16

WTLS (Wireless Transport Layer Security)

Le protocole WTLS (*Wireless Transport Layer Security*) est un protocole généraliste de sécurisation des échanges sur les liaisons sans fil [WTLS 99]. WTLS s'inspire du protocole TLS (*Transport Layer Securit*y, sécurité de la couche transport) de l'IETF (*Internet Engineering Task Force*), lui-même basé sur SSL à quelques détails près [RFC 2246].

Le déroulement des échanges de WTLS suit les grandes lignes de SSL, mais le contenu des messages de WTLS a été remanié de fond en comble. WTLS est ainsi capable de circonvenir les interruptions de flux dues aux pertes de paquets et de fonctionner au-dessus de protocoles de transport non fiables, entre autres, UDP (*User Datagram Protocol*). Il peut fonctionner lorsque les délais de transmission aller-retour sont particulièrement longs. Enfin, WTLS peut être configuré pour satisfaire les restrictions qu'imposent les faibles de capacité de calcul des terminaux mobiles et leurs mémoires limitées. Ces mécanismes permettent à WTLS de sécuriser le transport de la voix sur IP conformément aux spécifications de la norme H.323. Ces améliorations ne sont pas tout à fait gratuites, puisque WTLS n'est plus compatible avec SSL, ce qui pose des problèmes de sécurisation de bout en bout pour une même liaison. En outre, la résistance de WTLS aux attaques doit être évaluée comme s'il s'agissait d'un nouveau protocole et non d'une simple variante de SSL.

Architecture

La figure 16–1 illustre l'emplacement de WTLS dans la pile des protocoles WAP, qui est mise en parallèle avec la pile des protocoles du Web. On note que les différents protocoles de la couche réseau pour les connexions sans fil sont pris en compte. Ainsi, parmi les services des réseaux GSM (Groupe Spécial Mobile, *Global System for Mobile Communication*) sont pris en charge le système de courts messages SMS (*Short Messaging System*) pour les débits limités à 9,6 Kbit/s et les services généraux par paquets radios GPRS (*General Packet Radio Services*) pour le trafic IP entre 28 et 56 Kbit/s. Les services de WAP peuvent être offerts en utilisant des données sur le réseau commuté (*Circuit*

Switched Data ou CSD). Il accommode aussi le système de téléphonie mobile analogique le plus répandu en Amérique du nord, AMPS (*Advanced Mobile Phone System*, Système de téléphone mobile avancé), au moyen d'une interface CDPD (*Cellular Digital Packet Data*, données en paquet numérique cellulaire).

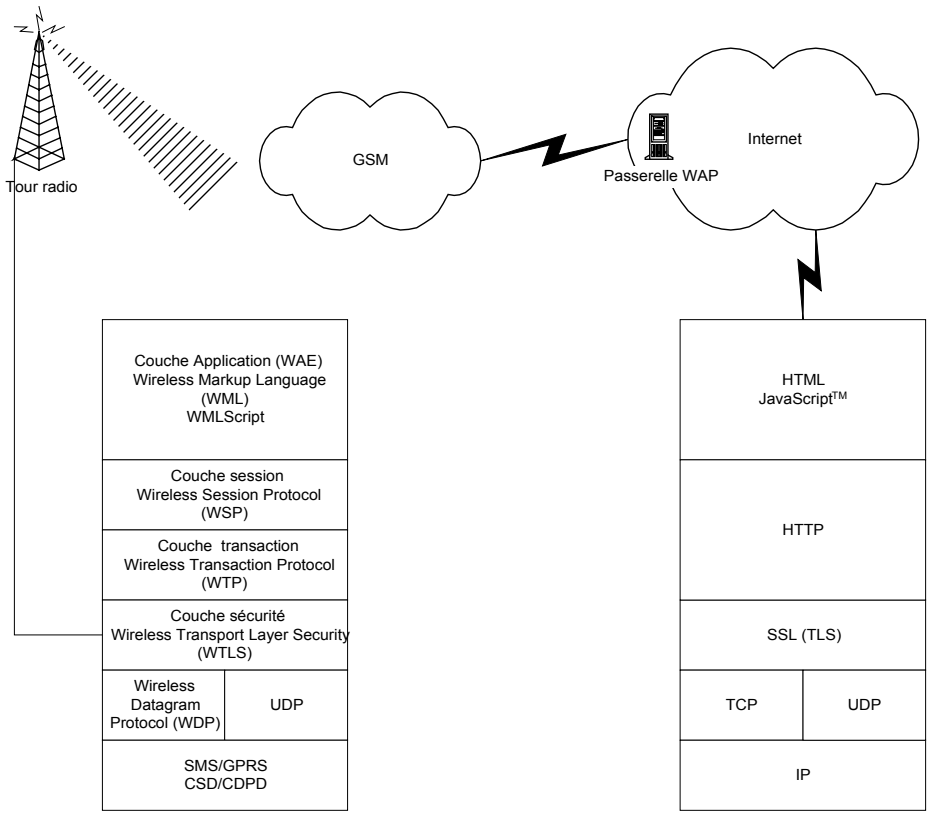


Figure 16–1. : Position de WTLS dans l'empilement protocolaire et sa correspondance avec la pile TCP/IP

Les services de sécurisation de WTLS

WTLS fournit quatre services de sécurisation : l'authentification des intervenants, l'intégrité et la confidentialité des messages ainsi que la non-répudiation au moyen de certificats numériques émis par des tiers de confiance.

L'identification et l'authentification

L'identification dans WTLS peut être déclinée de plusieurs façons : des noms distinctifs du type X.509, des condensâts SHA-1 d'une clé publique, une identité binaire secrète connue

des deux intervenants, une série de caractères dans un alphabet reconnue par les deux parties. Il est possible aussi de ne pas s'identifier.

L'authentification, si elle a lieu, est effectuée à l'établissement de la session et avant la première transmission de données. Elle se fait au moyen d'un de trois types de certificats :

1. des certificats conformes à la recommandation X.509 version 3 de l'UIT-T ;
2. des certificats conformes au projet de la norme ANSI X9.68 en court d'élaboration et qui concerne les certificats scellés au moyen de l'algorithme de signature DSA sur courbe elliptique (ECDSA) . Le format de ce type de certificats est toujours à l'étude ;
3. des certificats WTLS dont la taille a été réduite pour économiser la bande passante radio.

Rappelons qu'un certificat X509 a la forme suivante :

$$\text{certificat} = \text{Nom}_{\text{Serveur}} + \text{Nom}_{\text{AC}} + \text{Kp}_{\text{Serveur}} + \text{Ks}_{\text{AC}}\{\text{H}[\text{Nom}_{\text{Serveur}} + \text{Nom}_{\text{AC}} + \text{Kp}_{\text{Serveur}}]\}$$

où :

$\text{Nom}_{\text{Serveur}}$: le nom du serveur

Nom_{AC} : le nom de l'autorité de certification

$\text{Kp}_{\text{Serveur}}$: la clé publique du serveur

$\text{Ks}_{\text{AC}}\{x\}$: le sceau de *x* calculé avec la clé privée de l'autorité de certification

$\text{H}[y]$: le condensât de *y* au moyen d'une fonction de hachage H( )

Le certificat *sui generis* WTLS se présente sous la forme suivante :

certificat = version + algorithme de signature + identifiant de l'émetteur + date de début de validité + date de péremption + identifiant du porteur + type de la clé publique + paramètres de l'algorithme de chiffrement + clé publique

L'emploi de certificats est facultatif à l'heure actuelle. Comme pour SSL, il est possible d'échanger des clés sans authentification – l'échange est alors dit « anonyme » – mais ce mode d'opération est déconseillé, car vulnérable aux attaques par interception (*man-in-the-middle*).

L'échange de clés (avec ou sans authentification) emploie un des algorithmes suivants : RSA, Diffie-Hellman ou Diffie-Hellman sur courbe elliptique (ECDH). Dans ce dernier cas, le client doit être certifié. Le tableau 1 résume l'ensemble des algorithmes d'échange de clés pris en charge par WTLS. Le choix de l'algorithme est négocié pendant l'établissement de la connexion.

Tableau 1 - Algorithmes d'échange de clés négociés par le protocole Handshake

Code du mode d'échange de clés	Description	Taille limite des clés en bits
NULL	Aucun échange. Le secret *PreMastersecret* est de longueur nulle.	–

SHARED_SECRET	Le secret est échangé par d'autres moyens que WTLS ; souvent il est déjà physiquement incrusté dans chaque terminal.	–
DH_anon	Échange Diffie-Hellman sans authentification. Chaque partie calcule le *PreMastersecret* à partir de sa clé privée et de la clé publique du correspondant	–
DH_anon_512	"	512
DH_anon_768	"	768
RSA_anon	Échange de clés RSA sans authentification. Le serveur envoie sa clé publique au client, lequel génère un clé privée et la chiffre avec la clé publique du serveur. Il envoie le résultat au serveur. Le *PreMastersecret* est le résultat du chiffrement auquel est accolée la clé publique du serveur	–
RSA_anon_512	"	512
RSA_anon_768	"	768
RSA	Échange de clés RSA avec authentification au moyen de certificats. Le *PreMastersecret* est la clé privée du client chiffrée avec la clé publique du serveur t à laquelle est accolée la clé publique du serveur	–
RSA_512	"	512
RSA_768	"	768
ECDH_anon	Échange de clés Diffie-Hellman par la méthode des courbes elliptiques	–
ECDH_anon_113	"	113
ECDH_anon_131	"	131
ECDH_ECDSA	Échange de clés Diffie-Hellman par la méthode des courbes elliptiques avec des certificats scellés avec l'algorithme ECDSA	–

La clé échangée peut être inclue dans le certificat du serveur, chiffrée par l'algorithme de chiffrement convenu, ou peut être une clé temporaire, échangée au moyen des messages *ServerKeyExchange* ou *ClientKeyExchange* selon le type de l'échange choisi.

