Seguretat i criptografia

Joan Arnedo Moreno

Programació de serveis i processos

Índex

Introducció

Les aplicacions informàtiques permeten gestionar i processar de manera molt eficient grans quantitats d'informació. Depenent del tipus d'aplicació, la informació a tractar pot tenir diferents propòsits i formats: potser són les propietats dels personatges d'un videojoc, d'un document de text enriquit o del conjunt de cel·les d'un full de càlcul. Hi ha tantes possibilitats com tipus d'aplicacions us poden venir al cap. Qualsevol tipus d'informació que puguem traduir a format digital és susceptible de ser tractada per un ordinador. Per acabar de posar la cirereta al pastís, l'auge d'Internet ha multiplicat enormement aquest avantatge, fent que qualsevol dada, no només sigui fàcil de processar, sinó també de ser intercanviada entre aplicacions, o accedida pels usuaris. Com a resultat, cada cop és més i més atractiu convertir tota la informació, ja sigui d'organitzacions o individus, al format digital, per facilitar el seu tractament i transport.

Ara bé, això obre una problemàtica molt important. Què succeeix quan algunes d'aquestes dades que tracten les aplicacions estan vinculades a aspectes sensibles, i no s'hauria de poder accedir a elles lliurement? Per exemple, les dades d'una declaració de la renda, la llista d'afiliats a un partit polític, o l'historial clínic d'un pacient. O, si anem a exemples menys extrems, simplement una fotografia una mica compromesa que heu pujat a una xarxa social, només per riure amb els vostres amics i ningú més. No pot ser que qualsevol persona dins l'organització que gestiona aquestes dades hi tingui lliure accés. Només les persones autoritzades haurien de poder accedir-hi. Idealment, ni tan sols un treballador amb accés físic a les màquines, com ara l'administrador de sistemes, hauria de poder veure-les si no està pertinentment autoritzat.

Com a futurs desenvolupadors d'aplicacions, aquesta és una problemàtica que haureu de tenir sempre molt present, ja que aquesta situació va més enllà d'una qüestió de confiança amb els vostres usuaris, sinó que fins i tot pot ser d'obligació per llei. Des de la seva implantació l'any 1999, la Llei Oficial de Protecció de Dades (LOPD) obliga a tractar amb un mínim grau de seguretat certs tipus d'informació i preveu multes de fins centenars de milers d'euros pels casos més graus en els quals no es faci satisfactòriament. Hi ha entorns on, simplement, no és possible implantar una aplicació sense garantir la seguretat de les dades tractades (com ara hospitals, ajuntaments, bancs, etc). Garantir la seguretat a les vostres aplicacions, avui en dia, més que un valor afegit és una necessitat. I una habilitat molt demandada. Per assolir-ho, en aquesta unitat s'estudia de quines eines disposeu quan programeu en Java per poder garantir la seguretat de les dades que tracteu.

A l'apartat "Seguretat i Criptografia" s'introdueix un concepte fonamental per poder garantir la seguretat de la informació dins d'una aplicació: l'ús de mecanismes criptogràfics. Si bé els seus principis són genèrics, i aplicables dins de qualsevol llenguatge de programació, la redacció se centra en com aplicar-la exclusivament

mitjançant Java. Concretament, estudia el funcionament de la biblioteca JCE (*Java Cryptography Extension*), que és la que ofereix el conjunt de classes que permeten aplicar transformacions criptogràfiques sobre unes dades.

Un cop establerts els coneixements bàsics sobre criptografia, aplicada a Java, l'apartat "Aplicacions Java segures a Internet" mostra com resoldre algunes de les problemàtiques més comunes a l'hora de crear programari que es distribueix o necessita connectar-se Internet. Per una banda, com garantir qui és el propietari d'una biblioteca que us heu descarregat i com garantir que no durà a terme tasques malicioses en el vostre equip. D'altra banda, veureu com establir connexions segures a servidors d'Internet.

Al llarg de la unitat didàctica se seguirà tot un conjunt d'exemples que permeten fer més fàcil la comprensió dels conceptes exposats. D'altra banda, per treballar-ne els continguts és convenient anar fent les activitats i els exercicis d'autoavaluació, a més de consultar la bibliografia bàsica proposada. Dins la disciplina de la programació, només la pràctica porta al domini dels conceptes exposats.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Protegeix les aplicacions i les dades definint i aplicant criteris de seguretat en l'accés, emmagatzematge i transmissió de la informació.

- Identifica i aplica principis i pràctiques de programació segura.
- Analitza les principals tècniques i pràctiques criptogràfiques.
- Defineix i implanta polítiques de seguretat per limitar i controlar l'accés dels usuaris a les aplicacions desenvolupades.
- Utilitza esquemes de seguretat basats en rols.
- Usa algorismes criptogràfics per protegir l'accés a la informació emmagatzemada.
- Identifica mètodes per assegurar la informació transmesa.
- Desenvolupa aplicacions que utilitzin sòcols segurs per a la transmissió d'informació.
- Depura i documenta les aplicacions desenvolupades.

1. Criptografia i Java

Avui en dia, les aplicacions informàtiques s'utilitzen per processar o, si més no, com a via d'accés, a una gran quantitat d'informació en format digital, cosa que fa que aquesta sigui cada cop més i més important. Penseu que sota el concepte *informació en format digital* es pot incloure des de la llista de clients i contactes d'una important multinacional fins el PIN, el número i la data de caducitat de la vostra tarja de crèdit.

Malauradament, això també vol dir que aquesta ha esdevingut un objectiu molt atractiu per entitats malicioses que volen treure'n profit a costa dels altres. Per aquest motiu, avui en dia és molt important garantir que les aplicacions gestionen les dades de manera segura. Afortunadament, poc a poc, els desenvolupadors de llenguatges de programació han cobrat consciència d'aquest fet i han anat aportant un seguit d'eines que us poden facilitar assolir aquest objectiu.

Estrictament parlant, la necessitat d'emmagatzemar o intercanviar informació de manera segura es remunta a temps immemorials. Des de sempre ha estat necessari intercanviar informació de manera que es garantís que cap altra persona implicada en el procés pogués interceptar-la i descobrir-la. Ja sigui un papir egipci amb una proposta d'aliança, una carta manuscrita entre dos amants a l'edat mitjana, les comunicacions de ràdio entre tropes militars en plena guerra mundial o un missatge de correu electrònic amb informació sensible per una multinacional en ple segle XXI. Des que els humans intercanviem missatges que aquesta problemàtica existeix. En alguns casos, cal garantir la privadesa de les dades.

> La **privadesa de les dades** (*data privacy*, en anglès) és el servei de seguretat que garanteix que unes dades no puguin ser accedides lliurement. Només han de poder ser accedides per aquells individus que hi estiguin autoritzats.

Al segle XVI, la reina d'Escòcia, Maria Estuard, va perdre el cap, literalment, al ser interceptades i desxifrades les seves cartes conspiratòries.

D'entrada, el cas més fàcil de veure on caldria aplicar aquest servei és durant la transmissió d'un missatge. En el moment que el missatge deixa les mans de l'emissor i abans no arribin al receptor, aquest és susceptible de ser interceptat. En les comunicacions per Internet, el missatge es va desplaçant de manera que qualsevol part implicada en el seu recorregut pot fer-ne una còpia.

La privadesa també afecta a l'emmagatzematge de les dades. Per exemple, penseu en una aplicació d'emmagatzematge de fitxers al núvol. Les dades físiques, els fitxers en si, estan emmagatzemades en el disc dur d'un servidor en algun lloc del món. Per tant, qualsevol persona que pugui demanar l'accés a aquest servidor pot accedir també lliurement a les vostres dades. Tot el que deseu al núvol no és només vostre, sinó que també automàticament de l'empresa en la qual s'emmagatzema. Per exemple, això passaria a Dropbox.

El cas és que la privadesa de les dades s'hauria de poder garantir fins i tot en el cas que tercers tinguin a les seves mans el propi mitjà físic on estan escrites. Per ajudar-vos a resoldre aquest problema, teniu a la vostra disposició tota una branca de les matemàtiques: la criptografia.

La **criptografia** (escriptura secreta, del grec *kryptos*, amagat o secret, i *graphein*, escriptura) és la disciplina que s'encarrega d'estudiar com protegir la informació mitjançant la seva transformació, especialment mitjançant codis o xifrant.

El procés de **xifrat de dades** és el de transformar un conjunt de dades originals, les anomenades dades **en clar**, a un nou conjunt de dades totalment diferent, anomenades dades **xifrades**. Aquest resultat és totalment incomprensible i, idealment, té el mateix aspecte que si s'hagués generat totalment a l'atzar. Només el receptor disposa de les eines necessàries per recuperar les dades original. Aquest procés invers és el de **desxifrat**.

Per assolir aquesta fita, existeixen diferents **algorismes de xifrat**, o processos de transformació diferents. Cadascun té les seves característiques i es basa en l'aplicació de diferents algorismes de transformació sobre les dades originals. Java ofereix mecanismes per disposar alguns d'aquests algorismes, sense que els hàgiu d'implementar personalment.

El paquet **JCE (Java Cryptography Extension)** és una biblioteca que ofereix Java per poder executar diferents algorismes de xifrat als vostres programes.

L'objectiu de JCE és oferir una capa d'abstracció que permeti integrar diferents tipus d'algorismes criptogràfics, de manera que el codi sigui molt semblant o, si més no, només depengui d'uns pocs paràmetres al cridar alguns mètodes, independentment de l'algorisme triat. També està pensat perquè pugui ser estès per incloure nous algorismes que s'inventin en el futur mantenint la retrocompatibiltat de les aplicacions. Cal tenir en compte, però, que el JCE no disposa d'absolutament tots els algorismes existents a data d'avui.

1.1 Sistemes de xifrat simètric

El mecanisme de xifrat per antonomàsia, usat des de la invenció de la criptografia en temps pretèrits, és l'anomenat mecanisme de **clau simètrica**. En aquest, existeix un secret, la clau, que és compartida entre tots aquells que han de ser capaços de llegir les dades. Només qui en té possessió és capaç de xifrar-les i desxifrar-les correctament. Fent un símil planer, és com disposar d'un cofre indestructible, protegit amb un pany de tancament automàtic, on es pot desar un missatge. Només qui té una còpia de la clau del pany pot obrir el cofre i desar-hi

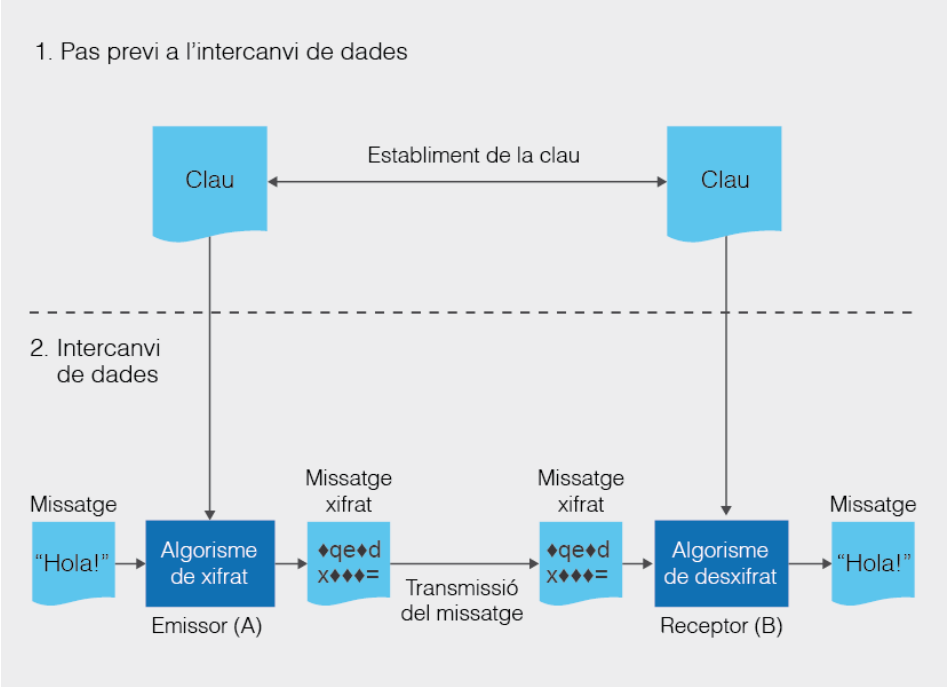
coses, o obrir-lo i recuperar el que hi ha a dins. Per tant, es donarà una còpia de la clau a qui es vulgui que pugui obrir el cofre. Encara que un intrús s'apoderi del cofre i se'l quedi, serà incapaç d'obtenir el seu contingut a menys que tingui la clau correcta.

En el cas d'un missatge de dades, és clar, no existeix tal cofre. La clau no és més que un conjunt de dades, també. En l'antiguitat, la clau era simplement una contrasenya en text, però en el cas d'un ordinador, pot ser qualsevol seqüència de bits. No cal ni que sigui representable com a text llegible. L'algorisme de xifrat combina les dades originals en clar amb la clau per generar com a resultat un conjunt de dades incomprensibles. Per recuperar el missatge original i desfer el procés d'ofuscació cal aplicar l'algorisme de desxifrat usant una còpia de la clau usada durant el procés de xifrat. Si no s'usa la clau correcta, el resultat seran també dades incomprensibles i, per tant, el missatge original romandrà protegit.

En el cas d'un intercanvi de missatges a dues bandes, abans de poder procedir a l'enviament de les dades, caldrà que les parts implicades acordin quina ha de ser la clau secreta. Normalment, la part interessada, ja sigui l'emissor o el receptor, és qui tria la clau i se la fa saber a l'altre. Ara bé, aquest pas és extremadament delicat, ja que cal tenir en compte que el procés d'enviament de la clau també és susceptible de ser interceptat. Per tant, cal fer-ho garantint que es duu a terme de manera segura. Això se sol fer en persona o mitjançant un missatger fiable.

Un esquema del mecanisme d'enviament de missatges xifrats usant clau simètrica es pot veure a la figura 1.1.

Figura 1.1. Esquema del procés de xifrat basat en clau simètrica



Al vostre dia a dia de ben segur que alguna vegada heu intercanviat de manera segura claus simètriques o contrasenyes compartides. Un cas típic és l'establiment de la contrasenya de la xarxa sense fils d'un encaminador ADSL, que s'envia en

un sobre tancat, en mà, per part d'un empleat de l'operador telefònic. Per resoldre aquesta problemàtica caldrà usar una mica d'imaginació.

Evidentment, en el cas d'usar aquest sistema per protegir les vostres dades d'ús totalment personal, vosaltres sou l'emissor i el receptor i, per tant, no cal amoïnar-se pel pas previ d'establiment de cap clau.

Abans de convertir-se en estàndard, l'algorisme AES es coneixia amb el nom de *Rijndael*.

Actualment, existeixen diferents algorismes de xifrat simètric en ús dins del context de les aplicacions informàtiques. Aquest apartat se centra en l'algorisme anomenat AES (*Advanced Encryption Standrad*), que és el que es considera més segur i l'estàndard en algorismes de xifrat des de l'any 2001. Aquest va ser creat per Joan Daemen i Vincent Rijmen i va guanyar el seu estatus d'estàndard després de participar en un concurs molt exigent a nivell internacional. Tot i així, cal tenir en compte que existeixen altres algorismes que encara estan en ús. Per exemple, l'anterior estàndard, el DES (*Data Encryption Standrard*) o el TripleDES. Les modificacions en el codi que cal fer per poder usar un algorisme o un altre mitjançant JCE són molt poques, fet pel qual centrar-se en només un és més que suficient per entendre com fer-ho amb qualsevol altre.

1.1.1 Generació de claus simètriques

Abans de plantejar-se l'execució d'un algorisme de xifrat, primer cal una condició indispensable: decidir quina serà la clau mitjançant la qual es protegirà la informació. Per tant, aquest és el primer pas a resoldre. Tothom qui tingui accés a aquesta clau serà capaç de generar dades xifrades i de desxifrar-les posteriorment.

Des del punt de vista d'un algorisme de xifrat que ha de ser executat per un ordinador, una clau simètrica, no és més que una seqüència aleatòria de bits d'una llargària determinada. Les llargàries vàlides vindran donades exclusivament pel tipus d'algorisme que es vol executar. A part d'això, no cal complir cap altra condició.

El JCE, però, no treballa normalment amb seqüències de bytes a l'hora de gestionar claus, sinó amb instàncies de la classe `javax.crypto.SecretKey`. Les biblioteques del JCE ofereixen una classe auxiliar capaç de generar per si mateixa claus segures per diferents tipus d'algorismes de xifrat. Es tracta de la classe `javax.crypto.KeyGenerator`. El seu ús es mostra en el següent bocí de codi:

L'algorisme AES només treballa amb claus de 128, 192 o 256 bits (a major mesura, més seguretat).

```
1  public SecretKey keygenKeyGeneration(int keySize) {
2      SecretKey sKey = null;
3      if ((keySize == 128)||(keySize == 192)||(keySize == 256)) {
4        try {
5          KeyGenerator kgen = KeyGenerator.getInstance("AES");
6          kgen.init(keySize);
7          sKey = kgen.generateKey();
8
9        } catch (NoSuchAlgorithmException ex) {
10         System.err.println("Generador no disponible.");
11       }
12     }
13     return sKey;
14   }
```

1.1.2 Claus simètriques basades en contrasenya

Una clau simètrica ha de poder ser intercanviada. Una possibilitat és desar-la en un fitxer o escriure els seus bytes, però això seria feixuc per les persones. Una manera de facilitar la gestió de claus criptogràfiques és generar-les a partir d'una contrasenya llegible. Que trieu una contrasenya fàcil o difícil d'endevinar ja és un altre tema.

Per fer-ho podeu aprofitar la particularitat que qualsevol seqüència de bytes de la mida correcta pot servir com una clau simètrica. Només cal, partint de la contrasenya, generar tants bytes com sigui necessari, però de manera que, donada una contrasenya, només aquesta generi una seqüència de bytes donada. I que mai sigui possible, o si més no extremadament improbable, que dues contrasenyes diferents arribin a generar la mateixa clau.

Un exemple de generació i compartició de clau amb contrasenya es troba a l'activació del xifrat als encaminadors ADSL domèstics.

Si hi penseu, aquesta tasca té certes dificultats d'entrada a menys que la contrasenya triada tingui exactament la mida de la clau desitjada. Si té més lletres que els bytes que calen, cal treure'n, i si és massa curta, cal afegir-ne. Haver d'usar una mida concreta i exacta pot ser una restricció una mica empipadora quan precisament el que es volia era facilitar la usabilitat (per una clau AES de 128 bits, 16 caràcters en alfabet llatí). Però si s'accepta qualsevol mida i després no es fan les coses amb traça, us trobareu que contrasenyes diferents poden arribar generar a la mateixa clau. Una manera correcta de resoldre aquest problema és usant un algorisme criptogràfic de *hash* sobre la contrasenya per generar la clau.

