
RA5. Tècniques de programació segura: fonaments i pràctica en Java

Índex

1. Per què cal programar de manera segura	1
2. Bones pràctiques de programació segura	2
2.1 Desconfiar per defecte de qualsevol entrada	2
2.2 Gestionar bé el cicle de vida del codi i dels secrets	3
2.3 Errors habituals que cal evitar	4
3. Criptografia: conceptes i implementació en Java	4
3.1 Fonaments	4
3.2 Integritat: funcions resum	4
3.3 Emmagatzematge segur de contrasenyes	5
3.4 Xifratge simètric	6
3.5 Xifratge asimètric i esquema mixt	8
3.6 Autenticació i no repudi: la firma digital	9
3.7 Protocols criptogràfics: combinant les primitives	10
4. Certificats digitals i cadena de confiança	10
5. Comunicacions segures amb Java	11
6. Control d'accés	13
6.1 El model natiu de la JVM	13
6.2 Control d'accés basat en rols (RBAC) a nivell d'aplicació	14
7. Depuració, manteniment i documentació	16
Resum	17

Cicle formatiu: Desenvolupament d'aplicacions multiplataforma (DAM)

Mòdul: 0490. Programació de serveis i processos

1. Per què cal programar de manera segura

La programació segura consisteix a escriure codi que no sigui vulnerable a atacs, que no exposi informació confidencial i que falli de manera controlada quan alguna cosa va malament. Els fabricants de programari publiquen constantment pedaços de seguretat per corregir vulnerabilitats, un fenomen especialment freqüent en aplicacions connectades a Internet, i la immensa majoria d'aquests problemes tenen el mateix origen: **no validar les dades d'entrada i confiar cegament** en l'usuari, en fitxers externs o en altres sistemes.

Uns quants principis resumeixen l'actitud correcta davant d'aquest risc:

Principi	Descripció
Validació d'entrada	No confiar mai en dades que vinguin de fora (formularis, APIs, fitxers, arguments, línia d'ordres)
Principi del mínim privilegi	Cada component només ha de tenir els permisos estrictament necessaris
Defensa en profunditat	Diverses capes de seguretat, no una única barrera
Fail-safe defaults	Si alguna cosa falla, ha de fallar cap al costat segur (denegar per defecte)
No exposar informació confidencial	Missatges d'error genèrics, sense stack traces ni rutes internes
Gestió segura de secrets	Contrasenyes i claus mai en el codi font ni en el control de versions

El [Top 10 d'OWASP](#) és la referència estàndard de vulnerabilitats web més comunes (injecció, trencament d'autenticació, exposició de dades confidencials, etc.) i val la pena revisar-lo abans de dissenyar qualsevol aplicació.

2. Bones pràctiques de programació segura

2.1 Desconfiar per defecte de qualsevol entrada

Cap dada que entri al programa des de fora és de fiar per si mateixa, tant se val si arriba per un formulari, un fitxer de configuració, un paràmetre de línia de comandaments, una cookie, una variable d'entorn o un camp ocult d'un HTML: totes són manipulables per algú amb intencions malicioses i s'han de tractar amb la mateixa desconfiança. Això es tradueix en unes quantes pràctiques concretes: netejar la cadena rebuda (rebutjar caràcters no previstos, evitar desbordaments de buffer), comprovar sempre límits i índexs abans d'accedir-hi, establir valors inicials vàlids per a qualsevol variable i construir amb cura les referències a noms de fitxer i directori.

N'és un exemple clàssic la injecció SQL:

```
// VULNERABLE: concatenació directa de l'entrada de l'usuari
String query = "SELECT * FROM usuaris WHERE nom = '" + nomUsuari + "'";
Statement st = connexio.createStatement();
ResultSet rs = st.executeQuery(query);
// Si nomUsuari = "' OR '1'='1", l'atacant obté totes les files
```

```
// SEGUR: consulta parametritzada (PreparedStatement)
String query = "SELECT * FROM usuaris WHERE nom = ?";
PreparedStatement ps = connexio.prepareStatement(query);
ps.setString(1, nomUsuari);
ResultSet rs = ps.executeQuery();
```

La mateixa lògica de “no confiar en l'entrada” es tradueix en codi amb validació per *allowlist* (definir què es permet, no què es prohibeix) i amb una gestió d'excepcions que no filtri detalls interns:

```
public class ValidadorEntrada {

    private static final Pattern USUARI_VALID =
        ↪ Pattern.compile("[a-zA-Z0-9_]{3,20}$");

    public static boolean esUsuariValid(String usuari) {
        return usuari != null &&
            ↪ USUARI_VALID.matcher(usuari).matches();
    }

    public static void processarLogin(String usuari, char[]
        ↪ contrasenya) {
        if (!esUsuariValid(usuari)) {
            throw new IllegalArgumentException("Format d'usuari no
                ↪ vàlid");
        }
        try {
            autenticar(usuari, contrasenya);
        } catch (AutenticacioException e) {
```

```

        // NO fer: System.out.println("Error: " + e.getMessage());
        ↪ (pot filtrar detalls interns)
        LOGGER.warning("Intent de login fallit per l'usuari: " +
        ↪ usuari);
        throw new RuntimeException("Usuari o contrasenya
        ↪ incorrectes");
    } finally {
        Arrays.fill(contrasenya, '0'); // esborrar la contrasenya
        ↪ de memòria
    }
}
}