La confidentialité

La confidentialité des messages s'appuie sur des algorithmes de chiffrement symétrique avec une clé de longueur effective allant de 40 bits à 168 bits. Le même algorithme de cryptographie est utilisé par les deux parties mais chacune se sert de sa propre clé secrète, qu'elle partage avec l'autre partie. On nomme ces clés *client_write_key* côté client et *server_write_key* côté serveur. Ainsi, chaque entité communicante possède une clé de chiffrement symétrique, différente de celle de son partenaire. Un flux est donc chiffré dans une direction par une clé distincte de celle avec laquelle est chiffré le flux correspondant dans la direction inverse.

Le tableau 2 contient les algorithmes de chiffrement employés dans WTLS. Tous ces algorithmes emploient un chiffrement en blocs de 64 bits et ayant un vecteur d'initialisation

de 64 bits. Certains algorithmes ont été délibérément affaiblis afin de satisfaire les restrictions sur l'exportation des algorithmes cryptographiques. Ainsi, une variable binaire signale la possibilité d'exportation. On remarque que WTLS ajoute d'autres algorithmes à ceux employés dans SSL.

Tableau 2 - Algorithmes de chiffrement employés dans WTLS

Représentation du chiffre	Limites à l'exportation	Longueur effective de la clé en bits	Longueur de la clé en écriture en bits	Emploi dans SSL
NULL	–	–	–	Ou
RC5_CBC_40	Non	40	128	
RC5_CBC_56	Non	56	128	
RC5_CBC	Ou	128	128	
DES_CBC_40	Non	40*	64	Ou
DES_CBC	Ou	56	64	Ou
3DES_CBC_EDE	Ou	168	192	Ou
IDEA_CBC_40	Non	40	128	
IDEA_CBC_56	Non	56	128	
IDEA_CBC IDEA	Ou	128	128	Ou

* seuls 35 bits de la clé sont utilisés pour le chiffrement, les 5 autres étant des bits de parité [SAA 00].

L'intégrité

L'intégrité des données est assurée par l'application de fonctions de hachage SHA-1 ou MD5 (même si l'efficacité de ce dernier est fortement contestée). Comme dans SSL, l'emploi de la procédure HMAC confère une meilleur protection contre les attaques [BEL 96]. Néanmoins, des procédures beaucoup plus faibles ont été retenues ainsi que l'indique le tableau 3.

Tableau 3 - Algorithmes de hachage employés dans WTLS

Représentation de la fonction de hachage	Description	Taille de la clé en bits	Taille du condensât en bits	Emploi dans SSL
SHA_0	Pas de paraphe	–	–	
SHA_40	Après avoir calculé le paraphe, seuls les premiers 40 bits du résultats sont employés	160	40	
SHA_80	Après avoir calculé le paraphe, seuls les premiers 80 bits du résultats sont employés	160	80	
SHA	Le paraphe est calculé selon les spécifications de SHA-1	160	160	Oui

SHA_XOR_40	L'entrée est divisée en blocs de 40 octets. Une opération *ou exclusif* est effectuée successivement sur les différents blocs. Cette opération, quoique déconseillée, est retenue pour les terminaux ayant une faible capacité de calcul. Elle ne peut garantir les chiffrements enfilés [SAA 00]	0	40	
MD5_40	Après avoir calculé le paraphe, seuls les premiers 40 bits du résultats sont employés	128 bits	40	
MD5_80	Après avoir calculé le paraphe, seuls les premiers 80 bits du résultats sont employés	128 bits	80	
MD5	Le paraphe est calculé selon les spécifications de MD5	128 bits	128	Oui

Les sous-protocoles de WTLS

À l'instar du protocole SSL, le protocole WTLS comporte quatre sous-protocoles :

1. *Handshake* qui est chargé de l'authentification des parties en communication, de la négociation des algorithmes de chiffrement et de hachage, et de l'échange d'un secret, le *PreMasterSecret* ;
2. *Record* qui met en œuvre les paramètres de sécurité négociés pour protéger les données d'application ainsi que les messages en provenance des protocoles Handshake, ChangeCipherSpec (CCS) et Alert ;
3. *ChangeCipherSpec* (CCS) qui a pour fonction de signaler à la couche Record toute modification des paramètres de sécurité ;
4. *Alert* qui est chargé de signaler les erreurs rencontrées pendant la vérification des messages ainsi que toute incompatibilité qui pourrait survenir pendant le Handshake.

La figure 16–2 illustre l'agencement de ces différentes éléments. On voit que le protocole Record se place au-dessus de la couche transport, tandis que les trois autres protocoles se situent entre l'application et la couche Record.

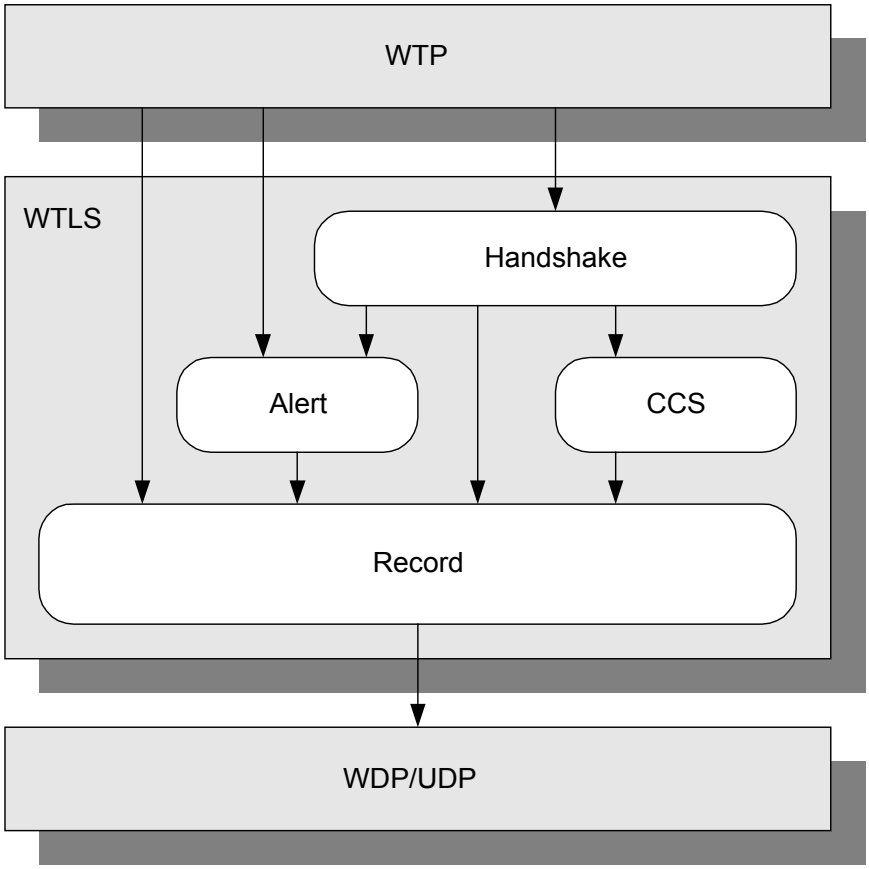


Figure 16–2. : Empilement des sous-couches protocolaires de WTLS

Déroulement des échanges WTLS

NOTE — Dans cet exposé, nous retenons autant que possible les mêmes noms de variables que ceux utilisés dans SSL pour faciliter la comparaison entre les deux protocoles, et ceci même lorsqu'ils ont été légèrement modifiés dans les documents du WAP Forum (*Wireless Application Forum*, Forum des applications sans fil).

Comme pour SSL, les échanges du protocole WTLS se déroulent en deux temps :

1. durant la phase préliminaire, ont lieu l'identification des parties, la négociation des attributs cryptographiques, la génération et le partage des clés ;
2. durant les échanges de données, la sécurisation opère à partir des algorithmes et des paramètres secrets négociés durant la phase préliminaire.

À tout moment, il est possible de signaler une intrusion ou une erreur d'opération.

Chaque fois qu'un client se connecte à un serveur, il déclenche une session WTLS. Si le client se connecte à un autre serveur, il engage une nouvelle session, sans interrompre la

session en cours. S'il revient par la suite au premier serveur et souhaite conserver les précédents choix cryptographiques, il demandera au premier serveur de reprendre une ancienne session plutôt que d'en commencer une nouvelle. Ainsi, selon WTLS, une session est une association entre deux entités ayant en commun un certain nombre de paramètres et attributs cryptographiques. La reprise d'une session interrompue est seulement possible si la procédure de suspension a été suivie scrupuleusement.

Une session peut contenir plusieurs connexions contrôlées par l'application. Grâce à la notion de connexion WTLS, une application est en mesure de « rafraîchir » (modifier) certains attributs de sécurité (tels que les clés de chiffrement) sans remettre en cause tous les attributs déjà négociés en début de session.

Par rapport à SSL, WTLS contient deux modification fondamentales : une volée de messages du protocole Handshake WTLS peut être consolidée en un seul envoi. Cette consolidation, obligatoire lorsque le transport n'est pas fiable, est utilisée pour les échanges SMS. De même, la retransmission des messages du Handshake est permise dans certaines conditions.

Voyons de plus près ce que recouvrent les notions de session et de connexion.

