Nouvelles technologies réseaux
SSH et TLS

Benoît Lamotte
Vincent Robert
Alexis Seigneurin
Ingénieurs 2000 – Informatique & Réseau 3

Année 2005

Table des matières

INTRODUCTION

Internet est un gigantesque réseau sur laquelle la confidentialité des échanges n'est pas assurée. Or, la plupart des protocoles de communication utilisés sont non sécurisés.

SSH (Secure Shell) et TLS (Transport Layer Security) sont deux protocoles destinés à pallier ce problème. Ils s'intercalent entre les protocoles habituels de transport et les protocoles applicatifs, et proposent ainsi une couche de sécurité pour tout type d'application.

SSH est particulièrement utilisé pour réaliser de l'administration de machines à distance. TLS, quant à lui, est plus généralement utilisé dans un contexte Web (sécurisation des échanges sur un site Web) ou mail (POP et SMTP).

Nous présenterons tout d'abord les différents problèmes de sécurités ainsi que des mécanismes cryptographiques destinés à y remédier. Dans un second temps, nous présenterons le protocole SSH. Enfin, nous présenterons le procole TLS.

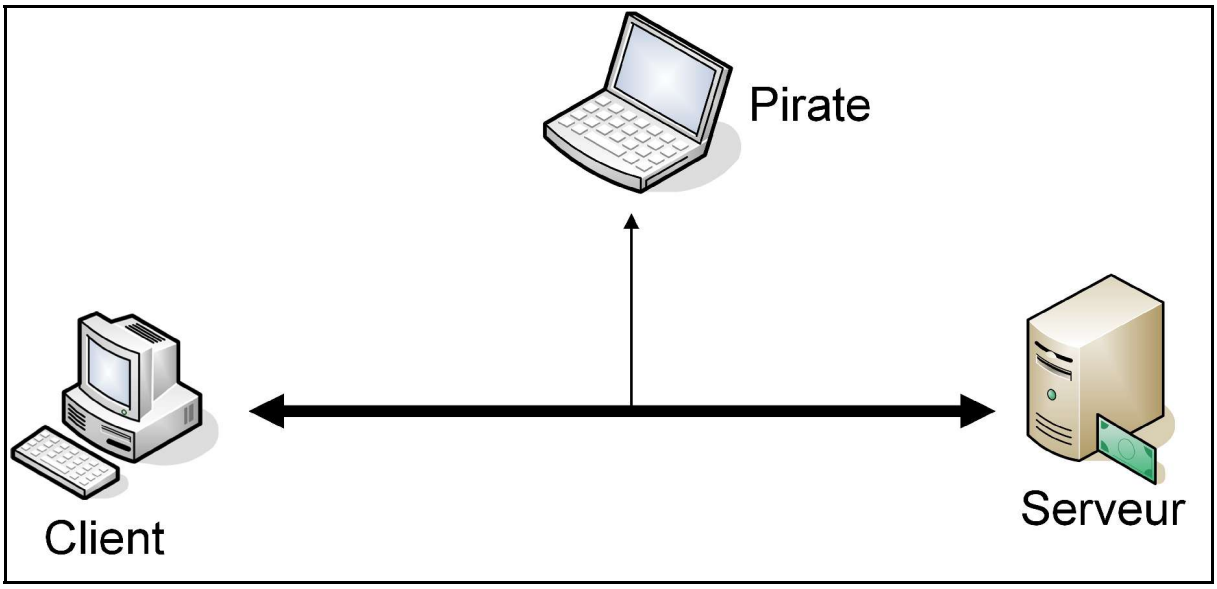
1.SÉCURISER UNE TRANSACTION

1 Quelle est l'utilité ?

Lorsque vous surfez sur Internet, la plupart du temps vous ne faites que de la consultation d'information publique (site d'information, etc...). Il n'est pas nécessaire de sécuriser ce type de dialogue, mais alors que faut il sécuriser ?

Il faut tout simplement sécuriser toutes les informations n'étant pas publiques mais personnelles (à une personne ou un groupe de personnes).

Prenons pour exemple les transactions commerciales. Pour valider votre achat vous allez envoyer vos informations bancaires (numéro de carte de crédit, etc.) au site de vente en ligne. Si cette transaction n'est pas sécurisée, voilà ce qu'il peut se passer :



Lorsque vous envoyez ces informations au serveur du site de vente, une personne malintentionnée (pirate) peut écouter votre envoi et récupérer vos informations pour les utiliser ensuite à votre insu. Ceci est très facile à réaliser à l'aide d'outils appelés « sniffeurs » permettant de récupérer les trames circulant sur un réseau donné. Il ne reste plus ensuite qu'à analyser les trames pour récupérer les informations désirées.

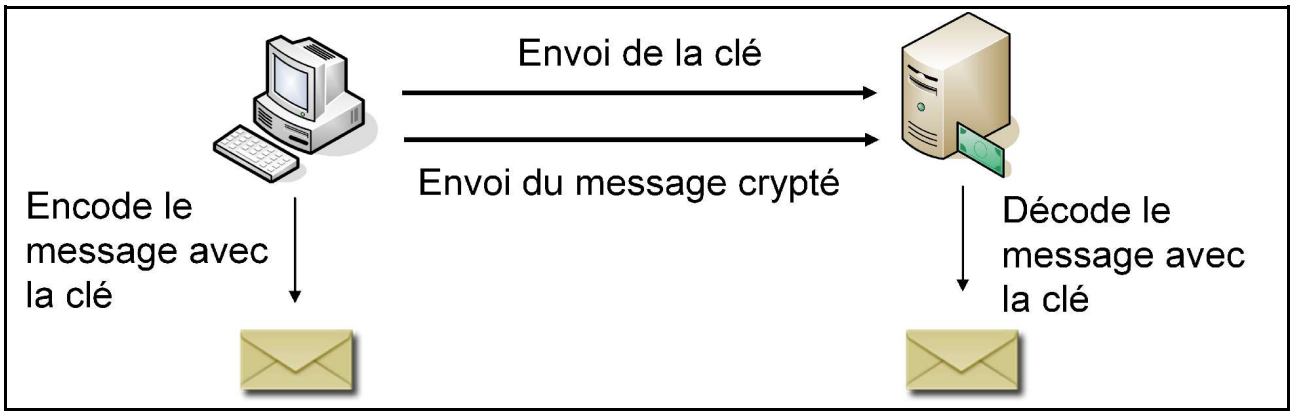
Il faut donc que votre message soit crypté pour que seul le serveur puisse comprendre votre message. Nous allons donc nous intéresser à des méthodes de cryptage de l'information utilisées dans les protocoles SSH, SSL et TLS.

2 Chiffrement symétrique

Le principe de chiffrement symétrique (aussi appelé *chiffrement à clé privée* ou *chiffrement à clé secrète)* utilise une seule clé de cryptage pour encoder le message et pour le décoder.

Cette méthode fonctionne en 5 étapes :

- L'expéditeur génère une clé symétrique (alogrithme DES, tripleDES...).
- L'expéditeur encode son message avec la clé symétrique.
- L'expéditeur envoi la clé.
- L'expéditeur envoi le message crypté.
- Le destinataire n'a plus qu'a décrypter le message avec la clé reçue.



Cette méthode est assez simple et permet d'envoyer un message crypté.

Cependant, elle possède quelques inconvénients, Claude Shannon à démontré dans les années 80 que pour que ce système soit totalement sûr qu'il fallait que la longueur de la clé générée soit au moins égale à la longueur du message à encoder.

De plus la la clé symétrique doit être envoyée au destinataire par un canal sécurisé. En reprenant l'exemple du pirate qui écoute sur le réseau pour récupérer les informations, avec cette méthode il récupérerait la clé symétrique et donc le message décrypté. Si ce message contenait des informations vous identifiant sur le destinataire (login, mot de passe), il pourrait ensuite se faire passer pour vous et accéder à vos informations.