```

2.2 Gestionar bé el cicle de vida del codi i dels secrets

Un programa segur no s'escriu una sola vegada i s'oblida: cal reaprofitar sempre que sigui possible codi ja provat i revisat en lloc de reinventar-lo, sotmetre les parts crítiques a una revisió per parells o, en projectes més exigents, a una validació independent línia per línia, i mantenir viva una llista de control pròpia (contrasenya obligatòria per accedir-hi, inicis de sessió únics, control d'accés basat en rols, xifratge de les dades transferides) que es vagi actualitzant amb cada nou tipus d'error detectat. El manteniment posterior compta igual: seguir unes normes de codificació compartides, retirar el codi que ja no s'utilitza i analitzar a fons qualsevol canvi abans de posar-lo en producció.

Dins d'aquest mateix cicle de vida hi entra la gestió dels secrets: mai s'ha de guardar una contrasenya, clau API o clau criptogràfica directament al codi font. Les alternatives habituals són les variables d'entorn, els fitxers de configuració externs (fora del dipòsit, amb `.gitignore`), el Java KeyStore per a claus i certificats, i gestors de secrets dedicats (HashiCorp Vault, AWS Secrets Manager, variables CI/CD xifrades):

```

String password = System.getenv("DB_PASSWORD");
if (password == null) {
    throw new IllegalStateException("Variable d'entorn DB_PASSWORD no
    ↪ definida");
}

```

2.3 Errors habituals que cal evitar

Alguns hàbits, per còmodes que semblin, comprometen la seguretat d'una aplicació de manera recurrent:

- Pel que fa a **contrasenyes i dades confidencials**: mostrar-les en pantalla o en un log, enviar-les per correu electrònic, guardar-les sense xifrar en un fitxer o base de dades, o transmetre-les en text pla per la xarxa.
- Pel que fa a **confiança mal situada**: assumir que cap usuari actuarà de mala fe, autenticar-se contra un origen que no és de confiança, invocar programes, serveis de tercers o un simple shell des del mateix codi per a operacions crítiques, o basar una decisió d'accés en variables d'entorn o paràmetres de línia de comandaments que qualsevol pot alterar en temps d'execució.
- Pel que fa a **hàbits de codificació**: donar per fet que una crida al sistema (obrir un fitxer, llegir-lo, consultar una variable d'entorn) sempre té èxit sense comprovar-ne el valor de retorn, referir-se diverses vegades a un mateix fitxer pel seu nom en lloc de reutilitzar l'identificador ja obert, o construir rutes de fitxer relatives en lloc de completes.

3. Criptografia: conceptes i implementació en Java

3.1 Fonaments

La criptografia amaga el significat d'un missatge mitjançant xifratge: un **text llegible** es transforma amb un **algorisme criptogràfic** i una **clau** en un **text xifrat**, i el **desxifratge** en permet recuperar l'original. Dona resposta a quatre necessitats de seguretat:

1. **Confidencialitat** --- que ningú no autoritzat pugui llegir la informació.
2. **Integritat** --- que la informació no s'hagi modificat.
3. **Autenticació** --- que se sàpiga qui és l'origen de la informació.
4. **No repudi** --- que l'emissor no pugui negar haver enviat un missatge.

Existeixen tres famílies d'algorismes: les **funcions d'una sola via (hash)**, que donat un missatge x calculen fàcilment $f(x)$ però fan pràcticament impossible recuperar x ; els **algorismes de clau secreta o simètrica**, on emissor i receptor comparteixen una única clau per xifrar i desxifrar; i els **algorismes de clau pública o asimètrica**, on cada participant té una parella clau pública/clau privada, i només el propietari de la privada pot desxifrar el que s'ha xifrat amb la seva pública.

3.2 Integritat: funcions resum

La classe `MessageDigest` implementa aquestes funcions mitjançant `getInstance(String algoritmo)`, i disposa de mètodes com `update()`, `digest()`, `reset()` i el mètode estàtic `isEqual()` per comparar dos resums de manera segura. Tot i que MD5 i SHA-1 van ser durant anys els algorismes de referència, avui es consideren **trencats** i no s'han d'utilitzar amb finalitats de seguretat (només serveixen per a comprovacions d'integritat, com per exemple, sumes de verificació de descàrregues); cal fer servir sempre SHA-256 o superior:

```

public class Empremta {
    public static String sha256(String text) throws Exception {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] resum = md.digest(text.getBytes("UTF-8"));
        StringBuilder sb = new StringBuilder();
        for (byte b : resum) sb.append(String.format("%02x", b));
        return sb.toString();
    }
}

```

Cal tenir present que el resum d'un missatge, per si sol, no ofereix un alt nivell de seguretat: algú podria alterar simultàniament el text i el seu resum. És per això que un ús habitual és guardar un contingut juntament amb el seu resum, de manera que en llegir-lo posteriorment es pugui recalcular i comparar per detectar-ne la manipulació.