Variables d'état d'une session WTLS

À chaque bout d'une liaison sécurisée par WTLS, une session est identifiée par six variables comme pour SSL. Ces variables sont les suivantes :

- l'identificateur de session (*session ID*) : séquence arbitraire de 8 octets (et non de 32 octets comme pour SSL) choisie par le serveur pour identifier une session active ou pouvant être réactivée ;
- la version du protocole utilisée (*protocol version*) ;
- le certificat du pair (*peer certificate*) : certificat du correspondant. Ce certificat peut être soit conforme à la version 3 de X.509, soit au format de ANSI X9.68, soit au format spécial de WTLS, optimisé pour réduire la taille des échanges . La valeur du certificat est nulle si le correspondant n'est pas certifié ;
- la méthode de compression, c'est-à-dire l'algorithme de compression utilisé. Le protocole WTLS prévoit en effet la possibilité de négocier une méthode de compression de données. Pour le moment, aucune méthode n'a été retenue et par conséquent, ce champ reste vide ;
- la suite de chiffrement (*cipher spec*) : elle définit les algorithmes de cryptage et de hachage utilisés à partir d'une liste préétablie ;
- le MasterSecret : c'est un secret de 20 octets (au lieu de 48 octets pour SSL) partagé entre le client et le serveur, à partir duquel on génère les autres secrets. Ce paramètre est donc valable pour toute la session ;
- le mode de numérotage des séquences de paquets (*sequence number mode*). En effet, WTLS offre trois modes de numérotage :
  - un numérotage implicite dans le cas où la couche de transport est fiable. Ce mode est identique à celui qu'emploie SSL puisque ce dernier repose sur TCP ;

  - un numérotage explicite, c'est-à-dire que les numéros de séquences sont envoyés en clair dans les messages Record afin d'être utilisés pour le calcul des condensâts. Ce mode est obligatoire lorsque la couche de transport n'est pas fiable ;
  - pas de numérotage. Cette option est déconseillée, car elle augmente la vulnérabilité aux attaques par rejeu si les couches supérieures n'ont pas incorporé les parades nécessaires contre ce genre d'attaques.
- l'intervalle de rafraîchissement des clés (*key refresh*) qui définit la fréquence de mise à jour de paramètres de chiffrement (clés de chiffrement, clé de calcul des paraphes et vecteurs d'initialisation). Les nouvelles clés sont calculées après un échange de $2 \times$ *key_refresh* messages ;
- le drapeau (*is resumable*) : ce paramètre signale s'il est permis d'ouvrir de nouvelles connexions sur la session en question.

Une suite de chiffrement est définie par l'algorithme de chiffrement utilisé et ses paramètres ainsi que par l'algorithme de hachage utilisé et la taille du condensât.

La négociation concernant les suites de chiffrement se fait en clair pendant l'établissement de la session.

Variables d'état d'une connexion WTLS

Les paramètres qui définissent l'état d'une connexion pendant une session sont ceux qui seront « rafraîchis » lors de l'établissement d'une nouvelle connexion. Ces paramètres sont :

- L'extrémité de la connexion (*connection end*), qui définit si l'extrémité est un client ou un serveur.
- Deux nombres aléatoires (*server_random* et *client_random*) de 16 octets (la taille était de 32 octets pour SSL), générés respectivement par le serveur et par le client lors de l'établissement d'une session et à chaque nouvelle connexion. Ils sont composés de deux champs chacun ; le premier champ est de 4 octets et comprend l'heure donnée par l'horloge interne du client exprimée en temps universel (GMT) ; le second champ est de 12 octets et contient un aléa produit par un générateur de nombres aléatoires. Les clés secrètes sont dérivées de ces nombres aléatoires ; par conséquent, ces derniers sont échangés en clair à l'ouverture d'une session. En revanche, lors de l'ouverture d'une nouvelle connexion pendant une session active, le processus de chiffrement fonctionne déjà et les nombres sont alors transmis chiffrés. L'utilisation de ces nombres protège contre les attaques par rejeu d'anciens messages.
- Deux clés secrètes, *server_MAC_write_secret* et *client_MAC_write_secret*, employées pendant la condensation pour calculer les paraphes hachés HMAC. La taille du paraphe dépend de l'algorithme de hachage choisi, soit 16 octets pour SHA et 20 octets pour MD5.
- Deux clés pour le chiffrement symétrique des données, une du côté serveur et l'autre du côté client. Bien que les deux parties utilisent le même algorithme, chacune a la possibilité de se servir de sa propre clé de chiffrement, *server_write_key* ou *client_write_key*, à condition de la partager avec l'autre partie. La taille des clés dépend de l'algorithme de chiffrement choisi et de la législation sur la cryptographie en vigueur.
- Deux vecteurs d'initialisation pour le chiffrement symétrique en mode CBC, l'un du côté serveur et l'autre du côté client. Leur taille dépend de l'algorithme choisi.

- Les numéros de séquence des messages échangés. Ces numéros sont codés sur 2 octets (au lieu de 8 octets pour SSL) de part et d'autre de la liaison (côté serveur et côté client). Ils sont maintenus séparément pour chaque connexion et augmenté d'une unité à chaque émission de message associé à cette connexion. Ainsi, la connexion doit être interrompue lorsque le numéro de séquence atteint 2^{16} - 1.

Calculs de paramètres

La figure 16–3 illustre le calcul du *MasterSecret* à partir du *PreMasterSecret* et des paramètres *client_random* et *server_random*.

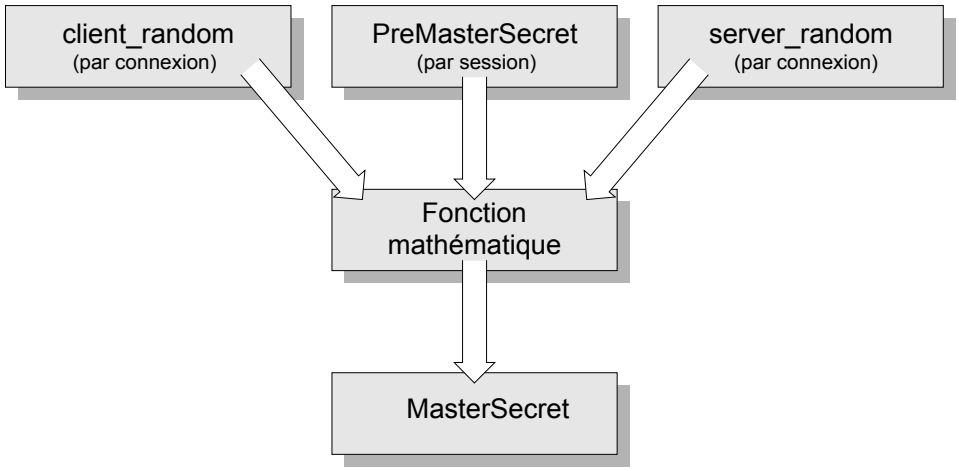


Figure 16–3. : Construction du MasterSecret à l'ouverture d'une session

Calcul de PreMasterSecret

Pour les échanges de clés RSA, le client génère un secret de 20 octets et le chiffre à l'aide de la clé publique du serveur. Ce dernier récupère la valeur en déchiffrant le message à l'aide de sa clé publique. *PreMasterSecret* est formé en accolant la clé publique du serveur au secret .

Pour les échanges de clés de type Diffie-Hellman, que les calculs conventionnels soient utilisés ou qu'ils soient effectués sur courbes elliptiques, la clé négociée constitue le *PreMasterSecret*.

Dans la méthode dite du secret partagé (*shared secret*), souvent en l'incrustant physiquement dans le terminal, le *PreMasterSecret* est identique à ce secret.

La taille de *PreMasterSecret* varie selon la méthode d'échange de clés. Elle est de 20 octets si le message est chiffré à l'aide de RSA – c'est le cas où RSA est employé pour l'échange de clés et l'authentification – au lieu de 48 octets pour SSL.

Calcul de MasterSecret

Le calcul de MasterSecret s'effectue en appliquant la formule suivante :

MasterSecret = FPA (PreMasterSecret, "master secret", ClientHello.random || ServerHello.random)

où FPA est une fonction pseudo-aléatoire et l'opérateur || dénote le rattachement des deux chaînes de caractères (la « concaténation »). Les paramètres *client_random* et *server_random* sont compris dans les variables *ClientHello.random* et *ServerHello.random* respectivement.

La FPA consiste à exécuter sur le secret les opérations suivantes :

$$\begin{aligned} \text{FPA (k, étiquette, germe)} = {} & \text{HMAC}_k\,[\text{A(1)} \,||\, \text{étiquette} \,||\, \text{germe}] \,|| \\ & \text{HMAC}_k\,[\text{A(2)} \,||\, \text{étiquette} \,||\, \text{germe}] \,|| \\ & \text{HMAC}_k\,[\text{A(3)} \,||\, \text{étiquette} \,||\, \text{germe}] \,|| \ldots \end{aligned}$$

où *germe* un nombre aléatoire et *étiquette* une chaîne de caractères. Dans ce cas précis, l'étiquette est le texte "*master secret*". Rappelons que le paraphe haché HMAC est calculé à partir de la fonction de hachage H() par la formule :

$$\text{HMAC}_k(m) = H\left[\bar{k} \oplus \text{opad} \,\|\, H(\bar{k} \oplus \text{ipad}, m)\right]$$

où $\oplus$ représente un « ou exclusif ». Le secret $\bar{k}$ est formé à partir de la clé initiale *k* de L bits en accolant une suite de « 0 » pour atteindre la taille de B bits. Par exemple, l'algorithme SHA-1 ayant L = 20 octets et B = 64 octets, $\bar{k}$ sera construit en ajoutant 44 octets au format 0x00. Les variables *opad* et *ipad* sont les constantes du remplissage extérieur (*outer pad*) et intérieur (*inner pad*), constituées respectivement par les octets 0x36 et 0x5C répétés autant de fois qu'il est nécessaire pour constituer un bloc de B bits. Ainsi, pour MD5 et SHA-1, il y aura 64 répétitions, la taille de leurs blocs étant 512 bits ou 64 octets.

Les valeurs A (1), A (2), A (3), ... sont calculées de manière récursive à partir de la valeur du germe de la manière suivante :

$$\begin{aligned} \text{A (0)} &= \text{étiquette} \,||\, \text{germe} \\ \text{A (i)} &= \text{HMAC}_k\,[\text{A(i-1)}] \end{aligned}$$

Calcul des paramètres algorithmiques

Le calcul des clés de chiffrement, des clés de hachage et des vecteurs d'initialisation se fait à partir des variables *MasterSecret*, *client_random* et *server_random* de la manière illustrée dans la figure 16–4. Comme il a été déjà mentionné, les paramètres *client_random* et *server_random* sont échangés en clair, alors que le *PreMasterSecret* est échangé confidentiellement à l'aide d'un algorithme d'échange de clés. La valeur du *MasterSecret* demeure constante pendant toute la durée d'une session. Comme il a été indiqué plus haut,

les variables *client_MAC_write_secret*, *server_MAC_write_secret*, *client_write_secret*, *server_write_secret* et les vecteurs d'initialisation sont renouvelés périodiquement.

