

Pràctica 2: *Criptografia de Asimètrica.*

Objectiu

L'Objectiu d'aquesta primera pràctica en Java és doble. En primer lloc, es tracta de veure el suport que ofereix el llenguatge JAVA per a la criptografia.

Un cop introduïts en l'arquitectura criptogràfica de Java, ens centrarem en l'objectiu principal d'aquesta pràctica. Per això, veurem amb més profunditat com es tracta la criptografia de clau pública dins d'aquesta arquitectura, concretament ens centrarem en els següents aspectes:

- Generació de parells de claus.
- Signat de dades
- Emmagatzemament de claus

Introducció

La API criptogràfica de java es troba dividida en dues parts, la JCA (*Java Cryptography Architecture*) i el JCE (*Java Cryptography Extension*). El motiu d'aquesta divisió no és només una decisió de disseny sinó que també es deu a motius polítics.

La criptografia sempre ha estat considerada una eina perillosa per governs com el d'EE.UU., ja que permet a qualsevol persona utilitzar-la per amagar informació sense que ningú, ni el propi govern, en pugui conèixer el contingut. Per aquest motiu, quan Sun va decidir incloure la criptografia dins el llenguatge sense incomplir les lleis americanes, va haver de condicionar el disseny de la API criptogràfica a aquestes lleis.

El JCA, són tot el conjunt de interfícies que defineixen les operacions criptogràfiques que es poden dur a terme usant el llenguatge Java. Aquestes interfícies, defineixen com usar una signatura digital, un algorisme de xifrat o una funció *hash*, sense proporcionar la implementació concreta de cap algorisme. Aquestes interfícies, conegudes com *Engines* o motors criptogràfics defineixen de forma molt general, quins són els paràmetres necessaris per un algorisme criptogràfic.

Com podem veure en la figura, podem entendre un motor criptogràfic, com una "caixa buida" amb certes dades que entren i que en surten. En el cas d'un algorisme de xifrat, podríem definir aquesta caixa amb tres paràmetres d'entrada i un de sortida:

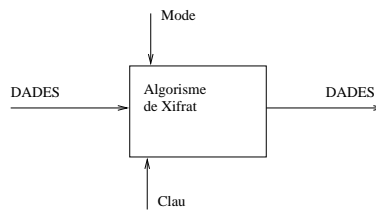


Figura 1: Motor de xifrat

- **Paràmetres d'entrada**

- Dades d'entrada (xifrades o en clar)
- Mode d'operació (Xifrar / Desxifrar)
- Clau

- **Paràmetres de sortida**

- Dades de sortida (xifrades o en clar)

Com hem explicat, la JCA defineix tot aquest conjunt de “caixes”, no només pel xifrat, sinó també per les signatures, funcions *hash*, etc. Tot i així, aquestes caixes es troben buides, no fan res, ja que simplement són interfícies d'alt nivell.

Els *proveïdors criptogràfics* o JCE's (*Java Cryptography Extension*) són paquets software que proporcionen implementacions concretes per als motors criptogràfics que proporciona el JCA. Aquests proveïdors els pot escriure qualsevol persona que vulgui proporcionar un conjunt de implementacions concretes d'algorismes criptogràfics.

El JCE (*Java Cryptography Extension*) proporcionat per Sun defineix el contingut d'aquestes caixes per alguns algorismes criptogràfics. En el cas concret del xifrador / desxifrador que hem vist abans, podem instanciar aquesta interfície amb multitud de tècniques diferents, com per exemple, RSA.

Ús d'una classe motor

Les classes que representen els motors criptogràfics en Java tenen un funcionament bastant comú ja que han estat dissenyades sota els mateixos criteris. Aquestes classes, com poden ser per exemple, *Cipher*, *Signature*, *MessageDigest*, *KeyPairGenerator*, ... no tenen un constructor públic. Això vol dir, que no podem crear un objecte d'aquestes classes amb una crida típica com per exemple:

```
MessageDigest md = new MessageDigest();
```

Així doncs, com creem, per exemple, un objecte de la classe *Signature* quan ens disposem a utilitzar-lo per signar quelcom ? Doncs això ho farem realitzant crides com les següents:

```
Signature sig = Signature.getInstance( ``nomAlgorisme`` );
```

o bé,

```
Signature sig = Signature.getInstance( ``nomAlgorisme`` ,  
``nomProveïdor`` );
```

Com es pot veure en els dos exemples, el mètode *getInstance()* és estàtic, i per tant, pot cridar-se sobre la classe directament, sense haver creat cap objecte concret. Quan invoquem aquest mètode com en el primer exemple, l'arquitectura criptogràfica de Java busca si disposa d'alguna implementació de signatura digital, per a l'algorisme especificat com a paràmetre. L'objecte retornat per el mètode, és la implementació concreta de la signatura llesta per a ser utilitzada.

