Les problèmes de sécurité sur Internet

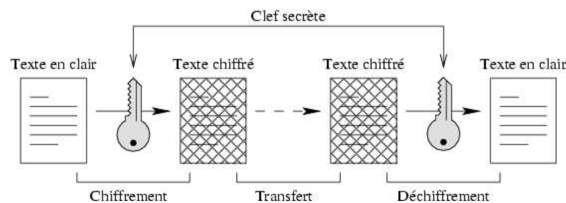
- **Confidentialité** (*eavesdropping*)
  - Les données sont intactes, mais peuvent être récupérées par un tiers.
- **Intégrité** (*tampering*)
  - Les données sont modifiées ou remplacées lors de leur transit.
  - Problème du **rejeu** de données interceptées
- **Usurpation d'identité** (*impersonation*): 2 variantes
  - *Spoofing* : typiquement, usurpation d'une identité valide
    - Envoyer des mails de chirac@elysee.gouv.fr, ou s'assigner l'@IP d'une machine valide.
  - *Misrepresentation* : maquillage de l'identité
    - Installer sur www.vendeur.org un site qui ressemble à www.vendeur.com, et enregistrer des commandes par carte bleue.
  - Problèmes d'**authentification** (client et/ou serveur)
  - Problèmes de **non-répudiation**

Moyens cryptographiques

- Cryptage/décryptage (chiffrement)
  - Confidentialité: données incompréhensibles lors du transit
  - Authentification: nécessité de disposer de « l'autre » clé
- Hachage cryptographique
  - Intégrité: toute modification est détectée
- Signature numérique
  - Authentification: s'assurer de l'identité de l'émetteur de données
  - Intégrité: s'assurer que, si la signature est valide, les données n'ont pas changées depuis leur signature
  - Non-répudiation: prouver que l'émetteur a bien émis ces données, et est le seul a avoir pu le faire (autorisation de prélèvement bancaire)

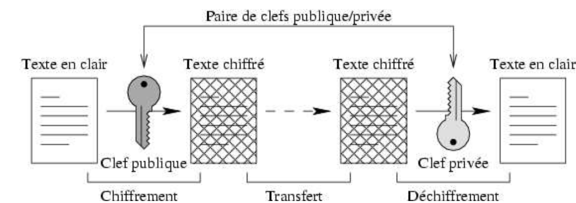
Chiffrement symétrique

- À **clé privée**, ou à **clé symétrique**
- Le récepteur DOIT avoir la clé pour déchiffrer (si qq'un d'autre la trouve, intégrité ET authentification sont perdues)
- Avantage: peut être très efficace (rapide); permutations, XOR...
- Inconvénient: connaître (ou communiquer) la clé aux deux parties
- Ex: DES, RC2, TripleDES (*bloc cipher*), RC4 (*stream cipher*).
- Sécurité dépend de la taille de la clé de chiffrement, de 40 à 168 bits



Chiffrement asymétrique

- À **clé publique**, ou **clé asymétrique**
- La clé privée est connue d'une seule partie, la clé publique est diffusée à tout le monde
  - Les données chiffrées avec la clé publique ne peuvent être déchiffrées que par le détenteur de la clé privée (le « bon sens » pour les données sensibles).
  - Pour la signature, c'est le contraire: le signataire chiffre les données avec sa clé privée, et le déchiffrage avec la clé publique prouve l'identité du signataire
- Plus lent (algos à base d'exponentiations). Ex: RSA, Elgamal



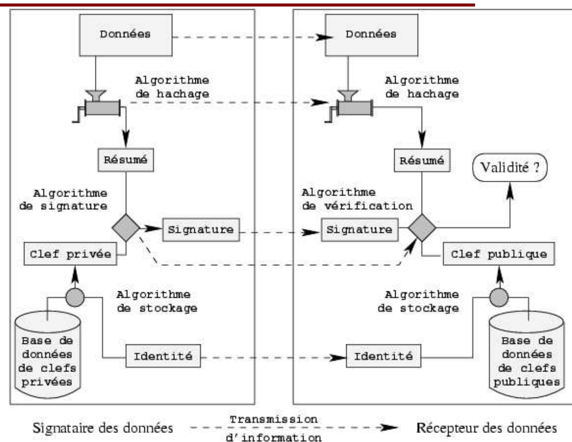
Hachage cryptographique

- Pour remédier au problème d'**intégrité**
  - Fonction de hachage: ***one-way hash*** ou ***message digest***
  - Propriétés:
    - La valeur de hachage produite pour les données hachées est « unique »
    - La moindre modification des données produit une valeur de hachage différente
    - Il est très difficile de retrouver les données à partir de la valeur de hachage
  - Couplé avec un algorithme de chiffrement asymétrique, permet de produire une *signature numérique* :
    - La signature d'un ensemble de données est une valeur de hachage de ces données, cryptée par une clé privée
      - plus la description des algos de hachage et de chiffrement
  - Exemple d'algorithmes: SHA-1, MD2, MD5
  - Sert de base à des algorithmes de chiffrement efficaces à clé secrete partagée, permettant d'assurer à la fois l'authentification et l'intégrité
    - Hash Message Authentication Code (HMAC), HMAC-MD5, HMAC-SHA1

