15. Expliquer le principe du chiffrement symétrique de type DES. Qu'apportent les notions d'EBC, CBC, padding ? Un tel chiffrement est-il réellement utilisé pour crypter l'ensemble d'une communication réseau sécurisée ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ? Expliquer les différences apportées par le chiffrement symétrique Rijndael/AES en CBC.

Un algorithme symétrique est celui auquel on pense immédiatement dans un contexte de cryptage. Il utilise la même clé pour chiffrer et déchiffrer un message. Bien sûr, le principal danger est que la clé soit interceptée, car tout qui la possède est capable de déchiffrer le message crypté au moyen de cette clé. Comme cette clé doit donc rester secrète, on parle encore d'algorithme à clé secrète.

Exemple :

Message : hello
Clé secrète : sydney
Type de chiffrement : XOR sur le code ASCII de chaque lettre

Message clair	« hello »	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
Clé	« sydney »	0111 0011	0110 1001	0110 0100	0110 1110	0110 0101
XOR						
Message crypté	*****	0001 1011	0000 1100	0000 1000	0000 0010	0000 1010
Clé	« sydney »	0111 0011	0110 1001	0110 0100	0110 1110	0110 0101
XOR						
Message obtenu	« hello »	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111

DES (Data Encryption Standard)

C'est le standard du gouvernement américain depuis 1977. Il utilise par défaut des blocs de 64 bits et une clé secrète de 56 bits. La clé est en fait de 64 bits mais 8 bits servent au contrôle de parité vérifiant l'intégrité de la clé. Les bits de la clé représentent en fait 16 sous-clés de 4 bits. Il a été certifié par le NIST (National Institute of Standards and Technology) jusqu'en 1993.

Les notions de EBC, CBC, padding

Ce sont des modes de chiffrement de blocs. Ils transforment un bloc de données claires d'une taille préalablement fixée en un bloc de données chiffrées de même taille. Pour chiffrer un message, il faut donc :

- découper celui-ci en blocs de taille acceptée par l'algorithme utilisé
- chiffrer chaque bloc : ceci se fait en utilisant un mode chaînage qui peut être l'un des modes de chiffrement courants :

ECB (Electronic Code Book) : un bloc de texte clair se chiffre en un bloc de texte chiffré, indépendamment des autres blocs. Il faut remarquer que deux blocs identiques produisent les mêmes blocs chiffrés et qu'il y a peu de protection sur l'intégrité du message, puisque l'indépendance des blocs chiffrés ne permet pas de détecter des permutations, duplications ou suppressions de blocs.
CBC (Cipher Bloc Chaining) : chaque bloc de texte clair est combiné par un XOR avec le bloc de texte précédent (un "vecteur d'initialisation" fournit le bloc chiffré pour le premier bloc clair). Cette fois, deux blocs identiques on peu de chance de produire les mêmes blocs chiffrés et il y a protection sur l'intégrité du message, puisque des permutations, duplications ou suppressions de blocs auront des implications sur les blocs résultants.
Evidemment, le message à coder sera le plus souvent d'une taille non multiple de la taille du bloc type de traitement. Il faudra donc compléter le message avec des caractères de remplissage (**padding**). Le padding le plus connu est celui de la recommandation PKCS#5 (algorithme symétrique de type DES) ou PKCS#7 (Public Key Cryptography Standards, algorithme asymétrique RSA). Le principe est des plus simples : on remplit les bytes non occupés par le nombre de ces bytes non occupés.
On peut encore épingler que, dans un chiffrement de bloc, le nombre de blocs chiffrés est le même que celui de blocs clairs, c'est un peu la faiblesse du système.

Un tel chiffrement est-il réellement utilisé pour crypter l'ensemble d'une communication réseau sécurisée ?

Un tel chiffrement n'est pas utilisé seul dans une communication réseau sécurisée. En fait, la clé symétrique (secrète) est appelée la clé de session. Elle permet de chiffrer et déchiffrer un message. Pour que la communication se fasse, il faut que les 2 parties aient la même clé de session. L'expéditeur du message doit donc d'abord envoyer la clé de session grâce à l'utilisation de paire de clé publique/privée. Un autre moyen de s'assurer que le destinataire a la bonne clé de session est d'utiliser l'algorithme de Diffie-Helleman qui se base sur des nombres premiers échangés entre l'expéditeur et le destinataire pour créer une clé de session identique des deux côtés.

Comment procède-t-on en programmation Java ?

Un générateur de clé :

Il nous faudra évidemment pour débuter nous procurer une clé de cryptage. Ce sera le travail d'un générateur de clé, objet instanciant une classe de type KeyGenerator. Nous utiliserons pour ce faire une méthode factory :

- public static KeyGenerator getInstance(String algorithm,String provider)

algo -> "DES"

ou

- public static KeyGenerator getInstance(String algorithm)

La génération d'une clé

On initialise le générateur de clé au moyen de (qui produit une valeur de départ totalement imprévisible) :

- public void initialize (SecureRandom random)

Il suffira donc de programmer :

- CleGen.initialize(new SecureRandom())

SecretKey est un interface qui dérive de l'interface Key et qui n'apporte rien de plus que l'algorithme qui lui correspond, sa forme codée utilisable en dehors de la machine virtuelle et le nom de ce format d'encodage. Nous obtenons la clé par :

- SecretKey Cle = CleGen.generateKey() ;

Obtenir un chiffrement

Il nous faut un objet « outil » dont le rôle est de réaliser le chiffrement d'un message selon l'algorithme DES que nous avons choisi. La classe Cipher représente un tel chiffrement. La factory à utiliser pour obtenir une instance de cette classe :

- public static Cipher getInstance(String algorithm,String provider)

On précisera donc dans une chaîne le nom de l'algorithme, le mode de chiffrement et le type de padding. (exemple « DES/ECB/PKCS5Padding »).

Le cryptage du message

Nous initialisons d'abord l'objet chiffrement avec la clé obtenue auparavant :

- public final void init(int opmode,Key key) throws InvalidKeyException

Le premier paramètre prenant l'une des valeurs :

- public static final int ENCRYPT_MODE = 1
- public static final int DECRYPT_MODE = 2

Il n'y a plus qu'à réaliser maintenant le cryptage sur notre tableau de bytes :

- public final byte [] doFinal(byte[] in) throws IllegalBlockSizeException

Expliquer les différences apportées par chiffrement symétrique Rijndael

A la différence de l'algorithme DES qui utilise des blocs de 64 bits et une clé secrète de 56 bits, l'algorithme de Rijndael utilise des clés de 128,196 ou 256 bits.

DES utilise un système de bloc indépendant les uns des autres (EBC) tandis que Rijndael (AES) utilise un système de blocs chaînés (CBC).Chaque bloc de texte clair est donc combiné avec le bloc de texte chiffré précédent, ce qui implique l'existence d'un vecteur d'initialisation (IV) afin de permettre le cryptage du premier bloc.