En el cas de que tinguem instal·lats més d'un JCE o proveïdor criptogràfic, i que disposem de més d'una implementació concreta per a un algorisme determinat, podem especificar quin proveïdor volem utilitzar per aquest algorisme concret. Per fer això, com veiem en el segon dels exemples, només cal especificar el nom del proveïdor que volem fer servir com a segon paràmetre del mètode.

Un cop disposem d'un objecte concret, que es correspon a una implementació concreta d'un algorisme criptogràfic, ja podem començar a utilitzar els mètodes que ofereix l'objecte per començar a treballar amb ell. En el cas de la signatura, ja podríem començar a signar digitalment alguna dada, o bé verificar una signatura feta prèviament.

Criptografia de Clau Pública

En aquesta pràctica utilitzarem la biblioteca criptogràfica de Java per a fer signatures digitals. En aquesta secció explicarem les classes principals que permeten la generació de parelles de claus asimètriques i el signat de dades utilitzant algorismes de criptografia asimètrica.

Generació de claus

Per començar a treballar amb qualsevol tipus d'algorisme relacionat amb la criptografia asimètrica, el primer que hem de aprendre és a generar les claus necessàries per fer servir aquests algorismes.

L'arquitectura criptogràfica de Java ens proporciona una classe motor per a dur a terme aquest propòsit anomenada *KeyPairGenerator*. Com totes les classes motor, disposa dels mètodes *getInstance* que hem esmentat anteriorment, que permeten obtenir un objecte del tipus *KeyPairGenerator* que generi claus per a un algorisme determinat. El proveïdor que proporciona *Sun* per defecte, proporciona els següents algorismes de generació de claus:

- DSA (Digital Signature Algorithm)
- RSA (Rivest Shamir Adleman)

Un cop instanciat un objecte que faci signatures digitals segons un dels dos algorismes, ja podem començar a utilitzar-lo per generar claus. Tot i que la classe *KeyPairGenerator* disposa de molts mètodes, només comentarem els més utilitzats a l'hora de començar a generar les claus:

- *void initialize(int bits)* Aquest mètode inicialitza el generador per tal de crear claus de la mida especificada a través del paràmetre *bits*.
- *KeyPair generateKeyPair()* Un cop inicialitzat el generador de claus ja podem obtenir el parell (clau pública i clau privada) cridant aquest mètode.

La classe *KeyPair* que retorna el generador de claus, conté dos objectes que implementen les interfícies *PublicKey* i *PrivateKey* que poden ser extrets mitjançant els mètodes corresponents (veure API [1]). Aquestes dues interfícies deriven de la interfície general *Key* que implementa qualsevol tipus de clau dins de Java. Aquesta interfície, defineix tres mètodes que tota clau (incloses les públiques i les privades) implementa:

- *String getAlgorithm()* Aquest mètode retorna el nom de l'algorisme criptogràfic per al qual ha estat generada aquesta clau, per exemple RSA o DSA.
- *byte[] getEncoded()* retorna un array de bytes corresponent a la clau, codificada segons algun format. Aquest mètode és molt útil quan volem, per exemple, emmagatzemar una clau a disc.
- *String getFormat()* retorna el format amb el qual s'ha codificat la clau amb el mètode *getEncoded()*.

Signat de dades

El motor criptogràfic que farem servir per al signat de dades és la classe *Signature* que es troba dins el paquet *java.security*. Com tots els motors criptogràfics, aquesta classe no disposa de constructor, sinó del mètode *getInstance(String Algorisme)*, que permet crear “signadors” segons un algorisme determinat.

- ECDSA Signatura amb DSA basat en corbes el·líptiques
- MD2withRSA Signatura amb RSA i el hash MD2
- MD5withRSA Signatura amb RSA i el hash MD5
- NONEwithDSA Signatura amb DSA sense hash. Les dades han de ser exactament 20 bytes.
- SHA1withDSA Signatura amb DSA i el hash SHA1
- SHA1withRSA Signatura amb RSA i el hash SHA1

Una vegada disposem d’un objecte *Signature* instanciat, ja podem començar a treballar amb aquest objecte utilitzant els mètodes que proporciona. Alguns dels mètodes més importants són els següents:

- *void initSign(PrivateKey pk)* Inicialitza l’objecte *Signature* creat per a començar a signar dades.
- *void initVerify(PublicKey pk)* Inicialitza l’objecte *Signature* creat per començar a verificar una signatura.
- *void update(byte[] b)*
- *void update(byte[] b, int offset, int len)* Aquest mètode i l’anterior, són utilitzats per anar introduint dades d’entrada al “signador”, ja sigui per signar-les o per verificar-les.
- *byte[] sign()* Quan totes les dades d’entrada han estat introduïdes mitjançant el mètode *update*, aquest mètode retorna un array de bytes corresponent a la signatura de les dades.
- *boolean verify(byte[] signature)* Donada una signatura, retorna cert, en el cas que aquesta signatura, autèntiqui correctament les dades introduïdes amb el mètode *update*.