3 Chiffrement asymétrique

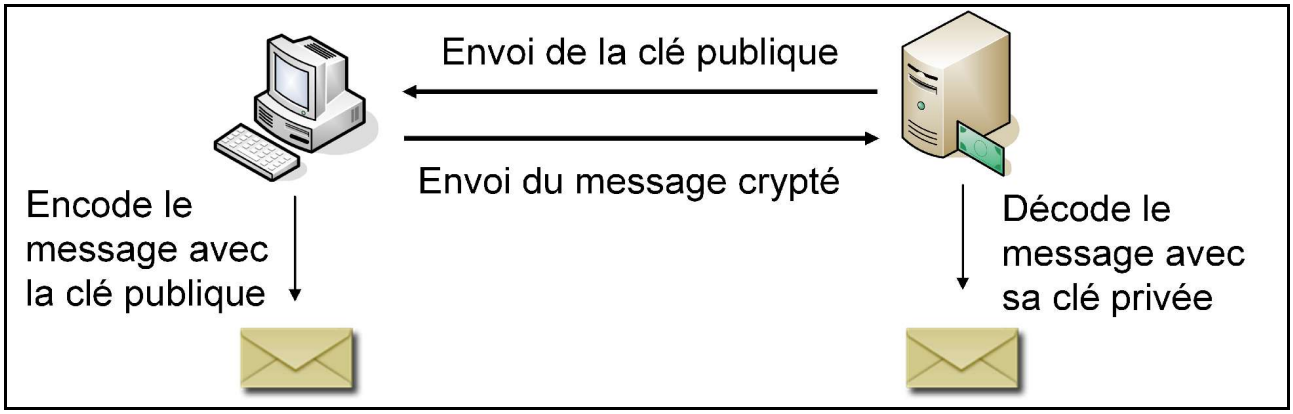
Contrairement au chiffrement symétrique cette méthode utilise deux clés, une clé publique pour l'encodage et une clé privée pour le décodage.

La clé privée est connue seulement par le destinataire. Toutes personnes voulant lui

envoyer le message récupère la clef publique pour encoder le message (disponible généralement dans un serveur de clé, de type LDAP généralement).

Il y a donc 4 étapes à cette méthode :

- L'expéditeur récupère la clé publique.
- L'expéditeur encode le message à l'aide de cette clé.
- Il envoie le message au destinataire.
- Le destinataire décode le message avec la clé privée que lui seul possède.



Ce principe repose sur des fonction faciles à calculer dans un sens, mais qui est très difficile mathématiquement à calculer dans l'autre sens sans posséder la clé privé (algorithme RSA). Cette méthode est extrêmement coûteuse en ressource lors des calcul d'encodage et de décodage du message.

Ainsi notre pirate ne pourra pas décrypter le message qu'il aura récupéré en écoutant le canal car il ne possédera pas la clé privée.

Cependant pour que ce système soit sûr il faut s'assurer que la clé publique l'on possède appartient bien au destinataire du message (voir attaque « Man in the middle »).

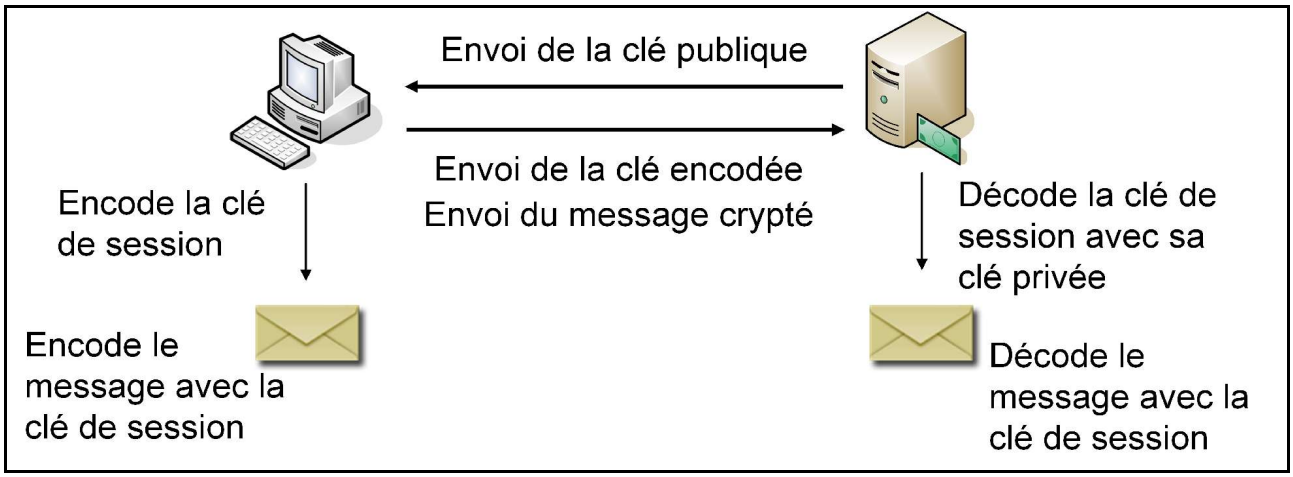
4 La combinaison des deux

Il existe une méthode utilisant à la fois le chiffrement symétrique et le chiffrement asymétrique. Cette méthode est appelée clé de session.

Comme dans le chiffrement asymétrique, le destinataire génère une clé privé ainsi qu'une clé publique. Cette dernière est transmise aux expéditeurs.

De son côté, l'expéditeur génère une clé symétrique (algorithme DES, tripleDES...) qu'il encode à l'aide de la clé publique du destinataire.

Il transmet ensuite sa clé crypté. Ainsi, les deux protagonistes possèdent tous les deux une clé qui leur est propre. Il peuvent donc ensuite communiquer à l'aide de cette clé de façon sécurisée.



L'échange de clé symétrique ne se fait donc qu'une seule fois, limitant le nombre de cryptage et décryptage de type asymétrique (coûteux en temps processeur).

La clé échangée est valable seulement pour une session, il faudra donc, à la prochaine connexion, établir une nouvelle clé symétrique pour évoluer de nouveau dans un système sécurisé.

Comme pour la méthode asymétrique, il faut être sûr que la clé publique utilisée appartient bien au destinataire voulu.

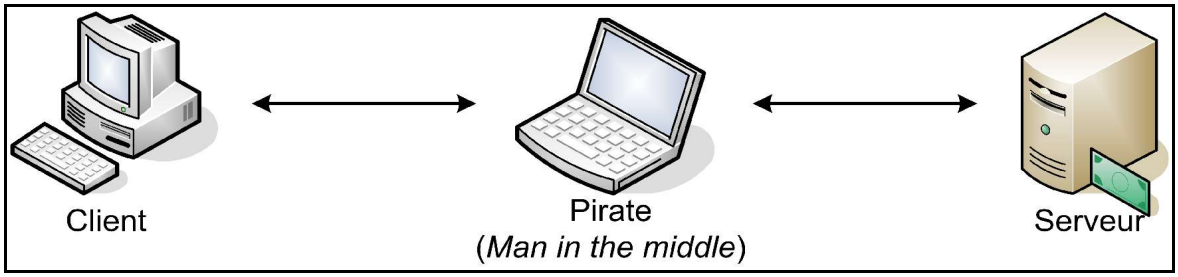
5 *Attaque* « Man in the middle »

Les méthodes de chiffrement asymétrique et symétrique permettent d'évoluer dans un système sécurisé empêchant les logiciels d'écoute de récupérer le message et de le décrypter.

Cependant, la technique du « Man in the middle » (attaque du milieu, ou attaque de l'intercepteur en Français) permet de contourner cette sécurité.

Le principe est simple même si il n'est pas toujours évident à mettre en place.

Le pirate se place sur le réseau entre les deux machines (expéditeur et destinataire) pour intercepter et modifier les messages qu'elles s'échangent.

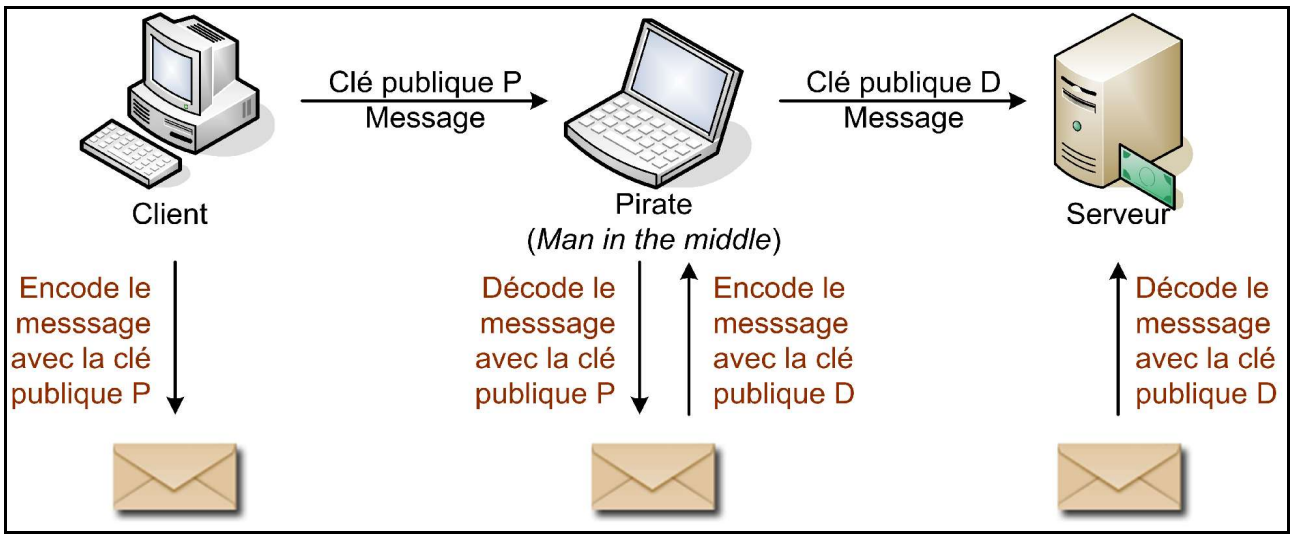


Nous allons prendre l'exemple de la méthode asymétrique pour illustrer ce système, mais il fonctionne de la même façon avec les autres méthodes. Son but étant de se faire passer pour le destinataire auprès de l'expéditeur et pour l'expéditeur auprès du destinataire.