C'est cet IV qui marquera la seule différence notable dans la programmation. Il nous faudra donc :

- Produire un second nombre aléatoire qui servira de vecteur d'initialisation; cela peut se faire au moyen de la méthode de SecureRandom :

public void nextBytes (byte[] bytes)

la dimension du paramètre déterminant le nombre de bytes à produire;

- spécifier l'existence de ce vecteur d'initialisation lors de l'initialisation de l'objet de chiffrement en utilisant cette fois la méthode :

public final void init (int opmode, Key key, AlgorithmParameterSpec params)
    throws InvalidKeyException, InvalidAlgorithmParameterException

La classe IvParameterSpec du package javax.crypto.spec implémente l'interface AlgorithmParameterSpec, qui n'est qu'un prête-nom puisqu'il s'agit d'un interface vide. La classe proprement dite a pour constructeur :

public IvParameterSpec(byte[] iv)

et possède la méthode :

public byte[] getIV()

16. Expliquer le principe du chiffrement asymétrique selon RSA (utiliser les notations K, PrK, PK, H, S, …). En particulier, à quoi ressemblent les clés publiques et privées de cet algorithme ? Un tel chiffrement est-il réellement utilisé pour crypter l'intégralité d'une communication réseau sécurisée ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ?

Principe général :

Le chiffrement symétrique, à une seule clé secrète, est sans doute le plus intuitif, mais il souffre d'une faiblesse de taille : la clé doit absolument rester secrète, c'est-à-dire connue de l'émetteur et du récepteur. Si la clé tombe dans des mains étrangères, tous les messages pourront être décryptés sans aucune difficulté. C'est pour cela que l'on a introduit la notion de chiffrement asymétrique qui utilise une paire de clé : une clé dite publique et une autre privée.

Une clé publique : (PK)

Cette clé peut être connue de tous. La seule contrainte est qu'elle soit accompagnée d'un certificat provenant d'une autorité en laquelle l'émetteur et le récepteur ont confiance. Cette clé sert à crypter les informations qui transiteront sur le réseau.

Une clé privée : (PrK)

C'est une clé qui va servir à décrypter le message reçu crypté avec la clé publique et que l'utilisateur peut générer sans devoir la faire circuler sur le réseau. Il est donc le seul à posséder cette clé : c'est ici qu'apparaît le caractère sécurisé.

RSA :

Un des chiffrements asymétriques les plus connus est RSA (Rivest Shamir Adleman). Il assure donc la confidentialité (via le chiffrement) et l'authentification (via la signature électronique). Le principe de RSA est basé sur la difficulté de factoriser les grands nombres, lesquels, en plus, sont premiers : modularité exponentielle. Cet actuellement l'algorithme le plus utilisé dans le monde et il est considéré comme efficace avec des clés 1024 bits générées à l'aide des factorisations des nombres premiers.

Algorithme :

Formule de cryptage avec la clé publique :

<caractère chiffré> = (<caractère a chiffrer>^{e}) % n

Et pour le décryptage avec la clé privée :

<caractère dechiffrer> = (<caractère chiffrer>^{d}) % n

Clé publique PK:

On choisit aléatoirement deux nombres premiers p et q relativement grands (ex : p=61 et q=53).
Alors, n (le module) est simplement donné par leur produit :

n = p*q=61*53=3233

C'est ici que se situe la difficulté de celui qui voudrait percer le code : il faut tenter de factorisé le module. L'exposant public (e) est choisi comme étant un nombre entier dans [3,n-1] (donc inferieur a n) et premier avec z=(p-1)*(q-1)=3120 ; comme 3120 =2*2*2*2*3*5*13, tout nombre non multiple de 2,3,5 et 13 peut convenir ; e=17 est donc un candidat valable (l'algorithme ne réclame pas forcément un nombre premier – 49 eût donc put convenir) ; en pratique, on prend e=3 ou encore 65537 si n le permet.

Clé privée PrK:

Le module est le même que pour la clé publique.
L'exposant privé d est un nombre tel que (e*d-1) soit divisible par z (autrement dit, e*d = 1 % z) ; donc d=2753 peut convenir, puisque 17*2753-1=46800 et est divisible par 3120(résultat=15).
En pratique, la taille des clés valides ne sont pas quelconques: elles doivent être comprise dans l'intervalle [384,16384] et, de plus, être multiple de 8 (taille est donc 384+i*8, i entier).
Seule contrainte, la longueur (exprimée en bytes) du message à crypter n'est pas quelconque non plus : en faite, il faut que :
longueur du message <= (longueur du % n)-2 * (longueur du digest-2)

Le digest dont il est question étant ici inexistant, mais apparaissant dans l'algorithme de signature associé. Si cette condition n'est pas satisfaite, on obtiendra un message « message too long » et il faudra découper le message original en morceaux plus petits.

Un tel chiffrement est-il réellement utilisé pour crypter l'intégralité d'une communication réseau sécurisée ?

Non, le chiffrement asymétrique tel RSA demande du temps et des ressources nécessaires très (trop) importants pour crypter l'ensemble d'une communication. En pratique, on utilise une approche hybride, c'est-à-dire une combinaison de chiffrement asymétrique et symétrique : le système de la clé de session qui sera utilisé lors de l'initialisation d'une connexion. Ce système, notamment utilisé avec SSL, permet de nécessiter moins de ressource et de temps machine.

Le mécanisme de base est le suivant :

1. L'expéditeur fabrique, d'une manière ou d'une autre, une clé secrète (K). Cette clé est encore appelé clé de session.
2. Il code le message avec cette clé secrète (chiffrement asymétrique).
3. Il code la clé secrète au moyen de la clé publique (PK) du destinataire (chiffrement asymétrique).
4. Il envoie le message crypté symétriquement et la clé secrète cryptée asymétriquement.
5. Le destinataire retrouve la clé secrète en la décryptant au moyen de sa clé privée (PrK).
6. Il utilise la clé secrète obtenue pour décrypter le message.

Programmation Java :

Comme dit précédemment, nous avons besoin d'un couple de clés. Ce couple peut être obtenu via la méthode factory de la classe KeyPairGenerator présente le package java.security :

public static KeyPairGenerator getInstance(String algo) throws NoSuchAlgorithmException ou

public static KeyPairGenerator getInstance(String algo, String provider) throws NoSuchAlgorithmException

Il subsiste pour cette méthode plusieurs constructeurs: un avec simplement l'algorithme (RSA) et un deuxième qui inclut en plus le provider (Cryptix/Bouncy Castle). Une fois l'objet désiré obtenu, on va l'initialiser via sa méthode:

public void initialize(int tailleCle, java.security.SecureRandom source) ;

où les paramètres correspondent à la taille de la clé (512/1024) et à un nombre aléatoire pour la génération des clés. On peut dès lors récupérer notre couple de clés dans un objet KeyPair via la méthode :

public java.security.KeyPair generateKeyPair()

Cet objet KeyPair n'est rien d'autre que la combinaison des clés privée et publique que l'on obtient via les méthodes:

public PrivateKey getPrivate() et public PublicKey getPublic()

où PublicKey et PrivateKey dérivent de l'interface Key.

