Groupe de travail Réseau
Request for Comments : 2246
Catégorie :En cours de normalisation
Traduction Claude Brière de L'Isle

T. Dierks, Certicom
C. Allen, Certicom
janvier 1999

Protocole TLS, version 1.0

Statut du présent mémoire

Le présent document spécifie un protocole de l'Internet en cours de normalisation pour la communauté de l'Internet, et appelle à des discussions et suggestions pour son amélioration. Prière de se référer à l'édition en cours des "Normes officielles des protocoles de l'Internet" (STD 1) pour connaître l'état de la normalisation et le statut de ce protocole. La distribution du présent mémoire n'est soumise à aucune restriction.

Notice de Copyright

Copyright (C) The Internet Society (1999). Tous droits réservés.

Résumé

Le présent document spécifie la version 1.0 du protocole de sécurité de la couche transport (TLS, *Transport Layer Security*). Le protocole TLS fournit la confidentialité des communications sur l'Internet. Le protocole permet aux applications client/serveur de communiquer d'une façon qui est conçue pour empêcher l'espionnage, l'altération ou la falsification du message.

Table des matières

1. Introduction....2
2. Objectifs....2
3. Objectifs du présent document....3
4. Langage de présentation....3
  4.1 Taille du bloc de base....3
  4.2 Divers....3
  4.3 Vecteurs....4
  4.4 Nombres....4
  4.5 Énumérations....4
  4.6 Types construits....5
  4.7 Attributs cryptographiques....6
  4.8 Constantes....7
5. HMAC et la fonction pseudo aléatoire....7
6. Protocole d'enregistrement TLS....8
  6.1 États de connexion....8
  6.2 Couche d'enregistrement....10
  6.3 Calcul des clés....13
7. Protocole de prise de contact TLS....14
  7.1 Protocole de changement de spécification de chiffrement....15
  7.2 Protocole d'alerte....15
  7.3 Présentation du protocole de prise de contact....18
  7.4 Protocole de prise de contact....19
8. Calculs cryptographiques....29
  8.1 Calcul du secret maître....29
9. Suites de chiffrement obligatoires....29
10. Protocole des données d'application....29
A. Valeurs des constantes du protocole....30
  A.1 Couche Enregistrement....30
  A.2 Message Changer la spécification de chiffrement....30
  A.3 Messages d'alerte....31
  A.4 Protocole de prise de contact....31
  A.5 Suite de chiffrement....34
  A.6 Paramètres de sécurité....35
B. Glossaire....35
C. Définitions des suites de chiffrement....38

D. Notes de mise en œuvre.....39
D.1 Clés RSA temporaires.....39
D.2 Génération de nombres aléatoires et germes.....40
D.3 Certificats et authentification.....40
D.4 Suites de chiffrement.....40
E. Rétro compatibilité avec SSL.....40
E.1 Client hello version 2.....41
E.2 Éviter la dégradation de version par interposition.....42
F. Analyse de la sécurité.....42
F.1 Protocole de prise de contact.....42
F.2 Protection des données d'application.....44
F.3 Notes finales.....45
G. Déclaration de brevets.....45
Références.....46
Déclaration complète de droits de reproduction.....48

1. Introduction

Le but premier du protocole TLS est de fournir la confidentialité et l'intégrité des données entre deux applications communicantes. Le protocole se compose de deux couches : le protocole d'enregistrement TLS et le protocole de prise de contact TLS. Au niveau inférieur, posé par dessus un protocole de transport fiable (par exemple, [TCP]), il y a le protocole d'enregistrement TLS. Le protocole d'enregistrement TLS fournit la sécurité de la connexion, qui a deux propriétés de base :

- La connexion est privée. Le chiffrement symétrique est utilisé pour le chiffrement des données (par exemple, [DES], [RC4], etc.) Les clés pour ce chiffrement symétrique sont générées de façon univoque pour chaque connexion et se fondent sur un secret négocié par un autre protocole (tel que le protocole de prise de contact TLS). Le protocole d'enregistrement peut aussi être utilisé sans chiffrement.

- La connexion est fiable. Le transport de message comporte une vérification d'intégrité du message qui utilise un MAC *(code d'authentification de message)* à clés. Des fonctions de hachage sécurisé (par exemple, SHA, MD5, etc.) sont utilisées pour les calculs du MAC. Le protocole d'enregistrement peut fonctionner sans MAC, mais n'est généralement utilisé que dans ce mode alors qu'un autre protocole utilise le protocole d'enregistrement comme transport pour négocier les paramètres de sécurité.

Le protocole d'enregistrement TLS est utilisé pour l'encapsulation de divers protocoles de niveau supérieur. Un de ces protocoles encapsulés, le protocole de prise de contact TLS, permet au serveur et au client de s'authentifier mutuellement et de négocier un algorithme et des clés de chiffrement avant que le protocole d'application émette ou reçoive son premier octet de données. Le protocole de prise de contact TLS fournit la sécurité de connexion qui a trois propriétés de base :

- L'identité de l'homologue peut être authentifiée en utilisant la cryptographie à clés asymétriques, ou publiques (par exemple, [RSA], [DSS], etc.). Cette authentification peut être rendue facultative, mais est généralement exigée pour au moins un des homologues.

- La négociation d'un secret partagé est sûre : le secret négocié n'est pas accessible aux espions, et pour toute connexion authentifiée, le secret ne peut pas être obtenu, même par un attaquant qui peut se placer dans le milieu de la connexion.

- La négociation est fiable : aucun attaquant ne peut modifier la communication de négociation sans être détecté par les parties à la communication.

Un avantage de TLS est qu'il est indépendant du protocole d'application. Les protocoles de niveau supérieur peuvent se poser en toute transparence par dessus le protocole TLS. La norme TLS ne spécifie cependant pas comment les protocoles ajoutent de la sécurité avec TLS ; les décisions sur la façon d'initier la prise de contact TLS et comment interpréter les certificats d'authentification échangés sont laissées au jugement des concepteurs et des mises en œuvre des protocoles qui se placent par dessus TLS.

2. Objectifs

Dans l'ordre de leur priorité, les objectifs du protocole TLS sont :

1. La sécurité du chiffrement: TLS devrait être utilisé pour établir une connexion sûre entre deux parties.
2. Interopérabilité : des programmeurs indépendants devraient être capables de développer des applications utilisant TLS et qui pourraient ensuite réussir à échanger des paramètres cryptographiques sans connaissance réciproque de leur code.
3. Extensibilité : TLS cherche à fournir un cadre dans lequel de nouvelles clés publiques et méthodes de chiffrement en vrac peuvent être incorporées en tant que de besoin. Cela va aussi réaliser deux sous objectifs : prévenir la nécessité de créer un nouveau protocole (et de risquer l'introduction de possibles nouvelles faiblesses) et éviter la nécessité de mettre en œuvre une bibliothèque de sécurité entièrement nouvelle.
4. Efficacité relative : les opérations de chiffrement tendent à être de fortes consommatrices de CPU, en particulier les opérations de clés publiques. Pour cette raison, le protocole TLS a incorporé un schéma facultatif de mise en antémémoire de session pour réduire le nombre de connexions qui doivent être établies à partir de rien. De plus, on a pris soin de réduire l'activité de réseau.

3. Objectifs du présent document

Le présent document et le protocole TLS lui-même se fondent sur la spécification du protocole SSL 3.0 telle que publiée par Netscape. Les différences entre le présent protocole et SSL 3.0 ne sont pas énormes, mais suffisamment significatives pour que TLS 1.0 et SSL 3.0 n'interopèrent pas (bien que TLS 1.0 n'incorpore pas de mécanisme par lequel une mise en œuvre de TLS puisse revenir à SSL 3.0). Le présent document est destiné principalement aux lecteurs qui veulent mettre en œuvre le protocole et pour ceux qui en font l'analyse cryptographique. La spécification a été écrite dans ce sens, et est destinée à refléter les besoins de ces deux groupes. Pour cette raison, beaucoup des structures et règles des données qui dépendent des algorithmes sont incluses dans le corps du texte (et non dans un appendice) pour y donner plus facilement accès.

Le présent document n'est pas destiné à fournir de détails sur les définition de service ni d'interface, bien qu'il couvre bien les zones de politique choisies lorsqu'elles sont nécessaires au maintien d'une solide sécurité.

4. Langage de présentation

Le présent document ne traite pas du formatage des données dans une représentation externe. La syntaxe de présentation définie suivante, très basique et assez terre à terre, sera utilisée. La syntaxe provient de plusieurs sources pour sa structure. Bien qu'elle ressemble au langage de programmation "C" dans sa syntaxe et à [XDR] à la fois par sa syntaxe et ses intentions, il serait hasardeux de tirer trop de parallèles. L'objet de ce langage de présentation est seulement de présenter TLS, et non d'avoir une application générale au delà de cet objectif particulier.

4.1 Taille du bloc de base

La représentation de tous les éléments de données est explicitement spécifiée. La taille du bloc de base des données est un octet (c'est-à-dire 8 bits). Les éléments de données de plusieurs octets sont des enchaînements d'octets, de gauche à droite, du haut vers le bas. Dans le flux d'octets un élément multi octet (un nombre dans l'exemple) est formé par (en utilisant la notation C) :

```
valeur = (octet[0] << 8*(n-1)) | (octet[1] << 8*(n-2)) |... | octet[n-1];
```

Cet ordre des octets pour les valeurs de plusieurs octets est l'ordre ordinaire des octets du réseau, appelé format gros boutien.

4.2 Divers

Les commentaires commencent par "/*" et se terminent par "*/".

Les composants facultatifs sont notés en les incluant dans des doubles crochets "[[ ]]".

Les entités d'un seul octet qui contiennent des données non interprétées sont de type opaque.

4.3 Vecteurs

Un vecteur (système à une seule dimension) est un flux d'éléments de données homogènes. La taille du vecteur peut être spécifiée au moment de sa documentation ou laissée non spécifiée jusqu'au moment de l'exécution. Dans l'un et l'autre cas, la longueur déclare le nombre d'octets, non le nombre d'éléments, dans le vecteur. La syntaxe pour spécifier un nouveau type T' qui est un vecteur de longueur fixe de type T est T "T'[n];".

Ici T' occupe n octets dans le flux des données, n est un multiple de la taille de T. La longueur du vecteur n'est pas incluse dans le flux codé.

Dans l'exemple qui suit, Datum est défini comme étant les trois octets consécutifs que le protocole n'interprète pas, alors que Data est trois Datum consécutifs, consommant un total de neuf octets.

```
opaque Datum[3];          /* trois octets non interprétés */
Datum Data[9];            /* trois vecteurs de trois octets consécutifs */
```

Les vecteurs de longueur variable sont définis en spécifiant une sous gamme des longueurs légales, inclusive, en utilisant la notation <plancher..plafond>. Lorsque elle est codée, la longueur réelle précède le contenu du vecteur dans le flux des octets. La longueur sera sous la forme d'un nombre qui va occuper autant d'octets que nécessaire pour contenir la longueur maximale spécifiée du vecteur (le plafond). Un vecteur de longueur variable avec un champ de longueur réelle de zéro est appelé un vecteur vide.

```
T T'<plancher..plafond>;
```

Dans l'exemple suivant, est obligatoire *(mandatory)* un vecteur qui doit contenir entre 300 et 400 octets de type opaque. Il ne peut jamais être vide. Le champ réel de longueur consomme deux octets, un uint16, suffisant pour représenter la valeur 400 (voir au paragraphe 4.4). D'un autre côté, "longer" peut représenter jusqu'à 800 octets de données, ou 400 éléments uint16, et il peut être vide. Son codage va comporter un champ Longueur réelle de deux octets ajouté au vecteur. La longueur d'un vecteur codé doit être un multiple pair de la longueur d'un seul élément (par exemple, un vecteur de 17 octets de uint16 serait illégal).

```
opaque mandatory<300..400>;   /* le champ longueur est de deux octets, ne peut pas être vide */
uint16 longer<0..800>;        /* de zéro à 400 entiers non signés de 16 bits */
```

4.4 Nombres

Le type de base des données numériques est un octet non signé (uint8). Tous les types de données numériques plus longs sont formés de séries de longueur fixe d'octets enchaînés comme décrit au paragraphe 4.1 et ils sont aussi non signés. Les types numériques suivants sont prédéfinis :

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

Toutes les valeurs, ici et ailleurs dans la présente spécification, sont mémorisées dans l'ordre des octets "du réseau" ou "gros boutien" ; le "uint32" représenté par les octets hexadécimaux 01 02 03 04 est équivalent à la valeur décimale 16 909 060.

4.5 Énumérations

Un type de données éparses supplémentaire est disponible qui est appelé "enum". Un champ de type enum peut seulement prendre les valeurs déclarées dans sa définition. Chaque définition est d'un type différent. Seuls les énumérés du même type peuvent être affectés ou comparés. Chaque élément d'une énumération doit être affecté d'une valeur, comme le montre l'exemple suivant. Comme les éléments de l'énumération ne sont pas ordonnés, ils peuvent recevoir toute valeur unique, dans n'importe quel ordre :

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

Les énumérations occupent autant d'espace dans le flux d'octets que le ferait la valeur ordinale maximale définie. Les définitions suivantes causeraient l'utilisation d'un octet pour porter les champs de type Couleur.

```
enum { rouge(3), bleu(5), blanc(7) } Couleur;
```

On peut facultativement spécifier une valeur dans son étiquette associée pour forcer la définition de largeur sans définir un élément superflu. Dans l'exemple suivant, Le goût va consommer deux octets dans le flux des données mais on ne peut assurer les valeurs que 1, 2 ou 4.

```
enum { doux(1), aigre(2), amer(4), (32000) } Goût;
```

Le nom des éléments d'une énumération a une portée limitée au sein du type défini. Dans le premier exemple, une référence pleinement qualifiée au second élément de l'énumération serait Couleur.bleu. Une telle qualification n'est pas exigée si la cible de l'affectation est bien spécifiée.

```
Couleur couleur = Couleur.bleu;     /* sur spécifié, légal */
Couleur couleur = bleu;             /* correct, type implicite */
```

Pour les énumérations qui ne sont jamais converties en représentation externe, les informations numériques peuvent être omises.

```
enum { bas, moyen, haut } Quantité
```

4.6 Types construits

Les types de structure peuvent être construits à partir de types de primitives par convenance. Chaque spécification déclare un nouveau type unique. La syntaxe pour la définition est très semblable à celle du langage C.

```
struct {
     T1 f1;
     T2 f2;
  ...
  Tn fn;
  } [[T]];
```

Les champs au sein d'une structure peuvent être qualifiés en utilisant le nom du type avec une syntaxe très proche de celle disponible pour les énumérations. Par exemple, T.f2 se réfère au second champ de la déclaration précédente. Les définitions de structure peuvent être incorporées.

4.6.1 Variantes

Les structures définies peuvent avoir des variantes sur la base de la connaissance de ce qui est disponible dans l'environnement. Le sélecteur doit être un type énuméré qui définit les variantes possibles proposées par la structure. Il doit y avoir un cas pour chaque élément de l'énumération déclarée dans la sélection. Le corps de la structure de la variante peut être muni d'une étiquette de référence. Le mécanisme par lequel est choisie la variante au moment de l'exécution n'est pas prescrit par le langage de présentation.

```
struct {
   T1 f1;
   T2 f2;
   ....
   Tn fn;
   select (E) {
      cas e1 : Te1;
      cas e2 : Te2;
      ....
      cas en : Ten;
   } [[fv]];
} [[Tv]];
```

Par exemple :

```
enum { pomme, orange } VariantTag;
struct {
   uint16 number;
   opaque string<0..10>;            /* longueur variable */
```

```
    } V1;
    struct {
        uint32 number;
        opaque string[10];              /* longueur fixe */
    } V2;
    struct {
        select (VariantTag) {           /* la valeur du sélecteur est implicite */
            cas pomme: V1;              /* VariantBody, tag = pomme */
            cas orange: V2;             /* VariantBody, tag = orange */
        } variant_body;                 /* étiquette facultative sur la variante */
    } VariantRecord;
```

Les structures de variante peuvent être qualifiées (rétrécies) en spécifiant une valeur pour le sélecteur avant le type. Par exemple, un "orange VariantRecord" est un type rétréci de "VariantRecord" contenant un variant_body de type V2.

4.7 Attributs cryptographiques

Les quatre opérations cryptographiques de signature numérique, de chiffrement de flux, de chiffrement de bloc et de chiffrement de clé publique sont respectivement désignées comme "digitally-signed", "stream-ciphered", "block-ciphered", et "public-key-encrypted". Le traitement cryptographique d'un champ est spécifié en ajoutant en tête une désignation de mot clé approprié avant la spécification du type du champ. Les clés de chiffrement sont impliquées par l'état en cours de la session (voir au paragraphe 6.1).

Dans la signature numérique, des fonctions de hachage unidirectionnelles sont utilisées comme entrée d'un algorithme de signature. Un élément signé numériquement est codé comme un vecteur opaque <0..2^16-1>, où la longueur est spécifiée par l'algorithme de signature et la clé.

Dans la signature RSA, une structure de 32 octets de deux hachages (un SHA et un MD5) est signée (chiffrée avec la clé privée). Elle est codée avec le type de bloc 0 ou 1 de PKCS n° 1, comme décrit dans [PKCS1].

Dans DSS, les 20 octets du hachage SHA sont traités directement par l'algorithme de signature numérique sans hachage supplémentaire. Cela produit deux valeurs, r et s. La signature DSS est un vecteur opaque, comme ci-dessus, dont le contenu est le codage DER de :

```
Dss-Sig-Value  ::=  SEQUENCE  {
        r           ENTIER,
        s           ENTIER
    }
```

Dans le chiffrement de flux, le texte en clair est traité par l'opération OU exclusif avec une quantité identique de données générées à partir d'un générateur de nombres pseudo aléatoires cryptographiquement sûr.

Dans le chiffrement de bloc, chaque bloc de texte en clair chiffre un bloc de texte chiffré. Tout bloc de chiffrement chiffré est fait en mode chaînage de bloc de chiffrement (CBC, *Cipher Block Chaining*) et tous les éléments qui sont chiffrés par bloc seront un exact multiple de la longueur du bloc de chiffrement.

Dans le chiffrement à clé publique, on utilise un algorithme à clé publique pour chiffrer les données de façon telle qu'elles ne puissent être déchiffrées qu'avec la clé privée correspondante. Un élément chiffré avec une clé publique est codé comme un vecteur opaque <0..2^16-1>, où la longueur est spécifiée par l'algorithme de signature et la clé.

Une valeur codée en RSA est codée avec les type de bloc 2 de PKCS n° 1, comme décrit dans [PKCS1].

Dans l'exemple suivant :

```
stream-ciphered struct {
        uint8 field1;
        uint8 field2;
        digitally-signed opaque hash[20];
} UserType;
```

Le contenu du hachage est utilisé comme entrée de l'algorithme de signature, puis la structure entière est chiffrée avec un chiffrement de flux. La longueur de cette structure, en octets, serait égale à deux octets pour field1 et field2, plus deux

octets pour la longueur de la signature, plus la longueur du résultat de l'algorithme de signature. Cela découle du fait que l'algorithme et la clé utilisés pour la signature sont connus avant le codage ou décodage de cette structure.

