SCIENCES SORBONNE UNIVERSITÉ

SRCS : Systèmes Répartis Client/Serveur

TME 5 : Implantation d'un canal de sécurité

Objectifs pédagogiques

- Utilisation d'objets liés à la sécurité réseau
- Manipulation de l'API JCA et JCE
- Implantation d'un canal de communication sécurisé

Prérequis

Pour mener à bien ce TME, il est nécessaire de connaître les notions vues en cours :

- principes et but d'un chiffrement à base de clés asymétriques
- principes et but d'un chiffrement à base de clés secrètes
- principe d'une fonction de hachage
- notion de signature numérique
- les concepts d'authentification, d'intégrité et de confidentialité et comment les implanter à l'aide d'outils de cryptologie.

Introduction

Dans ce TME le but est de programmer un canal permettant une communication bidirectionnelle entre deux entités distantes. Au fur et à mesure des exercices, nous allons enrichir les propriétés de sécurité sur ce canal. Pour implanter ces propriétés de sécurité, nous allons utiliser l'API JCA (Java Cryptograhpy Architecture) et JCE (Java Cryptography Extension) qui est une extension de JCA. Ces deux API complémentaires offrent des implantations d'outils cryptologiques comme par exemple le chiffrement et le déchiffrement, la génération de clés symétriques ou asymétriques ou bien encore la signature numérique. Il existe plusieurs fournisseurs de ces APIs qui implantent ces outils. Le Jdk possède une implantation par défaut (celle d'Oracle) et c'est cette implantation que nous utiliserons. Les interfaces des outils que nous allons utiliser se trouvent dans les packages `javax.crypto` et `java.security`. Vous pourrez trouver en annexe de ce sujet un résumé des principaux outils utiles à la réalisation de ce TME.

Principe des tests unitaires

Chaque test unitaire sur un canal déploie une mini application client-serveur. Le serveur offre un service de stockage de properties java qui se traduit par une table qui associe des clés (un nom de paramètre) à des valeurs (la valeur du paramètre) sous la forme de chaînes de caractères. Vous n'aurez pas à programmer cette application puisqu'elle est déjà fournie dans le package `srcs.securite.app`. Un test unitaire testant un canal va déployer un client et un serveur de cette application, ouvrir une socket de communication TCP puis déployer le canal à tester.

Exercice 1 – Programmation d'un canal bidirectionnel

Un canal de communication bidirectionnel permet à deux entités distantes de communiquer dans les deux sens : l'un peut envoyer ou recevoir des messages de l'autre et vice versa. Pour mettre

en œuvre ce canal de communication, chaque entité instancie une classe qui respecte l'interface `srcs.securite.Channel` (fournie dans les ressources). Un canal offre les méthodes suivantes :

- `void send(byte[] bytesArray)throws IOException;` : permet à une entité d'envoyer un message sérialisé à l'autre entité.
- `byte[] recv()throws IOException;` : permet à une entité de consommer un message provenant de l'autre entité. Cet appel est bloquant tant qu'aucun message n'a été reçu.
- `InetAddress getRemoteHost();` : renvoie l'adresse réseau de l'entité distante
- `int getRemotePort();` : renvoie le port de communication de l'entité distante
- `InetAddress getLocalHost();` : renvoie l'adresse réseau de l'entité locale
- `int getLocalPort();` : renvoie le port de communication de l'entité locale.

Question 1

Une première étape est de programmer une classe d'implantation qui puisse assurer le transfert réel de données entre les deux entités du canal. Dans le package `srcs.securite`, coder une classe `ChannelBasic` qui implante `Channel`. Cette classe offrira en plus des méthodes de `Channel` un constructeur qui prend une `Socket` de communication en paramètre.

