# API crypto en java

# API sécurité en Java

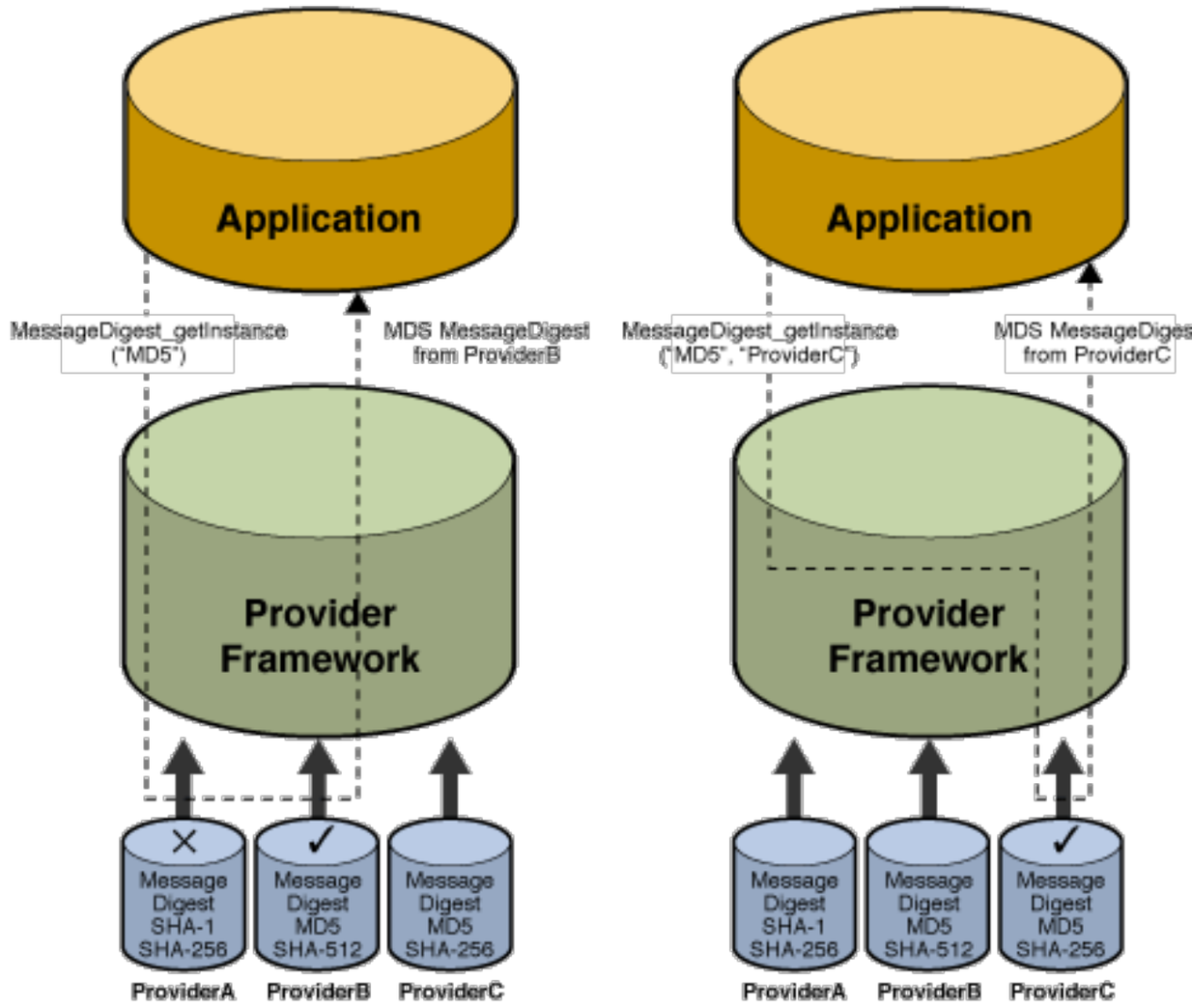
- Architecture:
  - Classe Provider
  - Classe Security
  - Classe SecureRandom
  - Classe MessageDigest
  - Classe Signature
  - Classe Cipher
  - Streams
    - The CipherInputStream Class
    - The CipherOutputStream Class
  - Classe SealedObject
  - Classe Mac

# Provider

- java.security.Provider
- un provider pour implémenter divers services de sécurité

md = MessageDigest.getInstance("MD5");

md = MessageDigest.getInstance("MD5", "ProviderC");



# Classes « engine » et algorithmes

- fournit des interfaces de services cryptographiques indépendants de l'implémentation d'un provider particulier:
  - *SecureRandom*: pour générer des nombres aléatoires et pseudo-aléatoires
  - *MessageDigest*: pour calculer des empreintes « messages digest » (hash)
  - *Signature*: initialisés avec des clés pour signer et vérifier des signatures digitales
  - *Cipher*: chiffrement/déchiffrement avec divers types d'algorithmes (symétriques AES DES … asymétriques RSA …)
  - Message Authentication Codes (*Mac*): génère des empreintes avec des clés pour protéger l'intégrité des messages
  - des classes pour les clés et les certificats

# Les providers…

```
public class SecInfo {
    public static void main(String[] args) {
        Provider [] lesProv= Security.getProviders();
        for(Provider p: lesProv){
            System.out.println(p.getName()+':'+p.getInfo());
        }
    }
}
```

SUN:SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests; SecureRandom; X.509 certificates; JKS & DKS keystores; PKIX CertPathValidator; PKIX CertPathBuilder; LDAP, Collection CertStores, JavaPolicy Policy; JavaLoginConfig Configuration)

SunRsaSign:Sun RSA signature provider

SunEC:Sun Elliptic Curve provider (EC, ECDSA, ECDH)

SunJSSE:Sun JSSE provider(PKCS12, SunX509/PKIX key/trust factories, SSLv3/TLSv1/TLSv1.1/TLSv1.2)

SunJCE:SunJCE Provider (implements RSA, DES, Triple DES, AES, Blowfish, ARCFOUR, RC2, PBE, Diffie-Hellman, HMAC)

SunJGSS:Sun (Kerberos v5, SPNEGO)

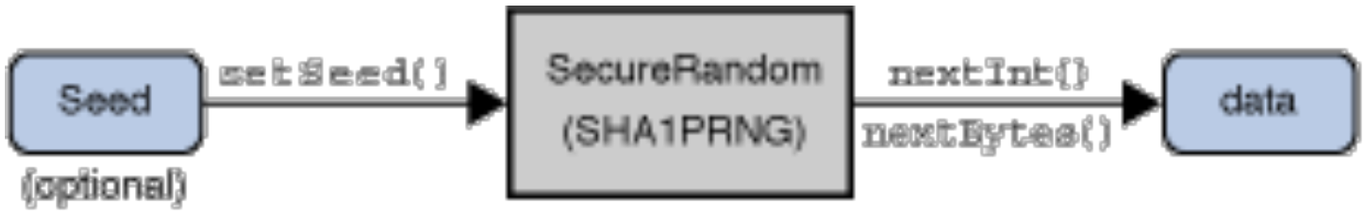
SunSASL:Sun SASL provider(implements client mechanisms for: DIGEST-MD5, GSSAPI, EXTERNAL, PLAIN, CRAM-MD5, NTLM; server mechanisms for: DIGEST-MD5, GSSAPI, CRAM-MD5, NTLM)

XMLDSig:XMLDSig (DOM XMLSignatureFactory; DOM KeyInfoFactory; C14N 1.0, C14N 1.1, Exclusive C14N, Base64, Enveloped, XPath, XPath2, XSLT TransformServices)

SunPCSC:Sun PC/SC provider

Apple:Apple Provider

# SecureRandom



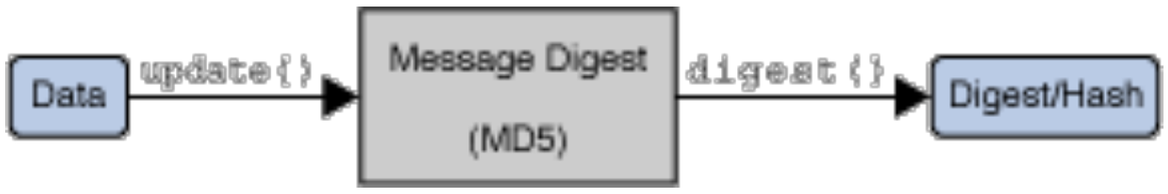

- génération de nombres aléatoires

  synchronized public void setSeed(byte[] seed)

  public void setSeed(long seed)

  synchronized public void nextBytes(byte[] bytes)

# MessageDigest




- Pour créer des « digest » : hash
  - Création de l'objet:

    MessageDigest sha = MessageDigest.*getInstance*("SHA-1");
  - hachage de i1, i2, i3 (byte array)

    sha.*update*(i1);

    sha.update(i2);

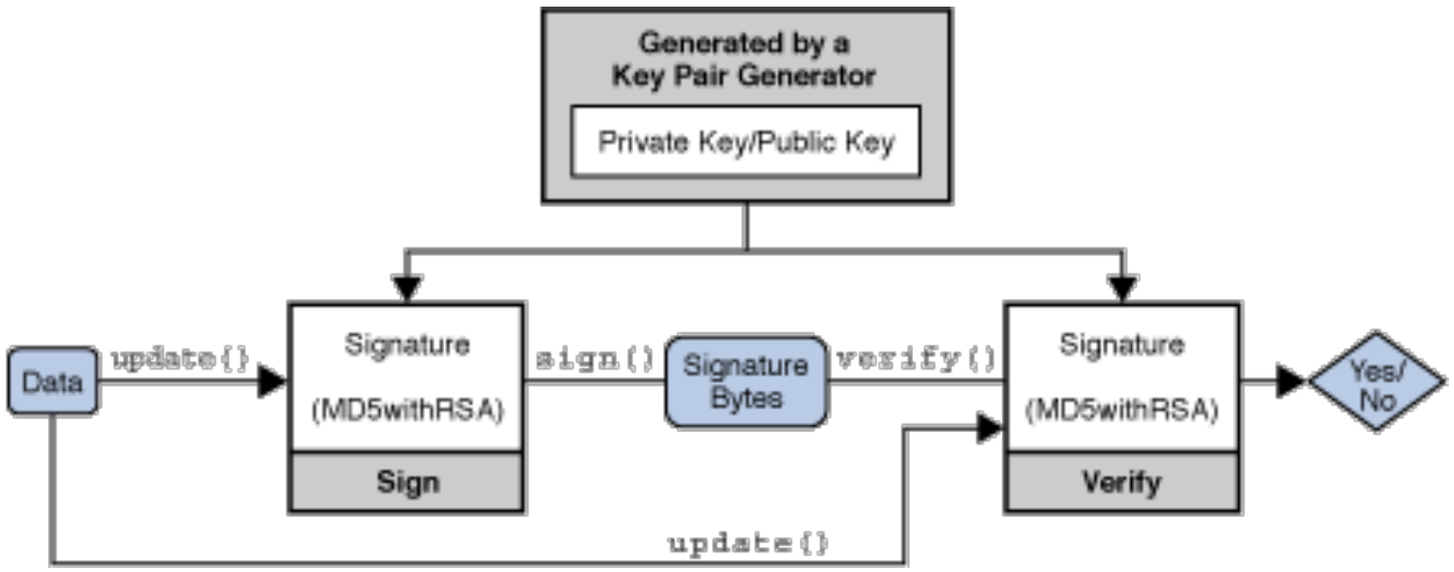
    sha.update(i3);
  - le digest

    byte[] hash = sha.*dig*est();

# Exemple MD5

```
public class DigestEx {
    public static void main(String[] args) {
       try{
            MessageDigest sha = MessageDigest.getInstance("MD5");
            FileInputStream fis = new FileInputStream("tmp.txt");
            BufferedInputStream bufin = new BufferedInputStream(fis);
            byte[] buffer = new byte[1024];
            int len;
            while (bufin.available() != 0) {
                len = bufin.read(buffer);
                sha.update(buffer, 0, len);
                };
            bufin.close();
            byte[] dig=sha.digest();
            System.out.println("digest :\n"+byte2Hexa(dig));
      }catch(NoSuchAlgorithmException e){e.printStackTrace();}
       catch(IOException e){e.printStackTrace();}
   }
}
```

# Signature




- « engine » pour les signatures digitales
  - *update sign*
  - *update verify*
  - en utilisant une paire clé publique/ clé privée