4.8 Constantes

Des constantes par type peuvent être définies pour les besoins d'une spécification en déclarant un symbole du type désiré et en lui allouant des valeurs. Les types sous spécifiés (opaque, vecteurs de longueur variable, et structures qui contiennent des parties opaques) ne peuvent pas recevoir de valeurs. Aucun champ d'une structure ou vecteur multi éléments ne peut être élidé.

Par exemple,

```
struct {
        uint8 f1;
        uint8 f2;
} Example1;

Example1 ex1 = {1, 4};              /* alloue f1 = 1, f2 = 4 */
```

5. HMAC et la fonction pseudo aléatoire

Un certain nombre d'opérations de la couche TLS Enregistrement et prise de contact nécessitent un code d'authentification de message (MAC) chiffré ; c'est un résumé sûr de certaines données protégées par un secret. La falsification du MAC est infaisable sans la connaissance du secret du MAC. La construction qu'on utilise pour cette opération est appelée HMAC, décrite dans [HMAC].

HMAC peut être utilisé avec plusieurs algorithmes de hachage différents. TLS l'utilise dans la prise de contact avec deux algorithmes différents : MD5 et SHA-1, les notant par HMAC_MD5(secret, données) et HMAC_SHA(secret, données). Des algorithmes de hachage supplémentaires peuvent être définis par des suites de chiffrement et utilisés pour protéger les données enregistrées, mais MD5 et SHA-1 sont incorporés en code fixe dans la description de la prise de contact pour cette version du protocole.

De plus, une construction est nécessaire pour faire l'expansion des secrets dans les blocs de données pour les besoins de la génération ou validation des clés. Cette fonction pseudo aléatoire (PRF, *pseudo-random function*) prend en entrée un secret, un germe, et une étiquette d'identification et produit un résultat d'une longueur arbitraire.

Afin de rendre le PRF aussi sûr que possible, il utilise deux algorithmes de hachage d'une façon qui devrait garantir sa sécurité si l'un ou l'autre algorithme reste sûr.

D'abord, on définit une fonction d'expansion des données, P_hash(secret, data) qui utilise une seule fonction de hachage pour l'expansion d'un secret et d'un germe dans une quantité arbitraire de résultat :

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

où + indique l'enchaînement.

A() est défini comme :

```
A(0) = germe
A(i) = HMAC_hash(secret, A(i-1))
```

P_hash peut être itéré autant de fois que nécessaire pour produire la quantité de données requise. Par exemple, si P_SHA-1 devait être utilisé pour créer 64 octets de données, il devrait être itéré 4 fois (à travers A(4)), créant 80 octets de données en sortie; les 16 derniers octets de l'itération finale seraient éliminés, laissant 64 octets de données en sortie.

La PRF de TLS est créée en partageant le secret en deux moitiés et en utilisant une moitié pour générer les données avec P_MD5 et l'autre moitié pour générer les données avec P_SHA-1, puis en pratiquant l'opération OU exclusif sur les résultats de ces deux fonctions d'expansion ensemble.

S1 et S2 sont les deux moitiés du secret et chacune a la même longueur. S1 est tiré de la première moitié du secret, S2 de l'autre moitié. Puis la longueur est créée en arrondissant à la longueur du secret total divisé par deux ; ainsi, si le secret

d'origine est un nombre impair d'octets, le dernier octet de S1 sera le même que le premier octet de S2.

L_S = longueur du secret en octets ;
L_S1 = L_S2 = ceil(L_S / 2) ;

Le secret est partagé en deux moitiés (avec la possibilité d'un octet partagé) comme décrit ci-dessus, S1 prenant les L_S1 premiers octets et S2 les L_S2 derniers octets.

La PRF est ensuite définie comme le résultat du mélange des deux flux pseudo aléatoires en leur faisant subir ensemble l'opération OU exclusif.

PRF(secret, étiquette, germe) = P_MD5(S1, étiquette + germe) XOR P_SHA-1(S2, étiquette + germe) ;

L'étiquette est une chaîne ASCII. Elle devrait être incluse dans la forme exacte où elle est donnée sans octet de longueur ni caractère nul en queue. Par exemple, l'étiquette "slithy toves" serait traitée en hachant les octets suivants :

73 6C 69 74 68 79 20 74 6F 76 65 73

Noter que parce que MD5 produit des résultats de 16 octets et SHA-1 des résultats de 20 octets, les frontières de leurs itérations internes ne seront pas alignées ; pour générer un résultat de 80 octets cela implique que P_MD5 soit itéré à travers A(5), alors que P_SHA-1 sera seulement itéré à travers A(4).

6. Protocole d'enregistrement TLS

Le protocole d'enregistrement TLS est un protocole en couches. À chaque couche, les messages peuvent inclure des champs de longueur, de description, et de contenu. Le protocole d'enregistrement prend les messages à transmettre, fragmente les données en blocs gérables, compresse facultativement les données, applique un MAC, chiffre, et transmet le résultat. Les données reçues sont déchiffrées, vérifiées, décompressées, et réassemblées, puis livrées aux clients de niveau supérieur.

Quatre clients de protocole d'enregistrement sont décrits dans le présent document : le protocole de prise de contact, le protocole d'alerte, le protocole de changement de spécification de chiffrement, et le protocole des données d'application. Afin de permettre l'extension du protocole TLS, des types d'enregistrement supplémentaires peuvent être pris en charge par le protocole d'enregistrement. Tout nouveau type d'enregistrement devrait allouer des valeurs de type immédiatement au delà des valeurs de ContentType pour les quatre types d'enregistrements décrits ici (voir l'Appendice A.2). Si une mise en œuvre de TLS reçoit un type d'enregistrement qu'elle ne comprend pas, elle devrait simplement l'ignorer. Tout protocole conçu pour être utilisé sur TLS doit être conçu avec soin pour traiter toutes les attaques possibles contre lui. Noter que parce que le type et la longueur d'un enregistrement ne sont pas protégés par le chiffrement, il faut veiller à minimiser la valeur de l'analyse de trafic de ces valeurs.

6.1 États de connexion

Un état de connexion TLS est l'environnement de fonctionnement du protocole d'enregistrement TLS. Il spécifie un algorithme de compression, un algorithme de chiffrement, et un algorithme de MAC. De plus, les paramètres pour ces algorithmes sont connus : le secret de MAC et les clés de chiffrement en brut et les vecteurs d'initialisation (IV) pour la connexion à la fois dans la direction lecture et écriture. Logiquement, il y a toujours quatre états de connexion en cours : les états courants de lecture et d'écriture, et les états d'écriture et de lecture en attente. Tous les enregistrements sont traités sous les états courants de lecture et d'écriture. Les paramètres de sécurité pour les états en attente peuvent être réglés par le protocole de prise de contact TLS, et le protocole de prise de contact peut choisir de rendre l'un ou l'autre des états en attente les états courants, auquel cas l'état courant approprié est supprimé et remplacé par l'état en attente ; l'état en attente étant alors réinitialisé par un état vide. Il est illégal de faire d'un état qui n'a pas été initialisé avec des paramètres de sécurité un état courant. L'état courant initial spécifie toujours qu'aucun chiffrement, compression, ou MAC, ne sera utilisé.

Les paramètres de sécurité pour un état TLS de connexion de lecture et d'écriture sont réglés en fournissant les valeurs suivantes :

bout de la connexion *(connection end)*
Dit si cette entité est considérée comme le "client" ou le "serveur" dans cette connexion.

algorithme de chiffrement en vrac *(bulk encryption algorithm)*
Algorithme à utiliser pour le chiffrement en vrac. Cette spécification comporte la taille de clé de cet algorithme, combien

de cette clé est secret, si c'est un chiffrement de bloc ou de flux, la taille de bloc du chiffrement (si c'est approprié) et si c'est considéré comme un chiffrement "export".

algorithme de MAC
Algorithme à utiliser pour l'authentification de message. Cette spécification comporte la taille du hachage qui est retourné par l'algorithme de MAC.

algorithme de compression
Algorithme à utiliser pour la compression des données. Cette spécification doit comporter toutes les informations que l'algorithme requiert pour faire la compression.

secret maître
C'est un secret de 48 octets partagé entre les deux homologues de la connexion.

aléa de client *(client random)*
Valeur de 32 octets fournie par le client.

aléa de serveur *(server random)*
Valeur de 32 octets fournie par le serveur.

Ces paramètres sont définis en langage de présentation par :

```
enum { serveur, client } ConnectionEnd;
enum { null, rc4, rc2, des, 3des, des40 } BulkCipherAlgorithm;
enum { flux, bloc } CipherType;
enum { vrai, faux } IsExportable;
enum { null, md5, sha } MACAlgorithm;
enum { null(0), (255) } CompressionMethod;

/* Les algorithmes spécifiés dans CompressionMethod, BulkCipherAlgorithm, et MACAlgorithm peuvent être ajoutés. */

struct {
        ConnectionEnd           entity;
        BulkCipherAlgorithm     bulk_cipher_algorithm;
        CipherType              cipher_type;
        uint8                   key_size;
        uint8                   key_material_length;
        IsExportable            is_exportable;
        MACAlgorithm            mac_algorithm;
        uint8                   hash_size;
        CompressionMethod       compression_algorithm;
        opaque                  master_secret[48];
        opaque                  client_random[32];
        opaque                  server_random[32];
    } SecurityParameters;
```

La couche d'enregistrement va utiliser les paramètres de sécurité pour générer les six éléments suivants :
- le client écrit le secret de MAC
- le serveur écrit le secret de MAC
- le client écrit la clé
- le serveur écrit la clé
- le client écrit le vecteur d'initialisation (IV) (seulement pour le chiffrement de bloc)
- le serveur écrit le IV (seulement pour le chiffrement de bloc)

Les paramètres écrits par le client sont utilisés par le serveur lors de la réception et du traitement des enregistrements et vice-versa. L'algorithme utilisé pour générer ces éléments à partir des paramètres de sécurité est décrit au paragraphe 6.3.

Une fois que les paramètres de sécurité ont été réglés et que les clés ont été générées, les états de connexion peuvent être instanciés en faisant d'eux les états en cours. Ces états en cours doivent être mis à jour pour chaque enregistrement traité. Chaque état de connexion comporte les éléments suivants :

état de compression
C'est l'état en cours de l'algorithme de compression.

état de chiffrement
C'est l'état en cours de l'algorithme de chiffrement. Cela va consister en la clé programmée pour cette connexion. De plus, pour les chiffrements de bloc qui fonctionnent en mode CBC (le seul mode spécifié pour TLS) cela va contenir initialement le vecteur d'initialisation (IV) pour cet état de connexion et sera mis à jour pour contenir le texte chiffré du dernier bloc chiffré ou déchiffré lorsque les enregistrements sont traités. Pour les chiffrements de flux, cela va contenir toutes les informations d'état nécessaires pour permettre au flux de continuer à chiffrer ou déchiffrer les données.

secret de MAC
C'est le secret de MAC pour cette connexion tel que généré ci-dessus.

numéro de séquence
Chaque état de connexion contient un numéro de séquence qui est entretenu séparément pour les états lecture et écriture. Le numéro de séquence doit être mis à zéro chaque fois qu'un état de connexion devient l'état actif. Les numéros de séquence sont de type uint64 et ne peuvent pas dépasser 2^64-1. Un numéro de séquence est incrémenté après chaque enregistrement : précisément, le premier enregistrement qui est transmis sur un état de connexion particulier devrait utiliser le numéro de séquence 0.

6.2 Couche d'enregistrement

La couche enregistrement TLS reçoit des données non interprétées des couches supérieures dans des blocs non vides de taille arbitraire.

6.2.1 Fragmentation

La couche enregistrement fragmente les blocs d'informations en enregistrements TLSPlaintext qui portent les données dans des tronçons de 2^14 octets ou moins. Les limites du message client ne sont pas préservées dans la couche d'enregistrement (c'est-à-dire que plusieurs messages client du même ContentType peuvent être unis dans un seul enregistrement TLSPlaintext, ou un seul message peut être fragmenté sur plusieurs enregistrements).

```
struct {
   uint8 major, minor;
} ProtocolVersion;

enum {
   change_cipher_spec(20), alert(21), handshake(22),
   application_data(23), (255)
} ContentType;

struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
   opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type
C'est le protocole de niveau le plus élevé utilisé pour traiter le fragment inclus.

version
C'est la version du protocole qui est employée. Le présent, document décrit TLS version 1.0, qui utilise la version { 3, 1 }. La valeur de version 3.1 est historique: TLS version 1.0 est une modification mineure du protocole SSL 3.0, qui porte la valeur de version 3.0. (Voir l'appendice A.1).

length
C'est la longueur (en octets) du fragment TLSPlaintext.fragment suivant. La longueur ne devrait pas excéder 2^14.

fragment
Ce sont les données d'application. Ces données sont transparentes et traitées comme un bloc indépendant dont disposera le protocole de niveau supérieur spécifié par le champ de type.

Note : Les données de différents types de contenu de couche Enregistrement TLS peuvent être entremêlées. Les données

d'application ont généralement une priorité moindre que celles des autres types de contenu en ce qui concerne la transmission.

6.2.2 Compression et décompression d'enregistrement

Tous les enregistrements sont compressés en utilisant l'algorithme de compression défini dans l'état actuel de la session. Il y a toujours un algorithme de compression actif ; cependant, il est initialement défini comme CompressionMethod.null. L'algorithme de compression traduit une structure TLSPlaintext en une structure TLSCompressed. Les fonctions de compression sont initialisées avec des informations d'état par défaut chaque fois qu'un état de connexion devient actif.

La compression doit être sans perte et ne doit pas augmenter la longueur du contenu de plus de 1024 octets. Si la fonction de décompression rencontre un TLSCompressed.fragment qui se décompresserait à une longueur dépassant 2^14 octets, il devrait faire rapport d'une erreur fatale d'échec de décompression.

```
struct {
   ContentType type;              /* le même que TLSPlaintext.type */
   ProtocolVersion version;       /* le même que TLSPlaintext.version */
   uint16 length;
   opaque fragment[TLSCompressed.length];
} TLSCompressed;
```

length
C'est la longueur (en octets) du fragment TLSCompressed.fragment suivant. La longueur ne devrait pas dépasser 2^14 + 1024.

fragment
C'est la forme compressée du fragment TLSPlaintext.fragment.

Note : Une opération CompressionMethod.null est une opération à l'identique ; aucun champ n'est altéré.

Note de mise en œuvre :
Les fonctions de décompression sont chargées de s'assurer que les messages ne peuvent pas causer de débordement de mémoire tampon interne.

6.2.3 Protection de la charge utile de l'enregistrement

Les fonctions de chiffrement et de MAC traduisent une structure TLSCompressed en une structure TLSCiphertext. Les fonctions de déchiffrement inversent le processus. Le MAC de l'enregistrement comporte aussi un numéro de séquence de façon à détecter les messages manquants, en trop, ou répétés.

```
struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
   select (CipherSpec.cipher_type) {
      cas flux : GenericStreamCipher;
      cas bloc : GenericBlockCipher;
   } fragment;
} TLSCiphertext;
```

type
Le champ type est identique au TLSCompressed.type.

version
Le champ version est identique au TLSCompressed.version.

length
C'est la longueur (en octets) du fragment TLSCiphertext.fragment suivant. La longueur ne doit pas excéder 2^14 + 2048.

fragment
C'est la forme chiffrée du fragment TLSCompressed.fragment, avec le MAC.

6.2.3.1 Chiffrement de flux nul ou standard

Les chiffrements de flux (y compris BulkCipherAlgorithm.null – voir l'Appendice A.6) convertissent les structures TLSCompressed.fragment en structures de flux TLSCiphertext.fragment et réciproquement.

```
stream-ciphered struct {
   opaque content[TLSCompressed.length];
   opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

Le MAC est généré comme :

```
HMAC_hash(MAC_write_secret, seq_num + TLSCompressed.type + TLSCompressed.version +
 TLSCompressed.length + TLSCompressed.fragment));
```

où "+" note l'enchaînement.

seq_num
C'est le numéro de séquence pour cet enregistrement.

hash
C'est l'algorithme de hachage spécifié par SecurityParameters.mac_algorithm.

Noter que le MAC est calculé avant le chiffrement. Le chiffrement de flux chiffre le bloc entier y compris le MAC. Pour les chiffrements de flux qui n'utilisent pas le vecteur de synchronisation (comme RC4) l'état de chiffrement de flux à partir de la fin d'un enregistrement est simplement utilisé sur le paquet suivant. Si la CipherSuite est TLS_NULL_WITH_NULL_NULL, le chiffrement consiste en l'opération d'identité (c'est-à-dire que les données ne sont pas chiffrées et que la taille du MAC est zéro, ce qui implique qu'aucun MAC n'est utilisé). TLSCiphertext.length est TLSCompressed.length plus CipherSpec.hash_size.

6.2.3.2 Chiffrement de bloc CBC

Pour les chiffrements de bloc (comme RC2 ou DES) les fonctions de chiffrement et de MAC convertissent les structures TLSCompressed.fragment en structures de bloc TLSCiphertext.fragment et réciproquement.

```
block-ciphered struct {
   opaque content[TLSCompressed.length];
   opaque MAC[CipherSpec.hash_size];
   uint8 padding[GenericBlockCipher.padding_length];
   uint8 padding_length;
} GenericBlockCipher;
```

Le MAC est généré comme décrit au paragraphe 6.2.3.1.

padding
C'est le bourrage qui est ajouté pour forcer la longueur du texte en clair à être un multiple entier de la longueur de bloc du chiffrement de bloc. Le bourrage peut être de toute longueur jusqu'à 255 octets, pour autant qu'il en résulte que TLSCiphertext.length soit un multiple entier de la longueur de bloc. Les longueurs supérieures à ce qui est nécessaire peuvent être souhaitables pour déjouer des attaques contre un protocole qui se fonderaient sur une analyse des longueurs des messages échangés. Chaque uint8 dans le vecteur de données de bourrage doit être rempli avec la valeur de longueur du bourrage.

padding_length
La longueur du bourrage devrait être telle que la taille totale de la structure GenericBlockCipher soit un multiple de la longueur de bloc du chiffrement. Les valeurs légales vont de zéro à 255, inclus. Cette longueur spécifie la longueur du champ de bourrage non compris le champ padding_length lui-même.

La longueur des données chiffrées (TLSCiphertext.length) est un, plus la somme de TLSCompressed.length, CipherSpec.hash_size, et padding_length.