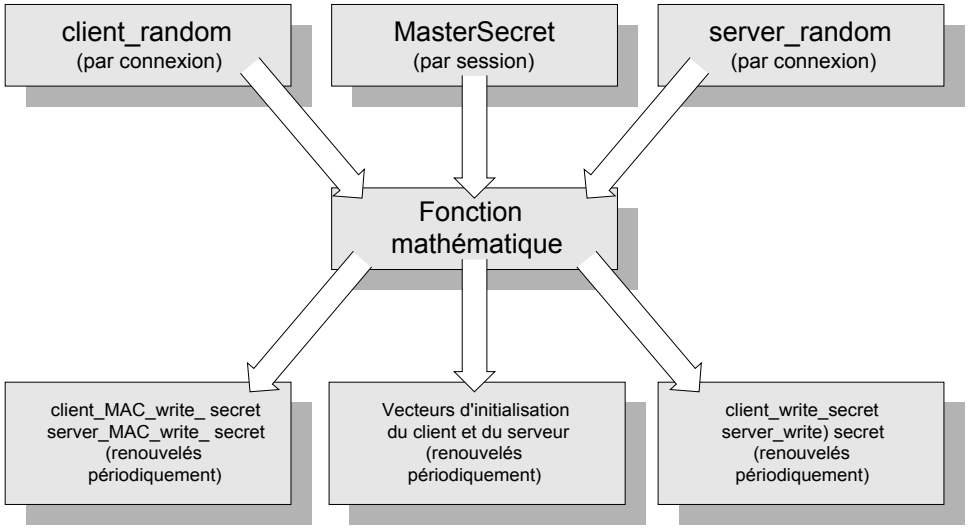


Figure 16–4. : Génération des secrets et des vecteurs d'initialisation à l'ouverture d'une session ou connexion

Une fois le *MasterSecret* obtenu, les clés de chiffrement sont déduites du bloc de chiffrement :

KeyBlock = FPA {MasterSecret, étiquette_d'expansion, numéro de séquence || server_random || client_random}

Le numéro de séquence du serveur ou du client est utilisé selon l'extrémité qui calcule *KeyBlock*. Côté serveur, l'étiquette d'expansion est "*server expansion*" et côté client, elle est "*client expansion*".

Le calcul est réitéré autant de fois qu'il est nécessaire de façon que le bloc de chiffrement soit suffisamment long pour le subdiviser en plusieurs champs. Du côté client, les partitions sont les suivantes :

```
client_MAC write_secret[taille_du_condensât]
client_write_key[longueur_de_la_clé_en_écriture]
client_write_IV[taille_du_vecteur_d'initialisation]
```

Du côté serveur, les partitions donnent :

```
server_MAC write_secret[taille_du_condensât]
server_write_key[longueur_de_la_clé_en_écriture]
```



```
server_write_IV[taille_du_vecteur_d'initialisation]
```

Pour les algorithmes de chiffrement qui sont exportables, la clé de chiffrement et les vecteurs d'initialisation sont bridés au moyen de la formule suivante :

```
final_client_write_key = FPA(client_write_key, "client write key",
                             client_random || server_random)
client_write_iv = FPA [" ","client write IV", numéro de séquence du
                       client || client_random || server_random]

final_server_write_key = FPA(server_write_key, "server write key",
                             client_random || server_random)
server_write_iv = FPA [" ","server write IV", numéro de séquence du
                       serveur || client_random || server_random]
```

On remarque que l'argument *secret* de la fonction pseudo-aléatoire est nul dans la dérivation des vecteurs d'initialisation finaux. Par conséquent, les vecteurs d'initialisation ne dépendent que des variables envoyés en clair : le numéro de séquence du client et les deux variables *client_random* et *server_random*.

Enfin, le vecteur d'initialisation finalement employé pour chiffrer un bloc est constitué au moyen d'un « ou exclusif » appliqué sur le vecteur d'initialisation :

```
vecteur d'initalisation bloc client = client_write_IV ⊕ S (numéro de
séquence du client)
vecteur d'initalisation bloc serveur = server_write_IV ⊕ S (numéro de
séquence du serveur )
```

où S est formé en accolant les 16 bits formés par la juxtaposition des 2 octets de numéro de séquence autant de fois qu'il est nécessaire pour obtenir le même nombre d'octets que dans le vecteur d'initialisation correspondant. Si le vecteur d'initialisation au départ, VI_0, est de 64 bits, le vecteur d'initialisation employé VI sera :

$$VI = VI_0 \oplus (S \,||\, S \,||\, S \,||\, S)$$

Renouvellement des paramètres cryptologiques

La cadence de renouvellement des paramètres cryptologiques est réglée au moyen de l'intervalle de rafraîchissement des clés, *key_refresh*, négocié en début de session. Ce renouvellement modifie les valeurs des paramètres des secrets MAC, des clés de chiffrement et des vecteurs d'initialisation, mais préserve les variables *client_random* et *server_random*. Celles sont modifiées à l'établissement d'une nouvelle connexion, tout en préservant *MasterSecret*, ce qui force le recalcul des autres variables.

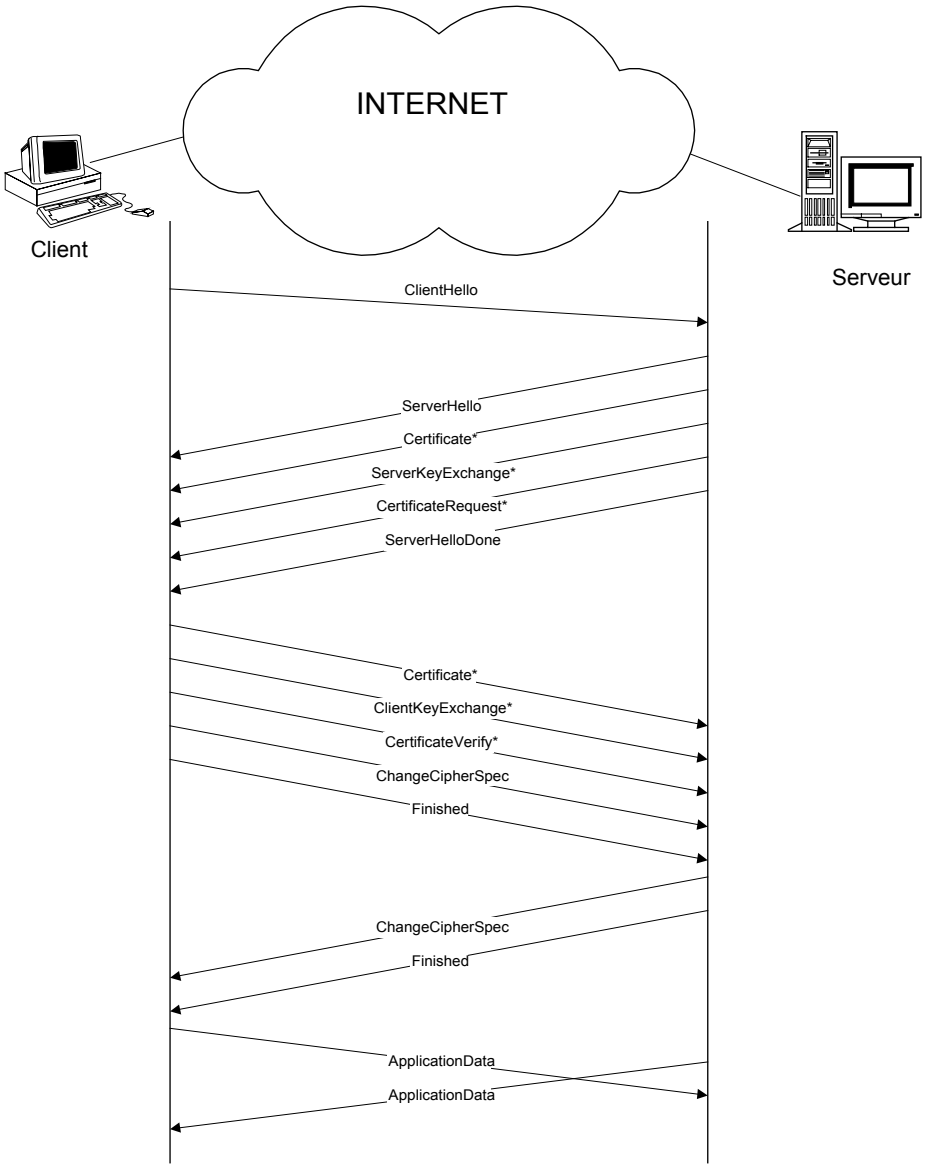
Le protocole Handshake

Fonctionnement général

Le protocole Handshake conditionne l'ensemble du processus de transfert sécurisé de données. Il commence par la négociation et l'accord sur l'algorithme d'échange des valeurs aléatoires qui permettront de partager le secret *PreMasterSecret* de part et d'autre.

L'échange de certificats a lieu ensuite, si au moins une des deux extrémité doit être authentifiée.

La figure 16–5 illustre les échanges pendant l'établissement d'une nouvelle session. La structure des messages du protocole est présentée dans l'annexe.



** message optionnel*

Figure 16–5. : Messages échangés pendant l'établissement d'une nouvelle session

Cette figure révèle deux premières différences entre l'établissement d'une nouvelle session dans WTLS et SSL :

1. le message ClientKeyExchange est optionnel dans WTLS mais obligatoire dans SSL ;
2. l'envoi des données ne commence dans WTLS qu'après l'échange du message *Finished* de part et d'autre, alors qu'il est possible juste après l'expédition de ce message dans SSL. Rappelons que cette restriction colmate la brèche qui permet à un intrus, en modifiant les messages *ClientHello* ou *ServerHello* pour forcer l'utilisation d'une suite de chiffrement faible, de récupérer le *PreMastersecret*. Après l'avoir déchiffré, il peut ensuite détourner le message de vérification *Finished* et se substituer à un ou aux deux partenaires [MIT 98].

Ouverture d'une nouvelle session

Les échanges effectués durant l'ouverture d'une nouvelle session comportent quatre étapes :

- l'accord sur les algorithmes d'échange de clés et des algorithmes de chiffrement disponibles des deux côtés ;
- l'authentification du serveur ;
- l'échange de secrets ;
- la vérification et la confirmation des messages échangés.