# Structure du programme

```
import java.io.*;
import java.security.*;
class GenSignature {
    public static void main(String[] args) {
        /* Générer la signature */
        if (args.length != 1) {
            System.out.println("Usage: GenSignature fichierAsigner");
            }
        else try{
            /* Générer les clés */
            /* Créer un objet signature initialisé avec le clé privée */
            /* Mis à jour et signature des données */
            /*  générer une signature for it */
            /* Sauver la signature dans un fichier */
            /* Sauver la clé publique dan un fichier */

        } catch (Exception e) {
            System.err.println("Exception " + e.toString());
        }

};
```

# Générer les clés, la signature, lire les données

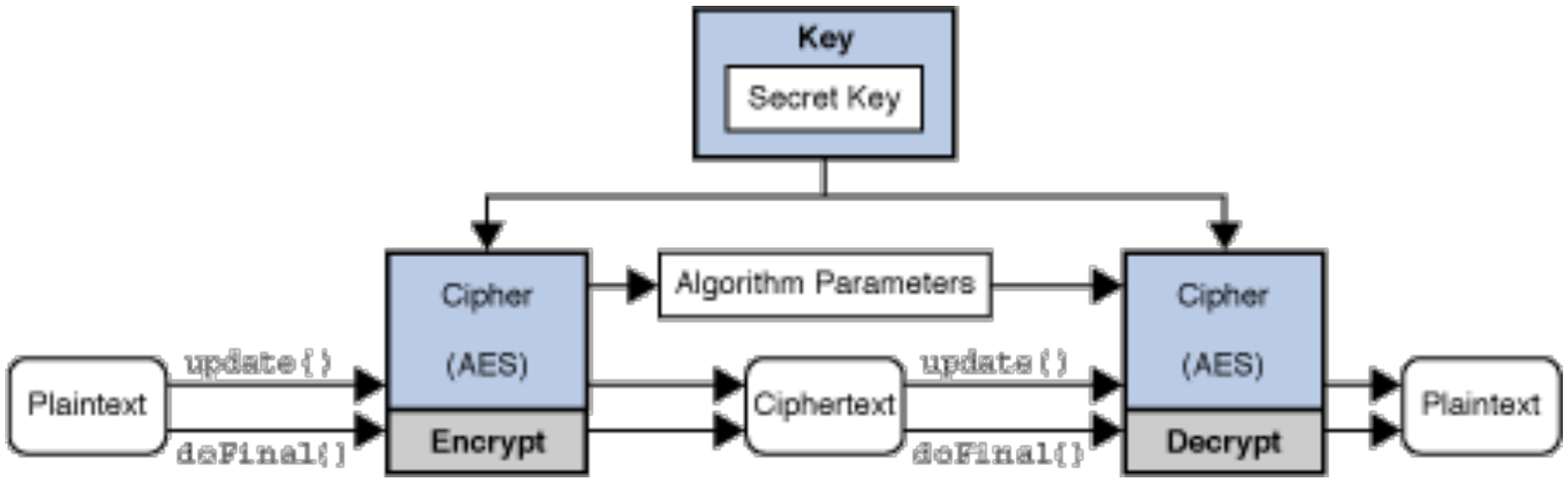
```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
KeyPair pair = keyGen.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();

Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
dsa.initSign(priv);

FileInputStream fis = new FileInputStream("fichier");
BufferedInputStream bufin = new BufferedInputStream(fis);
byte[] buffer = new byte[1024];
int len;
while (bufin.available() != 0) {
    len = bufin.read(buffer);
    dsa.update(buffer, 0, len);
 };
bufin.close();
```

# signer, sauvegarder

```
byte[] realSig = dsa.sign();


/* Sauver la signature dans un fichier */
FileOutputStream sigfos = new FileOutputStream("signature");
sigfos.write(realSig);

sigfos.close();


/* Sauvegarder la clé publique */
byte[] key = pub.getEncoded();
FileOutputStream keyfos = new FileOutputStream("cle");
keyfos.write(key);

keyfos.close();
```

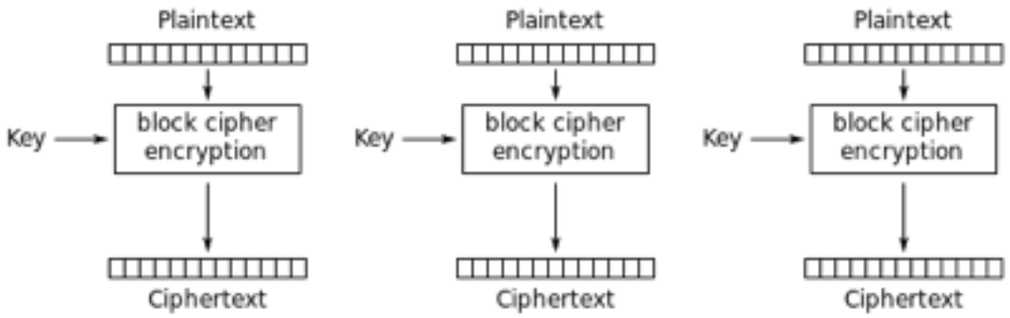
```
try{      /* importer la clé publique */
          FileInputStream keyfis = new FileInputStream("cle");
          byte[] encKey = new byte[keyfis.available()];
          keyfis.read(encKey);
          keyfis.close();
          X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
          KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
          PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
          /* entrer la signature */
          FileInputStream sigfis = new FileInputStream("signature");
          byte[] sigToVerify = new byte[sigfis.available()];
          sigfis.read(sigToVerify );
          sigfis.close();
          /* créer un objet Signature initialisé avec la clé publique*/
          Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
          sig.initVerify(pubKey);
          /*Vérifier */
          FileInputStream datafis = new FileInputStream("fichier");
          BufferedInputStream bufin = new BufferedInputStream(datafis);
          byte[] buffer = new byte[1024];
          int len;
          while (bufin.available() != 0) {
              len = bufin.read(buffer);
              sig.update(buffer, 0, len);
              };
        bufin.close();
       boolean verifies = sig.verify(sigToVerify);
         System.out.println("signature verifiée: " + verifies);
      } catch (Exception e) {
          System.err.println("exception " + e.toString());
};
```

# Cipher

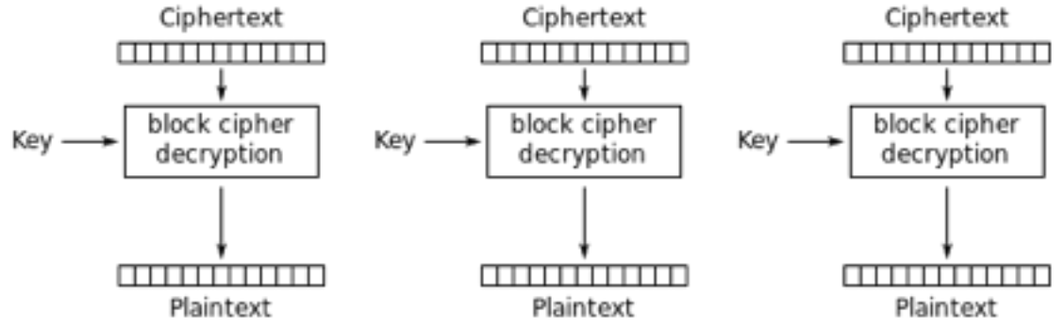



- symétrique /asymétrique
- par bloc en flot
- avec « padding » et vecteur d'initialisation (modes)
- init() DECRYPT_MODE ENCRYPT_MODE + autres paramètres
- update() doFinal()

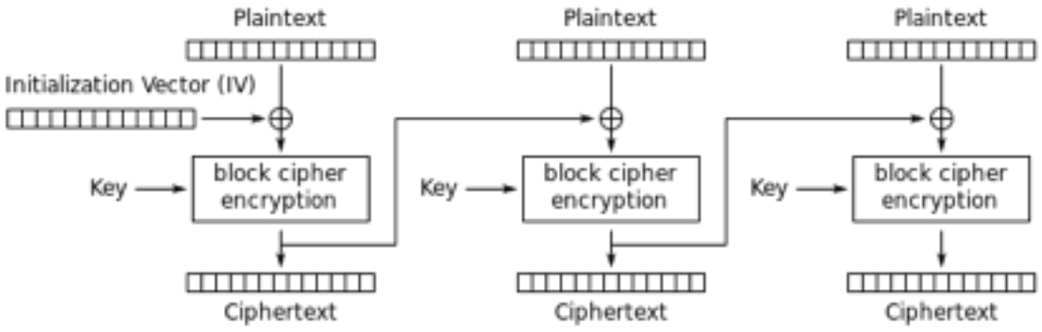
# modes…



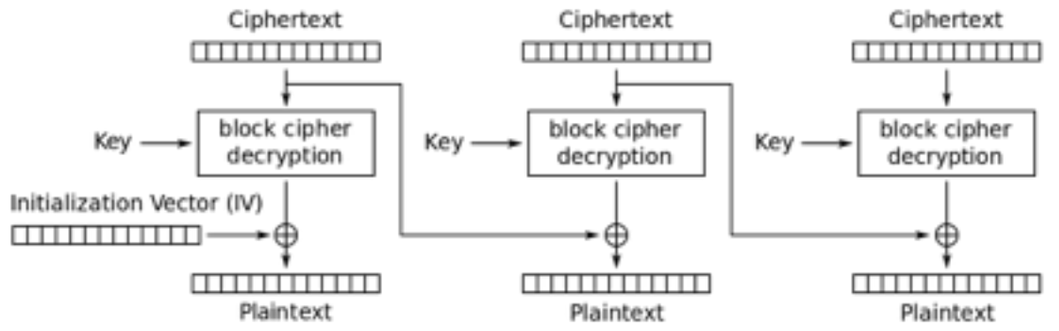

Electronic Codebook (ECB) mode encryption



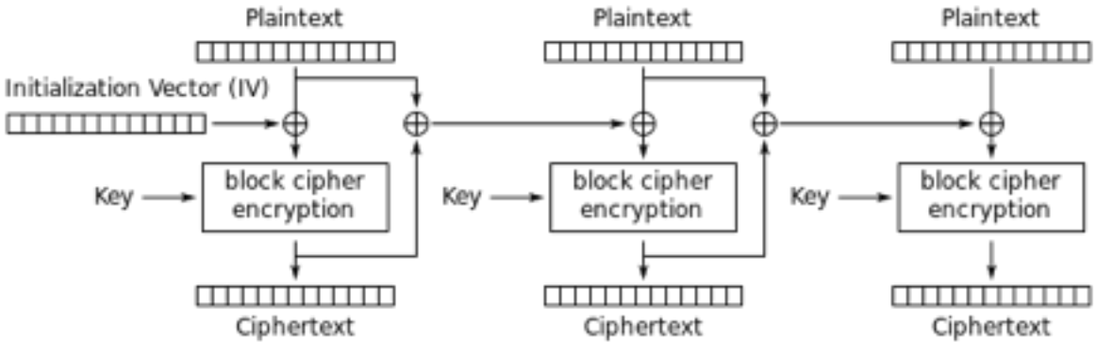

Electronic Codebook (ECB) mode decryption



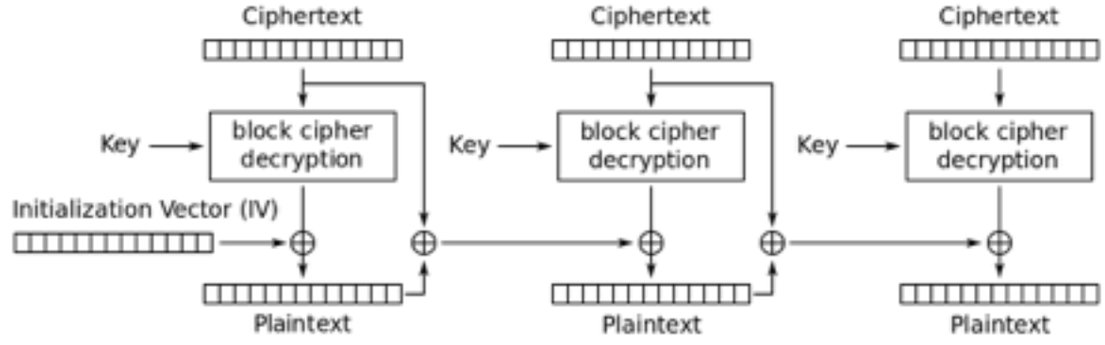

Cipher Block Chaining (CBC) mode encryption




Cipher Block Chaining (CBC) mode decryption




Propagating Cipher Block Chaining (PCBC) mode encryption




Propagating Cipher Block Chaining (PCBC) mode decryption

# remplissage

- chiffrement pas blocs:
  - DES chiffre des blocs de 8 octets (64 bits)
  - AES chiffre des blocs de 16 octets (128 bits)
  - il faut compléter les blocs (padding):
    - PKCS#5 pour DES: XX XX XX XX 04 04 04 04
    - (il manque 4 octets pour le bloc: on répète 04 dans les 4 derniers octets manquants)