> Un **algorisme de *hash***, o resum (*digest*, en anglès), és una transformació criptogràfica que es pot aplicar a un conjunt de dades de manera que compleix sempre un seguit de condicions. Primer de tot, que la relació entre possibles entrades i sortides sigui bijectiva. O sigui, que donada una mateixa entrada, sempre resulti en exactament la mateixa sortida i que dues sortides diferents vinguin sempre d'entrades diferents. D'altra banda, no ha de ser reversible. A partir només d'un resultat, ha de ser pràcticament impossible endevinar quina ha estat l'entrada original que l'ha generat.

En l'actualitat, hi ha diversos algorismes de *hash*. Tot i així, el més utilitzat avui en dia, i el que es considera un estàndard, és l'algorisme SHA-1 (*Secure Hash Algoritm*, o Algorisme Segur de *Hash*). Aquest, a partir d'una entrada de dades de mida arbitrària, genera un resultat de 160 bits. Si es desitja una sortida de més longitud, existeix una versió millorada, més segura, tot i que no tan popular: el SHA-256. Tal com diu el seu nom, aquest genera 256 bits de sortida. De totes formes, des de l'any 2012, ja existeix el SHA-3, una nova versió encara millor i més segura.

Al JCE, la classe `MessageDigest` permet executar algorismes de *hash* sobre unes dades. Les instàncies d'aquesta classe s'obtenen invocant el mètode `getInstance`, que té com a paràmetre l'algorisme desitjat (per exemple `SHA-1`

o SHA-256). Caldrà triar-ne un que generi una sortida de com a mínim la mida de clau desitjada.

Un cop es disposa del resultat d'executar l'algorisme de *hash*, n'hi ha prou amb extreure'n tants bytes com es necessitin per generar la clau de la mateixa manera que si s'haguessin triat a l'atzar. El següent codi mostra com s'obtindria una clau simètrica AES a partir d'un text d'una contrasenya, amb l'ajut de l'algorisme SHA-256.

```
1  public SecretKey passwordKeyGeneration(String text, int keySize) {
2      SecretKey sKey = null;
3      if ((keySize == 128)||(keySize == 192)||(keySize == 256)) {
4        try {
5          byte[] data = text.getBytes("UTF-8");
6          MessageDigest md = MessageDigest.getInstance("SHA-256");
7          byte[] hash = md.digest(data);
8          byte[] key = Arrays.copyOf(hash, keySize/8);
9          sKey = new SecretKeySpec(key, "AES");
10       } catch (Exception ex) {
11         System.err.println("Error generant la clau:" + ex);
12       }
13   }
14     return sKey;
15   }
```

1.1.3 Xifrat AES en mode ECB

En contraposició als algorismes de xifrat en bloc, existeixen els anomenats de xifrat en flux.

L'algorisme AES pertany a una família anomenada de **xifrat en bloc**. Això vol dir que l'algorisme no sap treballar directament amb dades de longitud arbitrària, sinó que només sap transformar bocins o blocs d'una mida molt concreta. Ni més gran ni més petit. Per tant, per poder transformar dades de qualsevol mida, el que es fa és dividir les dades d'entrada en blocs, de la mida adient, mantenint el seu ordre dins la seqüència. Llavors, l'algorisme processa cada bloc individualment, usant la clau completa, de manera que es van obtenint successius resultats parcials. El resultat final s'obté concatenant tots els resultats parcials en el mateix ordre que s'han anat processant per disposar de nou d'una única seqüència de bytes. Cada tipus d'algorisme treballa amb una mida de bloc diferent. Per exemple, l'algorisme AES té una mida de bloc de 16 bytes.

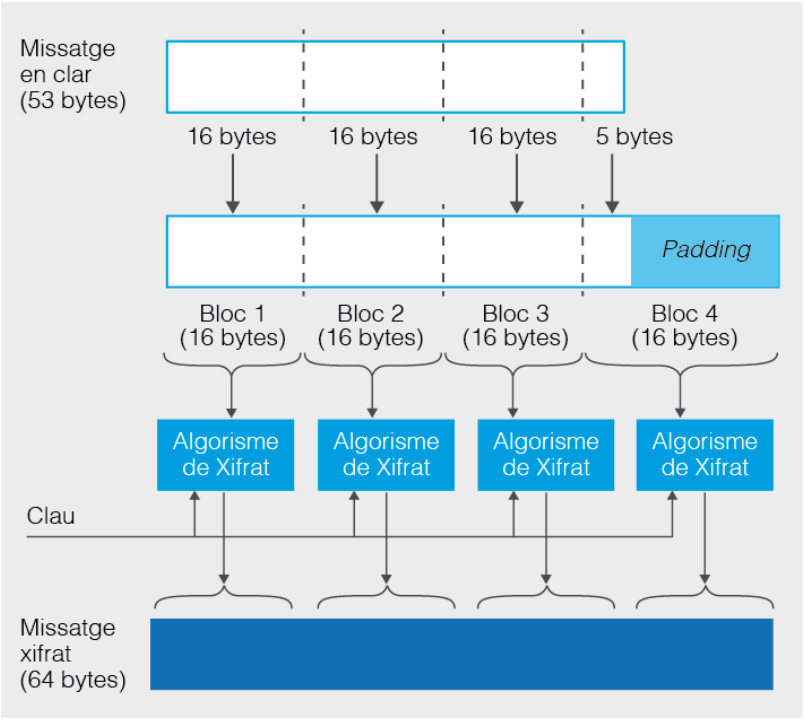
Al treballar d'aquesta manera, pot ser que la mida de les dades que es volen transformar no encaixi exactament amb un múltiple de la mida de bloc i, per tant, el darrer bocí no tingui suficients dades per arribar a la mida exacta d'un bloc. En el cas de l'AES, si l'entrada té 53 bytes, quedarà dividida en tres blocs de 16 bytes, i el darrer en tindrà només 5. Per arribar a la mida establerta, l'algorisme genera el que s'anomena un ***padding*** (un farcit) fins arribar a la mida de bloc. Això vol dir que s'afegeixen valors arbitraris, com poden ser zeros, a les dades que han quedat del darrer bloc. El *padding* simplement s'ignora en procés de desxifrat, però el seu format és un paràmetre que s'ha de subministrar a l'algorisme.

Pel cas de l'exemple d'una entrada amb 53 bytes, el darrer bloc tindria els 5 bytes finals de l'entrada, i tot seguit un *padding* d'11 bytes a 0, per assolir el

total necessari de 16 bytes. La figura 1.2 mostra un esquema gràfic d'aquest funcionament.

La mida total de les dades xifrades amb un algorisme de bloc sempre serà múltiple de la mida de bloc.

Figura 1.2. Esquema del procés de xifrat en mode ECB amb una mida de bloc de 16 bytes



Els algorismes de xifrat de bloc, a la vegada, poden funcionar principalment en dos modes d'operació: ECB (*Electronic Code Book*, Llibre de Codis Electrònic) i CBC (*Cyclic Block Chaining*, Encadenat de Blocs Cíclic). N'hi ha que en poden suportar més, però aquests són els més habituals.

El JCE ofereix la classe `Cipher`, que serveix tant per xifrar com per desxifrar dades. Com passa amb altres classes d'aquesta biblioteca, per obtenir una instància cal invocar el mètode estàtic `getInstance`. Aquest té com a paràmetre una cadena de text amb l'identificador de l'algorisme que es vol usar. Aquest identificador es composa sempre de tres valors separats per una barra (/):

- El nom de l'algorisme en majúscula. Per exemple, "AES".
- El mode d'operació, com ara "ECB" o "CBC".
- L'identificador del tipus de *padding*. N'hi ha de diferents, però se sol usar normalment "PKCS5PADDING".

Un exemple d'identificador d'algorisme podria ser "AES/ECB/PKCS5PADDING". A la documentació de la classe `Cipher` hi ha els tipus identificadors d'algorismes que és capaç d'acceptar.

Tant el procés de xifrat com el de desxifrat sempre segueix el mateix esquema:

1. Un cop instanciada la classe `Cipher`, s'inicia especificant si es vol usar per desxifrar o per xifrar i quina és la clau que cal usar. Això es fa amb el mètode `void init(int opmode, Key key)`. El primer paràmetre indica si es vol xifrar o desxifrar, triant una de les constants estàtiques `Cipher.ENCRYPT_MODE` o `Cipher.DECRYPT_MODE`. El segon paràmetre és la clau simètrica que voleu usar.

2. Si el conjunt de dades a processar no és gaire gran i es troben totes dins d'un *array* de bytes, es pot usar el mètode `byte[] doFinal(byte[] input)` per processar-les totes de cop. El resultat obtingut és el valor resultant. Tot el procés de tractament de blocs es fa de manera transparent dins de la instància de `Cipher`, vosaltres no us heu de preocupar de res.

3. També pot ser que no es disposi de totes les dades i que aquestes s'estiguin obtenint d'un flux d'entrada que requerirà successives lectures per captar tota la informació associada (per exemple, d'un `FileInputStream` si està tractant un fitxer). En aquest cas, s'ha d'usar un altre sistema. Caldrà fer crides successives al mètode `byte[] update(byte[] input)` amb els bocins de dades que es van llegint del flux, i anar desant els resultats parcials. Un cop s'hagin tractat totes les dades, cal finalitzar el procés amb la crida al mètode `byte[] doFinal()`, sense paràmetres, per obtenir el darrer bocí del resultat xifrat.

> La instal·lació per defecte de l'entorn de desenvolupament de Java té un seguit de limitacions en la mida de les claus que es poden usar per xifrar dades (no hi ha cap limitació simplement per generar les claus). Això es deu a antigues polítiques restrictives d'exportació d'eines criptogràfiques fora dels Estats Units. Es consideraven armament militar. Si es volen usar claus AES de més de 128 bits, cal descarregar-se i instal·lar els fitxers de polítiques per criptografia forta: per la versió 7 de Java, és l'anomenat *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 7*. Aquests estan disponibles lliurement a la pàgina de descàrregues de Java. Abans de seguir, és molt recomanable que us l'instal·leu.

El següent bocí de codi mostra com usar la classe `Cipher` per xifrar dades desades dins un *array* de bytes usant l'algorisme AES. El tipus de *padding* triat és `PKCS5Padding`.

```
1  public byte[] encryptData(SecretKey sKey, byte[] data) {
2      byte[] encryptedData = null;
3      try {
4        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
5        cipher.init(Cipher.ENCRYPT_MODE, sKey);
6        encryptedData =  cipher.doFinal(data);
7      } catch (Exception  ex) {
8        System.err.println("Error xifrant les dades: " + ex);
9      }
10     return encryptedData;
11   }
```

Per desxifrar les dades, el procés és exactament igual que el xifrat, amb la única diferència que el mode de l'objecte `Cipher` és `Cipher.DECRYPT_MODE`. En aquest cas, l'entrada seran les dades xifrades.

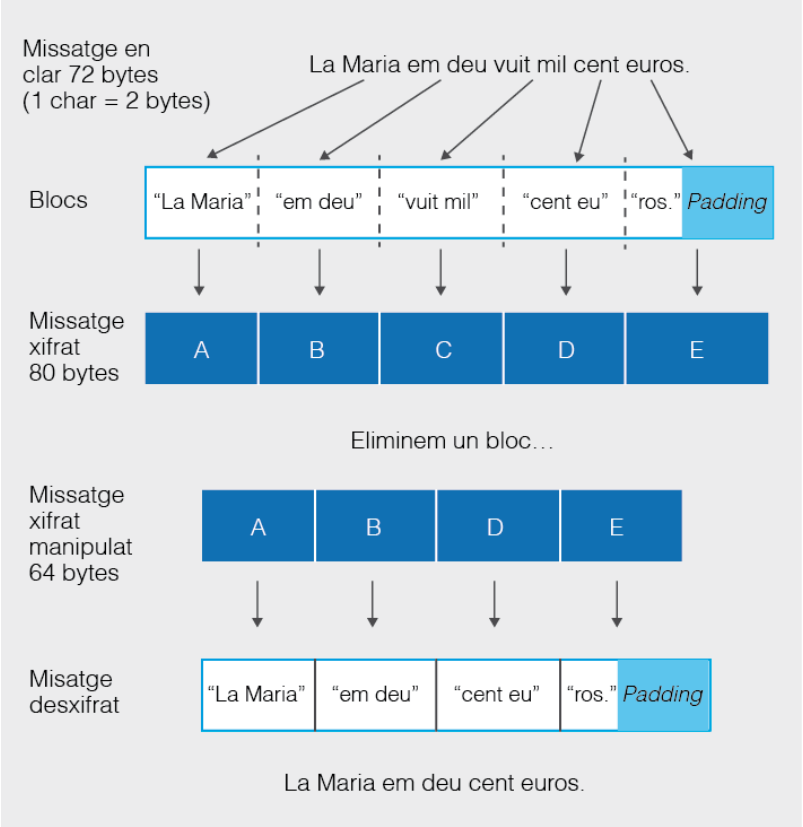
És important tenir present què passa quan s'intenten desxifrar unes dades amb una clau incorrecta. En aquest cas, els bytes resultants seran arbitraris i no tindran cap relació amb les dades originals en clar. A nivell d'un programa en Java, això pot dur a dues situacions diferents que cal controlar. El cas més habitual és que aquestes dades no tinguin un *padding* en el format adient i, per tant, el mètode `doFinal` falli, llançant una excepció del tipus `javax.crypto.BadPaddingException`. L'altra possibilitat, menys freqüent, és que, per pura casualitat, el resultat tingui un *padding* correcte. En aquest cas, el mètode s'executa correctament, però el resultat no té cap sentit. Per exemple, si s'havia xifrat un text llegible, el resultat són valors no representables textualment, o un text totalment incomprensible. És extremadament improbable, encara que no absolutament impossible, que unes dades desxifrades amb una clau incorrecta resultin en un text llegible, tot i que diferent de l'original.

1.1.4 Xifrat AES en mode CBC

El mode d'operació EBC és el més senzill d'usar per xifrar en bloc. Malauradament, la manera com processa les dades en clar per ser xifrades té un parell de problemes que el poden fer vulnerable a atacs si s'intercepten les dades protegides.

Per una banda, donat que cada bloc és totalment independent de l'altre i el resultat només és una concatenació de resultats parcials, l'atacant pot eliminar o reemplaçar blocs del text xifrat sense que sigui possible detectar, a priori, aquest fet. Per veure-ho millor, la figura 1.3 mostra un esquema d'aquest atac.

FIGURA 1.3. Esquema d'atac al mode ECB per modificar el significat d'un missatge



En aquest exemple se suposa una mida de bloc de 16 bytes.

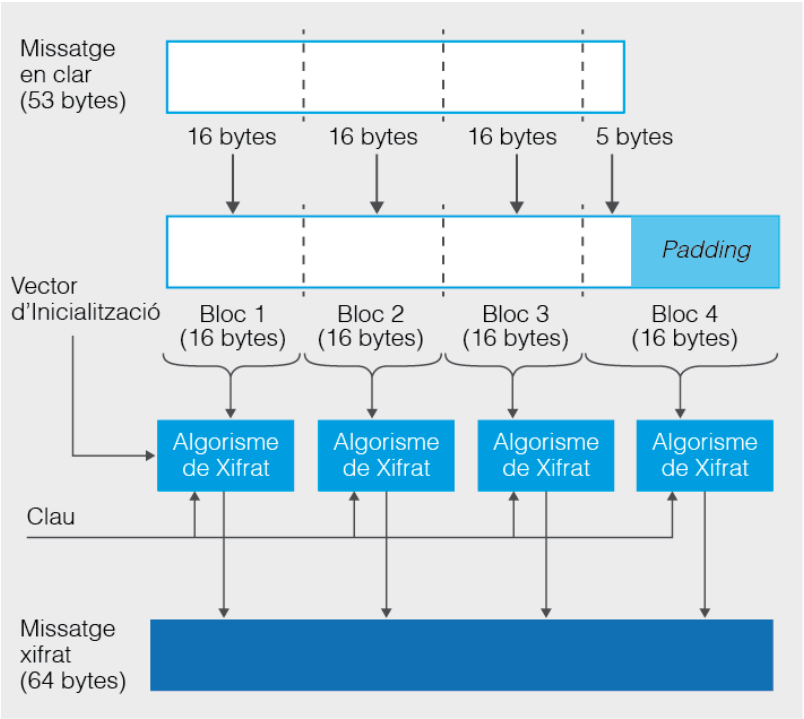
D'altra banda, i aquest és el problema més important, el mode ECB fa que, al dividir el text en clar en blocs, si apareixen blocs repetits a diversos llocs (que contenen exactament els mateixos bytes), el text xifrat també contindrà blocs repetits xifrats. Això pot semblar una minúcia sense importància, però, sense aprofundir més, cal fer palès que realment és per aquesta mena de detalls per on es comença a estudiar com trencar un missatge xifrat tot i no disposar de la clau.

La manera d'evitar aquest dos problemes és fer que els diferents blocs no es xifrin de manera totalment independent, sinó que el procés de xifrat depengui no només de la clau, sinó també del resultat de processar el bloc anterior. Aquest és el mode d'operació CBC.

A l'hora de programar en Java, l'única dificultat que cal superar per usar aquest mode d'operació és veure què fer amb el primer bloc, ja que no té cap bloc anterior. En aquest cas, el que se sol fer és generar un bloc especial anomenat **Vector d'Inicialització** (*Initialization Vector*, o IV per abreujar), que és el que cal aplicar pel primer bloc juntament amb la clau.

La Figura figura 1.4 resumeix el mode d'operació CBC a l'hora de xifrar els blocs i d'usar l'IV.

Figura 1.4. Esquema del procés de xifrat en mode CCB amb una mida de bloc de 16 bytes i un vector d'inicialització



Per veure com funciona un objecte `Cipher` quan es vol xifrar en mode CBC, el millor és un exemple. Si estudieu les instruccions Java del bocí de codi que es mostra tot seguit, podreu observar que el codi és molt semblant al mode ECB.

```
1  //Definició d'un IV estàtic. Per l'AES ha der ser de 16 bytes (un bloc)
2    public static final byte[] IV_PARAM = {0x00, 0x01, 0x02, 0x03,
3                                           0x04, 0x05, 0x06, 0x07,
4                                           0x08, 0x09, 0x0A, 0x0B,
5                                           0x0C, 0x0D, 0x0E, 0x0F};
6
7    public byte[] encryptDataCBC(SecretKey sKey, byte[] data) {
8      byte[] encryptedData = null;
9      try {
10       Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
11       IvParameterSpec iv = new IvParameterSpec(IV_PARAM);
12       cipher.init(Cipher.ENCRYPT_MODE, sKey, iv);
13       encryptedData =  cipher.doFinal(data);
14     } catch (Exception  ex) {
15       System.err.println("Error xifrant les dades: " + ex);
16     }
17     return encryptedData;
18   }
```

Al desxifrar caldrà garantir que el vector d'inicialització és exactament el mateix que es va usar per xifrar. En cas contrari, no obtindreu el text en clar original encara que la clau sigui correcta.