Signature numérique

- À partir des informations à « signer »
  - On calcule la valeur du hachage cryptographique de ces données
  - On chiffre de cette valeur avec sa propre clé privée => signature
  - On transmet les données, la signature, l'identité du signataire et les deux algorithmes utilisés
- Le récepteur
  - Reçoit les données D et la signature chiffrée SC
  - Applique le hachage cryptographique à D et trouve H1
  - Récupère la clé publique du signataire à partir de son identité
  - L'utilise pour déchiffrer la signature reçue SC et trouve H2
  - Compare H1 et H2
    - La signature est *valide* si ils sont identiques
- Ex: DSS (`SHA1withDSA`) ou d'autres comme `MD2withRSA`, `MD5withRSA`, `SHA1withDSA`

Mécanisme de signature à clés asymétriques



Contraintes pour la validité d'une signature

- Si (H1==H2), i.e. signature valide, cela assure:
  - L'**intégrité**: les données signées n'ont pas été modifiées
  - L'**authentification**: les données proviennent bien du signataire
    - Plus exactement, la clé publique utilisée pour déchiffrer la signature correspond bien à la clé privée qui a servi à la chiffrer
    - Savoir si ce signataire correspond bien à l'identité présumée de l'émetteur de la signature requiert un autre mécanisme (certificat)
  - La **non-répudiation**: seul le signataire a pu produire ces données
- Pour vérifier une signature, il faut avoir:
  - Les deux algorithmes (hachage cryptographique et chiffrement)
    - Les noms des algos peuvent être transmis en clair avec la signature
  - L'identité du signataire (pour obtenir sa clé publique)
    - Attention: la clé publique du signataire NE DOIT PAS être transmise (simplement) avec la signature
      - Possibilité d'interception

Interception de signature avec clé publique: le problème

- Alice veut envoyer s=« je suis Alice » à Bob
  - Signature d'Alice: HashCrypto(s) = 123, ChiffrAsym(privateAlice,123) = ~#^
- Si Alice envoit dans son message sa clé publique
  - [s, Alice, HashCrypto, ChiffrAsym, ~#^, publicAlice]
- Alors un intermédiaire Charlie peut intercepter ce message et:
  - Récupérer s et sa valeur de hachage ChiffrAsym(publicAlice, ~#^) = 123
  - Créer un couple de clé privateCharlie/publicCharlie
  - Signer le message avec privateSpy: ChiffrAsym(privateCharlie,123)=@&$
  - Envoyer à Bob l'information initiale, en se faisant passer pour Alice et en fournissant sa propre clé publique
  - Soit : [s, Alice, HashCrypto, ChiffrAsym, @&$, publicCharlie]
- Du point de vue de Bob, qui reçoit au final ces informations
  - La signature est « vérifiable », mais elle n'autentifie pas le signataire

Solution: le certificat

- Délivré par une autorité de certification, **Certificate Authority (CA)**,
- ***« certifie »*** une identité
  - Associe une **clé publique** au **nom** d'une entité (le ***sujet*** du certificat)
  - Contient d'autres informations, comme la date limite de validité du certificat, l'**identité de l'émetteur** du certificat, un numéro de série, etc.
  - Il permet d'éviter d'associer Alice à publicCharlie
  - Cette association est **signée par l'émetteur du certificat**
  - Si on ne connait pas Alice, mais qu'on connaît **l'autorité de certification** qui a émis le certificat associant publicAlice à Alice (l'émetteur et le signataire du certificat) et qu'on lui fait confiance
  - alors on utilise la clé publique connue de l'autorité de certification « de confiance » pour vérifier la signature du certificat et on accepte le fait que la clé publique trouvée dans le certificat soit bien celle d'Alice
- Reporte le problème sur le CA: notion de chaîne de certificats

Certificat X.509v3 (ITU 1998)

- X.509: Norme de description des certificats (RFC 2459)
- Chaque certificat contient les informations suivantes:
  - Numéro de version de X.509 utilisée
  - Numéro de série unique du certificat attribué par l'émetteur
  - Algorithme utilisé pour signer le certificat
  - Distinguish Name de l'émetteur du certificat
  - Période de validité du certificat
  - Distinguish Name du sujet
  - Clé publique du sujet et algorithme pour lequel elle est valable
  - Extensions éventuelles
  - Une partie signature
    - L'algorithme utilisé pour signer le certificat
    - La signature de ce certificat, chiffrée avec la clé privée de l'émetteur
- Les *distinguished name (DN)* sont de la forme
  - nom canonique (CN), organisme (O), ville (L), code pays (C), etc.