# des

```
public class DESEex{
        public static void main(String[] argv) {
            try{
                KeyGenerator keygenerator = KeyGenerator.getInstance("DES","SunJCE");
                SecretKey maCleDES = keygenerator.generateKey();
                Cipher chiffreDES;
                // Créer le chiffre
                chiffreDES = Cipher.getInstance("DES/ECB/PKCS5Padding");
                chiffreDES.init(Cipher.ENCRYPT_MODE, maCleDES); //initialiser
                //chiffrer
                byte[] text = "le texte secret ».getBytes(); //texte secret
                byte[] textChiffre = chiffreDES.doFinal(text); //chiffré
                System.out.println("Texte chiffré :\n"  + byte2Hexa(textChiffre));  //afficher en hexa
                //déchiffrer
                chiffreDES.init(Cipher.DECRYPT_MODE, maCleDES); //initialiser
                byte[] textDechiffre = chiffreDES.doFinal(textChiffre);
                System.out.println("Texte déchiffré : " + new String(textDechiffre));
            }
           catch(NoSuchProviderException e){ e.printStackTrace();
            }catch(NoSuchAlgorithmException e){e.printStackTrace();
            }catch(NoSuchPaddingException e){e.printStackTrace();
            }catch(InvalidKeyException e){e.printStackTrace();
            }catch(IllegalBlockSizeException e){e.printStackTrace();
            }catch(BadPaddingException e){e.printStackTrace();}
        }
}
```

# avec AES et iv

```
public static void main(String[] args) {
    try {
        String message = "Ceci est mon secret.";
        // générer une clé
        KeyGenerator keygen = KeyGenerator.getInstance("AES");
        keygen.init(128);
        byte[] key = keygen.generateKey().getEncoded();
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");

        // initialization vector (random).
        SecureRandom random = new SecureRandom();
        byte iv[] = new byte[16];    //générer IV de 16 bytes
        random.nextBytes(iv);
        IvParameterSpec ivspec = new IvParameterSpec(iv);

        // chiffre pour le chiffrement
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, skeySpec, ivspec);

        // chiffrer le message
        byte[] encrypted = cipher.doFinal(message.getBytes());
        System.out.println("texte chiffré: " + encrypted + "\n");
```

# Fin

```
            // reinitialiser le chiffre pour le déchiffrage
            cipher.init(Cipher.DECRYPT_MODE, skeySpec, ivspec);

            // déchiffrer
            byte[] decrypted = cipher.doFinal(encrypted
        } catch (IllegalBlockSizeException | BadPaddingException |
              UnsupportedEncodingException | InvalidKeyException |
                  InvalidAlgorithmParameterException | NoSuchPaddingException |
                  NoSuchAlgorithmException ex) {
            ex.printStackTrace();
        }
    }
```

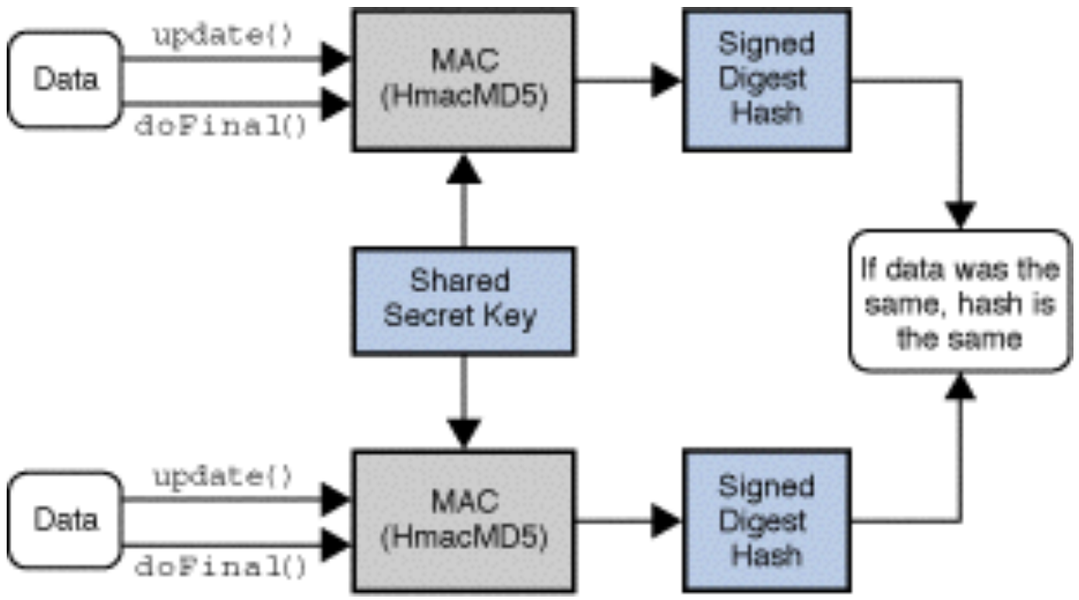
# CipherInputStream CipherOutputStream

```
public static void main(String[] args) throws Exception {
    // chiffre et clé
    Cipher cipher = Cipher.getInstance("AES");
    KeyGenerator keyGen = KeyGenerator.getInstance("AES");
    SecretKey secKey = keyGen.generateKey();
    // chiffrement
    cipher.init(Cipher.ENCRYPT_MODE, secKey);
    String cleartextFile = "texte_en_clair.txt";
    String ciphertextFile = "texte_chiffre.txt";

    FileInputStream fis = new FileInputStream(cleartextFile);
    FileOutputStream fos = new FileOutputStream(ciphertextFile);
    CipherOutputStream cos = new CipherOutputStream(fos, cipher);
    byte[] block = new byte[8];
    int i;
    while ((i = fis.read(block)) != -1) {
        cos.write(block, 0, i);
    }
    cos.close();
```

# CipherInputStream CipherOutputStream

```
        // Déchiffrer
        String cleartextAgainFile = "texte_dechiffre.txt";
        cipher.init(Cipher.DECRYPT_MODE, secKey);
        fis = new FileInputStream(ciphertextFile);
        CipherInputStream cis = new CipherInputStream(fis, cipher);
        fos = new FileOutputStream(cleartextAgainFile);
        while ((i = cis.read(block)) != -1) {
            fos.write(block, 0, i);
        }
        fos.close();
    }
}
```

# Mac




- update() doFinal()

```
public class MAC {
        public static void main(String[] args) {
                try {
                    KeyGenerator keyGen = KeyGenerator.getInstance("HmacMD5");
                    // générer la clé
                    SecretKey key = keyGen.generateKey();
                    // Créer le MAC
                    Mac mac = Mac.getInstance(key.getAlgorithm());
                    mac.init(key);
                    String message = "Le secret";
                    byte[] b = message.getBytes("UTF-8");
                    // créer le digest
                    byte[] digest = mac.doFinal(b);
                }
                catch (NoSuchAlgorithmException e) {
                        System.out.println("No Such Algorithm:" + e.getMessage());return;
                }
                catch (UnsupportedEncodingException e) {
                        System.out.println("Unsupported Encoding:" + e.getMessage());return;
                }
                catch (InvalidKeyException e) {
                        System.out.println("Invalid Key:" + e.getMessage());return;
                }
        }
}
```

# transférer du code

# Exécuter du code…

- exécuter un code dont on ne connaît pas la provenance est extrêmement dangereux:
  - soit limiter les possibilités de ce code (security manager)
  - soit avoir la garantie de sa provenance: signatures et authentifications

# Préliminaires…

# Authentification

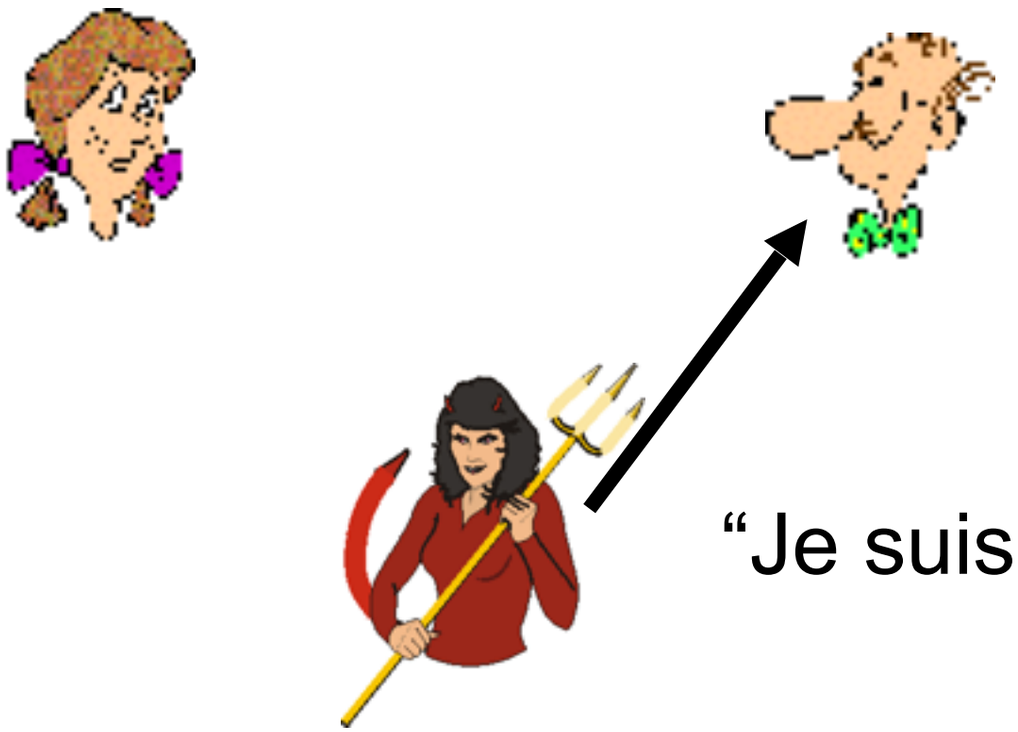
*Goal:* Bob veut que  Alice lui prouve son identité

Version 1 : Alice  dit “Je suis Alice”


# Authentification

*Goal:* Bob veut que  Alice lui prouve son identité

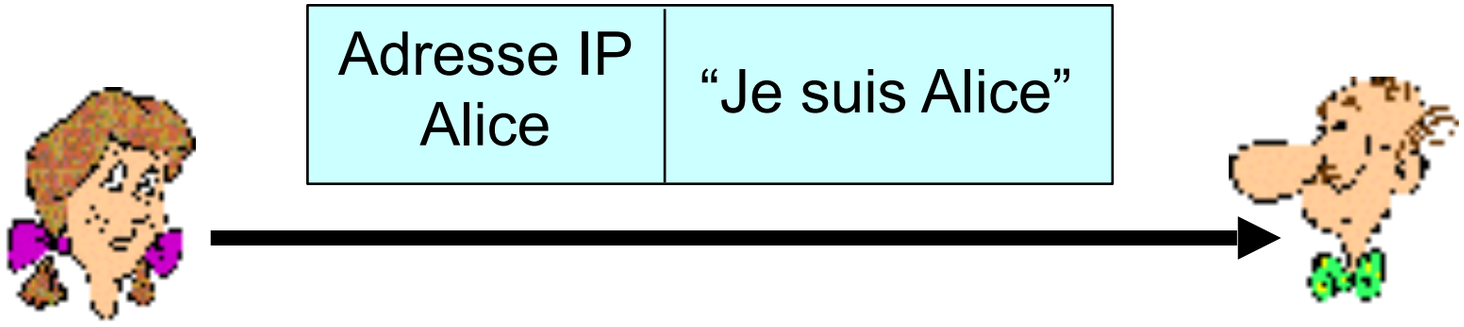
Alice  dit "Je suis Alice"




Bob ne voit pas Alice,
Trudy peut prétendre être Alice

# Authentification: autre essai

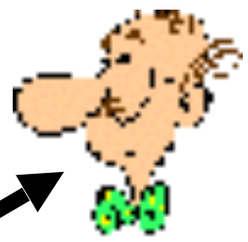
*Version 2:* Alice dit “je suis Alice”
dans un paquet IP avec son adresse IP

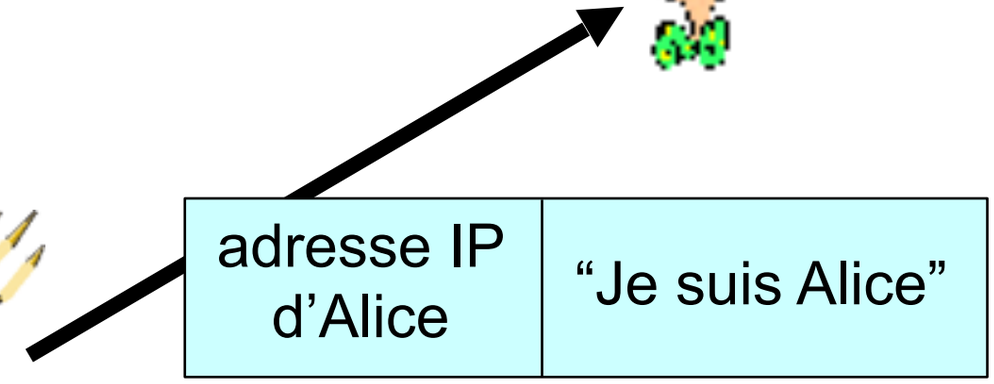



La faille??

