Groupe de travail Réseau
Request for Comments : 5246
RFC rendues obsolètes : 3268, 4346, 4366
RFC mise à jour : 4492
Catégorie : En cours de normalisation

T. Dierks,
E. Rescorla, RTFM, Inc.
août 2008
Traduction Claude Brière de L'Isle
décembre 2008

Protocole Sécurité de la couche Transport (TLS) version 1.2

Statut du présent mémoire

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

Résumé

Le présent document spécifie la version 1.2 du protocole Sécurité de la couche Transport (TLS). Le protocole TLS fournit la sécurité des communications sur l'Internet. Le protocole permet aux applications client/serveur de communiquer d'une façon conçue pour empêcher l'espionnage, l'altération ou la falsification du message.

Table des matières

1. Introduction
   1.1 Exigences de terminologie
   1.2 Différences majeures avec TLS 1.1
2. Objectifs
3. Objectifs de ce document
4. Langage de présentation
   4.1 Taille de bloc de base
   4.2 Divers
   4.3 Vecteurs
   4.4 Nombres
   4.5 Énumérations
   4.6 Types construits
      4.6.1 Variantes
   4.7 Attributs cryptographiques
   4.8 Constantes
5. HMAC et la fonction pseudo aléatoire
6. Protocole d'enregistrement TLS
   6.1 États de connexion
   6.2 Couche d'enregistrement
      6.2.1 Fragmentation
      6.2.2 Compression et décompression d'enregistrement
      6.2.3 Protection de charge utile d'enregistrement
   6.3 Calcul des clés
7. Protocoles TLS de prise de contact
   7.1 Protocole de changement de spécification de chiffrement
   7.2 Protocole d'alerte
      7.2.1 Alertes de clôture
      7.2.2 Alertes d'erreur
   7.3 Généralités sur le protocole de prise de contact
   7.4 Protocole de prise de contact
      7.4.1. Messages Hello
      7.4.2 Certificat de serveur
      7.4.3 Message d'échange de clés du serveur
      7.4.4 Demande de certificat
      7.4.5 Hello Done du serveur
      7.4.6 Certificat du client
      7.4.7 Message d'échange de clés de client
      7.4.8 Vérification de certificat
      7.4.9 Terminé
8. Calculs cryptographiques

8.1 Calcul du secret maître
8.1.1 RSA
8.1.2 Diffie-Hellman
9. Suites de chiffrement obligatoires
10. Protocole des données d'application
11. Considérations sur la sécurité
12. Considérations relatives à l'IANA
Appendice A. Structures des données et valeurs constantes du protocole
A.1 Couche d'enregistrement
A.2 Message de changement des spécifications de chiffrement
A.3 Messages d'alerte
A.4 Protocole de prise de contact
A.4.1 Messages Hello
A.4.2 Messages d'authentification de serveur et d'échange de clés
A.4.3 Messages d'authentification de client et d'échange de clés
A.4.4 Message de finalisation de prise de contact
A.5 Suite de chiffrement
A.6 Les paramètres de sécurité
A.7 Changements par rapport à la RFC 4492
Appendice B. Glossaire
Appendice C. Définitions des suites de chiffrement
Appendice D. Notes de mise en œuvre
D.1 Génération et germination de nombre aléatoire
D.2 Certificats et authentification
D.3 Suites de chiffrement
D.4 Pièges à éviter
Appendice E Rétro-compatibilité
E.1 Compatibilité avec TLS 1.0/1.1 et SSL 3.0
E.2 Compatibilité avec SSL 2.0
Appendice F. Analyse de la sécurité
F.1 Protocole de prise de contact
F.1.1 Authentification et échange de clé
F.1.2 Attaques de régression de version
F.1.3 Détection des attaques contre le protocole de prise de contact
F.1.4 Reprise des sessions
F.2 Protection des données d'application
F.3 IV explicites
F.4 Sécurité des modes de chiffrement composites
F.5 Déni de service
F.6 Notes finales

1. Introduction

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

- La connexion est privée. La cryptographie symétrique est utilisée pour le chiffrement des données (par exemple, AES [AES], RC4 [SCH], etc.). Les clés pour ce chiffrement symétrique sont générées de façon univoque pour chaque connexion et sont fondées 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 code d'authentification de message (MAC, *Message Authentication Code*) chiffré. On utilise des fonctions de hachage sécurisées (par exemple, SHA-1, etc.) pour le calcul du MAC. Le protocole d'enregistrement peut fonctionner sans MAC, mais il est généralement seulement utilisé 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 l'un l'autre et de

négocier un algorithme de chiffrement et des clés cryptographiques avant que le protocole d'application ne transmette ou ne reçoive son premier octet de données. Le protocole de prise de contact TLS fournit la sécurité de la connexion qui a trois propriétés de base :

- L'identité de l'homologue peut être authentifiée en utilisant un chiffrement asymétrique, ou par clé publique (par exemple, RSA [RSA], DSA [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 est inaccessible aux espions, et pour toute connexion authentifiée, le secret ne peut pas être obtenu, même par un attaquant qui peut s'interposer au 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 des avantages de TLS est que c'est un protocole indépendant de l'application. Les protocoles de niveau supérieur peuvent se placer de façon transparente sur le dessus du protocole TLS. La norme TLS ne spécifie cependant pas comment les protocoles ajoutent de la sécurité à TLS ; les décisions sur la façon d'initialiser la prise de contact TLS et comment interpréter les certificats d'authentification échangés sont laissées au jugement des concepteurs et de ceux qui mettent en œuvre les protocoles qui se placent par dessus TLS.

1.1 Exigences de terminologie

Les mots clés "DOIT", "NE DOIT PAS", "EXIGE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS", "RECOMMANDE", "PEUT", et "FACULTATIF" dans le présent document sont à interpréter comme décrit dans la RFC 2119 [REQ].

1.2 Différences majeures avec TLS 1.1

Le présent document est une révision du protocole TLS 1.1 [TLS1.1] qui comporte une souplesse améliorée, en particulier pour la négociation des algorithmes cryptographiques. Les changements majeurs sont :

- La combinaison MD5/SHA-1 dans la fonction pseudo aléatoire (PRF, *pseudorandom function*) a été remplacée par des PRF spécifiées par suite de chiffrement. Toutes les suites de chiffrement du présent document utilisent P_SHA256.

- La combinaison MD5/SHA-1 dans l'élément de signature numérique a été remplacée par un seul hachage. Les éléments signés comportent maintenant un champ qui spécifie explicitement l'algorithme de hachage utilisé.

- Un nettoyage substantiel de la capacité du client et du serveur à spécifier quels algorithmes de hachage et de signature ils acceptent. Noter que cela relâche aussi certaines des contraintes sur les algorithmes de signature et de hachage des précédentes versions de TLS.

- Ajout de la prise en charge du chiffrement authentifié avec des modes de données supplémentaires.

- Les définitions d'extensions TLS et de suites de chiffrement AES ont été fusionnées à partir des [TLSEXT] et [TLSAES] externes.

- Une vérification plus serrée des numéros de version EncryptedPreMasterSecret.

- Resserrement d'un certain nombre d'exigences.

- La longueur de Verify_data dépend maintenant de la suite de chiffrement (la valeur par défaut est toujours 12).

- Nettoyage de la description des défenses contre l'attaque Bleichenbacher/Klima.

- Les alertes DOIVENT maintenant être envoyées dans de nombreux cas.

- Après une certificate_request, si aucun certificat n'est disponible, les clients DOIVENT maintenant envoyer une liste de certificat vide.

- TLS_RSA_WITH_AES_128_CBC_SHA est maintenant la suite de chiffrement de mise en œuvre obligatoire.

- Ajout des suites de chiffrement HMAC-SHA256.
- Retrait des suites de chiffrement IDEA et DES. Elles sont maintenant déconseillées et seront exposées dans un document à part.
- La prise en charge du hello rétro compatible SSLv2 est maintenant un PEUT, et non un DEVRAIT, et son envoi est un NE DEVRAIT PAS. Sa prise en charge deviendra probablement un NE DEVRAIT PAS à l'avenir.
- Ajout d'un échec limité au langage de présentation pour permettre aux cas de dommages multiples d'avoir la même codification.
- Ajout d'un paragraphe de mise en œuvre des pièges à éviter
- Les éclaircissements usuels et les corrections rédactionnelles.

2. Objectifs

Les objectifs du protocole TLS sont, dans l'ordre des priorités, les suivants :

1. Sécurité cryptographique: 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 qui réussissent pleinement à échanger des paramètres cryptographique sans connaître leurs codes respectifs.
3. Extensibilité : TLS cherche à fournir un cadre dans lequel puissent être incorporées de nouvelles clés publiques et de nouvelles méthodes de chiffrement en vrac en tant que de besoin. Cela réalisera aussi deux sous objectifs : prévenir le besoin de créer un nouveau protocole (avec le risque d'introduction de possibles nouvelles faiblesses) et éviter le besoin de mettre en œuvre une bibliothèque de sécurité entièrement nouvelle.
4. Efficacité relative : les opérations cryptographiques tendent à être très consommatrices de CPU, en particulier les opérations de clé publique. Pour cette raison, le protocole TLS a incorporé un schéma facultatif de mise en mémoire cache de session pour réduire le nombre de connexions qui doivent être établies à partir de rien. De plus, on a veillé à réduire l'activité réseau.

