Introduction à la sécurité – Cours 5
Cryptographie et Java

Catalin Dima

Types de service de sécurité pour les communications

- *Confidentialité* : l'assurance que ses actions/données/communications ne soient pas examinées par des personnes/parties non-authorisées.
- *Authenticité* : l'assurance que l'accès/la conversation se passe avec une ou plusieurs personnes/parties légitimes.
- *Integrité* : l'assurance que ses propres données ne soient pas modifiées ou corrompues sans son accord, ou que ses communications ne soient pas pas modifiés après avoir été envoyées.
- *Disponibilité* : abilité d'utiliser une ressource à son gré, tant qu'on veut l'utiliser de manière correcte.
- *Non-repudiation* : l'impossibilité de nier ou désavouer une action/transaction/message dont on est la source.

Cryptographie : solution pour les trois premières types de services.

Criptographie : définitions

- **Cryptage** = transformation syntactique d'un text (*message clair*) de sorte qu'il devienne inintelligible par tout observateur.
  Le résultat = *text chiffré* ou *crypté*.
- **Décryptage** = opération inverse.
- **Cryptage à clés symmétriques** (ou **privées**, ou **classique**) :
  - Cryptage et décryptage à l'aide de la même clé,
  - ... ou la clé de decryptage peut être facilement calculable à partir de la clé de cryptage.
- **Cryptage à clé publique** (ou **asymmétrique**) :
  - Cryptage par une clé qui est disponible pour tout le monde (*clé publique*),
  - Décryptage par une clé privée, propriété du destinataire du message.
  - Essayer de decrypter à l'aide de la seule clé publique – complexité prohibitive.

Communication utilisant les clés symmétriques

1. Alice et Bob se mettent d'accord pour employer un système de chiffrement (ils peuvent le faire en public).
2. Alice et Bob se mettent d'accord pour utiliser une clé (il faut le faire en privé !)
3. Alice chiffre son texte à l'aide de sa clé et l'envoie à Bob.
4. Bob déchiffre le texte qu'il vient de recevoir.

Désavantages des systèmes à clés symmétriques :

– Cryptanalise un peu plus facile (que dans le cas des systèmes asymmétriques).

– Pour un réseau de n utilisateurs à qui on veut assurer la confidentialité mutuelle, on a besoin de $\frac{n(n+1)}{2}$ clés.

Fonctions à sens unique

- *Fonction à sens unique* : $f : A \longrightarrow B$ pour laquelle :
  - Il est "facile" de construire $f(a)$ pour tout a, mais
  - Étant donné $b \in B$, il est "difficile" de trouver a pour lequel $f(a) = b$.
  - "Facile" et "difficile" = *complexité algorithmique !*
- Aussi appelée *fonction de hashage*.
- Certaines fonctions de hashage se basent sur des clés cryptographiques (DES/CBC), d'autres non (MD4,MD5,SHA-1...).
- Utilisation dans l'**authentification** des messages :
  - On crée un *sommaire* (angl. *digest*) du message à transmettre.
  - On envoie le sommaire crypté.
  - Le recepteur pourra alors vérifier si le message qu'il a reçu est le bon message, en calculant de la même façon le sommaire sur le message reçu et en le comparant au sommaire reçu (qu'on a décripté en préalable).

Cryptage à clés publiques

- Alice crée une paire de clés dont la clé privée (notée K^{-1}) sera gardée soigneusement, tandis que la clé publique (notée K) sera mise à disposition de tout le monde.
- Bob peut alors chiffrer son message avec K, en étant sûr que c'est seulement Alice qui peut déchiffrer son message.
- En bref, cela revient à dire que $K^{-1}(K(M)) = M$ et $K^{-1}(L(M)) \neq M$ pour tout message M et toute clé $L \neq K$.
- Le système de cryptage d'Alice doit se baser sur une fonction à sens unique et à trappe.
- Il doit être très difficile (du point de vue de la puissance de calcul) de trouver K^{-1} à partir de K.
- Permet d'assuer la confidentialité du message sans nécessiter la confidentialité de la clé.
  - Bob est sûr que son message ne pourra être lu que par Alice !
- Gestion plus simple des clés, car deux clés par utilisateur (au lieu de n) si l'authenticité n'est pas requise !