Les clés obtenues, il nous reste à obtenir un chiffrement, ce que nous allons faire via la classe Cipher (javax.crypto) et ce via la méthode factory:

public static Cipher getInstance(String algo, String provider) ;

où les paramètres sont l'algorithme et le fournisseur. On va alors initialiser le chiffrement via la méthode :

public final void init(int opmode, Key key) throws InvalidKeyException

où le premier paramètre indique si l'on crypte (public static final int ENCRYPT_MODE = 1) ou décrypte (public static final int DECRYPT_MODE = 2). Ensuite, il ne reste plus qu'à effectuer le cryptage et récupérer la message crypté (qui devra se présenter sous la forme de bytes) via la méthode :

public final byte[] doFinal(byte[] message) throws IllegalBlockSizeException

Note: nous pourrions également travailler avec un objet BaseRSAKeyPairGenerator dérivant de KeyPairGenerator et se trouvant dans le package cryptix.provider.rsa.
Ce qui donne pour un extrait de code :

```
KeyPairGenerator MesCles=KeyPairGenerator.getInstace("RSA");
//Pas de provider dans ce cas ci, on laisse le soin à java.security
//de choisir parmis sa liste de providers.
MesCles.initialize(512,new SecureRandom());
KeyPair DeuxCles = MesCles.generateKeyPair();
PublicKey Publique = DeuxCles.getPublic();
PrivateKey Privee = DeuxCles.getPrivate();
//On a obtenu notre clé publique et privée
//Il nous faut dès lors un chiffrement
Cipher Chiffrement = Cipher.getInstance("RSA/ECB/PKCS#1","CryptixCrypto");

//Pour le cryptage:
Chiffrement.init(Cipher.ENCRYPT_MODE,Publique); //Clé publique

//Pour le décryptage:
Chiffrement.init(Cipher.DECRYPT_MODE,Privee); //Clé privée

String Message= "Crypte moi !";
byte[] Resultat = Chiffrement.doFinal(Message.getBytes());
```

17. Qu'est-ce qu'une clé de session ? Expliquer les deux mécanismes classiques de partage de ce type clé, l'un par transmission confidentielle et l'autre par génération simultanée (utiliser les notations K, PrK, PK, H, S, …).

Le système de la clé de session est une combinaison de chiffrement asymétrique et symétrique. Ce système permet aux deux intervenants d'obtenir une clé secrète avec laquelle ils vont pouvoir crypter leurs messages. Cela évite la charge de travail demandé dans le cas d'un cryptage totalement réalisé en asymétrique. Le mécanisme de base est le suivant :

L'expéditeur qui désire envoyer un message :

- fabrique une clé secrète qui sera appelée la clé de session.
- code le message avec cette clé (chiffrement symétrique).
- code la clé secrète au moyen de la clé publique du destinataire(chiffrement asymétrique).
- Envoi le message crypté (symétriquement) et la clé secrète cryptée(asymétriquement).

Le destinataire :

- Retrouve la clé en la décryptant avec sa clé privée,
- Utilise la clé secrète obtenue pour décrypter le message.

Une variante, qui évite de transférer la clé de session sur le réseau, consiste à permettre aux deux parties de générer localement la même clé de session sur base de renseignements échangés par le réseau. C'est ce que permet l'algorithme de Diffier-Hellman qui se base sur un échange de nombres premiers et aléatoires. Le danger est cependant qu'un intrus peut se glisser dans cette conversation préalable et se faire passer pour l'expéditeur quand il converse avec le destinataire et vice et versa. (man in the middle).
Le mécanisme est le suivant :

1. A et B se mette d'accord sur deux nombres, l'un étant inférieur à l'autre.
2. A et B choisissent un nombre aléatoire qu'ils gardent secrètement.
3. Un calcul est réalisé et le résultat donne la clé publique qu'il s'échange l'un l'autre.
4. A partir de la clé publique de l'autre, A et B construise une clé secrète qui sera identique.

Ce mécanisme utilise l'arithmétique modulaire qui offre des fonctions non réversibles telles que x%n. L'astuce ici est l'utilisation du modulo. Un hacker ne sachant pas calculer dans un délai raisonnable la même clé puisque il ne possède le nombre aléatoire que chaque partie à conservé secret.

Avantage d'une fonction modulaire :
a) d'une part, il permet d'associer à un nombre de valeur quelconque un nombre limité à une fourchette; par exemple, modulo 7 ne donnera jamais qu'un résultat compris entre 0 et 6.

b) d'autre part, l'opération modulo permet de simuler des comportements erratiques, aléatoires; ainsi, si nous considérons la fonction puissance de base 3 en arithmétique classique et en arithmétique modulaire 7 (par exemple), on constate que :

L'algorithme :

Voici comment Alice et Bob vont procéder pour arriver à partager une clé sans la faire passer sur le réseau.

1. Ils se mettent d'accord publiquement sur deux nombres n et p, n étant inférieur à p.
   Exemple : p=11 et n=7

   Il s'agit en fait des deux paramètres d'une fonction puissance ne arithmétique modulaire :

   $n^x\%p \rightarrow 7^x\%11$

2. Bob choisit un nombre aléatoire qu'il gardera SECRET.
   Exemple : A=3

   Alice choisit ausssi un nombre aléatoire qu'elle gardera SECRET.
   Exemple : B=6
3. Bob calcul $\alpha = 7^A\%11 = 7^3\%11 = 343\%11 = 2$

   2 Sera sa clé publique (PK) et Bob l'envoie à Alice.

   Alice calcul $\beta = 7^B\%11 = 7^6\%11 = 117649\%11 = 4$

   4 Sera sa clé publique (PK) et Alice l'envoie à Bob.

   Les deux correspondants disposent donc à présent chacun des deux clés publiques.

4. Bob utilise la clé publique (PK_{Alice}) d'Alice pour construire pour construire une clé secrète (K_{Bob}) selon

   $K_{Bob} = PK_{Alice}{}^A\%11 = 4^3\%11 = 64\%11 = 9$

De manière similaire, Alice utilise la clé publique (PK_{Bob}) de Bob pour construire une clé secrète (K_{Alice}) selon

$K_{Alice} = PK_{Bob}{}^B\%11 = 2^6\%11 = 64\%11 = 9$

Miracle, ils ont fabriqué la même clé -> 9. Ils ne leur reste plus qu'à l'utiliser dans un cryptage symétrique.

Mais comment peut-on être sûr qu'un hacker ne peut pas, lui aussi, construire la même clé avec les élements public échangés ?

Parce qu'il faut connaître ou A ou B, qui sont restés secrets et quil est très difficile (impossible dans un temps raisonnalbe) de deviner A et/ou B à parte de PK_{Alice} et/ou PK_{Bob} car la fonction % est une focntion non réversible.

(Attention voir prise de notes)

18. Expliquer comment les notions d'interfaces et de méthodes factory sont utilisées en programmation Java pour les contextes de la cryptographie. Comment une méthode comme getInstance() fonctionne-t-elle (en particulier, comment trouve-t-elle les classes providers et comment peut-elle les instancier) ?

Rappel :
Méthodes factory: méthodes statiques capables de fournir l'instance d'un objet en fonction des caractéristiques précisées comme paramètre et ce sans spécifier le moindre nom de classe.