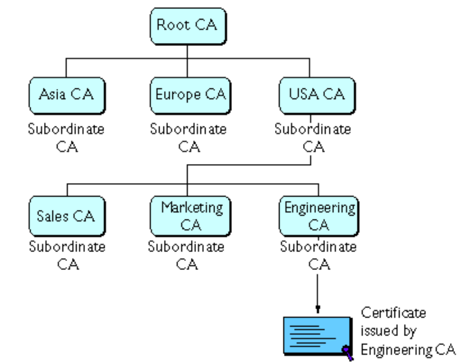

```
Certificate:
    Data:
        Version: v3 (0x2)
        Serial Number: 3 (0x3)
        Signature Algorithm: PKCS #1 MD5 With RSA Encryption
        Issuer: OU=Ace Certificate Authority, O=Ace Industry, C=US
        Validity:
            Not Before: Fri Oct 17 18:36:25 1997
            Not  After: Sun Oct 17 18:36:25 1999
        Subject: CN=Jane Doe, OU=Finance, O=Ace Industry, C=US
        Subject Public Key Info:
            Algorithm: PKCS #1 RSA Encryption
            Public Key:
                Modulus:
                  00:ca:fa:79:98:8f:19:f8:d7:de:e4:49:80:48:e6:2a:2a:86:
                  ... several lines deleted...
                  7d:d8:99:cb:0c:99:34:c9:ab:25:06:a8:31:ad:8c:4b:aa:54:
                  91:f4:15
                Public Exponent: 65537 (0x10001)
        Extensions:
            Identifier: Certificate Type
                Critical: no
                Certified Usage:
                    SSL Client
            Identifier: Authority Key Identifier
                Critical: no
                Key Identifier:
                  f2:f2:06:59:90:18:47:51:f5:89:33:5a:31:7a:e6:5c:fb:36:
                  26:c9
    Signature:
        Algorithm: PKCS #1 MD5 With RSA... several lines deleted...
```

Confiance dans un certificat

- Les CA (autorités de certification) valident des identités et émettent des certificats
  - Ils peuvent être indépendants ou gérer leur propres mécanisme de certification
- Clients ou serveurs manipulent des certificats
  - Ils gèrent des listes de certificats des autorités de certification en qui ils ont confiance (***Trusted CA certificates***)
- Organisation des CA de manière hiérarchique
  - La racine de la hiérarchie s'auto-certifie
    - On suppose qu'elle sera « de confiance » pour l'utilisateur
  - Chaque niveau en dessous est certifié par le CA du niveau au dessus.

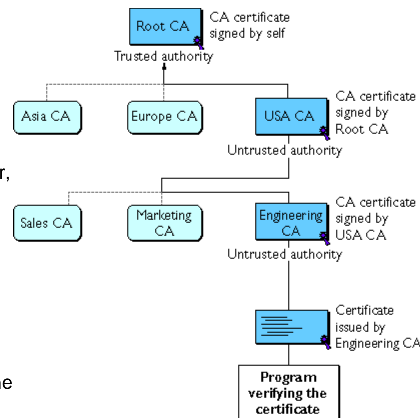
Exemple d'organisation hiérarchique de CA

- Un certificat émis par Root CA est certifié par Root CA
- Un certificat émis par USA CA est certifié par RootCA
- Ces hiérarchies sont représentées par des **chaînes de certificats**



Chaînes de certificats

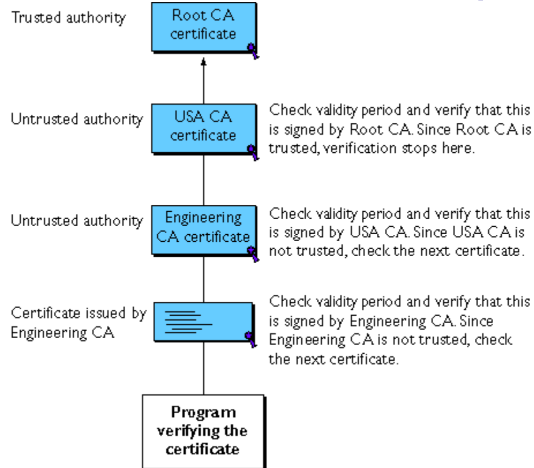
- Dans une chaîne de certificats:
  - Chaque certificat est suivi par le certificat de son émetteur
  - Chaque certificat contient le nom (DN) de son émetteur, qui est le sujet du certificat suivant dans la chaîne
  - Chaque certificat est signé avec la clé privée de son émetteur. Cette signature peut être vérifiée avec la clé publique située dans le certificat de l'émetteur, qui est le prochain dans la chaîne



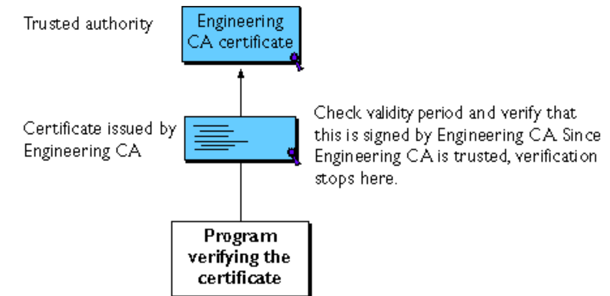
Vérifier une chaîne de certificats

- 1. Vérifier la période de validité w.r.t heure vérificateur.
- 2. Localiser l'émetteur du certificat
  - Soit dans la BD de certificats locale du vérificateur (client ou serveur)
  - Ou dans la chaîne de certificats fournie par le *sujet* (ex: SSL)
- 3. La signature du certificat est vérifiée avec la clé publique trouvée dans le certificat de l'émetteur
- 4. Si l'émetteur du certificat est « de confiance » dans la BD de certificat du vérificateur, alors la vérification s'arrête (OK). Sinon, le certificat de l'émetteur est vérifié en suivant la chaîne de certificat (retour étape 1).