1.2 Sistemes de xifrat asimètric

El sistemes de xifrat simètric són molt eficients ja que, en el fons, no són més que l'aplicació directa als ordinadors dels principis de criptografia tradicional: compartir un secret entre les entitats que es comuniquen. Simplement aprofiten que els ordinadors poden realitzar moltes més operacions de càlcul en pocs instants. Per xifrar algunes dades amb l'AES usant només paper i llapis us hi estaríeu una bona estona!

Ara bé, els esquemes de xifrat simètric tenen un seguit de problemes. Per això, a finals del segle XX sorgeixen el sistemes de **clau asimètrica**. Aquests replantegen l'escenari del cofre protegit amb pany d'una manera certament innovadora. Imagineu-vos que aquest cofre ara té dos panys, cadascun depenent d'una clau totalment diferent. Una de les claus permet obrir una escletxa per on únicament es pot fer entrar el missatge, però no permet treure'l de dins. Aquesta serà l'anomenada **clau pública**. L'altre pany obre el cofre i permet treure els missatges desats a dins. Aquesta s'anomenarà la **clau privada**.

En aquest escenari, el receptor disposa d'una clau de cada tipus i s'encarrega de distribuir una còpia de la clau pública a tothom de qui vulgui rebre missatges secrets. L'avantatge fonamental és que si un espia intercepta una còpia de la clau pública, li servirà de ben poc. Només podrà desar missatges dins el cofre, però mai obrir el cofre, que és el que realment vol. Per tant, aquesta clau no cal que es transmeti de manera segura. De fet, com més fàcil sigui que el màxim de gent en tingui una còpia, millor i tot. Per això aquests sistemes també s'anomenen **de clau pública**.

Descrit a nivell general, en un sistema basat en clau pública cada usuari disposa d'un parell de claus: la seva clau privada i la seva clau pública. Cada usuari genera personalment el seu parell de claus d'acord a un procediment molt concret. Al contrari que el sistema de clau simètrica, aquestes dues claus han de complir unes condicions matemàtiques molt especials, que no s'entrarà a explicar. El resultat és que no val qualsevol seqüència de bits per generar-les, el procés és força més complex i una mica més lent. Llavors, cadascú s'encarrega de distribuir a tothom la seva clau pública, sense haver de preocupar-se de si algú intercepta la transmissió. Per exemple, la pot penjar a la seva plana web. La clau privada se la guarda de manera segura, de manera que ningú altre la sàpiga mai. Cada clau privada només la coneix el seu creador i ningú més.

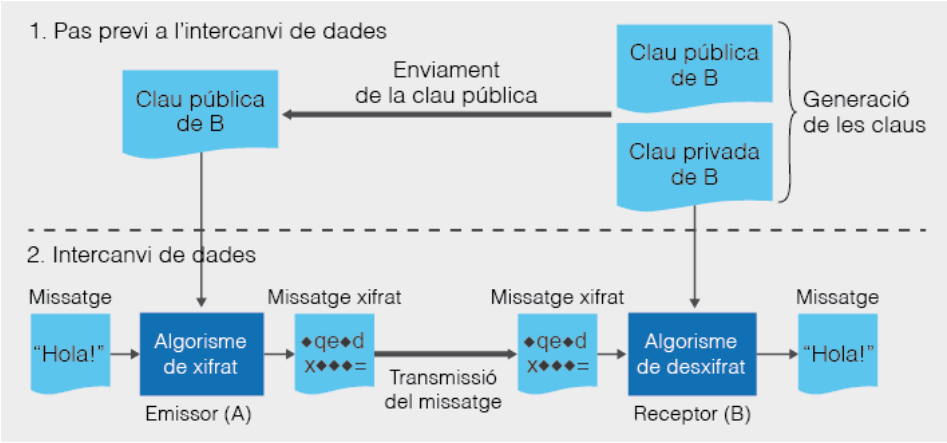
La figura 1.5 mostra l'esquema del mecanisme d'enviament de missatges xifrats usant clau pública.

Els creadors de l'RSA també són fundadors d'una empresa de seguretat molt important a nivell mundial amb el mateix nom.

Tal com passa amb els algorismes de clau simètrica, actualment existeixen diversos algorismes de clau pública. En aquest apartat s'usa com a fil argumental el que es pot anomenar actualment l'estàndard *de facto*, l'RSA. Aquest algorisme va ser publicat l'any 1977 i el seu nom prové de les inicials dels cognoms dels seus creadors, Ron Rivest, Adi Shamir i Leonard Adleman. A nivell d'implementacions pràctiques a gran escala, ara mateix és l´únic que s'utilitza.

De ben segur que vosaltres ja heu fet ús de sistemes basats en aquest algorisme algun cop, ja que és el que garanteix la seguretat a les pàgines web segures, com ara la banca en línia o les botigues virtuals.

Figura 1.5. Esquema del procés de xifrat basat en clau asimètrica o pública



1.2.1 Generació de claus privades i públiques

Al contrari que amb les claus simètriques, els parells de claus asimètriques no són un conjunt arbitrari de bytes que es poden generar de qualsevol manera. Donades les particularitats del xifrat asimètric, perquè el sistema funcioni, tant la clau pública com la clau privada han de tenir un conjunt de propietats matemàtiques molt concretes. Si no es compleixen aquestes propietats, o no funcionarà l'algorisme, o serà molt fàcil d'endevinar la clau privada a partir de la clau pública. Aquestes propietats varien depenent de l'algorisme.

Un nombre primer és aquell que només és divisible exactament, donant la resta de la divisió 0, entre ell mateix o 1.

Sense entrar en detall en els motius matemàtics, pel cas de l'RSA, la clau privada es genera a partir del producte de dos valors enters primers molt grans. La magnitud d'aquest *molt gran* depèn de la mida de la clau que es cerca. Pel cas d'una clau RSA de 2.048 bits, que és el mínim considerat segur avui en dia, això vol dir un valor que estigui entre 2^2.047 i 2^2.047, ni més gran ni més petit. O sigui, un enter codificat usant 256 **bytes** (no bits). Per donar certa perspectiva, el tipus primitiu enter més gran del Java, el `long`, ocupa 8 bytes.

Afortunadament, el JCE ja proporciona la classe `KeyPairGenerator` que permet generar automàticament parells de claus correctes, de manera segura. A menys que us interessi la base matemàtica de l'RSA, coneixent aquesta classe és més que suficient. Per usar-la, n'hi ha prou amb saber quin algorisme de xifrat es voldrà usar i la mida de la clau en bits.

Tot seguit es mostra un bocí de codi d'exemple sobre com usar aquesta classe.

```
public KeyPair randomGenerate(int len) {
    KeyPair keys = null;
    try {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
      keyGen.initialize(len);
      keys = keyGen.genKeyPair();
    } catch (Exception ex) {
      System.err.println("Generador no disponible.");
    }
    return keys;
  }
```

Els parells de claus es representen al JCE amb la classe `KeyPair`. Aquesta classe proporciona els mètodes `getPrivate()` i `getPublic()` per obtenir la clau privada i pública continguda, respectivament. En aquest sentit, les claus privades es representen amb la classe `PrivateKey` i les públiques amb la classe `PublicKey`.

1.2.2 Xifrat RSA directe

El sistema per xifrar i desxifrar mitjançant algorismes de clau pública que ofereix JCE és idèntic a l'emprat per clau simètrica. També es basa en la mateixa classe `Cipher`. Només cal anar amb una mica de compte amb el fet que ara, per xifrar, cal usar la clau pública com a paràmetre i, al desxifrar, la clau privada. Si us confoneu i useu una enlloc de l'altra on no correspon, l'algorisme no funcionarà.

El següent bloc de codi mostra com xifrar dades usant l'algorisme RSA en mode ECB i amb un format de *padding* `PKCS1Padding`, que és el que s'usa en el criptosistema RSA.

```
public byte[] encryptData(byte[] data, PublicKey pub) {
    byte[] encryptedData = null;
    try {
      Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding","SunJCE");
      cipher.init(Cipher.ENCRYPT_MODE, pub);
      encryptedData =  cipher.doFinal(data);
    } catch (Exception  ex) {
      System.err.println("Error xifrant: " + ex);
    }
    return encryptedData;
  }
```

Cal dir que, tot i que cal declarar un mode d'operació i un *padding* als paràmetres d'inicialització de la classe `Cipher`, la manera com aquests afecten l'algorisme no té res a veure amb el cas de clau simètrica. Tenen un significat diferent.

Per desxifrar unes dades, el procés és pràcticament igual, però caldrà usar la clau privada al inicialitzar `Cipher` en mode `Cipher.DECRYPT_MODE`.

1.2.3 Xifrat RSA amb clau embolcallada

Malauradament, el sistema de xifrat RSA té un petit parany en el qual és fàcil caure si no es coneixen els principis matemàtics amb els quals opera. Sense entrar en detall, n'hi ha prou amb saber que la mida de les dades que es poden xifrar està fitada per la mida d'un dels components de la clau privada, anomenat el seu *mòdul*. La mida d'aquest mòdul és el nombre de bytes que cal per representar la mida de la clau. Per exemple, una clau de 1.024 bits té un mòdul de 128 bytes, mentre que una clau de 2.048 en té un de 256 bytes.

Els 11 bytes que es resten a la fórmula es corresponen als que calen per generar el *padding*.

> Concretament, l'algorisme RSA només pot ser usat per xifrar dades de longitud **(mida clau RSA)/8 - 11**.

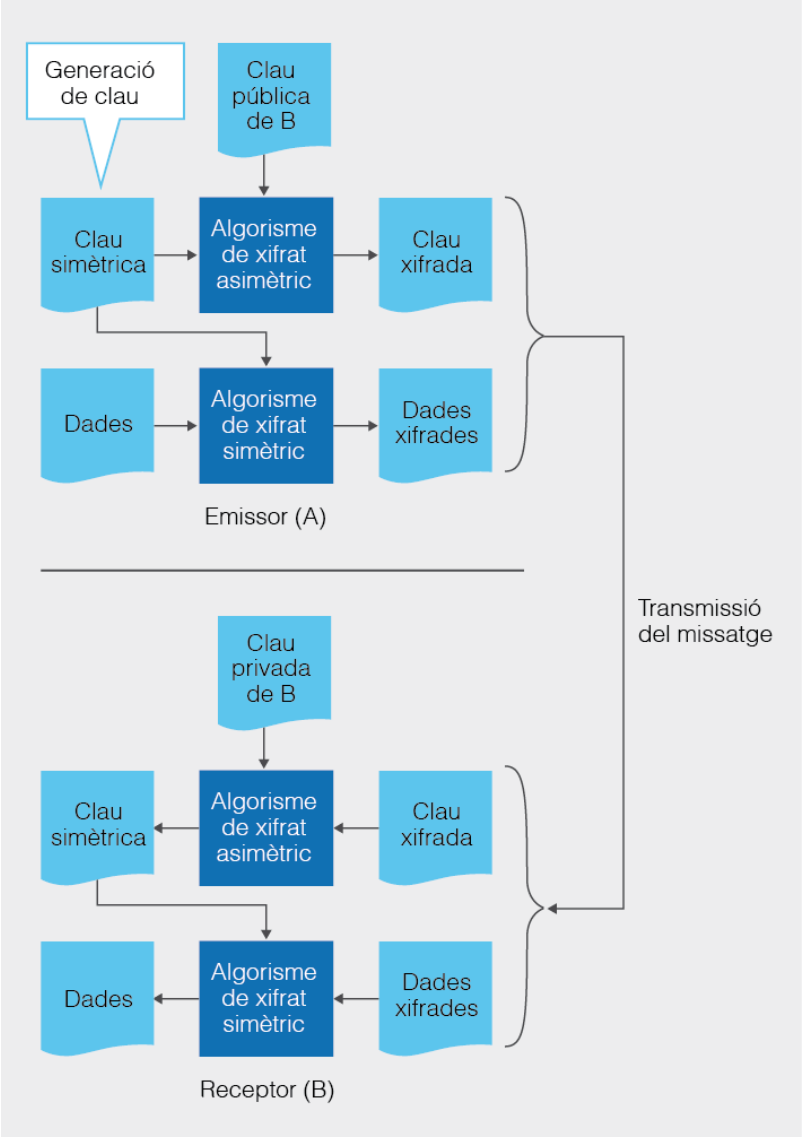
Per tant, una clau de 2.048 bits només pot ser usada per xifrar fins a 245 bytes. Si les dades tenen una mida superior, aquesta es trunca a 245 bytes i la resta es perden. Aquests bytes truncats mai es podran recuperar a l'aplicar l'algorisme de desxifrat.

Evidentment, aquesta és una limitació molt greu, ja que 245 bytes són molt pocs per la mena de dades que es processen avui en dia. Una solució podria ser usar simplement una clau més gran. Però per dades grans, la clau seria enorme, tan gran o més que les pròpies dades. A més a més, l'RSA és un algorisme lent, en comparació amb un sistema de clau simètrica, i com més gran és la clau, molt més lent es torna. Amb la tecnologia actual és inviable. La solució de compromís per resoldre aquest problema és una aproximació híbrida, que combina xifrat asimètric amb xifrat simètric.

> En un sistema de **clau embolcallada** (*wrapped key* en anglès), les dades es xifren usant una clau simètrica d'un sol ús, generada a l'atzar. Aquesta clau llavors es xifra usant la clau pública del destinatari del missatge. Finalment, s'envia al destinatari el missatge i la clau xifrades, conjuntament.

A causa d'aquest sistema, el destinatari el que ha de fer per accedir a les dades és desxifrar la clau simètrica amb la seva clau privada i llavors, un cop la té, usar-la per desxifrar el missatge original. Per tant, cal fer dos processos de xifrat i desxifrat, un usant un algorisme de clau simètrica i un altre de clau pública. Tot el procés es mostra en l'esquema a la figura 1.6.

FIGURA 1.6. Enviament de missatges xifrats usant el sistema de clau embolcallada



El següent codi us mostra un exemple de com fer-ho mitjançant la biblioteca JCE. Com podeu veure, tot plegat és una fusió de la part de xifrat simètric i xifrat asimètric. L'única particularitat amb la qual cal anar amb compte és en la inicialització i ús de la classe `Cipher` a l'embolcallar o desembolcallar la clau simètrica:

- El mode d'inicialització passa a ser `WRAP_MODE` o `UNWRAP_MODE`, enlloc de `ENCRYPT_MODE` o `DECRYPT_MODE`.
- Els mètodes per xifrar i desxifrar la clau que cal usar són `wrap` i `unwrap`, enlloc de `doFinal` o `update`.

```
1 public byte[][] encryptWrappedData(byte[] data, PublicKey pub) {
2     byte[][] encWrappedData = new byte[2][];
3     try {
4       KeyGenerator kgen = KeyGenerator.getInstance("AES");
5       kgen.init(128);
6       SecretKey sKey = kgen.generateKey();
7       Cipher cipher = Cipher.getInstance("AES");
8       cipher.init(Cipher.ENCRYPT_MODE, sKey);
```

```
9       byte[] encMsg = cipher.doFinal(data);
10      cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
11      cipher.init(Cipher.WRAP_MODE, pub);
12      byte[] encKey = cipher.wrap(sKey);
13      encWrappedData[0] = encMsg;
14      encWrappedData[1] = encKey;
15    } catch (Exception  ex) {
16      System.err.println("Ha succeït un error xifrant: " + ex);
17    }
18    return encWrappedData;
19  }
```

Per desxifrar només cal fer el procés invers.

1.3 Firma digital

Inicialment, l'ús dels termes *criptografia* i *xifrat* eren pràcticament intercanviables. L'aplicació d'aquesta disciplina tradicionalment ha estat per assolir la privadesa en l'intercanvi de missatges. Però amb la popularitat d'Internet i la proliferació en l'intercanvi telemàtic de missatges o documents, es fa patent la necessitat de disposar d'altres serveis de seguretat més enllà del de la privadesa. Principalment, els següents:

- **Integritat**: poder identificar si un document ha estat manipulat. Cal fer palès que aquest servei no evita la manipulació, només fa possible que sempre pugui ser detectada pel receptor.
- **Autenticació**: poder garantir quina és la identitat de l'autor del document, evitant que sigui suplantat.
- **No-repudi**: evitar que l'autor pugui negar que ell ha generat el document. El receptor pot demostrar a un tercer la identitat de qui ha emès realment el missatge.

Aquests serveis, si es contemplen tots tres junts, són els que us permeten dur a terme intercanvis fiables de dades, com ara realitzar transaccions o la venda de productes, fer la declaració de la renda telemàticament o connectar-vos a la banca en línia. També permeten garantir que si s'emmagatzema un document a un ordinador, ningú l'haurà modificat quan es torni a obrir sense que la manipulació no es faci patent.

Els grafòlegs són els encarregats de validar una firma manuscrita.

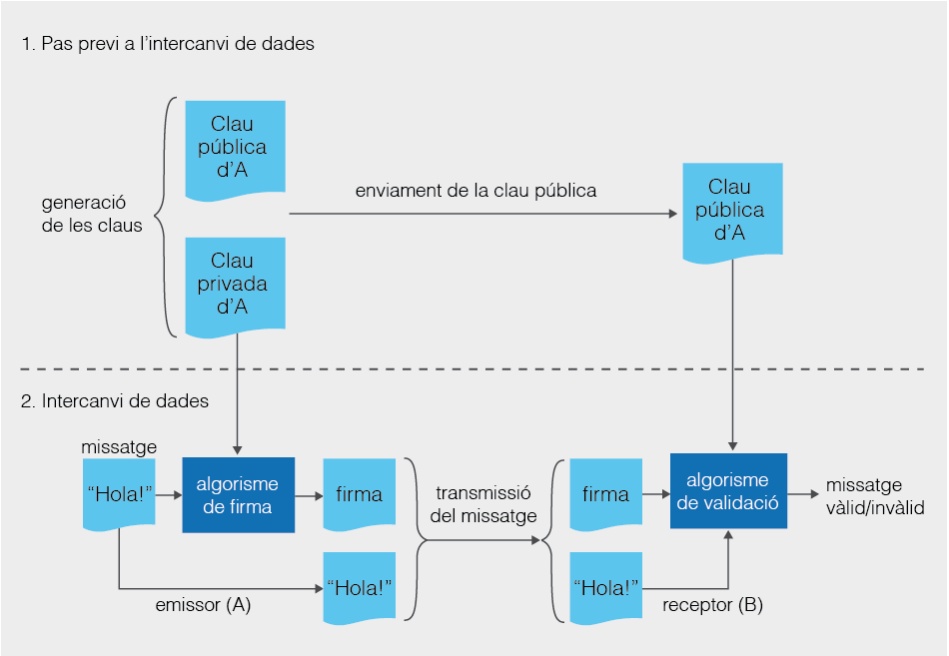
Quan les comunicacions a distància es feien majoritàriament per escrit, garantir aquests punts es considerava relativament simple mitjançant la firma manuscrita. Amb la digitalització de documents i la seva transmissió telemàtica, això ja es converteix en una tasca molt més complicada. Aquest cop no és possible traslladar directament el mecanisme que s'usa en paper. Els mecanismes de xifrat amb ordinadors tampoc no permeten obtenir exactament les mateixes propietats que una firma manuscrita. Caldrà un sistema diferent. La solució la teniu al vostre abast, de nou, gràcies a l'ajut d'una tècnica criptogràfica.