Exemple : Si la longueur de bloc est de 8 octets, la longueur du contenu (TLSCompressed.length) est de 61 octets, et la longueur du MAC est de 20 octets, la longueur avant bourrage est 82 octets. Donc, la longueur du bourrage modulo 8 doit être égale à 6 afin de rendre la longueur totale un multiple pair de 8 octets (la longueur de bloc). La longueur du bourrage

peut être 6, 14, 22, et ainsi de suite, jusqu'à 254. Si la longueur du bourrage était du minimum nécessaire, 6, le bourrage serait de 6 octets, contenant chacun la valeur 6. Donc, les 8 derniers octets de GenericBlockCipher avant le chiffrement de bloc seraient xx 06 06 06 06 06 06 06, où xx est le dernier octet du MAC.

Note : Avec les chiffrements de bloc en mode CBC (Cipher Block Chaining, *chaînage de bloc de chiffrement*) le vecteur d'initialisation (IV) pour le premier enregistrement est généré avec les autres clés et secrets lorsque les paramètres de sécurité sont établis. Le IV pour les enregistrements suivants est le dernier bloc en texte chiffré à partir de l'enregistrement précédent.

6.3 Calcul des clés

Le protocole d'enregistrement exige un algorithme pour générer les clés, les vecteurs d'initialisation, et les secrets de MAC à partir des paramètres de sécurité fournis par le protocole de prise de contact.

Le secret maître est haché en une séquence d'octets sûrs, qui sont alloués aux secrets de MAC, aux clés et aux IV non export exigés par l'état actuel de la connexion (voir à l'Appendice A.6). Les spécifications de chiffrement *(CipherSpec)* exigent qu'un client écrive le secret de MAC, qu'un serveur écrive un secret de MAC, qu'un client écrive une clé, qu'un serveur écrive une clé, qu'un client écrive le vecteur d'initialisation, et qu'un serveur écrive le IV, qui sont générés à partir du secret maître dans cet ordre. Les valeurs inutilisées sont laissées vides.

Lors de la génération de clés et de secrets de MAC, le secret maître est utilisé comme source d'entropie, et les valeurs aléatoires fournissent un matériel de sel et de vecteurs d'initialisation non chiffré pour des chiffrements exportables.

Pour générer le matériel de clés, calculer

```
key_block = PRF(SecurityParameters.master_secret, "key expansion", SecurityParameters.server_random +
                SecurityParameters.client_random);
```

jusqu'à ce que suffisamment de résultat ait été généré. Puis le key_block est partagé comme suit :

```
client_write_MAC_secret[SecurityParameters.hash_size]
server_write_MAC_secret[SecurityParameters.hash_size]
client_write_key[SecurityParameters.key_material_length]
server_write_key[SecurityParameters.key_material_length]
client_write_IV[SecurityParameters.IV_size]
server_write_IV[SecurityParameters.IV_size]
```

Le client_write_IV et le server_write_IV ne sont générés que pour les chiffrements de bloc non exportés. Pour les chiffrements de bloc exportables, les vecteurs d'initialisation sont générés plus tard, comme on le décrit ci-dessous. Tout le matériel excédentaire de key_block est éliminé.

Note de mise en œuvre :
La spécification de chiffrement qui est définie dans le présent document et exige la plupart du matériel est 3DES_EDE_CBC_SHA : elle exige 2 x 24 octets de clés, 2 x 20 octets de secrets de MAC, et 2 x 8 octets d'IV, pour un total de 104 octets de matériel de clés.

Les algorithmes de chiffrement exportables (pour lesquels CipherSpec.is_exportable est vrai) exigent le traitement supplémentaire suivant pour déduire leurs clés d'écriture finales :

```
final_client_write_key = PRF(SecurityParameters.client_write_key, "client write key",
                             SecurityParameters.client_random + SecurityParameters.server_random);
final_server_write_key = PRF(SecurityParameters.server_write_key, "server write key",
                             SecurityParameters.client_random + SecurityParameters.server_random);
```

Les algorithmes de chiffrement exportables ne déduisent leurs IV que des valeurs aléatoires provenant des messages hello :

```
iv_block = PRF("", "IV block", SecurityParameters.client_random + SecurityParameters.server_random);
```

Le iv_block est partagé en deux vecteurs d'initialisation comme le key_block l'a été ci-dessus :

```
client_write_IV[SecurityParameters.IV_size]
server_write_IV[SecurityParameters.IV_size]
```

Noter que dans ce cas, le PRF est utilisé sans secret : cela signifie simplement que le secret a une longueur de zéro octets et ne contribue en rien au hachage dans la PRF.

6.3.1 Exemple de génération de clé exportable

TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5 requiert cinq octets d'aléa pour chacune des deux clés de chiffrement et 16 octets pour chacune des clés du MAC, pour un total de 42 octets de matériel de clés. Le résultat de la PRF est mémorisé dans le key_block. Ce key_block est partagé, et les clés d'écriture sont salées parce que c'est un algorithme de chiffrement exportable.

```
key_block = PRF(master_secret, "key expansion", server_random + client_random)[0..41]
client_write_MAC_secret = key_block[0..15]
server_write_MAC_secret = key_block[16..31]
client_write_key = key_block[32..36]
server_write_key = key_block[37..41]
final_client_write_key  = PRF(client_write_key, "client write key", client_random + server_random)[0..15]
final_server_write_key  = PRF(server_write_key, "server write key", client_random + server_random)[0..15]

iv_block = PRF("", "IV block", client_random + server_random)[0..15]
client_write_IV = iv_block[0..7]
server_write_IV = iv_block[8..15]
```

7. Protocole de prise de contact TLS

Le protocole de prise de contact TLS consiste en une suite de trois sous protocoles qui sont utilisés pour permettre aux homologues de se mettre d'accord sur les paramètres de sécurité pour la couche d'enregistrement, s'authentifier mutuellement, créer les paramètres de sécurité négociés, et faire rapport l'un à l'autre des conditions d'erreur rencontrées.

Le protocole de prise de contact est chargé de négocier une session, ce qui comporte les éléments suivants :

identifiant de session
C'est une séquence d'octets arbitraire choisie par le serveur pour identifier un état de session active ou récupérable.

certificat d'homologue
C'est le certificat X509v3 [X509] de l'homologue. Cet élément de l'état peut être nul.

méthode de compression
C'est l'algorithme utilisé pour compresser les données avant le chiffrement.

spécification de chiffrement (cipher-spec)
Elle spécifie l'algorithme de chiffrement des données brutes (comme un nul, un DES, etc.) et un algorithme de MAC (comme MD5 ou SHA). Elle définit aussi les attributs cryptographiques tels que la taille de hachage (hash_size). (Voir à l'Appendice A.6 la définition formelle.)

secret maître
C'est le secret de 48 octets partagé entre le client et le serveur.

est récupérable
C'est un fanion qui indique si la session peut être utilisée pour initier de nouvelles connexions.

Ces éléments sont alors utilisés pour créer des paramètres de sécurité à l'usage de la couche Enregistrement lors de la protection des données d'application. De nombreuses connexions peuvent être matérialisées en utilisant la même session grâce à la caractéristique de récupération du protocole de prise de contact TLS.

7.1 Protocole de changement de spécification de chiffrement

Le but du protocole de changement de spécification de chiffrement est de signaler les transitions dans les stratégies de chiffrement. Le protocole consiste en un seul message, qui est chiffré et compressé dans l'état courant (et dans l'état en attente) de la connexion. Le message consiste en un seul octet de valeur 1.

```
struct {
   enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

Le message de changement de spécification de chiffrement est envoyé par le client et le serveur pour notifier à la partie receveuse que les enregistrements suivants seront protégés sous les CipherSpec et clés nouvellement négociées. La réception de ce message amène le receveur à donner instruction à la couche Enregistrement de copier immédiatement l'état de lecture en attente dans l'état de lecture courant. Immédiatement après l'envoi de ce message, l'envoyeur devrait donner pour instruction à la couche d'enregistrement de faire de l'état d'écriture en attente l'état d'écriture actif. (Voir au paragraphe 6.1.) Le message de changement de spécification de chiffrement est envoyé durant la prise de contact après l'accord sur les paramètres de sécurité, mais avant que ne soit envoyé le message de vérification Terminé (voir au paragraphe 7.4.9).

7.2 Protocole d'alerte

Un des types de contenu pris en charge par la couche Enregistrement de TLS est le type "alerte". Les messages d'alerte véhiculent le niveau de sévérité du message et une description de l'alerte. Les messages d'alerte avec un niveau de fatal résultent en la terminaison immédiate de la connexion. Dans ce cas, les autres connexions correspondant à la session peuvent continuer, mais l'identifiant de session doit être invalidé, pour empêcher la session en échec d'être utilisée pour établir de nouvelles connexions. Comme les autres messages, les messages d'alerte sont chiffrés et compressés, comme spécifié par l'état courant de connexion.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
   close_notify(0),
   unexpected_message(10),
   bad_record_mac(20),
   decryption_failed(21),
   record_overflow(22),
   decompression_failure(30),
   handshake_failure(40),
   bad_certificate(42),
   unsupported_certificate(43),
   certificate_revoked(44),
   certificate_expired(45),
   certificate_unknown(46),
   illegal_parameter(47),
   unknown_ca(48),
   access_denied(49),
   decode_error(50),
   decrypt_error(51),
   export_restriction(60),
   protocol_version(70),
   insufficient_security(71),
   internal_error(80),
   user_canceled(90),
   no_renegotiation(100),
   (255)
} AlertDescription;

struct {
   AlertLevel level;
   AlertDescription description;
} Alert;
```

7.2.1 Alertes de clôture

Le client et le serveur doivent partager la connaissance de ce que la connexion se termine afin d'éviter une attaque par troncature. L'une ou l'autre partie peut initier l'échange de messages de clôture.

close_notify
Ce message notifie au receveur que l'envoyeur ne va plus envoyer d'autres messages sur cette connexion. La session devient non récupérable si une connexion est terminée sans messages close_notify appropriés avec un niveau égal à "avertissement".

L'une ou l'autre partie peut initier une clôture par l'envoi d'une alerte close_notify. Toutes les données reçues après une alerte de clôture sont ignorées.

Il est exigé de chaque partie qu'elle envoie une alerte close_notify avant de clore le côté écriture de la connexion. Il est exigé de l'autre partie qu'elle réponde elle-même par une alerte close_notify et close immédiatement la connexion, en éliminant toutes les écritures en cours. Il n'est pas exigé de l'initiateur de la clôture qu'il attende l'alerte close_notify en réponse avant de clore le côté lecture de la connexion.

Si le protocole d'application qui utilise TLS prévoit que les données peuvent être portées sur le transport sous-jacent après la clôture de la connexion TLS, la mise en œuvre de TLS doit recevoir l'alerte close_notify en réponse avant d'indiquer à la couche d'application que la connexion TLS s'est terminée. Si le protocole d'application ne transfère aucune donnée supplémentaire, mais clôt seulement la connexion de transport sous-jacente, la mise en œuvre peut alors choisir de clore le transport sans attendre le close_notify en réponse. Aucune partie de la présente norme ne devrait être considérée comme imposant la manière dont un profil d'usage de TLS gère son transport de données, y compris quand les connexions sont ouvertes ou closes.

Note : On suppose que la clôture fiable d'une connexion livre les données en attente avant de détruire le transport.

7.2.2 Alertes d'erreurs

Le traitement des erreurs dans le protocole de prise de contact TLS est très simple. Quand une erreur est détectée, celui qui la détecte envoie un message à l'autre partie. À la transmission ou réception d'un message d'alerte fatale, les deux parties ferment immédiatement la connexion. Il est exigé des serveurs et des clients qu'ils oublient tous les identifiants de session, clés et secrets associés à une connexion défaillante. Les alertes d'erreur suivantes sont définies :

unexpected_message *(message inattendu)*
Un message inapproprié a été reçu. Cette alerte est toujours fatale et ne devrait jamais être observée dans des communications entre des mises en œuvre correctes.

bad_record_mac *(mauvais code MAC d'enregistrement)*
Cette alerte est retournée si un enregistrement est reçu avec un MAC incorrect. Ce message est toujours fatal.

decryption_failed *(échec du déchiffrement)*
Un TLSCiphertext a été déchiffré d'une façon invalide : soit qu'il n'était pas un multiple pair de la longueur de bloc, soit que ses valeurs de bourrage, lors de leur vérification, n'étaient pas correctes. Ce message est toujours fatal.

record_overflow *(débordement de l'enregistrement)*
Un enregistrement TLSCiphertext a été reçu avec une longueur de plus de 2^14+2048 octets, ou un enregistrement a été déchiffré en un enregistrement TLSCompressed de plus de 2^14+1024 octets. Ce message est toujours fatal.

decompression_failure *(échec de la décompression)*
La fonction de décompression a reçu une entrée impropre (par exemple, des données dont l'expansion serait d'une longueur excessive). Ce message est toujours fatal.

handshake_failure *(échec de la prise de contact)*
La réception d'un message d'alerte handshake_failure indique que l'envoyeur n'a pas été capable de négocier un ensemble acceptable de paramètres de sécurité en fonction des options disponibles. C'est une erreur fatale.

bad_certificate *(mauvais certificat)*
Un certificat était corrompu, contenait des signatures qui ne se vérifient pas correctement, etc.

unsupported_certificate *(certificat non pris en charge)*
Un certificat était d'un type non accepté.

certificate_revoked *(certificat révoqué)*
Un certificat était révoqué par son signataire.

certificate_expired *(certificat arrivé à expiration)*
Un certificat avait expiré ou n'est plus valide actuellement.

certificate_unknown *(certificat inconnu)*
Certains autres problèmes (non spécifiés) surviennent dans le traitement du certificat, le rendant inacceptable.

illegal_parameter *(paramètre illégal)*
Un champ dans la prise de contact était hors gamme ou incohérent avec les autres champs. Ceci est toujours fatal.

unknown_ca *(autorité de certification inconnue)*Un chaîne de certificat valide ou une chaîne partielle a été reçue, mais le certificat n'a pas été accepté parce que la CA *(autorité de certification)* du certificat n'a pas pu être localisée ou n'a pas pu être mise en correspondance avec une CA connue, de confiance. Ce message est toujours fatal.

access_denied *(accès refusé)*
Un certificat valide a été reçu, mais quand le contrôle d'accès a été appliqué, l'envoyeur a décidé de ne pas poursuivre la négociation. Ce message est toujours fatal.

decode_error *(erreur de décodage)*
Un message n'a pas pu être décodé parce qu'un champ était hors de la gamme spécifiée ou que la longueur du message était incorrecte. Ce message est toujours fatal.

decrypt_error *(erreur de déchiffrement)*
Une opération cryptographique de prise de contact a échoué, y compris d'être incapable de vérifier correctement une signature, de déchiffrer un échange de clé, ou de valider un message fini.

export_restriction *(interdiction d'exportation)*
On a détecté une négociation qui n'est pas conforme aux restrictions d'exportation ; par exemple, de tenter de transférer une clé RSA éphémère de 1024 bits pour la méthode de prise de contact RSA_EXPORT. Ce message est toujours fatal.

protocol_version *(version du protocole)*
La version du protocole que le client a tenté de négocier est reconnue, mais non prise en charge. (Par exemple, de vieilles versions du protocole pourraient être évitées pour des raisons de sécurité). Ce message est toujours fatal.

insufficient_security *(sécurité insuffisante)*
Retourné à la place d'un handshake_failure lorsque une négociation a échoué spécifiquement parce que le serveur exige des chiffrements plus sûrs que ceux acceptés par le client. Ce message est toujours fatal.

internal_error *(erreur interne)*
Une erreur interne sans relation avec l'homologue ou la correction du protocole rend impossible de continuer (comme une défaillance d'une allocation de mémoire). Ce message est toujours fatal.

user_canceled *(usager annulé)*
Cette prise de contact est annulée pour des raisons sans rapport avec une défaillance du protocole. Si l'usager annule une opération après l'achèvement de la prise de contact, il est plus approprié de juste clore la connexion en envoyant un close_notify. Cette alerte devrait être suivie par un close_notify. Ce message est généralement un avertissement.

no_renegotiation *(pas de renégociation)*
Envoyé par le client en réponse à une demande hello ou par le serveur en réponse à un hello du client après une prise de contact initiale. L'une et l'autre de ces situations devraient normalement conduire à une renégociation ; lorsque ce n'est pas approprié, le receveur devrait répondre par cette alerte ; le demandeur original peut alors décider si il va continuer avec cette connexion. Un cas où cela serait approprié serait celui où un serveur a créé un processus pour satisfaire une demande ; le processus peut recevoir des paramètres de sécurité (longueur de clé, authentification, etc.) au démarrage et il peut être ensuite difficile de communiquer des changements à ces paramètres. Ce message est toujours un avertissement.

Pour toutes les erreurs où un niveau d'alerte n'est pas explicitement spécifié, l'envoyeur peut déterminer à sa discrétion si c'est une erreur fatale ou non ; si une alerte est reçue avec un niveau d'avertissement, le receveur peut décider à sa discrétion de traiter cela comme une erreur fatale ou non. Cependant, tous les messages qui sont transmis avec un niveau de fatal doivent être traités comme des messages fatals.

7.3 Présentation du protocole de prise de contact

Les paramètres cryptographiques de l'état de session sont produits par le protocole de prise de contact TLS, qui fonctionne

par dessus la couche Enregistrement TLS. Lorsque un client et un serveur TLS commencent à communiquer, ils se mettent d'accord sur une version de protocole, choisissent des algorithmes cryptographiques, s'authentifient facultativement l'un l'autre, et utilisent des techniques de chiffrement à clés publiques pour générer des secrets partagés.

Le protocole de prise de contact TLS comporte les étapes suivantes :
- Échange des messages hello pour se mettre d'accord sur les algorithmes, échange des valeurs aléatoires, et vérification de reprise de session.
- Échange des paramètres cryptographiques nécessaires pour permettre au client et au serveur de s'accorder sur un secret pré maître.
- Échange des certificats et des informations cryptographiques pour permettre au client et au serveur de s'authentifier mutuellement.
- Générer un secret maître à partir du secret pré-maître et échanger les valeurs aléatoires.
- Fournir les paramètres de sécurité à la couche d'enregistrement.
- Permettre au client et au serveur de vérifier que leur homologue a calculé les mêmes paramètres de sécurité et que la prise de contact s'est déroulée sans être altérée par une attaque.

Noter que les couches supérieures ne devraient pas se fier exagérément à l'idée que TLS négocie toujours la connexion la plus forte possible entre deux homologues : il existe un certain nombre de façons pour qu'une attaque par interposition puisse tenter de faire que deux entités retombent à la méthode la moins sûre qu'elles prennent en charge. Le protocole a été conçu pour minimiser ce risque, mais il y a toujours des attaques disponibles : par exemple, une attaque peut bloquer l'accès sur lequel un service sûr fonctionne, ou tenter d'obtenir des homologues qu'ils négocient une connexion non authentifiée. La règle fondamentale est que les niveaux supérieurs doivent être pleinement informés de ce que sont leurs exigences de sécurité et ne jamais transmettre d'informations sur un canal moins sûr que ce qu'ils exigent. Le protocole TLS est sûr, en ce que toute suite de chiffrement offre son niveau de sécurité promis : si vous négociez 3DES avec un échange de clé RSA à 1024 bits avec un hôte dont vous avez vérifié le certificat, vous pouvez vous attendre à avoir cette sécurité là.

