UF1.NF1 – 2. Criptografia clau secreta

CFGS Desenvolupament d'Aplicacions Multiplataforma

Mòdul 9: Programació de serveis i processos

Apunts

Índex

Bibliografia

Java™ Cryptography Architecture (JCA) Reference Guide

http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

1. Protocol

El protocol de la criptografia de clau secreta és el següent:

1. A i B es posen d'acord en el sistema criptogràfic a utilitzar.
2. A i B es podin d'acord en la clau a utilitzar.
3. A xifrat un missatge usant l'algorisme i clau acordats.
4. A envia el missatge xifrat a B.
5. B desxifra el missatge usant l'algorisme i clau acordats.

La criptografia de clau secreta es pot usar bàsicament per a un d'aquestes dues finalitats:

- Xifrar canals de comunicació.
- Xifrar dades per al seu emmagatzematge.

1.1. XIFRAT DE CANALS DE COMUNICACIÓ

Encara que en teoria el xifrat es pot realitzar a qualsevol nivell de la capa OSI, en la pràctica se sol realitzar en un d'aquests quatre nivells:

- Xifrat a nivell d'enllaç de dades
- Xifrat a nivell de xarxa
- Xifrat a nivell de transport
- Xifrat a nivell d'aplicació

1.2. XIFRAT DE DADES

El xifrat de dades per al seu emmagatzematge pot modelar-se com un enviament de dades des de A fins a B, però en aquest cas l'emissor i el receptor són els mateixos.

No obstant això, apareixen alguns problemes nous que no existien abans.

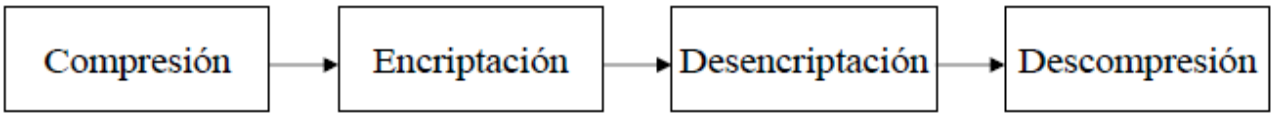
1. En el cas de les bases de dades xifrades no és eficient desxifrar tota una base de dades, ja que per accedir a un registre hauríem de desxifrar tota la base de dades, després **el xifrat es realitza per registres individuals**. Un altre problema són les cerques, en les quals caldria anar desxifrant tots els registres. Per evitar-ho s'utilitzen índexs.
2. En el xifrat de fitxers existeixen dues opcions: **Xifrar a nivell d'unitat de disc**, o **xifrar a nivell de fitxer**. La segona opció té l'inconvenient que si el criptoanalista veu els noms dels fitxers, pot obtenir molta informació sense tenir accés al seu contingut. Per contra, la primera opció té l'inconvenient que si es corromp la unitat es perden totes les dades.

2. Compressió, xifrat i control d'errors

A la pràctica és molt típic usar algorismes de xifrat juntament amb algorismes de compressió. Això és així per dues raons:

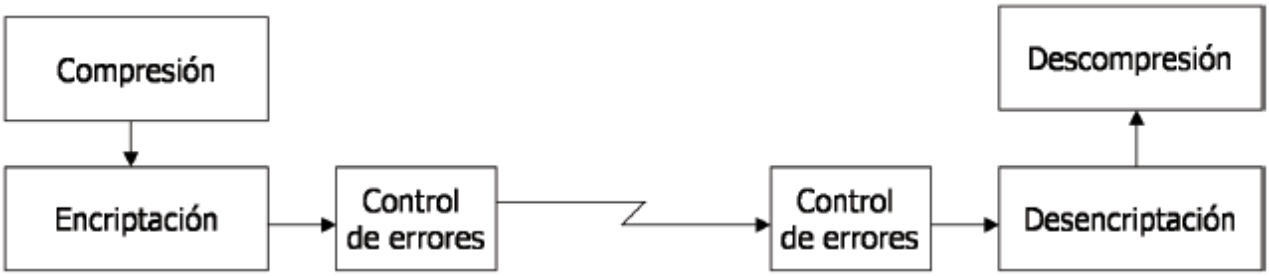
1. Una tècnica de criptoanàlisi molt usada consisteix a buscar redundàncies en el missatge xifrat, ja que els missatges plans solen tenir redundàncies. Si comprimim abans de xifrar estem eliminant aquestes redundàncies.
2. El xifrat és un procés costós que es pot reduir si primer comprimim el missatge.

És important que la compressió es realitzi abans del xifrat, o en cas contrari no podríem aprofitar els avantatges anteriors.



A més, els textos xifrats es comprimeixen molt poc, ja que un dels objectius que busca un algorisme de xifrat és que el missatge xifrat s'assembli a una sèrie aleatòria de bits.

D'altra banda, podem afegir un sistema de control d'errors, en aquest cas, es recomana afegir-ho després del xifrat.



D'aquesta forma, si el control d'errors detecta que les dades han arribat malament, no fa falta executar el procés costós de desxifrat.

3. DES (Data Encryption Standard)

DES és un algoritme de xifrat inventat inicialment per IBM en 1970 sota el nom "Llucifer".
En 1977 va ser adoptat pel NIST (National Institute of Standards and Technology) i per la*NSA (National Security Agency) com el seu estàndard nacional.
En 1981 aquest algorisme va ser adoptat també per ANSI.

DES és un xifrador de bloc amb grandària de bloc de 64 bits. Per a cada bloc de 64 bits d'entrada s'obtenen 64 bits a la sortida, és a dir, la grandària del fitxer resultant no creix ni decreix.
La clau és de 56 bits. Se sol expressar com un nombre de 64 bits, però l'últim bit de cada byte és de paritat, i es treu abans d'aplicar l'algoritme.

L'algoritme es limita a aplicar **substitucions** (canvis d'uns valors per uns altres) i **permutacions** (canvis en la posició que ocupen els bits). El procés de xifrat es realitza en 16 rounds, i cada round duu a terme la mateixa sèrie de substitucions i permutacions.
Després de la permutació inicial, el bloc es divideix en dos blocs de 32 bits, s'apliquen els 16 rounds, i al final es tornen a ajuntar les meitats i s'aplica una última permutació que és la inversa de la permutació inicial.

3.1. DESXIFRAT DE DES

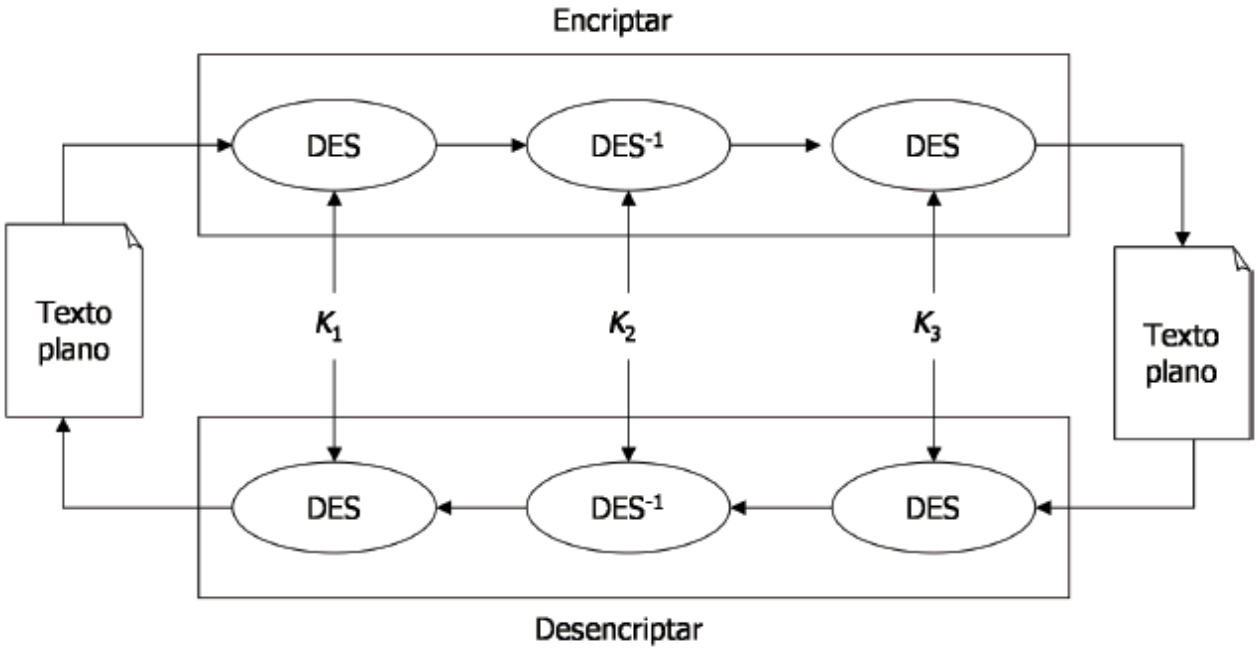
Després de totes les transformacions que es fan durant el xifrat, podem pensar que el desxifrat implica un altre munt d'operacions encara major. No obstant això, les operacions estan triades perquè **el mateix algoritme serveixi per a el xifrat i el desxifrat**, és a dir, podem usar la mateixa funció per xifrar i per desxifrar.
L'única diferència és que **la clau ha d'usar-se en ordre invers**, és a dir, si les claus usades en el xifrat van ser: k1, k2, k3,… k16 en el desxifrat usarem k16, k15, k14,…k1.