# Authentification: autre essai

*Version 2:* Alice dit “je suis Alice”
dans un paquet IP avec son adresse IP

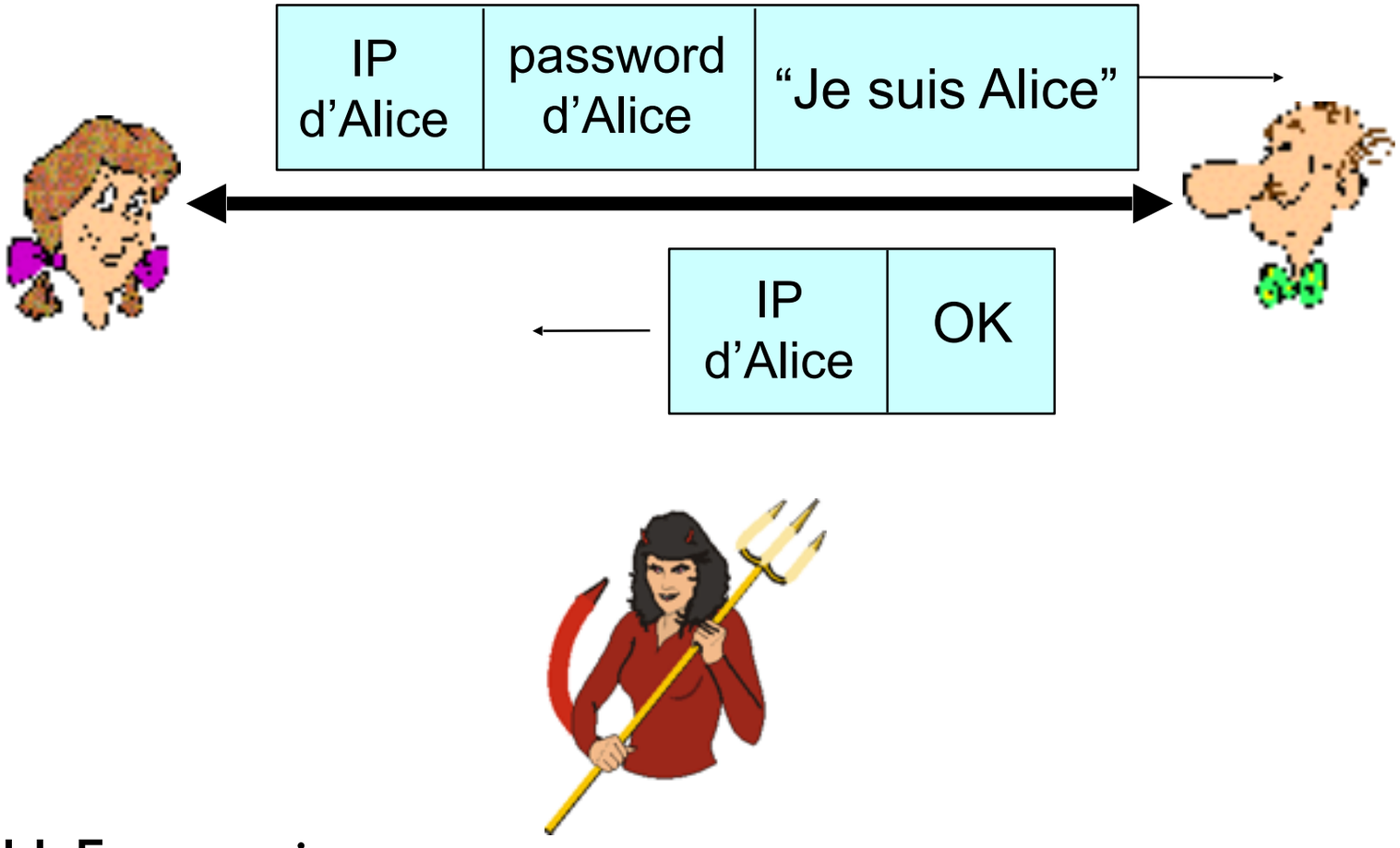





Trudy peut créer un paquet IP en “spoofant” l’adresse d’Alice

# Authentification: autre essai

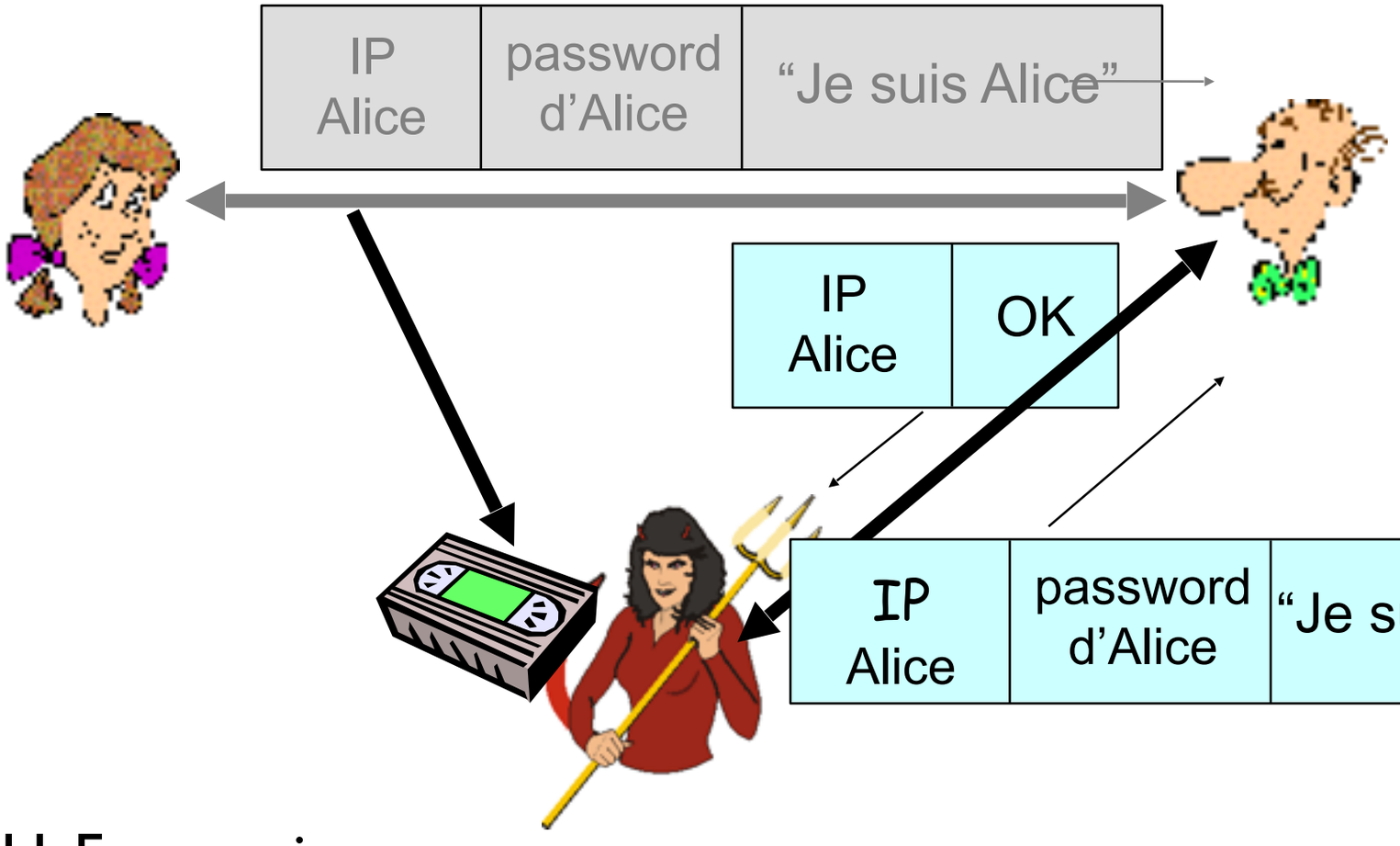
*Version 3:* Alice dit "Je suis Alice" et envoie son password pour le prouver.




Faiblesse ??

# Authentification: autre essai

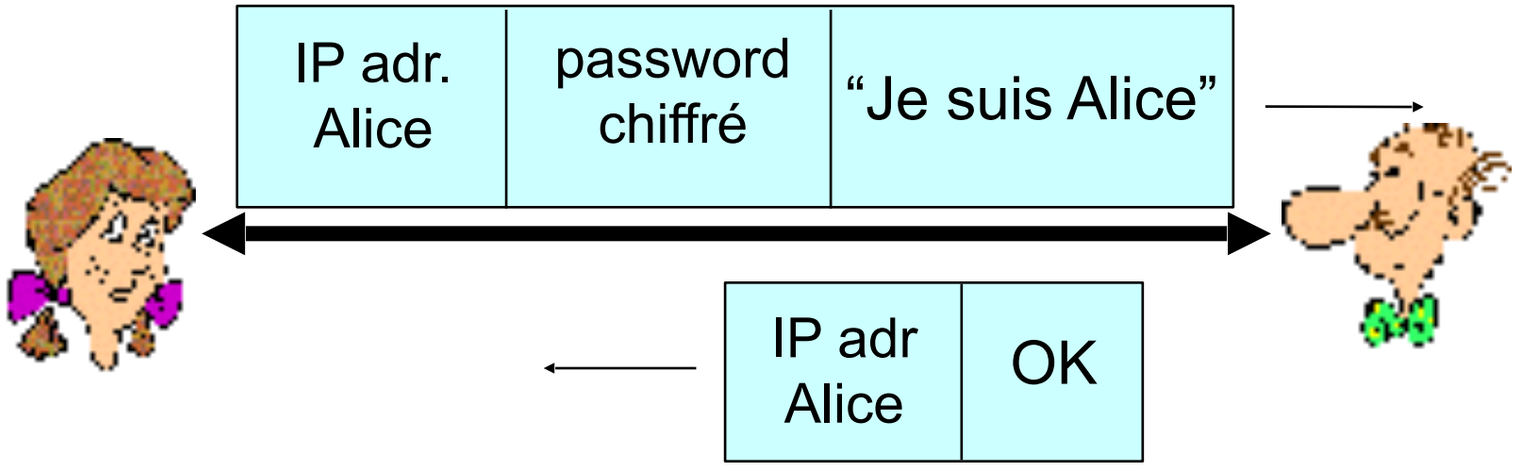
*Version 3:* Alice dit "Je suis Alice" et envoie son password pour le prouver.