3. Objectifs de ce document

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

Le présent document n'est pas destiné à fournir les détails des définitions de service ni des définitions d'interface, bien qu'il couvre effectivement des domaines choisis de la politique car ils sont nécessaires au maintien d'une sécurité solide.

4. Langage de présentation

Le présent document traite du formatage des données dans une représentation externe. La syntaxe de présentation très basique et parfois définie de façon un peu désinvolte suivante sera utilisée. La structure de la syntaxe provient de plusieurs sources. Bien qu'elle ressemble au langage de programmation "C" dans sa syntaxe et à XDR [XDR] à la fois dans sa syntaxe et son propos, il serait hasardeux d'en tirer trop de parallèles. L'intention de ce langage de présentation est seulement d'exposer TLS ; il n'a pas d'application générale au-delà de cet objectif particulier.

4.1 Taille de bloc de base

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

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

Cet ordre des octets pour les valeurs multi octets est l'ordre ordinaire des octets du réseau ou format gros-boutien.

4.2 Divers

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

Les composants facultatifs sont notés en les plaçant entre des crochets doubles "[[ ]]".

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

4.3 Vecteurs

Un vecteur (matrice unidimensionnelle) 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 son utilisation. Dans l'un ou l'autre cas, la longueur déclare le nombre d'octets, non le nombre des éléments dans le vecteur. La syntaxe pour spécifier un nouveau type, T', qui est un vecteur de longueur fixée de type T est T T'[n] ;

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

Dans l'exemple suivant, Datum est défini comme étant 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];      /* 3 vecteurs consécutifs de 3 octets */
```

Les vecteurs de longueur variable sont définis en spécifiant une sous gamme de longueurs légales, incluses, en utilisant la notation <plancher..plafond>. Lorsqu'elles sont codées, la longueur réelle précède le contenu du vecteur dans le flux d'octets. La longueur sera sous la forme d'un nombre consommant autant d'octets que nécessaire pour contenir la longueur maximum (plafond) spécifiée du vecteur. Un vecteur de longueur variable avec un champ de longueur réelle est appelé un vecteur vide.

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

Dans l'exemple suivant, est obligatoire un vecteur qui doit contenir entre 300 et 400 octets de type opaque. Il ne peut jamais être vide. Le champ de longueur réelle consomme deux octets, un uint16, qui est 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 inclure 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 2 octets, il ne peut pas être vide */
      uint16 longer<0..800>;   /* zéro à 400 entiers non signés de 16 bits */
```

4.4 Nombres

Le type de données numériques de base est un octet non signé (uint8). Tous les plus grands types de données numériques sont formés à partir d'une série d'octets de longueur fixe concaténés comme décrit au paragraphe 4.1 et 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 spécification, sont mémorisées dans l'ordre des octets du réseau (gros boutien) ; le uint32 représenté par les octets hex 01 02 03 04 est équivalent à la valeur décimale 16 909 060.

Noter que dans certains cas (par exemple, paramètres DH) il est nécessaire de représenter des entiers comme des vecteurs opaques. Dans de tels cas, ils sont représentés comme des entiers non signés (c'est-à-dire que les octets de zéros de gauche ne sont pas exigés même si le bit de plus fort poids est mis à 1).

4.5 Énumérations

Un type de données clairsemées additionnel disponible est appelé enum. Un champ de type enum peut supposer seulement les valeurs déclarées dans la définition. Chaque définition est un type différent. Seuls les énumérés du même type peuvent être alloués ou comparés. Chaque élément d'une énumération doit avoir une valeur allouée, comme montré dans l'exemple suivant. Comme les éléments de l'énumération ne sont pas ordonnés, il peut leur être allouée toute valeur unique, dans n'importe quel ordre.

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

Une énumération occupe autant d'espace dans le flux des octets que le ferait sa valeur ordinale définie maximale. La définition suivante amènerait un octet à être utilisé pour porter des champs de type Couleur.

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

On peut facultativement spécifier une valeur sans son étiquette associée pour forcer la définition de largeur sans définir un élément superflu.

Dans l'exemple suivant, Goût va consommer deux octets dans le flux des données mais peut seulement prendre les valeurs 1, 2, ou 4.

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

Les noms des éléments d'une énumération ont une portée limitée au 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'allocation est bien spécifiée.

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

Pour les énumérations qui ne sont jamais converties en représentation externe, l'information numérique peut être omise.

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

4.6 Types construits

Les types de structures peuvent être construits à partir de types de primitives si cela convient. Chaque spécification déclare un nouveau type, unique. La syntaxe pour la définition est très semblable à celle de 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 semblable à 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

Des structures définies peuvent avoir des variantes sur la base de connaissances disponibles dans l'environnement. Le sélecteur doit être un type énuméré qui spécifie les variantes possibles que définit la structure. Il doit y avoir une branche de cas pour chaque élément de l'énumération déclarée dans la sélection. Les branches de cas ont une portée limitée : si deux branches de cas se suivent immédiatement sans aucun champ entre elles, elles contiennent alors tous deux les mêmes champs. Et donc, dans l'exemple ci-dessous, "orange" et "banane" contiennent tous deux V2. Noter qu'il s'agit d'un nouvel élément de syntaxe dans TLS 1.2.

Le corps de la variante de structure peut être référencé par une étiquette. Le mécanisme de choix de la variante au moment de l'exécution n'est pas imposé par le langage de présentation.

```
struct {
      T1 f1;
      T2 f2;
      ....
      Tn fn;
        select (E) {
          cas e1: Te1;
          cas e2: Te2;
          cas e3: cas e4: Te3;
          ....
          cas en: Ten;
        } [[fv]];
} [[Tv]];
```

Par exemple :

```
enum { pomme, orange, banane } VariantTag;

struct {
      numéro uint16;
      chaîne opaque<0..10>;          /* longueur variable */
} V1;

struct {
      numéro uint32;
      chaîne opaque[10];             /* longueur fixée */
} V2;

struct {
      select (VariantTag) {          /* la valeur de sélecteur est implicite */
          cas pomme :
              V1;                    /* VariantBody, étiquette = pomme */
          cas orange :
          cas banane :
              V2;                    /* VariantBody, étiquette = orange ou banane */
      } variant_body;                /* étiquette facultative sur la variante*/
} VariantRecord;
```

4.7 Attributs cryptographiques

Les cinq opérations cryptographiques – signature numérique, cryptage du flux chiffré, cryptage du bloc chiffré, chiffrement authentifié avec données supplémentaires (AEAD, *authenticated encryption with additional data*), et chiffrement de clé publique – sont conçues respectivement comme signée numériquement, à flux chiffré, à bloc chiffré, à aead chiffré, et à clé publique chiffrée. Un traitement cryptographique du champ est spécifié avec l'ajout de la désignation d'un mot clé approprié devant la spécification du type du champ. Les clés cryptographiques sont impliquées par l'état de session en cours (voir au paragraphe 6.1).

Un élément de signature numérique est codé comme une structure DigitallySigned :

```
struct {
      algorithme SignatureAndHashAlgorithm;
      opaque signature<0..2^16-1>;
```

} DigitallySigned;

Le champ "algorithme" spécifie l'algorithme utilisé (voir au paragraphe 7.4.1.4.1 la définition de ce champ). Noter que l'introduction du champ "algorithme" est un changement par rapport aux versions précédentes. La signature est une signature numérique qui utilise ces algorithmes sur le contenu de l'élément. Les contenus eux-mêmes n'apparaissent pas sur le réseau mais sont simplement calculés. La longueur de la signature est spécifiée par l'algorithme de signature et la clé.

Dans la signature RSA, le vecteur opaque contient la signature générée en utilisant le schéma de signature RSASSA-PKCS1-v1_5 défini dans [PKCS1]. Comme exposé dans [PKCS1], le DigestInfo DOIT être codé en DER [X680] [X690]. Pour les algorithmes de hachage sans paramètre (qui incluent SHA-1), le champ DigestInfo.AlgorithmIdentifier.parameters DOIT être NULL, mais les mises en œuvre DOIVENT accepter à la fois sans paramètre et avec le paramètre NULL. Noter que les versions précédentes de TLS utilisaient un schéma de signature RSA différent qui ne comportait pas de codage de DigestInfo.

Dans DSA, les 20 octets du hachage SHA-1 sont passés directement à travers l'algorithme de signature numérique sans hachage supplémentaire. Cela produit deux valeurs, r et s. La signature DSA est un vecteur opaque, comme ci-dessus, dont le contenu est le codage en DER de :

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

Note : Dans la terminologie courante, DSA se réfère à l'algorithme de signature numérique et DSS se réfère à la norme du NIST. Dans les spécifications SSL et TLS d'origine, "DSS" était d'utilisation universelle. Le présent document utilise "DSA" pour se référer à l'algorithme, "DSS" pour se référer à la norme, et il utilise "DSS" dans les définitions de codets pour la continuité historique.

Dans le cryptage de flux chiffré, le libellé est combiné par opérateur OUX exclusif avec une quantité identique de résultat généré à partir d'un générateur de nombres pseudo aléatoires cryptographiquement sûr.

Dans le cryptage de bloc chiffré, chaque bloc de libellé crypte un bloc de texte chiffré. Tout le cryptage de bloc de chiffrement est effectué en mode CBC (chaînage de bloc de chiffrement), et tous les items qui sont chiffrés par bloc seront un multiple exact de la longueur du bloc de chiffrement.