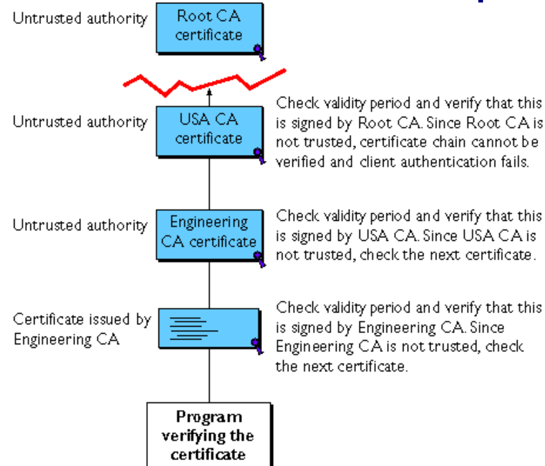
Vérification de chaîne: exemple 1



Vérification de chaîne: exemple 2



Vérification de chaîne: exemple 3

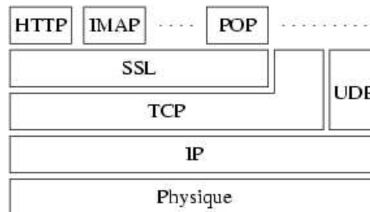


Secure Socket Layer

- Initialement développé par Netscape, SSL est devenu un standard permettant d'assurer au dessus de TCP/IP:
  - La **confidentialité** et l'**intégrité** des données échangées dans les deux sens (bi-directionnel)
  - La **détection des rejeux**
  - L'**authentification du serveur**
  - L'**authentification du client**
- Ne fournit pas la non-répudiation
  - Une requête peut avoir été générée par le serveur
- N'assure pas la confidentialité du trafic
  - Adresses des interlocuteurs et fréquence des échanges

SSL : modèle

- SSL en est à la version 3.1 chez Netscape, ce qui correspond chez IETF à la version 1.0 de TLS (*Transport Layer Security*), définie par la RFC 2246.
- S'intercale entre la couche transport TCP et les couches applicatives supérieures



Ce que permet SSL

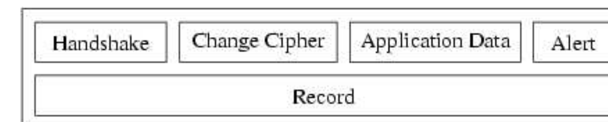
- L'**authentification**, par le client, **du serveur** contacté
- La **sélection**, par le client et le serveur, d'un ensemble d'**algorithmes cryptographiques** qu'ils supportent tous les deux
- L'**authentification** éventuelle, par le serveur, du **client**
- L'utilisation de **chiffrement à clé publique** pour **générer** des « **secrets partagés** »
- L'établissement d'une **connexion SSL chiffrée**

Ce dont SSL a besoin

- Pour cela, SSL nécessite
  - Un algorithme à clé publique d'échange de clés
  - Un algorithme de signature à clé publique
    - (éventuellement vide: ***anon***)
  - Un algorithme de chiffrement à clé secrète
  - Un algorithme de hachage cryptographique
  - Ces suites d'algorithmes sont représentés (RFC 2246) par des chaînes de caractères, par exemple
    - `SSL_RSA_WITH_RC4_128_MD5`
    - `TLS-DHE_anon_WITH_RC4_128_MD5`
    - `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`

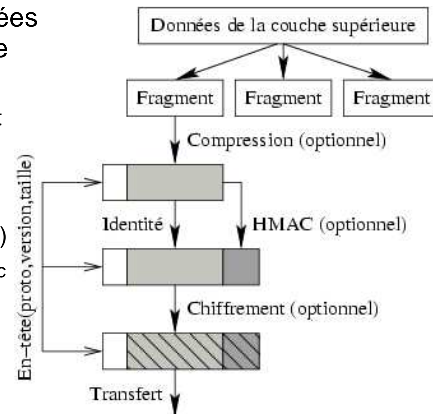
Le protocole SSL

- Le protocole SSL est composé de deux sous-couches
  - Sous-couche basse
    - Protocole *Record*
  - Sous-couche haute
    - Protocoles *Handshake, Change Cipher, Application Data et Alert*



Protocole Record

- Encapsule les données transmises et assure
  - La confidentialité grâce au chiffrement
  - L'intégrité grâce à l'algorithme HMAC (Hashed Message Authentication Code)
    - Hachage crypto avec clé secrète partagée
    - RFC 2104



Protocole Record (suite)

- Les clés secrètes utilisées par les algos de Record sont générées pour chaque connexion
  - À partir des valeurs négociées dans le Handshake
- Record est un protocole à état, défini par
  - Le côté (client ou serveur)
  - L'algorithme de chiffrement à utiliser
  - L'algorithme de hachage à utiliser
  - Une clé secrète primaire (*master secret*)
  - Une valeur aléatoire du client
  - Une valeur aléatoire du serveur
  - Une méthode de compression (qui vaut null)
  - Ces valeurs sont utilisées pour générer
    - Les clés secrètes du client et du serveur pour HMAC
    - Les clés secrètes du client et du serveur pour le chiffrement
    - Éventuellement des IV (*Initialisation Vector*) pour chaînage de blocs