*attaque par playback:* Trudy enregistre le paquet envoyé par Alice et le rejoue vers Bob

# Authentification: autre essai

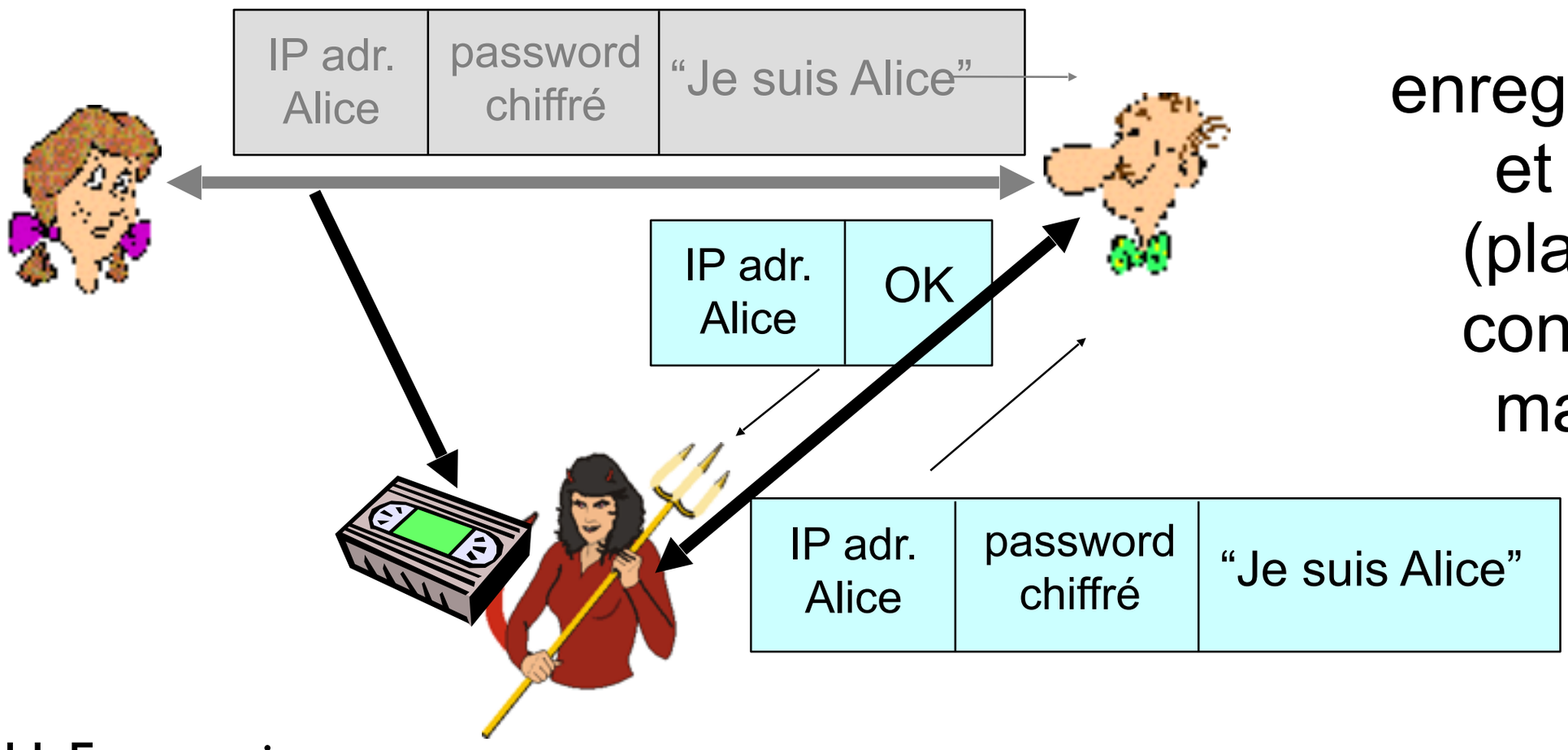
*Version 3:* Alice dit "Je suis Alice" et envoie son password *chiffré* pour le prouver.




La faille??

# Authentification: autre essai

*Version 3:* Alice dit "Je suis Alice" et envoie son password pour le prouver.



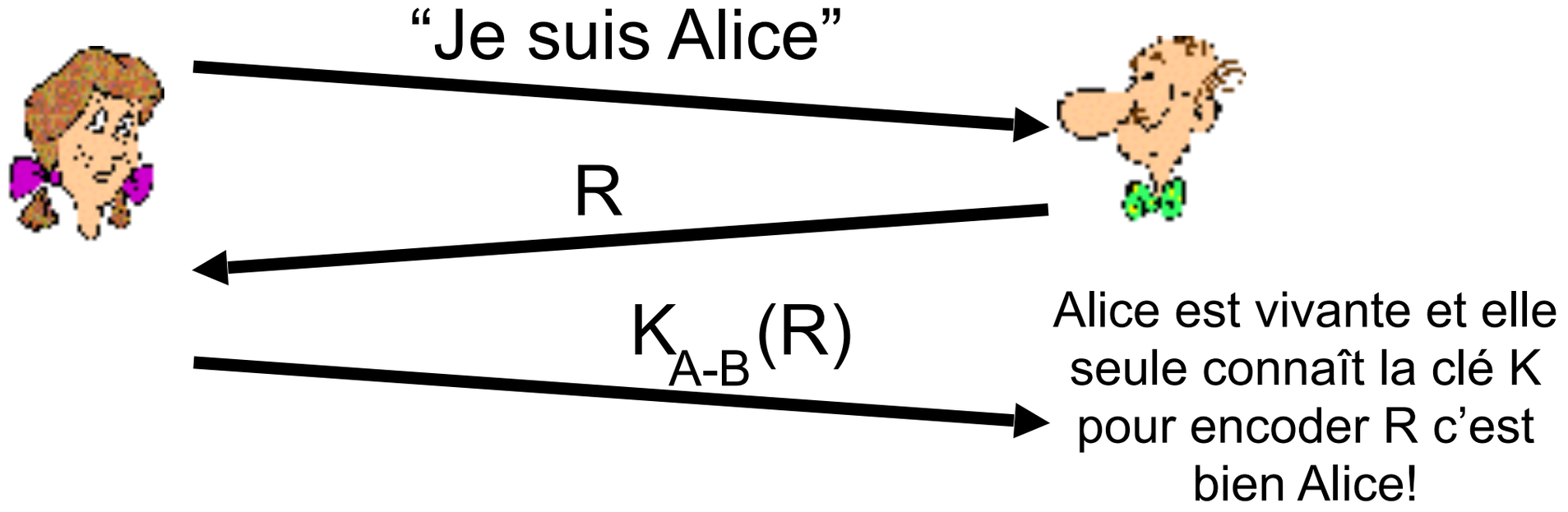

enregistrement et re-jeu (playback) continue à marcher

# Authentification: autre essai

*But:* éviter l'attaque par « playback »

*nonce:* nombre (R) utilisé *once-in-a-lifetime*

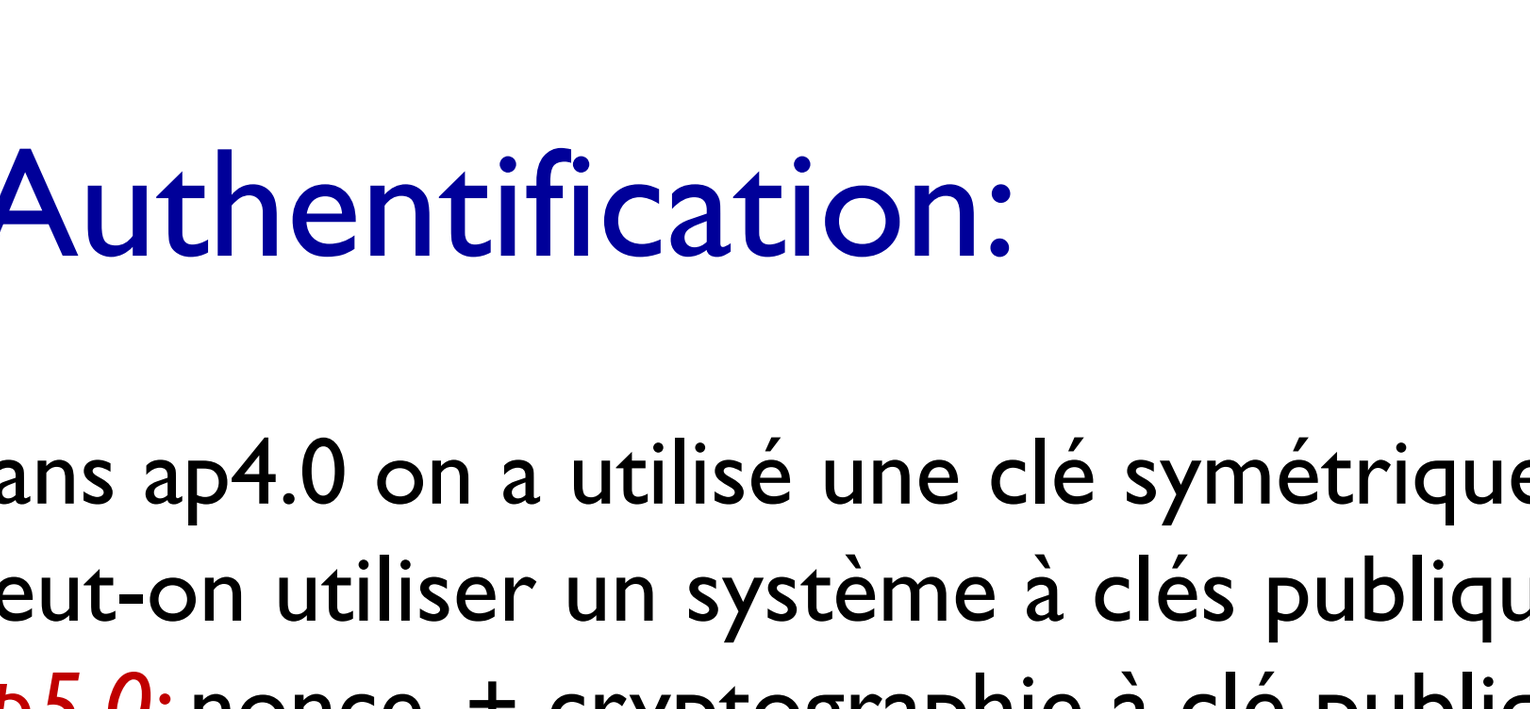
*v4:* pour prouver que c'est la vraie Alice, Bob envoie à Alice un *nonce* R. Alice renvoie R, chiffré avec la clé secrète partagée




Failles, inconvénients?

# Authentification:

dans ap4.0 on a utilisé une clé symétrique partagée
peut-on utiliser un système à clés publiques?
*ap5.0:* nonce, + cryptographie à clé publique