Identification des algorithmes d'échange de clés et de chiffrement

Les premiers messages échangés entre le client et serveur, *ClientHello* et *ServerHello*, permettent de négocier le choix des paramètres, les algorithmes de chiffrement et d'échange de clés ainsi que les caractéristiques de la session (usage des numéros de séquence, fréquence de renouvellement des paramètres, etc.). L'ouverture d'une nouvelle session par le client commence par l'envoi du message *ClientHello* au serveur. Le serveur peut aussi prendre l'initiative en envoyant le message *HelloRequest* qui ne contient aucune information et qui sert exclusivement à avertir le client que le serveur est prêt. Les échanges suivants se déroulent de la même manière dans les deux cas.

Le client et le serveur commencent obligatoirement par choisir la version du protocole WTLS commune aux deux parties (actuellement le protocole en est à la version 1.0).

Après l'envoi du message *ClientHello*, le client attend l'arrivée du message *ServerHello*. Ce message doit indiquer la version du protocole et l'identificateur unique de session, *Session_ID*, tous deux choisis par le serveur. L'identificateur de session permet la reprise d'une session ouverte précédemment. Le message comprend aussi le nombre aléatoire *server_random*. Grâce à l'échange des nombres aléatoires, *client_random* et *server_random*, chaque partie sera en mesure de reproduire les secrets de son correspondant et donc de les partager.

Le message doit comprendre la suite de chiffrement retenue par le serveur parmi les choix suggérés par le client. Cette suite sera utilisée au cours de la communication pour assurer la confidentialité et l'intégrité des données. Si aucune suite commune n'est disponible, le serveur (ou le client, selon le cas) génère le message d'erreur *close_notify* tel que le spécifie le protocole *Alert* et la session sera abandonnée.

Comme dans SSL, la négociation se déroule en clair. Un intrus peut tenter de subtiliser le message *ClientHello* et de le remplacer par un faux avec des algorithmes de chiffrement moins robustes. Grâce au message *Finished* échangé à la fin du Handshake, il est possible de déjouer cette attaque.

Authentification du serveur

Dans cette deuxième phase, le serveur s'authentifie en envoyant le message *Certificate* comprenant :

- le certificat X.509 version 3.0 du serveur qui renferme la clé publique reliée à la suite de chiffrement choisie précédemment. Le protocole *Alert* signalera une erreur si le certificat n'est pas compris ;
- l'ensemble de la chaîne de certification, y compris le certificat de l'autorité maîtresse.

Si le serveur ne possède pas de certificat, il transmet le message *ServerKeyExchange* à la place du message *Certificate*. Ce message comporte les paramètres qui seront utilisés pour chiffrer *PreMasterSecret*.

Le serveur peut être conduit à transmettre un message *ServerKeyExchange*, même s'il possède un certificat, lorsque celui-ci est uniquement un certificat de signature. La clé privée pour chiffrer *PreMasterSecret* correspondra à la clé publique contenue dans le certificat de signature scellé par l'autorité certifiante.

Ainsi, quand un certificat de signature est employé, le serveur transmet deux messages consécutifs :

1. Le message *Certificate* renferme la clé publique du serveur dûment scellé par la clé privée de l'autorité de certification. Le client vérifie la validité du certificat avec la clé publique de l'autorité de certification.
2. Le message *ServerKeyExchange* contient les paramètres publics de l'algorithme d'échange de la clé secrète (qui est utilisée pour le chiffrement symétrique). Le serveur scelle (signe) ces paramètres à l'aide de la clé privée de signature. À la réception de ce second message, le client vérifie la signature du serveur à l'aide de la clé publique déjà reçue.

Après s'être authentifié, le serveur peut demander au client d'agir de même en envoyant le message *CertificateRequest*. Ce message contient une liste des types de certificats requis, organisée selon l'ordre de préférence des autorités de certification.

Suite à ces échanges, le serveur envoie un message *ServerHelloDone* qui signifie au client qu'il a terminé et qu'il attend désormais une réponse.

Échanges de secrets

Si le serveur a demandé au client de s'authentifier en lui envoyant le message *CertificateRequest*, le client doit répliquer en incluant son certificat, s'il en possède un, dans le message *Certificate*. Si le client n'est pas en mesure de donner un certificat, sa réplique sera le message *no_certificate*. Cette alerte n'étant qu'un avertissement (*warning*) et non une erreur fatale, le serveur n'interrompra la procédure que s'il considère l'authentification du client indispensable.

Le client envoie ensuite le message *ClientKeyExchange* qui contient le paramètre *PreMasterSecret* chiffré. Lorsque le client est certifié et que l'échange de clés se fait par la méthode Diffie-Hellman sur courbe elliptique, les paramètres nécessaires ont déjà été échangés, et l'envoi de ce message peut être supprimé.

On voit donc que le contenu des messages *Certificate*, *ServerKeyExchange*, *ClientKeyExchange* varie selon les choix des méthodes d'échanges de clés et des types de certificats. Étudions en détail le déroulement des échanges du Handshake.

Vérification et confirmation par le serveur

Si le client est certifié et que le certificat lui confère la faculté d'apposer une signature numérique, ce qui est le cas pour les certificats RSA, il envoie également un message de confirmation explicite : *CertificateVerify*. Ce message, dont le but est de permettre au serveur de vérifier le sceau du client, contient le condensât de tous les messages depuis *ClientHello*, à l'exception du message conteneur ; ce condensât est chiffré avec la clé privée du client.

Le client fait ensuite appel au protocole ChangeCipherSpec (CCS) afin de déclencher le chiffrement des échanges selon les choix effectués dans les deux phases précédentes.

Le client envoie immédiatement le message *Finished*. Ce message est un condensé de tous les messages du Handshake que le client a déjà envoyés au serveur depuis *ClientHello* en employant les attributs cryptographiques qui viennent d'être négociés. On déjoue ainsi les attaques de subtilisation en vérifiant l'intégrité de l'ensemble des échanges.

Après avoir émis le message *Finished*, le client envoie les messages d'application chiffrés sans attendre un accusé de réception. À la réception du message *Finished*, le serveur tente de reproduire le même condensât à partir des messages précédemment reçus. Il compare le résultat au contenu du message *Finished* qu'il vient de recevoir du client. Cette étape lui permet de déceler si un intrus a intercepté et modifié les messages.

Le serveur envoie à son tour les messages *ChangeCipherSpec* et *Finished*. Là encore, le message *Finished* est généré à partir de tous les messages que le serveur a précédemment envoyés et il est transmis chiffré. Le serveur commence aussitôt à envoyer ses données d'application.

Le tableau 4 donne la liste chronologique des messages du protocole Handshake et leur signification. Les données en italique sont celles que WTLS a ajouté à celles de SSL.

Tableau 4 - Les différents messages du protocole Handshake par ordre chronologique

Message	Type de message	Sens de transmission	Signification
HelloRequest	optionnel	serveur → client	Ce message demande au client d'entamer le Handshake.
ClientHello	obligatoire	client → serveur	Ce message contient : – le numéro de version du protocole WTLS ; que prend en charge le client – le nombre aléatoire : client_random ; – l'identificateur de session : session_ID ; – *la liste des suites de méthodes d'échanges de clés et des identités que prend en charge le client ;* – *la liste des certificats admis par le client* ; – la liste des méthodes de compression choisies par le client ; – *le mode du numérotage des séquences de paquets ;* – *l'intervalle de mise à jour de paramètres de chiffrement*.
ServerHello	obligatoire	serveur → client	Ce message contient : – le numéro de version du protocole WTLS ; – un nombre aléatoire : serveur_random ; – l'identificateur de session : session_ID ; – une suite de chiffrement ; – une méthode de compression ; – *la version du protocole que le serveur accepte d'utiliser suite à la proposition du client ;* – *le mode du numérotage des séquences de paquets ;* – *l'intervalle de mise à jour de paramètres de chiffrement*.
Certificate	optionnel	serveur → client client → serveur	Ce message contient le certificat du serveur ou celui du client si le serveur le lui réclame et que le client en possède un.
ServerKeyExchange	optionnel	serveur → client	Ce message n'est envoyé par le serveur que si celui ci ne possède pas de certificat ou que si son certificat est uniquement de signature, afin de sécuriser l'échange de PreMasterSecret .
CertificateRequest	optionnel	serveur → client	Par ce message, le serveur réclame un certificat au client.

ServerHelloDone	obligatoire	serveur → client	Ce message signale la fin de l'envoi des messages ServerHello et subséquents.
ClientKeyExchange	*optionnel*	client → serveur	Ce message contient le PreMasterSecret chiffré *selon la méthode d'échange de clés choisie. Si l'échange de clés se fait selon la méthode Diffie-Hellman sur courbe elliptique et que le client possède un certificat, ce message peut être supprimé.*
CertificateVerify	optionnel	client → serveur	Ce message permet une vérification explicite du certificat du client.
Finished	obligatoire	serveur → client client → serveur	Ce message signale la fin du protocole Handshake et le début de l'émission des données protégées avec les nouveaux paramètres négociés.

Ouverture d'une connexion

Si la session WTLS est établie, les flux TCP peuvent transiter dans les deux sens. L'ouverture d'une nouvelle connexion consiste à rafraîchir les paramètres *client_random* et *server_random* à l'aide des messages *ClientHello* et *ServerHello*, tout en préservant les algorithmes de chiffrement et de hachage déjà choisis. Une nouvelle authentification est donc évitée et, contrairement à ce qui se passe pendant l'ouverture d'une session, les messages *ClientHello* et *ServerHello* sont chiffrés. Les échanges sont illustrés dans la figure 16–6. Remarquons que l'échange de données ne commence que lorsque chacune des deux parties a reçu le message Finished.