Question 2

Par la suite nous aurons besoin de décorer cette classe pour y ajouter et combiner des propriétés de sécurité. Programmer la classe `ChannelDecorator` qui implante l'interface `Channel` et qui possède un attribut qui référence un `Channel` à décorer que l'on initialisera à la construction via un paramètre. Les implantations de méthodes de l'interface `Channel` seront une simple délégation sur l'attribut (sous Eclipse, cette manipulation peut être faite automatiquement en faisant *Click droit sur le code* → *Source* ← *Generate Delegate Methods ...*)

Question 3

Tester ces deux classes avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestChannelBasic`

Exercice 2 – Génération de clés, certification et gestion de mots de passe

Dans cet exercice nous allons programmer des outils annexes nécessaires à la sécurisation d'un canal.

Question 1

Dans une classe `srcs.securite.Util` programmer une méthode statique `KeyPair generateNewKeyPair (String algorithm, int sizekey)throws NoSuchAlgorithmException`. Elle permet de générer une nouvelle paire de clés asymétrique pour un algorithme de chiffrement et une taille de clé donnée.

Question 2

Tester cette classe avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestUtil`

Dans la suite nous allons programmer une classe qui modélise une autorité de certification de confiance qui délivre des certificats. Pour rappel, un certificat est un objet qui permet d'authentifier l'association d'une clé publique avec son propriétaire. Dans le cadre de ce TP, nous n'allons pas utiliser de vrais certificats, mais nous allons définir une classe qui modélise un certificat simplifié.

Question 3

Programmer une classe `Certif` qui modélise un certificat. Cette classe possède a minima :

- 4 attributs constants : une `String` représentant l'identifiant du propriétaire de la clé, une `PublicKey` représentant la clé publique à certifier, un tableau java d'octets qui représente la signature de l'autorité de certification et enfin une `String` indiquant le nom de l'algorithme de signature.
- un constructeur de visibilité package qui permet d'initialiser ces 4 attributs via ses paramètres (dans cet ordre).
- un getter sur l'identifiant
- un getter sur la clé publique
- un getter sur la signature de l'autorité
- une méthode `boolean verify(PublicKey publickeyauthority)throws GeneralSecurityException` qui vérifie si le certificat est authentique en vérifiant la signature grâce à la clé publique de l'autorité passée en paramètre (la même qui est censée avoir forgé le certificat).

Question 4

Dans le package `srcs.securite`, programmer une classe `CertificationAuthority` qui offre :

- un constructeur prenant un nom d'algorithme de chiffrement asymétrique (String), une taille de clé (int) et un nom d'algorithme de signature (String). Ce constructeur crée la paire de clés publique/privée de l'autorité.
- `PublicKey getPublicKey()` qui renvoie la clé publique de l'autorité
- `Certif getCertificate(String idenfifier)` qui renvoie le certificat associé à l'identifiant passé en paramètre. Si le certificat associé à cet identifiant n'existe pas, la méthode renvoie la référence nulle.
- `Certif declarePublicKey(String identifier, PublicKey pubk)throws GeneralSecurityException` qui forge et stocke un nouveau certificat associant un identifiant à sa clé publique. On interdit le fait qu'un identifiant existant puisse changer sa clé publique via cette méthode. Ainsi, si l'autorité a déjà une clé publique pour cet identifiant, une `GeneralSecurityException` sera jetée.

Question 5

Tester ces deux classes avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestCertification`

À présent, nous souhaitons créer un outil qui permet d'associer un identifiant d'utilisateur à un mot de passe. Pour des raisons de sécurité, les mots de passe ne sont jamais stockés en clair. En effet, on les stocke toujours sous la forme d'une empreinte calculée à partir d'une fonction de hachage sécurisée.

Question 6

Dans le package `srcs.securite`, programmer une classe `PasswordStore` qui offre :

- un constructeur prenant en paramètre le nom d'un algorithme de hachage
- une méthode `void storePassword(String user, String passwd)` qui stocke le mot de passe d'un utilisateur. Le mot de passe passé en paramètre est en clair. Le but de la méthode est donc de stocker un haché de ce mot de passe avec l'algorithme passé en paramètre du constructeur.
- une méthode `boolean checkPassword(String user, String passwd)` qui répond vrai si le mot de passe (en clair) passé en paramètre correspond bien au mot de passe associé à l'utilisateur donné, faux sinon.