3.3 Emmagatzematge segur de contrasenyes

Les contrasenyes **mai** es guarden en clar ni amb un simple hash SHA-256, ja que aquest és vulnerable a taules *rainbow* i a la força bruta amb GPU. Cal un algorisme dissenyat expressament per ser lent, com PBKDF2, BCrypt o Argon2, combinat sempre amb una sal aleatòria:

```

public class GestorContrasenyes {

    public static byte[] generarSal() {
        byte[] sal = new byte[16];
        new SecureRandom().nextBytes(sal);
        return sal;
    }

    public static byte[] derivarHash(char[] contrasenya, byte[] sal)
        ↪ throws Exception {
        PBEKeySpec spec = new PBEKeySpec(contrasenya, sal, 210_000,
            ↪ 256);
        SecretKeyFactory factory =
            ↪ SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
        return factory.generateSecret(spec).getEncoded();
    }
}

```

3.4 Xifratge simètric

L'algorisme de referència històric és el **DES**, amb claus de 56 bits i blocs de 64 bits (amb la variant **Triple DES**, de 128 bits), però avui l'estàndard és l'**AES**, amb mides de clau habituals de 128 o 256 bits. La criptografia simètrica xifra molt més ràpid que l'asimètrica i és habitual en sistemes basats en maquinari, però necessita una distribució de claus molt segura, i el nombre de claus creix ràpidament amb el nombre d'usuaris.

En Java, la classe `Cipher` (paquet `javax.crypto`) és el nucli de l'extensió JCE per xifrar i desxifrar: s'obté amb `getInstance(String algoritmo)`, on l'algorisme té la forma *algorisme/mode/farciment*, i s'inicialitza amb `init(int modo, Key clave)` en els modes `ENCRYPT_MODE`, `DECRYPT_MODE`, `WRAP_MODE` o `UNWRAP_MODE`.

Pel que fa als modes de bloc, el **mode ECB** xifra cada bloc per separat amb la mateixa clau, però té l'inconvenient que blocs de text pla idèntics generen blocs xifrats idèntics, cosa que en facilita l'anàlisi de patrons; el **mode CBC** ho corregeix aplicant un XOR entre cada bloc i el bloc xifrat anterior, amb l'ajuda d'un vector d'inicialització. Avui, però, el mode recomanat per a AES és el **GCM**, que a més de xifrar proporciona autenticació de les dades (detecta si algú les ha manipulat):

```
public class XifrataAES {

    private static final int TAMANY_TAG_GCM = 128; // bits
    private static final int TAMANY_IV = 12;        // bytes,
    ↪ recomanat per a GCM

    public static SecretKey generarClau() throws
    ↪ NoSuchAlgorithmException {
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(256);
        return kg.generateKey();
    }

    public static String xifrar(String textPla, SecretKey clau) throws
    ↪ Exception {
        byte[] iv = new byte[TAMANY_IV];
        new SecureRandom().nextBytes(iv);

        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        cipher.init(Cipher.ENCRYPT_MODE, clau, new
        ↪ GCMParameterSpec(TAMANY_TAG_GCM, iv));
        byte[] xifrat = cipher.doFinal(textPla.getBytes("UTF-8"));

        byte[] resultat = new byte[iv.length + xifrat.length];
        System.arraycopy(iv, 0, resultat, 0, iv.length);
        System.arraycopy(xifrat, 0, resultat, iv.length,
        ↪ xifrat.length);
        return Base64.getEncoder().encodeToString(resultat);
    }

    public static String desxifrar(String textXifratBase64, SecretKey
    ↪ clau) throws Exception {
        byte[] dades = Base64.getDecoder().decode(textXifratBase64);
```

```

byte[] iv = Arrays.copyOfRange(dades, 0, TAMANY_IV);
byte[] xifrat = Arrays.copyOfRange(dades, TAMANY_IV,
    ↪ dades.length);

Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
cipher.init(Cipher.DECRYPT_MODE, clau, new
    ↪ GCMParameterSpec(TAMANY_TAG_GCM, iv));
return new String(cipher.doFinal(xifrat), "UTF-8");
    }
}

```

Mai s'ha de reutilitzar el mateix IV amb la mateixa clau: és l'error més comú i trenca completament la seguretat d'AES-GCM. Cal generar sempre un IV aleatori nou per a cada operació.