3.2. TRIPLEDES

El punt feble de DES no està en l'algorisme, sinó en la grandària de la clau, que és només de 56 bits, la qual cosa dóna lloc a massa poques claus possibles i és susceptible de sofrir un atac per força bruta. Actualment un algorisme de clau secreta es considera fort si té una clau de 128 bits o major.

Això ha donat lloc al fet que des de fa anys estiguin sorgint sistemes que desxifren DES en temps cada vegada menor.

Per acabar amb aquest problema han sorgit diverses variants de DES que sí que són segures. La més coneguda d'aquestes és TripleDES (també anomenat DESede), que no és més que **DES aplicat tres vegades amb tres claus diferents**. La primera vegada en mode de xifrat amb la primera clau, la segona en mode de desxifrat amb la segona clau, i la tercera en mode de xifrat amb la tercera clau.



El nom DESede ve just de la forma en què s'aplica l'algoritme, (Encryptation, Decryptation, Encryptation).

En TripleDES la grandària de la clau és de 168 bits (3*56 bits) que és suficientment segura com per eludir un atac per força bruta actual.

4. Altres algoritmes de clau secreta

4.1. BLOWFISH

És un algoritme inventat per Bruce Schneier en 1993. No està patentat, sinó que és de domini públic. Respecte als seus principals característiques, destaca que permet usar una clau variable de fins a 448 bits, i que està optimitzat per a microprocessadors de 32 bits i de 64 bits.

4.2. AES

Advanced Encryption Standard

És l'algoritme triat a l'octubre del 2000 pel NIST com a substitut a DES. Utilitza un algorisme anomenat Rijndael, claus de 128, 192 i 256 bits, i grandàries de bloc de 128, 192 i 256 bytes respectivament.

5. Modes de l'algoritme

Existeixen dos tipus d'algoritmes de xifrat: **xifradors en bloc** (el missatge es xifra/desxifra en blocs de 64 o 128 bits) i **xifradors en flux** (el missatge es xifra/desxifra bit a bit).

5.1. MODES EN ELS XIFRADORS DE BLOC

5.1.1. ECB. Electronic Code Book mode

Aquesta és la forma més senzilla de fer un xifrador de bloc: **Cada bloc de text pla xifra en un bloc de text xifrat**. En aquesta manera, com **el mateix text pla sempre dóna lloc al mateix text xifrat**, és possible fer un code book, és a dir, una taula que ens digui per a cada text pla quin és el seu corresponent text xifrat.

Avantatges:

- Al no dependre un bloc dels altres blocs, podem xifrar o desxifrar parts aïllades d'un fitxer, la qual cosa és molt útil per a aplicacions com les bases de dades xifrades, en les quals anem a poder accedir als seus registres aleatòriament.
- El procés de xifrat es pot paral·lelitzar.
- Si hi ha un error en la comunicació, només perdem el bloc afectat per l'error, ja que els blocs són independents uns d'uns altres. No obstant això, si accidentalment s'afegeix o perd un bit, s'espatlla tot el missatge a partir d'aquest bit.

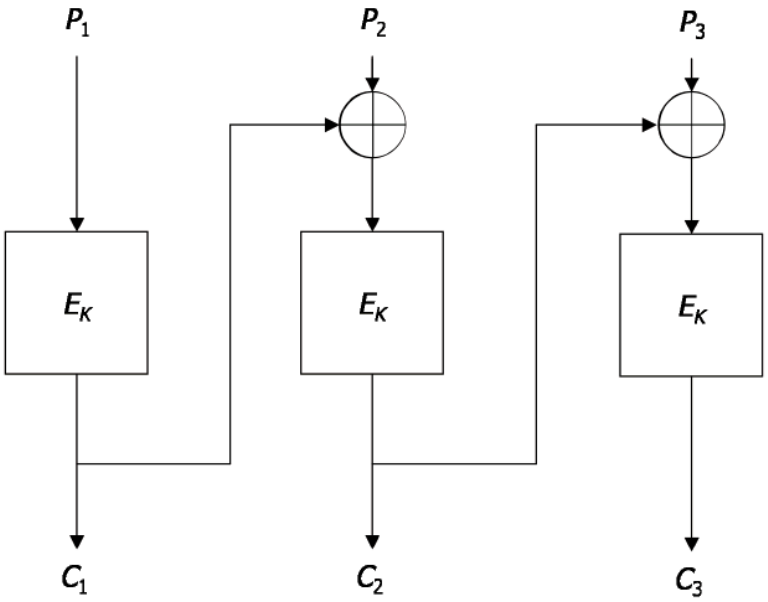
Inconvenients:

- El problema més seriós amb el qual es troba ECB és que l'atacant pot enviar missatges sense conèixer la clau, que és al que es diu un atac per block replay (o també playback attack).

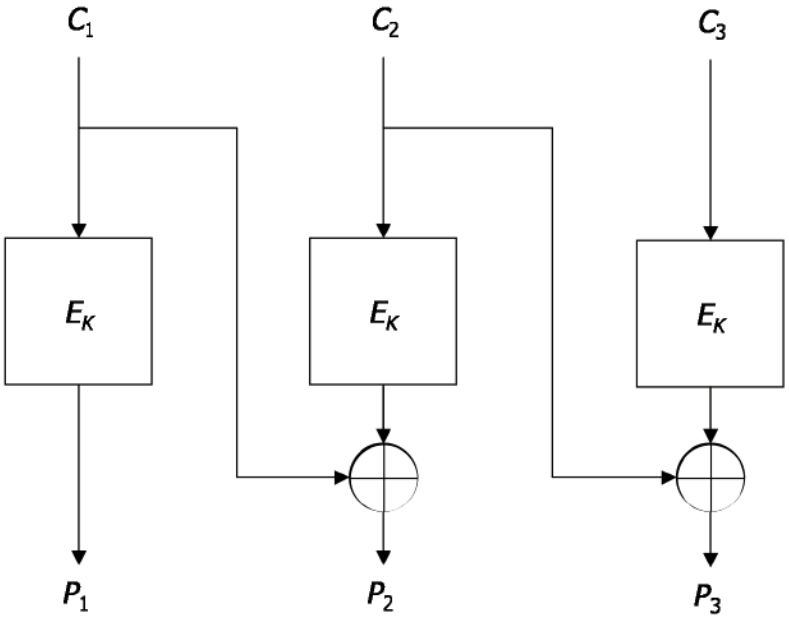
5.1.2. CBC. Cipher Block Chaining mode

Per solucionar el problema del mode anterior, aquest mode el que fa és que cada vegada que xifrem un bloc, **el text xifrat no només depèn de la clau i de l'algoritme utilitzat, sinó que també depèn dels anteriors blocs que hem xifrat**.