Question 7

Tester cette classe avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestPasswordStore`

Exercice 3 – Authentification

Dans cet exercice, nous allons implanter un mécanisme d'authentification permettant aux deux entités d'être certaines qu'elles s'adressent bien à la bonne personne. L'authentification du client auprès du serveur est double : une première authentification par clé asymétrique et si cette authentification réussie, le serveur attend l'envoie d'un login avec un mot de passe (ceci se fait par exemple quand un utilisateur a pu établir une connexion sécurisée depuis une machine cliente et doit ensuite entrer son numéro de compte/login et son mot de passe). L'authentification du serveur auprès du client se fait par clé asymétrique. Ainsi, le protocole est le suivant :

- Les deux entités envoient leurs certificats de clé publique
- À la réception du certificat, les deux entités vérifient son authenticité. Si le certificat est détecté corrompu alors une `CertificateCorruptedException` est jetée (classe d'exception fournie dans les ressources) et on met fin à l'établissement de la relation.
- Le client authentifie le serveur grâce aux clés. Si l'authentification échoue, le client met fin à l'établissement de la relation et jette une `AuthenticationFailedException` (classe d'exception fournie dans les ressources)
- Sur le même principe le serveur authentifie le client et jette une `AuthenticationFailedException` en cas d'échec.
- Une fois la connexion authentifiée des deux côtés, le client envoie un login et un mot de passe pour pouvoir utiliser le service offert par le serveur.
- Le serveur reçoit le login et le mot de passe envoyés par le client et vérifie que la correspondance est correcte. Si l'authentification avec mot de passe échoue, une `AuthenticationFailedException` est jetée côté serveur.

Il faut bien sûr s'assurer que le mot de passe ne transite pas sur le réseau en clair. Pour simplifier, on supposera que le serveur et le client utilisent la même autorité de certification.

Question 1

Dans le package `srcs.securite`, programmer une classe `Authentication` qui offre

- deux constructeurs pouvant jeter des `IOException` et des `GeneralSecurityException` : un pour le côté serveur et un pour le côté client
  - ◇ sur le serveur, le constructeur prendra 5 arguments : un canal de communication (`Channel`), le certificat local du serveur (`Certif`), une paire de clés asymétriques locales du serveur (`KeyPair`), un `PassworStore` qui est supposé être composé des utilisateurs enregistrés, et enfin la clé publique de l'autorité de certification (`PublicKey`)
  - ◇ sur le client, le constructeur prendra 6 arguments : un canal de communication (`Channel`), le certificat local du client (`Certif`), une paire de clés asymétriques locales du client (`KeyPair`), un login (`String`), un mot de passe en clair `String` et enfin la clé publique de l'autorité de certification (`PublicKey`)

  Les deux constructeurs implantent le protocole décrit ci-dessus pour le client et pour le serveur. s
- trois getters : un pour le certificat local, un pour le certificat distant (une fois qu'il a été reconnu authentique) et un pour la paire de clés locales.

Une fois l'authentification établie des deux cotés, l'objet `Authentication` ainsi instanciée contient l'ensemble des clés nécessaires à une communication par chiffrement asymétrique.

Question 2

Tester cette classe avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestAuthentication`

Attention, ce scénario de test considère plusieurs cas d'attaque que vous devez gérer dans votre implantation.

Exercice 4 – Confidentialité

Maintenant que nous pouvons avoir un canal avec une connexion authentifiée, nous allons enrichir les propriétés de sécurité d'un canal en assurant la confidentialité. Pour rappel la confidentialité permet d'assurer que seules des entités bien définies soient autorisées à lire les données.

Pour établir un canal qui chiffre les messages envoyés, il est nécessaire que les deux entités puissent utiliser un mécanisme de chiffrement symétrique, car beaucoup moins coûteux que le chiffrement asymétrique.

Le principe est que les deux entités génèrent une clé symétrique chacune de leur côté puis l'envoie à l'autre entité (toujours en s'assurant que personne d'autre ne puisse la lire). Une fois la clé secrète reçue des deux côtés, il est nécessaire d'avoir un protocole commun qui assure que les deux entités choisiront la même clé parmi les deux clés existantes (celle générée localement et celle générée par l'entité distante).