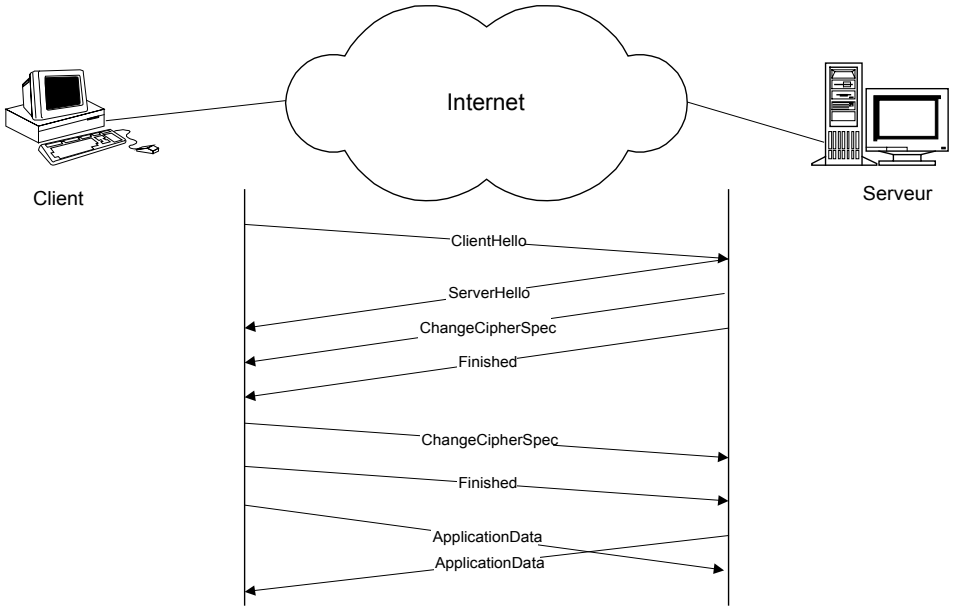


Figure 16–6. : Messages échangés pour l'ouverture d'une connexion

Le message *ClientHello* contient l'identificateur de la session sur laquelle on établit la connexion. Si cet identificateur ne figure pas dans les tables du serveur, soit parce qu'il est erroné, soit parce que la session à laquelle il renvoie est close, le client n'est pas rejeté et le serveur entame un Handshake complet afin d'établir une nouvelle session WTLS.

Le client et le serveur confirment leur accord en envoyant chacun un message *ChangeCipherSpec* et terminent le Handshake abrégé par le message *Finished* comme précédemment.

NOTE — Les spécifications ne sont pas claires concernant l'action qu'il faut mener si les nouveaux échanges introduisent une nouvelle suite de chiffrement.

L'ouverture de connexion après reprise d'une session suspendue est appelée « rafraîchissement d'une session ». Cette opération s'effectue selon la même procédure de Handshake simplifiée présentée ci-dessus.

Le protocole ChangeCipherSpec (CCS)

Le protocole ChangeCipherSpec (CCS) comprend un seul et unique message, qui porte le même nom que le protocole et tient sur un octet. Il a pour objectif d'indiquer au protocole Record la mise en place effective des algorithmes cryptographiques qui viennent d'être négociés afin que ce dernier entame le chiffrement.

Jusqu'à l'envoi de ce message, le chiffrement incombe au protocole Handshake. À la suite de ce message, la couche du protocole Record du côté émetteur doit modifier ses attributs d'écriture, c'est-à-dire la méthode de chiffrement des messages émis. Du côté récepteur de l'entité distante, la couche de protocole Record change ses attributs en lecture afin de pouvoir déchiffrer les messages reçus.

Le protocole Record

Le protocole Record intervient seulement après l'émission du message *ChangeCipherSpec*. Pendant la phase d'établissement de la session, le rôle de la couche du protocole Record est d'encapsuler les données du Handshake et de les transmettre sans aucune modification vers le protocole de transaction sans fil WTP. Contrairement à SSL, c'est le protocole WDP et non Record qui prend en charge la fragmentation des données.

Le protocole Record ajoute un en-tête de 1 octet à 5 octets à chaque message en provenance des couches supérieures. Cet en-tête, qui n'est pas chiffré, indique le type du message selon son origine : sous-protocoles Handshake, Alert, CCS, ou données d'applications, par exemple HTTP ou FTP.

L'organisation des champs de l'en-tête de Record est indiquée dans la figure 16-7. On remarque que le bit nº3 est réservé pour un usage ultérieur.

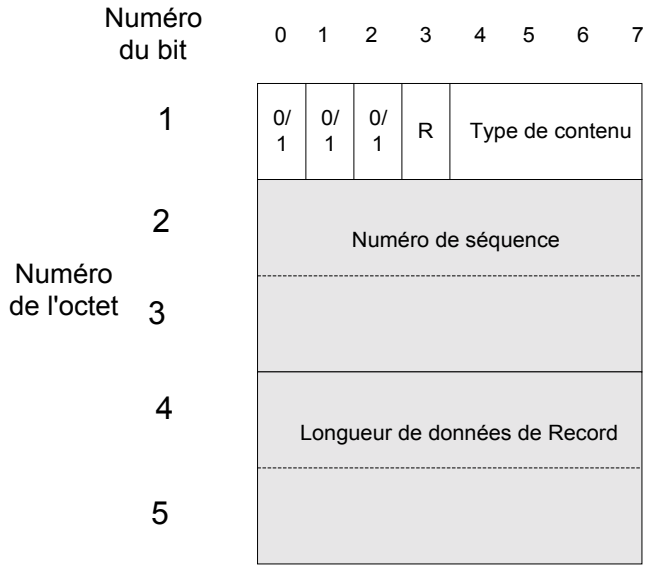


Figure 16–7. : Organisation de l'en-tête du protocole Record dans WTLS

Les deux premiers octets des champs facultatifs contiennent le numéro de séquence ; les deux derniers indiquent la longueur du bloc de données qui ont été encapsulées. La présence du champs de longueur de bloc est indiqué par un « 1 » dans le bit le plus à droite du premier octet (le bit "0"). De même, la présence la présence d'un « 1 » dans le bit "1" signale la présence d'un numéro de séquence. La présence d'un « 1 » dans le 3^{e} bit de gauche signale un traitement de chiffrement, de compression ou de protection de l'intégrité. Ainsi, un « 0 » dans ce cas indique que le message est envoyé en clair, cas des messages Handshake transmis en début de session ou de certains messages d'Alert. Enfin, ce sont les bits 4 à 7 qui indiquent le type du contenu.

Pendant la phase de chiffrement des données, le protocole Record reçoit les données des couches supérieures (Handshake, Alert, CCS, HTTP, FTP...) et les transmet au protocole de datagramme sans fil WDP après avoir effectué dans l'ordre les tâches suivantes :

1. la compression des données, fonction prévue mais non pourvue actuellement ;
2. la génération d'un condensât pour assurer le service d'intégrité ;
3. le chiffrement des données pour assurer le service de confidentialité.

Les tâches inverses sont effectuées du côté récepteur : déchiffrement, vérification de l'intégrité et décompression. Si le condensât calculé n'est pas identique à celui qui est reçu, le protocole Record invoque le protocole Alert afin de relayer le message d'erreur adéquat à l'entité émettrice.

Le protocole Alert

Le protocole Alert sert essentiellement à générer des messages d'alerte suite aux erreurs de parcours, et à signaler les changements d'état tels que la fermeture d'une connexion. Comme tout autre message provenant des couches supérieures, la couche Record se charge de chiffrer les messages avec les attributs de chiffrement en vigueur.

Selon la gravité de la menace, l'alerte peut constituer un simple avertissement ou déclencher l'abandon de la session. Un message d'avertissement est une simple mise en garde qui n'exige aucune action particulière. En revanche, après un message fatal, l'entité émettrice doit clore promptement la connexion sans attendre l'acquittement du partenaire. De son côté, le récepteur clôt la connexion dès l'arrivée du message d'alerte. Comme les messages de niveau « fatal » provoquent l'abandon d'une session, WTLS est sensible aux attaques du type « déni de service », si un intrus parvient à substituer aux messages réglementaires d'autres message non conformes, provoquant de la sorte la rupture de la session.

Le protocole Alert peut être invoqué :

- par l'application, quand elle souhaite, par exemple, signaler la fin de la connexion ;
- par le protocole Handshake, suite à un problème survenu au cours de son déroulement ;
- par la couche Record directement, par exemple si l'intégrité d'un message est douteuse.

Le tableau 5 contient la liste des messages du protocole Alert classés par ordre alphabétique. Les lignes en caractères italiques indiquent les messages supplémentaires de WTLS par rapport à SSL, alors que les lignes ombrées signalent les messages déjà présents dans TLS. WTLS introduit un nouveau type de message (message critique) dont le traitement est laissé à l'appréciation du destinataire. Le message *close_notify* de SSL a été scindé en deux messages pour distinguer l'interruption d'une connexion ou d'une session.