En effet, les méthodes factory permettent d'obtenir une instance d'une classe dont on ne connaît pas le nom mais dont on sait qu'elle implémente une interface donnée qui nous sera utile. Cela permet d'obtenir une instance de classe sans passer par un constructeur, et donc d'obtenir ainsi un objet qui reste une classe abstraite, qui implémente une interface. Le choix de la classe utilisée se fait donc au moment de l'exécution. En procédant de la sorte, on peut donc programmer d'une manière très générale indépendamment de la nature exacte de la classe réellement instanciée.

Contexte cryptographique :
Le JCA (Java Cryptography Architecture) fournit des classes et surtout des interfaces. Ces éléments se trouvent dans les packages java.security et javax.crypto.

On retrouve dans les méthodes de ces classes de cryptographie :

1. Des APIs (Application Programming Interface) : il s'agit des méthodes, publiques, que l'on peut appeler directement dans une application et qui implémentent des SPIs de manière encapsulée.
2. Des SPIs (Service Provider Interface) : il s'agit des méthodes d'interfaces, donc en faite des méthodes sans implémentation.

⇨ L'implémentation d'une SPI permet de se servir de ces méthodes via une API.

Le JCA propose la démarche suivante : les véritables classes nécessaires au travail cryptographique ne sont pas instanciées au moyen d'un constructeur mais plutôt en utilisant des méthodes factory, la plus répandue étant getInstance(). Cela permet, de laisser au système le choix d'instancier pour nous une instance du provider le plus adéquat et qui répond au mieux à nos attentes, selon le paramètre passé et ce sans pour autant connaître le nom classe qui sera finalement instanciée. De plus, cela permet d'acquérir l'instance la plus adéquate au moment de l'exécution.

Programmation Java :

Du côté de la programmation Java, on aura plutôt tendance à chercher du côté des factory pour instancier un objet notamment avec des méthodes du type getInstance() ou getConnection() auxquelles sont passées des paramètres tels que l'algorithme souhaité ou encore le provider de la base de données. L'utilisation massive de méthodes factory permet de rendre le code plus compréhensible pour le programmeur, mais aussi, entièrement modulaire et portable (en effet, un changement de provider devrait ne pas nécessiter de gros changement dans le code source).

Fonctionnement de getInstance :

A chacun des providers que l'on peut utiliser, correspondra un objet instanciant une classe dérivée de la classe du JDK Provider, qui encapsule les notions de nom et de version. Mais surtout la classe Provider hérite elle-même de Properties qui permet de fournir une liste de propriétés, ces propriétés serviront en cryptographie à renseigner sur les algorithmes effectivement implémentés ou encore les classes de providers tels Oracle et autres en base de données. Par exemple

Ainsi, la méthode getInstance(), qui est toujours une méthode de classe, va **vérifier la chaine de caractères passée en paramètre, la découper et dès lors,** rechercher par le biais de la classe Security la liste des providers enregistrés (cette liste se trouve dans le fichier java.security). Selon un ordre de préférence défini par l'utilisateur avec la syntaxe suivante : **security.provider.<numéro>=<classe du provider>**, la méthode demandera à chaque provider un objet de la classe algorithme recherché. Si le provider visé la possède, il en fournira une instance, sinon, la recherche passe au provider suivant ou donnera une exception si la liste est épuisée.
Ainsi, par exemple, si l'on désire un générateur de clé (objet KeyGenerator) utilisant l'algorithme DES, on écrira : KeyGenerator cleGen = DESKeyGenerator.getInstance("DES");

19. Expliquer la nature et la finalité d'un message digest. Comment on peut l'utiliser dans un contexte d'authentification se substituant au schéma classique login-password ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ? Quel rapport avec les HMACs et comment ces derniers assurent-ils une authentification (légère) fiable (utiliser les notations K, PrK, PK, H, S, …) ?

Il est important de savoir si les données que l'on obtient par le réseau sont restées ce qu'elles étaient à leur envoie. Dans ce contexte, on utilise un message digest qui représente « la valeur de hachage » d'un document. Celle-ci s'obtient en utilisant une fonction de hachage qui, recevant un ensemble de données d'un taille quelconque, fournit une chaîne de taille fixe (typiquement 128 bits). Comme c'est l'ensemble des données qui est utilisé, la probabilité de trouver deux fois la même valeur résultante pour des messages différents est extrêmement faible. De plus les fonctions de hachage idéales ne peuvent être inversées. Il est donc impossible de retrouver la donnée qui a produit un digest.

L'intégrité est donc vérifiée en :

1. Calculant le message digest sur le texte obtenu ;
2. Le comparant au message digest qui accompagnait le texte.

En cas d'égalité, on peut penser que le texte original n'a pas été modifié.

Lorsqu'un client s'authentifie auprès d'un serveur, il doit lui fournir un login et un mot de passe. On souhaite ne pas faire circuler en clair ce mot de passe sur le réseau. En voici le principe :

Le client :

1. Connaît son mot de passe pour accéder au serveur
2. Crée un digest avec son mot de passe
3. Envoie son nom et le digest au lieu de son mot de passe en clair

Le Serveur :

1. Reçoit le nom et le digest
2. Rechercher le nom dans son fichier ou sa BD et calcule le digest correspondant
3. Si le digest calculer correspond au digest reçu, l'utilisateur est authentifié.

Pour plus de sécurité, le digest sera calculé avec le mot de passe complété d'un nombre aléatoire et de l'heure. Ces deux informations seront envoyées en clair au serveur, afin qu'il puisse lui aussi calculer le digest pour le comparer à celui qu'il a reçu. On peut évidemment ajouter encore des éléments supplémentaires pour renforcer la méthode.

Les algorithmes de message digests courants :

- MD2, MD4 et MD5
- SHA-1
- RIPEMD-160

Programmation Java

Tout d'abord, il faut obtenir une instance de digest qui utilise l'algorithme SHA-1 et qui est fournie par le provider Cryptix, au moyen de la méthode factory :

- public static MessageDigest getInstance(String algorithm, String Provider)throws NoSuchAlgorithmException, NoSuchProviderException

Donc, dans notre cas :

- MessageDigest md = MessageDigest.getInstance("SHA-1", "Cryptix");

Il faut ensuite préparer le digest en lui passant nos « ingrédients » (nom, mot de passe, nombre aléatoire, heure) en utilisant la méthode update() :

- public void update(byte input[])

Et pour finir appeler la méthode :

- public bye[] digest(byte input[])

qui applique effectivement l'algorithme sélectionné pour fournir le digest sous forme de bytes.

HMAC

En généralisant la notion de digest, on définit un MAC (Message Authentification Code) comme un petit bloc de taille fixe et qui a également pour objectif d'assurer l'authentification et l'intégrité d'un message.

- Techniquement, un MAC simple peut être un digest résultant d'un hashage (MAC simple hashé).
- Mais il peut aussi utiliser plutôt une clé secrète (MAC simple à clé) : le résultat de l'application d'un chiffrement symétrique utilisant cette clé au message constitue alors le MAC. Celui-ci est envoyé au destinataire avec le message en clair. Le destinataire reçoit donc le message et le MAC. Comme il connaît la clé secrète, il peut calculer sur le message clair un second MAC. Si les deux MACs sont semblables,
    1. cela signifie que l'intégrité est vérifiée
    2. cela assure aussi de l'authentification puisqu'il faut partager le secret de la clé secrète pour vérifier cette intégrité.