Per això fem un XOR al text pla del bloc actual amb l'anterior bloc xifrat.



Observi's que el primer bloc es xifra normalment, però al segon se li fa un XOR amb el primer bloc xifrat abans de xifrar-ho.
Ara, per desxifrar, el primer bloc es desxifra normalment, però al segon després de desxifrar-lo se li fa un XOR amb el primer bloc xifrat anterior.



L'ordre en què es realitzen els XOR amb el bloc xifrat anterior és important perquè en xifrar i desxifrar s'obtinguin els mateixos resultats.

Utilitzant el mode CBC aconseguim que **blocs de text pla idèntics donin lloc a diferents blocs xifrats**, amb l'única condició que els blocs anteriors siguin diferents. Però encara si xifrem dos missatges idèntics obtenim el mateix missatge xifrat.

També si dos missatges comencen igual donen lloc al mateix text xifrat fins que hi hagi la primera diferència en el missatge, amb el que les capçaleres dels missatges (que solen ser sempre iguals) poden donar al criptoanalista informació útil.
Per prevenir això anem a usar un vector d'inicialització (IV Initialization Vector), que és un text aleatori que es posa en el primer bloc. Ara, cada vegada que xifrem, idèntics textos plans donaran lloc a textos xifrats completament diferents.

Convé fer un parell d'apreciacions:

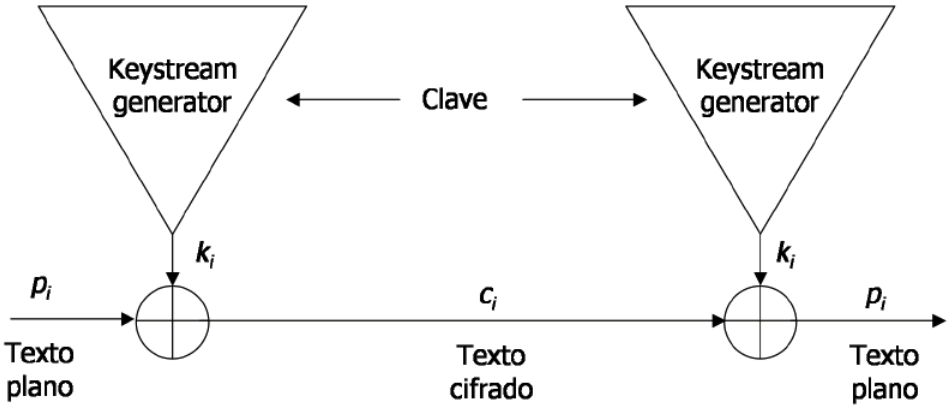
- La primera és que l'IV no ha de ser secret, ja que el coneixement de l'IV no li serveix a l'atacant, ni per desxifrar el missatge, ni per canviar-ho.
- La segona és que l'IV ha de ser únic en cada missatge, o l'atacant pot usar la tècnica de block replay per a missatges amb el mateix IV.

5.2. MODES EN ELS XIFRADORS DE FLUX

Per realitzar el xifrat anem a necessitar un **keystream generator** (també anomenat running-key generator), que no és més que un **generador de bits pseudoaleatoris** k1, k2 ,…kn, als quals se'ls fa després un XOR amb el text pla p1, p2,…pn per obtenir el text xifrat c1,c2,…cn. És a dir:

$$c_i = k_i \oplus p_i$$

Para desxifrar en el receptor tornem a usar el mateix keystream.



5.3. MODES MIXTOS

Els modes de xifrat en bloc tenen l'avantatge que s'implementen millor en programari, mentre que els modes de xifrat en flux s'implementen millor en maquinari (ja que processen bit a bit).

Per contra, els modes de xifrat en bloc tenen l'inconvenient que les dades es xifren en unitats de grandària de bloc (típicament 64 bits) amb el que, a diferència de els modes de xifrat en flux, no s'adapten bé a problemes en els quals cal transmetre unitats petites d'informació.

Els modes mixtos agafen el millor de tots dos mons, ja que **usen un algorisme de xifrat en bloc** (que són més fàcils d'implementar en programari), i **permeten transmetre petites unitats d'informació** (com els algoritmes de xifrat en flux).

Aquests algorismes estan pensats per xifrar byte a byte, però es podrien modificar fàcilment per xifrar en altres grandàries diferents a 8 bits.

Modes:

- CFB. Cipher FeedBack mode
- OFB. Output FeedBack mode

5.4. QUIN MODE UTILITZAR?

ECB és el més fàcil d'usar, i el més ràpid, però el més vulnerable. En conseqüència ECB no es recomana més que per xifrar petits missatges aleatoris com per exemple claus. No es recomana per xifrar fitxers (les capçaleres comunes es poden criptoanalitzar).

Per xifrar fitxers es recomana usar un de els altres modes: CBC, OFB.

Per a grans quantitats d'informació és més ràpid CBC, però si són missatges petits (tràfic interactiu) haurem d'usar OFB.

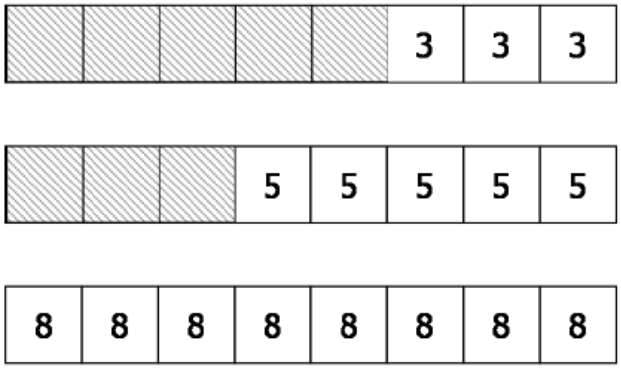
6. Padding

Els xifradors de bloc solen treballar amb grandàries de bloc de 64 o 128 bits, no obstant això les dades a xifrar no sempre són múltiples d'aquesta grandària. Per resoldre aquest problema hi ha dues solucions:

1. Exigir que l'usuari passi al xifrador unitats d'informació múltiple de la grandària de bloc.
2. Padding. És a dir, que el nostre algoritme criptogràfic empleni l'últim bloc amb dades de farcit abans de xifrar-ho, i després els tregui en el desxifrat.

L'avantatge d'usar padding és que és transparent a l'usuari.

De les tècniques de padding, la més utilitzada en els algorismes de clau secreta és PKCS#5 (Public Key Cryptography Standard 5). La tècnica consisteix a emplenar els bytes restants de l'últim bloc amb un nombre, que és el nombre de bytes que han quedat sense emplenar en l'últim bloc.

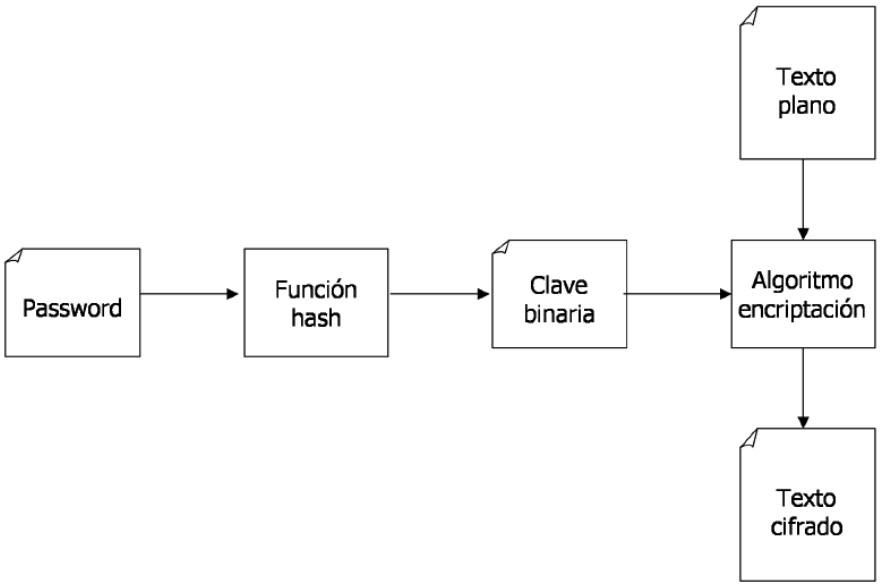


Si resultés que les dades a xifrar fossin múltiples de la grandària de bloc, es deixa un bloc més sencer farciment amb vuits, això es fa així perquè si no, no hi hauria forma que el receptor detectés el padding.