Bob calcule

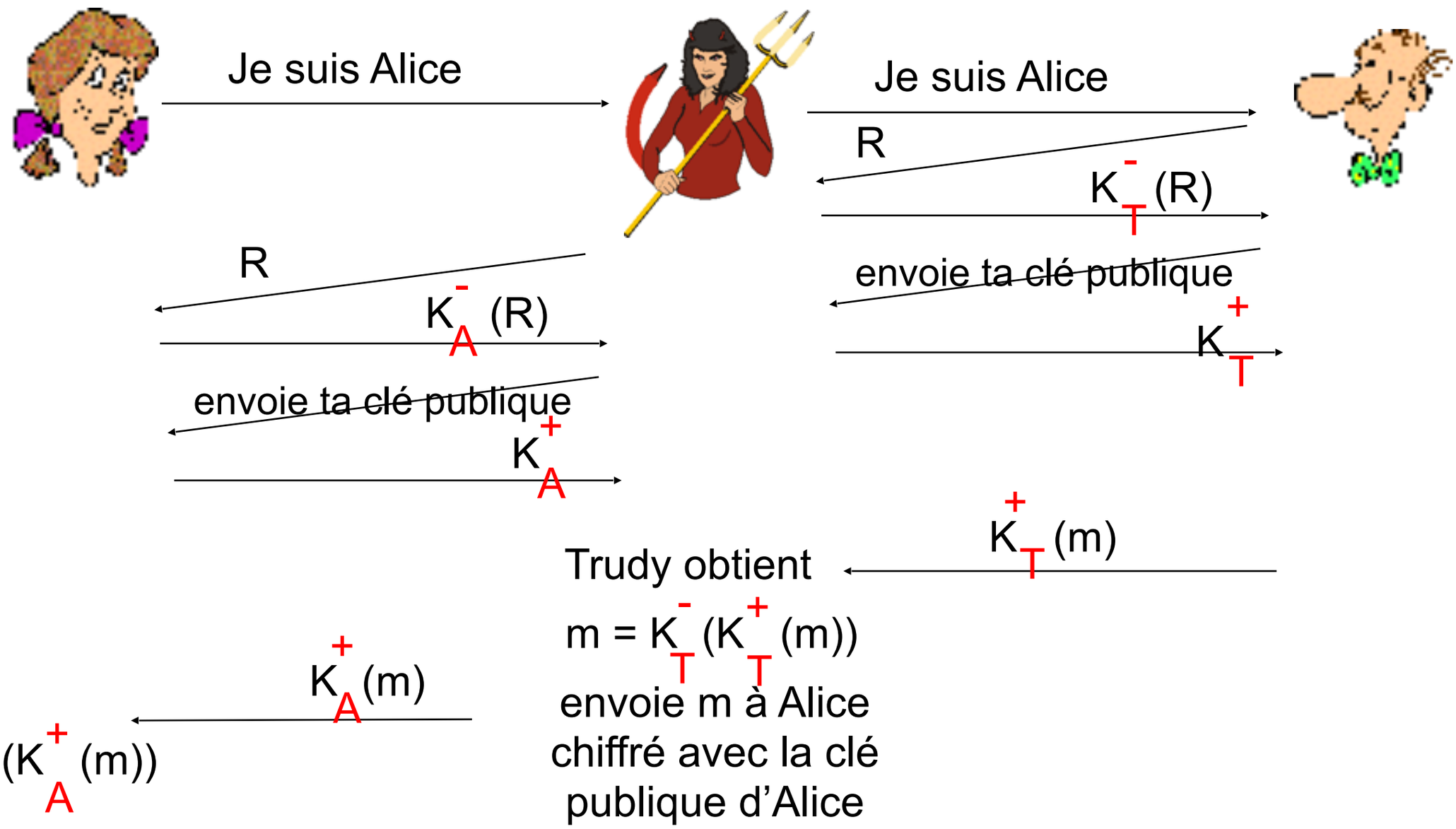
$$K_A^+(K_A^-(R)) = R$$

et sait que seule Alice a la clé privée qui a chiffré R de façon à ce que

$$K_A^+(K_A^-(R)) = R$$

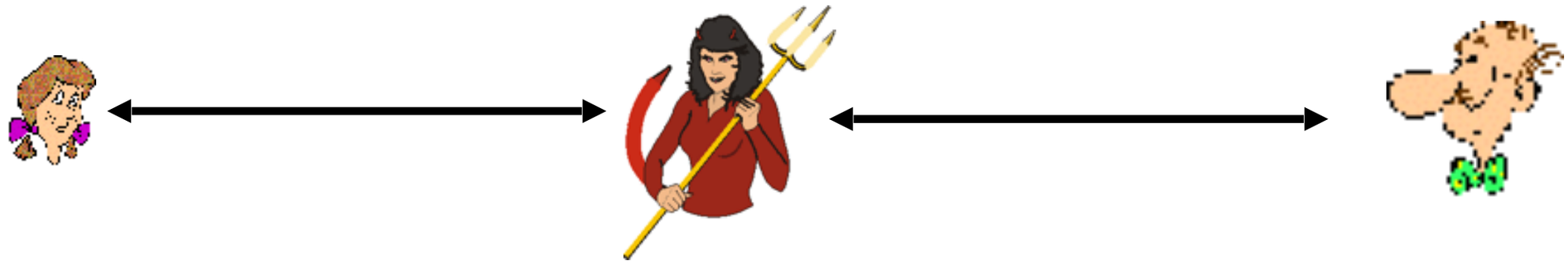
# ap5.0: trou de sécurité

*man in the middle:* Trudy se fait passer pour Alice auprès de Bob et pour Bob auprès d'Alice



# ap5.0: trou de sécurité

*man in the middle attack:* Trudy se fait passer pour Alice auprès de Bob et pour Bob auprès d'Alice

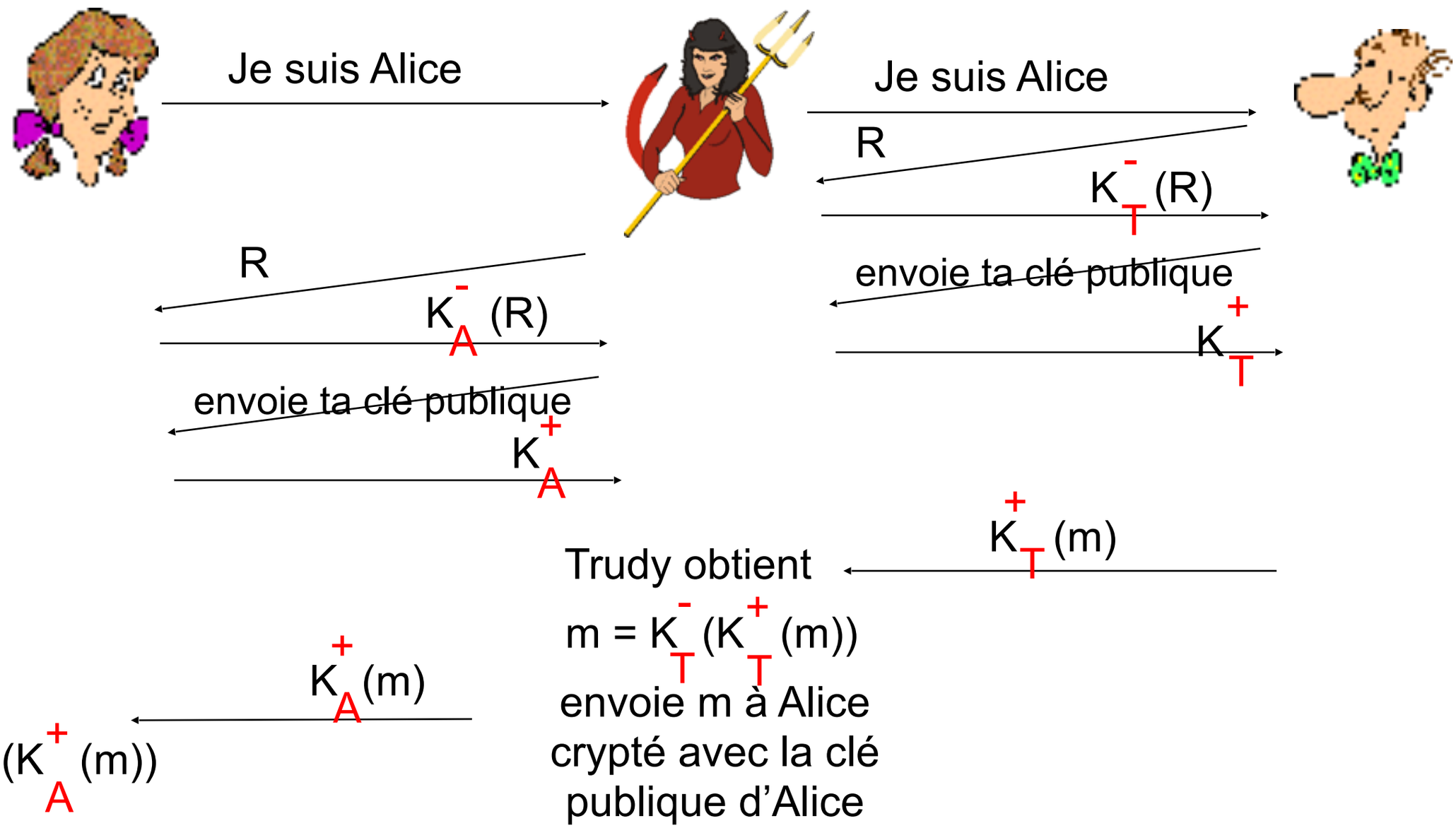


difficile à détecter:

- Bob reçoit tout ce qu'Alice envoie et vice-versa (Bob, Alice peuvent se rencontrer et se rappeler de leur conversation)
- mais Trudy reçoit tous les messages !

# ap5.0: trou de sécurité

*man in the middle:* Trudy se fait passer pour Alice auprès de Bob et pour Bob auprès d'Alice

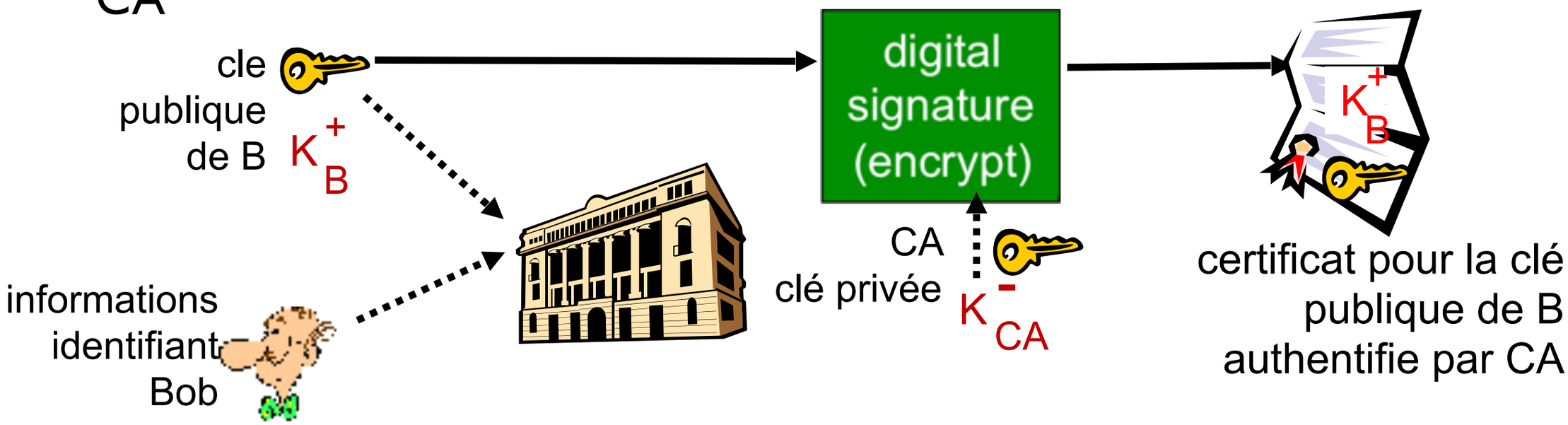


# Certification des clés publiques

- motivation:
  - Trudy envoie une commande par e-mail:
    *acheter 4 pizzas*
  - Trudy signe la commande avec sa clé privée
  - Trudy envoie au magasin sa clé publique mais prétend que c'est celle de Bob
  - le magasin vérifie la signature elle envoie les 4 pizzas à Bob
  - Bob n'a rien demandé!

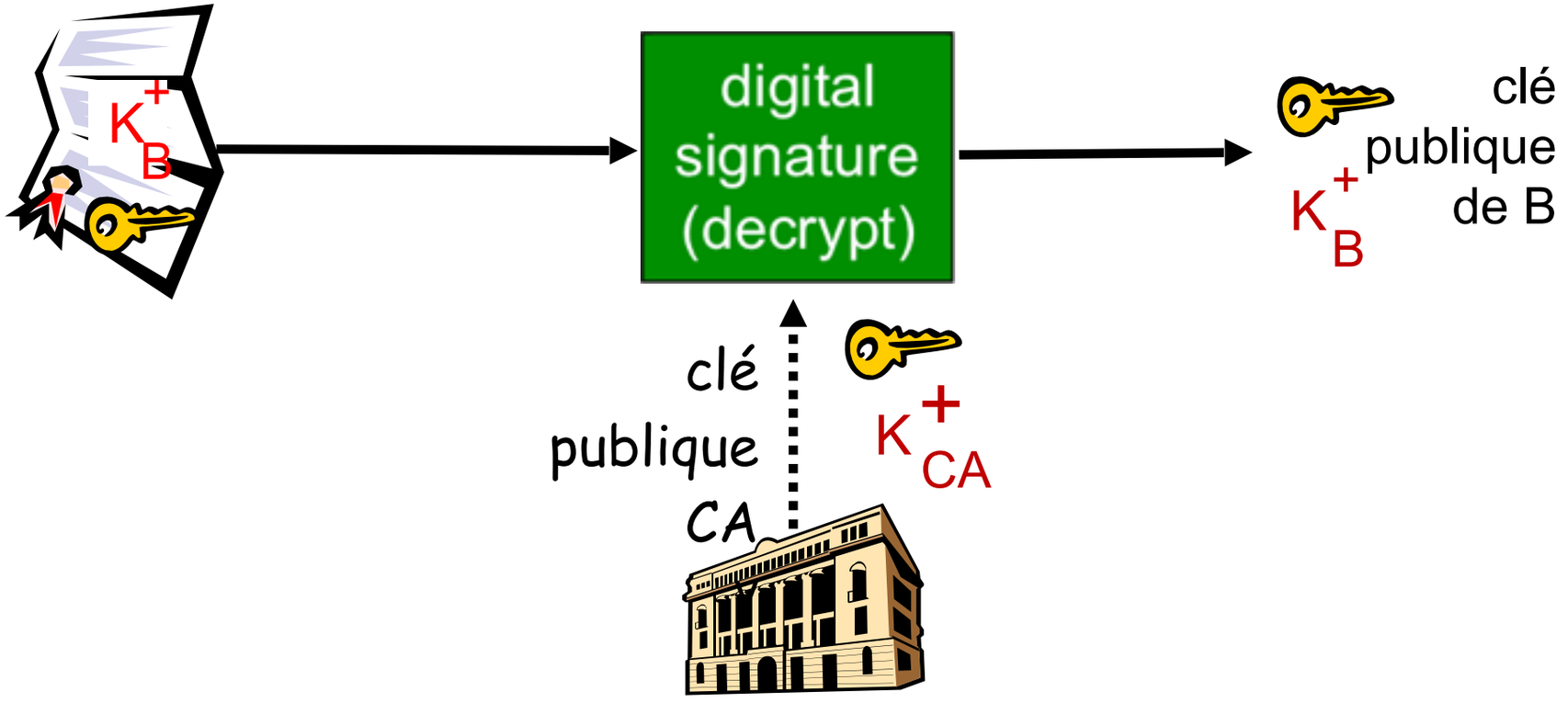
# Autorités de certification

- *certification authority (CA):* associe une clé publique à une entité E.
- E (personne, site) enregistre sa clé publique auprès de l'autorité de certification
- E fournit la preuve de l'identité par la CA.
  - CA crée un certificat associant E à sa clé publique.
  - ce certificat contient la clé publique de E signée numériquement par CA



# Autorités de certification

- quand Alice veut obtenir la clé publique de Bob:
  - elle obtient le certificat de Bob (de n'importe qui).
  - applique la clé publique de CA au certificat de Bob, et obtient la clé publique de Bob.