Change Cipher, Alert et Application Data

- Protocole Change Cipher
  - Sert à indiquer le changement de chiffrement
    - Un seul message d'un octet
- Protocole Alert
  - Permet d'indiquer, par des messages sur 1 ou 2 octets, tout type d'erreur à l'utilisateur, et de fermer la connexion
- Protocole Application Data
  - Simple relai des applications de la couche applicative supérieure vers le protocole Record

Protocole de Handshake

- Cette partie du protocole permet au client et au serveur d'échanger des informations permettant
  - Au serveur de s'authentifier auprès du client
  - Au client et au serveur de créer conjointement les clés symétriques utilisées pour assurer le chiffrement / déchiffrement rapide et l'intégrité des données durant la session.
  - En gros, quatre phases
    - 1. Échanges *Hello*
    - 2. Envoi d'informations du serveur au client
    - 3. Envoi d'informations du client au serveur
    - 4. Échanges *Finish*

Etapes du Handshake: phase 1

- Phase 1: *Hello*
  - Le client envoit au serveur un message *Hello*
    - Version de SSL supportée, suites d'algorithmes supportés, valeur aléatoire horodatée, identificateur de session (éventuellement vide)
  - Le serveur envoit au client un message *Hello*
    - Version de SSL compatible avec le client, algorithmes choisis supportés par le client, valeur aléatoire horodatée, identificateur de session
- L'identificateur de session permet de réutiliser la session pour renégotier quelques paramètres
- C'est toujours le client qui initie la communication par un *Hello*, mais ensuite, n'importe qui peut demander une renégotiation dans la session en cours

Etapes du Handshake: phase 2

- Phase 2: Envoi d'informations du serveur au client
  - Si la suite d'algos choisie contient un algo de signature (différent de ***anon***), le serveur envoit un message *Certificate*
    - Contient une chaîne de certificats X509 compatible avec l'algorithme de signature utilisé
  - Si les infos du certificat du serveur (éventuellement vide) sont insuffisantes, le serveur peut émettre un message *Server Key Exchange*
    - Contient des paramètres pour l'échange de clés, signés avec la clé privée associée au certificat émis
  - Si le serveur a besoin d'authentifier le client, il émet également un message *Certificate Request*
    - Décrit les types de certificats reconnus (DSA ou RSA) et la liste des DN des CA connus par le serveur
  - Le serveur termine par l'envoi d'un message *Server Hello Done*

Etapes du Handshake: phase 3

- Phase 3:
  - Si le serveur a émis un *Certificate Request*, le client lui renvoit
    - s'il le peut un certificat respectant les paramètres décrits par le serveur
    - sinon, il renvoit un message vide
  - Le client envoit un message *Client Key Exchange* pour échanger une **clé primaire préliminaire**
    - L'échange se fait en utilisant l'algo d'échange de clé et la clé publique trouvée dans le certificat du serveur ou dans les paramètres du *Server Key Exchange*
    - Cette clé primaire préliminaire est générée à partir des données échangées depuis le début du hanshake, chiffrée avec la clé publique du serveur (trouvée dans son certificat), et envoyée au serveur qui pourra la déchiffrer avec sa clé privée
  - Si le client a émis un certificat, il renvoit un *Certificate Verify*
    - Signé avec sa clé privé, que le serveur pourra lire avec la clé publique trouvée dans le certificat: termine l'authentification du client

Etapes du Handshake: phase 4

- Phase 4: Fin du handshake
  - À partir de la clé primaire préliminaire et des valeurs aléatoires échangées, le client comme le serveur génèrent une clé primaire de 48 octets en utilisant une PRF (*Pseudo Random Function*)
  - Le client envoit un message *Change Cipher*: il passe en mode chiffré
  - Puis un message *Finish*
    - contient une valeur calculée avec une PRF à partir de la clé primaire et les valeurs de hachage des messages précédents et d'un label
  - Le serveur envoit un message *Change Cipher*, puis *Finish*
    - Si le client réussit à déchiffrer les données, le serveur est authentifié
- Ensuite, les données sont échangées, chiffrées en fonction de ce qui a été négocié, avec le sous-protocole *Application Data*
  - À la fin de la connexion, les 2 parties s'échangent des *Close Message*

Client

Serveur

client-hello(version,aléa,id,cipher,compress)

server-hello(version,aléa,id,cipher,compress)

Phase 1

certificate(x509v3)

server-key-exchange(param,sign)

certificate-request(type,ca)

server_hello_done()

Phase 2

certificate(x509v3)

client_key_exchange(crypt(pre-key))

certificate_verify(signature)

Phase 3

[change_cipher_spec]

finished(hachage)

[change_cipher_spec]

Finished(hachage)