L'analogie avec le mécanisme du digest n'a certainement pas échappé au lecteur attentif. Et, de fait, un MAC peut aussi utiliser conjointement une clé et un digest (MAC complexe à clé et hashage) : celui-ci est construit avec la chaîne d'entrée et cette même chaîne préalablement cryptée au moyen d'une clé secrète en plus. Le représentant typique de cette famille est le HMAC (keyed-Hash Message Authentification Code).

$HMAC(m) = H(m + \{m\}K_{ab})$ ➔ (m+x) ➔ x ?= $H(m+\{m\} K_{ab})$

K_{ab} = clé de session (symétrique)
Authentification + intégrité

20. Qu'est-ce qu'une signature digitale (ou numérique ou électronique) et comment l'utiliser pour assurer l'intégrité et l'authentification (lourde) ? Qu'entend-on par "vérifier une signature" (utiliser les notations K, PrK, PK, H, S, …) ? Quelles sont les analogies et/ou les différences avec une signature manuscrite classique ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ?
Pourquoi le catalogue des fonctionnalités cryptographiques offertes par un provider ne citent-elles pas apparemment les algorithmes de cryptage asymétrique ?

Concept :
Une signature électronique est un bloc de données qui a été créé en utilisant une clé privée et qui peut être vérifié par la clé publique correspondante sans la connaissance de la clé privée. Elle permet d'authentifier l'expéditeur des données. En effet, avec la confidentialité et l'intégrité, l'authentification est certainement le problème majeur de la transmission sécurisée d'information au niveau applicatif. Signature numérique ou digitale ne sont que des synonymes. Les pays anglo-saxon préfèreront parler de signature digitale qu'électronique. Attention qu'une signature n'est pas nécessairement un certificat tel que nous les connaissons dans le monde de l'informatique.

La construction d'une signature basée sur un digest :
Du côté de l'expéditeur :

1. Construction d'un message digest du document à envoyer, selon un algorithme de digest (par exemple SHA-1 ou MD5)
2. Chiffrer au moyen d'un algorithme approprié de chiffrement asymétrique (par exemple RSA), la chaine obtenue pour le digest en utilisant la clé privée du signataire (qui restera chez ce signataire). C'est là que l'on obtient la signature électronique.
3. L'expéditeur envoie alors le message et la signature.

Du côté du destinataire :

1. Réception du message avec la signature et déchiffrement de cette signature au moyen de la clé publique du signataire du message (car chiffrement avec la clé privée du signataire). Il obtient ainsi le message digest calculé lors de l'envoi.
2. Calculer un second message digest pour le message obtenu. Si les deux digests correspondent, on peut être assuré que l'information n'a pas été falsifiée, ce qui assure donc l'intégrité ; dans ce cas aussi, la clé privée utilisée était bien celle du signataire, ce qui assure donc aussi l'authentification.
3. On peut également convaincre un tiers que le document reçu a bien été envoyé par le propriétaire de la clé prouvée, pas par quelqu'un d'autre : on résout ainsi le problème de la non-répudiation.

Vérification de la signature :

Comme expliqué ci-dessus, la vérification de la signature consiste à "déchiffrer" la signature reçue à l'aide de la clé publique de l'expéditeur puis, à comparer le clair obtenu avec le digest calculé à partir du message effectivement reçu. Si les deux digests sont identiques, alors la signature est valide.
La vérification d'une signature assure la détection de la perte d'intégrité du message ainsi que l'authentification de l'origine du message. La vérification ne garantit l'authentification que si l'auteur est bien le seul détenteur du secret qu'est la clé privée de signature.

Analogie avec la signature manuscrite :

Par rapport à la notion de signature manuscrite, une nuance est d'importance et mérite d'être bien soulignée :
"Une signature digitale est fonction, outre de la clé privée utilisée, du message qu'elle accompagne : elle est donc propre non seulement au propriétaire de la clé privée mais aussi à ce message."

Cela signifie que chaque utilisation d'une signature électronique, pour une même personne, sera différente pour chaque message alors que pour une signature manuscrite la signature reste identique quelque soit le message. Cependant, le but reste le même : authentifier l'expéditeur du message.

Cette différence entre les deux types de signature est, en pratique, due au fait qu'un hacker qui aurait dérobé une signature digitale ne saurait s'en servir pour signer un autre message, ce qui assure une notion de sécurité puisqu'est dès lors sûr à 100% de l'expéditeur du message.

Différents algorithmes de signature :

L'algorithme **DSA** (Digital Signature Algorithm) utilise une clé de 1024 bits. Il est adopté par le NIST sous le nom de DSS (Digital Signature Standard) et est utilisé pour effectuer des signatures électroniques et ce par l'intermédiaire de 3 paramètres publics.

RSA (R. Rivest, A.Shamir & L.Adleman) qui correspond à un algorithme asymétrique, adapté dans notre cas aux signatures, basé sur la difficulté de factoriser des grands nombres. Ces nombres ont pour caractéristique principale d'être premiers. Son efficacité se base sur des clés de 1024 bits.

Programmation Java :
Comme dit précédemment, nous avons besoin d'un algorithme asymétrique et donc d'un couple de clés. Ce couple peut être obtenu via la méthode factory de la classe KeyPairGenerator présente le package java.security :

public static KeyPairGenerator getInstance(String algo) throws NoSuchAlgorithmException ou
public static KeyPairGenerator getInstance(String algo, String provider) throws NoSuchAlgorithmException

Il subsiste pour cette méthode plusieurs constructeurs: un avec simplement l'algorithme (RSA) et un deuxième qui inclut en plus le provider (Cryptix/Bouncy Castle). Une fois l'objet désiré obtenu, on va l'initialiser via sa méthode:

public void initialize(int tailleCle, java.security.SecureRandom source) ;

où les paramètres correspondent à la taille de la clé (512/1024) et à un nombre aléatoire pour la génération des clés. On peut dès lors récupérer notre couple de clés dans un objet KeyPair via la méthode :

public java.security.KeyPair generateKeyPair()

Cet objet KeyPair n'est rien d'autre que la combinaison des clés privée et publique que l'on obtient via les méthodes:

public PrivateKey getPrivate() et public PublicKey getPublic()

où PublicKey et PrivateKey dérivent de l'interface Key.

Une fois les clés obtenus, on peut les sérialiser dans le but d'un usage ultérieure afin de conserver la même clé publique qui sera envoyée au destinataire pour décrypté le message (la clé privée elle restera comme d'habitude uniquement sur la machine de l'expéditeur).