La **firma digital** és el mecanisme criptogràfic que trasllada totes les propietats de la firma manuscrita del món físic a la informació en format digital.

La firma digital imita la firma manuscrita en documents.

Els sistemes de firma digital normalment es basen en un esquema de claus asimètriques, d'acord al procés que es mostra a la figura 1.7. En aquest s'inverteixen els rols de clau pública i privada. L'algorisme criptogràfic de firma emprat en l'emissor es basa en la seva clau privada, que només té ell i ningú, mentre que el procés de validació, en canvi, es basa en la seva clau pública, que pot tenir tothom. També és important remarcar que, al contrari que en el procés de xifrat, el resultat de la firma no reemplaça les dades originals. L'emissor envia les dades i la firma conjuntament, igual que en un document manuscrit signat la firma no reemplaça el seu contingut, sinó que n'és una addenda.

Figura 1.7. Esquema de firma digital basat en clau asimètrica



Perquè la validació de la firma sigui correcta a l'extrem del receptor, s'han de complir les següents condicions, cadascuna associada a un dels seus paràmetres d'entrada. Si qualsevol d'aquest tres paràmetres no és el que correspon a l'extrem de l'emissor, l'algorisme de validació indica que la firma no és vàlida.

L'algorisme de validació només detecta si un document i la seva firma són vàlids o no (cert o fals). Si és invàlid, no pot indicar-ne el motiu exacte.

- Les dades rebudes han de ser exactament les mateixes que les que es van usar a l'executar l'algorisme de firma.
- La firma pròpiament tampoc ha estat modificada.
- La clau pública usada per la validació ha de ser exactament l'associada a la clau privada usada en la generació de la firma. Si es valida amb una altra clau pública, l'algorisme dirà que la firma no és vàlida. Per tant, una firma vàlida només es pot haver generat amb una clau privada concreta, cosa que identifica unívocament al seu propietari com el signant.

Actualment, existeixen diferents algorismes de firma digital. Aquest apartat se centra en el sistema de firma RSA, aprofitant que ja s'ha explicat el sistema de xifrat amb el mateix nom. Es tracta del sistema actualment més popular i el més fàcil d'usar dins de la JCE de Java.

1.3.1 Generació de firma digital RSA

El punt de partida de la firma RSA és idèntic al del xifrat. Existeixen dues claus, pública i privada, que són exactament les mateixes que es poden usar per xifrar les dades. En aquest sentit, no hi ha absolutament cap canvi. Unes claus generades per xifrar es poden usar també per signar, i viceversa. Per tant, les clau RSA poden usar-se per totes dues coses, cosa que resulta un gran avantatge.

La classe responsable de gestionar una firma digital és `Signature`. Si ja se sap com xifrar dades no és gaire difícil d'usar, ja que el procés de firma és molt semblant. El següent codi d'exemple mostra aquest procés.

```
1  public byte[] signData(byte[] data, PrivateKey priv) {
2      byte[] signature = null;
3
4      try {
5        Signature signer = Signature.getInstance("SHA1withRSA");
6        signer.initSign(priv);
7        signer.update(data);
8        signature = signer.sign();
9      } catch (Exception ex) {
10       System.err.println("Error signant les dades: " + ex);
11     }
12     return signature;
13   }
```

Fixeu-vos que els noms dels algorismes de firma solen tenir el format "XXXwithYYY", de manera que són fàcils d'identificar a simple vista.

1.3.2 Validació de firma digital RSA

El procés de validació també es duu a terme amb la classe `Signature` i és molt semblant al de firma. Només cal recordar les seves particularitats: que ara s'usa la clau pública del signant per validar i que, a més a més de les dades originals, un dels paràmetres és la firma.

El següent exemple mostra com validar unes dades amb la seva firma. Fixeu-vos que ara cal cridar els mètodes `initVerify` per inicialitzar el procés amb la clau pública que cal usar i `verify` per executar la validació de la firma pròpiament.

```
1  public boolean validateSignature(byte[] data, byte[] signature, PublicKey pub)
       {
2      boolean isValid = false;
3      try {
4        Signature signer = Signature.getInstance("SHA1withRSA");
```

```
5       signer.initVerify(pub);
6       signer.update(data);
7       isValid = signer.verify(signature);
8     } catch (Exception ex) {
9       System.err.println("Error validant les dades: " + ex);
10    }
11    return isValid;
12  }
```

1.3.3 Certificats digitals

Un aspecte fonamental de la criptografia asimètrica, fora de la implementació de l'algorisme, és la distribució i gestió de la clau pública. Per poder fer un intercanvi segur de missatges, cal que totes les parts implicades disposin de les claus públiques de la resta. Al contrari que en el cas de les claus simètriques, no és necessari dur a terme aquest intercanvi en secret, ja que disposar de la clau pública d'un altre no posa en perill les comunicacions.

Ara bé, a l'hora de gestionar claus públiques, caldrà establir algun sistema per distingir clarament, però de manera senzilla, la identitat del seu propietari. És per aquest motiu que les claus públiques normalment no se solen gestionar directament, agafant la seva seqüència de bytes tal com s'han generat. Aquestes s'inclouen en una estructura de dades que, a part de la clau, conté un conjunt d'informació addicional. El que s'anomena un certificat digital.

> Un **certificat digital** és un document electrònic que estableix la identitat del propietari d'una clau pública.

Si es vol fer un símil, en certa manera, un certificat digital és com un carnet d'identitat. Però en lloc d'associar un nom o identitat a una cara, el que fa és associar-lo a una clau pública, i la clau privada associada.

La definició completa de l'estàndard x.509 es pot trobar a http://www.ietf.org/rfc/rfc2459.txt.

Els certificats digitals més usats segueixen l'anomenat estàndard X.509 per estructurar tota la informació continguda. Aquest estàndard és complex i no val la pena explicar-lo fins al darrer detall. Els continguts més importants per entendre què és un certificat digital són els següents:

- La clau pública del seu propietari
- La identitat del propietari (anomenat també *Subject*)
- La identitat de l'emissor del certificat (anomenat també *Issuer*)
- La seva data de caducitat
- Una firma digital, generada per l'emissor, de tota la informació continguda

Les identitats que conté un certificat digital, tant del propietari com de l'emissor, es codifiquen mitjançant un **nom distingit** (o *Distinguished Name*, en anglès, també

abreujat com a DN). Aquest s'estructura a partir d'una seqüència molt concreta de components, d'acord a la taula 1.1. Aquests s'indiquen en el mateix ordre que a la taula.

TAULA 1.1. Components d'un nom distingit (DN, Distinguised Name)

Acrònim	Nom llarg	Descripció
CN	*Common Name*	Nom i cognoms si és una persona. Nom DNS si és una màquina.
O	*Organization*	Nom de l'organització
OU	*Organizational Unit*	Nom de la unitat estructural
L	*Localty*	Localitat i ciutat
ST	*State*	Província o estat (dels EEUU)
CO	*Country*	Dues lletres inicials del país (en anglès)

Val la pena remarcar un dels elements més importants d'un certificat digital: la firma digital de tot el seu contingut. De fet, aquest element és el que dóna validesa al certificat com a tal, ja que, tractant-se d'un document electrònic, és susceptible de ser manipulat. En cas contrari, qualsevol persona podria agafar el certificat d'una altra persona i posar la seva clau, amb l'objectiu de suplantar-la.

La firma continguda dins d'un certificat digital es genera usant la clau privada de l'emissor.

1.3.4 Emissió de certificats

A l'hora de distribuir la nostra clau pública usant certificats digitals, una pregunta molt important és: qui me'l genera? O sigui, qui serà l'emissor? En aquest sentit, normalment, existeixen dues vies.

Una opció és que el mateix propietari de la clau sigui l'encarregat de generar-se ell mateix un certificat. Al contrari que un carnet físic, que pot tenir sistemes de protecció basades en el seu mitjà físic, de manera que sigui molt difícil l'autoedició (com passa amb un DNI), un document digital pot ser generat per qualsevol persona que sàpiga el seu format. Només cal disposar del programa adient. De fet, existeixen diversos programes, tant de lliure distribució com de pagament, que permeten generar certificats sense problemes.

Els certificats en els quals la identitat del propietari i l'emissor és la mateixa s'anomenen **autosignats**.

En aquest tipus de certificats, el nom distingit del propietari (*Subject*) i de l'emissor (*Issuer*) és exactament el mateix. Així mateix, el certificat està signat amb la clau privada de la pròpia clau pública continguda. Si bé aquest sistema és el més senzill, hi ha un problema força important. Si qualsevol persona pot generar-se un certificat autosignat amb el programa adient, això vol dir que també pot triar lliurement qualsevol de les dades que contingui. Aquestes poden ser reals o inventades. Res impedeix crear un certificat autosignat amb *Batman* al nom del propietari i *Ajuntament de Gotham City* al de l'emissor.

Els certificats autosignats resulten molt útils com certificats de prova per aplicacions en desenvolupament, ja que permeten la màxima flexibilitat en les dades que contenen.

Com podeu imaginar, aquest fet limita molt el seu ús. A nivell d'ús personal, només té sentit usar-los si, al fer l'intercanvi de certificats, les parts implicades poden comprovar, a allà i en aquell moment, que les dades que conté són certes. Per exemple, si es fa l'intercanvi en persona, o ja es coneix al propietari i el té penjat a una pàgina web personal que ja se sap a priori que és autèntica. En aquest sentit, ha d'existir un grau de confiança prèvia entre totes les parts implicades.

En la majoria de casos, un certificat no serà autosignat, ja que precisament a Internet sovint les parts implicades no es coneixen i sempre hi ha el perill que un atacant usi sistemes ben sofisticats per intentar fer-se passar per algú altre de manera expressament malintencionada.

La majoria de certificats digitals usats avui en dia en entorns reals han estat signats per una tercera entitat, una **autoritat de certificació**.

Una autoritat de certificació (o CA, *Certification Authority*) és una entitat que actua com a notari digital, de manera que tothom implicat en un intercanvi de claus sota el seu paraigües confia en què mai emetrà un certificat amb dades incorrectes. Qualsevol pot actuar com una CA, però és clar, que algú realment l'accepti com a tal ja dependrà del grau de confiança que inspiri. A petita escala, com una empresa, l'administrador de sistemes o el departament de recursos humans pot actuar com a CA de tots els treballadors. A gran escala, a nivell d'un país sencer, normalment es tracta de governs o organitzacions.

Quan algú vol generar un certificat, envia una petició amb la seva clau pública i la seva identitat a l'autoritat. Un cop la rep, l'autoritat comprova d'alguna manera que les dades del sol·licitant siguin reals. Per exemple, obligant-lo a presentar-se en persona a una de les seves oficines, amb algun document que acrediti la seva identitat (com un DNI). Només llavors genera el certificat, signant-lo amb la seva clau privada. Un usuari pot tenir diversos certificats, cadascun emès per una CA dins d'un àmbit diferent, igual que es pot tenir un DNI, un carnet de bibiloteca i un carnet d'estudiant, alhora.

La clau pública d'una CA es distribueix en un certificat digital autosignat.

Una de les responsabilitats d'una CA és disposar de tota una infraestructura per poder dur a terme aquest servei, així com posar a disposició del públic la seva clau pública de manera segura i poder donar de baixa certificats emesos.

S'anomena una **infraestructura de clau pública** (o PKI, *Public Key Infrastructure*) al marc per desplegar un mecanisme segur d'intercanvi de claus públiques.

Un cop es disposa d'un certificat generat per una CA, tothom que confiï en la veracitat de l'autoritat podrà confiar en què les dades que conté el certificat són correctes. Per les propietats de la firma digital, ningú pot haver-lo falsificat.

Per estar segur de la validesa d'un certificat, i veure si realment ha estat emès per l'autoritat que posa al seu camp de l'emissor, caldrà fer un seguit de comprovacions. El més important és veure si la firma digital del certificat és correcta, usant la clau pública de la CA. D'altra banda, també caldrà comprovar si el certificat és vigent en la data actual, tant per la seva data d'emissió i d'expiració, com mirant si està o no a la llista de certificats revocats, donats de baixa prematurament, que proporciona la mateixa CA.

Un parell d'exemples de CA propers a vosaltres els trobareu en l'Agència Catalana de Certificació (CATCert) o la Fábrica Nacional de Moneda i Timbre (FNMT), que generen certificats vàlids per fer la declaració de la renda per Internet, entre d'altres coses.

1.4 Gestió de claus

En un sistema amb N usuaris, cada usuari ha de gestionar N - 1 claus simètriques. En total hi ha en circulació (N *(N - 1))/2 claus simètriques.

Un dels principals problemes que cal resoldre per desplegar correctament una aplicació que usa sistemes criptogràfics és la gestió de les claus. Això sovint dóna més feina i és més susceptible a atacs que la implementació del sistema criptogràfic. Per una banda, hi ha la distribució de les claus de manera segura. D'altra banda, tampoc no s'ha d'oblidar el seu emmagatzematge un cop s'han rebut correctament. En tots dos casos és imprescindible que només les persones autoritzades puguin accedir a claus secretes (simètriques o privades), o si més no, garantir que fer-ho no sigui trivial. Cosa que no suceeix si simplement es desen els bytes de la clau a un fitxer o s'apunten a un paper. També cal veure com facilitar aquestes tasques quan una persona no només ha de treballar amb una clau, sinó amb moltes.

Per intentar pal·liar aquestes problemàtiques, les biblioteques criptogràfiques solen oferir la possibilitat de treballar amb magatzems de claus. Java no n'és una excepció.

Un **magatzem de claus** (o *keystore*, en anglès) és un repositori que permet emmagatzemar diverses claus criptogràfiques en un únic mitjà i que garanteix que el contigut es troba protegit, per exemple, mitjançant una contrasenya, en cas de pèrdua o robatori.

Un exemple de magatzem de claus associat a maquinari és el DNI electrònic.

Existeixen diferents tipus i formats de magatzems de claus. Els més habituals solen ser fitxers, però també hi ha magatzems associats a maquinari, com poden ser les targes intel·ligents (*smartcards*) o alguns llapis de memòria USB. El JCE treballa per defecte amb dos formats molt concrets, basats en fitxers: els anomenats JKS (*Java Keystore*) i els JCEKS (*JCE Keystore*). Ambdós són propietaris i exclusius de Java. Per tant, qualsevol clau emmagatzemada dins d'aquests tipus de magatzem només pot ser tractada des d'un programa fet amb Java.

Un magatzem de claus és com un clauer digital.

Un magatzem de claus Java pot desar una quantitat arbitrària d'informació criptogràfica en un únic fitxer, usualment amb extensió JKS. Aquest s'organitza internament en diferents entrades (*entries*, en anglès). Cada entrada s'identifica amb un **àlies**, una cadena de text triada pel propietari del magatzem, que la identifica de manera unívoca. Sempre que s'accedeix a la informació, cal dir quin àlies es vol accedir, comptant que mai n'hi poden haver de repetits.

Per garantir que la informació està protegida, un magaztem Java la pot xifrar amb contrasenya, si així es desitja. Les contasenyes es poden assignar a dos nivells. Per una banda, el magatzem es pot protegir a nivell general, de manera que simplement per accedir-hi i intentar explorar quin és el seu contingut faci falta una contrasenya. D'altra banda, cada entrada individual del magatzem es pot protegir amb una contrasenya diferent. Per tant, en el cas més restrictiu, un magaztem que contingui N entrades pot arribar a estar protegir amb 1 + N contrasenyes: una general al magatzem i una per cada entrada.

1.4.1 L'eina Keytool

Si bé el JCE ofereix la possibilitat de generar claus o de gestionar magatzems mitjançant codi Java, el cas més habitual és generar-les per línia de comandes, de manera que ja estiguin automàticament desades a un magatzem segur, llestes per ser usades als vostres programes.

Keytool és un petit executable que es proporciona amb totes les versions del Kit de Desenvolupament de Java (JDK), juntament amb el compilador i l'intèrpret. Permet gestionar magatzems de claus JKS i JCEKS desde línia de comandes.

El manual de referència de la Keytool el podeu trobar a http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html.

Aquesta eina té moltes opcions, però enumerem només les més típiques a la taula 1.2. Cal anar amb una mica de cura al triar les opcions i paràmetres amb les quals executar Keytool, ja que algunes d'elles tenen valors per defecte en cas que no s'especifiquin. El cas de les opcions "-storetype" i "-keystore" és especialment important.

TAULA 1.2. Opcions típiques de la Keytool

Opció	Tipus de paràmetre	Valor per defecte
-storetype	Tipus de magatzem (JKS, JCEKS)	JKS
-keystore	Nom del fitxer a tractar	fitxer ".keystore" a la carpeta $HOME de l'usuari
-alias	Nom de l'alies de l'entrada a tractar	myalias
-keyalg	Algorisme criptogràfic	DES per clau simètrica, DSA per asimètrica
-keysize	Mida de la clau	Varia segons l'algorisme. Per RSA és 2.048
-storepass	Contrasenya del magatzem	(Si no es posa, preguntarà pel teclat)
-list	Mostra el contingut del magatzem	N/A
-v	Mode "verbose". Aporta informació extra	N/A

En qualsevol cas, podeu consultar-les totes fent *keytool -help*.

1.4.2 Accés a un magatzem a través de codi Java

Finalment, tot i que l'eina Keytool ofereix un mecanisme senzill per generar tota mena de claus i certificats directament a dins de magatzems criptogràfics, una tasca inevitable serà accedir-hi des del codi de les vostres aplicacions per poder executar algorismes criptogràfics.

La gestió de magatzem de claus mitjançant les biblioteques de JCE es porta a terme a través de la classe `java.security.KeyStore`. El procés per poder interactuar amb un magatzem de claus mitjançant una instància d'aquesta classe sempre és el mateix:

1. S'obté una instància del tipus de magatzem amb què es vol treballar usant el mètode estàtic `getInstance` que té com a únic paràmetre el tipus de magatzem.
2. S'inicialitza la instància mitjançant el mètode `load`, que té dos paràmetres: la ruta al fitxer i la contrasenya amb la qual es va protegir al ser creat.
3. Un cop inicialitzat, ja es poden cridar lliurement els seus mètodes per consultar-ne el contingut.

El següent fragment de codi us mostra com dur a terme aquesta tasca:

```
1 public KeyStore loadKeyStore(String ksFile, String ksPwd) throws Exception {
2     KeyStore ks = KeyStore.getInstance("JCEKS");
```

```
3       File f = new File (ksFile);
4       if (f.isFile()) {
5           FileInputStream in = new FileInputStream (f);
6           ks.load(in, ksPwd.toCharArray());
7       }
8       return ks;
9     }
```

Un cop s'ha carregat el magatzem, ja és possible interactuar amb ell per poder accedir a les seves entrades, indexades amb un àlies. Això es fa cridant els mètodes que correspongui d'entre els definits a la classe `KeyStore`. Entre els més significatius, podeu trobar:

- `int size()`: consulta el nombre d'entrades al magatzem.
- `Key getKey(String alias, char[] password)`: obté la clau (simètrica o privada, el que correspongui per l'entrada) associada a un àlies, si existeix. Si la seva entrada està protegida amb contrasenya, cal proporcionar-la al segon paràmetre. Una clau simètrica és de tipus `SecretKey`, mentre que una de privada és de tipus `PrivateKey`.
- `Certificate getCertificate(String alias)`: permet extreure un certificat desat a un àlies concret.