En cryptage AEAD, le libellé est simultanément chiffré et protégé en intégrité. L'entrée peut être de n'importe quelle longueur, et le résultat chiffré en aead est généralement plus grand que l'entrée afin de s'accommoder de la valeur de preuve d'intégrité.

Dans le chiffrement par clé publique, un algorithme de clé publique est utilisé pour chiffrer les données de telle façon 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 chiffrement et la clé.

Le chiffrement RSA est fait à l'aide du schéma de cryptage RSAES-PKCS1-v1_5 défini dans [PKCS1].

Dans l'exemple suivant,

```
stream-ciphered struct {
      uint8 field1;
      uint8 field2;
      digitally-signed opaque {
        uint8 field3<0..255>;
        uint8 field4;
      };
} UserType;
```

Les contenus des structures internes (field3 et field4) sont utilisés comme entrée pour l'algorithme de signature/hachage, et ensuite la structure entière est cryptée avec le chiffrement de flux. La longueur de cette structure, en octets, serait égale à deux octets pour field1 et field2, plus deux octets pour la signature et l'algorithme de hachage, plus deux octets pour la longueur de la signature, plus la longueur du résultat de l'algorithme de signature. La longueur de la signature est connue parce que l'algorithme et la clé utilisés pour la signature sont connus avant de coder ou décoder cette structure.

4.8 Constantes

Des constantes typées peuvent être définies pour les besoins de la spécification en déclarant un symbole du type désiré et en lui allouant des valeurs.

Il ne peut pas être alloué de valeurs aux types sous spécifiés (opaque, vecteurs de longueur variable, et structures qui contiennent opaque). Aucun champ d'une structure ou vecteur multiéléments ne peut être élidé.

Par exemple :

```
struct {
      uint8 f1;
           uint8 f2;
} Exemple1;

Exemple1 ex1 = {1, 4};              /* alloue f1 = 1, f2 = 4 */
```

5. HMAC et la fonction pseudo aléatoire

La couche d'enregistrement TLS utilise un code d'authentification de message (MAC, *Message Authentication Code*) chiffré pour protéger l'intégrité du message. Les suites de chiffrement définies dans le présent document utilisent une construction connue sous le nom de HMAC, décrite dans [HMAC], qui se fonde sur une fonction de hachage. D'autres suites de chiffrement PEUVENT définir si nécessaire leur propres constructions de MAC.

De plus, il est exigé d'une construction qu'elle fasse l'expansion des secrets en blocs de données pour les besoins de la génération ou validation de clés. Cette fonction pseudo aléatoire (PRF, *pseudo random function*) prend en entrée un secret, un germe et une étiquette d'identification, et produit une sortie de longueur arbitraire.
Dans la présente section, on définit une PRF, fondée sur HMAC. Cette PRF est utilisée avec la fonction de hachage SHA-256 pour toutes les suites de chiffrement définies dans le présent document et dans les documents TLS publiés avant le présent document lorsque TLS 1.2 est négocié. De nouvelles suites de chiffrement DOIVENT explicitement spécifier une PRF et, en général, DEVRAIENT utiliser la PRF TLS avec SHA-256 ou une fonction de hachage standard plus forte.

D'abord, on définit une fonction d'expansion de données, P_hash(secret, données), qui utilise une seule fonction de hachage pour expanser un secret et un germe en une quantité arbitraire de sortie :

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

où + indique la concaténation.

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_SHA256 est utilisé pour créer 80 octets de données, il devra être itéré trois fois (à travers A(3)), créant 96 octets de données de sortie ; les 16 derniers octets de l'itération finale seront alors éliminés, laissant 80 octets de données de sortie.

La PRF de TLS est créée en appliquant P_hash au secret :

```
PRF(secret, étiquette, germe) = P_<hash>(secret, étiquette + germe)
```

L'étiquette est une chaîne ASCII. Elle devrait être incluse sous la forme exacte sous laquelle elle est donnée sans octet de longueur ou 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
```

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, description, et 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, réassemblées, et puis livrées aux clients de niveau supérieur.

Quatre protocoles qui utilisent le 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 de données d'application. Afin de permettre l'extension du protocole TLS, des types de contenu d'enregistrement supplémentaires peuvent être pris en charge par le protocole d'enregistrement. De nouvelles valeurs de type de contenu d'enregistrement sont allouées par l'IANA dans le registre Type de contenu TLS comme décrit à la Section 12.

Les mises en œuvre NE DOIVENT PAS envoyer de types d'enregistrement non définis dans le présent document sauf négociés par une extension. Si une mise en œuvre TLS reçoit un type d'enregistrement inattendu, elle DOIT envoyer une alerte message_inattendu.

Tout protocole destiné à être utilisé sur TLS doit être conçu avec soin pour traiter toutes les attaques possibles contre lui. En pratique, cela signifie que le concepteur du protocole doit être conscient des propriétés que TLS fournit et ne fournit pas en matière de sécurité et qu'il ne peut s'appuyer sur celles qui ne sont pas fournies.

Noter en particulier que le type et la longueur d'un enregistrement ne sont pas protégés par le chiffrement. Si ces informations ne sont pas sensibles par elles-mêmes, les concepteurs d'applications peuvent souhaiter prendre des mesures (bourrage, trafic masqué) pour minimiser la fuite d'information.

6.1 États de connexion

Un état de connexion TLS est l'environnement opérationnel 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 : la clé MAC et les clés de chiffrement brut pour la connexion pour les deux directions de lecture et d'écriture. Logiquement, il y a toujours quatre états de connexion en cours : les états de lecture et d'écriture en cours, et les états de lecture et d'écriture en instance. Tous les enregistrements sont traités dans les états de lecture et d'écriture en cours. Les paramètres de sécurité pour les états en instance peuvent être établis par le protocole de prise de contact TLS, et le ChangeCipherSpec peut au choix rendre les états en instance en cours, auquel cas l'état en cours approprié est abandonné et est remplacé par l'état en instance ; l'état en instance est alors réinitialisé comme état vide. Il est illégal de faire un état en cours d'un état qui n'a pas été initialisé avec des paramètres de sécurité. L'état initial en cours spécifie toujours qu'aucun chiffrement, compression, ou MAC ne sera utilisé.

Les paramètres de sécurité pour un état lecture et écriture d'une connexion TLS sont établis par la fourniture des valeurs suivantes :

fin de connexion
Selon que cette entité est considérée comme "client" ou comme "serveur" dans cette connexion.

algorithme PRF
Algorithme utilisé pour générer des clés à partir du secret maître (voir aux paragraphes 5 et 6.3).

algorithme de chiffrement brut
Algorithme à utiliser pour le chiffrement brut. La présente spécification comporte la taille de clé de cet algorithme, selon qu'il est un chiffrement de bloc, de flux, ou AEAD, la taille de bloc du chiffrement (si c'est appropriée), et les longueurs des vecteurs d'initialisation explicite et implicite (ou noms occasionnels).

algorithme MAC
Algorithme à utiliser pour l'authentification de message. La présente spécification inclut la taille de la valeur retournée par l'algorithme MAC.

algorithme de compression
Algorithme à utiliser pour la compression des données. La présente spécification doit inclure toutes les informations qu'exige l'algorithme pour effectuer la compression.

secret maître

Secret de 48 octets partagé par les deux homologues dans la connexion.

aléa de client
Valeur de 32 octets fournie par le client.

aléa de serveur
Valeur de 32 octets fournie par le serveur.

Ces paramètres sont définis dans le langage de présentation comme :

```
enum { serveur, client } ConnectionEnd;
enum { tls_prf_sha256 } PRFAlgorithm;
enum { null, rc4, 3des, aes } BulkCipherAlgorithm;
enum { flux, bloc, aead } CipherType;
enum { null, hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, hmac_sha512} MACAlgorithm;
enum { null(0), (255) } CompressionMethod;

/* Les algorithmes spécifiés dans CompressionMethod, PRFAlgorithm, BulkCipherAlgorithm, et MACAlgorithm peuvent
s'ajouter. */

struct {
    ConnectionEnd          entity;
    PRFAlgorithm           prf_algorithm;
    BulkCipherAlgorithm    bulk_cipher_algorithm;
    CipherType             cipher_type;
    uint8                  enc_key_length;
    uint8                  block_length;
    uint8                  fixed_iv_length;
    uint8                  record_iv_length;
    MACAlgorithm           mac_algorithm;
    uint8                  mac_length;
    uint8                  mac_key_length;
    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 (dont certains ne sont pas exigés par tous les chiffrements, et sont donc vides) :

- clé MAC écrite par le client
- clé MAC écrite par le serveur
- clé de chiffrement écrite par le client
- clé de chiffrement écrite par le serveur
- VI écrit par le client
- VI écrit par le serveur

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 établis 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
L'état en cours de l'algorithme de compression.

état de chiffrement
État en cours de l'algorithme de chiffrement. Cela va consister en la clé programmée pour cette connexion. Pour les chiffrements de flux, cela va aussi contenir toute information d'état nécessaire pour permettre que le flux continue de chiffrer ou déchiffrer les données.

clé MAC
La clé MAC pour cette connexion, telle que générée 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 réglé à zéro chaque fois qu'un état de connexion passe à l'état actif. Les numéros de séquence sont de type uint64 et ne peuvent pas excéder 2^64-1. Les numéros de séquence ne reviennent pas à zéro. Si une mise en œuvre TLS avait besoin de remettre à zéro un numéro de séquence, elle devrait plutôt le renégocier. Un numéro de séquence est incrémenté après chaque enregistrement : précisément, le premier enregistrement transmis sous un état de connexion particulier DOIT utiliser le numéro de séquence 0.