Ensuite, pour pouvoir créer une signature, il faut récupérer une instance de la classe abstraite appellée Signature du package java.security. Cette instance est obtenue via la méthode factory getInstance en lui passant comme paramètre les algorithmes (cryptage, mode de chiffrement et padding) que l'on désire.

```
public static Signature getInstance(String Algo) throws NoSuchAlgorithmException
public static Signature getInstance(String Algo, String Provider) throws NoSuchAlgorithmException
```

On initialise l'objet signature obtenu à l'aide de la fonction initSign qui reçoit la clé privée (obtenue précédemment ou via la récupération d'un objet sérialisé) contenue dans l'objet KeyPair générer juste avant.

```
public final void initSign(PrivateKey clePrivee) throws InvalidKeyException
```

Une fois l'initialisation effectuée, il faudra indiquer ce que l'on désire signer via la fonction update en lui passant ces données que l'on veut signer via un tableau de bytes.

```
public final void update(byte data[]) throws SignatureException
```

Enfin, pour récupérer la signature, il suffira d'appeler la méthode qui génère effectivement la signature à savoir la méthode sign qui renvoi la signature sous forme d'un tableau de byte.

```
public final byte[] sign() throws SignatureException
```

L'expéditeur/signataire enverra alors ses données auxquels il joindra la signature que l'on vient de créer.

En ce qui concerne le destinataire, il devra tout d'abord se procurer la clé publique qui lui sera transmise par l'expéditeur. Ensuite, il doit lui aussi générer un objet Signature sur le même algorithme que le client et initialiser cet objet avec la fonction initVerif qui reçoit la clé publique en paramètre.

```
public final void initVerif(PublicKey clePublique) throws InvalidKeyException
```

Il place les données reçu par le réseau dans l'objet via la fonction update(). Attention, il est important que le destinataire replace bien les mêmes données dans l'objet que le client sous peine de se retrouver avec une signature invalide.
Et enfin, il vérifie la signature par la fonction verify en passant la signature en paramètres. Le boolean renvoyé sera à true si la signature est valide.

```
public final boolean verify(byte signature[]) throws SignatureException
```

Exemple concret :

Expéditeur

```
…
PrivateKey clePrivee ;
…
clePrivee = (PrivateKey)oos.readObject() ;        //Récupération de la clé sérialisée sur le disque
Signature s = Signature.getInstance("SHA-1/RSA/PKCS#1","Cryptix") //Méthode factory pour obtenir l'instance d'une signature
s.initSign(clePrivee) ;                           //Initialisation en fonction de notre clé privée
s.update(message) ;                               //Ajout des données à signer
byte[] signature = s.sign() ;                     //Obtention de la signature
...

// Envoi du message et de la signature
```

Destinataire

```
…
//Lecture du message et de la signature envoyée par l'émetteur
…
PublicKey clePub ;
…
clePub  = (PublicKey)oos.readObject() ; //Récupération de la clé sérialisée sur le disque ou envoyée par l'expéditeur
Signature s = Signature.getInstance("SHA-1/RSA/PKCS#1","Cryptix") ;
s.initVerify(clePub) ;
s.update(message) ;
boolean ok = s.verify(SignatureRecue) ; //On compare avec la signature qui accompagnait le message
…
```

Pourquoi le catalogue des fonctionnalités cryptographiques offertes par un provider ne citent-elles pas apparemment les algorithmes de cryptage asymétrique ?

Le catalogue des fonctionnalités cryptographiques offertes par un provider ne citent pas les algorithmes de cryptages asymétriques car ceux-ci nécessite des fonctions mathématiques coûteuses en ressources machines aussi bien en puissance qu'en temps, ce qui implique un temps d'exécution des programmes trop important. Si l'on souhaite crypter toute une communication il faut utiliser un mélange de cryptage symétrique et asymétrique : la clé de session.

$S(m) = \{H(m)\}PrK_a = x$ ➔ m+x ➔ $H(m) ?= \{x\}PK_a$

S => signature
PrK_a = Private Key
PK_a = Public Key (provident d'un certificat)

Non repudiation + intégrité + authentification

21. Qu'entend-on par "certificat", "certificat X509", "niveau 1-2-3,3+)" et "vérifier un certificat" ? A quoi correspondent les formats DER et PEM ? Qu'est-ce qu'un keystore" ou "magasin à clés" ? Expliquer dans une liste des principales commandes comment les couples de clés et les certificats sont gérés au moyen de l'outil keytool. Comment procède-t- on en programmation Java (quelles classes, fichier et classe KeyStore, obtenues comment, quelles méthodes, …) ?

En utilisant une signature digitale, on peut certifier qu'une clé publique appartient bien à une certaine personne en créant ainsi ce que l'on appelle un certificat, c'est-à-dire un message qui a valeur officielle et qui a la reconnaissance de tous.
Concrètement on construit un message comportant :

1. l'identification de son propriétaire
2. la clé publique

et ce message est ensuite signé au moyen de la clé d'un organisme en qui on a toute confiance que l'on appelle les certification authority (CA) (par exemple : Verisign). Le message ainsi complété de l'identification du CA et de sa signature constitue ce que l'on appelle un certificat d'authentification. Il servira d'attestation officielle que la clé publique en question est bien celle d'une personne précise.

Certificat : Informations fondamentales
-Identification du propriétaire. -Clé publique. -Identification du CA -Signature digitale du CA

La norme la plus utilisée est dénommée X.509 et stipule qu'un certificat comporte :

Certificat X509 : Informations plus complètes
-Numéro de version -Numéro de série -Identifiant de l'algorithme de signature -Identification du propriétaire de la clé publique certifiée. -Période de validité -Identification du CA -Information sur l'algorithme de la clé publique -Clé publique. -Extensions diverses -Signature digitale du CA pour les champs ci-dessus

Formats de certificats

DER (Definite Encoding Rule) -> encodage ASN.1 (.der, .cer, .crt, .cert)
PEM (Privacy Enhanced Mail) -> DER en base64 + en-tetes en ASCII

Comment savoir si le certificat que l'on vient de recevoir, et qui fournit la clé publique de quelqu'un, est le vrai ?

On va d'une part calculer le digest des informations contenues dans le certificat et d'autre part utiliser la clé publique du CA pour obtenir le digest qui est à la base de la signature : si les deux digests coïncident, la signature est donc déclarée valide.

Pour vérifier cette signature, il faut donc disposer de la clé publique du CA, deux possibilités :

- le CA est bien connu, sa clé publique aussi et la vérification peut se faire sans problème.
- le CA est seulement local, il faut donc se procurer un certificat pour ce CA local, certificat signé par un autre CA plus connu. L'opération est à recommencer pour ce nouveau certificat et ainsi de suite jusqu'à parvenir à un CA connu (on parle de chaîne de certificat).

La classe KeyStore du package java.security permet de réaliser la gestion des clés. En fait il s'agit d'une sorte de dictionnaire qui contient, de manière cryptée, deux types d'entrées :

- Key Entry : une clé privée et une liste de certificats concernant la clé publique correspondante. On peut ainsi prouver qui l'on est. Rien n'interdit de disposer de plusieurs couples clé publique/clé privée.
- Trusted Certificate Entry : un certificat d'une personne considérée comme sûre. On peut ainsi authentifier une autre partie.