7. Claus binàries i PBE

Fins ara, els algoritmes de clau secreta que hem utilitzat usen una clau binària, que és una clau amb un contingut aleatori i una grandària predeterminada de bits. A causa de la seva aleatorietat, aquestes claus són difícils de recordar per les persones, amb el que és molt típic que l'usuari utilitzi una altra clau més senzilla de recordar, a la qual anomenem password.

El password estarà format per una o més paraules, i després de demanar-li-ho a l'usuari ho transformarem en una clau binària que és la que anem a passar a l'algoritme. En concret, anem a passar el password per una funció hash per obtenir una clau binària, que és la que acabarem donant a l'algoritme de xifrat. **S'anomena Password Based Encryption (PBE) a la tècniques criptogràfiques que obtenen la clau binària a partir d'un password**.



Un problema que tenen els passwords és que són molt més fàcils d'atacar per força bruta que les claus binàries.

A això s'uneix el fet que l'usuari tendeix a utilitzar paraules fàcils de l'idioma, amb el que es torna molt efectiu un atac per diccionari. Per dificultar aquest atac se solen usar tres estratègies:

1. **Passphrase**. El programa pot suggerir a l'usuari que en comptes de donar una paraula, d'una frase sencera, amb la finalitat d'augmentar el nombre de combinacions.
2. **Salt (Sal)**. El salt és un valor aleatori que es concatena al password abans de passar-ho per la funció hash que usem per obtenir la clau binària. Si no usem salt, el password sempre dóna lloc a la mateixa clau binària, amb el salt tenim 264 claus binàries diferents per al mateix password. D'aquesta forma dificultem els atacs per diccionari, obligant a l'atacant a calcular el hash de totes les claus amb tots els salt.
3. **Iteration count**. És una tècnica utilitzada amb la finalitat d'augmentar el temps necessari per calcular el hash de cada password. El iteration count és el nombre de vegades que cal fer hash al salt juntament amb el password per calcular la clau binària.

8. Les llibreries JCA i JCE

A causa de les restriccions d'exportació de programari criptogràfic que existien a EUA quan es van dissenyar les llibreries criptogràfiques de Java, JavaSoft va descompondre les seves API criptogràfiques en dues llibreries:

- La llibreria JCA (Java Cryptography Architecture)
- La llibreria JCE (Java Cryptography Extensions)

La llibreria JCA forma part del runtime de la màquina virtual de Java a partir de la versió 1.2, sota el paquet **java.security**. La llibreria JCA disposa d'elements de seguretat que no estan subjectes a restriccions d'exportació (p.ex. signatures digitals o funcions hash).

A la llibreria JCE s'inclouen classes de xifrat i desxifrat de missatges que són les que el govern dels EUA no deixava exportar. A causa que quan es va crear la llibreria JCE, només es podia obtenir a EUA i Canadà, altres fabricants d'altres països es van posar a fer implementacions de la llibreria JCE que distribuïen gratuïtament a la resta del món a través d'Internet.

- http://www.bouncycastle.org
- http://www.cryptix.org

A partir del JDK 1.4 van canviar les lleis, i Sun va poder exportar la llibreria JCE com a part de la seva màquina virtual. Així i tot els altres proveïdors segueixen sent útils, ja que disposen de més algorismes que la llibreria JCE de Sun.

La llibreria JCE es distribueix com una extensió estàndard. Les classes de la llibreria JCE estan sota el nom javax.crypto.*.

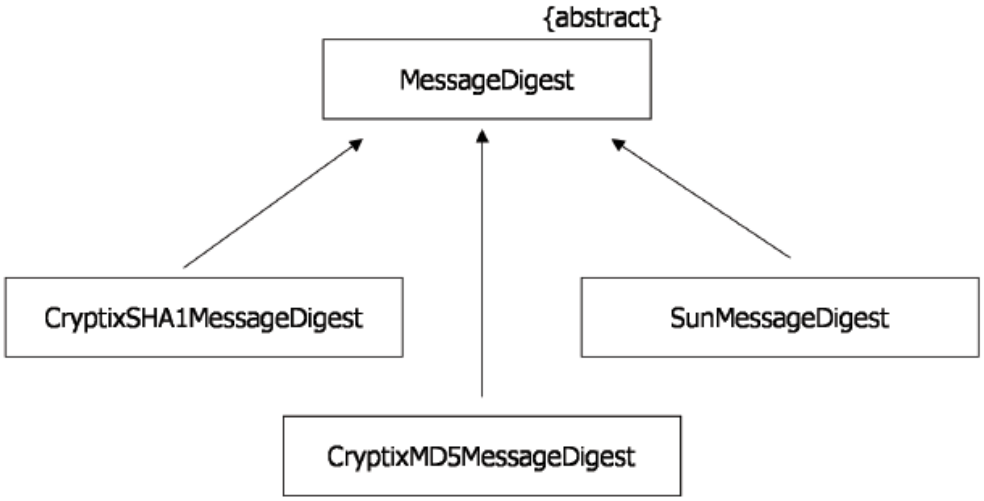
9. La llibreria JCA (Java Cryptography Architecture)

La llibreria JCA està formada per dos grups de classes:

1. Les **classes de la llibreria JCA**, que són classes que actuen com a interfícies en les quals es defineixen una sèrie d'operacions estàndard.
2. Els **proveïdors del seguretat**, que són terceres parts que implementen aquestes interfícies amb algorismes criptogràfics propis.

En Java anomenarem **engine o servei** a cadascuna de les operacions criptogràfiques que proporcionen les llibreries criptogràfiques de Java (p.ex. funcions hash, xifrat, signatures digitals ...).

Cada servei està representat per una **classe abstracta**. Per exemple, MessageDigest, Cipher o Signature. La implementació d'aquests serveis es realitza per classes derivades que implementen els proveïdors de seguretat.



La relació entre la classe abstracta i les classes del proveïdor no és 1 a 1, ja que per a un mateix proveïdor, un mateix servei pot estar implementat per diverses classes diferents.
A cada implementació se li anomena **algoritme**. Per exemple, Cryptix implementa MessageDigest amb els algorismes SHA-1 i MD5.

La raó per la qual es van implementar les llibreries criptogràfiques d'aquesta forma van ser dues:

1. Que un mateix servei es pugui implementar amb diversos algorismes.
2. Separar les interfícies del servei, de la implementació que fan els proveïdors (per evitar els problemes d'exportació).

En la classe abstracta sempre trobem el mètode estàtic **getInstance()** al qual podem demanar un objecte que implementi aquesta mateixa classe. Per a això hem d'indicar l'algorisme i el proveïdor. Per exemple, per MessageDigest faríem:

```
MessageDigest md =
   MessageDigest.getInstance("SHA1","SUN");
```

Des del principi, la màquina virtual de Sun sempre ha portat dos proveïdors per a la llibreria JCA: "SUN" i RSAJCA". Aquests proveïdors només implementen la llibreria JCA. Hi ha altres proveïdors que implementen la llibreria JCE, o bé altres algorismes per a la llibreria JCA.