De totes formes, el més recomanable és consultar directament la documentació de l'API de Java per aquesta classe per veure tots els mètodes disponibles per accedir a les dades contingudes.

2. Aplicacions Java segures a Internet

El rol d'Internet és increïblement important en el desenvolupament d'aplicacions en Java. Aquest fet es fa molt evident només observant el *boom* de les aplicacions mòbils, si bé tampoc cal oblidar-se de les aplicacions d'escriptori.

Per una banda, la xarxa de xarxes s'ha convertit en un canal de distribució ideal per aplicacions i biblioteques de classes. Aquest escenari s'aplica tant si sou algú que simplement vol baixar-se un joc o un desenvolupador que no vol haver de reinventar la roda i simplement desitja aprofitar un conjunt de classes que algú altre ha fet i ha publicat a Internet, llestes per ser descarregades. Ara bé, és segur executar en el vostre ordinador, sense la més mínima precaució, un codi que no heu fet vosaltres? Potser estaria bé que existís algun mecanisme per executar codi Java aliè en un entorn controlat, monitoritzant els accessos als recursos.

D'altra banda, i aquest és el cas segurament més evident, moltes aplicacions es basen actualment en les comunicacions en xarxa a través d'Internet per dur a terme les seves tasques. Aquestes aplicacions també han de garantir la seguretat de les dades intercanviades, però comptant que estan treballant en un entorn de sistemes heterogeni. Per tant, cal garantir la compatibilitat dels protocols emprats. Caldria una biblioteca que simplifiqui tot aquest procés, sense haver d'executar algorismes criptogràfics a baix nivell.

2.1 Seguretat a la plataforma Java

La màquina virtual es troba dins l'entorn d'execució de Java (o JRE, *Java Runtime Environment*)

Un dels trets més distintius de Java és el fet que es tracta d'un llenguatge interpretat. Tot i que el codi font dels vostres programes s'ha de compilar, aquest no es transforma en codi objecte directament interpretable per l'ordinador. En realitat, es tradueix a un llenguatge binari intermedi anomenat *bytecode*. Aquest necessita d'un intèrpret, que ha d'estar instal·lat a tots els equips on es vulgui executar el programa: la màquina virtual de Java (o JVM, *Java Virtual Machine*).

El benefici més evident d'usar un llenguatge interpretat és poder executar aplicacions en entorns heterogenis, de manera que les aplicacions siguin multi-plataforma. Mentre l'entorn d'execució estigui instal·lat a l'equip, no importa el seu sistema operatiu o model de processador. Les contrapartides són que cal una configuració prèvia del sistema abans no es pot executar cap programa i que les aplicacions són menys eficients en l'ús de recursos i velocitat d'execució, ja que el codi no s'executa directament sobre el maquinari.

Des del punt de vista exclusivament de seguretat, l'ús d'un intèrpret també aporta un seguit d'avantatges addicionals. Ja que tot programa en Java ha de ser processat per aquest intèrpret abans de que es doni cap ordre directa al maquinari, això

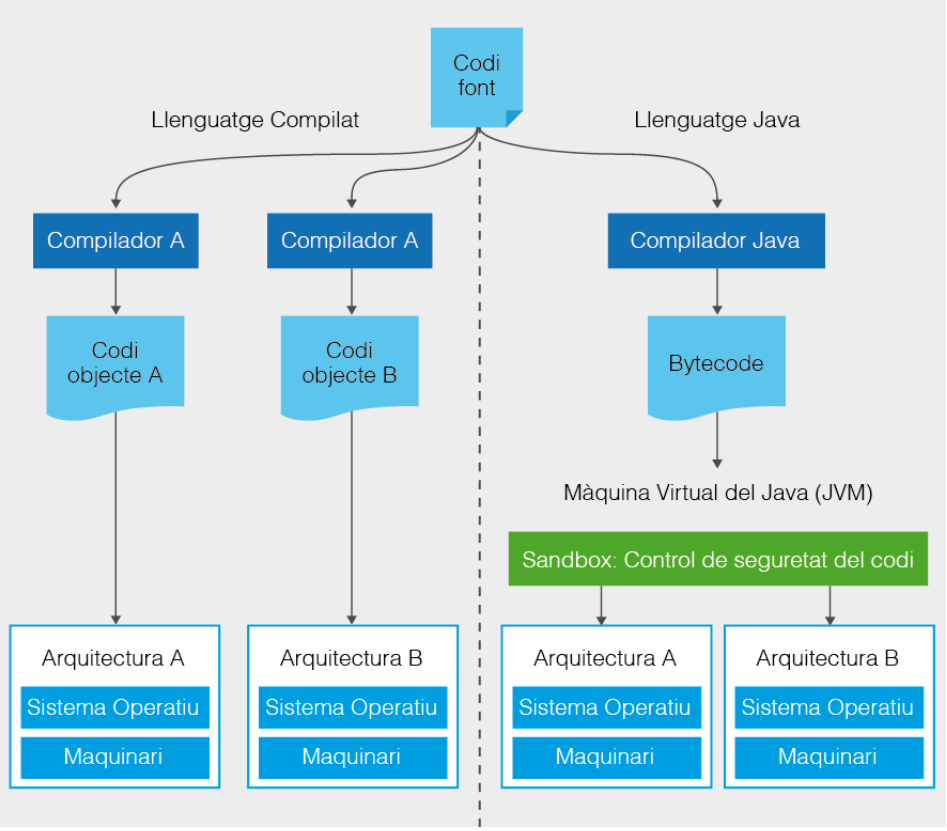
permet establir certs controls sobre els programes. Per una banda, pot arbitrar les accions de les aplicacions i el tipus de classes usades, vetant-ne l'accés a cert tipus, si escau. D'altra banda, també permet dur a terme comprovacions sobre el propi *bytecode* abans de ni tan sols carregar-lo i executar-lo, per evitar certs tipus de paranys.

Una capsa de sorra, un entorn controlat on poder jugar.

La **capsa de sorra del Java** (o *sandbox*) és com s'anomena popularment l'entorn controlat en el qual es s'executen les aplicacions fetes en aquest llenguatge, de manera que sigui possible restringir certes accions, per garantir-ne la seguretat.

La diferència entre el procés de generació d'una aplicació i la seva execució entre Java i els llenguatges compilats tradicionals, amb la participació de la capsa de sorra del Java, es mostra a la figura 2.1.

Figura 2.1. Generació d'executables i control del codi en temps d'execució



Si es treu profit d'aquesta funcionalitat de Java, ens trobem davant d'un gran avantatge, ja que permet executar les aplicacions dins un entorn inicialment controlat. Un cop tinguem clar que no són malicioses, llavors podem obviar aquests controls o fer-los més laxes.

2.1.1 El gestor de seguretat

Un tallafocs (o *firewall*) és un programa que permet controlar el trànsit de xarxa que surt o entra d'un ordinador.

Si bé la màquina virtual de Java té la capacitat de establir controls sobre el codi que s'hi vol executar, per defecte, no en força cap. Per tant, d'entrada, totes les aplicacions tenen accés lliure a qualsevol recurs de l'ordinador en el qual s'executen sense cap altra restricció que les imposades pel propi sistema operatiu. Per exemple, les proteccions al sistema de fitxer o els tipus de connexions permeses per un tallafocs.

Sempre que es vulgui establir algun tipus de control, caldrà indicar-ho mitjançant l'assignació d'un gestor de seguretat al programa que es vol executar.

Un **gestor de seguretat**, o *security manager* en anglès, és una classe mitjançant la qual es pot establir una política de seguretat sobre una aplicació quan s'executa. O sigui, indicar quin és el conjunt d'accions que té permès, o no, dur a terme.

En el cas de Java, per indicar a una aplicació que ha d'usar un gestor de seguretat concret, cal fer-ho usant l'opció d'execució `-D` a la seva línia de comandes, que serveix per assignar valors a les propietats del sistema.

```
1 java –Djava.security.manager=NomGestor [altres opcions] nomAplicació
```

Opcions d'execució i Netbeans

Les opcions d'execució de l'intèrpret de Java s'indiquen afegint-les a la pròpia comanda quan s'executa. Ara bé, en el cas d'usar eines en les quals aquest procés queda ocult, com pot ser l'edició mitjançant un IDE, caldrà cercar a les seves opcions com afegir-les. El procés pot variar per cada IDE.

En el cas de les versions 7.x del Netbeans, cal obrir les propietats del projecte, dins del menú *Fitxer* (o *File*), anar a l'opció d'*Execució* (o *Run*) i afegir l'opció dins de la capsa de text anomenada *Opcions* (*VM Options*).

Tot i que Java permet implementar els vostres propis gestors de seguretat, de manera que s'adaptin a les vostres necessitats, ens centrem ara en el gestor que es proporciona per defecte, la classe `java.security.manager.SecurityManager`. Per usar aquest gestor per defecte, n'hi ha prou amb usar l'opció esmentada, però sense indicar cap nom.

```
1 java –Djava.security.manager [altres opcions] nomAplicació
```

Un cop assignat el gestor de seguretat, l'aplicació veurà limitats els recursos als quals pot accedir durant la seva execució. En cas d'intentar accedir-hi igualment, succeirà un error d'execució (`AccessControlException`).

Pel cas del gestor proporcionat per defecte, les limitacions són les següents, a grans trets:

- No es permet la lectura del sistema de fitxers fora de la carpeta de l'aplicació.

El gestor de seguretat controla les accions dels programes.

- No es permet en cap cas l'escriptura al sistema de fitxers.
- No es poden consultar propietats del sistema vinculades a l'usuari o al sistema de fitxers. Per exemple, quina és la seva carpeta personal (carpeta *HOME*).
- No es poden dur a terme certes connexions de xarxa.
- No es poden llençar comandes locals mitjançat la crida `Runtime.exec`.
- No és possible llençar codi en biblioteques natives, no fetes amb Java.

Una prova molt senzilla per veure això és la següent. Genereu un fitxer de text anomenat Test.txt, amb qualsevol contingut, a l'arrel del vostre sistema de fitxers. Per exemple, en un sistema basat en Unix, a la carpeta "/" i en un basat en Windows, a la "C:\". Executeu el codi que es mostra a continuació sense assignar un gestor de seguretat i, tot seguit, torneu-ho a fer assignant el gestor per defecte.

```
1 public class TestFileAccess {
2
3   public static void main(String[] args) throws Exception {
4     File f = new File ("/Test.txt");
5     try (FileInputStream in = new FileInputStream(f)) {
6       in.read();
7     }
8   }
9 }
```

Si no hi ha cap gestor assignat, el programa actuarà tal com s'espera. Però si està activat el gestor, donarà un error del següent estil:

```
1 Exception in thread "main" java.security.AccessControlException: access denied
      (
2 "java.io.FilePermission" "\Test.txt" "read")
3         ...
```

En canvi, si copieu el fitxer a la carpeta del vostre programa i modifiqueu el codi font de manera que s'accedeixi a Test.txt enlloc de /Test.txt, el programa funcionarà de nou sense problemes. Per tant, aquí podeu veure que el gestor de seguretat, per defecte, controla l'accés als recursos de manera que un programa no pot accedir lliurement a qualsevol part del sistema en el qual s'executa.

2.1.2 Assignació de permisos

El conjunt de restriccions a les quals es veu sotmesa una aplicació Java quan és supervisada pel gestor de seguretat per defecte és bastant gran. Normalment, però, voldreu establir un seguit d'excepcions a la llista de controls, de manera que l'aplicació sí que tingui accés a alguns recursos, o pugui dur a terme certes accions concretes.

Per poder establir quines accions pot dur a terme una aplicació controlada pel gestor de seguretat, cal assignar-li també un **fitxer de polítiques** (o *policy file*, en anglès). Aquest conté una llista de les accions que sí que pot dur a terme sense problemes.

Per defecte, Java cerca el fitxer anomenat .java.policy, a la carpeta principal (*HOME*) de l'usuari que executa l'aplicació. Noteu que, en sistemes Unix, és un fitxer ocult. Normalment, aquest fitxer no existeix d'entrada, un cop instal·lat el Java. L'ha de generar i editar cada usuari, indicant, a títol individual, quina mena de controls vol dur a terme sobre les aplicacions que executa.

De totes formes, també és possible indicar un altre fitxer diferent a aquest amb l'opció "-D". Alerta amb el doble igual (==) a la comanda.

```
java –Djava.security.manager –Djava.security.policy==rutaFitxer nomAplicació
```

Un fitxer de polítiques permet incloure opcions de diversa índole i complexitat. Aquest es composa d'una llista de blocs `grant { ... }`, dins dels quals es llisten, mitjançant la comanda `permission`, els diferents permisos garantits. La comanda `permission` requereix tres paràmetres:

- El **tipus de permís**. Java organitza els diferents tipus de permisos mitjançant la definició de classes, que representen una acció possible que cal supervisar, a nivell genèric.

- El **nom del seu objectiu** (o *target name* en anglès), que és un identificador del recurs concret a controlar.

- El conjunt d'**accions** que es permet dur a terme sobre aquest objectiu. Els valors possibles per aquests dos casos solen dependre del tipus de permís, ja que no sempre tenen sentit per qualsevol tipus.

A títol d'exemple, l'estructura d'un fitxer de polítiques sol ser la següent:

```
//Un bloc de permisos
grant {
  permission NomTipusPermís1 "NomObjectiu1", "Acció1.1, Acció1.2,...";
  ...
  permission NomTipusPermísN "NomObjectiuN", "AcciónN.1, AcciónN.2,...";
};

//Un altre bloc de permisos
grant {
  permission NomTipusPermís1 "NomObjectiu1", "Acció1.1, Acció1.2,...";
  ...
  permission NomTipusPermísN "NomObjectiuN", "AcciónN.1, AcciónN.2,...";
};

...
```

Policytool

Per tal de no haver de barallar-se amb la sintaxi específica d'un fitxer de permisos, l'entorn de desenvolupament de Java ofereix una eina molt útil: un editor de polítiques.

> L'eina **Policytool** permet generar conjunts de permisos dins de fitxers de polítiques sense haver de conèixer la seva sintaxi. Tot es fa mitjançant menús desplegables en una interfície gràfica.

El gran avantatge d'aquesta eina és que, en molts casos, ja enumera les opcions disponibles i no cal cercar la seva llista a la documentació. Per exemple, mitjançant menús desplegables, ja mostra tots els tipus de permisos, o el conjunt d'accions disponibles.

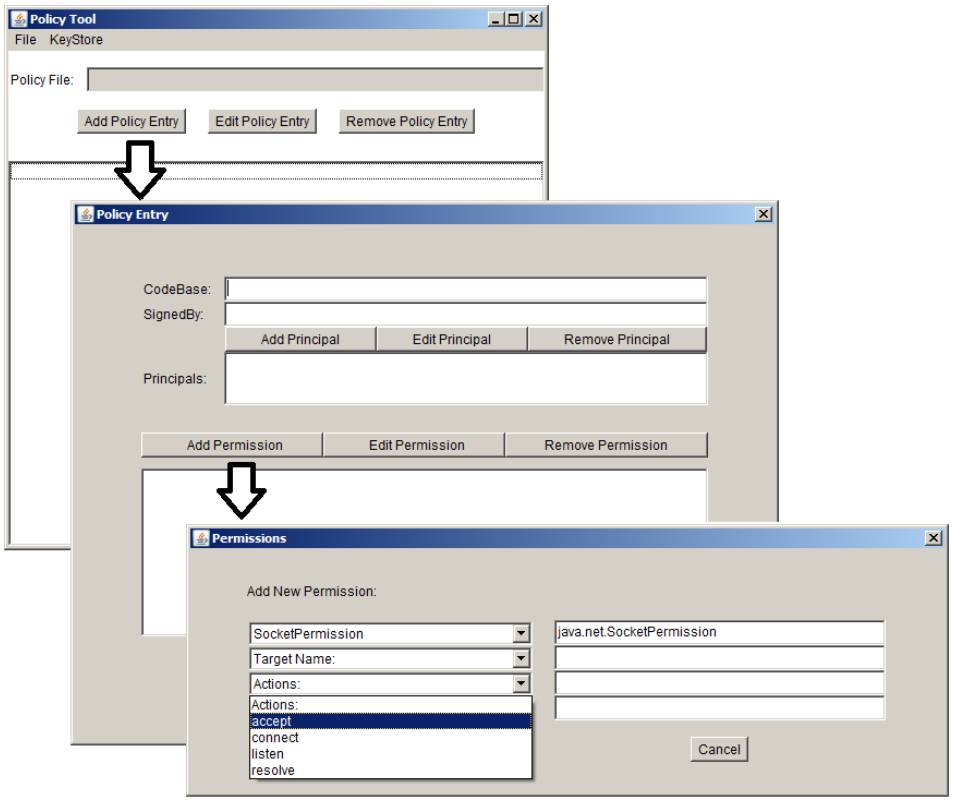
A grans trets, el seu funcionament és el següent, esquematitzat també a la figura 2.2.

Quan s'obre l'eina, es parteix d'un document buit. El menú *File* s'usa per desar el fitxer de polítiques un cop finalitzeu, generar-ne de nous o obrir d'altres que es vulguin editar.

A nivell d'opcions, les més destacades són:

- Allò que l'eina anomena *Entry* és un bloc `grant`. Se'n poden afegir de nous amb el botó *Add Policy Entry*, editar els existens amb *Edit* i esborrar-ne amb *Delete*.
- Dins de l'editor d'*Entries*, s'afegeixen els permisos. Novament, es poden afegir (*Add*), editar (*Edit*) o eliminar (*Delete*).
- El diàleg d'edició d'un permís es basa en tres menús desplegables. Un enumera tots els permisos possibles (*Permissions*), de manera que només cal triar l'adient. L'altra llista totes les accions (*Actions*) pel tipus de permís seleccionat. Finalment, el nom de l'objectiu s'ha d'escriure en un quadre de text (*Target Name*).

FIGURA 2.2. Ús de l'eina Policytool per afegir un nou permís



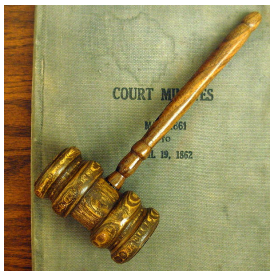
Tipus de permisos

Les classes associades als permisos solen estar definides al *package* relatiu al tipus d'acció a controlar (fitxers: `java.io`, xarxa: `java.net`, etc.)

Un cop sabeu com editar fitxers de polítiques, anem a veure un resum de quin és el conjunt de classes que defineixen tipus de permisos dins les biblioteques de Java. La llista es pot trobar a la taula 2.1.