Per generar i emmagatzemar claus simètriques amb KeyGenerator es pot recórrer a ObjectOutputStream/ObjectInputStream, però l'opció recomanada en un entorn de producció és guardar-les dins d'un **KeyStore** (PKCS12) protegit amb contrasenya, mai com a fitxers solts al disc:

```

public class LectorKeyStore {
    public static SecretKey carregarClau(String rutaKeystore, char[]
        ↪ passwordKeystore,
                                String alies, char[]
                                ↪ passwordClau) throws
                                ↪ Exception {
        KeyStore ks = KeyStore.getInstance("PKCS12");
        try (FileInputStream fis = new FileInputStream(rutaKeystore)) {
            ks.load(fis, passwordKeystore);
        }
        KeyStore.PasswordProtection proteccio = new
            ↪ KeyStore.PasswordProtection(passwordClau);
        KeyStore.SecretKeyEntry entrada = (KeyStore.SecretKeyEntry)
            ↪ ks.getEntry(alies, proteccio);
        return entrada.getSecretKey();
    }
}

```

Per xifrar fluxos sencers (per exemple, un document PDF) sense carregar tot el contingut en memòria, la biblioteca JCE ofereix les classes CipherOutputStream i CipherInputStream, que xifren o desxifren un fitxer de manera transparent gestionant internament les crides a `update()` i `doFinal()`.

3.5 Xifratge asimètric i esquema mixt

La criptografia asimètrica permet aconseguir autenticació i no repudi, i facilita l'administració de claus perquè no cal intercanviar-les de manera secreta, però és més lenta i, per a xarxes grans, requereix un sistema de certificació de l'autenticitat de les claus públiques. La classe `KeyPairGenerator` genera la parella de claus:

```
KeyPairGenerator generador = KeyPairGenerator.getInstance("RSA");
generador.initialize(2048); // mida mínima recomanada actualment
KeyPair parell = generador.generateKeyPair();

PublicKey clauPublica = parell.getPublic();
PrivateKey clauPrivada = parell.getPrivate();
```

I la classe `Cipher` també serveix per xifrar/desxifrar amb RSA, aplicant sempre un farciment segur com OAEP:

```
public class XifratRSA {
    public static byte[] xifrar(byte[] dades, PublicKey clauPublica)
        throws Exception {
        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(Cipher.ENCRYPT_MODE, clauPublica);
        return cipher.doFinal(dades);
    }

    public static byte[] desxifrar(byte[] dadesXifrades, PrivateKey
        clauPrivada) throws Exception {
        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(Cipher.DECRYPT_MODE, clauPrivada);
        return cipher.doFinal(dadesXifrades);
    }
}
```

RSA no s'utilitza per xifrar fitxers grans: és lent i limitat en mida per bloc. Per això, en la pràctica els sistemes reals (TLS n'és l'exemple més clar) combinen totes dues famílies mitjançant un **esquema mixt o clau de sessió**: es genera una clau simètrica per xifrar el missatge de text; aquesta clau simètrica s'encrypta al seu torn amb la clau pública del receptor (`Cipher.WRAP_MODE`, mètode `wrap()`); i s'envien conjuntament la clau xifrada i el missatge xifrat. El receptor desxifra primer la clau amb la seva clau privada (`unwrap()`, mode `UNWRAP_MODE`) i, amb aquesta clau ja recuperada, desxifra el missatge. Així, l'ús de xifratge asimètric, més costós, queda limitat a una petita porció de dades.

3.6 Autenticació i no repudi: la firma digital

Una firma digital assegura la identitat de l'emissor d'un missatge i la seva integritat. El procediment, basat generalment en RSA, consisteix a generar un resum del missatge (H1), xifrar aquest resum amb la clau privada de l'emissor per obtenir la firma, i enviar-la adjunta al missatge. El receptor calcula el seu mateix resum del missatge rebut (H2), desxifra la firma amb la clau pública de l'emissor per recuperar H1, i, si H1 i H2 coincideixen, el missatge és autèntic i no ha estat alterat. Tot aquest sistema descansa en un pilar fonamental: l'autenticitat de la clau pública de cada participant, garantida per les Autoritats de Certificació.

En Java, la classe `Signature` implementa aquest procés en tres fases ---inicialització (`initSign()/initVerify()`), actualització de dades (`update()`) i firma o verificació (`sign()/verify()`)---, i l'algorisme de firma inclou el resum utilitzat (per exemple, *SHA256withRSA*):

```
public class SignaturaDigital {

    public static byte[] signar(byte[] dades, PrivateKey clauPrivada)
        ↪ throws Exception {
        Signature signatura = Signature.getInstance("SHA256withRSA");
        signatura.initSign(clauPrivada);
        signatura.update(dades);
        return signatura.sign();
    }

    public static boolean verificar(byte[] dades, byte[] signat,
        ↪ PublicKey clauPublica) throws Exception {
        Signature signatura = Signature.getInstance("SHA256withRSA");
        signatura.initVerify(clauPublica);
        signatura.update(dades);
        return signatura.verify(signat);
    }
}
```

Per emmagatzemar les claus en disc cal codificar-les prèviament: la privada amb `PKCS8EncodedKeySpec` i la pública amb `X509EncodedKeySpec`; per recuperar-les, `KeyFactory` reconstrueix els objectes `PrivateKey`/`PublicKey` a partir d'aquestes especificacions. Aplicacions reals d'aquest mecanisme són els certificats digitals (DNle, idCAT), la signatura de codi (APK, executables), el correu signat (S/MIME), la firma de fitxers JAR, el blockchain i el mateix TLS/HTTPS.