Dans le système asymétrique, le destinataire envoyez sa clé publique à l'expéditeur. Le pirate la récupère et transfert sa propre clé publique à l'expéditeur.

L'expéditeur envoie ensuite son message encodé avec la clé publique du pirate. Le pirate décode le message, il récupère ainsi les informations de l'expéditeur (login et mot de passe par exemple). Il ne lui reste plus qu'a encoder ces informations avec la véritable clé publique du destinataire et de lui transmettre le message.

Le pirate est ainsi connecté sur le destinataire avec les informations de l'expéditeur sans que l'expéditeur et le destinataire ne soient au courant de la substitution.



6 Les Certificats

La méthode « Man in the middle » est possible car l'expéditeur à bien voulu utiliser la clef publique du pirate au lieu de celle du destinataire.

Les certificats permettent de s'assurer que la clé publique récupérée appartient bien au destinataire souhaité.

Les certificats sont comme une pièce d'identité prouvant que la clé appartient à telle ou telle personne, organisme ou autre.

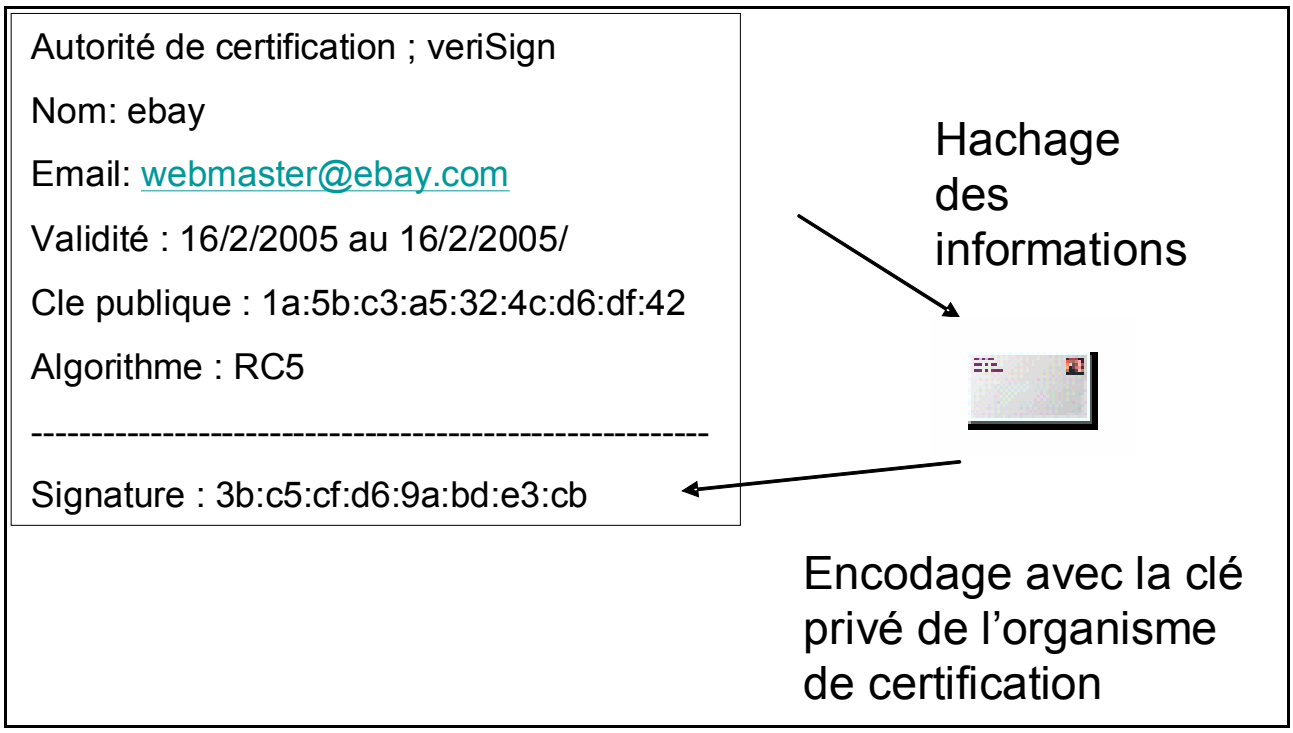
Lorsque vous acheter sur un site de vente en ligne et que vous vous apprêtez à envoyer vos coordonnées, vous devez être sûr que le destinataire est bien le site de vente.

Pour cela, il suffit de vérifier son certificat. Un certificat est composé de deux parties :

- les informations
- la signature.

Le site de vente en ligne, pour assurer ses clients que la clé publique qu'il utilise appartient bien au site, a envoyer ses informations et sa clé publique à un organisme de certification (VeriSign, par exemple).

L'organisme vérifie que les informations soumises sont correcte puis ajoute à celle-ci ses propres informations. Ensuite, il ajoute une signature au certificat. Cette signature est un hachage du résumé des informations encodé avec la clé privée de l'organisme de certification.



Lorsque l'expéditeur récupère le certificat il n'a plus qu'a calculer le hachage du résumé des informations, et à décoder la signature avec la clé publique de l'organisme de certification. Ensuite il n'a plus qu'a comparer les deux résultats pour savoir si le certificat est un vrai. Si oui, les informations sont donc exactes.

Cette méthode permet d'éviter d'encoder son message avec une clé publique inconnue et donc de révéler ses informations à des personnes malveillantes.

2.SSH : SECURE SHELL

1 SSH version 1

Ce protocole est la première version du protocole Secure Shell. Il existe aujourd'hui une version 2 corrigeant une faille de sécurité de ce protocole. Bien que SSH2 soit encore à l'état de brouillon (draft), l'IETF recommande de ne plus utiliser SSH1.

Nous allons tout de même présenter rapidement ce protocole afin de connaître la base qui a donné naissance à SSH2.

1 Description du protocole

La connexion s'effectue avec TCP/IP selon la spécification. Il serait possible d'effectuer cette connexion sur d'autres protocoles mais cela n'est pas prévu par la RFC.

Au début de la connexion, le serveur envoie une chaîne d'authentification puis le client répond par sa chaîne d'identification. Cette chaîne contient le port de connexion, la version du protocole ainsi que la version du logiciel pour des raisons de développement. Si le client ou le serveur n'accepte pas cette chaîne, la connexion est fermée.

Le serveur commence par envoyer sa clef RSA (cette clef est regénérée toutes les heures) ainsi que d'autres informations. Le client génère alors une clef de session de 256 bits et la crypte en utilisant les 2 clefs RSA (client et serveur), puis l'envoie au serveur avec le type de cryptage sélectionné. Les 2 parties activent alors le cryptage des données en utilisant le type de cryptage et la clef de session spécifiés par le client. Le serveur envoie alors un message de confirmation crypté au client.

2 Structure des paquets

<table>
<tr><th>0 - 7</th><th>8 - 15</th><th>16 - 23</th><th>24 - 31</th></tr>
<tr><td colspan="4">Taille du paquet</td></tr>
<tr><td colspan="2">Bourrage</td><td>Type de paquet</td><td>Données...</td></tr>
<tr><td colspan="4">...</td></tr>
<tr><td colspan="4">CRC32</td></tr>
</table>

La `taille du paquet` doit être codé en big endian.

Le `bourrage` est une suite d'entre 1 et 8 octets dont la valeur est aléatoire (ou 0 s'il n'y a pas de cryptage)

Le `type de paquet` est un octet non signé, la valeur 255 est réservée aux extensions futures

Le CRC32 utilise le polynôme `0xedb88320`. Il est calculé à partir du bourrage, du type de paquet et des données avant tout cryptage.

2 SSH version 2

C'est un protocole en cours de développement par l'IETF. Ce protocole est encore à l'état de brouillon (draft) mais a déjà majoritairement remplacé SSH1 chez les utilisateurs du fait des problèmes de sécurité que peut poser ce protocole.

Le protocole SSH2 se découpe en 3 niveaux. Un premier niveau transport chargé d'établir la communication entre le client et le serveur. C'est ce niveau qui est responsable de la manipulation des paquets et donc de leur cryptage et du mécanisme d'intégrité des données.