TAULA 2.1. Classses associades a cada tipus de permís

Tipus de permís	Recurs controlat
`java.io.FilePermission`	Sistema de fitxers
`java.net.SocketPermission`	Connexions de xarxa a través de sòcols
`java.util.PropertyPermission`	Propietats del sistema
`java.lang.RuntimePermission`	Màquina virtual de Java, classes i fils d'execució
`java.awt.AWTPermission`	Interfície gràfica amb l'usuari
`java.net.NetPermission`	Connexions de xarxa a través d'HTTP
`java.lang.reflect.ReflectPermission`	Estructura interna de les classes
`java.io.SerializablePermission`	Seriació d'objectes
`java.security.SecurityPermission`	Mecanismes criptogràfics i polítiques de seguretat



Els permisos dins d'un fitxer de polítiques estableixen què pot fer una aplicació Java.

Els valors possibles pels paràmetres vinculats als noms d'objectius i acció concreta permesa varien segons el tipus de permís, ja que per cada cas només tenen sentit un seguit de valors o uns altres. Per poder disposar d'una llista exhaustiva, caldrà mirar la documentació de cada classe.

Tot seguit es mostra amb més detall tres d'aquests tipus de permisos, els més habituals en una aplicació.

java.io.FilePermission

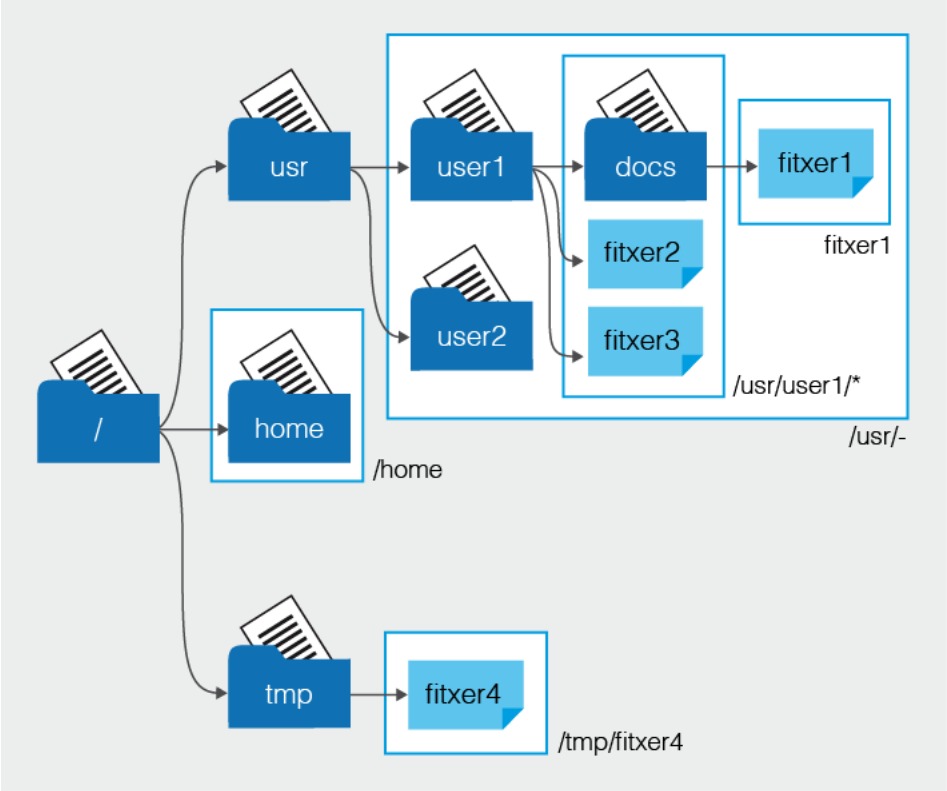
Aquest tipus de permís és un dels més utilitzats, ja que és el que permet establir excepcions en el control d'accés al sistema de fitxers.

El nom de l'objectiu que accepta aquest tipus de permís és el nom de la ruta, absoluta o relativa a l'aplicació, a la qual es vol permetre accedir. La sintaxi també accepta alguns caràcters especials per referir-se de manera senzilla a certs casos comuns. Posar al final de la ruta un asterisc (*) indica tot el contingut dins una carpeta concreta. Posar un guió (-) indica tots els fitxers i subcarpetes successives sota la ruta indicada. A part, existeix la cadena espacial `<<ALL FILES>>` que serveix per dir que s'aixequen les restriccions sobre tots els fitxers existents al sistema, sigui quina sigui la seva ruta.

Els noms d'objectiu d'una `FilePermission` també accepten, si es vol, un caràcter especial `${/}`. Aquest es tradueix al caràcter de separació dels elements d'una ruta que correspongui, segons el sistema operatiu de la màquina. A un sistema basat en Unix, es traduirà a "/" mentre que a un sistema Windows serà a "\".

La figura 2.3 mostra alguns exemples de rutes seguint la sintaxi esperada. En aquesta figura, es parteix de la premissa que el programa en Java s'executa a la carpeta `/usr/usr1/docs`, de cara a resoldre rutes relatives.

Figura 2.3. Especificació de rutes per seguint la sintaxi de l'objectiu d'un FilePermission



«ALL FILES»

El paràmetre que indica el valor d'aquest tipus de permís pot controlar quatre operacions possibles:

- `read`: permís de lectura.
- `write`: permís de lectura/escriptura. Es poden afegir o esborrar dades.
- `execute`: permís d'execució. És possible invocar un programa executable des del codi Java.
- `delete`: permís d'esborrament. El fitxer es pot eliminar des del codi Java.

java.net.SocketPermission

Aquest tipus de permís controla les connexions que fa una aplicació a Internet usant el mecanisme de sòcols. En aplicacions en xarxa és molt útil, ja que restringeix els equips als quals l'aplicació es pot connectar. Això pot evitar que un programa generi connexions a màquines no autoritzades.

Per referir-se a l'equip local on s'executa l'aplicació, es pot usar l'identificador *localhost*, o no posar-ne cap.

En aquest cas, el nom de l'objectiu que accepta aquest tipus de permís és el nom d'una connexió a Internet, o sigui, un parell de valors formats pel nom DNS de l'equip, o l'adreça IP directament, més el rang de ports del servei destí, separats per dos punts. Per exemple, si es vol donar permís perquè una aplicació es connecti al servidor web de l'IOC, usaria el nom d'objectiu "ioc.xtec.cat:80". També és possible usar el caràcter especial "*" per indicar qualsevol nom d'equip dins un domini.

En el cas del paràmetre que indica les accions que permet controlar aquest tipus de permís, hi ha les següents possibilitats. Aquestes es corresponen en major part a les diferents accions que es poden dur a terme amb les classes `java.net.Socket` i `java.net.ServerSocket`:

Un *bind* és el procés d'assignació d'un servei escoltant un port obert concret.

- `accept`: s'accepten connexions originades des de l'equip i port objectiu.
- `connect`: es permet establir connexions des de l'equip local a l'objectiu.
- `listen`: permet a l'objectiu posar en marxa un sòcol de servei (`java.net.ServerSocket`) per escoltar connexions entrants. O sigui, dur a terme el que s'anomena un *bind*.
- `resolve`: ens permet resoldre la traducció DNS del nom de màquina a una adreça IP.

Noteu que cap de les accions lligades al control mitjançant el gestor de seguretat tenen relació amb l'intercanvi de dades. Només es controla l'establiment de connexions.

java.util.PropertyPermission

Finalment, un altre tipus de permís que val la pena veure amb una mica de detall és la capacitat de llegir o alterar les propietats del sistema en el qual s'executa Java. Una propietat s'estructura com un parell: nom d'entrada = valor assignat. Per exemple : `java.vm.version = 23.6-b04`.

La llista de propietats és útil a la màquina virtual i a les aplicacions de cara a poder establir el sistema en el qual s'executen i saber com actuar en certes situacions. Per exemple, saber la versió de Java que s'està usant, la carpeta on està instal·lat o la carpeta personal de l'usuari que executa l'aplicació. Per un atacant poder obtenir, o fins i tot canviar, els valors d'aquestes propietats pot ser una bona oportunitat per dur a terme comportament maliciosos. Per exemple, si vol robar informació personal, saber la vostra carpeta personal és de molta utilitat per anar per feina i no haver de cercar per tot el disc.

Si voleu donar una ullada a les propietats del vostre sistema, es pot fer de manera senzilla amb el codi:

```
1 Properties props = System.getProperties();
2 props.list(System.out);
```

De cara a generar entrades al fitxer de polítiques d'acord amb la seva sintaxi, el nom de l'objectiu que cal posar és el del nom de la propietat. Les accions possibles són dues:

- `read`: permet consultar el valor d'una propietat. Això es pot fer amb el mètode `System.getProperty`.
- `write`: permet modificar en temps d'execució el valor d'una propietat. Això es pot fer amb el mètode `System.setProperty`.

Propietats del sistema al fitxer de polítiques

A diferents ordinadors, la ruta a la carpeta de l'usuari o el lloc on es troba instal·lat Java pot ser diferent. Això fa que un fitxer vàlid per un sistema concret no serveixi per un altre. Les propietats del sistema resulten molt útils per evitar això i crear fitxers de polítiques portables.

La sintaxi d'aquests fitxers també accepta incloure una propietat del sistema dins del paràmetre del nom d'objectiu, en el format `${nomPropietat}`, posant el nom de la propietat amb un signe de dòlar a l'inici i entre claus. Quan Java troba aquesta expressió, abans de prosseguir el tradueix al valor de la propietat corresponent.

Per exemple, si es vol donar permís per llegir els fitxers d'una carpeta anomenada *tmp* dins la carpeta de l'usuari que executa l'aplicació, es podria usar el nom d'objectiu `${user.home}${/}tmp`.

2.1.3 Codi signat

Les biblioteques de classes se solen empaquetar totes en un fitxer JAR.

Conèixer l'autor d'un codi Java pot ser un factor important de cara a decidir si el codi serà fiable i establir si cal executar-lo sota un gestor de seguretat i un seguit de polítiques. Aquest aspecte no es limita a saber qui el va generar originalment, sinó també poder estar segurs que ningú altre ha modificat la versió original, potser amb males intencions.

Una manera d'evitar aquesta mena de problemàtica seria disposar d'algun mecanisme que permeti controlar que una biblioteca, un cop generada, no ha estat alterada posteriorment. A la vegada, també caldria poder establir la identitat del seu creador, de manera que, abans d'instal·lar-la, puguem garantir que ha estat compilada per algú de confiança. I si fa coses insospitades, no pugui negar la seva autoria de manera que es dissuadeixi a atacants a generar codi maliciós, o com a mínim que sigui possible identificar-los fàcilment un cop hagin fet mal alguna vegada.

O sigui, es necessita garantir els serveis d'integritat, autenticació i no repudi als fitxers CLASS de les biblioteques Java. Per tant, la solució la trobem firmant digitalment el codi.

Jarsigner

L'entorn de treball de Java proporciona una eina per poder oferir unes mínimes garanties sobre l'origen del codi que s'executa.

L'eina **Jarsigner** permet signar digitalment i validar fitxers de biblioteques de classes, JAR. Aquesta es proporciona com un executable auxiliar, per línia de comandes, inclòs en totes les versions del *Kit de Desenvolupament de Java* (JDK).

La sintaxi d'ús del **Jarsigner** és molt senzilla i es resumeix de la següent forma.

```
jarsigner [opcions] nom_biblioteca alias_magatzem_claus
```

Totes les opcions del Jarsigner queden llistades en pantalla si s'executa sense cap paràmetre.

El següent exemple signa un fitxer JAR anomenat MyLib.jar mitjançant la clau privada emmagatzemada a l'àlies `dev1`, dins del fitxer dev1.jks (amb l'opció `-keystore`). El fitxer signat resultant queda desat amb el nom dev1-mylib.jar (amb l'opció `-signedjar`), de manera que no se sobreescriu l'original.

```
jarsigner –keystore dev1.jks –signedjar dev1–mylib.jar MyLib.jar dev1
```

Un cop un fitxer JAR està signat, la màquina virtual de Java en validarà la integritat mitjançant la firma abans de carregar-lo. Si la firma no és vàlida, prova que el fitxer ha estat modificat posteriorment de la generació de la firma, s'avorta l'execució del programa i es mostra un error del tipus `SecurityException`.

Validació mitjançant certificats de confiança

A part de veure si les dades que conté un JAR signat no s'han modificat posteriorment (integritat), també cal garantir la identitat de qui ha signat i la validesa del seu certificat. Ja que és possible generar certificats autosignats, aquest procés no és autocontingut. A partir només un certificat digital, i res més, no és possible saber si les dades que conté són reals o inventades. Cal veure qui l'ha emès i decidir si es considera algú fiable.

D'entrada, el procés de visualitzar la identitat continguda dins del certificat del signant d'un fitxer JAR es pot dur a terme manualment, abans d'executar el codi, mitjançant la mateixa eina Jarsigner. Aquesta també permet validar fitxers JAR signats i inspeccionar els certificats usats per signar el codi. Per fer-ho, cal usar l'opció `-verify` i indicar el nom del fitxer signat, amb les opcions `-verbose` i `-certs`.

Un exemple de sortida d'un fitxer signat correctament podria ser el que es mostra tot seguit. En cas de que el JAR no sigui vàlid, es mostra un error del tipus `java.lang.SecurityException`, indicant quin és el fitxer compromès detectat.

```
jarsigner –verbose –certs –verify dev–mylib.jar

...

sm      1650 Mon Apr 29 09:02:30 CEST 2013 ShowProperties.class

      X.509, CN=Developer 1, OU=DAM, O=IOC, L=Barcelona, ST=Catalunya, C=ES
      [certificate will expire on 28/07/13 08:46]
      [CertPath not validated: Path does not chain with any of the trust
          anchors]

...

jar verified.

Warning:
This jar contains entries whose signer certificate will expire within six
    months.
This jar contains entries whose certificate chain is not validated.
```

L'eina Jarsigner mostra una gran quantitat d'informació sobre els fitxers dins el JAR: estructura de carpetes, dates, etc. Però entre totes aquestes dades, hi ha uns quants aspectes destacables que val la pena comentar amb més detall.

Primer de tot, el resultat de tota la comanda es pot resumir en la breu frase, gairebé al final, de `jar verified`. O sigui, tot està en ordre des del punt de vista de la integritat del fitxer. No ha estat modificat després de la generació de la firma. Tot seguit, també es pot veure com, per cada fitxer protegit, es llista el certificat digital del signant. Finalment, dins del resultat del procés de validació, al final de tot de la sortida, hi ha un parell d'avisos, o *warnings*.

L'avís realment important, és el segon. Aquest diu, moltíssima atenció, que el fitxer ha estat signat amb la clau associada a un seguit de certificats que no estan validats (*whose certificate chain is not validated*). El significat del missatge és que els certificats i les firmes són correctes des d'un punt de visa criptogràfic, però Java no és capaç de saber si es pot fiar que les dades que conté el certificat siguin verídiques, ja que no coneix al seu emissor.

Un cas com aquest no vol dir que el signant sigui un atacant o que el certificat té dades inventades. Potser és un certificat generat sota un estricte control de seguretat, o potser se l'ha fet algú a casa seva amb Keytool. Simplement, no se sap. Com que Java no té prou informació per prendre una decisió de manera automàtica, llavors us informa al respecte perquè preneu les mesures que creieu adients. La decisió final és vostra.

Per poder prendre una decisió d'aquest tipus ell mateix, Java necessita disposar d'una llista d'emissors de certificats que vosaltres considereu de confiança (*trusted*). La decisió de quines entitats emissores són o no de confiança, i quins criteris usar per fer-ho, ja és cosa vostra. Amb qualsevol certificat emès per algú dins aquesta llista, Java considerarà que les dades que conté són automàticament verídiques. Per tant, per poder validar totalment la identitat d'algú que signa codi, abans us cal obtenir de manera segura el certificat digital del seu emissor.

> **Alerta**: si un atacant aconsegueix afegir un emissor de confiança a la vostra llista, us podrà passar identitats falses impunement!

La llista de certificats de confiança es desa dins d'un magatzem de claus, amb el seu àlies corresponent. Aquesta tasca es pot fer amb l'eina Keytool i l'opció `-import`.

El certificat de l'emissor d'un certificat autosignat és ell mateix.

Un cop teniu clar que `Developer 1` és algú de confiança, que fa codi no maliciós, obtindríeu el certificat del seu emissor i l'incloureu al vostre magatzem de claus de confiança. Aquest podria ser el d'una autoritat de certificació, o en el cas d'un certificat autosignat, el mateix certificat directament.

Si el vostre magatzem de claus es troba al fitxer user-cert.jks i el certificat de l'emissor el teniu al fitxer issuer.cer, el podeu importar dins el magatzem a l'àlies `dev1` amb la comanda:

```
1 keytool –import –alias dev1 –keystore user–certs.jks –file issuer.cer
```

Un cop es disposa d'un magatzem de claus amb una llista d'emissors de confiança, es pot passar al Jarsigner perquè el tingui en compte de cara a la validació amb l'opció `-keystore`.

```
jarsigner -keystore user-certs.jks -verbose -certs -verify dev-mylib.jar

...

smk     1650 Mon Apr 29 09:02:30 CEST 2013 ShowProperties.class

      X.509, CN=Developer 1, OU=DAM, O=IOC, L=Barcelona, ST=Catalunya, C=ES (
          dev1)
      [certificate will expire on 28/07/13 08:46]

...

jar verified.

Warning:
This jar contains entries whose signer certificate will expire within six
     months.
```

Si tot és correcte, com en aquest cas, la sortida ja no mostra els avisos relatius a la manca de confiança dels certificats.

Gestió de codi signat

Mitjançant la firma digital i la llista d'emissors de confiança, és possible establir, amb unes mínimes garanties, la identitat d'algú. Ara bé, sovint, en el nostre dia a dia, la identitat d'una persona en si no és una fita, sinó una eina per dur a terme alguna decisió.

Per exemple, suposeu que algú us demana permís per entrar al vostre portal. El seu nom en si, i ja està, no serveix de gran cosa. El que és important realment és que vosaltres relacionareu aquesta identitat amb algú que coneixeu o no, i decidireu si el voleu deixar passar. O potser fins i tot el tipus d'acció que li deixareu dur a terme serà diferent. Si és el carter, el rebeu a l'entrada i accepteu el paquet que us dóna. Si és un amic, el deixeu entrar a dins de casa. Si és algú de molta, molta confiança, potser ni tan sols us molestarà que obri la nevera i us agafi pel seu compte un refresc.

Aquest escenari també es pot dur a terme a nivell dels permisos que s'atorguen a una aplicació. Si s'executa codi signat, mitjançant un gestor de seguretat, és possible especificar diferents tipus de permisos per cada codi, depenent de la identitat del signant. Potser les aplicacions de `Developer 1` no volem que tinguin cap accés a la xarxa, mentre que les de `Developer 2` sí que volem que puguin resoldre noms, però limitar parcialment el seu accés al sistema de fitxers.

Per gestionar diferents permisos segons els signants d'un codi, Policytool permet configurar un magatzem de claus que serveix com a llista de certificats de confiança al menú principal de l'aplicació. Dins de l'apartat d'edició d'*Entries* es disposa del camp de text `SignedBy` en el qual cal posar el nom de l'àlies del certificat del signant a qui aplica aquest conjunt de permisos.

