Introduction à la sécurité – Cours 6
Cryptographie et Java (2)

Catalin Dima

Sommaires de messages et MAC

- Sommaires de message : classe `MessageDigest`.
  - Création d'un objet gérant la construction du digest : `MessageDigest md = MessageDigest.getInstance("MD5","BC")`
  - Insertion du message à chiffrer : `md.update(byte[] message)` (paramètre ≠ String !).
  - Calcul du digest : `byte[] resultat = md.digest()`.
    - Un `digest` reinitialise le `MessageDigest`.
  - Test d'égalité des digests : `boolean isEqual(byte[] digestA, byte[] digestB)`
- Message Authentication Codes : sommaires de messages cryptés avec des clés symmétriques
  - Création, intialisation et calcul du sommaire crypté comme pour les `Cipher`
  - `Mac getInstance("DESMac","BC");`
  - `void init(Key k);`
  - `void update(byte[] texte)` et `byte[] doFinal(byte[] texte)`.

Flux de `MessageDigest`

- Classes `DigestInputStream`
  - Association d'un `MessageDigest` à un `InputStream` permettant d'actualiser implicitement le digest en cours de communication.
  - Constructeur : `DigestInputStream(InputStream is, MessageDigest md)`
  - L'utilisation de l'objet digest peut être commutée avec `void on(boolean on)`.
- Si la fonctionnalité digest est "allumée", chaque appel à `in read(byte[] b, int off, int len)` ou `int read()` actualise le digest, comme si on ferait un `update`.
- Pour récupérer le digest à la fin de la transmission : `MessageDigest getMessageDigest()`.
- Similaire pour et `DigestOutputStream` :
  - Constructeur `DigestOutputStream(OutputStream is, MessageDigest md)`
  - Commutateur `void on(boolean on)` – par défaut "allumé".
  - `void write(byte[] b)` pour envoyer dans l'OutputStream les octets, et en même temps actualiser le digest.
  - `MessageDigest getMessageDigest()`.

Signatures digitales

Classe `Signature` et mêmes phases de génération que pour les Cipher :

1. Création d'un objet Signature : `static Signature getInstance(String algo, String prov)`.
2. Initialisation de l'objet :
   - Pour la vérification : `void final initVerify(PublicKey publicKey)`
   - Pour signer (on objet/message etc.) : `final void initSign(PrivateKey privateKey)`
3. Si objet créé pour signer,
   (a) Rajout d'octets à signer : `public final void update(byte[] b)`
   (b) Création de la signature : `public final byte[] sign()`
4. Si objet créé pour vérifier une signature :
   - Rajout d'octets à vérifier : `public final void update(byte[] b)`
   - Vérification : `public final boolean verify(byte[] signature)`

Confidentialité des objets

- Classe `SealedObject` : objet dont la confidentialité des *données* est protégé cryptographiquement.
- Création d'un objet scellé : `SealedObject(Serializable obj, Cipher c)`.
- Récupération de l'objet scellé, si on connaît la clé de chiffrement : `Object getObject(Key key, String provider)` (et aussi `Object getObject(Key key)`).
- Variante : si threads différents (avec des droits différents)
  - Un thread pourrait avoir accès à la clé, et donc créer un `Cipher cph`.
  - Mais c'est un autre thread qui demanderait l'utilisation de l'objet,
  - ... alors le deuxième pourrait récupérer le `cph` du premier et de-sceller : `Object getObject(Cipher c)`.

Authenticité des objets

- Classe `SignedObject` : object dont l'authenticité des *données* est garantie par les moyens d'un protocôle cryptographique.
- Création d'un objet signé : `SignedObject(Serializable o, PrivateKey pk, Signature engine)`
- Récupération de l'objet encapsulé dans un SignedObject : `Object getContent()`.
- Récupération de la signature qui accompagne un SignedObject : `byte[] getSignature()`.
- Vérification de la signature attachée à un SignedObject : `boolean verify(PublicKey pk, Signature s)`.

Clés de session

1. Création d'un gestionnaire de protocôle : `KeyAgreement kag = KeyAgreement.getInstance("Diffie-Hellman")`
2. Initialisation du gestionnaire avec une clé (créée auparavant selon le protocôle choisi) : `kag.init(cle)` (autres paramètres : paramètres d'algorithme, référence à un générateur sûr de nombres aléatoires).
3. Exécution d'une phase du protocôle : `kag.doPhase(cle, bool)` (renvoie une nouvelle clé, utile pour DH à trois parties).
   – Dans Diffie-Hellman à deux parties on l'apelle une seule fois, avec `bool = true`.
4. Génération de la clé de session : `SecretKey clesession = kag.generateSecret("DES")`.
5. Chiffrement/déchiffrement avec la clé de session – déjà vu.

Gestion des clés et des certificats

- Gestion des clés – processus important dans l'implémentation de l'authenticité.
- Deux buts :
  - Gérer/récupérer notre propre clé privée lorsque nous signons un document/message/code, etc.
  - Gérer/récupérer la clé publique d'une autre identité lorsque nous vérifions une signature digitale sur un document/message/code, etc.
- Chaque clé est associée à une identité.
- Chaque clé peut être accompagnée par un certificat
  - Preuve de la validité de l'association identité–clé.
  - Contient aussi l'identité de l'*autorité de certification*,
  - ... sa signature électronique,
  - ... et éventuellement d'autres informations (intervalle d'utilisation).
  - Format très répandu de certificats : *X509*.