6.2 Couche d'enregistrement

La couche d'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 de message client ne sont pas préservées dans la couche enregistrement (c'est-à-dire, plusieurs messages client du même ContentType PEUVENT être fondus en un seul enregistrement TLSPlaintext, ou un seul message PEUT être fragmenté en plusieurs enregistrements).

```
struct {
    uint8 majeur;
    uint8 mineur;
} ProtocolVersion;

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

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

type
Protocole de niveau supérieur utilisé pour traiter le fragment inclus.

version
Version du protocole utilisée. Le présent document décrit la version 1.2 de TLS, qui utilise la version { 3, 3 }. La valeur de version 3.3 est historique, et découle de l'utilisation de {3, 1} pour TLS 1.0. (Voir l'Appendice A.1.) Noter qu'un client qui accepte plusieurs versions de TLS peut ne pas savoir quelle version sera employée avant de recevoir le ServerHello. Voir à l'Appendice E l'exposé sur le numéro de version de couche d'enregistrement qui peut être employé pour ClientHello.

longueur
Longueur (en octets) du fragment TLSPlaintext.fragment suivant. La longueur NE DOIT PAS excéder 2^14.

fragment
Données d'application. Ces données sont transparentes et traitées comme un bloc indépendant à traiter par le protocole de niveau supérieur spécifié par le champ type.

Les mises en œuvre NE DOIVENT PAS envoyer de fragments de longueur zéro des types de contenu Handshake, Alert, ou ChangeCipherSpec. Les fragments de longueur zéro de données d'application PEUVENT être envoyés parce qu'ils peuvent être utiles pour des contre-mesures d'analyse de trafic.

Note : Des données de différents types de contenu de couche d'enregistrement TLS PEUVENT être entrelacées. Les données d'application sont généralement de préséance de transmission inférieure à celle des autres types de contenu. Cependant, les enregistrements DOIVENT être livrés au réseau dans le même ordre que celui dans lequel ils sont protégés par la couche enregistrement. Les receveurs DOIVENT recevoir et traiter le trafic entrelacé de couche d'application durant les prises de

contact qui suivent la première d'une connexion.

6.2.2 Compression et décompression d'enregistrement

Tous les enregistrements sont compressés en utilisant l'algorithme de compression défini dans l'état de session en cours. 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 structure TLSCompressed. Les fonctions de compression sont initialisées avec les informations d'état par défaut chaque fois qu'un état de connexion est rendu actif. La [RFC3749] décrit les algorithmes de compression pour TLS.

La compression doit se faire sans perte et ne doit pas accroître la longueur du contenu de plus de 1024 octets. Si la fonction de décompression rencontre un fragment TLSCompressed.fragment qui se décompresserait sur une longueur dépassant 2^14 octets, elle DOIT rapporter une erreur fatale d'échec de décompression.

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

longueur
Longueur (en octets) du fragment TLSCompressed.fragment suivant. La longueur NE DOIT PAS dépasser 2^14 + 1024.

fragment
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 causent pas de débordement de mémoire tampon interne.

6.2.3 Protection de charge utile d'enregistrement

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

```
struct {
    type ContentType;
    version ProtocolVersion;
    longueur uint16;
    choisir (SecurityParameters.cipher_type) {
        cas stream:   GenericStreamCipher;
        cas block:    GenericBlockCipher;
        cas aead:     GenericAEADCipher;
    } fragment;
} TLSCiphertext;
```

type
Le champ type est identique au TLSCompressed.type.

version
Le champ version est identique au TLSCompressed.version.

longueur
Longueur (en octets) du fragment TLSCiphertext.fragment suivant. La longueur NE DOIT PAS excéder 2^14 + 2048.
fragment
Forme chiffrée du TLSCompressed.fragment, avec le MAC.

6.2.3.1 Chiffrement de flux Null ou Standard

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

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

Le MAC est généré comme :

```
MAC(MAC_write_key, seq_num +
            TLSCompressed.type +
            TLSCompressed.version +
            TLSCompressed.length +
            TLSCompressed.fragment);
```

où "+" note l'enchaînement.

seq_num
Le numéro de séquence pour cet enregistrement.

MAC
L'algorithme de MAC 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 de vecteur de synchronisation (tels que RC4), l'état de chiffrement de flux à partir de la fin d'un enregistrement est simplement utilisé sur le paquet suivant. Si la suite de chiffrement est TLS_NULL_WITH_NULL_NULL, le chiffrement consiste en l'opération identique (c'est-à-dire que les données ne sont pas chiffrées, et la taille du MAC est zéro, ce qui implique qu'aucun MAC n'est utilisé). Aussi bien pour les chiffrements null que de flux, TLSCiphertext.length est TLSCompressed.length plus SecurityParameters.mac_length.

6.2.3.2 Bloc de chiffrement CBC

Pour le chiffrement de bloc (tel que 3DES ou AES), les fonctions de chiffrement et MAC convertissent les structures TLSCompressed.fragment de et vers les structures de bloc TLSCiphertext.fragment.

```
struct {
        opaque IV[SecurityParameters.record_iv_length];
        block-ciphered struct {
          content[TLSCompressed.length] opaque ;
          MAC[SecurityParameters.mac_length] opaque ;
          uint8 padding[GenericBlockCipher.padding_length];
          uint8 padding_length;
        };
} GenericBlockCipher;
```

Le MAC est généré comme décrit au paragraphe 6.2.3.1.

IV
Le vecteur d'initialisation (IV) DEVRAIT être choisi de façon aléatoire, et DOIT être imprévisible. Noter que dans les versions de TLS antérieures à 1.1, il n'y avait pas de champ IV, et le dernier bloc de texte chiffré de l'enregistrement précédent (le "résidu CBC") était utilisé comme IV. Cela a été changé pour empêcher les attaques décrites dans [CBCATT]. Pour les chiffrements de bloc, la longueur d'IV est la longueur SecurityParameters.record_iv_length, qui est égale à la taille SecurityParameters.block_size.

padding
Bourrage qui est ajouté pour forcer la longueur du libellé 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 que le résultat soit une TLSCiphertext.length multiple entier de la longueur de bloc. Des longueurs supérieures à ce qui est nécessaire peuvent être désirables pour déjouer les attaques contre un protocole fondées sur l'analyse des longueurs des messages échangés. Chaque uint8 dans le vecteur données de bourrage DOIT être rempli avec la valeur de la longueur du bourrage. Le receveur DOIT vérifier ce bourrage et DOIT utiliser l'alerte bad_record_mac pour indiquer les erreurs de bourrage.

padding_length
La longueur du bourrage DOIT être telle que la taille totale de la structure GenericBlockCipher soit un multiple de la longueur de bloc du chiffrement. Les valeurs légales sont dans la gamme de zéro à 255, inclus. Cette longueur spécifie la longueur du champ Bourrage, champ padding_length exclu.

La longueur des données chiffrées (TLSCiphertext.length) est supérieure de un à la somme de SecurityParameters.block_length, TLSCompressed.length, SecurityParameters.mac_length, et padding_length.

Exemple : Si la longueur de bloc est de 8 octets, la longueur du contenu (TLSCompressed.length) est de 61 octets, et longueur MAC est de 20 octets, la longueur avant bourrage est alors de 82 octets (ce qui n'inclut pas le IV. Et donc, la longueur du bourrage modulo 8 doit être égale à 6 afin de faire de 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 le minimum nécessaire, 6, le bourrage serait 6 octets, chacun contenant la valeur 6. Et 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 : Lorsque le chiffrement de bloc est en mode CBC (*Cipher Block Chaining,* chaînage de bloc de chiffrement), il est critique que le libellé entier de l'enregistrement soit connu avant la transmission d'aucun texte chiffré. Autrement, il serait possible à un agresseur de monter l'attaque décrite en [CBCATT].

Note pour la mise en œuvre : Canvel et al. [CBCTIME] ont démontré une attaque de cadencement contre le bourrage de CBC fondée sur le temps nécessaire au calcul du MAC. Afin de se défendre contre cette attaque, les mises en œuvre DOIVENT s'assurer que le temps de traitement de l'enregistrement est sensiblement le même, que le bourrage soit correct ou non. En général, la meilleure façon de le faire est de calculer le MAC même si le bourrage est incorrect, et de ne rejeter le paquet qu'ensuite. Par exemple, si le bourrage paraît être incorrect, la mise en œuvre pourrait supposer un bourrage de longueur zéro et calculer ensuite le MAC. Cela laisse un petit espace de temps, dans la mesure où la performance du MAC dépend dans une certaine mesure de la taille du fragment de données, qui n'est pas assez important pour être exploitable, du fait de la grande taille des blocs MAC existants et de la petite taille du signal de synchronisation.

6.2.3.3 Chiffrements AEAD

Pour les chiffrements AEAD [AEAD] (tels que [CCM] ou [GCM]), la fonction AEAD convertit les structures TLSCompressed.fragment de et en structures TLSCiphertext.fragment AEAD.