Per exemple, si es posa `dev1`, el grup de permisos només aplicarà al codi signat pel certificat desat al magatzem de claus sota l'àlies `dev1`, sigui quin sigui el nom de *Subject* del certificat.

Enlloc d'un únic àlies, també és possible afegir una llista d'àlies entre dobles cometes, separats per comes. En aquest cas, per garantir el permís cal que el codi estigui signat alhora pels certificats de tots els àlies. O sigui, si hi ha tres àlies, el codi ha d'estar signat pels tres. Molt de compte, ja que només que un dels àlies de la llista no hagi firmat el codi, es considerarà que no té permisos.

2.2 Connexions segures

L'estàndard del protocol SSL està definit al document RFC 6101: https://tools.ietf.org/html/rfc6101

Avui en dia moltes aplicacions fan ús de la xarxa per poder compartir informació. Per tant, és molt important garantir que aquests intercanvis es fan de manera segura. Això s'assoleix integrant dins les comunicacions en xarxa algun protocol que, fent ús de mecanismes criptogràfics, protegeixi les dades abans de ser enviades. Un dels protocols segurs més estesos actualment dins la indústria és **SSL** (*Secure Sockets Layer*, o capa de sòcols segurs, traduït al català). A manera il·lustrativa, aquest és el protocol que utilitzen tots els navegadors web per connectar-se a servidors segurs, com pot ser la banca o botigues en línia. El Java disposa de biblioteques que us permeten integrar-lo dins de les aplicacions que requereixen de comunicacions en xarxa.

Des del punt de vista d'un desenvolupador d'aplicacions, l'acceptació d'SSL garanteix un alt grau d'interoperabilitat entre les aplicacions, ja que permet comunicar aplicacions creades amb diferents llenguatges. A més a més, les biblioteques per defecte del Java ja disposen d'un conjunt de classes que permeten usar-lo sense gaires complicacions. Ambdós factors representen un gran avantatge respecte haver de dissenyar un protocol a mida per una aplicació concreta.

D'entrada, el protocol SSL garanteix sempre la privadesa i la integritat de les dades. Pel primer cas, això es fa mitjançant un sistema de clau simètrica, de manera que la seva eficiència sigui acceptable. La clau s'intercanvia de manera segura usant un algorisme d'intercanvi de claus. Pel que fa a la integritat, es fa ús d'un algorisme de *hash*.

> Un algorisme d'intercanvi de claus és un protocol criptogràfic mitjançant el qual és possible que dues entitats intercanviïn a distància una clau secreta de manera segura. Entre els més coneguts hi ha l'anomenat *Diffe-Hellman-Merkle*, que és el que usa SSL.

SSL també permet l'autenticació de les parts implicades mitjançant l'intercanvi de certificats digitals, en dos modes d'operació diferent. Per una banda, el més habitual és el mode d'**autenticació simple**, en el qual només s'identifica el servei. Tanmateix, també és possible fer que s'identifiquin tant el client com el servei, en mode d'**autenticació mútua**. Aquest cas és menys freqüent, ja que sovint en el

client s'usen sistemes, més senzills d'usar de cara als usuaris. Per exemple, l'ús de contrasenyes.

2.2.1 La negociació SSL

Quan s'inicia una connexió segura mitjançant SSL, les dues parts implicades (client i servidor) han de dur a terme prèviament un procés de negociació, o *handshake* en anglès, en el qual s'acorden els paràmetres i la informació criptogràfica que caldrà usar (bàsicament, les claus de xifrat o firma).

A la negociació SSL client i servidor es posen d'acord.

Tot seguit es descriuen les etapes d'aquest procés, indicant breument quines tasques es duen a terme en cada cas. El nom de cada etapa es precedeix pel nom de l'entitat qui l'executa (client o servei) i si és opcional o no.

1. **[CLIENT] Hello:** el client indica que vol iniciar una connexió segura SSL. Per fer-ho, indica fins i tot quina versió d'SSL, conjunt d'algorismes criptogràfics i mides de clau suporta.

2. **[SERVEI] Hello:** el servei accepta la petició i tria la darrera versió possible entre les suportades pel client de cara a usar el protocol segur. Llavors, escull l'algorisme i mida de clau més segur entre els disponibles. Un cop fetes aquestes tries, les comunica al client.

3. **[SERVEI, opcional] Enviament certificat:** aquesta etapa només es realitza si s'usa SSL per fer autenticació simple de servei, però cal dir que aquest és un cas molt habitual quan s'usa SSL. El servei envia el seu certificat digital al client, que conté la seva clau pública i la seva identitat.

4. **[SERVEI, opcional] Petició de certificat:** en cas d'usar SSL per fer autenticació mútua amb el client, el servei li demana el seu certificat digital.

5. **[SERVEI, opcional] Intercanvi de clau:** aquesta etapa només s'executa per alguns algorismes criptogràfics, que requereixen informació extra, a part d'una clau pública, per establir un canal de comunicacions segur. El servei envia al client aquesta informació addicional.

6. **[SERVEI] Hello finalitzat:** el servei passa el testimoni al client de cara a continuar amb el procés de negociació del protocol.

7. **[CLIENT, opcional] Enviament de certificat:** en cas que el servei demani el certificat del client a l'etapa 4, el client envia el seu certificat digital. Cal dir que aquest és un cas menys freqüent quan s'usa SSL.

8. **[CLIENT, opcional] Intercanvi de clau:** aquesta etapa és idèntica a l'etapa 5, però resolta per la part del client.

9. **[CLIENT, opcional] Validació certificat del client:** en el cas que es requereixi autenticació del client (veure etapes 4 i 7), s'envia al servei un missatge signat digitalment amb la clau privada del client. Això serveix

per demostrar que el client realment disposa de la clau privada associada al certificat que ha enviat, i no ha enviat el d'una altra entitat.

10. **[CLIENT] Canvi a mode xifrat:** el client avisa al servei que passa a mode xifrat de comunicacions. Fins ara, totes les comunicacions eren en clar.

11. **[CLIENT] Senyal de fi:** el client indica que la seva part de la negociació ha finalitzat i està llest per iniciar les comunicacions en mode segur. Aquesta etapa marca el final de la negociació SSL.

12. **[SERVEI] Canvi a mode xifrat:** el servei avisa que ell també passa a mode xifrat de comunicacions.

13. **[SERVEI] Senyal de fi:** el servei indica que, per la seva banda, també ha finalitzat la negociació. Aquesta etapa marca el final del *handshake* SSL.

14. **[CLIENT-SERVEI] Intercanvi de dades:** un cop finalitzada la negociació, el client i el servei porten a terme l'intercanvi de dades segur usant els paràmetres escollits a la negociació, fruit de les etapes anteriors.

15. **[CLIENT-SERVEI] Tancament de connexió:** un cop l'intercanvi de dades finalitza, es tanca la connexió a totes dues bandes.

Un cop la negociació finalitza i les dues parts ja han acordat tots els paràmetres, l'ús del protocol segur és transparent des del punt de vista de les aplicacions. Només cal gestionar l'enviament, recepció i processat de les dades intercanviades, sense haver-se de preocupar de cap aspecte criptogràfic. El protocol resol tots aquests detalls automàticament.

Tot i que normalment només s'usa quan s'inicia la connexió segura, la negociació SSL pot ser reiniciada en qualsevol moment dins d'un procés d'intercanvi de dades, de manera que es puguin canviar els paràmetres de manera dinàmica, si el client o el servei ho creuen adient.

2.2.2 Connexions SSL amb Java

Java disposa d'un seguit de biblioteques que permeten generar connexions sota SSL de manera relativament senzilla. Les classes implicades es troben a la biblioteca `javax.net.ssl`. D'entre aquestes, les més importants són `SSLServerSocket` i `SSLSocket`. Aquestes hereten de les classes que gestionen serveis no segurs dins la biblioteca `java.net`: `ServerSocket` i `Socket`, respectivament. En els dos casos, la primera serveix per escoltar connexions entrants a un servei, mentre que la segona serveix per gestionar l'intercanvi de dades, tant al servei com al client.

De fet, les diferències entre la versió de cada classe en format SSL o sense comunicacions segures només afecten al procés de negociació SSL i com configurar la validació dels elements criptogràfics intercanviats. Un cop la connexió s'ha

establert, la gestió de l'intercanvi de dades és exactament el mateix que si les dades s'enviessin sense seguretat.

Per veure més clarament com funciona l'ús d'SSL en una aplicació Java, el millor és veure un exemple d'aplicació client i servidor. Per començar, cal veure com generar una connexió amb autenticació simple, el més habitual. En aquest mecanisme, les dades estan xifrades i el client només accepta l'establiment de la connexió amb un servidor remot en el qual confiï. Si no és capaç d'establir aquesta confiança, interromp la connexió, ja que existeix el perill que algú l'estigui suplantant.

Després de veure això, cal afegir el codi per poder generar connexions autenticades mútuament.

Per disposar d'un magatzem amb una clau privada i un certificat, es pot usar l'eina Keytool.

Desplegament d'un servei amb autenticació simple

Abans de poder posar en marxa un servei fet en Java que accepti comunicacions mitjançant SSL, hi ha un procés previ de configuració de l'entorn. Aquest es deu al fet que tot servei que executi aquest protocol ha de comptar amb una clau privada i un certificat digital, que és el que usarà per dur a terme la negociació SSL amb qualsevol client que s'hi connecti. Aquests han d'estar desats a un magatzem de claus Java.

El magatzem ha de complir un seguit de condicions. Primer de tot, només ha de tenir una única entrada, amb un àlies qualsevol. Aquesta entrada conté el parell clau privada i certificat. Finalment, cal que la contrasenya del magatzem i la de l'entrada siguin exactament la mateixa. Si no es compleixen aquestes dues condicions, les classes que gestionen SSL a Java no sabran obtenir la informació que els cal per executar el protocol segur.

Un cop es disposa d'aquest magatzem, cal indicar al vostre programa la seva ruta dins el sistema de fitxers i la seva contrasenya. Això es fa mitjançant l'assignació dels valors adients a les propietats del sistema `javax.net.ssl.keyStore` i `javax.net.ssl.keyStorePassword`, respectivament.

Per assignar aquestes dues propietats del sistema, o bé amb el paràmetre `-D` de la línia de comandes a l'executar l'intèrpret de Java, o amb una crida al codi al mètode `System.setProperty`.

Fora d'aquest procés de configuració de les claus criptogràfiques del servei, la resta del procés és idèntic de com es programaria un servidor sense protocol segur, amb les classes `ServerSocket`, pel sòcol que accepta connexions, i `Socket`, pel sòcol de dades. Simplement són substituïdes per les classes `SSLServerSocket` i `SSLSocket`.

A nivell esquemàtic, les passes per configurar un servei amb connexió segura dins el codi Java són les següents:

1. Obtenir una fàbrica de sòcols, ja que aquests no es poden instanciar directament amb la sentència `new`.

2. Instanciar un `SSLServerSocket` usant el mètode `createServerSocket` sobre la fàbrica. En aquest pas s'indica el port del servei en el qual cal connectar-se.

3. Fer successives crides a `accept` per acceptar connexions entrants.

4. Servir al client a través del `SSLSocket` obtingut a l'acceptar una connexió entrant, al pas anterior.

Tot seguit es mostra un exemple de servidor mitjançant protocol segur SSL. Recordeu que per poder-lo provar correctament, cal generar un magatzem de claus. El més senzill és que useu un certificat autosignat generat automàticament amb l'eina Keytool.

```
keytool -genkey -alias "srvAlias" -keyalg RSA -keystore ServerKeyStore.jks -
    keysize 2048
```

El certificat del servidor web de l'IOC tindria com a *Common Name* el valor `ioc.xtec.cat`.

> Al generar un certificat per un servei, normalment es recomana que el camp de *Common Name* del seu nom distingit sigui el nom DNS de la màquina en la qual s'executa. Si ho proveu tot en local, podeu posar `localhost`.

En aquest cas, per establir la informació relativa al magatzem de claus, s'ha triat l'ús de crides a `System.setProperty`. El magatzem està a la mateixa carpeta del projecte de l'aplicació i s'anomena ServerKeyStore.jks. La contrasenya del magatzem i de la seva entrada amb la clau del servidor és *serverks*.

```
public class SSLServer {

  public static void main(String[] argv) throws Exception {

    System.setProperty("javax.net.ssl.keyStore", "ServerKeyStore.jks");
    System.setProperty("javax.net.ssl.keyStorePassword", "serverks");

    try {
      SSLServerSocketFactory sslFactory = (SSLServerSocketFactory)
          SSLServerSocketFactory.getDefault();
      SSLServerSocket srvSocket = (SSLServerSocket) sslFactory.
          createServerSocket(4043);

      int numClient = 1;

      while (true) {
        SSLSocket cliSocket = (SSLSocket) srvSocket.accept();

        Scanner reader = new Scanner (cliSocket.getInputStream());

        String text = reader.nextLine();
        while (!text.equals("<<FI>>")) {
          System.out.println("[Client " +numClient + "] " + text);
          System.out.flush();
          text = reader.nextLine();
        }
        System.out.println("[Client " +numClient + "] Tancant connexió...");
        cliSocket.close();
        numClient++;
      }
    } catch (Exception ex) {
      System.out.println("Error en les comuncacions: " + ex);
    }
```

```
32    }
33  }
```

Concretament, aquest servei va atenent a clients un a un, de forma no concurrent. A cadascun li assigna un identificador successivament i mostra per pantalla les dades que rep d'ell. La cadena de text `<<FI>>` serveix per indicar que el client ha acabat i, per tant, pot tancar la connexió i servir al següent.

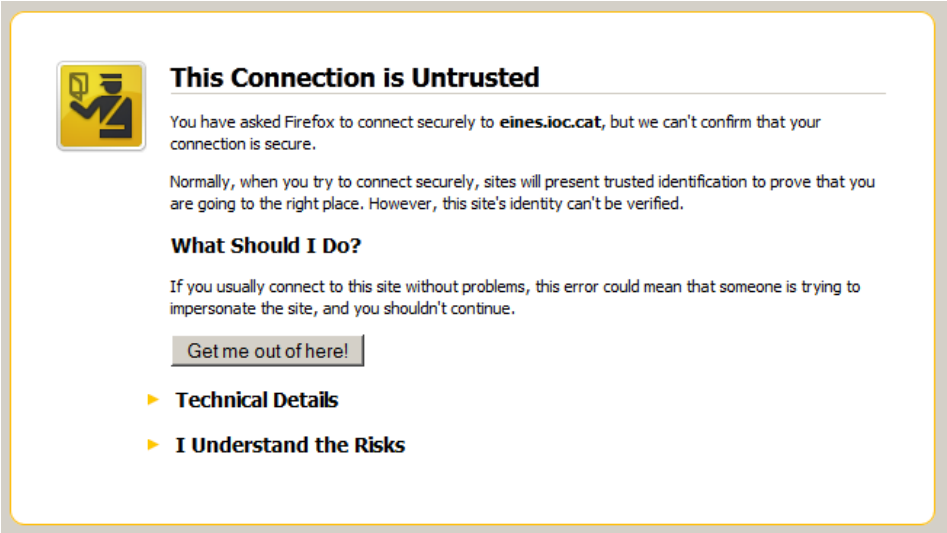
Desplegament d'un client amb autenticació simple

De la mateixa manera que passa amb un servidor, el client també necessita una configuració prèvia. En aquest cas, allò que cal especificar és la llista d'emissors de certificats digitals en els quals confia. D'aquesta manera, només s'acceptarà la creació d'una connexió cap a un servei que es consideri fiable. Si un servei s'identifica mitjançant SSL usant un certificat que el client no considera de confiança, la negociació fracassarà i no s'establirà cap connexió.

Els certificats d'emissors de confiança cal desar-los en un magatzem de claus Java especial, anomenat *truststore*, o magatzem de confiança. Aquest només conté entrades amb els certificats de les autoritats de certificació, o certificats autosignats, en els quals confiem. El client només es podrà connectar a serveis que hagin estat configurats amb un certificat emès per alguna de les entitats incloses al *truststore*.

Aquest comportament és comú a totes les aplicacions que usen SSL. Per exemple, la figura figura 2.4 mostra l'error que apareix al Navegador Mozilla Firefox (versió 22.0) quan es connecta un servidor web configurat amb un certificat que no es troba a la seva llista de confiança.

FIGURA 2.4. Error d'autenticació per la manca de confiança d'un servidor web en el navegador Mozilla Firefox 22.0.



Com passava amb el cas del servei, per indicar tota la informació vinculada a aquest magatzem de clau, cal usar propietats del sistema. La propietat `javax.net.ssl.trustStore` indica la ruta del fitxer, mentre que

`javax.net.ssl.trustStorePassword` indica la seva contrasenya. Novament, es pot optar per usar el flag `-D` per línia de comandes, o fer-ho a través de codi.

Fora de les particularitats lligades a la negociació SSL i les claus criptogràfiques, des de la perspectiva del client, l'establiment de la connexió també és gairebé idèntic a quan no s'usa protocol segur. Les passes serien les següents:

1. Obtenir una fàbrica de sòcols, ja que aquests no es poden instanciar directament amb la sentència `new`.
2. Instanciar un `SSLSocket` usant el mètode `createSocket` sobre la fàbrica. En aquest pas s'indica l'adreça i port del servei al qual cal connectar-se.
3. Iniciar la negociació SSL per establir la connexió segura.

Un cop fet, ja es pot iniciar l'intercanvi de dades.

El següent exemple mostra un client capaç de comunicar-se amb el servei anterior. Per fer-lo funcionar, abans cal configurar correctament el seu *truststore* Per això, haureu d'exportar el certificat autosignat des del magatzem de claus del servei i importar-lo al magatzem de confiança del client. Això es pot fer amb les comandes adients de Keytool.

Per exportar el certificat del magatzem del servidor, cal fer:

```
keytool -export -keystore ServerKeyStore.jks -alias "srvAlias" -file server.crt
```

A partir del fitxer resultant amb el certificat, per importar-lo es podria usar la comanda següent. Si el magatzem de claus no existeix, el crearà:

```
keytool -importcert -file server.crt -keystore ClientTrustStore.jks -alias "
    srvAlias"
```

Si voleu assegurar-vos que el contigut d'un magatzem de confiança és correcte, es pot usar la comanda:

```
keytool -list -v -keystore ClientTrustStore.jks
```

En el codi final, les propietats del sistema associades al *truststore* es configuren amb crides. El magatzem de confiança es considera que està al fitxer ClientTrustStore.jks, protegit amb la contrasenya `clientts`.

```
public class SSLClient {

  public static void main(String[] argv) throws Exception {

    System.setProperty("javax.net.ssl.trustStore", "ClientTrustStore.jks");
    System.setProperty("javax.net.ssl.trustStorePassword", "clientts");

    try {
      SSLSocketFactory sslFactory = (SSLSocketFactory) SSLSocketFactory.
          getDefault();
      SSLSocket cliSocket = (SSLSocket) sslFactory.createSocket("localhost",
          4043);

```

```
12        Scanner reader = new Scanner (System.in);
13        PrintStream writer = new PrintStream(cliSocket.getOutputStream());
14
15        System.out.println("Deixa una línia en blanc per acabar:");
16        String text = reader.nextLine();
17        while (!text.equals("")) {
18          writer.println(text);
19          writer.flush();
20          text = reader.nextLine();
21        }
22        writer.println("<<FI>>");
23        writer.flush();
24        cliSocket.close();
25      } catch (Exception ex) {
26        System.out.println("Error en les comuncacions: " + ex);
27      }
28    }
29  }
```

Aquest client simplement llegeix dades pel teclat i les envia al servidor en format text, tal com les ha llegides. Si en algun moment es pitja només la tecla de retorn, deixant la línia en blanc, aquest acaba. Abans, però, es duu a terme una desconnexió ordenada enviant el text `<<FI>>`, que serveix per avisar el servei que el client ha acabat.