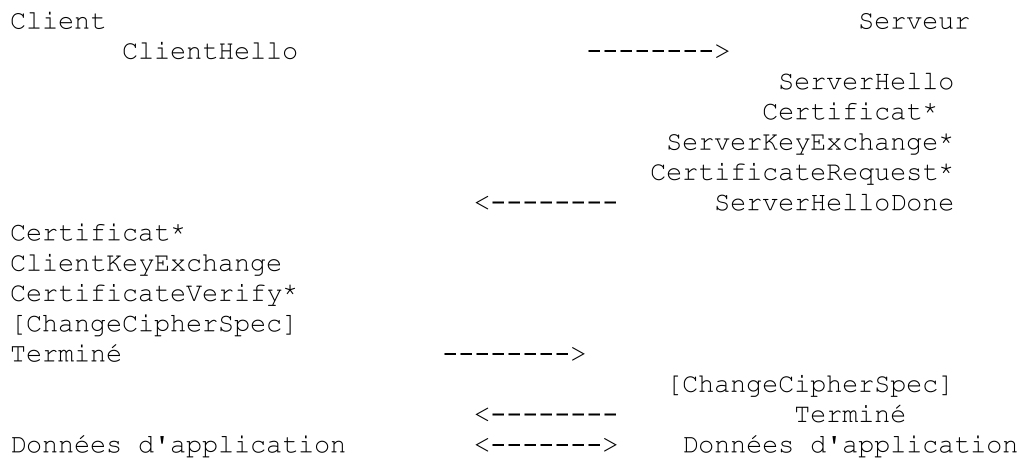
Cependant, vous ne devriez jamais envoyer des données sur une liaison chiffrée avec une sécurité de 40 bits à moins que vous n'estimiez que ces données ne valent pas plus que l'effort requis pour casser ce chiffrement.

Ces objectifs sont réalisés par le protocole de prise de contact, qui peut se résumer comme suit : le client envoie un message hello de client auquel le serveur doit répondre par un message hello de serveur, sinon une erreur fatale surviendra et la connexion échouera. Le hello du client et le hello du serveur sont utilisés pour établir des capacités de sécurité améliorées entre client et serveur. Le hello du client et le hello du serveur établissent les attributs suivants : Version du protocole, Identifiant de session, Suite de chiffrement, et Méthode de compression. De plus, deux valeurs aléatoires sont générées et échangées : ClientHello.random et ServerHello.random.

L'échange de clés réel utilise jusqu'à quatre messages : le certificat du serveur, l'échange de clé de serveur, le certificat de client, et l'échange de clé de client. De nouvelles méthodes d'échange de clés peuvent être créées en spécifiant un format pour ces messages et en définissant l'utilisation des messages de façon à permettre au client et au serveur de se mettre d'accord sur un secret partagé. Ce secret devrait être assez long ; les méthodes d'échange de clés actuellement définies échangent des secrets qui vont de 48 à 128 octets de long.

À la suite des messages hello, le serveur va envoyer son certificat, si il doit être authentifié. De plus, un message d'échange de clés de serveur peut être envoyé, si c'est exigé (par exemple, si leur serveur n'a pas de certificat, ou si son certificat n'est que pour la signature). Si le serveur est authentifié, il peut demander au client de produire un certificat, si c'est approprié à la suite de chiffrement choisie. Maintenant, le serveur va envoyer le message serveur hello effectué, qui indique que la phase du message hello de la prise de contact est terminée. Le serveur va alors attendre une réponse du client. Si le serveur a envoyé un message de demande de certificat, le client doit envoyer le message de certificat. Le message d'échange de clés de client est ensuite envoyé, et le contenu de ce message va dépendre de l'algorithme de clé publique choisi entre le hello de client et le hello de serveur. Si le client a envoyé un certificat avec une capacité de signature, un message de vérification de certificat à signature numérique sera envoyé pour vérifier explicitement le certificat.

C'est à ce moment qu'est envoyé par le client un message de changement de spécification de chiffrement, et le client copie la spécification de chiffrement en attente dans la spécification de chiffrement courante. Le client envoie alors immédiatement le message fini sous les nouveaux algorithmes, clés, et secrets. En réponse, le serveur va envoyer son propre message de changement de spécification de chiffrement, transférer la spécification de chiffrement de l'état en attente à l'état courant, et envoyer le message fini selon la nouvelle spécification de chiffrement. À ce moment, la prise de contact est terminée et client et serveur peuvent commencer à échanger des données de couche application. (Voir le diagramme de flux ci-dessous.)



* Indique des messages facultatifs ou dépendants de la situation qui ne sont pas toujours envoyés.

Figure 1 – Flux de messages pour une prise de contact complète

Note : Pour aider à éviter les blocages de traitement en parallèle, ChangeCipherSpec est un type de contenu de protocole TLS indépendant, et n'est en fait pas un message de prise de contact TLS.

Lorsque client et serveur décident de reprendre une session précédente ou de dupliquer une session existante (au lieu de négocier de nouveaux paramètres de sécurité) le flux de messages est le suivant :

Le client envoie un ClientHello en utilisant l'identifiant de session de la session à reprendre. Le serveur cherche alors une correspondance dans son antémémoire de session. Si il trouve une correspondance, et si le serveur veut rétablir la connexion sous l'état de connexion spécifié, il va envoyer un ServerHello avec la même valeur d'identifiant de session. C'est à ce moment que client et serveur doivent envoyer les messages de changement de spécification de chiffrement et passer directement aux messages finis. Une fois le rétablissement achevé, client et serveur peuvent commencer à échanger des données de couche application. (Voir le diagramme de flux ci-dessous.) Si on ne trouve pas de correspondance d'identifiant de session, le serveur génère un nouvel identifiant de session et le client et serveur TLS effectuent une prise de contact complète.

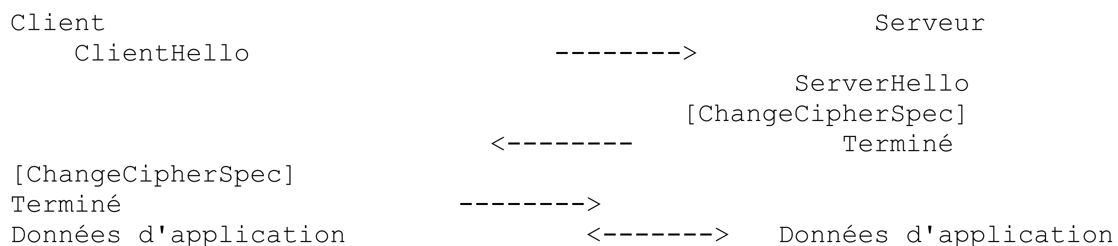


Figure 2 – Flux de messages pour une prise de contact abrégée

Le contenu et la signification de chaque message seront présentés en détails dans les paragraphes qui suivent.

7.4 Protocole de prise de contact

Le protocole de prise de contact TLS est un des clients de niveau supérieur définis pour le protocole d'enregistrement TLS. Ce protocole est utilisé pour négocier les attributs sûrs d'une session. Les messages de prise de contact sont fournis à la couche Enregistrement TLS, où ils sont encapsulés au sein d'une ou plusieurs structures TLSPlaintext, qui sont traitées et transmises comme spécifié par l'état de session actif courant.

```
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;

struct {
   HandshakeType msg_type;        /* type de prise de contact */
```

```
    uint24 length;                 /* octets dans le message */
    select (HandshakeType) {
        cas hello_request :           HelloRequest;
        cas client_hello :            ClientHello;
        cas server_hello :            ServerHello;
        cas certificate :             Certificate;
        cas server_key_exchange :     ServerKeyExchange;
        cas certificate_request :     CertificateRequest;
        cas server_hello_done :       ServerHelloDone;
        cas certificate_verify :      CertificateVerify;
        cas client_key_exchange :     ClientKeyExchange;
        cas finished :                Finished;
    } body;
} Handshake;
```

Les messages du protocole de prise de contact sont présentés ci-dessous dans l'ordre dans lequel ils doivent être envoyés ; l'envoi des messages de prise de contact dans un ordre non attendu résulte en une erreur fatale. Des messages de prise de contact non nécessaires peuvent être cependant omis. Noter une exception à l'ordonnancement : le message Certificat est utilisé deux fois dans la prise de contact (du serveur au client, puis du client au serveur) mais n'est décrit que dans sa première position. C'est le seul message qui n'est pas lié par ces règles d'ordre dans le message Demande de Hello, qui peut être envoyé à tout moment, mais qui devrait être ignoré par le client si il arrive au milieu d'une prise de contact.

7.4.1 Messages Hello

Les messages de la phase hello sont utilisés pour échanger les capacités d'amélioration de la sécurité entre le client et le serveur. Lorsque commence une nouvelle session, les algorithmes de chiffrement, de hachage et de compression de l'état de connexion de la couche d'enregistrement sont initialisés à nul. L'état de connexion courant est utilisé pour les messages de renégociation.

7.4.1.1 Demande Hello

Quand ce message sera t-il envoyé ?
Le message Demande hello peut être envoyé à tout moment par le serveur.

Signification du message :
Demande hello est une simple notification au client qu'il devrait commencer à nouveau le processus de négociation en envoyant un message hello de client lorsque il le jugera convenable. Ce message sera ignoré par le client si il est actuellement en train de négocier une session. Ce message peut être ignoré par le client si il ne souhaite pas renégocier une session, ou le client peut, si il le souhaite, répondre par une alerte no_renegotiation. Comme les messages de prise de contact sont destinés à avoir la priorité de transmission sur les données d'application, on s'attend à ce que la négociation commence avant que pas plus de quelques enregistrements ne soient reçus du client. Si le serveur envoie une demande hello mais ne reçoit pas de client hello en réponse, il peut clore la connexion avec une alerte fatale.

Après l'envoi d'une demande hello, les serveurs ne devraient pas répéter la demande jusqu'à l'achèvement de la négociation de prise de contact suivante.

Structure de ce message :

```
struct { } HelloRequest;
```

Note : Ce message ne devrait jamais être inclus dans les hachages de message qui sont conservés pendant toute la prise de contact et utilisés dans les messages Terminé et le message de vérification de certificat.

7.4.1.2 Hello du client

Quand ce message sera–t-il envoyé ?
Lorsque un client se connecte pour la première fois à un serveur, il est obligé d'envoyer le client hello comme premier message. Le client peut aussi envoyer un client hello en réponse à une demande de hello ou de sa propre initiative afin de renégocier les paramètres de sécurité dans une connexion existante.

Structure de ce message :
Le message hello de client comporte une structure aléatoire, qui est utilisée plus tard dans le protocole.

```
struct {
  uint32 gmt_unix_time;
  opaque random_bytes[28];
} Random;
```

gmt_unix_time
C'est la date et l'heure actuelle en format standard UNIX à 32 bits (les secondes écoulées depuis minuit du 1er janvier 1970, GMT) selon l'horloge interne de l'envoyeur. Les horloges ne sont pas obligées d'être réglées correctement par le protocole TLS de base ; des protocoles de niveau supérieur ou d'application peuvent définir des exigences supplémentaires.

random_bytesCe sont 28 octets générés par un générateur sûr de nombres aléatoires.

Le message hello de client comporte un identifiant de session de longueur variable. Si il n'est pas vide, sa valeur identifie une session entre le même client et serveur dont le client souhaite réutiliser les paramètres de sécurité. L'identifiant de session peut venir d'une connexion antérieure, de cette connexion, ou d'une autre connexion actuellement active. La seconde option est utile si le client souhaite seulement mettre à jour les structures aléatoires et déduire des valeurs d'une connexion, alors que la troisième option rend possible d'établir plusieurs connexions sûres indépendantes sans répéter entièrement le protocole de prise de contact. Ces connexions indépendantes peuvent survenir en séquence ou simultanément ; un identifiant de session devient valide lorsque la prise de contact qui le négocie se termine avec l'échange des messages Terminé, et persiste jusqu'à ce qu'il soit retiré suite à sa péremption ou parce qu'une erreur fatale s'est rencontrée sur une connexion associée à la session. Le contenu réel de l'identifiant de session est défini par le serveur.

```
opaque SessionID<0..32>;
```

Avertissement
Comme l'identifiant de session est transmis sans chiffrement ou protection de MAC immédiate, les serveurs ne doivent pas placer d'informations confidentielles dans les identifiants de session ou laisser le contenu de faux identifiants de session causer de brèches dans la sécurité. (Noter que le contenu global de la prise de contact, y compris l'identifiant de session, est protégé par les messages Terminé échangés à la fin de la prise de contact.)

La liste CipherSuite, passée du client au serveur dans le message hello de client, contient les combinaisons des algorithmes cryptographiques pris en charge par le client dans l'ordre des préférences du client (le choix favori en premier). Chaque CipherSuite définit un algorithme d'échange de clés, un algorithme de chiffrement brut (y compris la longueur de la clé secrète) et un algorithme de MAC. Le serveur va choisir une suite de chiffrement ou, si aucun choix acceptable n'est présenté, retourner une alerte de défaillance de prise de contact, et fermer la connexion.

```
uint8 CipherSuite[2];          /* sélecteur de suite cryptographique */
```

Le hello de client comporte une liste d'algorithmes de compression acceptés par le client,dans l'ordre des préférences du client.

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

struct {
   ProtocolVersion client_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suites<2..2^16-1>;
   CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

client_version *(version de client)*
C'est la version du protocole TLS par laquelle le client souhaite communiquer durant cette session. Ce devrait être la dernière version (de plus forte valeur) acceptée par le client. Pour la présente version de la spécification, le numéro de version sera 3.1 (Voir à l'Appendice E les détails sur la rétro compatibilité).

random *(aléa)*
C'est une structure aléatoire générée par le client.

session_id *(identifiant de session)*
C'est l'identifiant de session que le client souhaite utiliser pour cette connexion. Ce champ devrait être vide si aucun session_id n'est disponible ou si le client souhaite générer de nouveaux paramètres de sécurité.

cipher_suites *(suites de chiffrement)*
C'est une liste des options de chiffrement qui sont prises en charge par le client, dans l'ordre des préférences du client. Si le champ session_id n'est pas vide (ce qui implique une demande de reprise de session) ce vecteur doit comporter au moins la cipher_suite provenant de cette session. Les valeurs sont définies à l'Appendice A.5.

compression_methods *(méthodes de compression)*
C'est une liste des méthodes de compression acceptées par le client, triées par préférence du client. Si le champ session_id n'est pas vide (ce qui implique une demande de reprise de session) il doit comporter la méthode de compression de cette session. Ce vecteur doit contenir, et toutes les mises en œuvre doivent le prendre en charge, CompressionMethod.null. Donc, un client et un serveur vont toujours être capables de s'accorder sur une méthode de compression.

Après l'envoi du message hello de client, le client attend un message hello du serveur. Tout autre message de prise de contact retourné par le serveur sauf une demande de hello est traitée comme une erreur fatale.

Note pour la compatibilité future :
Dans l'intérêt de la compatibilité future, il est permis qu'un message hello de client comporte des données supplémentaires après la méthode de compression. Ces données doivent être incluses dans les hachages de prise de contact, mais doivent autrement être ignorées. C'est le seul message de prise de contact pour lequel ceci est légal ; pour tous les autres messages, la quantité de données dans le message doit correspondre précisément à la description du message.

7.4.1.3 Hello du serveur

Quand ce message doit-il être envoyé ?
Le serveur va envoyer ce message en réponse à un message hello de client lorsque il a été capable de trouve un ensemble acceptable d'algorithmes. Si il ne peut pas trouver une telle correspondance, il va répondre par une alerte d'échec de prise de contact.

Structure de ce message :

```
struct {
   ProtocolVersion server_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suite;
   CompressionMethod compression_method;
} ServerHello;
```

server_version
Ce champ va contenir la valeur la plus faible de celles suggérées par le client dans le hello de client et la plus élevée de celles acceptées par le serveur. Pour la présente version de la spécification, la version est 3.1 (Voir à l'Appendice E les détails sur la rétro compatibilité).

random
Cette structure est générée par le serveur et doit être différente (et indépendante) du ClientHello.random.

session_id
C'est l'identité de la session qui correspond à cette connexion. Si le ClientHello.session_id n'était pas vide, le serveur va chercher une correspondance dans son antémémoire de session. Si il trouve une correspondance et si le serveur veut établir ma nouvelle connexion en utilisant l'état de session spécifiée, il va répondre avec la même valeur que celle fournie par le client. Cela indique une reprise de session et impose aux parties de procéder directement aux messages de fin. Autrement, ce champ va contenir une valeur différente qui identifiera la nouvelle session. Le serveur peut retourner un session_id vide pour indiquer que la session ne sera pas mise en antémémoire et ne pourra donc pas être reprise. Si une session est reprise, cela doit être fait en utilisant la même suite de chiffrement qu'avec laquelle elle a été négociée à l'origine.

cipher_suite
C'est la seule suite de chiffrement qui a été choisie par le serveur à partir de la liste dans ClientHello.cipher_suites. Pour les sessions reprises, ce champ a la valeur provenant de l'état de la session reprise.

compression_method
C'est le seul algorithme de compression choisi par le serveur à partir de la liste de ClientHello.compression_methods. Pour les sessions reprises, ce champ a la valeur provenant de l'état de la session reprise.

7.4.2 Certificat du serveur

Quand ce message sera-t-il envoyé :
Le serveur doit envoyer un certificat chaque fois que la méthode d'échange de clés qui a fait l'objet de l'accord n'est pas une méthode anonyme. Ce message va toujours immédiatement suivre le message hello du serveur.

Signification de ce message :
Le type de certificat doit être approprié à l'algorithme d'échange de clé de la suite de chiffrement choisie, et c'est généralement un certificat X.509v3. Il doit contenir une clé qui correspond à la méthode d'échange de clés comme décrit ci-après. Sauf spécification contraire, l'algorithme de signature pour le certificat doit être le même que l'algorithme pour la clé du certificat. Sauf spécification contraire, la clé publique peut avoir une longueur quelconque.

Algorithme d'échange de clé	**Type de clé de certificat**
RSA	Clé publique RSA ; le certificat doit permettre que la clé soit utilisée pour le chiffrement
RSA_EXPORT	Clé publique RSA de longueur supérieure à 512 qui peut être utilisée pour signer, ou clé de 512 bits ou moins qui peut être utilisée pour chiffrer ou signer.
DHE_DSS	Clé publique DSS.
DHE_DSS_EXPORT	Clé publique DSS.
DHE_RSA	Clé publique RSA qui peut être utilisée pour signer.
DHE_RSA_EXPORT	Clé publique RSA qui peut être utilisée pour signer.
DH_DSS	Clé Diffie-Hellman. L'algorithme utilisé pour signer le certificat devrait être DSS.
DH_RSA	Clé Diffie-Hellman. L'algorithme utilisé pour signer le certificat devrait être RSA.