```
struct {
      nonce_explicit[SecurityParameters.record_iv_length] opaque ;
      aead-ciphered struct {
        content[TLSCompressed.length] opaque ;
        };
} GenericAEADCipher;
```

Les chiffrements AEAD prennent en entrée une seule clé, un nom occasionnel, un libellé, et des "données additionnelles" à inclure dans la vérification d'authentification, comme décrit au paragraphe 2.1 de [AEAD]. La clé est client_write_key ou server_write_key. Aucune clé MAC n'est utilisée.

Chaque suite de chiffrement AEAD DOIT spécifier comment est construit le nom occasionnel fourni à l'opération AEAD, et quelle est la longueur de la partie GenericAEADCipher.nonce_explicit. Dans de nombreux cas, il est approprié d'utiliser la technique de nom occasionnel partiellement implicite décrite au paragraphe 3.2.1 de [AEAD] ; avec record_iv_length comme longueur de la partie explicite. Dans ce cas, la partie implicite DEVRAIT être déduite de key_block comme client_write_iv et server_write_iv (comme décrit au paragraphe 6.3), et la partie explicite est incluse dans GenericAEAEDCipher.nonce_explicit.

Le libellé est le TLSCompressed.fragment.

Les données authentifiées additionnelles, qu'on note additional_data, sont définies comme suit :

additional_data = seq_num + TLSCompressed.type + TLSCompressed.version + TLSCompressed.length;

où "+" note l'enchaînement.

Le aead_output consiste en la sortie de texte chiffré par l'opération de chiffrement AEAD. La longueur sera généralement supérieure à TLSCompressed.length, mais d'une quantité qui varie avec le chiffrement AEAD. Comme les chiffrements peuvent incorporer du bourrage, la quantité de redondance peut varier avec les différentes valeurs de TLSCompressed.length. Aucun chiffrement AEAD NE DOIT produire une expansion supérieure à 1024 octets. Symboliquement,

AEADEncrypted = AEAD-Encrypt(write_key, nom occasionnel, libellé, additional_data)

Afin de déchiffrer et vérifier, le chiffrement prend en entrée la clé, le nom occasionnel, les "additional_data", et la valeur AEADEncrypted. Le résultat est soit le libellé soit une erreur qui indique l'échec du déchiffrement. Il n'y a pas de vérification d'intégrité distincte. C'est à dire :

TLSCompressed.fragment = AEAD-Decrypt(write_key, nom occasionnel, AEADEncrypted, additional_data)

Si le déchiffrement échoue, une alerte bad_record_mac fatale DOIT être générée.

6.3 Calcul des clés

Le protocole d'enregistrement exige un algorithme pour générer les clés requises par l'état de connexion en cours (voir l'Appendice A.6) à partir des paramètres de sécurité fournis par le protocole de prise de contact.

Le secret maître est expansé en une séquence d'octets sûrs, qui est ensuite partagée en une clé MAC écrite par le client, une clé MAC écrite par le serveur, une clé de chiffrement écrite par le client, et une clé de chiffrement écrite par le serveur. Chacune d'elles est générée à partir de la séquence d'octets dans cet ordre. Les valeurs non utilisées sont vides. Certains chiffrements AEAD peuvent de plus exiger un IV écrit par le client et un IV écrit par le serveur IV (voir au paragraphe 6.2.3.3).

Lorsque les clés et les clés MAC sont générées, le secret maître est utilisé comme source d'entropie.

Pour générer le matériel de clés, calculer

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

jusqu'à ce que suffisamment de sortie ait été générée. Puis, le key_block est partagé comme suit :

```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

Actuellement, le client_write_IV et le server_write_IV sont seulement générés pour les techniques de nom occasionnel implicite comme décrit au paragraphe 3.2.1 de [AEAD].

Note de mise en œuvre : La suite de chiffrement actuellement définie qui exige le plus de matériel est AES_256_CBC_SHA256. Elle exige des clés de 2 x 32 octets et des clés MAC de 2 x 32 octets, pour un total de 128 octets de matériel de clé.

7. Protocoles TLS de prise de contact

TLS a 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, pour s'authentifier mutuellement, pour instancier les paramètres de sécurité négociés, et pour se rapporter l'un l'autre les conditions d'erreur.

Le protocole de prise de contact est chargé de la négociation d'une session, qui comporte les éléments suivants :

identifiant de session
Séquence d'octets arbitraire choisie par le serveur pour identifier un état de session actif ou reprenable.

certificat d'homologue
Certificat X509v3 [PKIX] de l'homologue. Cet élément de l'état peut être nul.

méthode de compression
Algorithme utilisé pour compresser les données avant le chiffrement.

spécification du chiffrement
Spécifie la fonction pseudo aléatoire (PRF) utilisée pour générer le matériel de clé, l'algorithme de chiffrement de données brutes (tel que null, AES, etc.) et l'algorithme de MAC (tel que HMAC-SHA1). Il définit aussi les attributs cryptographiques tels que mac_length. (Voir les définitions formelles à l'Appendice A.6.)

secret maître
Secret de 48 octets partagé entre le client et le serveur.

est reprenable
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 qui serviront à la couche d'enregistrement lors de la protection des données d'application. De nombreuses connexions peuvent être instanciées en utilisant la même session grâce au dispositif de reprise du protocole de prise de contact de TLS.

7.1 Protocole de changement de spécification de chiffrement

Le protocole de changement de spécification de chiffrement existe pour signaler les transitions de stratégie de chiffrement. Le protocole consiste en un seul message, qui est chiffré et compressé sous l'état de connexion en cours (pas celui en instance). Le message consiste en un seul octet de valeur 1.

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

Le message ChangeCipherSpec est envoyé à la fois par le client et le serveur pour notifier au receveur que les enregistrements suivants seront protégés avec le CipherSpec et les clés nouvellement négociés. La réception de ce message amène le receveur à donner pour instruction à la couche d'enregistrement de copier immédiatement l'état de lecture en instance dans l'état de lecture en cours. Immédiatement après l'envoi de ce message, l'envoyeur DOIT donner pour instruction à la couche enregistrement de faire de l'état écriture en instance l'état écriture actif.

(Voir au paragraphe 6.1.) Le message ChangeCipherSpec est envoyé durant la prise de contact après l'accord sur les paramètres de sécurité, mais avant l'envoi du message de fin de vérification.

Note : Si une reprise de contact survient alors que des données s'écoulent sur une connexion, les parties communicantes peuvent continuer à envoyer des données en utilisant le vieux CipherSpec. Cependant, une fois que ChangeCipherSpec a été envoyé, le nouveau CipherSpec DOIT être utilisé. Le premier côté qui envoie le ChangeCipherSpec ne sait pas si l'autre côté a fini de calculer le nouveau matériel de clés (par exemple, si il doit effectuer une longue opération de clé publique). Et donc, une petite fenêtre temporelle, durant laquelle le receveur doit mettre les données en mémoire tampon, PEUT exister. En pratique, avec les machines modernes, cet intervalle sera vraisemblablement très court.

7.2 Protocole d'alerte

Un des types de contenu pris en charge par la couche d'enregistrement TLS est le type alerte. Les messages d'alerte portent la sévérité du message (avertissement ou fatal) et une description de l'alerte. Les messages d'alerte de niveau 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 qui a échoué 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 de connexion en cours.

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

enum {
     close_notify(0),                   /* notification de clôture*/
     unexpected_message(10),            /* message inattendu*/
     bad_record_mac(20),                /* mauvais enregisrement de MAC*/
     decryption_failed_RESERVED(21),    /* échec du déchiffrement*/
     record_overflow(22),               /* débordement de l'enregistrement*/
     decompression_failure(30),         /* échec de décompression*/
     handshake_failure(40),             /* échec de prise de contact*/
     no_certificate_RESERVED(41),       /* pas de certificat*/
```

```
        bad_certificate(42),                    /* mauvais certificat*/
        unsupported_certificate(43),            /* certificat non accepté*/
        certificate_revoked(44),                /* certificat révoqué*/
        certificate_expired(45),                /* certificat périmée*/
        certificate_unknown(46),                /* certicat inconnu*/
        illegal_parameter(47),                  /* paramètre illégal*/
        unknown_ca(48),                         /* autorité de certificat inconnue*/
        access_denied(49),                      /* accès refusé*/
        decode_error(50),                       /* erreur de décodage*/
        decrypt_error(51),                      /* erreur de déchiffrement*/
        export_restriction_RESERVED(60),        /* interdiction d'exporter*/
        protocol_version(70),                   /* version du protocole*/
        insufficient_security(71),              /* sécurité insuffisante*/
        internal_error(80),                     /* erreur interne*/
        user_canceled(90),                      /* utilisateur annulé*/
        no_renegotiation(100),                  /* pas de renégociation*/
        unsupported_extension(110),             /* extension non acceptée*/
        (255)
    } AlertDescription;

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

7.2.1 Alertes de clôture

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

close_notifyCe message notifie au receveur que l'envoyeur n'enverra plus aucun message sur cette connexion. Noter que comme dans TLS 1.1, l'échec à clore de façon appropriée une connexion n'exige plus qu'une session ne soit pas reprise. C'est un changement par rapport à TLS 1.0 pour se conformer à une pratique largement répandue des mises en œuvre.