Un second niveau est responsable de l'authentification des utilisateurs. Il se base sur le protocole de transport qui est chargé d'établir une communication sécurisé entre le client et le serveur en ayant effectuer une authentification des machines.

Un troisième niveau est responsable de la gestion des connexions pouvant circuler via SSH, notamment le transfert de port.

1 Transport

Il est recommandé dans la spécification de transmettre SSH2 via un protocole possédant un contrôle d'erreur. De telle erreurs sont en effet détectées par le protocole SSH et impliquent une coupure immédiate de la connexion. Il est donc préférable d'utiliser le contrôle d'erreur du protocole de transport afin de garantir les connexions.

Le protocole principalement utilisé est TCP/IP. Avec ce protocole de transport, le serveur écoute généralement sur le port 22 qui a été officiellement assigné à SSH par l'IANA.

I. Initialisation de la connexion

Quand la connexion a été établie, les deux parties doivent envoyer une chaîne d'identification. Cette chaîne doit être :

`SSH-Protocole-Logiciel [<SP> Commentaire] <CR> <LF>`

`Protocole` doit contenir la version du protocole. Pour SSH2, cette chaîne doit être `2.0`.

`Logiciel` doit contenir la version du logiciel. Les numéros de version doivent être composés de caractères `US-ASCII` à l'exception des caractères d'espacement et du caractère `moins (-)`.

`Commentaire` est optionnel. Si ce champs est omis, l'espace le séparant des champs précédents peut être omis aussi. Il ne doit pas contenir le caractère `NULL (0)`.

La taille maximum de la cette chaîne est 255 caractères y compris le retour chariot et le caractère de nouvelle ligne de fin.

Il est possible d'envoyer des lignes supplémentaires avant d'envoyer la chaîne d'identification. Ces lignes ne doivent pas commencer par `SSH-` et doivent être gérées par le client. Cependant ce dernier peut simplement les ignorer ou les afficher à l'utilisateur.

II. Compatibilité

Un client SSH1 peut se connecter à un serveur SSH2.

Il est possible pour un serveur de supporter les 2 version de SSH, il devra alors envoyer la chaîne `1.99` comme version de protocole. Un client SSH2 devra interpréter cette version comme étant égale à `2.0`.

Un serveur compatible avec une ancienne version ne devra pas envoyer le caractère nouvelle ligne lors de l'envoi de la chaîne d'identification, et devra attendre un envoi du client avant tout envoi afin d'identifier la version avec laquelle le client discute.

Un serveur SSH2 seulement ne peut pas accepter de connexions de client SSH1 et doit les rejeter automatiquement.

III. Structure d'un paquet

0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63
Longueur paquet				Longueur bourrage	Données...		
...					Bourrage aléatoire		
Code d'authentification du message							

La `longueur du paquet` n'inclut pas le `code d'authentification du message` ni le champs de `longueur du paquet` lui-même.

Les `données` peuvent utiliser un système de compression. Cependant, initialement, il n'y a aucune compression.

Le `bourrage aléatoire` est une série de données choisies aléatoirement. Leur longueur doit être un multiple de la longueur du code de cryptage ou de 8, en prenant le plus grand. La longueur du bourrage ne peut pas dépasser 255 octets.

Le code d'authentification du message est le code négocié afin de crypter les messages. Initialement, le cryptage est désactivé.

Un paquet ne peut pas dépasser 35000 octets. Cette valeur a été arbitrairement choisi dans la spécification de SSH2. Il est possible d'utiliser des paquets plus grands si besoin, mais les parties doivent annoncer qu'elles le supportent en passant par le système d'extensions.

IV. Compression

La compression s'effectue exclusivement sur les données. Elle se fait en utilisant l'algorithme de compression qui a été négocié entre le serveur et le client. Il est possible de choisir une méthode de compression différente pour les 2 sens de la communication, mais la spécification recommande d'utiliser la même méthode de compression pour toute la communication.

Voici les méthodes de compression définies par la spécification :

- aucune, cet algorithme est forcément requis ;
- zlib, la compression bien connue, cet algorithme est optionnel.

V. Cryptage

Le cryptage se fait au moyen d'un algorithme de cryptage négocié lors de l'échange des clefs entre le client et le serveur. Lorsque le cryptage est effectif, toutes les données d'un paquet sont cryptées sauf le code d'authentification du message. Le cryptage se fait toujours après la compression.

Lors d'une communication, les 2 directions doivent utiliser des algorithmes indépendants. Les client et serveur doivent d'ailleurs permettre de choisir le type d'algorithme pour chaque direction de la communication. Cependant, la spécification recommande d'utiliser le même algorithme dans les 2 directions.

Les algorithmes suivants sont définis par la spécifications :

Nom	Description	Requis ?
3des-cbc	3 clefs 3DES en mode CBC Cet algorithme utilise des clefs de 112 bits seulement mais est requis car c'est le plus répandu dans le monde de la cryptographie	requis
blowfish-cbc	Blowfish en mode CBC	
twofish256-cbc	Twofish en mode CBC avec clef de 256 bits	
twofish-cbc	Alias de twofish256-cbc conservé pour des raisons de compatibilité	
twofish192-cbc	Twofish en mode CBC avec clef de 192 bits	
twofish128-cbc	Twofish en mode CBC avec clef de 128 bits	
aes256-cbc	AES en mode CBC avec clef de 256 bits	
aes192-cbc	AES en mode CBC avec clef de 192 bits	
aes128-cbc	AES en mode CBC avec clef de 128 bits	recommandé
serpent256-cbc	Serpent en mode CBC avec clef de 256 bits	
serpent192-cbc	Serpent en mode CBC avec clef de 192 bits	
serpent128-cbc	Serpent en mode CBC avec clef de 128 bits	
arcfour	Le cryptage de flux ARCFOUR	
idea-cbc	IDEA en mode CBC	
cast128-cbc	CAST-128 en mode CBC	
none	Aucun cryptage - Déconseillé	

VI.Intégrité des données

Chaque hôte participant à la communication doit effectuer un comptage des paquets de manière secrète. Ce comptage permet ensuite de calculer le code d'authentification de chaque message (ou Message Authentication Code : MAC) en utilisant l'algorithme MAC négocier en début de communication.

La spécification propose plusieurs algorithmes MAC :

Nom	Description	Requis ?
hmac-sha1	HMAC-SHA1	requis
hmac-sha1-96	96 premiers bits de HMAC-SHA1	recommandé
hmac-md5	HMAC-MD5	
hmac-md5-96	96 premiers bits de HMAC-MD5	
none	Aucun algorithme, déconseillé	

VII.Échange de clefs

Lors de la connexion, les 2 parties doivent négocier un algorithme de génération des clefs de sessions qui serviront pour le cryptage et l'authentification auprès du serveur.

La spécification définit 2 méthodes d'échanges des clefs que les logiciels doivent posséder : `diffie-hellman-group1-sha1` et `diffie-hellman-group14-sha1`.

VIII.Algorithmes des clefs publiques

Le protocole a été conçu pour travailler avec de nombreux formats de clef publique. Il y a plusieurs aspects qui définissent une clef publique :

- le format de clef, c'est à dire comment la clef est encodé et comment les certificats sont représentés ;
- les algorithmes de signature et/ou cryptage, certains types de clefs peuvent ne pas supporter à la fois la signature et le cryptage ;
- codage des signatures et/ou des données cryptées.

La spécification définit les formats de clefs publiques et/ou de certificats suivants :

Nom	Description	Requis ?
ssh-dss	Clef DSS brut	requis
ssh-rsa	Clef RSA brut	recommandé
spki-sign-rsa	Certificats SPKI avec clef RSA	
spki-sign-dss	Certificats SPKI avec clef DSS	
pgp-sign-rsa	Certificats OpenPGP avec clef RSA	
pgp-sign-dss	Certificats OpenPGP avec clef DSS	

Pour s'échanger les clefs publiques, chaque machine va annoncer les algorithmes de clefs qu'il supporte. Une fois les méthodes annoncées, les 2 machines vont choisir une méthode d'échange des clefs et s'envoyer leurs clefs publiques respectives.

2 Authentification

Le protocole d'authentification SSH permet l'authentification des utilisateurs en utilisant le protocole de transport SSH définit plus haut.

IX.Échange initial

L'authentification est déclenché par le client avec l'envoi d'un paquet `SSH_MSG_USERAUTH_REQUEST`.

La `langue` est déprécié et doit être la chaîne vide.

Les sous-méthodes est une liste séparée par des virgules de sous-méthodes d'authentification. Il est notamment possible pour le client de spécifier un certains nombres de sous-méthodes à utiliser basé sur une configuration de l'utilisateur.