Tous les profils de certificat, de clés et de formats cryptographiques sont définis par le groupe de travail PKIX de l'IETF [PKIX]. Lorsque une extension d'usage de clé est présente, le bit digitalSignature doit être établi pour la clé de façon à être éligible à la signature, comme décrit ci-dessus, et le bit keyEncipherment doit être présent pour permettre le chiffrement, comme décrit ci-dessus. Le bit keyAgreement doit être établi sur les certificats Diffie-Hellman.

Comme les suites de chiffrement qui spécifient de nouvelles méthodes d'échange de clés sont spécifiées pour le protocole TLS, elles comporteront le format de certificat et les informations de codage de clé nécessaires.

Structure de ce message :
```
opaque ASN.1Cert<1..2^24-1>;

struct {
   ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

certificate_list *(liste de certificats)*
C'est une séquence (chaîne) de certificats X.509v3. Le certificat de l'envoyeur doit être le premier de la liste. Chaque certificat suivant doit certifier directement celui qui le précède. Comme la validation de certificat exige que les clés racines soient distribuées de façon indépendante, le certificat auto signé qui spécifie l'autorité du certificat racine peut éventuellement être omis de la chaîne, en supposant que l'extrémité distante doit déjà le posséder afin de le valider en tout état de cause.

Les mêmes types et structures de message seront utilisés pour la réponse du client à un message de demande de certificat. Noter qu'un client peut n'envoyer aucun certificat si il n'a pas de certificat approprié à envoyer en réponse à la demande d'authentification du serveur.

Note : PKCS n° 7 [PKCS7] n'est pas utilisé comme format du vecteur de certificat parce que les certificats étendus de PKCS n° 6 [PKCS6] ne sont pas utilisés. PKCS n° 7 définit aussi un SET plutôt qu'une SEQUENCE, rendant la tâche d'analyse de la liste plus difficile.

7.4.3 Message d'échange de clés du serveur

Quand ce message sera-t-il envoyé ?
Ce message sera envoyé immédiatement après le message de certificat de serveur (ou le message hello du serveur, si c'est une négociation anonyme).

Le message d'échange de clés du serveur n'est envoyé par le serveur que lorsque le message de certificat du serveur (si il est envoyé) ne contient pas assez de données pour permettre au client d'échanger un secret pré-maître. Ceci est vrai pour les méthodes d'échange de clés suivantes :

RSA_EXPORT (si la clé publique dans le certificat de serveur fait plus de 512 bits)
DHE_DSS
DHE_DSS_EXPORT
DHE_RSA
DHE_RSA_EXPORT
DH_anon

Il n'est pas légal d'envoyer le message d'échange de clé du serveur pour les méthodes d'échange de clés suivantes :

RSA
RSA_EXPORT (lorsque la clé publique dans le certificat de serveur est inférieure ou égale à 512 bits)
DH_DSS
DH_RSA

Signification de ce message :
Ce message transporte les informations cryptographiques qui permettent au client de communiquer le secret pré-maître : soit une clé publique RSA pour chiffrer le secret pré-maître, soit une clé publique Diffie-Hellman avec laquelle le client peut achever un échange de clés (dont le résultat sera le secret pré-maître).

Comme des suites de chiffrement supplémentaires sont définies pour TLS et qu'elles incluent de nouveaux algorithmes d'échange de clés, le message d'échange de clés du serveur ne sera envoyé que si et seulement si le type de certificat associé à l'algorithme d'échange de clés ne fournit pas assez d'informations pour que le client échange un secret pré-maître.

Note : Selon la loi américaine actuelle, les modules RSA supérieurs à 512 bits ne doivent pas être utilisés pour les échanges de clés dans les logiciels exportés au dehors des USA. Avec ce message, les plus grandes clés RSA codées dans les certificats peuvent devoir être utilisées pour signer des clés RSA temporaires plus courtes pour la méthode d'échange de clés RSA_EXPORT.

Structure de ce message :

```
enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

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

rsa_modulus
C'est le module de la clé RSA temporaire du serveur.

rsa_exponent
C'est l'exposant public de la clé RSA temporaire du serveur.

```
struct {
   opaque dh_p<1..2^16-1>;
   opaque dh_g<1..2^16-1>;
   opaque dh_Ys<1..2^16-1>;
} ServerDHParams;     /* Paramètres DH éphémères */
```

dh_p
C'est le module principal utilisé pour le fonctionnement en Diffie-Hellman.

dh_g
C'est le générateur utilisé pour le fonctionnement en Diffie-Hellman.

dh_Ys
C'est la valeur publique Diffie-Hellman du serveur (g^X mod p).

```
struct {
   select (KeyExchangeAlgorithm) {
      cas diffie_hellman :
         ServerDHParams params;
         Signature signed_params;
      cas rsa :
```

```
            ServerRSAParams params;
            Signature signed_params;
        };
    } ServerKeyExchange;
```

params
Ce sont les paramètres d'échange de clé du serveur.

signed_params
Pour les échanges de clés non anonymes, c'est un hachage de la valeur des paramètres correspondants, avec la signature appropriée appliquée à ce hachage.

md5_hash
MD5(ClientHello.random + ServerHello.random + ServerParams);

sha_hash
SHA(ClientHello.random + ServerHello.random + ServerParams);

```
    enum { anonymous, rsa, dsa } SignatureAlgorithm;

    select (SignatureAlgorithm)
        {   cas anonyme : struct { };
            cas rsa :
                digitally-signed struct {
                    opaque md5_hash[16];
                    opaque sha_hash[20];
                };
            cas dsa :
                digitally-signed struct {
                    opaque sha_hash[20];
                };
        } Signature;
```

7.4.4 Demande de certificat

Quand ce message sera-t-il envoyé ?
Un serveur non anonyme peut facultativement demander un certificat au client, si c'est approprié pour la suite de chiffrement choisie. Ce message, si il est envoyé, va immédiatement suivre le message Échange de clé du serveur, (si il est envoyé ; autrement, il suit le message Certificat de serveur).

Structure de ce message :

```
    enum {
        rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4), (255)
    } ClientCertificateType;

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

    struct {
        ClientCertificateType certificate_types<1..2^8-1>;
        DistinguishedName certificate_authorities<3..2^16-1>;
    } CertificateRequest;
```

certificate_types
Ce champ est une liste des types de certificats demandés, triés dans l'ordre de préférence du serveur.

certificate_authorities
C'est une liste des noms distinctifs des autorités de certificat acceptables. Ces noms distinctifs peuvent spécifier un nom distinctif désiré pour une CA racine ou pour une CA subordonnée ; et donc, ce message peut être utilisé aussi bien pour décrire des racines connues qu'un espace d'autorisation désiré.

Note : DistinguishedName est dérivé de [X509].

Note : C'est une alerte d'échec de prise de contact fatale pour un serveur anonyme de demander l'identification du client.

7.4.5 Hello de serveur terminé

Quand ce message est-il envoyé ?
Le message hello terminé du serveur est envoyé par le serveur pour indiquer la fin du hello de serveur et des messages associés. Après l'envoi de ce message, le serveur va attendre la réponse d'un client.

Signification de ce message :
Ce message signifie que le serveur a fini d'envoyer des messages pour la prise en charge de l'échange de clés, et que le client peut procéder à sa phase de l'échange de clés.

À réception du message hello terminé du serveur, le client devrait vérifier que le serveur a fourni un certificat valide si nécessaire, et vérifier que les paramètres hello de serveur sont acceptables.

Structure de ce message :

```
struct { } ServerHelloDone;
```

7.4.6 Certificat du client

Quand ce message est-il envoyé ?
C'est le premier message que le client peut envoyer après avoir reçu un message hello terminé du serveur. Ce message n'est envoyé que si le serveur demande un certificat. Si aucun certificat convenable n'est disponible, le client devrait envoyer un message Certificat ne contenant pas de certificat. Si l'authentification du client est exigée par le serveur pour que la prise de contact continue, il peut répondre par une alerte d'échec fatal de prise de contact. Les certificats de client sont envoyés en utilisant la structure Certificate définie au paragraphe 7.4.2.

Note : Lors de l'utilisation d'une méthode d'échange de clé fondée sur un Diffie-Hellman statique (DH_DSS or DH_RSA), si l'authentification du client est exigée, le groupe et le générateur Diffie-Hellman codés dans le certificat du client doivent correspondre aux paramètres Diffie-Hellman spécifiés par le serveur si les paramètres du client doivent être utilisés pour l'échange de clés.

7.4.7 Message d'échange de clé du client

Quand ce message est-il envoyé ?
Ce message est toujours envoyé par le client. Il sera immédiatement après le message certificate du client, s'il est envoyé. Autrement, il sera le premier message envoyé par le client après avoir reçu le message terminé du serveur.

Signification de ce message :
Avec ce message est établi le secret pré-maître, soit par transmission directe du secret chiffré avec RSA, soit par la transmission des paramètres Diffie-Hellman qui vont permettre à chaque côté de se mettre d'accord sur le même secret pré-maître. Lorsque la méthode d'échange de clés est DH_RSA ou DH_DSS, la certification du client a été demandée, et le client a été capable de répondre avec un certificat qui contient une clé publique Diffie-Hellman dont les paramètres (groupe et générateur) correspondent à ceux spécifiés par le serveur dans son certificat, ce message ne contiendra aucune donnée.

Structure de ce message :
Le choix des messages dépend de la méthode d'échange de clés qui a été choisie. Voir au paragraphe 7.4.3 la définition de KeyExchangeAlgorithm.

```
struct {
   select (KeyExchangeAlgorithm) {
      cas rsa : EncryptedPreMasterSecret;
      cas diffie_hellman : ClientDiffieHellmanPublic;
   } exchange_keys;
} ClientKeyExchange;
```

7.4.7.1 Message chiffré en RSA du secret pré maître

Signification de ce message :
Si RSA est utilisé pour l'accord de clés et l'authentification, le client génère un secret pré-maître de 48 octets, le chiffre en utilisant la clé publique provenant du certificat du serveur ou la clé RSA temporaire fournie dans un message d'échange de

clé de serveur, et envoie le résultat dans un message de secret pré-maître chiffré. Cette structure est une variante du message d'échange de clé de client, et n'est pas un message en elle-même.

Structure de ce message :

```
struct {
   ProtocolVersion client_version; opaque random[46];
} PreMasterSecret;
```

client_version
C'est la dernière (plus récente) version supportée par le client. C'est utilisé pour détecter les attaques de dégradation de version. À réception du secret pré-maître, le serveur devrait vérifier que cette valeur correspond à la valeur transmise par le client dans le message hello de client.

random
46 octets aléatoires générés de façon sûre.

```
struct {
   public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;
```

Note : Une attaque découverte par Daniel Bleichenbacher [BLEI] peut être utilisée pour attaquer un serveur TLS qui utilise RSA codé en PKCS n°1. L'attaque tire parti du fait qu'en échouant de différentes façons, un serveur TLS peut être contraint à révéler si un message particulier, lorsqu'il est déchiffré, est correctement formaté en PKCS n° 1 ou non. La meilleure façon d'éviter d'être vulnérable à cette attaque est de traiter incorrectement les messages formatés d'une façon indistinguable des blocs RSA correctement formatés. Donc, lorsque il reçoit un bloc RSA incorrectement formaté, un serveur devrait générer une valeur aléatoire de 48 octets et continuer en l'utilisant comme secret pré-maître. Donc, le serveur va agir de façon identique qu'il reçoive un bloc RSA codé correctement ou non.

pre_master_secret
Cette valeur aléatoire générée par le client est utilisée pour générer le secret maître, comme spécifié au paragraphe 8.1.

7.4.7.2 Valeur publique Diffie-Hellman du client

Signification de ce message :
Cette structure transporte la valeur publique Diffie-Hellman du client (Yc) si elle n'était pas déjà incluse dans le certificat du client. Le codage utilisé pour Yc est déterminé par l'énumération PublicValueEncoding. Cette structure est une variante du message d'échange de clé de client, et non un message en elle-même.

Structure de ce message :

```
enum { implicit, explicit } PublicValueEncoding;
```

implicit
Si le certificat de client contenait déjà une clé Diffie-Hellman convenable, alors Yc est implicite et il n'est pas nécessaire de l'envoyer à nouveau. Dans ce cas, le message d'échange de clé de client sera envoyé, mais il sera vide.

explicit
Yc doit être envoyé.

```
struct {
   select (PublicValueEncoding) {
      cas implicite : struct { };
      cas explicit : opaque dh_Yc<1..2^16-1>;
   } dh_public;
} ClientDiffieHellmanPublic;
```

dh_Yc
C'est la valeur publique Diffie-Hellman du client (Yc).

7.4.8 Vérification de certificat

Quand ce message est-il envoyé ?
Ce message est utilisé pour fournir une vérification explicite d'un certificat de client. Ce message n'est envoyé qu'à la suite

d'un certificat de client qui a une capacité de signature (c'est-à-dire, tous les certificats sauf ceux qui contiennent des paramètres Diffie-Hellman fixes). Lorsque il est envoyé, il va immédiatement suivre le message d'échange de clé du client.

Structure de ce message :

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

Le type Signature est défini au paragraphe 7.4.3.

```
CertificateVerify.signature.md5_hash
MD5(handshake_messages);

Certificate.signature.sha_hash
SHA(handshake_messages);
```

Où handshake_messages se réfère à tous les messages de prise de contact envoyés ou reçus en commençant par le hello du client jusqu'à ce message non inclus, incluant les champs de type et de longueur des messages de prise de contact. C'est l'enchaînement de toutes les structures Handshake échangées jusqu'alors, comme défini au paragraphe 7.4.

7.4.9 Terminé

Quand ce message est-il envoyé ?
Un message Terminé est toujours envoyé immédiatement après un message Changement de spécification de chiffrement pour vérifier que les processus d'échange de clés et d'authentification ont réussi. Il est essentiel qu'un message Changement de spécification de chiffrement soit reçu entre les autres messages de prise de contact et le message Terminé.

Signification de ce message :
Le message Terminé est le premier protégé avec les algorithmes, clés, et secrets qui viennent d'être négociés. Les receveurs des messages Terminé doivent vérifier que leur contenu est correct. Une fois qu'un côté a envoyé son message Terminé et reçu et validé le message Terminé de son homologue, il peut commencer à envoyer et recevoir des données d'application sur la connexion.

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

verify_data
PRF(master_secret, finished_label, MD5(handshake_messages) + SHA-1(handshake_messages)) [0..11];

finished_label
Pour les messages Terminé envoyés par le client, la chaîne "client terminé". Pour les messages Terminé envoyés par le serveur, la chaîne "serveur terminé".

handshake_messages
Toutes les données depuis les messages de prise de contact jusqu'à ce message, non inclus. Ce sont seulement les données visibles à la couche de prise de contact et cela n'inclut pas les en-têtes de couche enregistrement.

C'est la concaténation de toutes les structures Prise de contact telles que définies au paragraphe 7.4 échangées jusqu'alors.

C'est une erreur fatale si un message terminé n'est pas précédé par un message Changement de spécification de chiffrement au point approprié de la prise de contact.

Le hachage contenu dans les messages Terminé envoyés par le serveur incorporent Sender.server ; ceux qui sont envoyés par le client incorporent Sender.client. La valeur handshake_messages inclut tous les messages de prise de contact en commençant au hello du client jusqu'à ce message Terminé, mais non inclus. Cela peut être différent des handshake_messages du paragraphe 7.4.8 parce que cela inclurait le message Vérifier le certificat (s'il est envoyé). Aussi, les handshake_messages pour le message Terminé envoyé par le client seront différents du message Terminé envoyé par le serveur, parce que celui qui est envoyé en second va inclure le premier.

Note : Les messages Changer la spécification de chiffrement, les alertes et tous les autres types d'enregistrement ne sont pas des messages de prise de contact et ne sont pas inclus dans le calcul du hachage. Aussi, les messages de demande de Hello sont-ils omis des hachages de prise de contact.

8. Calculs cryptographiques

Afin de commencer la protection de la connexion, le protocole d'enregistrement TLS exige la spécification d'une suite d'algorithmes, d'un secret maître, et de valeurs aléatoires du client et du serveur. Les algorithmes d'authentification, de chiffrement et de MAC sont déterminés par la cipher_suite choisie par le serveur et révélés dans le message hello du serveur. L'algorithme de compression est négocié dans les messages hello, et les valeurs aléatoires sont échangées dans les messages hello. Tous ce qui reste à faire est de calculer le secret maître.

8.1 Calcul du secret maître

Pour toutes les méthodes d'échange de clés, le même algorithme est utilisé pour convertir le secret pré-maître en secret maître. Le secret pré-maître devrait être supprimé de la mémoire une fois que le secret maître a été calculé.

master_secret = PRF(pre_master_secret, "master secret", ClientHello.random + ServerHello.random) [0..47];

Le secret maître est toujours exactement long de 48 octets. La longueur du secret pré-maître va varier selon la méthode d'échange de clés.

8.1.1 RSA

Lorsqu'on utilise RSA pour l'authentification du serveur et l'échange de clé, un secret pré-maître de 48 octets est généré par le client, chiffré avec la clé publique du serveur, et envoyé au serveur. Le serveur utilise sa clé privée pour déchiffrer le secret pré-maître. Les deux parties convertissent alors le secret pré-maître en secret maître, comme spécifié ci-dessus.

Les signatures numériques RSA sont effectuées en utilisant le type 1 de bloc PKCS n° 1 [PKCS1]. Le chiffrement de clé publique RSA est effectué en utilisant le type 2 de bloc PKCS n° 1.

8.1.2 Diffie-Hellman

Un calcul conventionnel Diffie-Hellman est effectué. La clé (Z) négociée est utilisée comme secret pré-maître, et est convertie en secret maître, comme spécifié ci-dessus.

Note : Les paramètres Diffie-Hellman sont spécifiés par le serveur, et peuvent être éphémères ou contenus au sein du certificat du serveur.

9. Suites de chiffrement obligatoires

En l'absence d'une norme de profil d'application spécifiant autre chose, une application conforme à TLS DOIT mettre en œuvre la suite de chiffrement TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA.

10. Protocole des données d'application

Les messages de données d'application sont portés par la couche Enregistrement et sont fragmentés, compressés et chiffrés sur la base de l'état de connexion courant. Les messages sont traités comme données transparentes pour la couche d'enregistrement.

A. Valeurs des constantes du protocole

La présente section décrit les types et les constantes du protocole.

