KeyStore i el TrustStore ho fem per línia de paràmetres. Al executar l'aplicació hem de passar els següents paràmetres respectant majúscules i minúscules:

```
java -Djavax.net.ssl.keyStore=arxiu_keystore
     -Djavax.net.ssl.keyStorePassword=password
     -Djavax.net.ssl.trustStore=arxiu_trustore
     -Djavax.net.ssl.trustStorePassword=password Aplicacio
```

Contextes, KeyManagers i TrustManagers

L'ús de sockets SSL “per defecte” és bastant restrictiu i farragós com ja hem pogut veure. Si volem controlar molt millor tot el procés d'autenticació podem fer-ho des del propi programa sense especificar paràmetres.

Un context, representat en java per la classe motor *SSLContext* permet configurar de forma molt millor la connexió SSL o TLS que volem crear. Aquesta classe motor, com totes les que hem vist fins ara, instancia objectes utilitzant el mètode *getInstance(String protocol)*. L'String que passarem en aquest cas per paràmetres seria la versió del protocol criptogràfic que volem utilitzar (per exemple SSLv3 o TLS).

Una vegada creat l'objecte podem configurar el protocol mitjançant el mètode *.init()*. Aquest mètode permet passar al context una serie de KeyManagers i TrustManagers. A diferència dels KeyStores i TrustStores, aquests managers permeten anar a buscar els certificats necessaris per a l'autenticació allà on sigui. De la mateixa manera, permeten també programar el criteri de decisió a utilitzar a l'hora de decidir si un certificat és fiable o no.

Per crear el nostre propi TrustManager o KeyManager només hem de crear-nos una classe que implementi la interfície X509TrustManager o X509KeyManager i escriure els mètodes que aquestes interfícies defineixen (veure API [1]).

Una vegada un context està creat i inicialitzat, els mètodes *getServerSocketFactory()* i *getSocketFactory()* ens retornaran factories que crearan sockets SSL configurats segons els paràmetres introduïts en la inicialització del context.

Enunciat

En aquesta pràctica haureu de protegir una aplicació client-servidor proporcionada que utilitza comunicacions per sockets estandard.

Per realitzar això haureu de canviar les definicions dels sockets i sockets de servidor de l'aplicació per tal de que utilitzi sockets SSL.

Part 1: Sockets SSL per defecte

Mitjançant l'aplicació *CertMaker* de la pràctica anterior creeu un KeyStore amb el vostre certificat i clau privada (pot ser perfectament un certificat autosignat).

Una vegada creat el KeyStore, creareu un TrustStore, és a dir, un KeyStore amb una llista de certificats en els que confiem.

Finalment, actualitzareu l'aplicació per tal de que utilitzi sockets SSL “per defecte” i l'executareu passant per paràmetres la localització i password del KeyStore i TrustStore.

L'Aplicació client-servidor ha de ser modificada (en la part del servidor) de tal manera que es pugui especificar per paràmetres si desitgem autenticació mútua o no.

Part 2: Managers

En aquesta part crearem un KeyManager i dos TrustManagers propis de manera que no haguem de especificar per paràmetres la localització i password dels nostres KeyStores. El comportament d'aquestes entitats haurà de ser la següent:

KeyManager Aquest KeyManager haurà de ser inicialitzat (a través del seu constructor) amb un nom de KeyStore, un password i un alias. Quan es creï una connexió SSL i es necessiti un certificat o una clau privada, aquest KeyManager retornarà el certificat o clau privada corresponent al alias especificat al constructor.

TrustManager1 Aquest primer TrustManager haurà de ignorar el pas d'autenticació. Quan el client o servidor que utilitzin aquest TrustManager rebin un certificat, no comprovaran la validesa del certificat i simplement retornaran un resultat d'autenticació positiu.

TrustManager2 Aquest segon TrustManager prendrà les decisions basant-se en un KeyStore a disc. Quan en l'establiment d'una comunicació SSL es rebi un certificat, aquest serà acceptat en el cas de que es trobi dins del KeyStore especificat. El nom del KeyStore així com el password hauran d'haver estat passats per constructor al crear la instància d'aquest TrustManager.

Part Opcional: HTTPS

La part opcional d'aquesta pràctica consisteix en crear un programa similar al Wget de linux per a connexions HTTPS.

Aquest programa haurà de funcionar segons la següent sintaxi:

```
java Wget https://adreça arxiu
```

Per dur a terme aquesta pràctica utilitzareu les classes *URL* i *HttpsURLConnection*. Aquestes dues classes permeten llegir molt fàcilment el contingut d'una plana web i, per exemple, escriure'l a un arxiu. Més concretament, ens proporcionen un *stream* de java que apunta a la plana web. Si anem llegint dades d'aquest *stream* i les anem escrivint a un arxiu, no farem res més que el que se'ns demana en l'enunciat.

La gràcia d'aquesta part de la pràctica estarà en la configuració de la connexió HTTPS. La classe *HttpsURLConnection* permet especificar quina factoria de sockets hem de fer servir per manejar la connexió HTTPS. La factoria de sockets que feu servir haurà d'estar configurada amb un *TrustManager* que haureu de crear, i que haurà de actuar com els *TrustManagers* dels navegadors que utilitzem habitualment. Quan ens connectem en un servidor HTTPS aquest ens enviarà el seu certificat. En aquest punt el vostre *TrustManger* haurà d'actuar de la següent manera:

- Disposarem d'un *KeyStore* propi per aquest programa anomenat *httpstrust* i que inicialment estarà buit.
- En el cas que el certificat procedent del servidor es trobi dins el *KeyStore*, acceptarem la connexió SSL i continuarem endavant.
- En el cas de que el certificat sigui desconegut, es mostrarà per pantalla i es preguntarà al usuari si confia en el certificat.
- En el cas de que l'usuari contesti afirmativament, el certificat serà inclòs al *KeyStore*.
- En cas que l'usuari contesti negativament, la comunicació SSL serà refusada donant un error d'autenticació.

Referències

- [1] Java API. <http://java.sun.com/j2se/1.5.0/docs/api/>
- [2] IAIK API. http://jce.iaik.tugraz.at/products/01_jce/documentation/javadoc/index.html
- [3] Java Cryptography Architecture. <http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html>.
- [4] Programació en java. <http://java.programacion.net>