Quan es fan proves amb clients i serveis amb SSL, sovint pot passar que hi hagi algun error en la negociació SSL i no es vegi a simple vista. Per casos com aquest, hi ha una propietat del sistema anomenada `javax.net.debug` que permet posar en marxa la depuració. El valor que cal assignar per fer-ho és:

```
1  System.setProperty("javax.net.debug", "SSL,handshake");
```

Desplegament d'autenticació mútua

En un intercanvi de missatges autenticat mútuament, tots els clients també s'han d'identificar davant del servidor mitjançant un certificat digital de confiança. En cas contrari, el servei rebutjarà l'intent de connexió, al no poder garantir la identitat del client. Aquest mecanisme és menys freqüent que l'autenticació simple ja que implica que tots els clients han de disposar d'una clau privada i un certificat digital per identificar-se. Tradicionalment, per identificar els clients se sol recórrer a eines com ara contrasenyes, que sovint es consideren més còmodes d'usar.

Des del punt de vista del codi Java, les rols de client i servidor s'inverteixen, i per tant l'autenticació mútua implica desplegar tant al client com al servei totes les dades que en l'autenticació simple només calia a un extrem:

- El client ha de disposar d'un magatzem criptogràfic amb una clau privada i un certificat digital.
- El servei ha de disposar d'un magatzem on desar els certificats de les entitats de certificació de confiança. Només acceptarà connexions de clients que proporcionin un certificat emès per alguna de les autoritats incloses en aquest magatzem.

En un cas com aquest, sovint se sol usar un únic fitxer que fa alhora de magatzem de claus privades i de certificats de confiança.

La manera com es desplega aquesta informació és igual que abans, mitjançant propietats del sistema. Per exemple, en el client ara caldria afegir les propietats que declaren quin és el magatzem de claus i la seva contrasenya, tal com mostra el següent fragment de codi.

```
public class SSLClient {

  public static void main(String[] argv) throws Exception {

    //Magatzem de confiança del client
    System.setProperty("javax.net.ssl.trustStore", "ClientTrustStore.jks");
    System.setProperty("javax.net.ssl.trustStorePassword", "clientts");
    //Clau del client
    System.setProperty("javax.net.ssl.keyStore", "ClientKeyStore.jks");
    System.setProperty("javax.net.ssl.keyStorePassword", "clientks");
    ...
```

Pel cas del servei, seria exactament igual, però amb els seus fitxers.

Un cop totes les parts implicades disposen de tota la informació criptogràfica, cal configurar el servei perquè requereix autenticació mútua, ja que per defecte actua amb autenticació simple. Això es fa invocant el mètode `setNeedClientAuth` sobre el seu sòcol de servei, l'objecte `SSLServerSocket`.

Si un client s'intenta connectar a un servei que requereix autenticació mútua, la negociació SSL fracassarà. En aquest cas, les dades que rebrà el client si estableix cap tipus d'intercanvi de dades amb el servei seran sempre invàlides.

```
public class SSLServer {

    ...

    try {
      SSLServerSocketFactory sslFactory = (SSLServerSocketFactory)
          SSLServerSocketFactory.getDefault();
      SSLServerSocket srvSocket = (SSLServerSocket) sslFactory.
          createServerSocket(4043);
      srvSocket.setNeedClientAuth(true);
      ...
```

També és possible configurar un servei de manera que demani als clients autenticar-se, però només de manera opcional. Si els clients no disposen de cap certificat, ja que no estan configurats per desplegar autenticació mútua, la negociació segueix igualment endavant. Per fer-ho, cal usar el mètode `setWantClientAuth`. Si es criden els dos mètodes dins del codi del programa, `setNeedClientAuth` sempre té prioritat, independentment de l'ordre com s'invoquin.

2.2.3 Gestió personalitzada de la informació criptogràfica

Les biblioteques d'SSL que incorpora Java permeten crear connexions segures de manera relativament simple. Aquestes s'encarreguen de gestionar de manera automàtica la gran majoria dels detalls vinculats a les dades criptogràfiques. Com a desenvolupadors, només cal indicar la ruta dels magatzems criptogràfics

corresponents, les seves contrasenyes, i ja està. En realitat, no cal saber els detalls sobre com funciona tot plegat internament.

Tot i així, sempre cap la possibilitat que el comportament per defecte de les biblioteques SSL de Java no siguin suficients pels requeriments d'una aplicació, ja que el seu funcionament per defecte és relativament simplista. Per una banda, les claus i certificats han d'estar sempre a un fitxer local. D'altra banda, el model d'autenticació es basa exclusivament en si el certificat aportat per un extrem ha estat emès per una entitat de confiança. Si és així, es permet la connexió i en cas contrari, es descarta. No hi ha possibilitat d'usar altres mecanismes més complexos per gestionar claus i certificats, o filar més prim a partir de la informació continguda al certificat per decidir si s'accepta la connexió.

Afortunadament, Java ofereix una via d'accés a les dades criptogràfiques a més baix nivell, permetent la personalització d'alguns aspectes vinculats a la gestió de la informació criptogràfica i a la validació dels certificats.

Inspecció de la identitat als extrems

Un cop s'ha establert una connexió mitjançant SSL, pot ser interessant poder consultar la informació continguda dins el certificat digital de l'altre extrem, ja sigui servei o client. Per exemple, poder saber saber quina és la identitat exacta de l'extrem a través de les dades contingudes al nom distingit del propietari o l'emissor del certificat: el seu nom (al camp *Common Name*), la seva organització (al camp *Organization*), etc.

> La negociació SSL només permet garantir que l'extrem disposa d'una identitat validada per un emissor de certificats de confiança, però no quina és aquesta identitat, o quines accions pot dur a terme.

Per tant, es pot donar el cas que, tot i saber que un extrem té accés i es pot connectar, també sigui necessari saber-ne la identitat exacta per actuar en conseqüència. El cas més fàcil de conceptualitzar és quan un client es connecta a un servei. Per exemple, en un cas molt senzill, potser es vol donar un aspecte personalitzat a l'aplicació, mostrant el nom de l'usuari. En un entorn més complex, pot donar-se el cas que, diferents usuaris que poden accedir a l'ítem, i que disposen de certificats admesos, disposin de diferents privilegis depenent del departament. Per tant, què poden fer depèn dels camps adients del certificat presentat.

Per consultar la informació vinculada a la identitat de l'extrem en una connexió segura SSL, existeix la classe `SSLSession`. Tota connexió segura disposa d'un objecte d'aquest tipus associat, accessible mitjançant la invocació del mètode `getSession()` sobre el sòcol segur de dades (l'objecte `SSLSocket`).

A la documentació de l'API de Java podeu trobar els mètodes de les classes `Certificate` i `X500Principal`, per tal d'accedir a la informació que contenen.

Entre els seus mètodes destacats podeu trobar:

- `String getCipherSuite()`: permet veure quin algorisme ha decidit usar Java per protegir les dades, com a resultat de la negocicació SSL.

- `Certificate[] getPeerCertificates()`: permet obtenir el certificat que ens ha enviat l'altre extrem amb qui s'estan intercanviant missatges. Per exemple, si s'executa al client, ens diu quin certificat ha enviat el servei, i viceversa en el cas d'autenticació mútua.

- `Principal getPeerPrincipal()`: obté el nom distingit dins del *Subject* del certificat de l'altre extrem. Això permet obtenir la identitat de l'extrem sense haver de processar el certificat. Normalment, retorna un objecte de tipus `javax.security.auth.x500.X500Principal`.

- `String getPeerHost()`: obté el nom de l'equip amb qui s'intercanvien missatges.

Personalització del gestor de confiança

També existeix la interfície `TrustManager`, que és aplicable a altres tipus de credencials a part de certificats.

Una altra opció força interessant que ofereixen les biblioteques d'SSL de Java és permetre redefinir els criteris per considerar si un certificat aportat per l'altre extrem a les comunicacions s'ha de considerar admissible o no. Això es fa a través de la implementació de la interfície `javax.net.ssl.X509TrustManager` i la integració de la nova classe al motor d'SSL.

Java ja proporciona una implementació per defecte, que és tot just la que executa el procediment mitjançant el qual es consulten les propietats del sistema `trustStore` i `trustStorePassword`. S'obre el fitxer corresponent i carrega la llista amb tots els certificats d'emissors de confiança que a partir de llavors, s'usa com a referència per veure si cal acceptar o no un certificat. Però en una implementació personalitzada es pot fer una classe que llegeixi els certificats d'un altre lloc, com un servei remot, o que apliqui altres polítiques de validació.

Per implementar la interfície `X509TrustManager` cal fer el codi dels següents mètodes:

- `void checkClientTrusted(X509Certificate[] certs, String tipus) throws CertificateException`: el codi que s'escriu en aquest mètode serveix per indicar si un certificat subministrat per un client remot es considera vàlid o no. Per tant, només té sentit implementar-lo a un servei. Si es considera correcte, el mètode simplement acaba (no hi ha valor de retorn). Per indicar que el certificat no es considera vàlid, es llança una excepció del tipus `CertificateException`. El seu primer paràmetre, `certs`, és un *array* amb el certificat del client i del seu emissor. En cas de ser un certificat autosignat, només hi haurà un únic certificat. El paràmetre `tipus` és una cadena de text que indica l'algorisme emprat per l'autenticació. Per exemple, si les claus del client són RSA, seria el text `RSA`.

- `void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException`: aquest és el cas per validar un certificat obtingut des d'un servei remot. Per tant, només té

sentit posar codi a si s'usa a una aplicació client. La resta de detalls són exactament els mateixos que al mètode anterior.

- `X509Certificate[] getAcceptedIssuers()`: el codi d'aquest mètode ha de retornar un *array* amb una llista dels certificats d'emissor considerats de confiança.

El següent codi mostra un gestor personalitzat molt simple. Per una banda, com que el mètode `checkServerTrusted` sempre llança una excepció, això vol dir que si el gestor s'usa a un client, aquest mai acceptarà cap certificat d'un servei com a vàlid. D'altra banda, ja que el mètode `checkClientTrusted` no fa res i, per tant, mai llança cap excepció, això vol dir que si s'usa el gestor en un servei, aquest acceptarà qualsevol certificat subministrat per qualsevol client. Alerta, ja que aquest mètode comprova la validesa global del certificat, no només si el seu emissor és de confiança. Per tant, amb codi buit, accepta certificats encara que tinguin una signatura incorrecta (i per tant, falsos) o una data d'expiració invàlida, s'acceptarà.

```
1  public class MyTrustManager implements X509TrustManager {
2
3    @Override
4       public void checkClientTrusted(X509Certificate[] chain, String authType)
5                                      throws CertificateException {
6       }
7
8    @Override
9       public void checkServerTrusted(X509Certificate[] chain, String authType)
10                                     throws CertificateException {
11        throw new CertificateException();
12      }
13
14   @Override
15      public X509Certificate[] getAcceptedIssuers() {
16          return new X509Certificate[0];
17      }
18 }
```

En cas que un dels mètodes de validació no s'hagi d'implementar, ja que no s'usarà, sempre és millor que al seu codi es llanci la `CertificateException`. O sigui, si s'està creant un gestor per una aplicació client, fer que el codi del mètode `checkClientTrusted` tingui només el codi `throw new CertificateException()`. Això evita que si s'usa per error en un tipus d'aplicació incorrecta, aquesta mostri un error enlloc d'acceptar qualsevol certificat sense adonar-vos-en.

Una biblioteca oculta no està documentada a l'API de Java, però es pot accedir igualment a les seves classes.

Normalment, una implementació de *X509TrustManager* tindrà mètodes amb codi. Ara bé, una dificultat important al fer-ho és que, a part de les comprovacions personalitzades que vosaltres vulgueu fer, també ha de fer totes les vinculades a la correcció del certificat. En aquest sentit, no només cal comprovar que l'emissor és de confiança. A més a més, cal veure si les dades contingudes al certificat són correctes: la signatura digital, la data d'expiració, etc. Això no és trivial de fer, ja que significa aplicar operacions criptogràfiques. Afortunadament, Java ofereix la classe `Validator` a la biblioteca oculta `sun.security.validator`. Aquesta

permet validar certificats fàcilment. De fet, és la que usa el gestor de confiança per defecte.

Per instanciar una classe validadora, cal usar el mètode estàtic `getInstance` de la classe `Validator`. Aquest té tres paràmetres. Pel cas de certificats digitals, el primer sempre és el definit per la constant `Validator.TYPE_PKIX`. El segon també es basa en una altra constant, que indica si estem executant la validació a una aplicació client (`Validator.VAR_TLS_CLIENT`) o a un servei (`Validator.VAR_TLS_SERVER`). El tercer és la llista de certificats de confiança. Els certificats d'aquesta llista poden provenir d'on vulgueu, no necessàriament d'un únic magatzem de claus, com passa al gestor per defecte.

Un cop fet, la validació del certificat es fa amb el mètode `validate`. El paràmetre és un *array* amb el conjunt de certificats aportat per l'altre extrem de les comunicacions, i que cal validar. De fet, aquest valor ja us el proporciona el motor d'SSL a través del primer paràmetre de `checkClientTrusted` i `checkServerTrusted`.

Concretament, el mètode `validate` comprova:

- El format del certificat
- La data d'expiració
- Si el seu emissor es correspon a algun de la llista proporcionada
- La firma digital del certificat, a partir de la clau pública de l'emissor localitzat

Si alguna cosa no és correcta, llança una excepció del tipus `CertificateException`.

El següent codi mostra un exemple de gestor de confiança una mica més complex, pensat només per una aplicació client. Aquest carrega els certificats d'emissors de confiança des d'un magatzem de claus i els usa per validar el certificat d'un servidor. Si el certificat és correcte, mostra per pantalla el del seu emissor. Estudieu-lo amb atenció per veure com s'extreuen els certificats de confiança del magatzem de claus original, i després s'usen als diferents mètodes per validar el certificat d'un servei.

```
1  public class MyTrustManager implements X509TrustManager {
2
3    private ArrayList<X509Certificate> certs = new ArrayList<X509Certificate>();
4
5    public MyTrustManager() throws Exception {
6      KeyStore keyStore = KeyStore.getInstance("JKS");
7      keyStore.load(new FileInputStream("SSL/ClientKeyStore.jks"), "client".
           toCharArray());
8      Enumeration<String> aliases = keyStore.aliases();
9      while(aliases.hasMoreElements()) {
10       String a = aliases.nextElement();
11       if (keyStore.isCertificateEntry(a)) {
12         certs.add((X509Certificate)keyStore.getCertificate(a));
13       }
14
15     }
16   }
```

```
17
18    @Override
19       public void checkClientTrusted(X509Certificate[] chain, String authType)
20                   throws CertificateException {
21         throw new CertificateException("Aquest gestor només és per aplicacions
               client.");
22       }
23
24    @Override
25       public void checkServerTrusted(X509Certificate[] chain, String authType)
26                   throws CertificateException {
27           try {
28             Validator val = Validator.getInstance(Validator.TYPE_PKIX, Validator
                   .VAR_TLS_CLIENT, certs);
29             X509Certificate[] result = val.validate(chain);
30             for (X509Certificate c: result) {
31               System.out.println("El següent certificat de confiança valida el
                     servei:");
32               System.out.println(c);
33             }
34           } catch (Exception e) {
35             throw new CertificateException("El certificat del servei no és de
                   confiança.");
36           }
37       }
38
39    @Override
40       public X509Certificate[] getAcceptedIssuers() {
41         X509Certificate[] issuers = new X509Certificate[certs.size()];
42         for (int i = 0; i< issuers.length; i++){
43           issuers[i] = certs.get(i);
44         }
45         return issuers;
46       }
47  }
```

Un cop es disposa d'una implementació correcta de la interfície `X509TrustManager`, cal integrar-la al motor d'SSL de les biblioteques de Java, de manera que reemplaci al comportament per defecte. Això es fa amb l'ajut de la classe `SSLContext` i el seu mètode `init`. Un petit inconvenient és que aquest sistema obliga a indicar quin gestor de claus, o *KeyManager*, cal usar.

La interfície usada per personalitzar un gestor de claus és la `X509KeyManager`.

De la mateixa manera que amb el gestor de confiança, Java permet personalitzar també el mecanisme de gestió de claus, associat a les propietats `keyStore` i `keyStorePassword`, de manera que s'obre la porta a usar altres sistemes que un magatzem de claus amb una única entrada amb una clau privada. Ja que la interfície associada és força més complicada que la del gestor de confiança, la solució recomanada és crear una instància del gestor de claus per defecte i assignar-lo, enlloc de fer-ne un de personalitzat.

La integració es fa tal com mostra el següent fragment de codi, generant un objecte `SSLSocketFactory` especial. En lloc d'usar el que es proporciona per defecte amb la crida estàtica `SSLSocketFactory.getDefault()`, se'n genera un a partir de la classe `SSLContext`. El mètode clau és `init`, que és el que serveix per configurar un context d'SSL perquè utilitzi gestors personalitzats. El tercer paràmetre és un objecte `SecureRandom` qualsevol.

```
1  //Inicialització d'un gestor de claus per defecte.
2    //Les antigues propietats "keyStore" i "keyStorePassword" són paràmetres
         directament
3    KeyManagerFactory kmf = kmf = KeyManagerFactory.getInstance("PKIX");
```

```
4   FileInputStream fin = new FileInputStream("SSL/ClientKeyStore.jks");
5   KeyStore ks = KeyStore.getInstance("JKS");
6   ks.load(fin, "client".toCharArray());
7   kmf.init(ks, "client".toCharArray());
8
9   //Creació del gestor de confiança personalitzat
10  TrustManager[] myTM = new TrustManager[] { new MyTrustManager() };
11
12  SSLContext sslContext = SSLContext.getInstance( "SSL" );
13  sslContext.init( kmf.getKeyManagers(), myTM, new java.security.SecureRandom()
        );
14
15  SSLSocketFactory sslFactory = sslContext.getSocketFactory();
16  Socket cliSocket = sslFactory.createSocket("localhost", 10000);
17  ...
```

Un cop fet, tots els sòcols generats a partir d'aquest context usarà els nou gestor de confiança. Pel cas d'un sòcol de servei, caldrà usar el mètode `getServerSocketFactory()` de `SSLContext` a la darrera línia.