# Signatures digitales

- On suppose un système à clés asymétriques et que seul le possesseur de la clé privée connaît la clé privée et que la clé publique peut être connue de tous.
- Pour signer un document:
  - on signe le document avec la clé privée
  - on envoie le document signé
  - on fournit la clé publique
  - le récepteur vérifie la signature avec la clé publique
- Problème: le récepteur vérifie que le document a été signé par quelqu'un ayant une clé privée correspondant à la clé publique mais cela ne prouve rien…
- il faut certifier la clé publique

# Certificat

- Un certificat contient:
  - une clé publique
  - des informations d'identification « certificate subject » (nom, organisation etc..)
  - une signature digitale: le certificat est signé par une entité
  - les informations d'identification de la source du certificat.

# Vérifier le certificat

- Pour vérifier le certificat:
  - la source (issuer) est une autorité « connue » avec une clé publique connue qui permet d'authentifier le certificat
  - sinon il faut vérifier la clé publique du certificat avec un autre certificat..
  - jusqu'à obtenir une clé publique en qui on a confiance
  - on a ainsi une chaine de certificats
- Mais… ce n'est pas toujours possible: on peut générer une empreinte (fingerprint) du certificat et vérifier physiquement auprès de la source que cette empreinte est la bonne

# Autorités de certifications

- Certification authority (CA) certifie des clés publiques:
  - requête: self-signed certificate (CSR) vers une authority de certification,
  - CA vérifie votre identité (avec des moyens autres) et établit un certificat pour votre clé publique signée avec la clé privée du CA.
  - (éventuellement une chaine de certificats)

# Keystores

- Les certificats de confiance sont stockés dans le « keystore » comme étant des « trusted certificates »
- les clés publiques de ces certificats peuvent être utilisées pour vérifier les signatures
- Pour envoyer un code ou un document signé il faut joinder le certificat qui certifie la clé publique correspondant à la clé privée utilisée dans la signature
- les « keystores » contiennent:
  - les certificats de confiance
  - les couples clés privée / certificat de la clé publique

# keytool

- keytool est un outil java pour:
  - créer des clés privées/ publiques
  - créer des requêtes de certificat vers des CA
  - importer des réponses à ces requêtes provenant de CA
  - importer des certificats de clés publiques
  - gérer le keystore

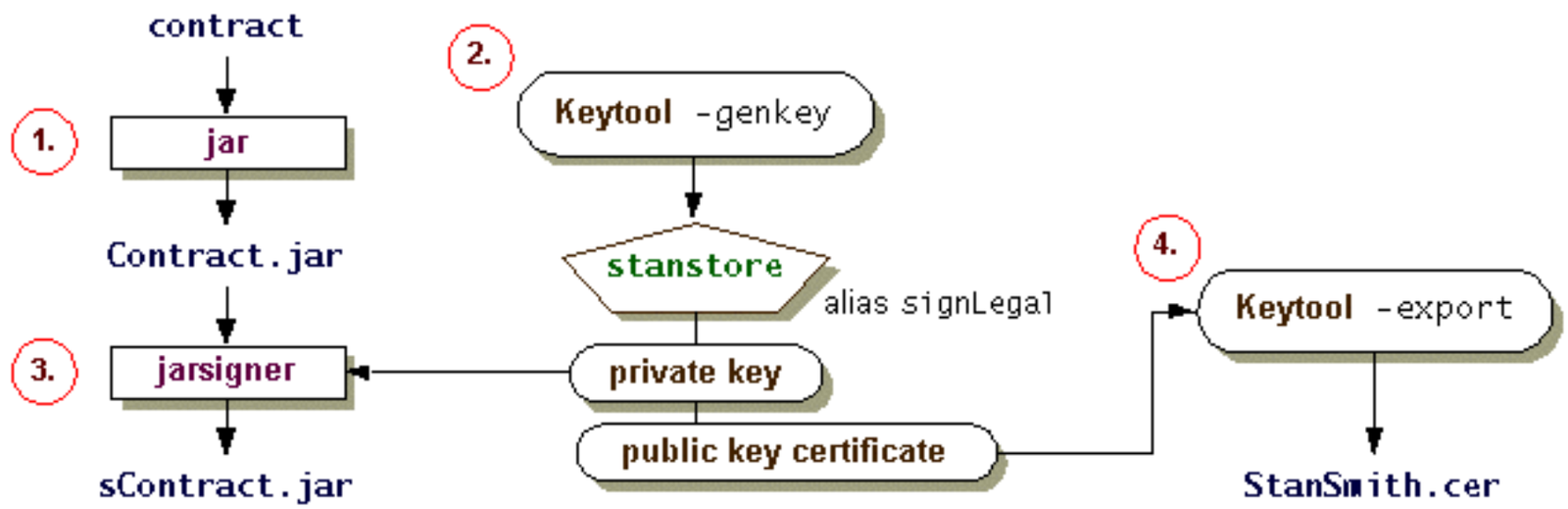
# Comment faire?

- en utilisant JAR et (jarsigner)
  - créer un fichier JAR
  - générer des clés (keytool -genkey)
    - avec keytool -certreq faire une requête de certificat auprès d'une CA
    - avec keytool -import importer la réponse
  - signer le fichier JAR avec jarsigner et la clé privée
  - exporter le certificat de la clé publique: keytool -export
  - (On peut aussi comment réaliser les mêmes choses avec l'API sécurité du jdk)

# Avec les autorités

- keytool génère un certificat auto-signé (signé avec la clé privée)
- Pour avoir un certificat signé par une autorité de certification:
- générer un Certificat Signing Request (CSR)
  - keytool -certreq -alias unAlias -file csrFile
  - soumettre csrFile à une autorité de certification
    - qui retourne un certificat signé (ou une chaîne) par le CA
      - keytool -import -alias alias -file ABCCA.cer -keystore storefile  ajoute le certificat
  - keytool -import -trustcacerts -keystore storefile -alias alias
  -  -file certReplyFile insère la réponse au CSR (avec vérification)
  - (*java.home*/lib/security/cacerts pour le système)

# Envoyer un contrat de StanSmith




- Créer le JAR (jar cvf Contract.jar contract)
- générer les clés
- signer le JAR
- exporter le certificat

# Générer les clés

```
hf$ keytool -genkey -alias signLegal -keystore examplestore
Entrez le mot de passe du fichier de clés :
Le mot de passe du fichier de clés est trop court : il doit comporter au moins 6 caractères
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Quels sont vos nom et prénom ?
  [Unknown]:  Stan Smith
Quel est le nom de votre unité organisationnelle ?
  [Unknown]:  Legal
Quel est le nom de votre entreprise ?
  [Unknown]:  UFRInfo
Quel est le nom de votre ville de résidence ?
  [Unknown]:  Paris
Quel est le nom de votre état ou province ?
  [Unknown]:  PARIS
Quel est le code pays à deux lettres pour cette unité ?
  [Unknown]:  FR
Est-ce CN=Stan Smith, OU=Legal, O=UFRInfo, L=Paris, ST=PARIS, C=FR ?
  [non]:  oui

Entrez le mot de passe de la clé pour <signLegal>
   (appuyez sur Entrée s'il s'agit du mot de passe du fichier de clés) :

Un keystore exmplestore a été créé avec clé privée clé publique pour Stan Smith avec un
certificat auto-signé  (il est valide 90 jours) et est associé à la clé privée identifiée
comme signLegal
```

# Signer le JAR

```
hf$ jarsigner -keystore examplestore -signedjar sContract.jar Contract.jar signLegal
Enter Passphrase for keystore:
jar signed.

Warning:
The signer certificate will expire within six months.
No -tsa or -tsacert is provided and this jar is not timestamped. Without a
timestamp, users may not be able to validate this jar after the signer certificate's
expiration date (2015-06-30) or after any future revocation date.
```

# Exporter le certificat

```
hf$ keytool -export -keystore examplestore -alias signLegal -file StanSmith.cer
Entrez le mot de passe du fichier de clés :
Certificat stocké dans le fichier <StanSmith.cer>
```

# Récepteur…

- Ruth a reçu de StanSmith
  - sContract.jar le JAR signé
  - le certificat *StanSmith.cer* (qui contient la clé privée et la clé publique)
  - il faut importer le certificat dans le keystore:

# Récepteur…

```
hf$ keytool -import -alias -stan -file StanSmith.cer -keystore exempleruth
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Propriétaire : CN=Stan Smith, OU=Legal, O=UFRInfo, L=Paris, ST=PARIS, C=FR
Emetteur : CN=Stan Smith, OU=Legal, O=UFRInfo, L=Paris, ST=PARIS, C=FR
Numéro de série : f0dce85
Valide du : Wed Apr 01 17:47:39 CEST 2015 au : Tue Jun 30 17:47:39 CEST 2015
Empreintes du certificat :
    MD5:  2B:AC:09:AB:AE:D7:A0:76:68:16:13:22:AC:BF:D9:87
    SHA1 : 8E:3B:C3:24:C5:23:02:0E:B2:4A:BA:41:3A:2E:E7:79:66:BF:89:1D
    SHA256 : B0:22:7B:F8:3E:21:A3:F9:A2:02:F5:59:80:A8:92:FB:5C:FE:A1:AF:1F:1F:
28:A7:41:9E:EE:F5:D4:9D:15:AE
    Nom de l'algorithme de signature : SHA1withDSA
    Version : 3

Extensions :

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: B3 53 53 0F F2 E6 30 90   FD B8 2E 25 1E 82 72 B3  .SS...0....%..r.
0010: 79 05 87 C6                                        y...
]
]

Faire confiance à ce certificat ? [non] :  oui
Certificat ajouté au fichier de clés
```

# Vérifier le certificat

- Ruth téléphone à Stan. Stan a le « fingerprint » du certificat:

```
 hf$ keytool -printcert -file StanSmith.cer
Propriétaire : CN=Stan Smith, OU=Legal, O=UFRInfo, L=Paris, ST=PARIS, C=FR
Emetteur : CN=Stan Smith, OU=Legal, O=UFRInfo, L=Paris, ST=PARIS, C=FR
Numéro de série : f0dce85
Valide du : Wed Apr 01 17:47:39 CEST 2015 au : Tue Jun 30 17:47:39 CEST 2015
Empreintes du certificat :
    MD5:  2B:AC:09:AB:AE:D7:A0:76:68:16:13:22:AC:BF:D9:87
    SHA1 : 8E:3B:C3:24:C5:23:02:0E:B2:4A:BA:41:3A:2E:E7:79:66:BF:89:1D
    SHA256 : B0:22:7B:F8:3E:21:A3:F9:A2:02:F5:59:80:A8:92:FB:5C:FE:A1:AF:1F:1F:
28:A7:41:9E:EE:F5:D4:9D:15:AE
    Nom de l'algorithme de signature : SHA1withDSA
    Version : 3

Extensions :

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: B3 53 53 0F F2 E6 30 90   FD B8 2E 25 1E 82 72 B3  .SS...0....%..r.
0010: 79 05 87 C6                                        y...
]
]
```

# Vérifier la signature..

- Ruth:

```
hf$ jarsigner -verify -verbose -keystore exempleruth sContract.jar

s k      148 Wed Apr 01 17:58:38 CEST 2015 META-INF/MANIFEST.MF
         310 Wed Apr 01 17:58:38 CEST 2015 META-INF/SIGNLEGA.SF
        1057 Wed Apr 01 17:58:38 CEST 2015 META-INF/SIGNLEGA.DSA
           0 Wed Apr 01 17:42:00 CEST 2015 META-INF/
smk       15 Wed Apr 01 17:41:42 CEST 2015 contract

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.

Warning:
This jar contains entries whose signer certificate will expire within six months.
This jar contains signatures that does not include a timestamp. Without a timestamp,
users may not be able to validate this jar after the signer certificate's expiration date
(2015-06-30) or after any future revocation date.

Re-run with the -verbose and -certs options for more details.
```

# Résultat:

```
hf $jar xvf sContract.jar
décompressé : META-INF/MANIFEST.MF
décompressé : META-INF/SIGNLEGA.SF
décompressé : META-INF/SIGNLEGA.DSA
 créé : META-INF/
décompressé : contract
hf$ more contract
Je m'engage...
```