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

Sauf si une autre alerte fatale a été transmise, chaque partie est obligée d'envoyer une alerte close_notify avant de clore le côté écriture de la connexion. L'autre partie DOIT répondre par l'envoi d'une alerte close_notify et clore immédiatement la connexion, en éliminant toutes les écritures en instance. Il n'est pas exigé que l'initiateur de la clôture attende l'alerte close_notify de réponse avant de clore le côté lecture de la connexion.

Si le protocole d'application qui utilise TLS s'arrange pour que toutes les données puissent être transportées sur le transport sous-jacent après la clôture de la connexion TLS, la mise en œuvre TLS doit recevoir l'alerte close_notify de réponse avant d'indiquer à la couche application que la connexion TLS s'est terminée. Si le protocole d'application ne transfère aucune donnée supplémentaire, mais va seulement clore la connexion de transport sous-jacent, la mise en œuvre PEUT alors choisir de clore le transport sans attendre le close_notify de réponse. Aucune partie de la présente norme ne devrait être comprise comme dictant la manière dont un profil d'utilisation de TLS gère son transport de données, y compris lorsque les connexions sont ouvertes ou fermées.

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

7.2.2 Alertes d'erreur

Le traitement d'erreur est très simple dans le protocole de prise de contact TLS. Lorsqu'une erreur est détectée, la partie qui l'a détectée envoie un message à l'autre partie. À réception ou émission d'un message d'alerte fatale, les deux parties closent immédiatement la connexion. Les serveurs et clients DOIVENT oublier tous les identifiants de session, clés, et secrets associés à une connexion échouée. Et donc, toute connexion terminée par une alerte fatale NE DOIT PAS être reprise.

Chaque fois qu'une mise en œuvre rencontre une condition définie comme alerte fatale, elle DOIT envoyer l'alerte appropriée avant de clore la connexion. Pour toutes les erreurs où un niveau d'alerte n'est pas explicitement spécifié, l'envoyeur PEUT déterminer, à sa discrétion si il doit la traiter comme erreur fatale ou non. Si la mise en œuvre choisit d'envoyer une alerte mais

à l'intention de clore la connexion immédiatement après, il DOIT envoyer cette alerte au niveau alerte fatale.

Si une alerte du niveau avertissement est envoyée et reçue, la connexion peut généralement continuer normalement. Si la partie receveuse décide de ne pas poursuivre la connexion (par exemple, après avoir reçu une alerte no_renegotiation qu'il ne veut pas accepter), elle DEVRAIT envoyer une alerte fatale pour terminer la connexion. Cela étant dit, l'envoyeur ne peut pas, en général, savoir comment le receveur va se comporter. Donc, les alertes d'avertissement ne sont pas très utiles lorsque l'envoyeur veut continuer la connexion, et elles sont donc parfois omises. Par exemple, si un homologue décide d'accepter un certificat arrivé à expiration (peut-être après l'avoir confirmé auprès de l'usager) et veut continuer la connexion, il ne va généralement pas envoyer une alerte certificate_expired.

Les alertes d'erreur suivantes sont définies :

unexpected_message
Un message inapproprié a été reçu. Cette alerte est toujours fatale et ne devrait jamais être observée dans une communication entre des mises en œuvre appropriées.

bad_record_mac
Cette alerte est retournée si un enregistrement est reçu avec un MAC incorrect. Cette alerte DOIT aussi être retournée si une alerte est envoyée à cause d'un TLSCiphertext déchiffré d'une façon invalide : soit il n'est pas un multiple pair de la longueur de bloc, soit ses valeurs de bourrage n'apparaissent pas correctes à la vérification. Ce message est toujours fatal et ne devrait jamais être observé dans une communication entre des mises en œuvre appropriées (sauf lorsque des messages ont été lésés dans le réseau).

decryption_failed_RESERVED
Cette alerte était utilisée dans certaines anciennes versions de TLS, et peut avoir permis certaines attaques contre le mode CBC [CBCATT]. Elle NE DOIT PAS être envoyée par les mises en œuvre conformes.

record_overflow
Un enregistrement TLSCiphertext a été reçu avec une longueur supérieure à 2^14+2048 octets, ou un enregistrement a été déchiffré en un enregistrement TLSCompressed avec plus de 2^14+1024 octets. Ce message est toujours fatal et ne devrait jamais être observé dans une communication entre des mises en œuvre appropriées (sauf lorsque des messages ont été lésés dans le réseau).

decompression_failure
La fonction de décompression a reçu une entrée inappropriée (par exemple, des données dont l'expansion donnerait une longueur excessive). Ce message est toujours fatal et ne devrait jamais être observé dans une communication entre des mises en œuvre appropriées.

handshake_failure
La réception d'un message d'alerte handshake_failure indique que l'envoyeur s'est révélé incapable de négocier un ensemble de paramètres de sécurité acceptable étant données les options disponibles. C'est une erreur fatale.

no_certificate_RESERVED
Cette alerte était utilisée dans SSLv3 mais ne l'était dans aucune version de TLS. Elle NE DOIT PAS être envoyée par les mises en œuvre conformes.

bad_certificate
Un certificat a été lésé, contenait des signatures qui ne sont pas vérifiées correctement, etc.

unsupported_certificate
Un certificat est d'un type non pris en charge.

certificate_revoked
Un certificat a été révoqué par son signataire.

certificate_expired
Un certificat est arrivé à expiration ou n'est plus actuellement valide.

certificate_unknown
Quelque autre problème (non spécifié) est survenu dans le traitement du certificat, le rendant inacceptable.

illegal_parameter
Un champ de la prise de contact se trouve hors gamme ou incohérent avec d'autres champs. Ce message est toujours fatal.

unknown_ca
Une chaîne de certificat ou une chaîne partielle de certificat a été reçue, mais le certificat n'a pas été accepté parce que l'autorité de certification (CA) du certificat n'a pas pu être localisée ou n'a pu être mise en relation avec une CA connue et de confiance. Ce message est toujours fatal.

access_denied
Un certificat valide a été reçu, mais lors de l'application du contrôle d'accès, l'envoyeur a décidé de ne pas poursuivre la négociation. Ce message est toujours fatal.

decode_error
Un message n'a pas pu être décodé parce qu'un champ se trouve hors de la gamme spécifiée ou la longueur du message était incorrecte. Ce message est toujours fatal et ne devrait jamais être observé dans une communication entre des mises en œuvre appropriées (sauf lorsque des messages ont été lésés dans le réseau).

decrypt_error
Une opération cryptographique de la prise de contact a échoué, qui peut être de ne pas réussir à vérifier correctement une signature ou valider un message Terminé. Ce message est toujours fatal.

export_restriction_RESERVED
Cette alerte était utilisée dans certaines versions antérieures de TLS. Elle NE DOIT PAS être envoyée par des mises en œuvre conformes.

protocol_version
La version du protocole que le client a tenté de négocier est reconnue mais pas prise en charge. (Par exemple, d'anciennes versions du protocole peuvent être évitées pour des raisons de sécurité.) Ce message est toujours fatal.

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

internal_error
Une erreur interne sans relation avec l'homologue ou la rectitude du protocole (comme un défaut d'allocation de mémoire) rend impossible sa poursuite. Ce message est toujours fatal.

user_canceled
Cette prise de contact est annulée pour une raison sans rapport avec une défaillance du protocole. Si l'utilisateur annulé une opération après l'achèvement de la prise de contact, il est plus approprié de clore simplement la connexion en envoyant un close_notify. Cette alerte devrait être suivie d'un close_notify. Ce message est généralement un avertissement.

no_renegotiation
Envoyé par le client en réponse à une demande hello ou par le serveur en réponse au hello d'un client après la prise de contact initiale. L'un et l'autre devrait normalement conduire à renégociation ; lorsque ce n'est pas approprié, le receveur devrait répondre par cette alerte. À ce point, le demandeur d'origine peut décider s'il va poursuivre la connexion. Un cas où ce serait approprié est celui d'un serveur qui a engagé 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 difficile de communiquer des changements à ces paramètres après ce point. Ce message est toujours un avertissement.

unsupported_extension
Envoyé par des clients qui reçoivent une extension de hello de serveur qui contient une extension qu'ils n'avaient pas mise dans le hello de client correspondant. Ce message est toujours fatal.

Les nouvelles valeurs d'alerte allouées par l'IANA sont décrites à la Section 12.

7.3 Généralités sur le 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 d'enregistrement TLS. Lorsqu'un client et un serveur TLS commencent à communiquer pour la première fois, ils se mettent d'accord sur une version de protocole, choisissent des algorithmes cryptographiques, s'authentifient mutuellement en option, et utilisent des techniques de chiffrement à clés publiques pour générer des secrets partagés.

Le protocole de prise de contact TLS englobe les étapes suivantes :

- Échange des messages hello pour l'accord sur les algorithmes, échange des valeurs aléatoires, et vérification de reprise de session.
- Échange des paramètres cryptographiques nécessaires pour permettre au client et serveur de s'accorder sur un modèle initial de secret.
- Échanger des certificats et des informations cryptographiques pour permettre l'authentification mutuelle du client et du serveur.
- Générer un secret maître à partir du modèle initial de secret et échanger des valeurs aléatoires.
- Fournir des 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 falsification par un attaquant.