9.1. PRIORITAT DELS PROVEÏDORS SE SEGURETAT

L'ordre de prioritat d'aquests proveïdors s'especifica en el fitxer java.security:

- En Mac OS X aquest fitxer es troba en:
  $JAVA_HOME/lib/security/java.security
- En la implementació de Java per Linux que proporciona Sun es troba en:
  $JAVA_HOME/jre/lib/security/java.security
- En Windows es troba en:
  %JAVA_HOME%\jre\lib\security\java.security

La imatge següent mostra un exemple del contingut d'aquest fitxer. El nombre indica l'ordre de preferència de la cerca d'un proveïdor que implementi l'algoritme demanat. Això es fa així amb la finalitat de que si ometem el proveïdor:

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

El mètode getInstance() triï el primer proveïdor de la llista que implementi aquest algoritme:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.apple.crypto.provider.Apple
security.provider.3=sun.security.rsa.SunRsaSign
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
```

9.2. ENGINE CLASSES

- **SecureRandom**: used to generate random or pseudo-random numbers.
- **MessageDigest**: used to calculate the message digest (hash) of specified data.
- **Signature**: initilialized with keys, these are used to sign data and verify digital signatures.
- **Cipher**: initialized with keys, these used for encrypting/decrypting data. There are various types of algorithms: symmetric bulk encryption (e.g. AES, DES, DESede, Blowfish, IDEA), stream encryption (e.g. RC4), asymmetric encryption (e.g. RSA), and password-based encryption (PBE).
- **Message Authentication Codes (MAC)**: like MessageDigests, these also generate hash values, but are first initialized with keys to protect the integrity of messages.

- **KeyFactory**: used to convert existing opaque cryptographic keys of type Key into key specifications (transparent representations of the underlying key material), and vice versa.
- **SecretKeyFactory**: used to convert existing opaque cryptographic keys of type SecretKey into key specifications (transparent representations of the underlying key material), and vice versa.SecretKeyFactorys are specialized KeyFactorys that create secret (symmetric) keys only.
- **KeyPairGenerator**: used to generate a new pair of public and private keys suitable for use with a specified algorithm.
- **KeyGenerator**: used to generate new secret keys for use with a specified algorithm.
- **KeyAgreement**: used by two or more parties to agree upon and establish a specific key to use for a particular cryptographic operation.

- **AlgorithmParameters**: used to store the parameters for a particular algorithm, including parameter encoding and decoding.
- **AlgorithmParameterGenerator** : used to generate a set of AlgorithmParameters suitable for a specified algorithm.

- **KeyStore**: used to create and manage a keystore. A keystore is a database of keys. Private keys in a keystore have a certificate chain associated with them, which authenticates the corresponding public key. A keystore also contains certificates from trusted entities.

- **CertificateFactory**: used to create public key certificates and Certificate Revocation Lists (CRLs).
- **CertPathBuilder**: used to build certificate chains (also known as certification paths).
- **CertPathValidator**: used to validate certificate chains.
- **CertStore**: used to retrieve Certificates and CRLs from a repository.

10. La llibreria JCE (Java Cryptography Extensions)

La llibreria JCE, igual que la llibreria JCA es pot dividir per dues parts:

1. Les classes de la llibreria JCE.
2. Els proveïdors de seguretat.

Fins a la versió Java 1.3 inclusivament, Sun no podia exportar les seves llibreries de JCE, raó per la qual van posar dues restriccions:

1. Les classes de la llibreria JCE s'exportaven a tothom però, perquè Sun pogués exportar la llibreria JCE fora dels EUA, va haver de ficar algunes restriccions de seguretat en aquesta, les quals feien impossible l'usar algorismes "forts" d'altres proveïdors.
2. Només s'exportava la llibreria JCE, però no els algorismes criptogràfics.

Per solucionar el primer problema els proveïdors van crear les seves pròpies implementacions de la llibreria JCE que no tenen aquestes restriccions, i a la qual van cridar JCE clean room. El segon problema se solucionava proporcionant les classes que implementaven els algorismes criptogràfics.

Amb l'arribada de Java 1.4 desapareixen totes les restriccions de seguretat i Java ve amb el seu propi proveïdor de seguretat per a la llibreria JCE anomenat "SunJCE", el qual disposa dels principals algoritmes criptogràfics. A més podem instal·lar-nos lliurement altres proveïdors de seguretat que portin més algorismes criptogràfics. Per a això simplement hem d'afegir les llibreries criptogràfiques del proveïdor al CLASSPATH així:

```
$ export CLASSPATH=$CLASSPATH:…/bcprov-jdk14-121.jar
```

O el seu equivalent en un altre sistema operatiu.

11. Proveïdors de seguretat

En Java, a més de les classes abstractes que representen els serveis, i les classes derivades de les abstractes que representen els algoritmes, hi ha altres dues classes importants:

- **Provider**: Cada instància d'aquesta classe representa a un proveïdor de seguretat que tenim instal·lat en el sistema.
- **Security**: Aquesta classe actua com a gestor de proveïdors. La classe és final i té un constructor privat amb el que no es pot instanciar, però té mètodes estàtics que gestionen els proveïdors instal·lats. També té un bloc estàtic que és el que carrega als proveïdors del fitxer java.security.

Podem obtenir els proveïdors instal·lats amb el mètode:

static Provider[] <Security> getProviders()

També podem afegir o eliminar proveïdors en temps d'execució amb els mètodes:

static void <Security> addProvider(Provider provider)

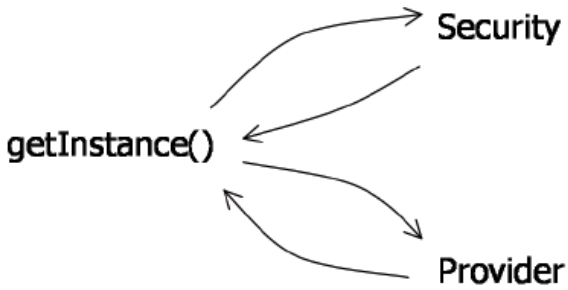
static void <Security> removeProvider(String name)

La qual cosa és molt útil per no haver de demanar a l'usuari del nostre programa que modifiqui el fitxer java.security.

Quan executem el mètode estàtic getInstance(), indirectament estem demanant a Security un algoritme:

```
MessageDigest md = MessageDigest.getInstance("SHA1");
```

Aquesta sentència demana a Security un proveïdor que implementi aquest servei, i després pregunta al proveïdor quina classe instanciar perquè li doni aquest servei.



La forma que té getInstance() de demanar un servei a Security és passar-li un String amb el nom del servei i l'algoritme de la forma: **"MessageDigest.SHA1"**

I el mètode que utilitza per fer aquesta petició és:

static Provider[] <Security> getProvider(String filter)

En filter passem el text anterior i ens retorna un array amb tots els proveïdors que implementen aquest servei ordenat per prioritat.

Després, getInstance() va al primer dels proveïdors i li demana una classe que implementi aquest servei, una vegada que té el nom de la classe, la instància per obtenir aquest servei.

11.1. LA CLASSE PROVIDER

Provider és una classe abstracta de la qual deriven les classes dels diferents proveïdors.

En el fitxer java.security podem veure quin és la classe de cada proveïdor.

Per exemple, security.provider.1=sun.security.provider.Sun, significa que security.provider.Sun és una derivada de Provider amb informació sobre aquest proveïdor.

La classe Provider disposa de mètodes amb informació sobre el proveïdor com:

- String <Provider> getName() → que retorna el nom del proveïdor
- double <Provider> getVersion() → que retorna la versió d'implementació..
- String <Provider> getInfo() → amb una altra informació que vulgui publicar el proveïdor.

Nota: Recordeu que s'han d'afegir el fitxer amb els algorismes criptogràfics al CLASSPATH.

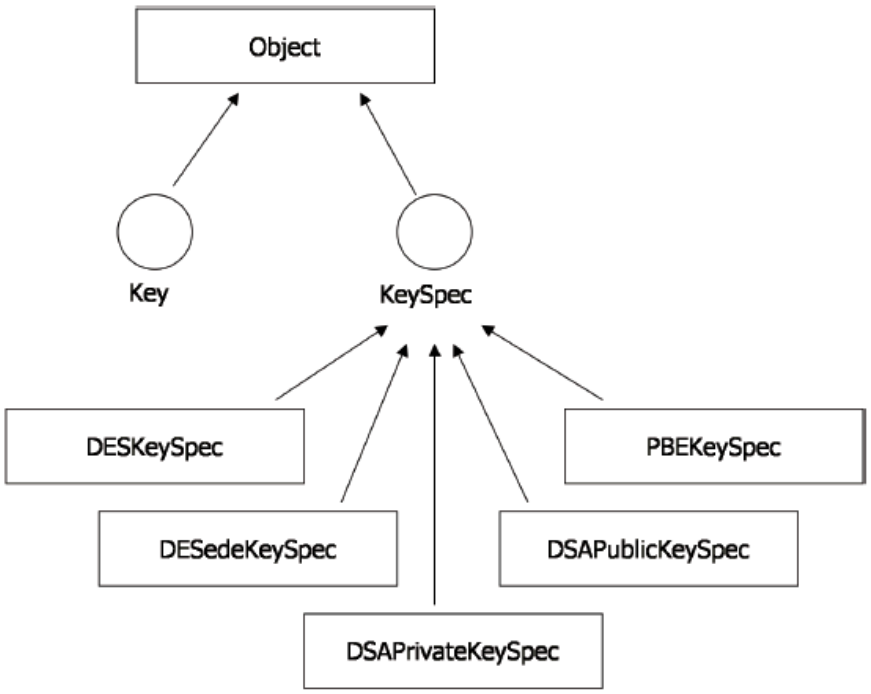
Exercici 1

12. Tipus opacs i especificacions

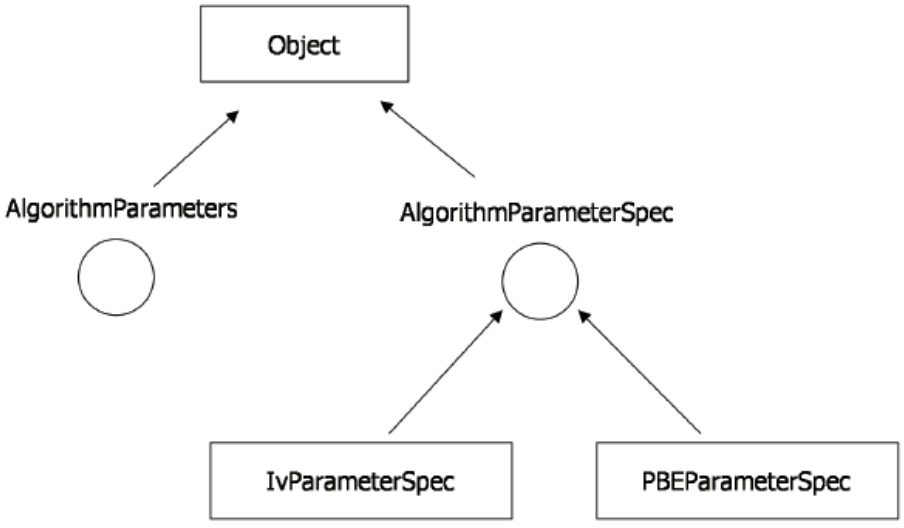
Molts dels conceptes criptogràfics de Java estan tancats sota **tipus opacs**, que són objectes dels quals només es coneix una interfície d'operacions, però no es coneixen els valors que emmagatzemen.

No obstant això, de vegades és necessari conèixer els valors que oculten aquests tipus opacs, per a això existeixen les **especificacions** (també conegudes com a **tipus transparents**).

Per exemple, en el cas de les claus, aquestes s'encapsulen en la interfície Key, no obstant això, de vegades ens pot interessar saber què tipus de clau és, i quines parts té. Per a això, tenim la interfície KeySpec, que té tantes classes derivades com a tipus de claus defineix Java.



Un altre exemple d'especificacions són els paràmetres que necessiten els algorismes per realitzar el seu treball. La imatge següent mostra la jerarquia de classes en el cas de la interfície AlgorithmParameterSpec, i les classes que la implementen. D'altra banda AlgorithmParameters representa el tipus opac.



Totes aquestes especificacions s'emmagatzemen en els paquets java.security.spec.* i java.crypto.spec.*

13. Ús de claus des de Java

13.1. LA INTERFÍCIE KEY

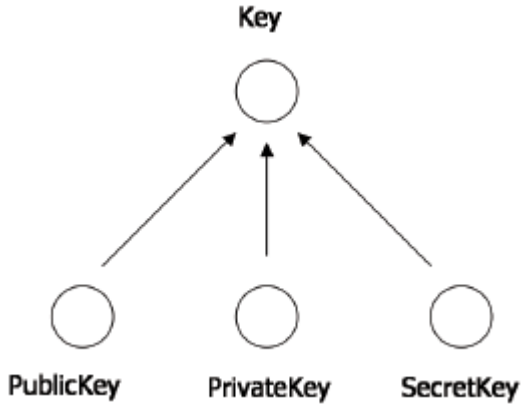
La interfície java.security.Key és la base de tots els tipus de claus de Java. Implementa Serializable, amb el que totes les claus en Java són serializables, i actua com un tipus opac, ja que no dóna informació sobre els seus atributs.

La interfície Key únicament té tres mètodes. En conseqüència tot tipus de clau ha d'implementar aquests mètodes:

- **String <Key> getAlgorithm()** → Retorna el nom de l'algorisme pel qual serveix la clau (p.ex. "DES", "Blowfish", "DSA",…).
- **byte[] <Key> getEncoded()** → Retorna una tira de bytes amb la clau binària.
- **String <Key> getFormat()** → Indica el format usat per a la clau retornada pel mètode anterior (p.ex. "X.509" o "PKCS#8"). Aquest format és necessari per poder interpretar la tira de bytes.

13.2. PUBLICKEY, PRIVATEKEY I SECRETKEY

Les interfícies PublicKey, PrivateKey i SecretKey representen els diferents tipus de claus. Observi's que aquestes interfícies deriven de Key i no de KeySpec, ja que són tipus opacs i no especificacions.



Aquestes interfícies no afegeixen més mètodes, sinó que només es creen per classificar els diferents tipus de claus.