A.1 Couche Enregistrement

```
struct {
    uint8 major, minor;
} ProtocolVersion;
```

```
ProtocolVersion version = { 3, 1 };     /* TLS v1.0 */

enum {
   change_cipher_spec(20), alert(21), handshake(22),
   application_data(23), (255)
} ContentType;

struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
   opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
   opaque fragment[TLSCompressed.length];
} TLSCompressed;

struct {
   ContentType type;
   ProtocolVersion version;
   uint16 length;
   select (CipherSpec.cipher_type) {
      cas flux : GenericStreamCipher;
      cas bloc :  GenericBlockCipher;
   } fragment;
} TLSCiphertext;

stream-ciphered struct {
   opaque content[TLSCompressed.length];
   opaque MAC[CipherSpec.hash_size];
} GenericStreamCipher;

block-ciphered struct {
   opaque content[TLSCompressed.length];
   opaque MAC[CipherSpec.hash_size];
   uint8 padding[GenericBlockCipher.padding_length];
   uint8 padding_length;
} GenericBlockCipher;
```

A.2 Message Changer la spécification de chiffrement

```
struct {
   enum { change_cipher_spec(1), (255) } type;
} ChangeCipherSpec;
```

A.3 Messages d'alerte

```
enum { warning(1), fatal(2), (255) } AlertLevel;

   enum {
      close_notify(0),
      unexpected_message(10),
      bad_record_mac(20),
      decryption_failed(21),
      record_overflow(22),
      decompression_failure(30),
      handshake_failure(40),
      bad_certificate(42),
```

```
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        export_restriction(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        (255)
    } AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

A.4 Protocole de prise de contact

```
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;

struct {
    HandshakeType msg_type;
    uint24 length;
    select (HandshakeType) {
        cas hello_request :          HelloRequest;
        cas client_hello :           ClientHello;
        cas server_hello :           ServerHello;
        cas certificate :            Certificate;
        cas server_key_exchange :    ServerKeyExchange;
        cas certificate_request :    CertificateRequest;
        cas server_hello_done :      ServerHelloDone;
        cas certificate_verify :     CertificateVerify;
        cas client_key_exchange :    ClientKeyExchange;
        cas finished :               Finished;
    } body;
} Handshake;
```

A.4.1 Messages Hello

```
struct { } HelloRequest;

struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;

opaque SessionID<0..32>;

uint8 CipherSuite[2];

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

```
struct {
   ProtocolVersion client_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suites<2..2^16-1>;
   CompressionMethod compression_methods<1..2^8-1>;

} ClientHello;

struct {
   ProtocolVersion server_version;
   Random random;
   SessionID session_id;
   CipherSuite cipher_suite;
   CompressionMethod compression_method;
} ServerHello;
```

A.4.2 Authentification du serveur et messages d'échange de clés

```
opaque ASN.1Cert<2^24-1>;

struct {
   ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;

enum { rsa, diffie_hellman } KeyExchangeAlgorithm;

struct {
   opaque RSA_modulus<1..2^16-1>;
   opaque RSA_exponent<1..2^16-1>;
} ServerRSAParams;

struct {
   opaque DH_p<1..2^16-1>;
   opaque DH_g<1..2^16-1>;
   opaque DH_Ys<1..2^16-1>;
} ServerDHParams;

struct {
   select (KeyExchangeAlgorithm) {
      cas diffie_hellman :
         ServerDHParams params;
         Signature signed_params;
      cas rsa :
         ServerRSAParams params;
         Signature signed_params;
   };
} ServerKeyExchange;

enum { anonymous, rsa, dsa } SignatureAlgorithm;

select (SignatureAlgorithm)
{   cas anonymous :   struct { };
   cas rsa :          digitally-signed struct {
                          opaque md5_hash[16];
                          opaque sha_hash[20];
                          };
   cas dsa :          digitally-signed struct {
                          opaque sha_hash[20];
                          };
} Signature;
```

```
enum {
   rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4), (255)
} ClientCertificateType;

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

struct {
   ClientCertificateType certificate_types<1..2^8-1>;
   DistinguishedName certificate_authorities<3..2^16-1>;
} CertificateRequest;

struct { } ServerHelloDone;
```

A.4.3 Authentification du client et messages d'échange de clés

```
struct {
   select (KeyExchangeAlgorithm) {
      cas rsa :                  EncryptedPreMasterSecret;
      cas diffie_hellman :       DiffieHellmanClientPublicValue;
   } exchange_keys;
} ClientKeyExchange;

struct {
   ProtocolVersion client_version;
   opaque random[46];

} PreMasterSecret;

struct {
   public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

struct {
   select (PublicValueEncoding) {
      cas implicite :     struct {};
      cas explicite :     opaque DH_Yc<1..2^16-1>;

   } dh_public;
} ClientDiffieHellmanPublic;

struct {
   Signature signature;
} CertificateVerify;
```

A.4.4 Message de finalisation de la prise de contact

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

A.5 Suite de chiffrement

Les valeurs suivantes définissent les codes de suites de chiffrement utilisées dans les messages hello client et serveur.

Une suite de chiffrement définit une spécification de chiffrement prise en charge dans TLS version 1.0.

TLS_NULL_WITH_NULL_NULL est spécifié et est l'état initial d'une connexion TLS durant la première prise de contact sur ce canal, mais ne doit pas être négocié, car il ne procure pas plus de protection qu'une connexion non sécurisée.

```
CipherSuite TLS_NULL_WITH_NULL_NULL = { 0x00,0x00 };
```

Les définitions de suite de chiffrement suivantes exigent que le serveur fournisse un certificat qui puisse être utilisé pour l'échange de clés. Le serveur peut demander un certificat à capacité de signature RSA ou DSS dans le message de demande de certificat.

```
CipherSuite TLS_RSA_WITH_NULL_MD5                  = { 0x00,0x01 };
CipherSuite TLS_RSA_WITH_NULL_SHA                  = { 0x00,0x02 };
CipherSuite TLS_RSA_EXPORT_WITH_RC4_40_MD5         = { 0x00,0x03 };
CipherSuite TLS_RSA_WITH_RC4_128_MD5               = { 0x00,0x04 };
CipherSuite TLS_RSA_WITH_RC4_128_SHA               = { 0x00,0x05 };
CipherSuite TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5     = { 0x00,0x06 };
CipherSuite TLS_RSA_WITH_IDEA_CBC_SHA              = { 0x00,0x07 };
CipherSuite TLS_RSA_EXPORT_WITH_DES40_CBC_SHA      = { 0x00,0x08 };
CipherSuite TLS_RSA_WITH_DES_CBC_SHA               = { 0x00,0x09 };
CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA          = { 0x00,0x0A };
```

Les définitions de suite de chiffrement suivantes sont utilisées pour le Diffie-Hellman authentifié par le serveur (et facultativement authentifié par le client). DH note les suites de chiffrement dans lesquelles le certificat du serveur contient les paramètres Diffie-Hellman signés par l'autorité de certificat (CA). DHE note le Diffie-Hellman éphémère, où les paramètres Diffie-Hellman sont signés par un certificat DSS ou RSA, qui a été signé par le CA. L'algorithme de signature utilisé est spécifié après le paramètre DH ou DHE. Le serveur peut demander un certificat à capacité de signature RSA ou DSS au client pour l'authentification de client ou il peut demander un certificat Diffie-Hellman. Tout certificat Diffie-Hellman fourni par le client doit utiliser les paramètres (groupe et générateur) décrits par le serveur.

```
CipherSuite TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0B };
CipherSuite TLS_DH_DSS_WITH_DES_CBC_SHA            = { 0x00,0x0C };
CipherSuite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x0D };
CipherSuite TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA   = { 0x00,0x0E };
CipherSuite TLS_DH_RSA_WITH_DES_CBC_SHA            = { 0x00,0x0F };
CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x10 };
CipherSuite TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x11 };
CipherSuite TLS_DHE_DSS_WITH_DES_CBC_SHA           = { 0x00,0x12 };
CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x13 };
CipherSuite TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x14 };
CipherSuite TLS_DHE_RSA_WITH_DES_CBC_SHA           = { 0x00,0x15 };
CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x16 };
```

Les suites de chiffrement suivantes sont utilisées pour les communications Diffie-Hellman complètement anonymes dans lesquelles aucune des parties n'est authentifiée. Noter que ce mode est vulnérable aux attaques par interposition et est donc déconseillé.

```
CipherSuite TLS_DH_anon_EXPORT_WITH_RC4_40_MD5     = { 0x00,0x17 };
CipherSuite TLS_DH_anon_WITH_RC4_128_MD5           = { 0x00,0x18 };
CipherSuite TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA  = { 0x00,0x19 };
CipherSuite TLS_DH_anon_WITH_DES_CBC_SHA           = { 0x00,0x1A };
CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA      = { 0x00,0x1B };
```

Note : Toutes les suites de chiffrement dont le premier octet est 0xFF sont considérées comme privées et peuvent être utilisées pour définir des algorithmes locaux/expérimentaux. L'interopérabilité de ces types est une affaire locale.

Note : Des suites de chiffrement supplémentaires peuvent être enregistrées en publiant une RFC qui spécifie les suites de chiffrement, incluant les informations de protocole TLS nécessaires, y compris le codage du message, la déduction du secret pré-maître, le chiffrement symétrique et le calcul du MAC et les informations de référence appropriées pour les algorithmes impliqués. Le bureau de l'éditeur des RFC peut, à sa discrétion, choisir de publier des spécifications pour les suites de chiffrement qui ne sont pas complètement décrites (par exemple, pour des algorithmes secrets) si il trouve que la spécification est d'intérêt technique et complètement spécifiée.

Note : Les valeurs de suite de chiffrement { 0x00, 0x1C } et { 0x00, 0x1D } sont réservées pour éviter des collisions avec les suites de chiffrement fondées sur Fortezza dans SSL 3.

A.6 Paramètres de sécurité

Ces paramètres de sécurité sont déterminés par le protocole de prise de contact TLS et fournis comme paramètres à la couche Enregistrement TLS afin d'initialiser un état de connexion. Les paramètres de sécurité incluent :

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

enum { server, client } ConnectionEnd;

enum { null, rc4, rc2, des, 3des, des40, idea } BulkCipherAlgorithm;

enum { stream, block } CipherType;

enum { true, false } IsExportable;

enum { null, md5, sha } MACAlgorithm;
```

/* Les algorithmes spécifiés dans CompressionMethod, BulkCipherAlgorithm, et MACAlgorithm peuvent y être ajoutés. */