Phase 4

Message obligatoire

Message optionnel

Message d'un autre protocole

Précautions particulières

- Une connexion SSL est aussi sûre que les algorithmes cryptographiques qu'elle utilise: pas plus!
  - C'est à l'utilisateur (application) de vérifier que SSL négocie des algorithmes qu'il considère comme sûrs
- De plus, avant d'accepter un certificat, client et serveur
  - Doivent vérifier la date de validité
  - Doivent vérifier la signature du certificat
  - Ensuite, doivent décider s'ils ont confiance ou non en ce certificat
    - En fonction de l'émetteur (trusted CA)
    - En fonction du DN présent dans le certificat (le **nom canonique CN** doit correspondre au **nom DNS du serveur**).
      - Peut nécessiter de demander à l'utilisateur
    - En fonction d'une liste de certificats révoqués

Attaque Man in the Middle

- Lorsqu'un client a constaté que la clé publique d'un émetteur de certificat permettait de vérifier la signature de ce certificat, il lui reste à assurer que cet émetteur est bien celui prétendu (et la manière de récupérer sa clé publique)
  - Pour cela, il doit comparer le nom DNS du serveur au nom de l'émetteur du certificat

Alice

Charlie

Bob

Alice
publicKey
Algos
Signature
privateKey

Alice
publicKey
Algos
Signature

Alice
publicKey
Algos
Signature
privateKey

Alice
publicKey
Algos
Signature

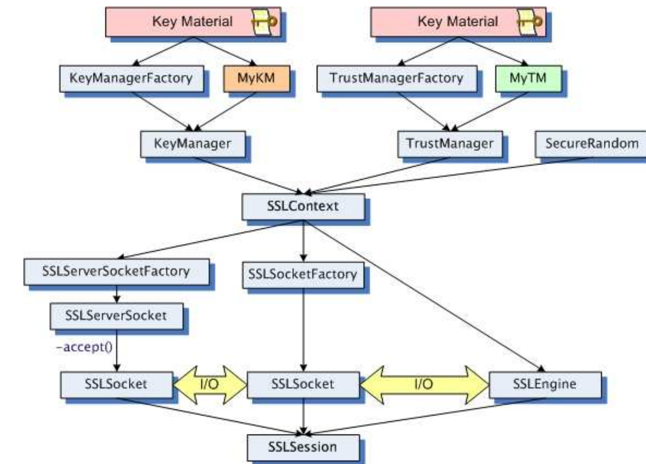
JSSE (Java Secure Socket Extension)

- Support pour SSL
  - Packages javax.net, javax.net.ssl, javax.security.cert
- Fait partie des différents « frameworks » concernant la sécurité en Java, qui s'entrecoupent
  - JCA (Java Cryptography Architecture) et JCE (Java Cryptography Extension)
    - Tout ce qui permet de gérer les algorithmes cryptographiques et le stockage des clés, des certificats, etc., majoritairement dans javax.security et ses sous-packages, et javax.crypto
  - JAAS (Java Authentication and Authorization Service)
    - La gestion de tous les aspects d'authentification, d'autorisation et leurs acteurs, principalement dans javax.security.auth
  - Autres: Java SASL, SSL-based RMI socket, etc.

JSSE, fonctionalités

- Depuis version 1.4, classe « standard »
  - Avant, classes d'« extension standard »
- Découpage interface/implémentation (notion de « provider »)
  - Implémentation découplée de l'API
    - Paramétrage facile
  - Dans J2SDK, depuis 1.4,
    - Interface (API) supporte SSL 2.0, 3.0 et TLS 1.0
    - Implémentation (`SunJSSE`) supporte SSL 3.0 et TLS 1.0
- Avantages
  - Offre les classes SSLSocket, SSLServerSocket et SSLEngine (nio)
  - 100% pur Java, accès « simple » à SSL, fonctions HTTPS
  - Support pour l'implantation et la négotiation de nombreux algorithmes
  - Support pour gestion et l'accès au stockage des clés / certificats

Classes principales



Les classes principales

- Une SSLSocket peut être créée
  - Soit par un accept() sur un SSLServerSocket
  - Soit par une SSLSocketFactory
- De même, une SSLServerSocket doit être créée par une SSLServerSocketFactory
- Les SSLSocketFactory comme les SSLServerSocketFactory sont créées par un SSLContext, qui regroupe
  - L'implantation du protocole SSL à utiliser
  - Un objet (KeyManager) qui permet de décider quelle clé privée et quel certificat utiliser pour s'authentifier
  - Un objet (TrustManager) qui permet de décider si l'on fait confiance ou non en un certificat reçu
- Les SSLEngine (nio) sont créés directement par le SSLContext
- Les connexions SSL sont ensuite associées à des SSLSession

Usage simple

- Pour récupérer un SSLSocket ou un SSLServerSocket
  - Par un appel à getDefault() sur la Factory correspondante
  - Ces 2 méthodes statiques créent
    - Un SSLContext par défaut
    - Avec un KeyManager par défaut (et son KeyManagerFactory)
    - Un TrustManager par défaut (et son TrustManagerFactory)
    - Un RandomGenerator par défaut
  - Utilisent pour cela les fichiers par défaut de gestionnaire de clé/confiance, définis par des propriétés système
    - KeyStore/TrustStore
      - Objets de la classe javax.security
      - Nom de format de stockage (JKS, PKCS12)
      - Fournisseur d'implantation (SunJSSE)