- Les interfícies PublicKey i PrivateKey s'utilitzen en criptografia asimètrica i vénen en el paquet java.security.*
- La interfície SecretKey s'utilitza en criptografia simètrica, i està en el paquet javax.crypto.*

13.3. KEYPAIRGENERATOR I KEYGENERATOR

Per generar objectes que representin claus s'usa una d'aquestes classes:

- **java.security.KeyPairGenerator** s'usa per generar claus públiques i privades. Aquestes sempre es generen en parella.
- **javax.crypto.KeyGenerator** s'usa per crear objectes de tipus SecretKey.

KeyGenerator és una classe abstracta que representa un servei, i els algorismes del qual estan implementats pels proveïdors de seguretat.

1. Com a servei que és, per obtenir una instància usem el mètode estàtic:

   static KeyGenerator <KeyGenerator> getInstance(String algorithm)

   En el paràmetre algorithm indiquem l'algoritme criptogràfic pel qual volem generar la clau (p.ex. "DES").
2. Després hem d'inicialitzar el generador de claus, per a això tenim el mètode init(). Aquest mètode està sobrecarregat per permetre'ns introduir paràmetres comuns a tots els tipus de claus com són:

   void <KeyGenerator> init(int keySize)

   void <KeyGenerator> init(SecureRandom sr)

   Que ens permeten indicar la grandària de clau que volem, o un generador de nombres aleatoris usat per generar la clau. Si no indiquem un SecureRandom, l'objecte KeyGenerator instància un per a si només, encara que és millor un de propi si anem a crear diverses instàncies de KeyGenerator, ja que instanciar un SecureRandom consumeix bastant temps.
3. Finalment cridem al mètode:

   SecretKey <KeyGenerator> generateKey()

   El qual ens permet generar tantes claus com vulguem, una vegada inicialitzat l'objecte. Observi's que KeyGenerator ens permet generar **només claus binàries aleatòries**. Per crear claus PBE a partir d'un password hi ha altres mètodes.

13.4. KEYFACTORY I SECRETKEYFACTORY

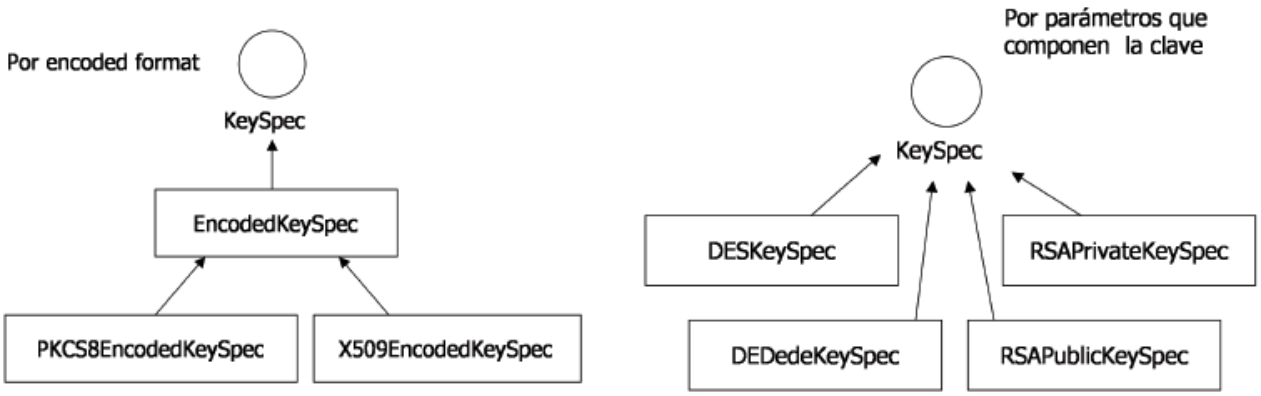
Els key factories s'usen per convertir objectes de tipus Key (tipus opacs) en especificacions d'aquests objectes (tipus transparents), i viceversa.

La seva principal aplicació és permetre exportar i importar claus.

Existeixen dues formes de representar externament una clau:

1. Pel seu encoded format
2. Amb els paràmetres que componen la clau

Ambdues formes estan implementades per classes derivades de KeySpec.



Si tenim el encoded format retornat per getEncoded(), i el format retornat per getFormat() (que pot ser "PKCS#8" o "X.509"), podem crear un objecte derivat de EncodedKeySpec usant un d'aquests constructors:

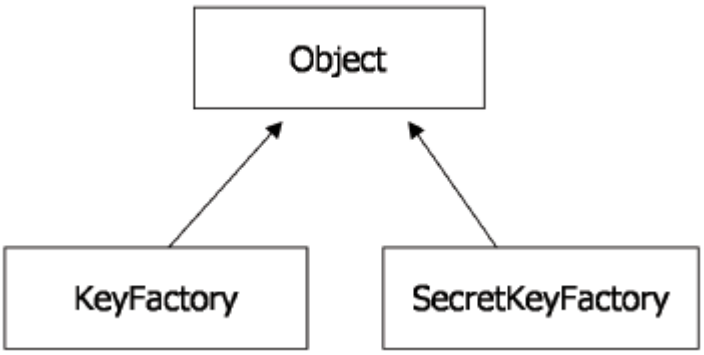
PKCS8 EncodedKeySpec(byte[] encodedKey)

X509 EncodedKeySpec(byte[] encodedKey)

Si del que disposem és dels paràmetres que componen una clau, també hi ha classes derivades de KeySpec que en el seu constructor reben aquests paràmetres.

Existeixen dues classes factory que actuen com key factories, és a dir, que serveixen per generar claus:

- **KeyFactory** és la clau que serveix per importar i exportar claus asimètriques.
- **SecretKeyFactory** és la classe que serveix per importar i exportar claus secretes.



SecretKeyFactory és un servei, i com a tal, per obtenir un algoritme d'un proveïdor usem:

static SecretKeyFactory <SecretKeyFactory> getInstance(String algorithm)

Cada proveïdor ha de documentar les especificacions que suporta la seva clau. Per exemple "SunJCE" suporta les especificacions DESKeySpec i DESedeKeySpec per a l'algorisme "DES".

Això significa que una clau "DES" (tipus opac) de "SunJCE" es pot transformar en els tipus transparents DESKeySpec i DESedeKeySpec, o viceversa.

13.4.1. EXPORTAR UNA CLAU

Per exportar una clau (obtenir el seu tipus transparent) usarem el mètode:

KeySpec <SecretKeyFactory> getKeySpec(SecretKey key, Class classSpec)

On el paràmetre classSpec hauria de ser, o bé DESKeySpec.class, o bé DESedeKeySpec.class.

Per exemple, per obtenir un DESKeySpec d'un SecretKey aleatori obtingut de tipus "DES" usant "SunJCE" fem:

```
SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
DESKeySpec dks = (DESKeySpec) skf.getKeySpec(clau, DESKEYSPEC.class);
```

On clau és la SecretKey de la que es vol obtenir el seu tipus transparent.

13.4.2. IMPORTAR UNA CLAU

Si el que volem és importar una clau (obtenir el tipus opac d'un tipus transparent), vam crear un objecte d'algun tipus derivat de KeySpec, i l'hi passem a:

SecretKey <SecretKeyFactory> generateSecret(KeySpec ks)

Per exemple, si tenim un array de 8 bytes amb una clau DONIS podem fer:

```
byte[] buffer; // Array de 64 bits carregat amb la clau
DESKeySpec dks = new DESKeySpec(buffer);
SecretKey clave = skf.generateSecret(dks);
```

Exercici 2

14. Ús d'un xifrador

14.1. LA CLASSE CIPHER

La classe abstracta javax.crypto.Cipher és un servei que serveix per xifrar i desxifrar informació. Com tots els serveis, per obtenir una instància usem:

static Cipher <Cipher> getInstance(String algorithm)

static Cipher <Cipher> getInstance(String algorithm, String provider)

En aquest cas en algorithm hem d'especificar un dels algoritmes vàlids per al proveïdor.

Algoritmes vàlids pel proveïdor "SunJCE":

Algoritmo	Descripción
DES	Data Encryption Standard.
DESede	Triple DES.
PBEWithMD5AndDES	Password Based Encryption definido en PKCS #5. PKCS #5 fue desarrollado por RSA Data Security Inc. Para encriptar claves privadas, aunque se puede usar para encriptar cualquier cosa.
Blowfish	Algoritmo open source desarrollado por Bruce Schneier

Modes de "SunJCE":

Modo	Descripción
ECB	Electronic Code Book. Sólo puede operar en bloques completos por lo que se usa con padding. No requiere vector de inicialización.
CBC	Cipher Block Chaining Mode. Sólo puede operar en bloques completos por lo que se usa con padding. Requiere vector de inicialización.
CFB	Cipher FeedBack Mode. Puede operar en bloques de 8 bits, en cuyo caso no requiere padding. Sí lo requiere si ponemos un bloque mayor. Requiere vector de inicialización.
OFB	Output FeedBack Mode. Puede operar en bloques de 8 bits, en cuyo caso no requiere padding. Si lo requiere si ponemos un bloque mayor. Requiere vector de inicialización.
PCBC	Propagating Cipher Block Chaining Mode. Es popular en sistemas como Kerberos versión 4. Kerbedos 5 paso a usar CBC. Requiere padding y vector de inicialización.

Padding de “SunJCE”:

Padding	Descripción
PKCS5Padding	Asegura que los datos resultantes son múltiplos de 8 bytes
NoPadding	Sin padding.

Per exemple, per obtenir un algorisme para DES fem:

Cipher cifrador = Cipher.getInstance("DES");

Encara que també en algorithm podem especificar el mode i el padding usant els valors de les taules anteriors de la següent manera:

Cipher cifrador = Cipher.getInstance("Blowfish/CBC/PKCS5Padding");

Si no s'indica mode i padding cada proveïdor té uns valors per defecte.

Cipher permet xifrar tant en mode en bloc (p.ex. ECB o CBC), com en mode mixt (p.ex. CFB o OFB), i mode en flux.
Si volem usar un dels modes mixtos (que ens permetien xifrar en blocs més petits que els modes en bloc) hem d'indicar el nombre de bits a processar alhora, afegint aquest nombre a la mode. Per exemple, si volem processar byte a byte usaríem: "DES/CFB8/NoPadding"
Observi's que si anem a processar byte a byte no fa falta padding, i que posant la grandària de bloc a 1 podem aconseguir un xifrador de flux.

Una vegada tinguem l'objecte Cipher, haurem d'inicialitzar-ho amb un dels molts mètodes init() sobrecarregats que disposa la classe:

void <Cipher> init(int op, Key key)

void <Cipher> init(int op, Key key, AlgorithmParameterSpec aps)

- El paràmetre op indica si va a usar-se el xifrador per xifrar o desxifrar, usant els valors:
  Cipher.ENCRYPT_MODE
  Cipher.DECRYPT_MODE
- El paràmetre key és un objecte de tipus SecretKey amb la clau a utilitzar.
- A més, l'algoritme pot requerir altres paràmetres, com per exemple un IV, en aquest cas l'hi passem en un tercer paràmetre en un objecte de tipus IvParameterSpec, que deriva de AlgorithmParameterSpec. En "SunJCE" tots els modes excepte ECB requereixen IV.

Cal tenir en compte que quan anem a xifrar, si intentem usar el xifrador sense estar ben inicialitzat, el xifrador llança una IllegalStateException per avisar-nos del problema.

Per xifrar les dades, hi ha dues alternatives:

- Xifrar totes les dades en una sola crida, en aquest cas usem:
  byte[] <Cipher> doFinal(byte[] input)

- Si tenim moltes dades a xifrar, o les rebem a poc a poc, podem anar-les processant segons els anem rebent cridant a:
  byte[] <Cipher> update(byte[] input)
  I l'últim array ho passem a:
  byte[] <Cipher> doFinal()
  byte[] <Cipher> doFinal(byte[] input)

També convé tenir en compte que quan cridem a init() es torna a reiniciar el xifrador.

Existeix una classe derivada de Cipher anomenada javax.crypto.NullCipher que no realitza cap xifrat, i que a diferència de Cipher es pot crear amb el constructor per defecte:

Cipher cifrador = new NullCipher();

Aquesta classe es pot usar per provar la lògica del nostre programa sense realitzar cap xifrat o desxifrat.

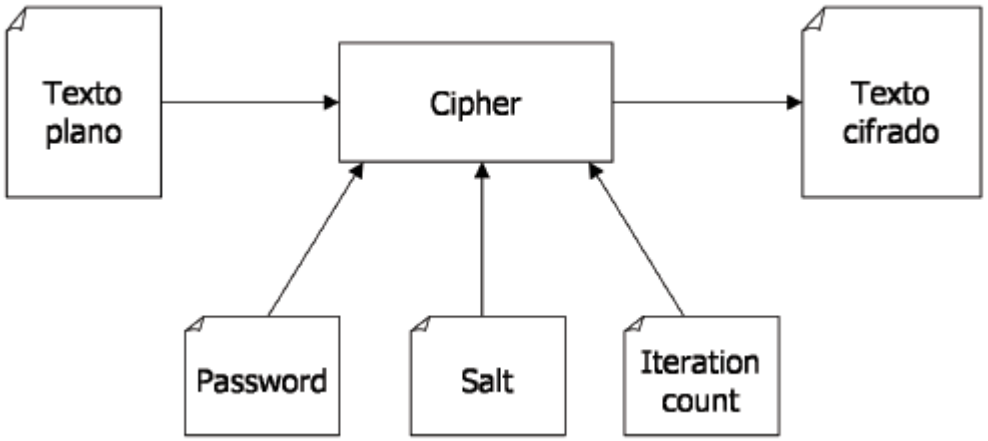
Exercici 3

15. PBE (Password Based Encryption)

És molt comú que l'usuari en comptes d'una clau binària, la qual cosa tingui sigui un password, i sigui el programa el que generi la clau binària a partir del password.

Si volem fer això en Java, al xifrador hem de passar-li tres paràmetres:

1. Password
2. Salt
3. Iteration count



Per a què el xifrador funcioni en aquest modes, hem de passar a:

static Cipher <Cipher> getInstance(String algorithm)

en el paràmetre algorithm un algoritme de PBE.

Proveedor	Algoritmo
Sun	"PBEWithMD5AndDES"
Sun	"PBEWithMD5AndTripleDES"
Bouncy Castle	"PBEWITHMD5ANDDES"
Bouncy Castle	"PBEWITHSHA1ANDDES"
Bouncy Castle	"PBEWITHSHAAND40BITRC2-CBC"
Bouncy Castle	"PBEWITHSHAAND128BITRC2-CBC"
Bouncy Castle	"PBEWITHSHAAND40BITRC4"
Bouncy Castle	"PBEWITHSHAAND128BITRC4"
Bouncy Castle	"PBEWITHSHAAND2-KEYTRIPLEDES-CBC"
Bouncy Castle	"PBEWITHSHAAND3-KEYTRIPLEDES-CBC"
Bouncy Castle	"PBEWITHSHAANDTWOFISH-CBC"
Bouncy Castle	"PBEWITHSHAANDIDEA-CBC"

Com a exemple suposarem que anem a usar "PBEWithMD5AndDES". Aquest algoritme està descrit en l'especificació PKCS #5 de RSA Data Security Inc. Inicialment va ser concebut per xifrar claus privades, però pot usar-se per xifrar qualsevol dada.

Anem a veure ara com passar els tres paràmetres que necessita el xifrador.

1. Per passar el **password** al xifrador, hem de convertir primer aquest a una clau binària, per a això hem de començar creant un objecte de la classe PBEKeySpec, la qual té el constructor:

   PBEKeySpec(char[] password)

   Observi's que rep un array de caràcters, i no un String.

   Ara podem crear una clau binària (un objecte de tipus SecretKey) a partir del PBEKeySpec seguint els següents passos:

   1. Crear un objecte de tipus SecretKeyFactory usant el seu mètode estàtic getInstance() al que passem "PBEWithMD5AndDES" com a argument.
   2. Crear un objecte de tipus SecretKey usant:

      SecretKey <SecretKeyFactory> generateSecret(KeySpec ks)

      A aquest mètode li passem el PBEKeySpec i ens retorna la clau binària en un objecte de tipus SecretKey.
2. Per passar el **salt** i **iteration count**, vam crear un objecte de tipus PBEParameterSpec que té el següent constructor:

   PBEParameterSpec(byte[] salt, int iterationCount)

   Per crear el salt, es pot fer de la següent forma:

   ```
   byte[] salt = new byte[8];
   new Random().nextBytes(salt);
   ```

Una vegada que tinguem els objectes SecretKey i PBEParameterSpec, els usem per inicialitzar el xifrador amb:

void <Cipher> init(int op, Key key, AlgorithmParameterSpec aps)

PBEParameterSpec deriva de AlgorithmParameterSpec, i per tant es pot passar a aquest mètode.

Convé ressaltar que el xifrador, a més del password, ha rebut dos paràmetres per al xifrat (salt, i iteration count). Aquests mateixos paràmetres els ha de tenir el receptor per poder desxifrar.

Si el que es vol és enviar els paràmetres al receptor, hem de treure el encoded de l'objecte AlgorithmParameters usant el mètode:

byte[] <AlgorithmParameters> getEncoded()

I en recepció es torna a reconstruir l'objecte AlgorithmParameters a partir del encoded de la següent forma:

1. Instanciem un objecte de tipus AlgorithmParameters:

   AlgorithmParameters ap = AlgorithmParamter.getInstance("PBEWithMD5AndDES");
2. Fiquem el encoded en l'objecte:

   ap.init(encoded);

Finalment passaríem l'objecte ap al xifrador del receptor usant el init() que existeix per a tal propòsit:

void <Cipher> init(int op,Key key,AlgorithmParameter ap)

Pràctica