Enunciat

En aquesta pràctica es tracta de fer una aplicació per signar digitalment arxius. Per tal de programar aquesta aplicació es proporcionarà un arxiu amb l'esquelet (*FileSigner.java*) de la pràctica amb funcions buides que haureu de omplir.

Per a la part obligatòria de la pràctica les funcions que hauran de ser implementades són les següents:

- *void generaClaus(String algorisme, int bits* Aquest mètode ha de generar una parella de claus per a l'algorisme especificat com a paràmetre. La mida de les claus en bits ha de ser l'indicat pel paràmetre *bits*. Les claus generades s'han de guardar en el membre de la classe *FileSigner* anomenat *_keyPair*.
- *void signaFitxer(String fname, String fsig, String alg)* Aquest mètode ha de signar el fitxer de nom *fname*, emmagatzemant la signatura en un fitxer anomenat segons especifici la variable *fsig* i segons l'algorisme especificat per *alg*. La clau privada usada per signar s'ha d'extreure del parell de claus emmagatzemat al membre *_keyPair* de la classe *FileSigner*.
- *boolean verificaFitxer(String fname, String fsig, String alg)* Aquest mètode ha de verificar la signatura emmagatzemada en el fitxer especificat per *fsig* efectuada sobre el fitxer *fname* segons l'algorisme *alg*. La clau pública usada per verificar la signatura s'ha d'extreure del parell de claus emmagatzemat al membre *_keyPair* de la classe *FileSigner*.

Per provar el funcionament de la classe *FileSigner*, crearem una altra classe que la utilitzi. Aquesta segona classe crearà un objecte del tipus *FileSigner* i anirà invocant els mètodes anteriors.

Part opcional

La part opcional de la pràctica consisteix en implementar unes funcions que permetin emmagatzemar en disc la clau pública i privada generades. Aquestes claus hauran de ser emmagatzemades en arxius separats de manera que puguin ser carregades en posteriors execucions de l'aplicació. Les signatures de les funcions són les següents:

- *void guardaClaus(String arxiuPublica, String arxiuPrivada)* Aquest mètode agafa el parell de claus prèviament generat emmagatzemat en la variable membre *_keyPair* i les emmagatzemma en fitxers separats.

- *void carregaClaus(String arxiuPublic, String arxiuPrivada, String alg)* Aquest mètode llegeix una clau pública i una de privada dels arxius especificats, i construeix un objecte *KeyPair* amb les dues que s'emmagatzemarà dins la variable membre *_keyPair*.

Per implementar aquestes funcions podem guardar la clau obtenint el seu format codificat, a través del mètode *getEncoded()* i emmagatzemar-lo a disc. Per realitzar la operació inversa, és a dir, recuperar l'objecte *PublicKey* i *PrivateKey* a partir de la seva forma codificada, haurem de utilitzar el que anomenem *Key Factories*.

Key Factories

Les *Key Factories* són entitats que permeten construir objectes Java que representen claus criptogràfiques, objectes del tipus *Key*, *PublicKey*, *PrivateKey*, etc. La diferència entre una *KeyFactory* i un *KeyGenerator* és que aquests últims generen claus del no res, generalment de forma aleatòria. Les *Key Factories* en canvi, creen claus a partir de cert "material criptogràfic". Aquest material pot ser, per exemple, un array de bits que hem generat nosaltres aleatòriament, uns nombres *n*, *p* i *q*, en cas de que usem RSA, o bé, a partir d'un array de bytes que representa una clau generada prèviament emmagatzemada usant una codificació determinada.

La classe *KeyFactory* és una classe motor que s'encarrega de generar les claus a partir de material previ. Per inicialitzar-la, utilitzarem com sempre el mètode *.getInstance(algorisme)* especificant com a paràmetre l'algorisme per al qual volem usar la clau que generem.

Un cop inicialitzat l'objecte podem obtenir un objecte del tipus *PublicKey* o *PrivateKey* invocant un dels següents mètodes de la classe *KeyFactory*:

- *PublicKey generatePublic(KeySpec)* Genera una clau pública a partir d'una especificació.
- *PrivateKey generatePrivate(KeySpec)* Genera una clau privada a partir d'una especificació

L'especificació no és res més que un objecte que conté el material per crear la clau, així com informació sobre el format d'aquest material. Les especificacions possibles són classes que es troben dins el paquet *java.security.spec*.

Referències

- [1] Java API. <http://java.sun.com/j2se/1.5.0/docs/api/>

[2] Java Cryptography Architecture. <http://java.sun.com/j2se/1.5.0/docs/guide/s>

[3] Programació en java. <http://java.programacion.net>