Tableau 5 - Messages du protocole Alert

Message	Contexte	Type
access_denied	*certificat validé mais accès refusé*	*fatal*
bad_certificate	échec de vérification d'un certificat	fatal
bad_record_mac	réception d'un MAC erroné	fatal
certificate_expired	certificat périmé	fatal
certificate_revoked	certificat mis en opposition (révoqué)	fatal
certificate_unknown	certificat invalide pour d'autres motifs que ceux précisés précédemment	fatal
connection_close_notify	interruption volontaire de connexion	fatal
decode_error	*message non décodé pour taille incorrect ou paramètre hors fourchette*	*fatal ou critique*
decompression_failure	les données appliquées à la fonction de décompression sont invalides (par exemple, trop longues)	fatal
decrypt_error	*erreur de déchiffrement*	*fatal*

decryption_failed	*arrivée d'un bloc avec un paraphe incorrect*	*avertissement*
disabled_key_id	*clés fournies par le client mises en opposition*	*alerte critique*
duplicate_finished_received	*un second Finished reçu par le serveur*	*avertissement*
export_restriction	*négociation non conforme aux restrictions d'exportation*	*fatal*
handshake_ failure	impossibilité de négocier des paramètres satisfaisants	fatal
illegal_parameter	un paramètre échangé au cours du protocole Handshake dépasse les bornes admises ou ne concorde pas avec les autres paramètres	fatal
internal_error	*erreur interne indépendante du protocole ou du pair*	*fatal ou critique*
insufficient_security	*suite proposée par le client en deçà des exigences de sécurité du serveur*	*fatal*
key_exchange_disabled	*échange de clés anonyme neutralisé*	*
no_certificate	réponse négative à une requête de certificat	avertissement ou fatal
no_connection	*message reçu sur une connexion non sécurisée*	*fatal ou critique*
no_renegotiation	*renégociation refusée après la prise de contact initiale*	-
protocol_version	*version requise non prise en charge*	*fatal*
record_overflow	*bloc reçu dépassant la taille limite*	*avertissement*
session_close_notify	interruption volontaire de connexion	fatal
session_not_ready	*session sécurisé non disponible pour raisons administratives*	*alerte critique*
unexpected_message	arrivée inopportune d'un message	fatal **ou** *critique*
unknown_ca	*autorité certifiante non reconnue*	*fatal*
unknown_key_id	*clés du client non reconnues*	*alerte fatale*
unknown_parameter_index	*suite d'échange de clés non reconnue par le serveur*	*alerte critique*
unsupported_certificate	certificat reçu non reconnu par le destinataire	avertissement ou fatal
user_canceled	*prise de contact annulée sans faille de protocole (message suivant doit être connexion_close_notify)*	*avertissement*

***** Type laissé à l'appréciation de l'expéditeur.

Comme pour SSL/TLS, les messages du protocole Alert ne sont pas authentifiés. Comme eux, WTS est susceptible aux attaques par troncature, lorsqu'un attaquant envoie de faux messages avec les mêmes numéros de séquence que des messages légitimes pour provoquer l'abandon de ces derniers [SAA 00].

Récapitulatif des différences entre SSL et WTLS

Handshake

Par rapport à SSL, WTLS montre les différence suivantes :

1. Une volée de messages du Handshake WTLS peut être consolidée en un seul envoi.
2. La retransmission des messages du Handshake est permise dans certaines conditions.
3. Le message *ClientHello* ajoute aux champs employés par SSL les champs suivants :
   - la liste des suites de méthodes d'échanges de clés et des identités que prend en charge le client ;
   - la liste des certificats admis par le client ;
   - la liste des méthodes de compression choisies par le client ;
   - le mode du numérotage des séquences de paquets ;
   - l'intervalle de mise à jour de paramètres de chiffrement.
4. Le message *ServerHello* contient en plus des champs utilisés par SSL :
   - la version du protocole que le serveur accepte d'utiliser (la plus récente parmi celles qu'a proposées le client dans le message *ClientHello* si elle se trouve parmi celles que prend en charge le serveur) ;
   - le mode du numérotage des séquences de paquets ;
   - l'intervalle de mise à jour de paramètres de chiffrement ;
5. Le contenu des messages *Certificate*, *ServerKeyExchange*, *ClientKeyExchange* varie selon les choix des méthodes d'échanges de clés et des types de certificats.
6. Le message *ClientKeyExchange*, obligatoire dans SSL, est facultatif dans WTLS. Il est superflu si l'échange de clés se fait avec la méthode Diffie-Hellman sur courbe elliptique et que le client est certifié .
7. La taille du message *Finished* est de 12 octets et non de 36 octets.
8. Les numéros de séquence des messages échangés.
9. La transmission de données ne commence qu'après l'échange du message *Finished* de part et d'autre, alors qu'elle est permise, mais non recommandée, dans SSL, juste après l'envoi de ce message.

Alert

Le protocole Alert dans WTLS contient 20 nouveaux messages.

Record

Contrairement à SSL, c'est le protocole WDP et non Record qui prend en charge la fragmentation des données.

Paramètres du chiffrement

1. L'authentification dans WTLS peut se faire à l'aide de trois types de certificats (des certificats X.509, des certificats ANSI X9.68 et des certificats au format propre de WTLS).
2. WTLS retient quelques algorithmes de chiffrement symétrique employés dans SSL (DES_CBC_40, DES_CBC, 3DES_CBC_EDE, IDEA_CBC), ajoute de nouveaux algorithmes (RC5_CBC_40, RC5_CBC_56,RC5_CBC, IDEA_CBC_40, IDEA_CBC_56) et élimine l'algorithme de Fortezza .
3. Les paramètres de chiffrement peuvent être modifiés en cours d'une session, ce qui réduit le risque de cassage des clés à long terme.
4. Les variables ont des tailles réduites ainsi que le montre le tableau 6.

Tableau 6 - Comparaison entre la taille des différentes variables de SSL et WTLS

Variable	Taille en octets	
	SSL	WTLS
clé de chiffrement symétrique	5 à 16	5 à 21
client_random	32	16
identificateur de session	3	2
MasterSecret	48	20
numéro de séquence	8	2
PreMasterSecret	48	variable (20 pour RSA)
server_random	32	16

Implémentations

Le WAP correspond à certaines catégories d'application où la mobilité joue un rôle fondamental : accès au système d'information d'une entreprise ou aux services offerts par une banque. Cependant, les applications déjà disponibles sur la Toile devront être réécrites, car les pages WAP devront être décrites à l'aide du langage de balisage pour terminaux mobiles WML (*Wireless Markup Language*), plus adapté que HTML aux contraintes de la communication mobile (mémoire restreinte, puissance de calcul réduite, alimentation en batterie de courte durée, etc.). La maintenance des applications se fera sous deux formes, une pour les sédentaires et l'autre pour les usagers mobiles.

L'interface WAP/WEB doit effectuer deux opérations :

1. la conversion de protocole entre le réseau mobile et l'Internet (de SMS ou GPRS vers IP et de WDP vers TCP, ou vice-versa) ;
2. la sécurisation de bout en bout. TLS/SSL et WTLS étant également des protocoles de point-à-point, on devra obligatoirement déchiffrer les messages échangés à mi-chemin

puis les rechiffrer selon le protocole suivant. Ceci demande l'intervention d'un tiers de confiance qui doit être sécurisé contre toute attaque éventuelle.

Il existe plusieurs possibilités pour l'emplacement de la passerelle de conversion :

1. L'entreprise peut déployer en interne la passerelle et la plate-forme WAP derrière le coupe-feu et gérer l'accès distant. L'utilisateur nomade se connecte alors au numéro fourni par son employeur et c'est ce dernier qui se charge de tous les maillons de la chaîne de sécurité. L'avantage pour l'entreprise est d'être tout à fait indépendante de l'opérateur dont le rôle se limite à la fourniture du canal radio. Cependant, l'entreprise doit pouvoir maîtriser les expertises nécessaires pour maintenir le système en phase avec l'évolution technologique et en gérer l'opération.
2. L'opérateur téléphonique se charge du canal de communication radio et de l'accès distant et réoriente le trafic vers le réseau de l'entreprise après avoir répondu à l'appel.
3. L'opérateur téléphonique se charge en plus du canal radio, du contrôle de la passerelle en amont du coupe-feu de l'entreprise. Cependant, la sécurité de la passerelle de l'opérateur est le point faible de cette architecture, car les textes y sont déchiffrés puis rechiffrés.
4. Enfin, l'entreprise se décharge sur l'opérateur téléphonique de toute la convertibilité WAP/WEB, c'est-à-dire, de la passerelle et de l'hébergement des applications des entreprises.

Évaluation

Le protocole SSL a dépassé le cadre des applications transactionnelles habituelles, comme le prouve son adoption par le consortium WAP (*Wireless Application Protocol*) pour les communications radio. On peut envisager dans un proche avenir des cartes à puce avec une version du protocole à disposition des usagers itinérants.

Parmi les points forts de WTLS citons la possibilité de fonctionner sur des canaux non fiables et la possibilité de modifier les paramètres de chiffrement pendant une session afin de réduire le risque de casser les clés dans de session de longue durée.

Cependant, il faut associer WTLS à SSL, ce qui risque d'alourdir l'architecture des systèmes de communication.

Une infrastructure de certification à clé publique doit être installé sous le contrôle des banques. Les difficultés d'interopérabilité et certification croisée. Un module d'identification sans fil WIM (*Wireless Indentification Module*) comprendra les clés nécessaires et le certificat à clé publique.

Enfin, le fonctionnement de WTLS est sujet à caution dans certains cas, surtout lorsque les algorithmes respectant les limites de chiffrement sont employés [SAA 00].

ANNEXE 1

Structures des messages du Handshake de WTLS

Cette annexe résume les spécifications de WTLS pour les messages du Handshake. Chaque message contient soit des champs simples du type *uint8*, *uint16*, *uint24* ou *opaque*, soit des champs complexes composés à partir de champs simples. Les éléments de type *uint8*, *uint16*, *uint24* correspondent respectivement à des éléments de 1, 2 et 3 octets. Un élément de type *opaque* est un élément de 1 octet qui contient des données chiffrées.

Conformément aux notations introduites dans les spécifications du protocole SSL, un tableau de longueur constante et comportant des éléments de type *Type_Simple* sera noté :

Type_Simple T[n]

où *n* correspond au nombre d'octets constituant le tableau (et non pas au nombre d'éléments de type *Type_Simple* présents). Ainsi, si *Type_Simple* est du type *uint16* (c'est à dire, constitué de 2 octets), la valeur de *n* est de 4 pour un tableau *T* comprenant deux éléments de type *Type_Simple*.

Les tableaux de longueur variable sont notés :

Type_Simple T'<n_1…n_2>

où n_1 et n_2 sont respectivement les nombres minimal et maximal d'octets que peut contenir le tableau *T'*.

Les messages échangés au cours du protocole Handshake ont tous un en-tête possédant deux champs :

- un identificateur du type du message, codé sur 1 octet ;
- la longueur du message, codée sur 2 octets et non sur 3 comme pour SSL.

Dans ce qui suit, on utilise les caractères en gras pour marquer les différences entre SSL et WTLS.