Clés asymmètriques pour l'authentification

- La confidentialité n'est pas suffisante :
  - Alice reçoit un message "`Charlie est un traitre`".
  - ... qu'elle est la seule à pouvoir décrypter.
  - C'est bien pour l'expéditeur !
  - ... mais ça a l'air d'une anonyme...
  - Comment peut savoir Alice qui est la source du message ?
- Bob veut transmettre un message à Alice,
- ...et veut assurer Alice de l'**authenticité** du message qu'elle va recevoir.
- Alors il va chiffrer son message *avec sa clé* ***privée*** !
- Alice va pouvoir déchiffrer le message qu'elle vient de recevoir *avec la clé* ***publique*** *de Bob*.
- Cela revient à dire que $K(K^{-1}(M)) = M$ et $K(L^{-1}(M)) \neq M$ pour toute clé $L \neq K$ et pour tout message M.
- Ce qui est le cas de la totalité des protocôles cryptographiques.

Authentification de messages

Aspects de l'authentification :

- Chiffrer un document de taille très importante peut s'avérer trop coûteux,...
- En particulier quand le document ne doit pas être en soi-même un sécret partagé que par Alice et Bob (ex. chèques).
- L'idée serait alors d'associer au document une **signature digitale** : le chiffrement par la clé privée d'Alice d'une valeur connue par tous.
- Valeur connue par tous : *le résultat du hashage du document !*
- Cela assure
  1. D'une part l'association unique document – valeur chiffrée.
  2. D'autre part, l'assurance que toute modification du document par un tiers entraînera l'invalidation du document.
- Variante d'authentification : **MAC** (*Message Authentication Code*)
  - Similaire à la signature digitale, mais chiffrement symmétrique.

Rassembler les fonctionnalités de sécurité – schéma général

- Coté expéditeur :
  1. Calculer un sommaire (digest) du message clair.
  2. Chiffrer le digest avec sa propre clé privée (*authentification*).
  3. Rassembler le digest avec le message clair.
  4. Chiffrer le résultat avec la clé privée du destinataire (*confidentialité*).
- Coté destinataire : ordre inverse.
  1. Déchiffrer le texte reçu avec sa propre clé privée – obtention de la paire (digest chiffré, message clair)
  2. Déchiffrer le digest avec la clé publique de l'expéditeur – obtention du digest clair.
  3. Calculer le digest du message clair.
  4. Comparer le digest reçu avec le digest calculé – si différents, alors rejet de la communication.
- Prérequis :
  - Chaque participant connait la clé publique de l'autre.
  - Schéma intéressant pour des envois simples – un seul message.

Mise en oeuvre de la cryptographie à clé publique

- Deux types de génération :
  1. Génération asymmètrique et transportation – le cas RSA.
     - calcul d'une clé de session à partir de la clé publique du recepteur.
     - renvoi au recepteur de la clé de session dans un message chiffré avec sa clé publique.
  2. Génération "symmétrique" – le cas Diffie-Hellman.
     - calcul d'un "secret partagé" entre les deux participants, à partir de l'information detenue dans leurs clés publiques et privées.
     - génération d'une clé de session symmétrique à partir de ce secret partagé.
- Remplissage du message avec du texte généré aléatoirement = **padding**
  - Pour éviter que les attaquants puissent faire des *dictionnaires de messages cryptés* !
  - Ou comparer un message crypté avec d'autres.
  - Problème spécifique dans le cas des systèmes à clés publiques.

Java Cryptography Extension (JCE)

- Cadre pour le déploiement des mésures de sécurisation des données et/ou communications.
- Basé sur la possibilité d'utiliser différents *Provider*s pour les solutions cryptographiques.
  - Dû en principal aux raisons légales : limitations d'utilisation de la cryptographie dans certains pays, limitations dans l'*exportation* des solutions cryptographiques.
- Couvre un grand évantail de techniques cryptographiques :
  - Crypto symmétrique (par bloc ou en flux) et asymmétrique, "key agreement", crypto basée sur les mots de passe (**Password-Based Encryption**, PBE), MAC et signatures digitales.
- Le provider "de base", nommé `"SunJCE"`, implémente un certain nombre d'algorithmes "légers" :
  - Chiffrement symmétrique, signature, hashage, MAC, Diffie-Hellman.
  - ...mais pas de chiffrement asymmétrique : RSA seulement pour les signatures digitales, pas pour la confidentialité !
- Un provider "free" et implémentant suffisamment d'algorithmes (mais des solutions "lightweight"...) : *Bouncy Castle*, `"BC"`.