Per signar i distribuir un fitxer, Java ofereix eines de línia de comandaments: es crea el JAR amb `jar`, es genera la parella de claus amb `keytool -genkey`, se signa el JAR amb `jarsigner`, s'exporta el certificat públic amb `keytool -export`, i s'envien al receptor tant el JAR com el certificat. Aquest, per la seva banda, importa el certificat com a certificat de confiança (`keytool -import`) i verifica la firma amb `jarsigner -verify -verbose -certs`; en la sortida, la lletra "s" indica que la firma s'ha verificat correctament i la "k" que la clau pública és de confiança i es troba al magatzem de claus.

3.7 Protocols criptogràfics: combinant les primitives

Un protocol criptogràfic defineix la seqüència de passos i regles perquè dues parts es comuniquin de forma segura, combinant xifrat, hash i signatura. L'**intercanvi de claus Diffie-Hellman** permet que dues parts acordin una clau secreta compartida a través d'un canal insegur sense transmetre-la mai directament:

```
public class IntercanviDH {
    public static KeyPair generarParell() throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
        kpg.initialize(2048);
        return kpg.generateKeyPair();
    }

    public static byte[] calcularSecretCompartit/PrivateKey
    ↪ clauPropia, PublicKey clauAliena) throws Exception {
        KeyAgreement ka = KeyAgreement.getInstance("DH");
        ka.init(clauPropia);
        ka.doPhase(clauAliena, true);
        return ka.generateSecret(); // aquest secret es fa servir per
        ↪ derivar una clau AES
    }
}
```

TLS (successor d'SSL) és l'exemple més complet de protocol criptogràfic, perquè combina totes les primitives anteriors en el seu *handshake*: el client saluda el servidor indicant els algorismes que suporta; el servidor respon amb el seu certificat, que conté la seva clau pública signada per una Autoritat de Certificació; totes dues parts acorden una clau de sessió simètrica (mitjançant Diffie-Hellman efímer en el TLS modern); i, a partir d'aquí, tota la comunicació es xifra amb AES fent servir aquesta clau. Altres protocols que apliquen aquests mateixos principis en contextos diferents són SSH (accés remot), IPsec (VPN a escala de xarxa), Kerberos (autenticació centralitzada en dominis, base d'Active Directory) i OAuth2/OpenID Connect (autorització i autenticació web).

4. Certificats digitals i cadena de confiança

Un certificat digital certifica que una entitat determinada (usuari, màquina, dispositiu o procés) posseeix una clau pública concreta, enllaçant aquesta clau amb el nom del titular. El format estàndard universalment acceptat és el **X.509**, que conté camps com la versió, el número de sèrie, l'algorisme de firma, l'emissor, les dates de vigència, el nom del propietari, la clau pública i, finalment, la firma digital de l'Autoritat de Certificació que el valida.

Un certificat és de confiança perquè es confia en la CA que l'ha signat; la CA arrel sol venir preinstal·lada al sistema operatiu o al navegador. Els certificats es poden consultar des del navegador (accedint al cademat de l'URL) o inspeccionar-los des de la línia d'ordres:

```
openssl s_client -connect proferamon.com:443 -showcerts
```

Avui en dia els certificats X.509 s'utilitzen sobretot per a dos usos molt diferents: com a peça central de TLS/HTTPS, on el certificat del servidor permet al navegador comprovar

que es connecta a qui diu ser (aquest ús és, amb diferència, el més estès a Internet), i com a identificació personal o d'empresa davant d'administracions i altres organismes (per exemple, el DNle o l'idCAT), que permet signar documents, presentar tràmits electrònics o xifrar informació perquè només el destinatari legítim la pugui llegir. A l'Estat espanyol, aquest segon ús el cobreixen entitats certificadores com la CATCert catalana, la FNMT estatal o proveïdors privats acreditats.

5. Comunicacions segures amb Java

Les dades que circulen per la xarxa poden ser interceptades, per la qual cosa cal protegir-ne tant la privadesa com la integritat, especialment quan inclouen contrasenyes o dades bancàries. Els protocols SSL i TLS es van dissenyar amb aquest objectiu, i **JSSE** (*Java Secure Socket Extension*) n'és la implementació Java, dins dels paquets `javax.net` i `javax.net.ssl`, amb xifratge de dades, autenticació de servidor i client, i integritat dels missatges.