Question 1

Dans le package `srcs.securite`, coder une classe `SecureChannelConfidentiality` qui étend `ChannelDecorator` et qui offre :

- un constructeur prenant 4 paramètres : le canal décoré (`Channel`), un descripteur d'authentification (`Authentication`), le nom d'un algorithme de chiffrement symétrique (`String`) et une taille de clé (`int`). Ce constructeur implante la négociation de la clé secrète entre les deux entités.
- un getter sur la clé secrète (une fois celle-ci élaborée par le constructeur)
- une redéfinition de la méthode `send` permettant de chiffrer le message passé en paramètre et de le retransmettre ensuite au canal décoré.
- une redéfinition de la méthode `recv` permettant de déchiffrer un message reçu depuis le canal décoré.

Question 2

Tester cette classe avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestSecureChannelConfidentiality`

Exercice 5 – Intégrité

De manière orthogonale, nous souhaitons maintenant avoir un décorateur de canal assurant l'intégrité des messages, c'est-à-dire assurer qu'un message n'a pas été altéré pendant son transfert [1].

Le but de ce décorateur sera donc de signer chaque message qui est envoyé et de vérifier chaque message reçu.

Question 1

Dans le package `srcs.securite`, coder une classe `SecureChannelIntegrity` qui étend `ChannelDecorator` et qui offre :

- un constructeur prenant trois arguments : le canal décoré (`Channel`), un descripteur d'authentification (`Authentication`) et le nom d'un algorithme de signature (`String`)
- une redéfinition de la méthode `send` qui signe chaque message passé en paramètre avant de l'envoyé sur le canal décoré
- une redéfinition de la méthode `recv` qui vérifie l'intégrité de chaque message reçu sur le canal décoré avant de le renvoyer. Si un message est détecté corrompu, une exception `CorruptedMessageException` (fournie dans les ressources) est jetée.

Question 2

Tester cette classe avec la classe Junit suivante fournie dans les ressources de TP :

`srcs.securite.test.TestSecureChannelIntegrity`

1. On pourra noter qu'il sera possible ensuite de combiner les deux décorateurs (confidentialité et intégrité) et d'avoir ainsi un canal qui assure les deux propriétés, mais ceci ne sera pas traité dans ce tme

Annexe : Outils JCA et JCE pouvant être utiles à ce TME

Vous pourrez trouver des informations complémentaires à cette annexe sur les liens suivants :
- `https://www.jmdoudoux.fr/java/dej/chap-jca.htm`
- `https://www.jmdoudoux.fr/java/dej/chap-jce.htm`

Gestion de clé de chiffrement

Toute clé de chiffrement respecte le type `java.security.Key` qui une interface et qui étend `Serializable`. Cette interface offre trois méthodes dont deux qui pourraient servir ici : `String getAlgorithm()` qui renvoie le nom de l'algorithme associé (ex : RSA, DSA, AES, DES ...), `byte[] getEncoded` qui renvoie une version binaire de la clé.

Clés asymétriques

Une clé asymétrique peut être soit privée (interface `java.security.PrivateKey` qui étend `Key`) ou soit publique (interface `java.security.PublicKey` qui étend `Key`). Elles vont toujours par paire.

Pour générer une paire de clés (classe `java.security.KeyPair`) associant une clé publique et une clé privée, il est nécessaire de passer par un générateur de paire de clé (classe `java.security.KeyPairGenerat`

```
1 KeyPairGenerator kpgen =KeyPairGenerator.getInstance("nom d'algo chiffrement
      asymetrique");
2 kpgen.initialize(taillecle);
3 KeyPair kp = kpgen.generateKeyPair();
4 PublicKey pub = kp.getPublic();
5 PrivateKey priv = kp.getPrivate();
```