Usage contrôlé

- Pour gérer toutes les étapes et tous les paramètres
  - Créer explicitement un SSLContext
    - javax.net.ssl.SSLContext.**getInstance**(protocole,provider)
    - Exemples:
      - Protocole: "SSL", "SSLv3", "TLS", "TLSv1"
      - Provider: "SunJSSE"
  - Puis l'initialiser
    - init(KeyManager[] km,TrustManager[] tm,SecureRandom rdm)
    - Les Key/TrustManager peuvent être créés en implantant les interfaces correspondantes de toute pièce, ou en utilisant les fabriques (Key/TrustManagerFactory), avec leurs méthodes statiques getInstance() et leurs méthodes init()
    - Celles-ci peuvent être initialisées par des KeyStore/TrustStore

Les clés et les certificats

- Nécessaire à SSL, pour déterminer une identité
  - Gérés dans des « objets de stockage des clés » (KeyStore)
  - Pour obtenir la clé publique d'un signataire à partir de son identité
    - Associe une chaîne de certificats (clé publique) à une identité, référencée par un alias (usage public)
  - Pour la signature, on a également besoin de cet objet de stockage,
    - mais alors il doit associer à l'identité, en plus du certificat, la clé privée qui permettra de signer (usage personnel)
  - Outil `keytool` permet, via la ligne de commande,
    - de créer des clés, de gérer les fichiers de stockage des clés (.keystore) et de stockage des certificats des autorités de certification (cacerts)

java.security.KeyStore (général)

- Objets construits par getInstance(format, provider)
  - Initialisé (à partir d'un fichier) par
    - load(InputStream stream,char[] password)
- Chaque entrée de l'objet de stockage est un ***alias***
  - Possibilité de récupérer les clés et certificats associés
    - Key getKey(String alias, char[] password)
    - Certificate[] getCertificateChain()
  - Possibilité de modifier l'objet de stockage
    - setKeyEntry(String alias, Key key, char[] password, Certificate[] chain)
    - setCertificateEntry(String alias, Certificate cert)
    - Sauvegarde par store(OutputStream stream, char[] password)

javax.net.ssl.KeyManager (SSL)

- Ce sont des gestionnaires de clés SSL
  - Ils gèrent des paires certificat-clé privée d'un type particulier
    - Seul type disponible en standard est **X509**
  - Doivent implanter l'interface (vide) KeyManager
    - Utilisé dans le Handshake pour l'authentification
    - chooseServerAlias() et chooseClientAlias()
    - Trouver un alias qui restectent des contraintes d'algos (RSA, DSA...)
    - Une fois un alias satisfaisant trouvé
    - getPrivateKey() et getCertificate()
- Ils sont créés part des fabriques
    - KeyManagerFactory.getInstance("SunX509").getKeyManagers()
    - Retourne 1 KeyManager par type de clé (1 seul: X509KeyManager)

javax.net.ssl.TrustManager (SSL)

- Ce sont les gestionnaires de certificats de confiance d'SSL
  - Ils gèrent des certificats de confiance d'un type particulier
    - Seul type disponible en standard est **X509**
  - Doivent implanter l'interface (vide) TrustManager
    - Utilisé dans le Handshake pour validité de date, de signature, etc.
    - checkServerTrusted() et checkClientTrusted() vérifient les chaînes de certificats
    - getAcceptedIssuers() pour déterminer les émetteurs de certificats acceptés
- Ils sont créés part des fabriques
    - TrustManagerFactory.getInstance("SunX509").getTrustManagers()
    - Retourne 1 TrustManager par type de clé (1 seul: X509TrustManager)

Les fabriques de socket SSL

- Fabriques de socket cliente SSL
  - SSLSocketFactory sf =
    (SSLSocketFactory) SSLSocketFactory.getDefault();
  - Utilisation pour créer une socket et la connecter au port (SSL par défaut) **443** du serveur "machine":
  - SSLSocket socket =
    (SSLSocket) sf.createSocket("machine",443)
- Fabriques de socket serveur SSL
  - SSLServerSocketFactory ssf =
    (SSLServerSocketFactory) SSLServerSocketFactory.getDefault()
  - Utilisation pour créer une SSLServerSocket (sur le port 4321):
  - SSLServerSocket ss =
    (SSLServerSocket) ssf.createServerSocket(4321)
- getSupportedCipherSuite() [possible] et getDefaultCipherSuite() [utilisé]

Exemple Simple Serveur

```
import java.io.*;
import javax.net.ssl.*;
// ...
int port = serverPortNumber;

SSLServerSocketFactory sslSrvFact =
  (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

SSLServerSocket s =
  (SSLServerSocket)sslSrvFact.createServerSocket(port);

SSLSocket c = (SSLSocket)s.accept();
OutputStream out = c.getOutputStream();
InputStream in = c.getInputStream();
// Send messages to the client through
// the OutputStream
// Receive messages from the client
// through the InputStream
```


Exemple Simple Client

```
import java.io.*;
import javax.net.ssl.*;
// ...
int port = serverPortNumber;
String host = "hostname";

SSLSocketFactory sslFact =
   (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket s =
   (SSLSocket)sslFact.createSocket(host, port);

OutputStream out = s.getOutputStream();
InputStream in = s.getInputStream();

// Send messages to the server through
// the OutputStream
// Receive messages from the server
// through the InputStream
```