Quan l'aplicació és client d'un servei HTTPS, el `HttpClient` modern gestiona tot el TLS de manera transparent:

```
public class ClientHttps {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient(); // TLS
        ↪ gestionat automàticament

        HttpRequest peticio = HttpRequest.newBuilder()
            .uri(URI.create("https://proferamon.com/tic/"))
            .GET()
            .build();

        HttpResponse<String> resposta = client.send(peticio,
            ↪ HttpResponse.BodyHandlers.ofString());
        System.out.println("Codi d'estat: " + resposta.statusCode());
    }
}
```

Quan cal implementar el mateix servidor o client de sockets segurs, les classes `SSLSocket` i `SSLServerSocket` deriven de les convencionals `Socket` i `ServerSocket`, i s'obtenen a partir de les fàbriques `SSLServerSocketFactory`/`SSLSocketFactory` amb el mètode estàtic `getDefault()`. El servidor necessita un certificat, generat amb `keytool -genkey` o `-genkeypair`, indicat mitjançant les propietats `javax.net.ssl.keyStore` i `javax.net.ssl.keyStorePassword`; el client, per la seva banda, ha d'importar aquest certificat (o la CA que l'ha signat) al seu magatzem de confiança i indicar-lo amb `javax.net.ssl.trustStore`/`javax.net.ssl.trustStorePassword`. En proves internes és habitual generar un certificat autosignat i importar-lo manualment amb `keytool -importcert`; en producció s'utilitza una CA reconeguda com Let's Encrypt.

```
// Servidor
System.setProperty("javax.net.ssl.keyStore", "servidor.p12");
```

```

System.setProperty("javax.net.ssl.keyStorePassword",
    ↪ System.getenv("KEYSTORE_PASS"));
System.setProperty("javax.net.ssl.keyStoreType", "PKCS12");

SSLServerSocketFactory factory = (SSLServerSocketFactory)
    ↪ SSLServerSocketFactory.getDefault();
try (SSLServerSocket servidor = (SSLServerSocket)
    ↪ factory.createServerSocket(8443)) {
    servidor.setEnabledProtocols(new String[]{"TLSv1.3", "TLSv1.2"});
    ↪ // sense protocols antics
    while (true) {
        try (SSLSocket socket = (SSLSocket) servidor.accept();
            BufferedReader in = new BufferedReader(new
                ↪ InputStreamReader(socket.getInputStream()));
            PrintWriter out = new
                ↪ PrintWriter(socket.getOutputStream(), true)) {
            out.println("ECO: " + in.readLine());
        }
    }
}

```

```

// Client
System.setProperty("javax.net.ssl.trustStore",
    ↪ "client-truststore.p12");
System.setProperty("javax.net.ssl.trustStorePassword",
    ↪ System.getenv("TRUSTSTORE_PASS"));

SSLSocketFactory factory = (SSLSocketFactory)
    ↪ SSLSocketFactory.getDefault();
try (SSLSocket socket = (SSLSocket) factory.createSocket("localhost",
    ↪ 8443);
    PrintWriter out = new PrintWriter(socket.getOutputStream(),
        ↪ true);
    BufferedReader in = new BufferedReader(new
        ↪ InputStreamReader(socket.getInputStream())) {
    out.println("Hola, servidor!");
    System.out.println("Resposta: " + in.readLine());
}

```

Un cop establerta la connexió, el mètode `getSession()` de `SSLSocket` retorna un objecte `SSLSession` amb informació sobre la sessió: l'amfitrió, el conjunt de xifratge utilitzat, el protocol, l'identificador de sessió i el certificat de l'altra part (`getPeerCertificates()`). Altres protocols que resolen necessitats semblants a diferents capes són SSH i SFTP/FTPS (accés remot i transferència de fitxers), IPsec i WireGuard (VPN), DNSSEC/DoH/DoT (resolució DNS autenticada) i S/MIME o PGP (correu xifrat i signat).

6. Control d'accés

El control d'accés implica tres processos essencials: la **identificació** (el subjecte indica qui és), l'**autenticació** (es verifica que és realment qui diu ser ---típicament amb usuari i contrasenya, però també amb targetes intel·ligents, biometria o tokens) i l'**autorització** (es determina si, un cop autenticat, té accés al recurs sol·licitat). Aplicat a una aplicació, això es tradueix habitualment en tres tasques: autenticació, autorització i **auditoria** (registrar qui ha fet què i quan).

6.1 El model natiu de la JVM

Abans d'executar cap bytecode, la màquina virtual Java prepara l'entorn amb tres components de seguretat: el **carregador de classes** (amb els carregadors *bootstrap*, d'extensió i d'aplicació), el **verificador de fitxers de classes** (que comprova, per exemple, que les variables s'inicialitzin abans d'usar-se o que no es vulnerin les regles d'accés a mètodes privats) i el **gestor de seguretat**, una classe que decideix si una operació concreta (carregar un subprocés, llegir una propietat del sistema, obrir una connexió de xarxa, eliminar un fitxer...) està permesa. Per defecte no s'instal·la cap gestor de seguretat, i totes les operacions es permeten; en activar-lo (-Djava.security.manager o `System.setSecurityManager(new SecurityManager())`), qualsevol operació no autoritzada explícitament llança una `AccessControlException`.