Quand ce message est envoyé au serveur, l'utilisateur n'a pas encore spécifié de mot de passe de connexion et le mot de passe n'est pas encore inclue dans ce message (contrairement à la méthode `password`).

À la réception de ce message, le serveur doit renvoyer un des messages suivants :

- `SSH_MSG_USERAUTH_SUCCESS`, pour accepter l'authentification ;
- `SSH_MSG_USERAUTH_FAILURE`, pour refuser l'authentification ;
- `SSH_MSG_USERAUTH_INFO_REQUEST`, pour demander des informations supplémentaires afin de poursuivre l'authentification.

La spécification ne conseille pas de répondre le message `SSH_MSG_USERAUTH_FAILURE` au premier message d'authentification par nom d'utilisateur. Il est préférable de répondre la réponse la plus probable (`SSH_MSG_USERAUTH_INFO_REQUEST`), puis d'envoyer un message `SSH_MSG_USERAUTH_FAILURE` après quelques secondes. Ce comportement permet d'empêcher les algorithmes « brute force » qui validerait un nom d'utilisateur par un simple message envoyé au serveur.

De même, il est préférable de ne pas répondre le message `SSH_MSG_USERAUTH_SUCCESS` même si l'utilisateur est correctement authentifié, mais d'envoyer le message `SSH_MSG_USERAUTH_INFO_REQUEST` et de ne pas tenir compte des réponses.

X. Demandes d'informations

Comme nous l'avons vu, le serveur peut demander des informations d'authentification au client avec un message `SSH_MSG_USERAUTH_INFO_REQUEST`.

1 octet	`SSH_MSG_USERAUTH_REQUEST`
chaîne	Nom de l'utilisateur
chaîne	Nom de service
chaîne	`"keyboard-interactive"`
chaîne	Langue
chaîne	Sous-méthodes

Un serveur ne doit jamais envoyer plusieurs requêtes au client en parallèle mais doit attendre la réponse du client. Cependant, le nombre de requêtes est inconnu et le client doit s'attendre à en recevoir un nombre indéfini.

Le message définit un tableau de valeur à récupérer avec pour chaque valeur : un prompt à afficher à l'utilisateur et un booléen précisant si la réponse de l'utilisateur doit s'afficher. C'est au client de définir cette notion.

Une fois toutes les valeurs renseignées, le client répond alors avec un message `SSH_MSG_USERAUTH_INFO_RESPONSE`.

Ce message contient un tableau des réponses du client à chacune des requêtes du serveur.

Le nombre de réponses dans ce message doit être égale au nombre de requêtes dans le message du serveur et doivent être disposées dans le même ordre que les requêtes respectives dans le message du serveur.

À ces requêtes, le serveur doit répondre immédiatement par une des 3 réponses possibles : succès, échec ou demande d'informations supplémentaires, notamment en cas de succès.

En cas d'échec, la spécification recommande de n'envoyer le message d'échec qu'au bout de quelques secondes d'attentes, généralement 2 secondes.

XI.Protocole de connexion

SSH2 définit un 3e protocole s'utilisant au dessus de protocole d'authentification. Ce protocole permet l'exécution de commandes à distance, le transfert de connexions TCP/IP et de connexions X11.

Toutes ces fonctionnalités passent par la création de canaux. Ces canaux peuvent être initiés par l'une ou l'autre des parties de la communication mise en place par un échange de messages.

1 octet	SSH_MSG_USERAUTH_INFO_REQUEST
chaîne	Nom
chaîne	Instruction
chaîne	Langue (déprécié)
entier	Nombre de requêtes
chaîne	Prompt 1
booléen	Affichage 1
...	

1 octet	SSH_MSG_USERAUTH_INFO_RESPONSE
entier	Nombre de réponses
chaîne	Réponse 1
...	

3.TLS : TRANSPORT LAYER SECURITY

1 Description de TLS

TLS est un protocole permettant de sécuriser des communications. Il s'appuie sur un protocole de transport fiable (TCP, par exemple). TLS est indépendant des protocoles de la couche supérieure et peut ainsi sécuriser tout type de trafic : HTTP, FTP, etc.

TLS est le prolongement de SSL (*Secure Socket Layer*). SSL a initialement été développé par Netscape et a fait l'objet de trois versions. La version 1.0, publiée en juillet 1994, n'a pas été utilisée. Elle a été suivie de la version 2.0 en novembre de la même année, puis de la version 3.0 en août 1996. Ces deux versions proposent sensiblement les mêmes fonctionnalités.

L'IETF (*Internet Engineering Task Force*) a poursuivi les travaux de Netscape sur base de SSL 3.0 et a publié TLS 1.0 (considéré comme étant SSL 3.1). TLS 1.0 fait l'objet de la *RFC 2246* publiée en janvier 1999. Notons que l'IETF standardise le protocole mais ne standardise pas l'API (*Application Programming Interface*, interface de programmation d'applications) offerte par les implémentations.

Sécurité apportée par TLS

La sécurisation porte sur plusieurs points. Tout d'abord, le protocole assure que **la connexion est privée**, c'est-à-dire qu'un tiers ne peut pas récupérer les données (*eavesdropping*). Pour cela, **les données transmises sont cryptées.**

Le protocole permet également de s'assurer de l'identité des acteurs (**authentification du client et du serveur**) grâce à l'échange de certificats signés par une autorité de certification. Cela permet également d'assurer la non-répudiation des échanges (l'émetteur ne peut pas nier avoir envoyé les données).

La **protection contre l'altération des données** est quant à elle assurée grâce à l'utilisation d'un code d'authenticité des messages (code MAC obtenu par hachage). Enfin, le « rejeu » des communications est évité grâce à l'introduction de paramètres échangés à l'ouverture de session dans le calcul de ce code.

Technologies utilisées

TLS utilise un chiffrement symétrique (chiffrement à clé secrète) car cela est moins coûteux qu'un chiffrement à clé publique du point de vue du temps de calcul. La clé secrète est quant à elle calculée à partir d'une « pré-clé secrète », laquelle est échangée grâce à un chiffrement à clé publique.

De nombreux algorithmes de hachage, de chiffrement et de compression peuvent être utilisés. En effet, TLS est modulaire afin de permettre l'utilisation de nouveaux algorithmes sans devoir standardiser un nouveau protocole. Une négociation entre le client et le serveur est alors opérée afin de déterminer les algorithmes qui seront utilisés.

Les algorithmes les plus couramment utilisés sont :

- MD5 et SHA-1 pour le hachage ;
- RSA, Diffie-Hellman pour le chiffrement de la pré-clé secrète ;
- 3DES, RC4 et DES pour le chiffrement symétrique.

2 Le protocole de communication : TLS Record Protocol

Le protocole TLS est conçu de telle sorte à pouvoir sécuriser des données pour tout type d'application. Dans le modèle en couches OSI, TLS définit la couche *TLS Record Protocol*. Celle-ci s'appuie sur un protocole de transport fiable et offre ses services à une couche supérieure.

TLS Record Protocol assure que la connexion est privée, et peut également assurer l'identité des interlocuteurs ainsi que l'intégrité et la compression des données.

Les fonctions suivantes sont assurées lors de l'émission de messages :

- la fragmentation des données en blocs ;
- la compression (optionnel) ;
- le calcul d'un code d'authenticité des données (optionnel) ;
- le cryptage.

À la réception, les fonctions symétriques sont assurées :

- le décryptage ;
- la vérification à l'aide du code d'authenticité ;
- la décompression ;
- le réassemblage des paquets fragmentés.

TLS Record Protocol se subdivise en quatre protocoles, lesquels n'interviennent pas aux mêmes moments dans la communication :

- le protocole ***Handshake protocol*** initie la connexion et permet la négociation ainsi que l'échange des paramètres de connexion ;
- le protocole ***Change cipher spec protocol*** indique à l'autre interlocuteur le passage en mode chiffré ;
- le protocole ***Application data protocol*** transmet les données chiffrées ;
- le protocole ***Alert protocol*** assure la fermeture sécurisée de la connexion.

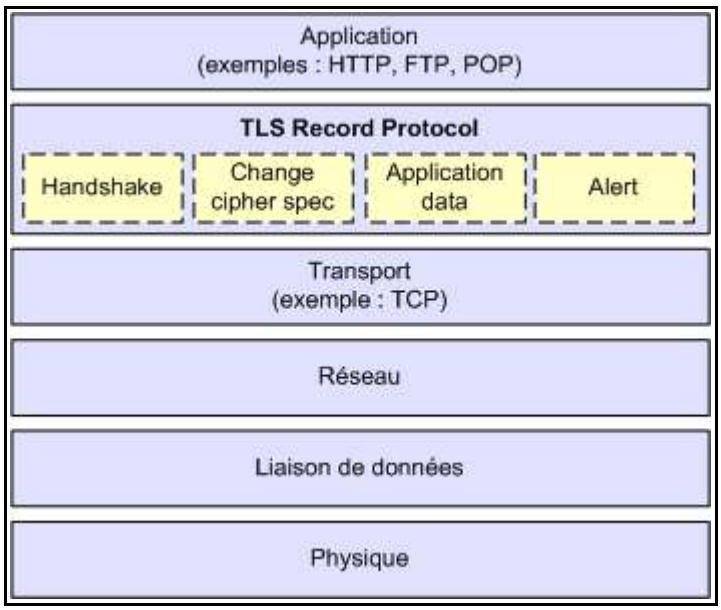


Illustration 1 : TLS dans le modèle en couches OSI.