Noter que les couches supérieures ne devraient pas trop se fier au fait que TLS négocie toujours la connexion la plus forte possible entre deux homologues. Il y a un certain nombre de façons dont une attaque par interposition peut 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 encore des attaques disponibles : par exemple, un attaquant pourrait bloquer l'accès sur lequel fonctionne un service sécurisé, ou tenter d'obtenir de l'homologue qu'il négocie une connexion non authentifiée. La règle fondamentale est que les niveaux supérieurs doivent être conscients de ce que sont leurs exigences de sécurité et ne jamais transmettre des informations sur un canal moins sûr que ce qu'ils exigent. Le protocole TLS est sûr dans la mesure où toute suite de chiffrement offre le niveau de sécurité qu'elle promet : si on négocie 3DES avec un échange de clés RSA à 1024 bits avec un hôte dont on a vérifié le certificat, on peut s'attendre à être en sécurité.

Ces objectifs sont réalisés par le protocole de prise de contact, qui peut se résumer comme suit : le client envoie un message ClientHello auquel le serveur doit répondre par un message ServerHello, sinon une erreur fatale surviendra et la connexion sera en échec. Le ClientHello et le ServerHello sont utilisés pour établir des capacités d'amélioration de la sécurité entre client et serveur. Le ClientHello et ServerHello établissent les attributs suivants : Version de 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 de serveur, le ServerKeyExchange, le certificat de client, et le ClientKeyExchange. De nouvelles méthodes d'échange de clés peuvent être créées en spécifiant un format pour ces messages et en définissant l'utilisation des messages pour permettre au client et serveur de s'accorder sur un secret partagé. Ce secret DOIT être assez long ; Les méthodes d'échange de clés actuellement définies échangent des secrets qui font 46 octets et plus.

À la suite des messages hello, le serveur va envoyer son certificat dans un message Certificate s'il doit être authentifié. De plus, un message ServerKeyExchange peut être envoyé, si c'est exigé (par exemple, si le serveur n'a pas de certificat, ou si son certificat est seulement pour signature). Si le serveur est authentifié, il peut demander un certificat au client, si c'est approprié pour la suite de chiffrement choisie. Ensuite, le serveur va envoyer le message ServerHelloDone, qui indique que la phase message hello de la prise de contact est achevée. Le serveur va ensuite attendre une réponse du client. Si le serveur a envoyé un message CertificateRequest, le client DOIT envoyer le message Certificate. Le message ClientKeyExchange est maintenant envoyé et le contenu de ce message va dépendre de l'algorithme de clé publique choisi entre le ClientHello et le ServerHello. Si le client a envoyé un certificat avec une capacité de signature, un message CertificateVerify à signature numérique est envoyé pour vérifier explicitement la possession de la clé privée dans le certificat.

À ce point, un message ChangeCipherSpec est envoyé par le client, et le client copie la spécification de chiffrement en instance *(Cipher Spec)* dans la CipherSpec en cours. Le client envoie alors immédiatement le message Terminé sous les nouveaux algorithmes, clés, et secrets. En réponse, le serveur va envoyer son propre message ChangeCipherSpec, transférer la CipherSpec en instance en en cours, et envoyer son message Terminé sous la nouvelle CipherSpec. À ce point, la prise de contact est achevée, et client et serveur peuvent commencer à échanger des données de couche application. (Voir le diagramme des flux ci-dessous.) Les données d'application NE DOIVENT PAS être envoyées avant l'achèvement de la première prise de contact (avant que soit établie une suite de chiffrement autre que TLS_NULL_WITH_NULL_NULL).

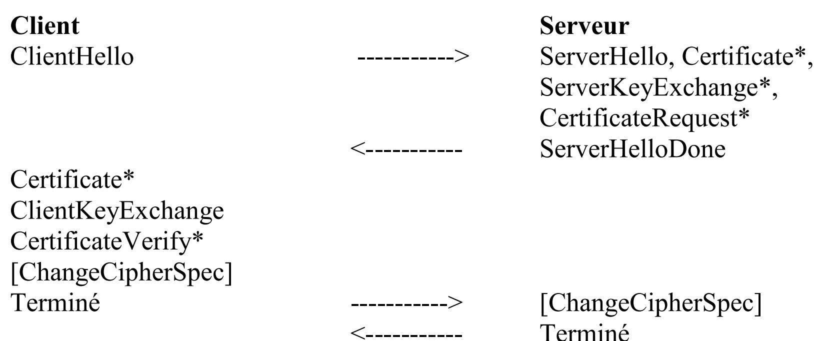

Figure 1 : Flux de messages pour une prise de contact complète

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

Note : Pour aider à éviter les calages de la conduite, ChangeCipherSpec est un type de contenu de protocole indépendant de TLS, et n'est pas réellement un message de prise de contact TLS.

Lorsque le client et le 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 comme suit :

Le client envoie un ClientHello en utilisant l'identifiant de session de la session à reprendre. Le serveur vérifie alors la correspondance de sa mémoire cache de session. Si il trouve une correspondance, et si le serveur veut rétablir la connexion sous l'état de session spécifié, il va envoyer un ServerHello avec la même valeur d'identifiant de session. À ce point, le client et le serveur DOIVENT tous deux envoyer des messages ChangeCipherSpec et procéder directement aux messages Terminé. Une fois le rétablissement achevé, le client et le 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 le serveur TLS effectuent une prise de contact complète.

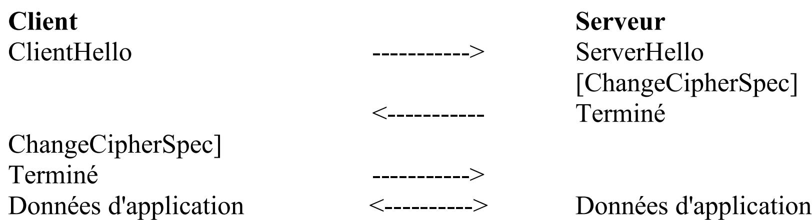

Figure 2 : Flux de messages pour une prise de contact abrégée

Le contenu et la signification de chaque message sera présenté en détail dans les paragraphes suivants.

7.4 Protocole de prise de contact

Le protocole de prise de contact TLS est un des clients de niveau supérieur définis dans le protocole d'enregistrement TLS .Ce protocole est utilisé pour négocier les attributs sécurisés d'une session. Les messages de prise de contact sont fournis à la couche d'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 active en cours.

```
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 où ils DOIVENT être envoyés ; l'envoi des messages de prise de contact dans un ordre non attendu résulte en une erreur fatale. Les messages de prise de contact non nécessaires peuvent cependant être omis. Noter une exception à l'ordre de présentation : le message Certificate est utilisé deux fois dans la prise de contact (du serveur au client, puis du client au serveur), mais décrit seulement dans sa première position. Le seul message qui n'est pas lié par ces règles d'ordre est le message HelloRequest, 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.

Les nouveaux types de message de prise de contact sont alloués par l'IANA comme décrit à la Section 12.

7.4.1. Messages Hello

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

7.4.1.1 Demande Hello

Quand est envoyé ce message :
Le message HelloRequest PEUT être envoyé par le serveur à tout moment.

Signification du message :
HelloRequest est une simple notification que le client devrait recommencer le processus de négociation. En réponse, le client devrait envoyer un message ClientHello lorsque cela convient. Ce message n'est pas destiné à établir quel côté est le client ou le serveur mais simplement à initier une nouvelle négociation. Les serveurs NE DEVRAIENT PAS envoyer de HelloRequest immédiatement à la connexion initiale du client. C'est au client d'envoyer un ClientHello à ce moment.

Ce message sera ignoré par le client si il est déjà 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 préséance 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 HelloRequest mais ne reçoit pas de ClientHello en réponse, il peut clore la connexion avec une alerte fatale.

Après l'envoi d'une HelloRequest, les serveurs NE DEVRAIENT PAS répéter la demande avant l'achèvement de la négociation de prise de contact qui s'ensuit.

Structure de ce message :
struct { } HelloRequest;

Ce message NE DOIT PAS être inclus dans les hachages de message qui sont maintenus tout au long de 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 est envoyé ce message :
Lorsque un client se connecte pour la première fois à un serveur, il est obligé d'envoyer le ClientHello comme premier message. Le client peut aussi envoyer un ClientHello en réponse à une HelloRequest ou on de sa propre initiative afin de renégocier les paramètres de sécurité dans une connexion existante.

Structure de ce message :
Le message ClientHello comporte une structure aléatoire, qui sera utilisée plus tard dans le protocole.

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

gmt_unix_time
La date et l'heure courantes en format standard UNIX de 32 bits (les secondes depuis le 1er janvier 1970 à minuit, en UTC, en ignorant les secondes d'ajustement) conformément à 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. Noter que, pour des raisons historiques, l'élément de données est dénommé à l'aide de GMT *(heure de Greenwich)*, le prédécesseur de la base horaire mondiale actuelle, UTC *(temps universel)*.

random_bytes
28 octets générés par un générateur sûr de nombres aléatoires.

Le message ClientHello comporte un identifiant de session de longueur variable. S'il n'est pas vide, sa valeur identifie une session dont le client souhaite réutiliser les paramètres de sécurité entre le même client et serveur. L'identifiant de session PEUT provenir d'une connexion précédente, 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 les valeurs dérivées d'une connexion, et la troisième option rend possible l'établissement de plusieurs connexions sûres indépendantes sans répéter tout le protocole de prise de contact. Ces connexions indépendantes peuvent survenir en séquence ou simultanément ; un SessionID devient valide lorsque la négociation de la prise de contact est achevée avec l'échange des messages Terminé et persiste jusqu'à ce qu'il soit retiré à cause de son vieillissement ou à cause d'une erreur fatale rencontrée sur une connexion associée à la session. Le contenu réel du SessionID est défini par le serveur.

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

Avertissement : Comme le SessionID est transmis sans chiffrement ou protection 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 d'atteinte à la sécurité. (Noter que le contenu de la prise de contact globale, y compris le SessionID, est protégé par les messages Terminé échangés à la fin de la prise de contact.)

La liste des suites de chiffrement, passée du client au serveur dans le message ClientHello, contient les combinaisons des algorithmes cryptographiques pris en charge par le client dans l'ordre des préférences du client (le favori en premier). Chaque suite de chiffrement définit un algorithme d'échange de clés, un algorithme de chiffrement brut (incluant la longueur de la clé secrète), un algorithme de MAC, et une PRF. Le serveur va choisir une suite de chiffrement ou, si aucun choix acceptable n'est présenté, retourner une alerte d'échec de prise de contact et clore la connexion. Si la liste contient des suites de chiffrement que le serveur ne reconnaît pas, ne prend pas en charge, ou ne souhaite pas utiliser, le serveur DOIT ignorer ces suites de chiffrement, et traiter celles qui restent comme d'ordinaire.

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

Le ClientHello comporte une liste des algorithmes de compression pris en charge par le client, ordonnés selon la préférence du client.

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

struct {
      ProtocolVersion client_version;
      Random random;
      SessionID session_id;
      CipherSuite cipher_suites<2..2^16-2>;
      CompressionMethod compression_methods<1..2^8-1>;
      select (extensions_present) {
        cas faux:
          struct {};
        cas vrai:
          Extension extensions<0..2^16-1>;
      };
} ClientHello;
```