Aquesta autorització es defineix en **fitxers de polítiques** (el global, *java.policy*, i opcionalment un de personal, *.java.policy*, a la carpeta de l'usuari), amb entrades del tipus:

```
grant codeBase "URL" {  
    permission Nombre_classe "Nombre_destino", "Accion";  
}
```

Els noms de classe de permís més habituals són `java.io.FilePermission` (amb convencions com `"/*"` per a un directori, `"/*-"` per a un directori recursiu, o `"<<ALL FILES>>"` per a qualsevol fitxer), `java.net.SocketPermission` (amb accions com *accept*, *listen*, *connect* o *resolve*) i `java.util.PropertyPermission`. L'eina **PolicyTool**, inclosa al JDK, permet editar gràficament aquests fitxers i verificar-ne la sintaxi.

Sobre aquesta base es construeix **JAAS** (*Java Authentication and Authorization Service*), que amplia el model de seguretat centrat en el codi perquè també tingui en compte **qui** l'executa, no només **quin** codi s'executa. Hi intervenen el `LoginContext` (inicia i gestiona l'autenticació, creant un `Subject` en cridar `login()`), el `LoginModule` (defineix el mecanisme concret d'autenticació), el `Subject` (l'entitat autenticable), el `Principal` (un atribut concret d'un `Subject` ja autenticat, com el seu nom d'usuari) i el `CallbackHandler` (recull les dades d'autenticació mitjançant diferents tipus de *callback*: `NameCallback`, `PasswordCallback`, `ChoiceCallback`, `ConfirmationCallback` o `TextOutputCallback`). Un cop autenticat l'usuari, l'autorització s'aconsegueix configurant entrades per als seus principals al fitxer de polítiques i associant el `Subject` al context d'accés amb `Subject.doAs()` o `doAsPrivileged()`; si l'usuari no té prou permisos per a l'operació sol·licitada, es captura una `SecurityException`.

6.2 Control d'accés basat en rols (RBAC) a nivell d'aplicació

En comptes d'assignar permisos usuari per usuari (inviàble a gran escala) o de dependre del model natiu de la JVM ---força verbós per a la lògica de negoci d'una aplicació---, el patró més habitual en aplicacions empresarials és el **RBAC**: els permisos s'agrupen en rols, i a cada usuari se li assignen un o més rols (Usuari → Rol(s) → Permís(os) → Recurs). Per exemple, en una aplicació de gestió acadèmica:

Rol	Permisos
ALUMNE	Consultar les seves qualificacions
PROFESSOR	Consultar i modificar qualificacions dels seus grups
CAP_ESTUDIS	Consultar totes les qualificacions del centre
ADMIN	Gestionar usuaris i rols

Altres models de control d'accés que val la pena conèixer són el **DAC** (el propietari del recurs decideix qui hi accedeix), el **MAC** (obligatori, basat en etiquetes de classificació, com en sistemes militars) i l'**ABAC** (basat en atributs com l'hora, la ubicació o el dispositiu). RBAC és el més habitual per la seva simplicitat de manteniment.

En Java, aquest model es pot implementar de manera senzilla amb enumeracions i un mapa de permisos per rol:

```
public enum Rol { ALUMNE, PROFESSOR, CAP_ESTUDIS, ADMIN }

public enum Permis {
    VEURE_QUALIFICACIONS_PROPIES, VEURE_QUALIFICACIONS_GRUP,
    MODIFICAR_QUALIFICACIONS_GRUP, VEURE_QUALIFICACIONS_CENTRE,
    ↪ GESTIONAR_USUARIS
}

public class GestorPermisos {
    private static final Map<Rol, Set<Permis>> PERMISOS_PER_ROL = new
    ↪ EnumMap<>(Rol.class);
    static {
        PERMISOS_PER_ROL.put(Rol.ALUMNE,
            ↪ EnumSet.of(Permis.VEURE_QUALIFICACIONS_PROPIES));
        PERMISOS_PER_ROL.put(Rol.PROFESSOR, EnumSet.of(
            Permis.VEURE_QUALIFICACIONS_GRUP,
            ↪ Permis.MODIFICAR_QUALIFICACIONS_GRUP));
        PERMISOS_PER_ROL.put(Rol.CAP_ESTUDIS,
            ↪ EnumSet.of(Permis.VEURE_QUALIFICACIONS_CENTRE));
        PERMISOS_PER_ROL.put(Rol.ADMIN, EnumSet.allOf(Permis.class));
    }

    public static boolean tePermis(Set<Rol> rolsUsuari, Permis permis)
    ↪ {
        return rolsUsuari.stream()
            .flatMap(rol -> PERMISOS_PER_ROL.getOrDefault(rol,
            ↪ Set.of()).stream())
            .anyMatch(p -> p == permis);
    }
}
```

I la comprovació es fa explícita al servei de negoci, de manera anàloga a com el gestor de seguretat de la JVM denega una operació no autoritzada:

```
public class ServeiQualificacions {
    public void modificarQualificacio(Usuari usuari, String alumneId,
        ↪ double nota) {
        if (!GestorPermisos.tePermis(usuari.getRols(),
            ↪ Permis.MODIFICAR_QUALIFICACIONS_GRUP)) {
            throw new ControlAccesException(
                usuari.getNom() + " no té permís per modificar
                ↪ qualificacions");
        }
        // ... lògica de negoci
    }
}
```