1 Déroulement typique d'une connexion

Une connexion typique comporte trois phases :

- l'initialisation de la connexion à l'aide des protocoles *Handshake et Change cipher spec* ;
- l'échange de données à l'aide du protocole *Application data* ;
- la fermeture de la connexion à l'aide du protocole *Alert*.

Ce déroulement est décrit par l'illustration 2 et est celui qui est obtenu, par exemple, lorsque l'on se connecte au site d'une banque pour gérer ses comptes en ligne. Les différents types de messages sont décrits ci-après.

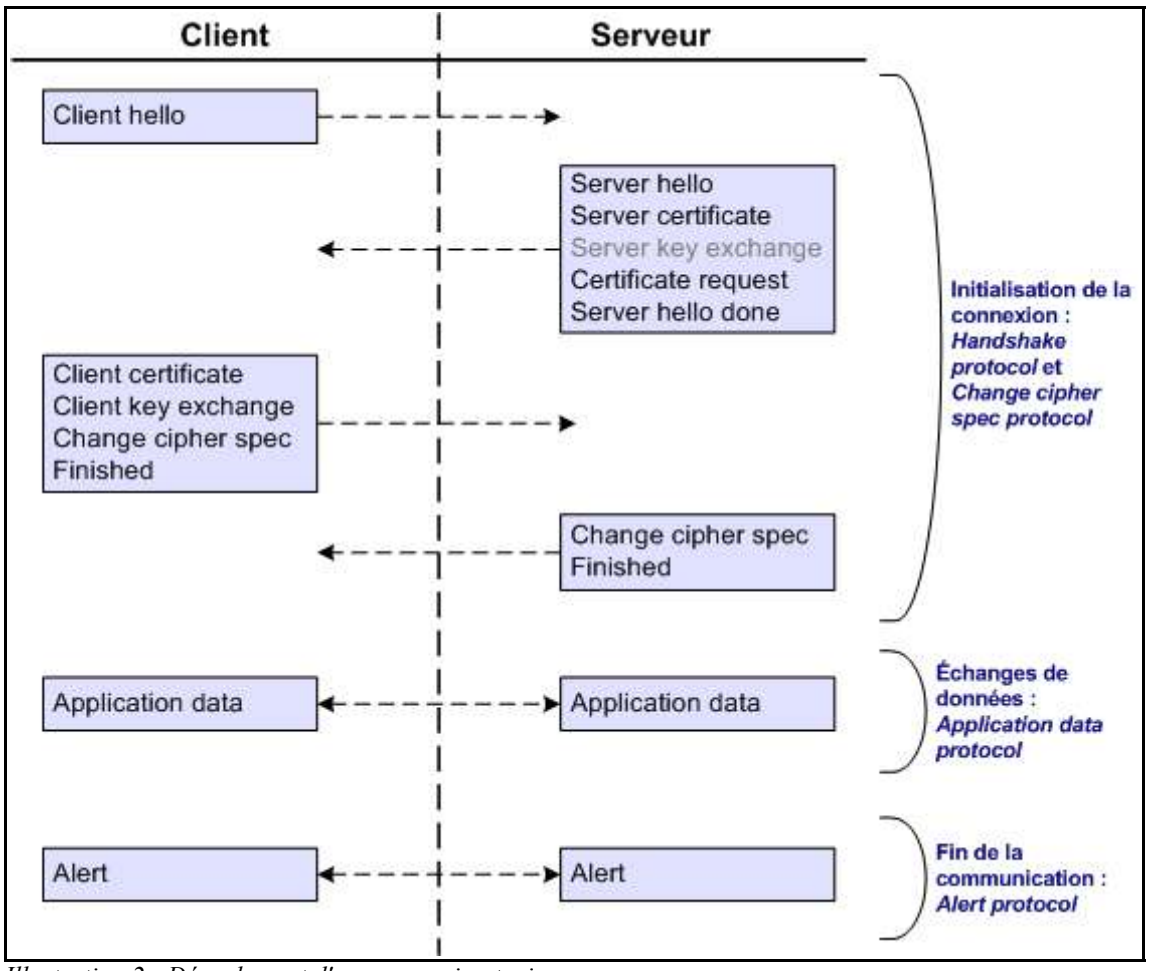


Illustration 2 : Déroulement d'une connexion typique

2 Paramétrage de la connexion : *Handshake protocol* et *Change cipher spec protocol*

Le protocole de *Handshake* est responsable de l'initialisation de la connexion. A l'issue de cette étape, les deux partis auront choisi des paramètres qui seront utilisés tout au long de la communication :

- un vecteur d'initialisation (voir la partie traitant du code d'authenticité des messages) ;
- un algorithme de chiffrement ;
- un algorithme de compression (optionnel) ;
- un algorithme de vérification de l'authenticité des messages (optionnel).

L'initialisation se déroule de la manière suivante :

- le client envoie un message *Client hello* indiquant les protocoles (chiffrement...) qu'il supporte ;
- le serveur répond par un message *Server hello* indiquant les protocoles choisis et attribuant un numéro de session à la communication. Il y joint son certificat (*Server certificate*) et fait éventuellement une demande de certificat au client (*Certificate request*). Il termine par un message *Server hello done* ;
- le client vérifie l'identité du serveur à l'aide du certificat ;

- le client envoie ensuite son certificat (*Client certificate*, si le serveur le lui a demandé) et envoie également une pré-clé secrète (*Client key exchange*). Cette clé va servir à calculer la clé secrète (clé de session) qui sera utilisée pour chiffrer les données ;
- le client et le serveur calculent la clé de session à partir de la pré-clé secrète ;
- le client annonce le passage en mode chiffré (*Change cipher spec* suivi de *Finished*) ;
- le serveur annonce également le passage en mode chiffré.

Notons que les paramètres de connexion peuvent être renégociés à tout moment par l'un ou l'autre des interlocuteurs par l'envoi d'un message de *Hello*.

***Message* Client hello**

Lorsqu'un client souhaite établir une connexion sécurisée avec un serveur, il lui envoie un message de type *Client hello* qui contient les informations suivantes :

- des données aléatoires qui devront être utilisées par une fonction pseudo-aléatoire ;
- un identifiant de session (*Session ID*) permettant d'indiquer une session dont on souhaite réutiliser les paramètres. Cet identifiant de session est mis à zéro la première fois. Lors d'une connexion par mot de passe sur un site Web, par exemple, on réutilisera l'identifiant de session pour ne pas avoir à saisir le mot de passe à chaque connexion. Il n'y aura alors pas d'échange de clés (la clé de session précédente sera réutilisée).
- la liste des suites d'algorithmes de chiffrement supportés par le client (voir ci-après) ;
- la liste des méthodes de compression supportées par le client.

Chaque suite d'algorithmes annoncées par le client définit trois algorithmes :

- un algorithme d'échanges des clés privées ;
- un algorithme de chiffrement par clés privées ;
- un algorithme de calcul de code d'authenticité des messages (MAC).

Les suites d'algorithmes sont standardisées et on associe une valeur ainsi qu'une description à chacune. Par exemple, la suite `TLS_RSA_WITH_RC4_128_SHA` définit que l'algorithme d'échange de clés est RSA, que l'algorithme de chiffrement est RC4 avec une clé privée de 128 bits, et que l'algorithme de MAC est SHA.

L'annonce des suites d'algorithmes supportées permet au client et au serveur de s'entendre sur les algorithmes à utiliser. Ainsi, le serveur devra choisir la première suite d'algorithmes qu'il supporte parmi les suites annoncées par le client. Le choix d'un algorithme de compression est réalisé de la même manière.

***Message* Server hello**

Le message *Client hello* est directement suivi d'un message *Server hello* émis par le serveur. Ce message a pour but d'annoncer au client les informations choisies par le

serveur. Il contient les informations suivantes :

- des données aléatoires pour la fonction pseudo-aléatoire ;
- l'identifiant de session (*Session ID*) qui sera utilisé ;
- la suite d'algorithmes de chiffrement choisie par le serveur parmi celles supportées par le client ;
- la méthode de compression choisie parmi celles supportées par le client.