Chiffrement en JCE

- `Cipher` : classe encapsulant les fonctionnalités d'un chiffreur/déchiffreur cryptographique.
- Un objet de type `Cipher` encapsule une instance d'un *algorithme* de chiffrement/déchiffrement, avec ses paramètres spécifiques.
- Création d'un chiffreur : méthodes `getInstance` :
  - `static Cipher getInstance(String transf)` ou `static Cipher getInstance(String transf, Provider prov)`
  - `transf` = `"algorithm/mode/padding"`.
- Initialisation du chiffreur/déchiffreur : quatre modes (membres *statiques* de la classe `Cipher` !)
  1. `ENCRYPT_MODE` : pour utiliser l'instance comme chiffreur
  2. `DECRYPT_MODE` : instance = déchiffreur.
  3. `WRAP_MODE` et `UNWRAP_MODE` : pour "envelopper" ou "déballer" la clé (pas trop utilisés).

Chiffrement en JCE (2)

- Initialiser : donner le mode de fonctionnement et la clé de chiffrement
  - `void init(int opmode, Key key)`, où le `opmode` est une des constantes ci-dessus.
  - Sept autres variantes, permettant de donner aussi les *paramètres de l'algorithme* et un *générateur de nombres aléatoires* (sécurisé).
  - `Key` : référence encapsulant une clé pour l'algorithme.
  - Si l'algorithme nécessite des paramètres : paramètres implicits au moment du chiffrement
  - ...*mais* `InvalidAlgorithmParameterException` si paramètres non fournis au moment du déchiffrement.
- Chiffrer ou déchiffrer d'un seul coup : `byte[] doFinal(byte[] input)`
  - Texte : transformé en séquence de bits !
  - Trois autres méthodes `doFinal`.
- Chiffrer ou déchiffrer un texte au fur et à mésure qu'il est créé : `byte[] update(byte[] input)`
  - Méthode `int getOutputSize(int inputLength)` pour déterminer, avant un `update`, la taille du tableau de bits retourné par le `update`.
  - Quand on a fini d'insérer le texte dans le chiffreur, on doit toujours terminer la séquence d'`update` par un `doFinal()` !

Génération des clés

- `KeyPair` : conteneur d'une paire de clés pour chiffrement asymmétrique.
  - Méthodes `PrivateKey getPrivate()` et `PublicKey getPublic()` pour retrouver la paire de clés.
- `KeyPairGenerator` : génération des paires de clés pour les algorithmes symmétriques.
  - Classe "moteur" = dépendante du provider.
  - Création : `static KeyPairGenerator getInstance(String algorithm, String provider).`
  - Initialisation : `void initialize(int keysize, SecureRandom random).`
  - Éventuellement, initialisation avec des paramètres : `void initialize(AlgorithmParameterSpec params).`
  - Création d'une paire : `KeyPair generateKeyPair().`
- Alternative pour les algorithmes symmétriques : classe `KeyGenerator`.
  - Création : `static KeyGenerator getInstance(String algorithm, String provider)`
  - Initialisation : `void init(int keysize)` ou `void init(AlgorithmParameterSpec params)` (ou encore 3 variantes !).
  - Création d'une clé : `SecretKey generateKey().`

Portabilité des clés

- Deux types de clés : *opaques* et *transparentes*.
  - Clés opaques : accès limité, structure dépendante du *Provider*.
  - Clés transparentes : le matériel de construction de la clé est portable.
- Interface `Key`, pour représenter les clés (des deux types).
- Caractéristiques :
  - *Algorithme* de chiffrement pour lequel la clé est utilisable, renvoyé par `String getAlgorithm()`
  - *Forme codée*, retrouvable par `byte[] getEncoded()`.
  - *Format de codage*, retrouvable par `String getFormat()`.
- 12 sous-interfaces pour les différents types de clés
  - Implémentées dans les classes fournies par le *Provider*.
- Interface `KeySpec`, regroupant les spécifications des clés des différents algorithmes.
  - Sans méthode, car trop d'hétérogénéité entre les algorithmes...
- Variantes pour les algorithmes symmétriques : interfaces `SecretKey` et `SecretKeySpec`.