Un patró força utilitzat en aplicacions reals (per exemple, amb Spring Security) és declarar el permís necessari directament sobre el mètode mitjançant una anotació, en comptes d'escriure la comprovació manualment cada vegada; un *proxy* o un aspecte (AOP) l'intercepta i en verifica el compliment abans d'executar el mètode, exactament el mateix mecanisme que fan servir `@PreAuthorize` a Spring o `@RolesAllowed` a Jakarta EE:

```
@Retention(RetentionPolicy.RUNTIME)
@interface RequereixPermis { Permis value(); }

public class ServeiQualificacionsAnotat {
    @RequereixPermis(Permis.MODIFICAR_QUALIFICACIONS_GRUP)
    public void modificarQualificacio(Usuari usuari, String alumneId,
        ↪ double nota) {
        // ... lògica de negoci
    }
}
```

Val la pena notar el paral·lisme entre els dos nivells de control d'accés: JAAS treballa a escala de plataforma, associant permisos molt granulars (lectura d'un fitxer, obertura d'un socket) a principals concrets mitjançant fitxers de polítiques, mentre que l'RBAC d'aplicació treballa a escala de lògica de negoci, associant operacions de domini (modificar una qualificació) a rols. En sistemes complexos és habitual combinar-los: el gestor de seguretat de la JVM i les seves polítiques protegeixen els recursos del sistema (fitxers, xarxa, propietats), mentre que l'RBAC de l'aplicació en protegeix la lògica.

7. Depuració, manteniment i documentació

El manteniment és clau per a la seguretat a llarg termini d'una aplicació: cal seguir normes de codificació i documentació, retirar el codi obsolet i analitzar a fons tots els canvis abans de posar-los en producció. En la fase de depuració orientada a seguretat convé:

- Utilitzar el depurador de l'IDE per verificar que les dades confidencials (contrasenyes, claus) no queden mai en text pla en memòria més temps de l'estrictament necessari.
- Revisar els logs generats: no hi ha d'aparèixer cap contrasenya, token, clau privada ni dada personal confidencial.
- Provar casos límit: entrades buides, molt llargues o amb caràcters especials (' OR 1=1 --, <script>, rutes com ../../etc/passwd).
- Fer proves amb usuaris de cada rol per confirmar que el control d'accés denega correctament allò que no els correspon ---proves negatives, no només positives.

Pel que fa a la documentació, cal Javadoc a totes les classes i mètodes públics relacionats amb seguretat (indicant, per exemple, quin algorisme s'utilitza i per què), documentar per escrit la política de seguretat (rols existents, permisos de cadascun, gestió de claus), mantenir un registre de canvis ---cada modificació en la lògica de seguretat és sensible i s'ha de poder rastrejar--- i no documentar mai contrasenyes, claus o dades de connexió reals, ni en comentaris ni en cap README:

```
/**
 * Xifra un text amb AES-256 en mode GCM.
 * <p>
 * S'utilitza un IV aleatori de 12 bytes per a cada operació, requisit
 * imprescindible per a la seguretat del mode GCM (vegeu NIST SP
 * ↪ 800-38D).
 *
 * @param textPla text a xifrar en UTF-8
 * @param clau clau AES de 256 bits
 * @return text xifrat codificat en Base64 (IV + text xifrat + tag
 * ↪ d'autenticació)
 */
public static String xifrar(String textPla, SecretKey clau) throws
    ↪ Exception { ... }
```

Resum

Els fonaments teòrics de la criptografia ---resums, xifratge simètric i asimètric, firma digital, certificats--- es tradueixen en Java en un conjunt reduït d'API estables (JCA/JCE per a les operacions criptogràfiques, JSSE per a les comunicacions segures i el parell gestor de seguretat/JAAS per al control d'accés a escala de plataforma), però la manera de fer-les servir a la pràctica ha evolucionat: DES i MD5/SHA-1 han quedat substituïts per AES-GCM i SHA-256 o superior; les contrasenyes ja no es guarden amb un simple resum, sinó amb algorismes lents com PBKDF2 o BCrypt; i el control d'accés d'una aplicació moderna sol resoldre's amb un RBAC senzill a escala de lògica de negoci ---sovint declaratiu, amb anotacions--- més que amb fitxers de polítiques de la JVM. En qualsevol cas, cap d'aquestes eines compensa una mala disciplina bàsica: validar tota entrada externa, no confiar mai en res que vingui de fora del mateix codi, no exposar informació confidencial en errors ni en logs, i mantenir i documentar la lògica de seguretat amb el mateix rigor amb què es dissenya.

Versions d'aquest document

- [HTML](#)
- [PDF](#)
- [ODT](#)
- [MD](#)

[Domini Públic \(CC0\)](#)