***Message* Server certificate**

Si la méthode d'échange de clés n'est pas anonyme, le serveur doit envoyer un certificat au client. Ce message accompagne généralement le message *Server hello* et est destiné à permettre au client de s'assurer de l'identité du serveur.

Un certificat répond généralement à la norme X.509 de l'*ITU* (*International Telecommunication Union*). Il est émis par une autorité de certification, telle *RSA Security*. Cette autorité utilise une clé privée pour signer les certificats et diffuse la clé publique associée.

Le certificat est composé des éléments suivants :

- le nom de l'autorité de certification ;
- le nom du propriétaire du certificat ;
- la date de validité du certificat ;
- l'algorithme de chiffrement utilisé ;
- la clé publique du propriétaire ;
- la signature du certificat générée par l'autorité de certification à l'aide de la clé privée de celle-ci.

Lorsque le client reçoit le certificat, il est en mesure de vérifier, à l'aide de la clé publique de l'autorité de certification, que le serveur est bien celui qu'il prétend être. De plus, le client reçoit la clé publique du serveur. Celle-ci sera utilisée pour l'échange de la *pré-clé privée*.

***Message* Server key exchange**

Le message *Server key exchange* permet au serveur d'envoyer au client une clé publique, laquelle sera utilisée pour l'échange de la pré-clé privée. Toutefois, ce message n'est pas nécessaire si le certificat du serveur a été envoyé au client (la clé publique du serveur est contenue dans le certificat).

***Message* Certificate request**

Un serveur non anonyme, c'est-à-dire un serveur qui permet la vérification de son identité en fournissant un certificat, peut demander l'identification du client. Pour cela, il peut demander au client de lui envoyer un certificat à l'aide d'un message *Certificate request*. Ce message est donc optionnel.

***Message* Server hello done**

Lorsque le serveur a émis ses différents messages (choix des algorithmes, envoi du certificat, envoi de la clé publique, demande de certificat), il envoie un message *Server*

hello done afin d'indiquer la fin de l'envoi des informations.

***Message* Client certificate**

Dans le cas où le serveur a demandé l'identification du client, ce dernier doit envoyer un certificat. Il le fait donc à l'aide d'un message de type *Client certificate*.

***Message* Client key exchange**

Le serveur et le client se sont mutuellement authentifié. Par ailleurs, le client dispose de la clé publique du serveur et il peut donc envoyer, en toute sécurité, une pré-clé privée au serveur. La pré-clé privée va donc être chiffrée à l'aide de la clé publique du serveur, puis envoyée à celui-ci. Le serveur pourra déchiffrer la pré-clé privée à l'aide sa clé privée.

Les deux interlocuteurs disposent alors tous deux de la pré-clé privée et ils vont donc pouvoir calculer la *clé privée de session* par le calcul suivant :

```
master_secret = PRF(pre_master_secret, "master secret",
                           ClientHello.random + ServerHello.random)
```

Ici, la fonction pseudo aléatoire (`PRF`) est utilisée pour calculer la clé privée de session (`master_secret`) à l'aide de la pré-clé privée (`pre_master_secret`), de la chaîne « `master secret` » et des valeurs aléatoires générées par le client (ClientHello.random) et le serveur (`ServerHello.random`).

Notons que la clé de session ainsi générée est la même pour le client et le serveur : il s'agit d'une clé symétrique.

***Messages* Change cipher spec *et* Finished**

Dès l'instant où la clé de session a pu être calculée, la connexion peut être chiffrée à l'aide de celle-ci. Le passage en mode chiffré est indiqué au serveur par le client à l'aide d'un message *Change cipher spec*. Ce message est accompagné d'un message *Finished* annonçant la fin de la négociation réalisée grâce au protocole de *Handshake*.

Le serveur peut à son tour passer en mode chiffré en émettant, lui aussi, un message *Change cipher spec* accompagné d'un message *Finished*.

La connexion est alors sécurisée. Les paramètres pourront être renégociés par l'émission d'un message *Hello*.

3 Signalisation : *Alert protocol*

À l'instar du protocole ICMP pour le protocole IP, le protocole *Alert protocol* gère la signalisation au cours de la connexion. Ainsi, ce protocole est responsable de la fermeture des connexions, que ce soit dans le cadre normal de la communication ou à la suite d'un problème.

Deux types de messages sont définis. Les messages de type « Closure alert » mettent fin à une connexion lorsque la communication est terminée. Les messages de type « Error alert » indiquent un problème concernant les données reçues.

Les messages d'erreur sont associés à un niveau de sévérité de l'erreur : « warning » ou « fatal ». Le premier niveau caractérise une erreur récupérable, tandis que le second niveau implique une fermeture immédiate de la connexion.

Tous les messages émis par le protocole *Alert* sont chiffrés. Ainsi, il n'est pas possible, pour un tiers malveillant, d'émettre un message de demande de fermeture de connexion, par exemple.

4 Transmission de données : *Application data protocol*

Les données envoyées par le client ou par le serveur sont placées dans des messages de type *Application data*. Ces messages contiennent les données chiffrées.

La gestion de ces messages (fragmentation, chiffrement, compression...) est laissée à la couche *TLS Record protocol*, et c'est pourquoi les messages *Application data* contiennent uniquement les données accompagnées de leur longueur. Notons que la longueur est indiquée dans le premier fragment. Ainsi, si le message est fragmenté, le premier paquet contiendra la longueur totale ainsi qu'un premier fragment de données, et les paquets suivants contiendront les fragments suivants.

3 Code d'authenticité des messages : HMAC

Lorsqu'un message est reçu, il est nécessaire de s'assurer de son authenticité. Cela signifie deux choses :

- s'assurer que les données proviennent bien de l'interlocuteur ;
- s'assurer que les données n'ont pas été modifiées par une tierce personne.

On utilise un code MAC (Message Authentication Code) qui repose sur un hachage du message. Pour assurer la sécurité, le hachage est réalisé avec la clé secrète partagée par les deux interlocuteurs. On parle alors de HMAC qui est un standard IETF normalisé dans la RFC numéro 2104.

Le code HMAC ainsi généré sera envoyé avec les données. À la réception, le destinataire calculera le code HMAC avec la clé secrète et pourra ainsi déterminer si les données sont valides. En effet, il n'est pas possible de calculer le code si on ne dispose pas de la clé secrète.

HMAC permet l'utilisation de tout type d'algorithme de hachage sans requérir de modification de celui-ci. Ainsi, des algorithmes tels MD5 ou SHA-1 peuvent être utilisés.

Le fonctionnement de HMAC repose sur un hachage en deux temps, tel que cela est illustré par la figure 3.

Pour obtenir le code HMAC, il faut procéder de la manière suivante :

1. Étendre la clé secrète avec le code *ipad* afin d'obtenir une taille multiple de la taille des blocs utilisés, puis réaliser une opération de « ou exclusif » entre ces données et *ipad*.
2. Concaténer le résultat de l'opération 1 avec les données à coder.
3. Hacher le résultat de l'opération 2.
4. Reproduire l'opération 1 avec le code *opad* et le concaténer avec le résultat de l'opération 3.
5. Hacher le résultat de l'opération 4.