```
struct {
   ConnectionEnd entity;
   BulkCipherAlgorithm bulk_cipher_algorithm;
   CipherType cipher_type;
   uint8 key_size;
   uint8 key_material_length;
   IsExportable is_exportable;
   MACAlgorithm mac_algorithm;
   uint8 hash_size;
   CompressionMethod compression_algorithm;
   opaque master_secret[48];
   opaque client_random[32];
   opaque server_random[32];
} SecurityParameters;
```

B. Glossaire

protocole d'application
Un protocole d'application est un protocole qui se met normalement en couches directement par dessus la couche transport (par exemple, TCP/IP). Les exemples incluent HTTP, TELNET, FTP, et SMTP.

chiffrement asymétrique
Voir cryptographie à clés publique.

authentification
C'est la capacité d'une entité à déterminer l'identité d'une autre entité.

chiffrement de bloc
Un chiffrement de bloc est un algorithme qui fonctionne sur le texte en clair en groupes de bits, appelés blocs. 64 bits est une taille de bloc courante.

chiffrement en vrac *(bulk cipher)*
C'est un algorithme de chiffrement symétrique utilisé pour chiffrer de grandes quantités de données.

chaînage de bloc de chiffrement (CBC, *cipher block chaining*)
CBC est un mode dans lequel chaque bloc de texte en clair chiffré avec un chiffrement de bloc est d'abord traité avec l'opération OU exclusif avec le bloc de texte chiffré précédent (ou, dans le cas du premier bloc, avec le vecteur d'initialisation). Pour le déchiffrement, chaque bloc est d'abord déchiffré, puis combiné par l'opération OU exclusif avec le bloc de texte chiffré précédent (ou vecteur d'initialisation).

certificat
Au titre du protocole X.509 (dit aussi cadre d'authentification ISO) les certificats sont alloués par une autorité de certificat de confiance et apportent une forte liaison entre l'identité d'une partie, ou quelque autre attribut, et sa clé publique.

client
L'entité d'application qui initie une connexion TLS avec un serveur. Cela peut impliquer ou non que le client a initié la connexion de transport sous-jacente. La principale différence opérationnelle entre le serveur et le client est que le serveur est généralement authentifié, alors que le client n'est seulement authentifié que facultativement.

clé d'écriture client
C'est la clé utilisée pour chiffrer les données écrites par le client.

secret MAC d'écriture client
Ce sont les données secrètes utilisées pour authentifier les données écrites par le client.

connexion
Une connexion est un transport (dans la définition du modèle de mise en couches de l'OSI) qui fournit un type convenable de service. Pour TLS, de telles connexions sont des relations d'homologue à homologue. Les connexions sont provisoires. Chaque connexion est associée à une session.

norme de chiffrement de données (DES, *Data Encryption Standard*)
DES est un algorithme de chiffrement symétrique très largement utilisé. C'est un chiffrement de bloc avec une clé de 56 bits et une taille de bloc de 8 octets. Noter que dans TLS, pour les besoins de la génération des clés, DES est traité comme ayant une longueur de clé de 8 octets (64 bits) mais il ne fournit quand même qu'une protection de 56 bits. (Le bit de moindre poids de chaque octet de clé est présumé être mis à un pour produire une imparité dans cet octet de clé.) DES peut aussi fonctionner dans un mode où trois clés indépendantes et trois chiffrements sont utilisés pour chaque bloc de données ; cela utilise 168 bits de clé (24 octets dans la méthode de génération de clé TLS) et fournit l'équivalent de 112 bits de sécurité. [DES], [3DES]

norme de signature numérique (DSS, *Digital Signature Standard*)
C'est une norme de signature numérique, incluant l'algorithme de signature numérique, approuvé par l'Institut national des normes et des technologie (NIST), définie dans NIST FIPS PUB 186, "Digital Signature Standard," publiée en mai 1994 par le Ministère U.S. du commerce. [DSS]

signatures numériques
Les signatures numériques utilisent le chiffrement à clé publique et des fonctions de hachage unidirectionnelles pour produire une signature des données qui peut être authentifiée, et est difficile à falsifier ou répudier.

prise de contact *(handshake)*
C'est une négociation initiale entre client et serveur et qui établit les paramètres de leurs transactions.

vecteur d'initialisation (IV, *Initialization Vector*)
Lorsque un chiffrement de bloc est utilisé en mode CBC, le vecteur d'initialisation est combiné par l'opérateur OU exclusif avec le premier bloc de texte en clair avant le chiffrement.

IDEA
C'est un chiffrement de bloc de 64 bits conçu par Xuejia Lai et James Massey. [IDEA]

code d'authentification de message (MAC, *Message Authentication Code*)
C'est un hachage unidirectionnel calculé à partir d'un message et de certaines données secrètes. Il est difficile de le falsifier sans connaître les données secrètes. Son objet est de détecter si le message a été altéré.

secret maître
Ce sont des données secrètes utilisées pour générer des clés de chiffrement, des MAC secrets, et des IV.

MD5
MD5 est une fonction de hachage sûre qui convertit un flux de données de longueur arbitraire en un résumé de taille fixe (16 octets). [MD5]

cryptographie à clés publiques
C'est une classe de techniques de chiffrement qui emploient des chiffrements à deux clés. Les messages chiffrés avec la clé publique ne peuvent être déchiffrés qu'avec la clé privée associée. À l'inverse, les messages signés avec la clé privée ne peuvent être vérifiés qu'avec la clé publique.

fonction de hachage unidirectionnelle
C'est une transformation unidirectionnelle qui convertit une quantité arbitraire de données en un hachage de longueur fixe.

Il est difficile d'inverser le calcul de la transformation ou de trouver des collisions. MD5 et SHA sont des exemples de fonctions de hachage unidirectionnelles.

RC2
Chiffrement de bloc développé par Ron Rivest à RSA Data Security, Inc. [RSADSI] décrit dans [RC2].

RC4
Chiffrement de flux breveté par RSA Data Security [RSADSI]. Un chiffrement compatible est décrit dans [RC4].

RSA
Algorithme à clé publique très répandu qui peut être utilisé pour le chiffrement ou la signature numérique. [RSA]

sel
Données aléatoires non secrètes utilisées pour rendre les clés de chiffrement exportées résistantes aux attaques de pré-calcul.

serveur
Le serveur est l'entité d'application qui répond aux demandes de connexions provenant des clients. Voir aussi client.

session
Une session TLS est une association entre un client et un serveur. Les sessions sont crées par le protocole de prise de contact. Les sessions définissent un ensemble de paramètres de sécurité cryptographique, qui peut être partagé entre plusieurs connexions. Les sessions sont utilisées pour éviter de coûteuses négociations de nouveaux paramètres de sécurité pour chaque connexion.

identifiant de session
C'est une valeur générée par un serveur et qui identifie une session particulière.

clé d'écriture serveur
La clé utilisée pour chiffrer les données écrites par le serveur.

secret MAC d'écriture serveur
Ce sont les données secrètes utilisées pour authentifier les données écrites par le serveur.

algorithme de hachage sécurisé (SHA, *Secure Hash Algorithm*)
Il est défini dans la publication PUB 180-1 de FIPS. Il produit un résultat de 20 octets. Noter que toutes les références à SHA utilisent en fait l'algorithme modifié SHA-1. [SHA]

couche de prise sécurisée (SSL, *Secure Socket Layer*)
C'est le protocole de couche de prise sécurisée de Netscape [SSL3]. TLS se fonde sur SSL version 3.0.

chiffrement de flux
C'est un algorithme de chiffrement qui convertit une clé en un flux de clé cryptographiquement fort, qui est ensuite combiné par opérateur OU exclusif avec le texte en clair.

chiffrement symétrique
Voir chiffrement en vrac

sécurité de la couche transport (TLS, *Transport Layer Security*)
C'est le présent protocole ; mais aussi, le groupe de travail Transport Layer Security de l'équipe d'ingénierie de l'Internet (IETF, *Internet Engineering Task Force*). Voir le paragraphe "Commentaires" à la fin du présent document.

C. Définitions des suites de chiffrement

Suite de chiffrement	IsExportable	Échange de clés	Chiffrement	Hachage
TLS_NULL_WITH_NULL_NULL	*	NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5	*	RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA	*	RSA	NULL	SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5	*	RSA_EXPORT	RC4_40	MD5
TLS_RSA_WITH_RC4_128_MD5		RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA		RSA	RC4_128	SHA

TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	*	RSA_EXPORT	RC2_CBC_40	MD5
TLS_RSA_WITH_IDEA_CBC_SHA		RSA	IDEA_CBC	SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	*	RSA_EXPORT	DES40_CBC	SHA
TLS_RSA_WITH_DES_CBC_SHA		RSA	DES_CBC	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA		RSA	3DES_EDE_CBC	SHA
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	*	DH_DSS_EXPORT	DES40_CBC	SHA
TLS_DH_DSS_WITH_DES_CBC_SHA		DH_DSS	DES_CBC	SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA		DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	*	DH_RSA_EXPORT	DES40_CBC	SHA
TLS_DH_RSA_WITH_DES_CBC_SHA		DH_RSA	DES_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA		DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	*	DHE_DSS_EXPORT	DES40_CBC	SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA		DHE_DSS	DES_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA		DHE_DSS	3DES_EDE_CBC	SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	*	DHE_RSA_EXPORT	DES40_CBC	SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA		DHE_RSA	DES_CBC	SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA		DHE_RSA	3DES_EDE_CBC	SHA
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	*	DH_anon_EXPORT	RC4_40	MD5
TLS_DH_anon_WITH_RC4_128_MD5		DH_anon	RC4_128	MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA		DH_anon	DES40_CBC	SHA
TLS_DH_anon_WITH_DES_CBC_SHA		DH_anon	DES_CBC	SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA		DH_anon	3DES_EDE_CBC	SHA

* Indique que IsExportable est vrai

Algorithme d'échange de clés	**Description**	**Limite de taille de clé**
DHE_DSS	DH éphémère avec signatures DSS	aucune
DHE_DSS_EXPORT	DH éphémère avec signatures DSS	DH = 512 bits
DHE_RSA	DH éphémère avec signatures RSA	aucune
DHE_RSA_EXPORT	DH éphémère avec signatures RSA	DH = 512 bits, RSA = aucune
DH_anon	DH anonyme, pas de signature	aucune
DH_anon_EXPORT	DH anonyme, pas de signature	DH = 512 bits
DH_DSS	DH avec certificats fondés sur DSS	aucune
DH_DSS_EXPORT	DH avec certificats fondés sur DSS	DH = 512 bits
DH_RSA	DH avec certificats fondés sur RSA	aucune
DH_RSA_EXPORT	DH avec certificats fondés sur RSA	DH = 512 bits, RSA = aucune
NULL	Pas d'échange de clés	non disponible
RSA	Échange de clés RSA	aucune
RSA_EXPORT	Échange de clés RSA	RSA = 512 bits

Limite de taille de clé
La limite de taille de clé donne la taille de la plus grande clé publique qui peut être légalement utilisée pour le chiffrement dans les suites de chiffrement qui sont exportables.

Chiffrement	Type	Matériel de clé	Matériel de clé développé	Bits de clé effectifs	Taille de l'IV	Taille de bloc
NULL	* Flux	0	0	0	0	N/A
IDEA_CBC	Bloc	16	16	128	8	8
RC2_CBC_40	* Bloc	5	16	40	8	8
RC4_40	* Flux	5	16	40	0	N/A
RC4_128	Flux	16	16	128	0	N/A
DES40_CBC	* Bloc	5	8	40	8	8
DES_CBC	Bloc	8	8	56	8	8
3DES_EDE_CBC	Bloc	24	24	168	8	8

* Indique que IsExportable est vrai.

Type
Indique si c'est un chiffrement de flux ou un chiffrement de bloc fonctionnant en mode CBC.

Matériel de clé
C'est le nombre d'octets provenant du key_block qui sont utilisés pour générer les clés d'écriture.

Matériel de clé développé
C'est le nombre d'octets réellement fournis à l'algorithme de chiffrement.

Bits de clé effectifs
Combien de matériel d'entropie est dans le matériel de clé alimenté dans les programmes de chiffrement.

Taille de l'IV
Quelle quantité de données doit être générée pour le vecteur d'initialisation. Zéro pour les chiffrements de flux ; égal à la taille de bloc pour les chiffrements de bloc.

Taille du bloc
Quantité de données qu'un chiffrement de bloc chiffre en un seul tronçon ; un chiffrement de bloc fonctionnant en mode CBC peut seulement chiffrer un multiple pair de se taille de bloc.

Fonction de hachage	**Taille de hachage**	**Taille du bourrage**
NULL	0	0
MD5	16	48
SHA	20	40

D. Notes de mise en œuvre

Le protocole TLS ne peut pas empêcher de nombreuses fautes de sécurité courantes. La présente section donne plusieurs recommandations pour aider les mises en œuvre.

D.1 Clés RSA temporaires

Les restrictions américaines à l'export limitent les clés RSA utilisées pour le chiffrement à 512 bits, mais ne fixent aucune limite sur la longueur des clés RSA utilisées pour les opérations de signature. Les certificats ont souvent besoin d'être de plus de 512 bits, car les clés RSA de 512 bits ne sont pas assez sûres pour les transactions de grande valeur ou pour des applications qui exigent une sécurité à long terme. Certains certificats sont aussi conçus seulement pour la signature, auquel cas ils ne peuvent pas être utilisés pour l'échange de clés.

Lorsque la clé publique dans le certificat ne peut pas être utilisée pour le chiffrement, le serveur signe une clé RSA temporaire, qui est ensuite échangée. Dans les applications exportables, la clé RSA temporaire devrait être de la longueur maximum admissible (c'est-à-dire, 512 bits). Comme les clés RSA de 512 bits sont relativement peu sûres, elles devraient être souvent changées. Pour les applications typiques du commerce électronique, il est suggéré que les clés soient changées chaque jour ou toutes les 500 transactions, et plus souvent si possible. Noter qu'alors qu'il est acceptable d'utiliser la même clé temporaire pour plusieurs transactions, elle doit être signée chaque fois qu'elle est utilisée.

La génération de clés RSA est un processus qui consomme du temps. Dans de nombreux cas, un processus de faible priorité peut être alloué à la tâche de génération de clés.

Chaque fois qu'une nouvelle clé est achevée, la clé temporaire existante peut être remplacée par la nouvelle.

D.2 Génération de nombres aléatoires et germes

TLS exige un générateur de nombres pseudo aléatoires (PRNG, *pseudorandom number generator*) cryptographiquement sûr. Il faut porter une grande attention à la conception et à l'alimentation des PRNG. Les PRNG fondés sur des opérations de hachage sécurisé, principalement MD5 et/ou SHA, sont acceptables, mais ne peuvent pas fournir plus de sécurité que l'état déclaré par le générateur de nombres aléatoires. (Par exemple, les PRNG fondés sur MD5 fournissent habituellement 128 bits d'état.)

Pour estimer la quantité de matériel de germe qui est produite, ajouter le nombre de bits d'informations imprévisibles dans chaque octet du germe. Par exemple, les valeurs d'espacement de frappe des touches tirées du temporisateur à 18,2 Hz d'un micro ordinateur fournissent chacune un ou deux bits sûrs, même si la taille totale de la valeur du compteur est de 16 bits ou plus. Pour établir un germe de PRNG de 128 bits, on aurait donc besoin approximativement de 100 de ces valeurs de temporisateur.

Attention : Les fonctions de germe dans RSAREF et les versions de BSAFE antérieures à 3.0 sont indépendantes de l'ordre. Par exemple, si 1000 bits de germe sont fournies, une à la fois, dans 1000 appels distincts à la fonction de germe, le PRNG

va finir dans un état qui dépend seulement du nombre de bits 0 ou 1 de germe dans les données du germe (c'est-à-dire, il y a 1001 états finaux possibles). Les applications qui utilisent BSAFE ou RSAREF doivent être très attentives à assurer un germe approprié. Cela peut être accompli par l'accumulation de bits de germe dans une mémoire tampon et en les traitant tous en une fois, ou en procédant à l'incrémentation d'un compteur à chaque bit de germe traité ; l'une et l'autre méthode réintroduisent d'autres dépendances dans le traitement du germe.

D.3 Certificats et authentification

Les mises en œuvre sont responsables de la vérification de l'intégrité des certificats et devraient généralement accepter les messages de révocation de certificats. Les certificats devraient toujours être vérifiés pour s'assurer d'une signature appropriée par une autorité de certificat (CA, *Certificate Authority*) de confiance. Le choix et l'ajout de CA de confiance devrait être fait avec grand soin. Les usagers devraient être capables de voir les informations sur le certificat et la CA racine.

D.4 Suites de chiffrement

TLS prend en charge une gamme de tailles de clés et de niveaux de sécurité, y compris certains qui ne fournissent aucune sécurité ou qu'une sécurité minimale. Une mise en œuvre appropriée ne va probablement pas accepter beaucoup de suites de chiffrement. Par exemple, un chiffrement à 40 bits est facilement cassé, de sorte que les mises en œuvre qui exigent une forte sécurité ne devraient pas admettre de clés de 40 bits. De même, le Diffie-Hellman anonyme est fortement déconseillé parce qu'il ne permet pas d'empêcher les attaques par interposition. Les applications devraient aussi mettre en application des tailles de clé minimum et maximum. Par exemple, les chaînes de certificat qui contiennent des clés ou signatures RSA de 512 bits ne sont pas appropriées pour les applications de haute sécurité.

E. Rétro compatibilité avec SSL

Pour des raisons historiques et afin d'éviter une consommation inconsidérée de numéros d'accès réservés, les protocoles d'application qui sont sécurisés par TLS 1.0, SSL 3.0, et SSL 2.0 partagent fréquemment tous le même accès de connexion : par exemple, le protocole https (HTTP sécurisé par SSL ou TLS) utilise l'accès 443 sans considération du protocole de sécurité qu'il utilisé. Donc, certains mécanismes doivent être déterminés pour distinguer et négocier parmi les divers protocoles.

TLS version 1.0 et SSL 3.0 sont très similaires ; donc, il est facile d'accepter les deux. Les clients TLS qui souhaitent négocier avec les serveurs SSL 3.0 devraient envoyer les messages client hello en utilisant le format d'enregistrement SSL 3.0 et la structure client hello, envoyant {3, 1} pour le champ Version pour noter qu'ils acceptent TLS 1.0. Si le serveur n'accepte que SSL 3.0, il va répondre par un serveur hello SSL 3.0 ; si il accepte TLS, il répondra par un serveur hello TLS. La négociation se poursuit alors comme approprié pour le protocole négocié.

De même, un serveur TLS qui souhaite interopérer avec des clients SSL 3.0 devrait accepter les messages client hello SSL 3.0 et répondre par un serveur hello SSL 3.0 si un client hello SSL 3.0 est reçu avec un champ Version de {3, 0}, ce qui note que ce client n'accepte pas TLS.

Chaque fois qu'un client sait déjà quelle est la plus haute version de protocole connue d'un serveur (par exemple, lors de la reprise d'une session) il devrait initier la connexion dans ce protocole.

Les clients TLS 1.0 qui prennent en charge les serveurs SSL version 2.0 doivent envoyer des messages client hello en SSL version 2.0 [SSL2]. Les serveurs TLS devraient accepter l'un et l'autre format de client hello si ils souhaitent accepter les clients SSL 2.0 sur le même accès de connexion. Les seules variantes à la spécification de la version 2.0 sont la capacité à spécifier une version avec une valeur de trois et la prise en charge de plus de types de chiffrements dans la spécification de chiffrement (CipherSpec).

Attention : La capacité à envoyer des messages client hello de version 2.0 sera éliminée en temps voulu. Les mises en œuvre devraient porter tous leurs efforts à migrer sur la nouvelle version aussi vite que possible. La version 3.0 fournit de meilleurs mécanismes pour passer aux nouvelles versions.

Les spécifications de chiffrement suivantes sont des passerelles à partir de SSL version 2.0. Elles sont supposées utiliser RSA pour l'échange de clés et l'authentification.

```
V2CipherSpec TLS_RC4_128_WITH_MD5              = { 0x01,0x00,0x80 };
V2CipherSpec TLS_RC4_128_EXPORT40_WITH_MD5     = { 0x02,0x00,0x80 };
```

```
V2CipherSpec TLS_RC2_CBC_128_CBC_WITH_MD5            = { 0x03,0x00,0x80 };
V2CipherSpec TLS_RC2_CBC_128_CBC_EXPORT40_WITH_MD5   = { 0x04,0x00,0x80 };
V2CipherSpec TLS_IDEA_128_CBC_WITH_MD5               = { 0x05,0x00,0x80 };
V2CipherSpec TLS_DES_64_CBC_WITH_MD5                 = { 0x06,0x00,0x40 };
V2CipherSpec TLS_DES_192_EDE3_CBC_WITH_MD5           = { 0x07,0x00,0xC0 };
```

Les spécifications en TLS natif peuvent être incluses dans les messages client hello de version 2.0 en utilisant la syntaxe donnée ci-dessous. Tout élément V2CipherSpec qui a son premier octet égal à zéro sera ignoré par les serveurs de version 2.0. Les clients qui envoient l'une des spécifications de chiffrement de version 2 ci-dessus devraient aussi inclure l'équivalent TLS (voir au paragraphe A.5):

```
V2CipherSpec (voir le nom TLS) = { 0x00, CipherSuite };
```

E.1 Client hello version 2

Le message client hello de version 2.0 est présenté ci-dessous en utilisant le modèle de présentation du présent document. La vraie définition est toujours supposée être la spécification SSL version 2.0.

```
uint8 V2CipherSpec[3];