# Et maintenant avec du code…

```
import java.io.*;
public class Compter {
    public static void countChars(InputStream in) throws IOException
    {
        int cmp = 0;
        while (in.read() != -1)
            cmp++;
        System.out.println("On a " + cmp + " caractères.");
    }
    public static void main(String[] args) throws Exception
    {
        if (args.length >= 1)
            countChars(new FileInputStream(args[0]));
        else
            System.err.println("Usage: Compte fichier");
    }
}
```

```
java Compter.java
jar cvf Compter.jar Compter.class
```

# keytool

```
hf$ keytool -genkey -alias signFiles -keystore exemplestore
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Quels sont vos nom et prénom ?
  [Unknown]:  Hugues Fauconnier
Quel est le nom de votre unité organisationnelle ?
  [Unknown]:  LIAFA
Quel est le nom de votre entreprise ?
  [Unknown]:  paris-diderot
Quel est le nom de votre ville de résidence ?
  [Unknown]:  PARIS
Quel est le nom de votre état ou province ?
  [Unknown]:  FR
Quel est le code pays à deux lettres pour cette unité ?
  [Unknown]:  FR
Est-ce CN=Hugues Fauconnier, OU=LIAFA, O=paris-diderot, L=PARIS, ST=FR, C=FR ?
  [non]:  oui

Entrez le mot de passe de la clé pour <signFiles>
   (appuyez sur Entrée s'il s'agit du mot de passe du fichier de clés) :
Ressaisissez le nouveau mot de passe :

(un alias signFiles pour Hugues Fauconnier a été ajouté dans le keystore
exemplestore)
```

# Signer le JAR

```
jarsigner -keystore exemplestore -signedjar sCompter.jar Compter.jar
signFiles
Enter Passphrase for keystore:
jar signed.

Warning:
The signer certificate will expire within six months.
No -tsa or -tsacert is provided and this jar is not timestamped. Without
a timestamp, users may not be able to validate this jar after the signer
certificate's expiration date (2015-06-30) or after any future
revocation date.
```

*le jar est signé par l'alias signFiles du keystrore exemplestore*

*pour exporter le certificat:*

```
hf$ keytool -export -keystore exemplestore -alias signFiles -file
Exemple.cer
Entrez le mot de passe du fichier de clés :
Certificat stocké dans le fichier <Exemple.cer>
```

*un certifcicat exportable dans Exemple.cer*

# Du côté du récepteur…

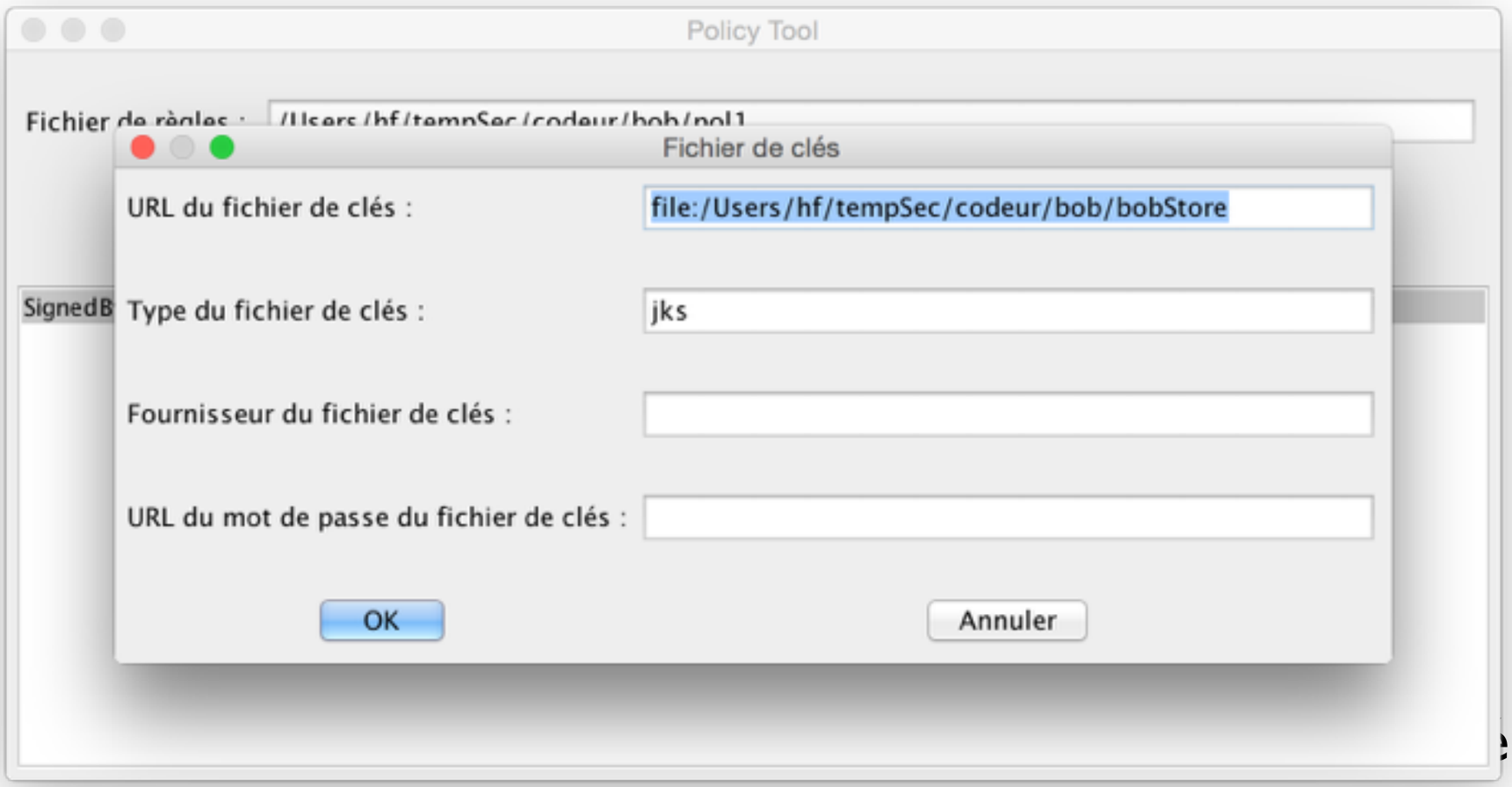
- Bob importe le certificat: (et le vérifie auprès de Hugues)

```
hf$ keytool -import -alias bob -file Exemple.cer -keystore bobStore
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Propriétaire : CN=Hugues Fauconnier, OU=LIAFA, O=paris-diderot, L=PARIS, ST=FR, C=FR
Emetteur : CN=Hugues Fauconnier, OU=LIAFA, O=paris-diderot, L=PARIS, ST=FR, C=FR
Numéro de série : 7136d79f
Valide du : Wed Apr 01 18:35:15 CEST 2015 au : Tue Jun 30 18:35:15 CEST 2015
Empreintes du certificat :
    MD5:  DD:2C:63:A0:8C:41:EC:8B:93:C8:3B:63:FA:30:C0:1D
    SHA1 : CC:D3:DE:04:63:F7:CE:AD:7D:3B:BC:2E:5D:C7:6D:B6:82:79:66:D6
    SHA256 : CD:3A:A9:01:AD:CE:35:C3:87:22:B0:30:AA:34:06:C9:CF:DA:EB:C0:F4:70:B4:1C:
13:39:1C:CE:A2:75:50:55
    Nom de l'algorithme de signature : SHA1withDSA
    Version : 3

Extensions :

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 14 29 A9 EF 27 B1 CF 18   35 C3 AC A2 E2 FD C6 98  .)..'...5.......
0010: D6 37 5C CE                                        .7\.
]
]

Faire confiance à ce certificat ? [non] :  oui
Certificat ajouté au fichier de clés
```

*dans le keystore de bob (bobStore) bob est un alias pour le nouveau certificat Exemple.cer*

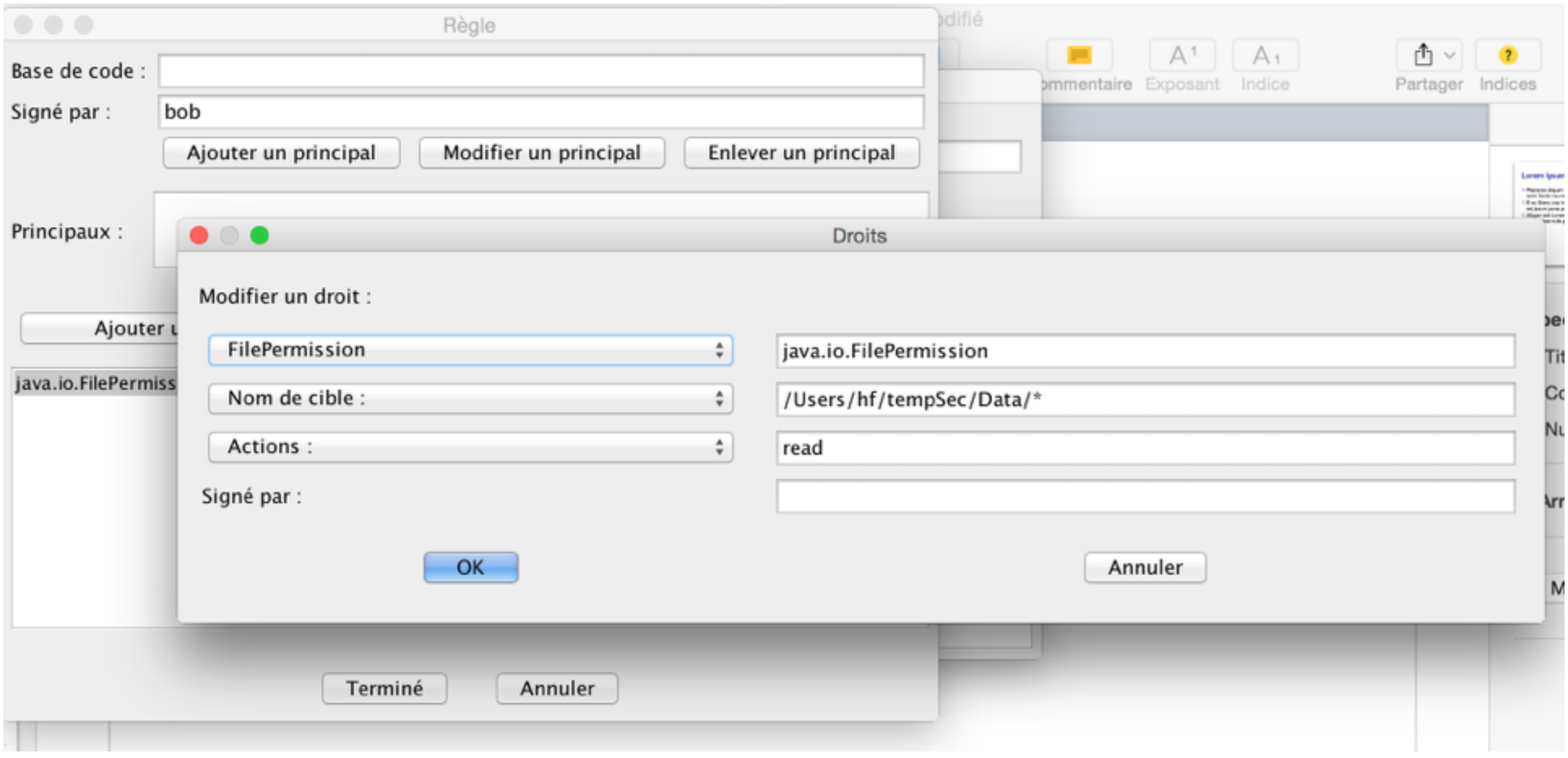
# Côté du récepteur: security manager

```
java -cp sCompter.jar Compter /Users/hf/tempSec/Data/Fichier
On a 491 caractères
: ok pas de security manager

java -Djava.security.manager -cp sCompter.jar Compter /Users/hf/Data/Fichier :
exception droit en lecture

policytool&

définir l’url du keystore:
```



# Côté du récepteur

- keytool: donner les droits en lecture



# Côté récepteur

sauvegarde de la politique de sécurité dans le fichier pol1:

```
/* AUTOMATICALLY GENERATED ON Wed Apr 01 19:19:19 CEST 2015*/
/* DO NOT EDIT */

keystore "file:/Users/hf/tempSec/codeur/bob/bobStore", "jks";

grant signedBy "bob" {
  permission java.io.FilePermission "/Users/hf/tempSec/Data/*", "read";
};
```

```
java -Djava.security.manager -Djava.security.policy=pol1 -
cp sCompter.jar Compter /Users/hf/tempSec/Data/Fichier
```

On a 491 caractères