Portabilité des clés symmétriques

Classe `SecretKeyFactory` pour convertir entre clés opaques et transparentes :

- Définit de manière *abstraite* les caractéristiques de conversion.
- Classe "moteur" (angl. *engine class*).
  - L'implémentation est dépendante du provider !
- Création : `static SecretKeyFactory getInstance(String algorithm, String provider)`
- Conversion du mode opaque en mode transparent : `SecretKeySpec getKeySpec(Key key, Class keySpec)`
- Conversion du mode transparent en mode dépendant du provider : `SecretKey generateSecret(KeySpec keySpec)`

Portabilité des clés asymmétriques

Classe `KeyFactory` pour convertir entre clés opaques et transparentes.

- Création : `static KeyFactory getInstance(String algorithm, String provider)`
- Conversion du mode opaque en mode transparent : `KeySpec getKeySpec(Key key, Class keySpec)`
- Conversion inverse, cas des clés asymmétriques :
  - Il faut générer la clé privée et la clé publique séparément – pas de `generateKey` !
  - `PublicKey generatePublic(KeySpec keySpec)`
  - `PrivateKey generatePrivate(KeySpec keySpec)`
- Pas de `PublicKeySpec/PrivateKeySpec` génériques !
  - Classes spéciales de clés transparentes pour chaque algorithme : `RSAPrivateKeySpec/RSAPublicKeySpec`, `DHPrivateKeySpec/DHPublicKeySpec`.
  - E.g. : `RSAPrivateKeySpec(BigInteger modulus, BigInteger privateExponent)`.
- D'habitude, la transmission du matériel pour les clés symmétriques ne permet de reconstituer que les clés publiques !
  - Utilisation des certificats – à voir plus tard...

Exemples

- Classes `RSAPublickKeySpec` et `RSAPrivateKeySpec`.
- Constructeurs :
  - `RSAPrivateKeySpec(BigInteger modulus, BigInteger privateExponent)`.
  - `RSAPublicKeySpec(BigInteger modulus, BigInteger publicExponent)`.
- Récupération du matériel de clé en version transparente :
  - `BigInteger getModulus()`,
  - `BigInteger getPublicExponent()`, resp. `BigInteger getPrivateExponent()`.
- Classe `DESKeySpec` :
  - `DESKeySpec(byte[] key)` : n'utilise que les *8 premiers octets* de `key` !
  - `byte[] getKey()`.
  - `boolean isWeak(byte[] key, int offset)`.

Schéma PBE

- Parfois utile de chiffrer/déchiffrer à l'aide d'une clé = mot de passe
- Clé facile à mémorer, pas besoin de la sauvegarder quelque part...
- Mais longueur de clé sensible aux attaques *"password dictionnaries"*.
- Alors, les algorithmes de type PBE (PKCS#5) nécessitent le rajout du sel.
  - Valeur pseudo-aléatoire, (la même au chiffrement et au déchiffrement !) à combiner avec le mot de passe.
- Défi supplémentaire pour les attaquants : nombre d'itérations pour calculer la clé de chiffrement à partir du mot de passe = paramètre.
- PBE en JCE :
  1. Classe `PBEKeySpec` de constructeur `PBEKeySpec(char [] passwd, byte[] salt, int itcount)`.
     - Pas `String` : un String est immutable, un tableau de chars non !
  2. Exemple d'algorithme de chiffrement : `"PBEWithMD5andDES"` (du provider par défaut, *SunJCE*).
     - Il faut aussi donner des paramètres de chiffrement, ainsi que le sel !
  3. Classe `PBEParameterSpec`, constructeur `PBEParameterSpec(byte[] salt, int iterationCount)` ;
  4. Utilisé à l'initialisation du chiffreur ! `cipher.init(type,cle,params)`

Flux cryptés

- Encapsulation de flux d'entrée-sortie chiffrés par le moyen d'un objet `Cipher`.
- Classes `CipherInputStream` et `CipherOutputStream`.
  - Constructeur `CipherInputStream(InputStream fin, Cipher ciph)`, similaire pour le OutputStream.
- Méthodes similaires au `InputStream` :
  - `int read(byte b), int read(byte[] b)`
  - `int available()`
  - `void close()`
- Toutes les transmissions d'octets par le flux seront cryptées avec l'algorithme de chiffrement implémenté par l'instance de `Cipher` encapsulée.