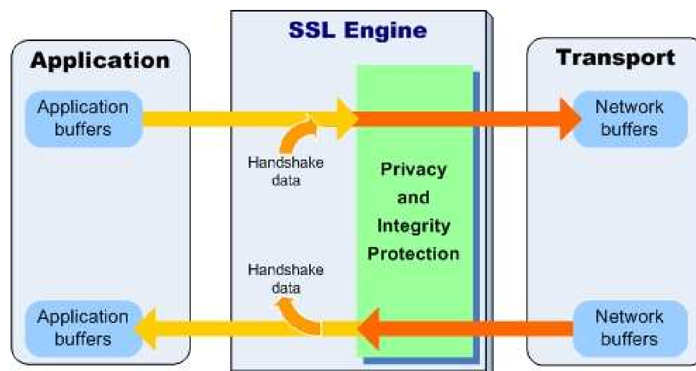
SSLSocket et SSLServerSocket

- Par héritage, elles disposent des méthodes des Sockets et ServerSockets
  - Elles ajoutent des méthodes propres à la gestion fine du protocole
- SSLSocket
  - startHandshake()
    - Par défaut, appelé initialement par le client avant tout échange. Cet appel, implicite ou explicite, est synchrone (bloquant)
    - Ensuite, peut être appelé à nouveau pour renégocier. Alors asynchrone
    - Possibilité, dans ce cas, d'être prévenu de manière asynchrone addHandshakeCompletedListener(HandshakeCompletedListener listener)
- SSLServerSocket
  - Pour demander l'authentification du client (ce n'est pas le cas par défaut), appeler setWantClientAuth(true) avant l'appel à accept().
  - Pour l'exiger, appeler setNeedClientAuth(true)
  - getWantClientAuth() et getNeedClientAuth() pour connaître la config.

SSLEngine

- Accès éventuellement non bloquant aux ressources sur une connexion SSL, mais:
  - Protocole SSL complexe
  - Problèmes d'interopérabilité avec les selecteurs
  - Volonté de ne pas fixer de choix dans les « providers »
- D'où:
  - Implémentation indépendante du modèle d'entrées/sortie et du modèle de thread
  - Doit être géré par l'application
  - Pas très facile à manipuler

SSLEngine (schéma)



SSLEngine (états)

- Un SSLEngine doit gérer simultanément les données de l'application et les données du Handshake
- Il peut être, à tout moment, dans l'un des 5 états:
  - ***Creation*** : prêt à être configuré
  - ***Initial handshaking*** : en cours d'authentification et de négociation des paramètres de communication
  - ***Application data*** : prêt à échanger des données de l'application
  - ***Rehandshaking*** : en cours de renégociation des paramètres de communication. Les données de handshake peuvent être mélangées aux données de l'application
  - ***Closure :*** prêt à fermer la connection.
- Représenté par SSLEngineResult et SSLEngineResult.HandshakeStatus
- Sur les données à envoyer: wrap(), sur les données reçues unwrap()
  - Permet de « démultiplexer » les données Application et Handshake

Négotiation SSL

- Par défaut, en SSL, c'est le serveur qui attend les messages *Hello*
  - On peut changer cet état, sur SSLSocket comme sur SSLServerSocket grâce à la méthode setUseClientMode()
  - Il est ensuite possible de demander l'authentification de l'interlocuteur (setWant..., setNeed...)
- Il est possible de paramétrer les versions du protocole et les suites d'algorithmes à utiliser
  - Spécifier avec setEnabledProtocols() et setEnabledCipherSuites() des sous-ensembles de getSupportedProtocols() et de getSupportedCipherSuites()

Sessions SSL

- Une session est associée à chaque connexion
  - 1 même session peut être partagée par plusieurs connexions
  - Par défaut, s'il n'exsite pas de session associée à une demande de connexion, une nouvelle est créée.
    - Consultation/modification par get/setEnableSessionCreate()
  - Sur l'objet SSLSocket, getSession() retourne la SSLSession associée. Il est alors possible de consulter
    - getId(): identificateur sous la forme de tableau d'octets
    - getProtocol() et getCipherSuite(): version du protocole et algorithmes cryptographiques utilisés pour cette session
    - getPeerHost() et getPeerCertificates(): le nom et les certificats (éventuels) de l'interlocuteur associé à cette session
    - getLocalCertificates(): certificats (éventuels) envoyés à l'interlocuteur pour s'authentifier
    - getCreationTime() et getLastAccessedTime(): création et dernier accès
    - invalidate(): interdit la réutilisation d'une session. Par défaut, réutilisation.

HttpsURLConnection

- Hérite de HttpURLConnection
  - La fabrique de socket SSL à utiliser peut être spécifiée
    - Globalement: setDefaultSSLSocketFactory()
    - Pour la connexion: setSSLSocketFactory()
  - HTTPS impose de comparer le nom DNS et le CN du DN
    - S'ils ne concordent pas, il est possible d'utiliser un HostnameVerifier
    - Interface avec une unique méthode
      - boolean verify(String hostname,SSLSession session)
    - Puis l'enregistrer auprès du gestionnaire de connexion
      - Globalement : setDefaultHostnameVerifier(hv)
      - Pour la connexion : setHostnameVerifier(hv)