En-tête

```
struct {
   HandshakeType msg_type;
   uint16 length;
   select (HandshakeType)  {
      case hello_request: HelloRequest;
      case client_hello: ClientHello;
      case server_hello: ServerHello;
      case certificate: Certificate;
      case server_key_exchange: ServerKeyExchange;
      case certificate_request: CertificateRequest;
      case server_hello_done: ServerHelloDone;
      case certificate_request: CertificateRequest;
      case client_key_exchange: ClientKeyExchange;
      case finished: Finished;
   } body;
} Handshake;

enum {
   hello_request(0), client_hello(1), server_hello(2),
   certificate(11), server_key_exchange(12), certificate_request(13),
   server_hello_done(14), certificate_verify(15),
client_key_exchange(16),
   finished(20), (255)
} HandshakeType;

enum {server (1), client (2) } ConnectionEnd ;

enum {stream (1), block (2), (255) } CipherType ;
enum {true, false} IsExportable ;
enum {off(0), implicit (1), explicit (2), 255 } SequenceNumberMode ;
```

Message HelloRequest

```
struct {} HelloRequest;
```

Message ClientHello

```
struct {
   uint8 client_version; /* la version*/
   Random random; /* nombre aléatoire généré par le client*/
   SessionID session_id; /* identificateur de la session */
   KeyExchangeId client_key_ids<0..2^16 -1>; /* clés et identités du
client*/
   KeyExchangeId trusted_key_ids<0.. 2^16 -1>; /* certificats du client */
   CipherSuite cipher suites<2…2^8 -1>; /* suites de chiffrement proposées
par le client */
   CompressionMethod compression_methods<1…2^8-1>; /*méthode de compression
*/
   SequenceNumberMode sequence_number_mode; /* mode d'usage des numéros
de séquence dans les messages de Record */
   uint8 key_refresh; /* intervalle de rafraîchissement des paramètres de
sécurité */
} ClientHello;
```



```
struct {
   uint32 gmt_unix_time;
   opaque random_bytes[12];
} Random

opaque SessionID<0..8>;

struct {
   KeyExchangeSuite key_exchange_suite;
   ParameterSpecifier parameter_specifier;
   Identifier identifier;
} KeyExchangeId;

uint8 KeyExchangeSuite /* selection de la méthode d'échange de clés*/

struct {
   ParameterIndex parameter_index; /* indique la méthode d'échange de
clés*/
   Select (parameter_index) {
      case 255: ParameterSet parameter_set;/* les paramètres Diffie-
Hellman ou ECDH sont indiqués explicitement dans les champs suivants*/
      default: struct {}; /* les paramètres sont indiqués implicitement
dans les combinaisons retenues par la spécification de WTLS */
   }
} ParameterSpecifier;

uint8 ParameterIndex;

struct {
   uint16 length; /* la taille en octets des données comprises dans la
structure ParameterSet */
   select (PublicKeyAlgorithm) {
      case rsa: struct {};
      case diffie_hellman: DHParameters params;
      case elliptic_curve: ECParameters params;
   }
} ParameterSet;

enum {rsa, diffie_hellman, elliptic curve} PublicKeyAlgorithm;


struct {       /* paramètres pour les échanges Diffie-Hellman */
   uint8 dh_e;
   opaque dh_p<1..2^16-1>; /* p est le nombre premier module des calculs*/
   opaque dh_g<1..2^16-1>; /* g est l'entier primitif par rapport à p */
} DHParameters;

/* Diffie-Hillman sur courbe elliptique */

enum {ec_prime_p(1), ec_characteristic_two(2), (255) } ECFieldID;

enum {ec_basis_onb(1), ec_basis_trinomial(2), ec_basis_pentanomial(3),
   ec_basis_polynomial (4) } ECBasisType;

struct {
   opaque a <1..2^8 -1>; /* a, b coefficients de la courbe elliptique */
   opaque b <1..2^8 -1>;
```



```
   opaque seed <1..2^8 -1>; /* paramètre facultatif pour calculer les
coefficients de la courbe. */
} ECCurve;

struct {
   opaque point <1..2^8 -1>;
} ECPoint;

struct {
   ECFieldID field;
   select (field) {
   case ec_prime_p:
      opaque prime_p <1..2^8 -1>;
   case ec_characteristics_two:
      uint16 m;
      ECBasisType basis;
      select (basis) {
      case ec_basis_onb:
         struct { };
      case ec_trinomial:
         unint16 k;
      case ec_pentanomial:
         uint16 k1;
         uint16 k2;
         uint16 k3;
      case ec_basis_polynomial:
         opaque irreducible <1..2^8 -1>;
      };
   };
ECCurve curve;
ECPoint base;
opaque order <1..2^8 -1>;
opaque cofactor <1..2^8 -1>;
} ECParameters
/* L'identification dans WTLS peut être déclinée de plusieurs façons : à
l'aide de noms distinctifs du type X.509, soit à l'aide du condensât SHA-
1 de la clé publique, soit à l'aide d'une identité binaire secrète connue
des deux intervenants, soit à l'aide d'une série de caractères dans un
alphabet reconnue par les deux parties. Il est possible aussi de ne pas
décliner d'identité. */

struct {
   IdentifierType identifier_type;
   select (identifier_type) {
      case null: struct {};
      case text:
         CharacterSet character_set;
         opaque name<1.. 2^8 -1>;
      case binary: opaque identifier<1..2^8 -1>;
      case key_hash_sha:opaque_key_hash[20];
      case x509_name:opaque distinguished_name<1.. 2^8 -1>; /* remplace
opaque DistinguishedName<3...2^16-1> dans SSL*/
} Identifier;

enum ( null (0), text (1), binary (2), key_hash_sha(254), x509_name(255)}
   IdentifierType;
```



```
uint16 CharacterSet; /* le jeu de caractères utilisé pour représenter
l'indentifiant textuel*/


struct {
BulkCipherAlgorithm bulk_cipher_algorithm;
MACAlgorithm mac_algorithm;
} CipherSuite;

uint8 BulkCipherAlgorithm ;
uint8 MACAlgorithm ;

enum {null(0), (255)} CompressionMethod;
```

Message ServerHello

```
struct {
   uint8 server_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suite;
   CompressionMethod compression_method;
   SequenceNumberMode sequence_number_mode;
   uint8 key_refresh;
} ServerHello;
```

Message Certificate

```
/* Le type de ce message est Certificates dans WTLS et non Certificate
comme pour SSL*/

struct {
   Certificate certificate_list<1..2^16-1>;
} Certificates;


struct {
   CertificateFormat certificate_format:
   select (certificate_format) {
      case WTLSCert: WTLSCertificate;
      case X509Cert: X509Certificate;
      case X988Cert: X968Certificate;
   }
}Certificate

enum { WTLCert(1), X509Cert(2), X968Cert(3), (255)} CertificateFormat;

opaque X509Certificate<1..2^16 -1>;

opaque X969Certificate<1..2^16 -1>;

struct {
   ToBeSigned Certificate to_be_signed_certificate;
   Signature signature; /*sceau*/
} WTLSCertificate; /* Cette structure de certificat économise la bande
passante */
```



```
struct {
   uint8 certificate_version;
   SignatureAlgorithm signature_algorithm
   Identifier issuer;
   uint32 valid_not_before;
   uint32 valid_not_after;
   Identifier subject;
   PublicKeyType public_key_type;
   ParameterSpecifier parameter_specifier;
   PublicKey public_key;
} ToBeSigned Certificate;

enum {anonymous(0),ecdsa_sha(1);rsa_sha(2), (255)} SignatureAlgorithm;

select (SignatureAlgorithm){
   case anonymous: ();
   case ecdsa_sha:
      digitally-signed struct {
         opaque sha_hash[20]; /* condensât SHA-1 de données à sceller */
      }
   case rsa_sha:
      digitally-signed struct {
         opaque sha_hash[20]; /* condensât SHA-1 de données à sceller */
} Signature; /* sceau (condensât chiffré avec la clé privée de l'émetteur
*/

enumb ( rsa(2), ecdh(3), ecdsa(4), (255) } PublicKeyType;

struct {
   select (PublicKeyType) {
      case ecdh: ECPublicKey;
      case ecdsa: ECPublicKey;
      case rsa: RSAPublicKey;
} PublicKey;

ECPoint ECPublicKey;
struct {
   opaque rsa_exponent<1..2^16-1>;
   opaque rsa_modulus<1.. 2^16-1>;
} RSAPublicKey;
```

Message ServerKeyExchange

```
struct   {
   ParameterSpecifier parameter_specifier;
   select (KeyExchangeAlgorithm) {
      case rsa_anon:
         RSAPublicKey params;
      case dh_anon:
         DHPublicKey params;
      case ecdh_anon:
         EcPublicKey params;

} ServerKeyExchange ;
```



```
enum { rsa, rsa_anon, dh_anon, ecdh_anon, ecdh_ecdsa}
   KeyExchangeAlgorithm ;

struct {
   opaque dh_Y<1...2^16-1>;
} DHPublicKey
```

Message CertificateRequest

```
struct {
   KeyExchangeId trusted_authorities<0..2^16-1>;
}CertificateRequest;
```

Message ServerHelloDone

```
struct {} ServerHelloDone;
```

Message ClientKeyExchange

```
struct {
   uint8 client_version;
   opaque random[19];
} Secret;

struct {
public-key-encypted Secret secret;
}EncryptedSecret;

struct   {
   select (KeyExchangeAlgorithm) {
      case rsa: RSAEncryptedSecret param;
      case rsa_anon: RSAEncryptedSecret param; /* clé publique du
client*/
      case dh_anon: DHPublicKey param; /* clé publique du client*/
      case ecdh_anon: ECPublicKey param; /* clé publique du client*/
      case ecdh_ecdsa: ECPublicKey param; /* clé publique du client*/
;
   } exchange_keys;
} ClientKeyExchange;
```

Message CertificateVerify

```
struct {
Signature signature;
} CertificateVerify;
```

Message Finished

```
struct{
   opaque verify_data[12];
} Finished;
```


ANNEXE 2

Références

[SAA 00] M.-J. Saarinen, “Attacks against the WAP WTLS protocol”. http://www.jyu.fi/~mjos/wtls.pdf.

[WTLS 99] Wireless Application Forum, *Wireless Application Protocol – Wireless Transport Layer Security Specification WAP WTLS*, version 05-nov-1999.

Index