struct {
   uint8 msg_type;
   Version version;
   uint16 cipher_spec_length;
   uint16 session_id_length;
   uint16 challenge_length;
   V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
   opaque session_id[V2ClientHello.session_id_length];
   Random challenge;
} V2ClientHello;
```

msg_type
Ce champ, en conjonction avec le champ Version, identifie un messages client hello de version 2. La valeur devrait être un (1).

version
La version la plus élevée du protocole acceptée par le client (égale ProtocolVersion.version, voir au paragraphe A.1).

cipher_spec_length
Ce champ est la longueur totale du champ cipher_specs. Il ne peut pas être zéro et doit être un multiple de la longueur V2CipherSpec (3).

session_id_length
Ce champ doit avoir une valeur de zéro ou de 16. Si elle est zéro, le client crée une nouvelle session. Si elle est 16, le champ session_id va contenir les 16 octets de l'identification de session.

challenge_length
Longueur en octets du défi du client au serveur de s'authentifier. Cette valeur doit être 32.

cipher_specs
C'est une liste de toutes les spécifications de chiffrement que le client veut et est capable d'utiliser. Il doit y avoir au moins une spécification de chiffrement acceptable pour le serveur.

session_id
Si la longueur de ce champ n'est pas zéro, elle va contenir l'identification d'une session que le client souhaite reprendre.

challenge
Le défi du client au serveur pour que celui-ci s'identifie lui-même est (presque) d'une longueur aléatoire arbitraire. Le serveur TLS va justifier à droite les données du défi pour qu'elles deviennent les données de ClientHello.random (bourrées avec des zéros en tête, si nécessaire) comme spécifié dans la présente spécification. Si la longueur du défi est supérieure à 32 octets, seuls les 32 derniers octets sont utilisés. Il est légitime (mais non nécessaire) qu'un serveur de version 3 rejette un ClientHello de version 2 qui a moins de 16 octets de données de défi.

Note : Les demandes de reprise d'une session TLS devraient utiliser un client hello TLS.

E.2 Éviter la dégradation de version par interposition

Lorsque les clients TLS reviennent en mode de compatibilité avec la version 2.0, ils devraient utiliser le format spécial de bloc PKCS n° 1. Cela est fait afin que les serveurs TLS rejettent les sessions de version 2.0 avec des clients à capacité TLS.

Lorsque les clients TLS sont en mode de compatibilité avec la version 2.0, ils règlent les huit octets aléatoires de droite (de moindre poids) du bourrage du PKCS (non inclus le nul terminal du bourrage) pour le chiffrement RSA du champ ENCRYPTED-KEY-DATA de la clé maître de client à 0x03 (les autres octets du bourrage sont aléatoires). Après déchiffrement du champ ENCRYPTED-KEY-DATA, les serveurs qui prennent en charge TLS devraient produire une erreur si ces huit octets de bourrage sont 0x03. Les serveurs de version 2.0 qui reçoivent des blocs bourrés de cette manière vont procéder normalement.

F. Analyse de la sécurité

Le protocole TLS est conçu pour établir une connexion sûre entre un client et un serveur qui communiquent sur un canal non sécurisé. Le présent document fait plusieurs hypothèses traditionnelles, y compris que les attaquants ont des ressources de calcul substantielles et ne peuvent pas obtenir d'informations secrètes de sources en dehors du protocole. Les attaquants sont supposés avoir la capacité de capturer, modifier, supprimer, répéter, et autrement altérer les messages envoyés sur le canal de communication. La présente section met en lumière comment TLS a été conçu pour résister à diverses attaques.

F.1 Protocole de prise de contact

Le protocole de prise de contact est chargé de choisir une spécification de chiffrement et de générer un secret maître, qui ensemble comportent les principaux paramètres cryptographiques associés à une session sécurisée. Le protocole de prise de contact peut aussi facultativement authentifier les parties qui ont des certificats signés par une autorité de certificat de confiance.

F.1.1 Authentification et échange de clés

TLS prend en charge trois modes d'authentification : l'authentification des deux parties, l'authentification du serveur avec un client non authentifié, et l'anonymat total. Chaque fois que le serveur est authentifié, le canal est sécurisé contre les attaques par interposition, mais les sessions complètement anonymes sont par nature vulnérables à de telles attaques. Les serveurs anonymes ne peuvent pas authentifier les clients. Si le serveur est authentifié, ses messages de certificat doivent fournir une chaîne de certificats valide conduisant à une autorité de certificat acceptable. De même, les clients authentifiés doivent produire un certificat acceptable au serveur. Chaque partie est responsable de la vérification de la validité du certificat de l'autre, et qu'il n'est ni expiré ni révoqué.

Le but général du processus d'échange de clés est de créer un secret pré-maître connu des parties en communication et pas des attaquants. Le secret pré-maître sera utilisé pour générer le secret maître (voir le paragraphe 8.1). Le secret maître est exigé pour générer les messages Vérification de certificat et Terminé, les clés de chiffrement et les MAC secrets (voir les paragraphes 7.4.8, 7.4.9 et 6.3). Par l'envoi d'un message Terminé correct, les parties prouvent ainsi qu'elles connaissent le secret pré-maître correct.

F.1.1.1 Échange de clés anonyme

Les sessions complètement anonymes peuvent être établies en utilisant RSA ou Diffie-Hellman pour l'échange de clés. Avec RSA anonyme, le client chiffre un secret pré-maître avec la clé publique non certifiée du serveur, extraite du message d'échange de clé du serveur. Le résultat est envoyé dans un message d'échange de clé de client. Comme les espions ne connaissent pas la clé privée du serveur, il leur sera impossible de décoder le secret pré-maître. (Noter qu'aucune suite de chiffrement RSA anonyme n'est définie dans le présent document).

Avec Diffie-Hellman, les paramètre publics du serveur sont contenus dans le message d'échange de clé du serveur et ceux du client sont envoyés dans le message d'échange de clé du client. Les espions qui ne connaissent pas les valeurs privées ne devraient pas être capables de trouver le résultat Diffie-Hellman (c'est-à-dire le secret pré-maître).

Attention : Les connexions complètement anonymes ne fournissent de protection que contre l'espionnage passif. Sauf si un canal indépendant à l'épreuve des altérations est utilisé pour vérifier que les messages Terminé n'ont pas été

remplacés par un attaquant, l'authentification du serveur est requise dans les environnements où des attaques actives d'interposition sont à craindre.

F.1.1.2 Échange de clés RSA et authentification

Avec RSA, l'échange de clé et l'authentification du serveur sont combinées. La clé publique peut être contenue dans le certificat du serveur ou peut être une clé RSA temporaire envoyée dans un message d'échange de clé de serveur. Lorsque des clés RSA temporaires sont utilisées, elles sont signées par le certificat RSA ou DSS du serveur. La signature comporte l'aléa ClientHello.actuel, de sorte que les vieilles signatures et les clés temporaires ne peuvent pas être répétées. Les serveurs peuvent utiliser une seule clé RSA temporaire pour plusieurs sessions de négociation.

Note : L'option de clé RSA temporaire est utile si les serveurs ont besoin de certificats de grande taille mais elle doit se plier aux limites de taille imposées par les gouvernements sur les clés utilisées dans les échanges de clés.

Après vérification du certificat du serveur, le client chiffre un secret pré-maître avec la clé publique du serveur. En réussissant à décoder le secret pré-maître et en produisant un message Terminé correct, le serveur démontre qu'il connaît la clé privée qui correspond au certificat du serveur.

Lorsque RSA est utilisé pour l'échange de clé, les clients sont authentifiés en utilisant le message Vérification de certificat (voir au paragraphe 7.4.8). Le client signe une valeur déduite du secret maître et de tous les messages de prise de contact précédents. Ces messages de prise de contact incluent le certificat du serveur, qui lie la signature au serveur, et ServerHello.random, qui lie la signature au processus de prise de contact en cours.

F.1.1.3 Échange de clés Diffie-Hellman avec authentification

Lorsque l'échange de clés Diffie-Hellman est utilisé, le serveur peut fournir un certificat contenant des paramètres Diffie-Hellman fixés ou il peut utiliser le message d'échange de clé du serveur pour envoyer un ensemble de paramètres Diffie-Hellman temporaires signés avec un certificat DSS ou RSA. Les paramètres temporaires sont hachés avec les valeurs du hello.random avant la signature afin de s'assurer que des attaquants ne répètent pas de vieux paramètres. Dans l'un ou l'autre cas, le client peut vérifier le certificat ou la signature pour s'assurer que les paramètres appartiennent au serveur.

Si le client a un certificat qui contient des paramètres Diffie-Hellman fixes, son certificat contient les informations requises pour terminer l'échange de clés. Noter que dans ce cas, le client et le serveur vont générer le même résultat Diffie-Hellman (c'est-à-dire, le secret pré-maître) chaque fois qu'ils communiquent. Pour empêcher que le secret pré-maître ne reste en mémoire plus longtemps que nécessaire, il devrait être converti aussitôt que possible en secret maître. Les paramètres Diffie-Hellman du client doivent être compatibles avec ceux fournis par le serveur pour l'échange de clés pour fonctionner.

Si le client a un certificat DSS ou RSA standard ou si il n'est pas authentifié, il envoie un ensemble de paramètres temporaires au serveur dans le message client d'échange de clés, puis utilise facultativement un message Vérification de certificat pour s'authentifier.

F.1.2 Attaque de dégradation de version

Comme TLS comporte des améliorations substantielles par rapport à SSL version 2.0, les attaquants peuvent essayer de faire revenir les clients et serveurs à capacité TLS à la version 2.0. Cette attaque ne peut survenir que si (et seulement si) deux parties à capacité TLS utilisent une prise de contact SSL 2.0.

Bien que la solution d'utiliser un bourrage non aléatoire de message de type 2 de bloc PKCS n° 1 soit inélégante, elle fournit un moyen raisonnablement sûr pour que les serveur de version 3.0 détectent l'attaque. Cette solution n'est pas sûre contre les attaquants qui peuvent rechercher la clé à force brute et substituer un nouveau message ENCRYPTED-KEY-DATA contenant la même clé (mais avec un bourrage normal) avant que n'arrive à expiration le seuil spécifié par l'application. Les parties concernées par des attaques à cette échelle ne devraient de toutes façons pas utiliser de clés de chiffrement de 40 bits. Altérer le bourrage des huit octets de moindre poids du bourrage du PKCS n'a pas d'impact sur la sécurité pour la taille des hachages signés et les longueurs de clés RSA utilisées dans le protocole, car ceci est essentiellement équivalent à augmenter la taille du bloc d'entrées de huit octets.

F.1.3 Détection des attaques contre le protocole de prise de contact

Un attaquant peut essayer d'influencer l'échange de prise de contact pour faire que les parties choisissent des algorithmes de chiffrement différents de ce qu'elles auraient normalement choisi. Comme de nombreuses mises en œuvre vont prendre en charge le chiffrement exportable à 40 bits et que certaines peuvent même accepter le chiffrement nul ou des algorithmes de

MAC, cette attaque pose un problème particulièrement préoccupant.

Pour cette attaque, l'agresseur doit changer activement un ou plusieurs des messages de prise de contact. Si cela arrive, le client et le serveur vont calculer des valeurs différentes pour les hachages de message de prise de contact. Il en résultera que les parties ne vont pas accepter les messages Terminé de l'autre partie. Sans le secret maître, l'agresseur ne peut pas réparer les messages Terminé, de sorte que l'attaque sera découverte.

F.1.4 Reprise de sessions

Lorsque une connexion est établie par la reprise d'une session, de nouvelles valeurs de ClientHello.random et de ServerHello.random sont hachées avec le secret maître de la session. Pourvu que le secret maître n'ait pas été compromis et que les opérations de hachage sécurisées utilisées pour produire les clés de chiffrement et les MAC secrets soient sûres, la connexion devrait être sûre et effectivement indépendante des connexions antérieures. Les attaquants ne peuvent pas utiliser des clés de chiffrement connues ou des MAC secrets pour compromettre le secret maître sans casser les opérations de hachage sécurisé (qui utilisent tous deux SHA et MD5).

Les sessions ne peuvent être reprisses qui si le client et le serveur en sont tous deux d'accord. Si l'une des parties soupçonne que la session pourrait avoir été compromise, ou que les certificats pourraient être arrivés à expiration ou avoir été révoqués, elle devrait forcer une prise de contact complète. Une limite supérieure de 24 heures est suggérée pour la durée de vie des identifiants de session, car un attaquant qui obtient un secret maître peut être capable de se faire passer pour la partie compromise jusqu'à ce que l'identifiant de session correspondant soit retiré. Les applications qui pourraient fonctionner dans des environnements relativement peu sûrs ne devraient pas inscrire d'identifiants de session dans des mémoires permanentes.

F.1.5 MD5 et SHA

TLS utilise les fonctions de hachage de façon très prudente. Lorsque c'est possible, MD5 et SHA sont utilisés en tandem pour assurer que des fautes non catastrophiques dans un algorithme ne vont pas casser le protocole global.

F.2 Protection des données d'application

Le secret maître est haché avec le ClientHello.random et le ServerHello.random pour produire des clés univoques de chiffrement des données et des MAC secrets pour chaque connexion.

Les données sortantes sont protégées avec un MAC avant leur transmission. Pour empêcher la répétition de message ou les attaques de modification, le MAC est calculé à partir du secret MAC, du numéro de séquence, de la longueur du message, du contenu du message, et de deux chaînes de caractères fixes. Le champ Type de message est nécessaire pour assurer que des messages destinés à un client de couche Enregistrement TLS ne sont pas redirigés sur un autre. Le numéro de séquence assure que les tentatives de supprimer ou réordonner les messages seront détectées. Comme les numéros de séquence sont longs de 64 bits, ils ne devraient jamais déborder. Les messages provenant d'une des parties ne peuvent pas être insérés dans le résultat de l'autre, car ils utilisent des secrets MAC indépendants. De même, les clés d'écriture serveur et d'écriture client sont indépendantes de sorte que les clés de chiffrement de flux ne sont utilisées qu'une seule fois.

Si un attaquant casse une clé de chiffrement, tous les messages chiffrés avec elle pourront être lus. De même, la compromission d'une clé de MAC peut rendre possible des attaques de modification de message. Parce que les MAC sont aussi chiffrés, les attaques par altération de message exigent généralement de casser aussi l'algorithme de chiffrement en plus du MAC.

Note : Les secrets de MAC peuvent être plus gros que les clés de chiffrement, de sorte que les messages peuvent rester résistants aux altérations même si les clés de chiffrement sont cassées

F.3 Notes finales

Pour que TLS soit capable de fournir une connexion sûre, les systèmes client et serveur, les clés, et les applications doivent être sûres. De plus, la mise en œuvre doit être libre d'erreurs quant à la sécurité.

Le système n'est pas plus fort que le plus faible algorithme d'échange de clés et d'authentification pris en charge, et seules devraient être utilisées des fonctions cryptographiques dignes de confiance, des clés de chiffrement de 40 bits en brut, et des serveurs anonymes ne devraient être utilisés qu'avec une grande prudence. Les mises en œuvre et les usagers doivent être prudents quand ils décident quels certificats et autorités de certificats sont acceptables ; une autorité de certificat malhonnête peut causer des dommage terrifiants.

G. Déclaration de brevets

Certains des algorithmes cryptographiques proposés à l'utilisation dans le présent protocole sont soumis à des déclarations de brevets. De plus Netscape Communications Corporation a une revendication de brevet sur les travaux couvrant la couche de connexion sécurisée (SSL, *Secure Sockets Layer*) sur lesquels se fonde la présente norme. Le processus de normalisation de l'Internet a défini dans la RFC2026 qu'une déclaration peut être obtenue d'un détenteur de brevet indiquant qu'une licence sera rendue disponible aux demandeurs sous des termes et conditions raisonnables.

Le Massachusetts Institute of Technology a accordé à RSA Data Security, Inc., des droits exclusifs de sous licence sur le brevet suivant, produit aux États-Unis d'Amérique :
Cryptographic Communications System and Method ("RSA"), n° 4,405,829

Netscape Communications Corporation a produit le brevet suivant aux États-Unis d'Amérique :
Secure Socket Layer Application Program Apparatus And Method ("SSL"), n° 5,657,390

Netscape Communications a produit la déclaration suivante :

Droits de propriété intellectuelle

Secure Sockets Layer (SSL, *couche de prises sécurisées*)
Le bureau des brevets et marques commerciales des États-Unis d'Amérique ("PTO") a récemment accordé le brevet U.S. n° 5 657 390 ("brevet SSL") à Netscape pour les inventions décrites sous le nom de Secure Sockets Layers ("SSL"). L'IETF projette actuellement d'adopter SSL comme protocole de transport avec des caractéristiques de sécurité. Netscape encourage l'adoption et l'utilisation sans redevance du protocole SSL sous les termes et conditions suivants :

* Si vous avez déjà aujourd'hui une licence ref. SSL valide qui comporte le code source de Netscape, une licence additionnelle au brevet SSL n'est pas nécessaire.

* Si vous n'avez pas une licence Ref. SSL, vous pouvez avoir une licence sans redevance pour construire des mises en œuvre couvertes par les revendications du brevet SSL ou la spécification TLS de l'IETF, à condition que vous ne souteniez pas de revendication de brevet contre Netscape ou d'autres compagnies pour la mise en œuvre de SSL ou de la recommandation TLS de l'IETF.

Que sont les "revendications de brevets" :
Les revendications de brevet sont des revendications qui, dans un brevet produit à l'étranger ou sur le territoire national :
1) doivent être enfreintes afin de mettre en œuvre les méthodes ou construire des produits conformément à la spécification TLS de l'IETF ; ou
2) des revendications de brevet qui exigent que des éléments des revendications du brevet SSL et/ou leurs équivalents soient enfreints.

La Société Internet, le bureau de l'Architecture de l'Internet, le groupe de pilotage de l'ingénierie de l'Internet et la Corporation pour les initiatives de recherche nationales ne prennent pas position sur la validité ou la portée des brevets et des applications de brevets, ni sur l'adéquation des termes de l'assurance. La société Internet et les autres groupes mentionnés ci-dessus n'ont fait aucune détermination d'autres droits de propriété intellectuelle qui pourraient s'appliquer à la mise en œuvre de cette norme. Toute autre considération sur ces sujets est de la propre responsabilité de l'utilisateur.

Considérations pour la sécurité

Les questions de sécurité sont exposées tout au long du présent mémoire.

Références

[3DES] W. Tuchman, "Hellman Presents No Shortcut Solutions To DES," IEEE Spectrum, v. 16, n. 7 juillet 1979, pages 40-41.

[BLEI] Bleichenbacher D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1" dans Advances in Cryptology -- CRYPTO'98, LNCS vol. 1462, pages 1--12, 1998.

[DES] ANSI X3.106, "American National Standard for Information Systems-Data Link Encryption," American National

Standards Institute, 1983.

[DH1] W. Diffie and M. E. Hellman, "New Directions in Cryptography," IEEE Transactions on Information Theory, V. IT-22, n° 6, juin 1977, pages 74-84.

[DSS] NIST FIPS PUB 186, "Digital Signature Standard," National Institute of Standards and Technology, U.S. Department of Commerce, 18 mai 1994.

[FTP] J. Postel et J. Reynolds, "Protocole de transfert de fichiers (FTP)", RFC0959, STD 9, octobre 1985.

[HTTP] T. Berners-Lee, R. Fielding, H. Frystyk, "Protocole de transfert Hypertext -- HTTP/1.0", RFC1945, mai 1996. *(Information)*

[HMAC] H. Krawczyk, M. Bellare et R. Canetti, "HMAC : Hachage de clés pour l'authentification de message", RFC2104, février 1997.

[IDEA] X. Lai, "On the Design and Security of Block Ciphers," ETH Series in Information Processing, v. 1, Konstanz: Hartung-Gorre Verlag, 1992.

[MD2] B. Kaliski, "Algorithme de résumé de message MD2", RFC1319, avril 1992. *(Information)*

[MD5] R. Rivest, "Algorithme de résumé de message MD5", RFC1321, avril 1992. *(Information)*

[PKCS1] RSA Laboratories, "PKCS #1: RSA Encryption Standard," version 1.5, novembre 1993.

[PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard," version 1.5, novembre 1993.

[PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard," version 1.5, novembre 1993.

[PKIX] R. Housley, W. Ford, W. Polk et D. Solo, "Profil de certificat d'infrastructure de clé publique X.509 et de CRL pour l'Internet", RFC2459, janvier 1999. *(Obsolète, voir la RFC3280) (P.S.)*

[RC2] R. Rivest, "Description de l'algorithme de chiffrement RC2(r)", RFC2268, mars 1998. *(Information)*

[RC4] Thayer, R. and K. Kaukonen, "A Stream Cipher Encryption Algorithm", Travail en cours.

[RSA] R. Rivest, A. Shamir, et L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications de l' ACM, v. 21, n. 2, février 1978, pages 120 à 126.

[RSADSI] Contact pour RSA Data Security, Inc., tél : + (1) 415-595-8782

[SCH] B. Schneier. "Applied Cryptography: Protocols, Algorithms, and Source Code in C", publié par John Wiley & Sons, Inc. 1994.

[SHA] NIST FIPS PUB 180-1, "Secure Hash Standard," National Institute of Standards and Technology, U.S. Department of Commerce, Travail en cours, 31 mai 1994.

[SSL2] Hickman, Kipp, "The SSL Protocol", Netscape Communications Corp., 9 février 1995.

[SSL3] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., 18 novembre 1996.

[TCP] J. Postel (éd.), "Protocole de commande de transmission – Spécification du protocole du programme Internet DARPA", RFC0793, (STD 7), septembre 1981.

[TEL] J. Postel et J. Reynolds," Spécification du protocole TELNET", RFC0854, mai 1983.

[TEL] J. Postel et J. Reynolds,"Spécifications des options TELNET", RFC0855, mai 1983.

[X509] CCITT. Recommandation X.509 "L'annuaire – Cadre d'authentification ". 1988.

[XDR] R. Srinivasan, "XDR : norme de représentation des données externes", RFC1832, août 1995. *(Obsolète, voir* RFC4506) *(D.S.)*

Crédits

Win Treese
Open Market
mél : treese@openmarket.com

Éditeurs

Christopher Allen
Certicom
mél : callen@certicom.com

Tim Dierks
Certicom
mél : tdierks@certicom.com

Adresse des auteurs

Tim Dierks
Certicom
mél : tdierks@certicom.com

Philip L. Karlton
Netscape Communications

Alan O. Freier
Netscape Communications
mél : freier@netscape.com

Paul C. Kocher
Independent Consultant
mél : pck@netcom.com

Autres contributeurs

Martin Abadi
Digital Equipment Corporation
mél : ma@pa.dec.com

Robert Relyea
Netscape Communications
mél : relyea@netscape.com

Ran Canetti
IBM Watson Research Center
mél : canetti@watson.ibm.com

Jim Roskind
Netscape Communications
mél : jar@netscape.com

Taher Elgamal
Securify
mél : elgamal@securify.com

Micheal J. Sabin, Ph. D.
Consulting Engineer
mél : msabin@netcom.com

Anil R. Gangolli
Structured Arts Computing Corp.
mél : gangolli@structuredarts.com

Dan Simon
Microsoft
mél : dansimon@microsoft.com

Kipp E.B. Hickman
Netscape Communications
mél : kipp@netscape.com

Tom Weinstein
Netscape Communications
mél : tomw@netscape.com

Hugo Krawczyk
IBM Watson Research Center
mél : hugo@watson.ibm.com

Commentaires

La liste de discussion du groupe de travail TLS de l'IETF se trouve à l'adresse ietf-tls@lists.consensus.com. Les informations sur le groupe et sur la façon de s'abonner à la liste sont à http://lists.consensus.com/.

Les archives de la liste se trouvent à http://www.imc.org/ietf-tls/mail-archive/.

Déclaration complète de droits de reproduction

Copyright (C) The Internet Society (1999). Tous droits réservés.

Ce document et les traductions de celui-ci peuvent être copiés et diffusés, et les travaux dérivés qui commentent ou expliquent autrement ou aident à sa mise en œuvre peuvent être préparés, copiés, publiés et distribués, partiellement ou en

totalité, sans restriction d'aucune sorte, à condition que l'avis de droits de reproduction ci-dessus et ce paragraphe soient inclus sur toutes ces copies et œuvres dérivées. Toutefois, ce document lui-même ne peut être modifié en aucune façon, par exemple en supprimant le droit d'auteur ou les références à l'Internet Society ou d'autres organisations Internet, sauf si c'est nécessaire à l'élaboration des normes Internet, auquel cas les procédures pour les droits de reproduction définies dans les processus des normes de l'Internet doivent être suivies, ou si nécessaire pour le traduire dans des langues autres que l'anglais.

Les permissions limitées accordées ci-dessus sont perpétuelles et ne seront pas révoquées par la Société Internet ou ses successeurs ou ayants droit.

Ce document et les renseignements qu'il contient sont fournis "TELS QUELS" et l'INTERNET SOCIETY et l'INTERNET ENGINEERING TASK FORCE déclinent toute garantie, expresse ou implicite, y compris mais sans s'y limiter, toute garantie que l'utilisation de l'information ici présente n'enfreindra aucun droit ou aucune garantie implicite de commercialisation ou d'adaptation à un objet particulier.