Gestion des certificats

- Classe `Certificate` : classe abstraite, divers implémentations (dépendant des providers).
  - Vérification d'un certificat : `void verify(PublicKey pk)` et `void verify(PublicKey pk, String provider)`
  - Récupérer la clé publique dans le certificat donné : `PublicKey getPublicKey()`.
  - Pas de constructeur publique – seulement un constructeur *protégé* !
  - Extension de cette classe : `X509Certificate`.
    - Vérification de la validité : `checkValidity(Date)` (date quelconque) ou `void checkValidity()` (aujourd'hui).
    - Renvoie soit `CertificateExpiredException`, soit `CertificateNotYetValidException`.
- Importer un certificat dans un programme : classe `CertificateFactory`
  - Instancier un objet de type CertificateFactory : `static CertificateFactory getInstance(String type, String provider)`.
  - Générer un certificat lu sur un certain flux d'entrée : `Certificate generateCertificate(InputStream is)`.
  - Variante : générer une collection de certificats : `Collection generateCertificates(InputStream is)`.

Création de certificats

- Classes dépendentes de provider.
- Série de classes `X509V1CertificateGenerator,..,X509V3CertificateGenerator` disponibles en BouncyCastle.
- Éléments d'un certificat (qqs uns...) :
  - No. de série – `setSerialNumber(BigInteger)`.
  - Algorithme utilisé – `setSignatureAlgorithm(String)`.
  - Validité – `setNotBefore(Date)` et `setNotAfter(Date)`.
  - Émetteur (élément *essentiel*, permettant l'association de la clé de décryptage !) – `setIssuerDN(X500Principal)`
    - peut être créé en passant un `String`.
  - Sujet – `setSubject(X500Principal)` et sa clé – `setPublicKey(PublicKey)`.
- Création d'un `X500Principal` : constructeur `X500Principal(String)`.
  - En réalité, le paramètre doit satisfaire certaines contraintes.
  - Exemple : `"CN=Duke, OU=JavaSoft, O=Sun Microsystems, C=US"`.

Keystores

- **Keystore :** magasin de clés, peut contenir deux types d'entrées :
  - Des clés secretes (`KeyStore.SecretKeyEntry`) ou des paires *clé privée/clé publique* (`KeyStore.PrivateKeyEntry`) générées par nous-mêmes, éventuellement certifiées par une (ou plusieurs) Autorité(s) de Certification.
  - Des clés publiques d'autres agents avec leurs certificats d'authenticité (`KeyStore.TrustedCertificateEntry`), récuperés auprès des Autorités de Certification.
- Les entrées sont identifiées par des *alias* et protégés par mot de passe.
- Le keystore lui-même peut (et devrait !) être protégé par mot de passe.
- Classe `KeyStore` :
  - Instancier un objet KeyStore : `KeyStore getInstance(String type, String provider)`
  - Type par défaut : `"JKS"`, celui du provider `SunJCE`.
  - Peut être récupéré par `String getDefaultType()`
  - Charger un keystore en mémoire : `void load(InputStream is, char[] password)`
  - Envoyer le keystore dans un flux de sortie : `void store(OutputStream os, char[] password)`.

Keystores (2)

- Tester du type d'entrée : `boolean isKeyEntry(String alias)`, resp. `boolean isCertificateEntry(String alias)`.
- Récupérer la clé avec un alias : `Key getKey(String alias, char[] password)`
  - On ne peut pas récupérer l'entrée correspondant à un alias !
- Récupérer le certificat correspondant à un alias :`Certificate getCertificate(String alias)`
  - Et aussi `String getCertificateAlias(Certificate cert)`
- Créer une entrée : `void setKeyEntry(String alias, Key k, char[] password, Certificate chain[])`,
  - Et aussi `void setKeyEntry(String alias, byte key[], Certificate chain[])`
  - Et enfin `void setCertificateEntry(String alias, Certificate c)`
- Malheureusement, JKS ne sait pas gérer des entrées de type clé secrète...
  - Que Bouncy Castle peut gérer !

L'outil `keytool`

- Outil de gestion des clés et des certificats en mode console.
- Qqs options pour `keytool` :
  - `-genkey` : génération de nouvelle paire (options supplémentaires : `-keyalg, -sigalg, -keysize`).
  - `-import` et `-export` : importe/exporte un nouveau certificat dans le keystore (avec option supplémentaire `-file`).
  - `-selfcert` : création d'un certificat auto-signé (utilisé à des fins de test !).
  - `-list` et `-printcert`.
- Options supplémentaires pour chaque option :
  - `-keystore` : le nom de fichier représentant le magasin de clés.
  - `-storepass` : mot de passe pour protéger le magasin de clés (si non fourni, alors demandé en ligne de commande).
  - `-alias` : l'alias d'une entrée.
  - `-keypass` : mot de passe pour protéger la clé privée (pour les entrées de ce type !).

`KeyStore, keytool` et Bouncy Castle

- Bouncy Castle implémente deux types de keystore : `BKS` et `PKCS12`.
  - Différents de `JKS`, donc incompatibles...
- Pas de problème particulier d'utilisation avec `KeyStore` :
  - Déclarer le type de "magasin" lors de la création du `KeyStore`.
- Pour les utiliser avec `keytool` :
  - Installer le jar dans le répertoire `/lib/ext` du JRE (ou JDK).
  - Déclarer le provider dans la liste des providers existente dans `{java.home}/lib/security/java.security`
    (Ce qui vous dispensera de `addProvider` !)
  - Utiliser l'option `-storetype` de `keytool`.
- Les deux types nécessitent le passage en ligne de commande des mots de passe.