Clés secrètes

Une clé secrète `javax.crypto.SecretKey` étend l'interface `Key`. De la même manière une clé secrète peut être générée à partir d'un `javax.crypto.KeyGenerator`.

```
1 KeyGenerator keyGen = KeyGenerator.getInstance("nom d'algo chiffrement symetrique");
2 keyGen.init(taillecle);
3 SecretKey key = keyGen.generateKey();
```

Chiffrer et déchiffrer des données

Pour chiffrer ou déchiffrer des données, il est nécessaire d'utiliser une instance de `javax.crypto.Cipher`.

```
1 byte[] mess_part1= ...
2 byte[] mess_part2= ...
3
4 Cipher cipher = Cipher.getInstance("algo de chiffrement");//instanciation
5 cipher.init(mode,cle);//initialisation obligatoire
6 //le mode peut être soit Cipher.ENCRYPT pour chiffrer ou bien Cipher.DECRUPYT pour dé
      chiffrer
```

```
7 cipher.update(mess_part1);//bufferisation de la première partie des données àchiffrer
      ou dechiffrer
8 cipher.update(mess_part2);//bufferisation de la deuxième partie des données àchiffrer
      ou dechiffrer
9 byte[] res = cipher.doFinal();//on procède au chiffrement ou déchiffrement
```

Si les données à chiffrer se résument à un seul tableau d'octets `mess` il est également possible d'appeler directement `doFinal(mess)`.

Il est à noter qu'il existe des outils plus haut niveau comme :

- `javax.crypto.SealedObject` qui permet de chiffrer/déchiffrer un objet java
- `javax.crypto.CipherInputStream` qui permet de chiffrer/déchiffrer un flux d'octets entrant
- `javax.crypto.CipherOutputStream` qui permet de chiffrer/déchiffrer un flux d'octets sortant

Important : La taille des données pour un chiffrement asymétrique ne peut pas dépasser une certaine limite. Cette limite dépend de la taille de la clé au moment de la génération de la paire (exemple : pour une taille de 2048 bits, le nombre maximal d'octets d'un message est de 245 octets). En effet, le chiffrement asymétrique n'est pas adapté pour chiffrer de gros messages dus à son coût.

Fonctions de hachage

Pour hacher des données il faut utiliser la classe `java.security.MessageDigest`. Ceci fonctionne sur le même principe que le `Cipher`

```
1 byte[] data_part1=...
2 byte[] data_part2=...
3
4 MessageDigest md = MessageDigest.getInstance("algo de hachage");
5 md.update(data_part1);
6 md.update(data_part2);
7 byte[] digest = md.digest();
```

Signature numérique

Pour signer des données, on utilise la classe `java.security.Signature`.

Pour produire la signature :

```
1  PrivateKey privkey = ...
2
3  byte[] data_part1=...
4  byte[] data_part2=...
5
6  Signature signature = Signature.getInstance("algorithme de signature");
7  //NB: l'algo de signature est une combinaison d'un algorithme de chiffrement asymé
       trique et d'un algorithme de hachage
8  signature.initSign(privkey);
9  signature.update(data_part1);
10 signature.update(data_part2);
11 byte[] res = signature.sign();
```

Pour vérifier une signature

```
1 PublicKey pubkey=... //la clé publique associée àla clé privée qui a généré la
      signature
2 byte[] data=...
3 byte[] sign =... // production d'un sign() avec une clé privé
4 Signature signature = Signature.getInstance("algorithme de signature");
5 signature.initVerify(pubkey);
6 signature.update(data);
7 boolean res = signature.verify(sign);//vrai si la signature correspond au message
```

Exceptions liées à la sécurité

- `java.security.GeneralSecurityException` : exception mère à toute exception relative à la sécurité
- `java.security.NoSuchAlgorithmException` : exception levée lors de l'instanciation d'un outil cryptologique qui se base sur un algorithme, mais que celui si n'est pas défini ou que le nom est erroné.
- `java.security.InvalidKeyException` : exception levée lorsque qu'une clé est invalide (mauvais encodage, longueur erronée ...)
- `java.security.BadPaddingException`
- `java.security.SignatureException`