Ces informations sont désignées, au sein de la structure, par des identifiants appelés des alias qui permettront de retrouver l'information demandée (comme une clé).
De manière sémantique, un objet KeyStore exemple ressemble à ceci :

Alias	Information	Usage
cleApplets	Clé privée+clé publique + certificat(s)	Signer des applets
cleMails	Clé privée+clé publique + certificat(s)	Signer des mails
Philippe	Certificat	Authentifier
Roland	Certificat	Authentifier
Virginie	Certificat	Authentifier

L'implémentation de l'objet KeyStore est laissée à la discrétion des providers. Une implémentation par défaut est cependant fournie par Sun : appelé « JKS », elle utilise un fichier au format propriétaire (nommé classiquement .keystore).

On peut spécifier un type de keystore dans la méthode de la classe :

- public static KeyStore getInstance(String type)

La classe comporte des methods permettant d'ajouter des entrées à l'objet crée :

- public final void setKeyEntry(String alias, byte[] key, Certificate[] chain)
- public final void setKeyEntry(String alias, byte[] key, char[] password, Certificate[] chain)
- public final void setCertificateEntry(String alias, Certificate cert)

On retrouve les informations au moyen des methodes :

- public final Key getKey(String alias, char[] password)
- public final Certificate getCertificate(String alias)

Cette dernière méthode fournit donc le certificat associé à l'alias précis.
Concernant le certificat on retrouve la méthode :

- public abstract void verify(PublicKey key)

qui permet de verifier que le certificate a été signé au moyen de la clé privée associée à la clé publique passée en argument.

On peut aussi gérer une keystore au moyen d'un outil en ligne de commande appelé keytool, outil qui est apparu avec le jdk1.2. On peut générer un certificat signé par l'utilisateur. La paire de clé produite (option –keygen), selon l'algorithme précisé (option –keyalg) correspondra à une entrée de type « key entry », la clé publique étant celle faisant l'objet du certificat. L'entrée en question peut être désignée par un alias (option –alias). Le nombre de bits utilisés peut être spécifié (option –keysize). On se souviendra enfin qu'un certificat contient une identification du propriétaire. L'option –dname permet de définir un distinguished name (DN), qui décrit une personne de manière normalisée.

Dans tout certificat on retrouve un "Certificate fingerprint", qu'il faut comprendre comme un digest qui a été calculé sur le certificat entier.

Il existe 2 types de certificats :

- certificat auto-signé le signataire d'un tel certificat est le même que celui dont la clé publique fait l'objet de la certification.
- certificat réel atteste de sa clé publique, un utilisateur quelconque doit construire une demande que l'on appelle un CSR (Certificate Signing Request). Il s'agit d'un fichier qui contient :
    - la clé publique du demandeur
    - la signature construite au moyen de la clé privée du demandeur

Le CA, après réception, pourra alors vérifier le bien fondé de la demande et, dans l'affirmative, fournir le certificat demandé.

Dans le cas où l'on recoit un certificat d'un tiers (qui n'a donc pas d'alioas dans le Keystore), on créera une « Trusted Certificate Entry ».

(Attention voir prise de notes)

22. Expliquer les principes généraux de paramétrage des sockets (primitives, types de paramètres, …). Expliquer avec du code concret des situations courantes où il est nécessaire de paramétrer les sockets TCP et deux autres (différentes) pour les sockets UDP.

Même si dans la plupart des cas on utilise le comportement par défaut des sockets, on peut dans certains cas modifier ceux-ci afin de mieux maîtriser leurs comportements. On peut par exemple donner à une socket des caractéristiques non bloquantes en utilisant deux fonctions qui ne sont pas propres aux sockets mais concernent tout fichier au sens d'UNIX, il s'agit de fcntl() et ioctl().

La fonction fcntl : contrôle des descripteurs de sockets
En utilisant les headers fcntl.h, sys/types.h et unistd.h, la fonction fcntl() classique a pour prototype:

int **fcntl**(<descripteur de fichier ou de socket – int>,
<requête – int>,
<argument de la requête – int ou struct flock *>) ;

L'utilisation de cette fonction, au spectre plus large rappelons-le, est ici celle qui emploi comme requête:
#DEFINE F_SETFL 4 /*Set file flags*/
Avec les indicateurs combinables:

Indicateur	Signification
# define **O_NONBLOCK** 00000004	/* non-blocking I/O, **POSIX** style */
# define **O_NDELAY** 00100000	/* non-blocking I/O*/ System V style
# define **FNDELAY** O_NDELAY	BSD style

On peut donc passer en mode non bloquant mais selon des indicateurs variables selon le système hôte.

La fonction ioctl : contrôle des périphériques flux
Pour le contrôle des périphériques flux, il existe une fonction prototypée dans stropts.h et qui s'écrit :

int **ioctl**(<descripteur de fichier ou de socket – int>,
<requête – int>,
<argument de la requête>);

On passe en mode bloquant si le troisième paramètre est non nul. La requête de passage en mode asynchrone (permettant l'utilisation du signal SIGIO) est celle qui utilise :
#define FIOASYNC _IOW('f',125,int) /*set or clear async IO*/

La requête concernant l'aspect bloquant ou non correspond à l'indicateur :
#define FIONBIO _IOW('f',126,int) /*set or clear non_blocking IO */

Différents paramétrages:

Etre le propriétaire d'une socket :

Il peut être utile qu'un processus se rende propriétaire d'une socket. Ceci lui permet ainsi de pouvoir être avisé de l'un ou l'autre signal (comme SIGURG ou SIGIO). L'opération peut se faire :

- avec fcntl :
  En utilisant la requête
  #define F_SETOWN 6
  le troisième paramètre étant le PID du process

  On obtient, à l'inverse, le PID du propriétaire dans le troisième paramètre avec :
  #define F_GETOWN 5

- avec ioctl
  #define FIOSETOWN _IOW('f', 124, int)
  #define FIOGETOWN _IOR('f', 123, int)
  ou
  #define SIOCSPGRP _IOW('s', 8, pid_t)
  #define SIOCGPGRP _IOR('s', 9, pid_t)

Obtenir des informations sur une socket :

La primitive à utiliser pour se faire une idée des caractéristiques d'une socket est prototypée dans socket.h et a pour forme :

int getsockopt (<handle de la socket – int>,
<niveau (socket ou protocole) – int>,

Ainsi, dans le domaine AF_INET, deux niveaux sont le plus fréquemment utilisés :

- le niveau socket : #define **SOL_SOCKET** 0xffff
- le niveau protocole : les informations obtenues s'appliquent donc en correspondance avec le protocole spécifié au moyen des constantes définies dans netinet/in.h
  Ex: **IPPROTO_IP, IPPROTO_TCP ou IPROTO_UDP**

<option – int>,

- booléenne : elle s'applique à la socket ou pas à l'instant considéré
- non booléenne : l'information reçue est donc cette fois une véritable valeur qui est celle d'une caractéristique de la communication dont la socket est chargée.

<valeur correspondante reçue – void *>,
<longueur de la zone pointée – int *>);

La nature du quatrième paramètre dépend de la nature du renseignement demandé tandis que le dernier paramètre contient la longueur de la zone recevant l'information: il est initialisé à la longueur attendue, mais rectifié éventuellement par la primitive sur la longueur de son résultat.

Cette primitive renvoie -1 en cas d'erreur et 0 en cas de succès.