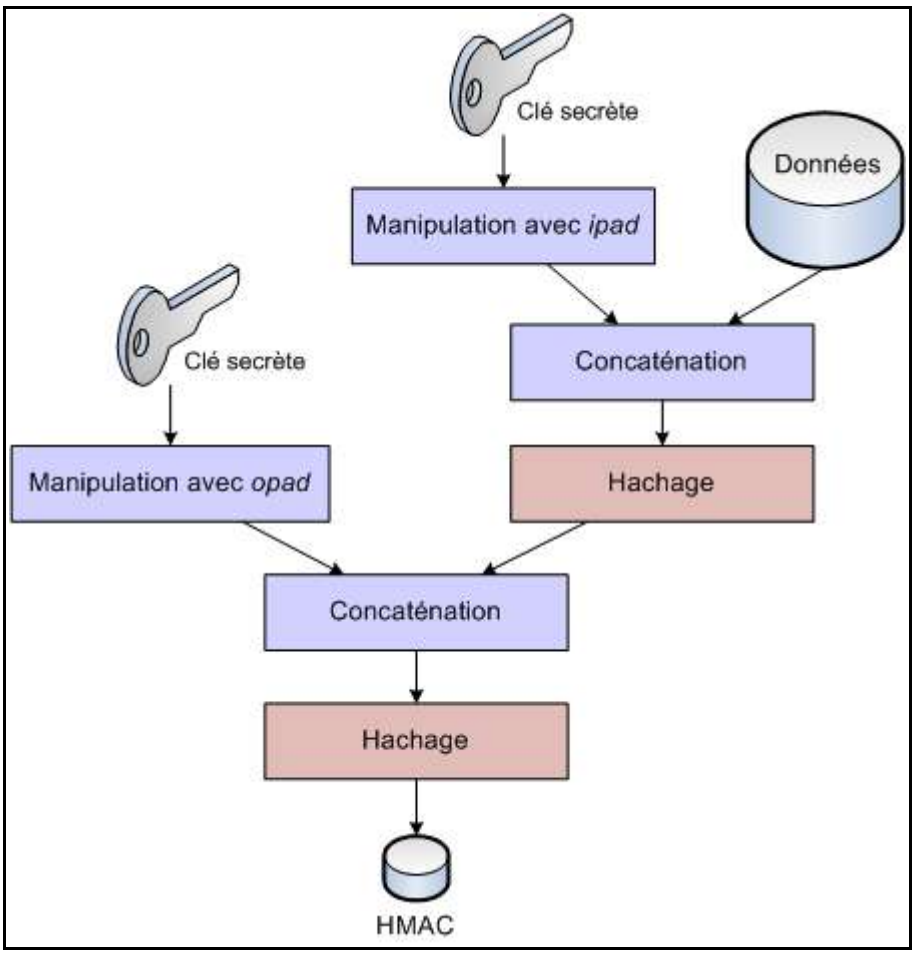


Illustration 3 : Calcul du code HMAC

L'algorithme mis en oeuvre par HMAC permet d'assurer qu'une modification par une tierce personne sera détectée à l'arrivée. Des attaques de type « man in the middle » seront ainsi détectées. Toutefois, cela ne permet pas de prévenir des attaques de type « rejeu » (ré-émission ultérieures des données). Pour y pallier, l'algorithme HMAC n'est pas directement utilisé avec la clé secrète mais avec le résultat d'une fonction pseudo-aléatoire (*PRF*, *Pseudo Random Function*), laquelle utilise des données échangées à l'initialisation de la connexion (un *Initialisation Vector*, autrement dit un vecteur d'initialisation) ainsi qu'un numéro de séquence. De cette façon, des échanges ne pourront pas être rejoués au cours de la même session ou au cours d'une autre session.

La fonction pseudo aléatoire dépend de la clé secrète (*secret*), de données aléatoires (*label*) et d'une graine (*seed*) :

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) XOR
                           P_SHA-1(S2, label + seed);
```

`S1` représente la première moitiée de la clé secrète, tandis que `S2` représente l'autre moitiée. Les fonctions `P_MD5` et `P_SHA-1` sont elles-mêmes définies comme suit :

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

Enfin, la fonction `A` est définie par récurrence avec les paramètres suivants :

```
A(0) = seed
```

```
A(i) = HMAC_hash(secret, A(i-1))
```

4 Évolutions récentes de TLS

1 TLS version 1.1

Le groupe de travail de l'IEFT assure les évolutions du protocole. Un *draft* est notamment en cours de rédaction dans le but de spécifier TLS 1.1. À l'heure actuelle, TLS 1.1 reste très proche de TLS 1.0.

TLS 1.1 apporte des modifications mineures à TLS 1.0 et définit des algorithmes de chiffrements qui ne doivent plus être utilisés. Ces algorithmes proposent en effet un niveau de sécurité trop faible en regard de la puissance de calcul des ordinateurs actuels.

La principale évolution du protocole réside dans l'introduction d'un vecteur d'initialisation permettant de pallier un problème de sécurité pouvant survenir en utilisant un chiffrement de type CBC.

2 Sécurité WiFi

Les communications sans fil de type 802.11 (WiFi) sont très exposées aux écoutes malveillantes du réseau (*eavesdropping*) et sont par conséquent très sensibles aux problèmes de sécurité.

Le protocole *EAP* (*Extensible Authentication Protocol*) permet l'utilisation de tout type de protocole d'authentification en rendant abstraite leur utilisation.

EAP-TLS, qui fait l'objet de la *RFC 2716*, apporte un niveau de sécurité intéressant en permettant l'authentification du client et du serveur par clé publique. Toutefois, l'identité du client (son nom d'utilisateur) est transmise en clair sur le réseau avant l'authentification. Cela provoque un problème de sécurité permettant à un tiers d'obtenir l'identité du client.

EAP-TTLS (*EAP-Tunneled TLS*), actuellement à l'état de *draft*, est destiné à résoudre ce problème. Le protocole met en jeu trois acteurs : le client, le serveur et un serveur TTLS. Un tunnel TLS est d'abord créé entre le client et le serveur TTLS. L'authentification du client est ensuite réalisée à l'intérieur de ce tunnel. La communication entre le client et le serveur peut ensuite démarrer.

3 WAP (Wireless Application Protocol)

WAP est un protocole de transfert de données permettant à des terminaux mobiles (des téléphones portables, notamment) d'accéder à des ressources situées sur Internet. L'organisme à l'origine de sa standardisation est le *WAP Forum*.

Le protocole WAP est l'équivalent du protocole HTTP couramment utilisé sur Internet. De même, le langage HTML (*HyperText Markup Language*) trouve un équivalent dans WML (*Wireless Markup Language*). Le protocole TLS, quant à lui, n'a pas été adapté au WAP mais des recommandations ont été émises quant à son utilisation. En effet, une des caractéristiques principales des terminaux mobiles utilisant le protocole WAP est qu'ils utilisent un accès à bas débit. Dès lors, il faut prendre garde au volume de données nécessaire à l'initialisation de la connexion sécurisée.

Deux recommandations sont destinées à minimiser les échanges de données :

- la clé de session doit être la plus courte possible (8 octets, voire moins si possible) ;

- la durée de validité de la clé de session doit être la plus longue possible (12 heures, voire plus si possible).

Ces deux points permettent à un terminal de se reconnecter à un site sans nécessiter un nouvel échange de clés. Il s'agit donc d'une réelle optimisation dès lors que le client s'est identifié une première fois.

Outre cela, le WAP Forum émet des recommandations quant à l'authentification :

- l'authentification du serveur par le client est obligatoire et doit être réalisée grâce à un certificat X.509 ;
- l'authentification du client par le serveur est fortement recommandée. Si celle-ci est supportée, elle doit être réalisée grâce à un certificat X.509.

Le document complet est disponible à l'adresse suivante :

http://www1.wapforum.org/tech/terms.asp?doc=WAP-219-TLS-20010411-a.pdf

4 FTP

Le protocole TLS sécurise une connexion dans son ensemble. C'est ce qui est réalisé lorsque TLS est utilisé pour sécuriser une connexion FTP. L'IETF travaille actuellement à la mise au point d'une extension du jeu de commandes FTP destinée à rendre plus modulaire la sécurisation du protocole.

Le *draft Murray* définit la manière dont le client et le serveur pourront communiquer afin de sécuriser de manière indépendante chaque échange. L'authentification du client pourra être réalisée avec un algorithme différent de celui qui sera utilisé pour transférer des données, par exemple. Il est également possible de ne sécuriser que l'authentification sans chiffrer les données.

Les commandes utilisées sont au nombre de quatre : `AUTH`, `PBSZ`, `PROT` et `CCC`. La commande `AUTH`, associée au paramètre TLS, permet l'authentification. Les commandes `PBSZ` (*protection buffer size*) et `PROT` permettent quant à elle le passage en mode chiffré. Enfin, la commande `CCC` permet de revenir un mode non sécurisé.

CONCLUSION

Nous avons tout d'abord introduit différents problèmes de sécurité liés à l'échange de données privées sur un support de communication public. Puis, nous avons présentés des mécanismes cryptographiques généraux destinés à y remédier. Nous avons enfin présentés les protocoles SSH et TLS.

SSH et TLS permettent d'apporter un haut niveau de sécurisation à des protocoles couramment utilisés mais généralement non sécurisés. De plus, ces protocoles utilisent des algorithmes cryptographiques ouverts et qui peuvent donc être étudiés afin de vérifier leur sécurité. En ce sens, ils sont intéressants.

De par leur conception, SSH et TLS ne sont pas adaptés aux mêmes applications. SSH est en effet particulièrement adapté à l'administration distante. Toutefois, ce protocole n'est pas adapté au transfert de données en raison de l'utilisation d'algorithmes à clé publique (coûteux en temps processeur). En revanche, TLS est parfaitement adapté à ce besoin.

On notera que des extensions des deux protocoles existent afin de s'adapter plus particulièrement aux utilisations courantes (FTP sur TLS, par exemple).

Notons enfin que des implémentations *open source* des deux protocoles existent. On peut notamment citer *OpenSSH* et *OpenSSL*. Grâce à ces deux bibliothèques, tout développeur peut sécuriser une application en utilisant des mécanismes maîtrisés et standardisés.

SSH et TLS sont donc deux protocoles complémentaires très intéressants pour leur facilité d'intégration et le haut niveau de sécurité qu'ils apportent.