Modifier les options d'une socket:

Grâce à la primitive : int setsockopt (<handle de la socket – int>,
<niveau (socket ou protocole) – int>,
<option – int>,
<valeur à donner à l'option – void *>,
<longueur de la zone pointée – in>);

Celle-ci renvoie -1 en cas d'erreur et 0 en cas de succès.

Exemple de paramétrage courant en UDP :

Pour pouvoir recevoir des datagrammes multicast, nous avons besoin d'une socket qui est liée à une adresse multicast définie (exemple : 234.5.5.9) et qui va se joindre au groupe multicast en utilisant à primitive setsockop :

setsockopt(hSocket, IPPROTO_UDP, IP_ADD_MEMBERSHIP,&mreq, sizeof (mreq))

Ici, on spécifie bien les paramètres nécessaires à savoir : la socket concernée, le protocole (UDP), la constante signifiant l'ajout au groupe multicast, mreq correspondant ici à l'adresse de multicast que l'on veut atteindre via une structure ip_mreq qui contient dans son champ imr_multiaddr le champ sin_addr de la stuct sockaddr_in paramètrée (adresse multicast, port) et enfin la taille de cette structure.

Néanmoins, dans l'exemple du multicast, la même adresse et le même port seront utilisés. Comme une adresse locale ne peut correspondre qu'à un seul socket, il faut contourner cela en, pour les sockets (ne marche que pour celle du domaine AF_INET), positionnant l'option correspondante au moyen des constantes :

#define SO_REUSEADDR 0x0004
#define SO_REUSEPORT 0x0200

Ensuite il suffit de programmer :

```
int flagResue = 1;
intMaSocket ;
if (setsockopt (MaSocket,SQL_SOCKET,SO_REUSEADDR, (char *) &flagReuse, sizeof (flagResue)))
{
        exit (1);
}
if (setsockopt (MaSocket,SQL_SOCKET,SO_REUSEPORT, (char *) &flagReuse, sizeof (flagResue)))
{
        exit (1) ;
}
```

Exemple en TCP :
La constante #define TCP_MAXSEG 0x02 permet d'obtenir ou de diminuer la taille du segment. On ne peut utiliser que des sockets SOCK_STREAM. Si la socket est connectée, on aura la valeur du MTU diminuée de la taille des en-têtes IP et TCP.

```
Int tailleS, tailleO ;
…
tailleO = sizeof(int) ;
if(getsockopt(hSocketConnectee,IPPROTO_TCP,TCP_MAXSEG, &tailleS, & tailleO) == -1)
{
        Printf("erreur sur le getsockopt ») ;
        Exit(1) ;
}
else
{
        Printf("OK");
        //la taille max est donc tailleS
}
```

23. Expliquer (utiliser les notations K, PrK, PK, H, S, …) ce que l'on entend par "authentification légère" (symétrique) et "authentification forte" (asymétrique). Comparer, notamment du point de vue charge machine, fiabilité, difficultés d'échanges, programmation en Java …

Avec la confidentialité, l'authentification est certainement le problème majeur de la transmission sécurisée d'informations au niveau applicatif. Le problème étant de vérifier que le message reçu provient réellement de celui que l'émetteur prétend être.

Authentification légère

Un MAC simple peut être un digest résultant d'un hashage (MAC simple hashé). Mais il peut aussi utiliser une clé secrète (MAC simple à clé) : le résultat de l'application d'un chiffrement symétrique utilisant cette clé au message constitue alors le MAC. Celui-ci est envoyé au destinataire avec le message en clair. Le destinataire reçoit donc le message et le MAC. Comme il connaît la clé secrète, il peut calculer sur le message clair un second MAC. Si les deux MACs sont semblables :

1. cela signifie que l'intégrité est vérifiée
2. cela assure aussi de l'authentification puisqu'il faut partager le secret de la clé secrète pour vérifier cette intégrité.

Un MAC peut aussi utiliser conjointement une clé et un digest (MAC complexe à clé et hashage). Celui-ci est construit avec la chaîne d'entrée et cette même chaîne préalablement cryptée au moyen d'une clé secrète en plus. Le représentant typique de cette famille est le HMAC

L'authentification se fera au moyen du HMAC fabriqué à partir du message. Ce HMAC est donc totalement dépendant de ce message et de la connaissance de la clé symétrique nécessaire : c'est ce dernier point qui est le fondement de l'authentification. Les deux parties devront donc disposer de cette clé symétrique.

Ce mécanisme d'authentification est qualifié d'authentification légère car il ne fait intervenir que des algorithmes peu coûteux en opérations machines, principalement des chiffrements symétriques. Bien sûr la faiblesse que l'on peut formuler à la cryptographie symétrique est que la clé doit absolument rester secrète. Il faudra donc pouvoir échanger la clé secrète en toute sécurité.

$HMAC(m) = H(m + \{m\}k_{ab})$ ➔ $(m+x)$ ➔ $x \stackrel{?}{=} H(m+\{m\}\ k_{ab})$

k_{ab} = clé de session (symétrique)
Authentification + intégrité

Authentification lourde

Une authentification lourde est assurée par une signature électronique. Cette signature est un bloc de données qui a été créé en utilisant une clé privée et qui peut être vérifié par la clé publique correspondante sans la connaissance de la clé privée. Le terme de signature électronique possède des synonymes, on parle aussi de signature numérique ou encore de signature digitale.

Construction d'une signature basée sur un digest :

Pour construire une signature,

1. on construit un message digest du document à envoyer, selon un algorithme de digest (par exemple, SHA-1)
2. on chiffre au moyen d'un algorithme approprié de chiffrement asymétrique (par exemple, RSA), la chaîne obtenue pour le digest en utilisant la clé privée du signataire. Ce que l'on obtient est la signature électronique ou digitale.

L'expéditeur envoie alors le message et la signature. Le destinataire :

1. déchiffre la signature au moyen de la clé publique du signataire du message, il obtient ainsi le message digest calculé lors de l'envoi.
2. calcule un second message digest pour le message obtenu, si les deux digests correspondent, on peut être assuré que l'information n'a pas été falsifiée, ce qui assure donc l'intégrité. Dans ce cas aussi, la clé privée utilisée était bien celle du signataire, ce qui assure donc aussi l'authentification.
3. peut convaincre un tires que le document reçu a bien été envoyé par le propriétaire de la clé privée, pas par quelqu'un d'autre : on résout ainsi le problème de la non-répudiation.

Une signature électronique est propre non seulement au propriétaire de la clé privée mais aussi à ce message. Ce qui signifie donc qu'un hacker qui aurait dérobé une signature électronique ne saurait s'en servir pour signer un autre message. Cette méthode est donc très sûr car d'une part une signature électronique ne peut être réutilisée et la clé privée ne doit pas circuler sur le réseau, seul la clé publique est distribuée.

Le chiffrement asymétrique est par contre plus complexe que le chiffrement symétrique et donc le temps et les ressources nécessaires sont beaucoup plus importants.

$S(m) = \{H(m)\}PrK_a = x$ ➔ $m+x$ ➔ $H(m) ?= \{x\}PK_a$

S => signature
PrK_a = Private Key
PK_a = Public Key (provident d'un certificat)

Non repudiation + intégrité + authentification