TLS permet aux extensions de suivre le champ compression_methods dans un bloc d'extensions. La présence des extensions peut être détectée en déterminant si il y a des octets qui suivent compression_methods à la fin de ClientHello. Noter que cette méthode de détection des données facultatives diffère de la méthode normale de TLS qui est d'avoir un champ de longueur variable, mais elle est utilisée pour être compatible avec le TLS d'avant la définition des extensions.

client_version
Version du protocole TLS par laquelle le client souhaite communiquer durant cette session. Cela DEVRAIT être la dernière (plus haute valeur) version acceptée par le client. Pour la présente version de la spécification, la version sera 3.3 (voir à l'Appendice E les détails sur la rétro compatibilité).

random

Structure aléatoire générée par le client.

session_id
Identifiant de session que le client souhaite utiliser pour cette connexion. Ce champ est vide si aucun session_id n'est disponible, ou si le client souhaite générer de nouveaux paramètres de sécurité.

cipher_suites
C'est une liste des options cryptographiques acceptées par le client, avec la préférence du client en premier. Si le champ session_id n'est pas vide (ce qui implique une demande de reprise de session), ce vecteur DOIT inclure au moins la cipher_suite de cette session là. Les valeurs sont définies à l'Appendice A.5.

compression_methods
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), elle DOIT inclure la compression_method tirée de cette session. Ce vecteur DOIT contenir, et toutes les mises en œuvre DOIVENT accepter, CompressionMethod.null. Et donc, un client et un serveur seront toujours capables de se mettre d'accord sur une méthode de compression.

extensions
Les clients PEUVENT demander des fonctionnalités étendues aux serveurs par l'envoi de données dans le champ extensions. Le format réel de "Extension" est défini au paragraphe 7.4.1.4.

Au cas où un client demande des fonctionnalités supplémentaires en utilisant extensions, et où cette fonctionnalité n'est pas fournie par le serveur, le client PEUT interrompre la prise de contact. Un serveur DOIT accepter les messages ClientHello avec et sans le champ extensions, et (comme pour tous les autres messages) il DOIT vérifier que la quantité de données dans le message correspond précisément à un de ces formats ; sinon, il DOIT alors envoyer une alerte fatale "decode_error".

Après l'envoi du message ClientHello, le client attend un message ServerHello. Tout message de prise de contact retourné par le serveur, sauf une HelloRequest, est traité comme une erreur fatale.

7.4.1.3 Hello de serveur

Quand est envoyé ce message :
Le serveur va envoyer ce message en réponse à un message ClientHello lorsqu'il a été capable de trouver 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;
      select (extensions_present) {
        cas faux:
          struct {};
        cas vrai:
          Extension extensions<0..2^16-1>;
      };
} ServerHello;
```

La présence des extensions peut être détectée en déterminant si il y a des octets à la suite du champ compression_method à la fin du ServerHello.

server_version
Ce champ va contenir la plus basse de celles suggérées par le client dans le hello de client et la plus forte acceptée par le serveur. Pour la présente version de la spécification, la version est 3.3. (Voir à l'Appendice E les détails sur la rétro compatibilité.)

random
Cette structure est générée par le serveur et DOIT être générée indépendamment du ClientHello.random.

session_id
C'est l'identité de la session correspondant à cette connexion. Si le ClientHello.session_id était non vide, le serveur va chercher une correspondance dans sa mémoire cache de session. Si une correspondance est trouvée et si le serveur est d'accord pour établir la nouvelle connexion en utilisant l'état de session spécifié, le serveur va répondre avec la même valeur que celle fournie par le client. Cela indique une reprise de session et impose que les parties passent directement aux messages Terminé. Autrement, ce champ contiendra 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 mémoire cache et donc qu'elle ne pourra pas être reprise. Si une session est reprise, elle doit l'être en utilisant la même suite de chiffrement avec laquelle elle a été négociée à l'origine. Noter qu'il n'est pas exigé que le serveur reprenne une session même si il avait précédemment fourni un session_id. Les clients DOIVENT être prêts à effectuer une négociation complète – y compris de négocier de nouvelles suites de chiffrement – durant toute prise de contact.

cipher_suite
La seule suite de chiffrement choisie par le serveur dans la liste de ClientHello.cipher_suites. Pour les reprises de sessions, ce champ est de la valeur de l'état de la session reprise.

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

extensions
Liste des extensions. Noter que seules les extensions offertes par le client peuvent apparaître dans la liste du serveur.

7.4.1.4. Extensions à Hello

Le format d'extension est :

```
struct {
      ExtensionType extension_type;
      extension_data<0..2^16-1> opaque ;
} Extension;

enum {
      signature_algorithms(13), (65535)
} ExtensionType;
```

Ici :
- "extension_type" identifie le type d'extension particulier.
- "extension_data" contient des informations spécifiques du type particulier d'extension.

L'ensemble initial d'extensions est défini dans un document voisin [TLSEXT]. La liste des types d'extensions est tenue par l'IANA comme indiqué à la Section 12.

Un type d'extension NE DOIT PAS apparaître dans le ServerHello à moins que le même type d'extension n'apparaisse dans le ClientHello correspondant. Si un client reçoit dans un ServerHello un type d'extension qu'il n'a pas demandé dans le ClientHello associé, il DOIT interrompre la prise de contact avec une alerte fatale unsupported_extension.

Néanmoins, les extensions "orientées serveur" pourront être fournies à l'avenir dans ce cadre. Une telle extension (disons, de type x) exigerait que le client envoie d'abord une extension du type x dans un ClientHello avec extension_data vide pour indiquer qu'il accepte le type d'extension. Dans ce cas, le client offre la capacité de comprendre le type d'extension, et le serveur rebondit sur l'offre du client.

Lorsque plusieurs extensions de différents types sont présentes dans les messages ClientHello ou ServerHello, les extensions PEUVENT apparaître dans n'importe quel ordre. Il NE DOIT PAS y avoir plus d'une extension du même type.

Finalement, noter que les extensions peuvent être envoyées aussi bien en commençant une nouvelle session qu'en demandant la reprise d'une session. Bien sûr, un client qui demande la reprise d'une session ne sait en général pas si le serveur va accepter cette demande, et donc, il DEVRAIT envoyer les mêmes extensions que s'il ne tentait pas la reprise.

En général, la spécification de chaque type d'extension doit décrire l'effet de l'extension à la fois durant une prise de contact complète et une reprise de session. La plupart des extensions TLS courantes ne sont pertinentes que lors de l'initialisation d'une session : lorsqu'une session plus ancienne est reprise, le serveur ne traite pas ces extensions dans le Client Hello, et ne les inclut pas dans le Server Hello. Cependant, certaines extensions peuvent spécifier des comportements différents durant la reprise de

session.

Il y a des interactions subtiles (et pas si subtiles) qui peuvent survenir dans ce protocole entre des caractéristiques nouvelles et des caractéristiques existantes qui peuvent avoir pour résultat une réduction significative de la sécurité globale. Les considérations suivantes devraient être prises en compte lors de la conception de nouvelles extensions :

- Certains cas où un serveur n'accepte pas une extension sont des conditions d'erreur, et certains sont simplement des refus de prendre en charge des caractéristiques particulières. En général, les alertes d'erreur devraient être utilisées pour les premiers, et un champ dans la réponse d'extension du serveur pour les derniers.

- Les extensions devraient, autant que possible, être conçues pour empêcher toute attaque qui force l'usage (ou le non-usage) d'une caractéristique particulière par la manipulation des messages de prise de contact. Ce principe devrait être suivi sans considération du fait qu'on pense que la caractéristique pourrait causer un problème pour la sécurité.

Le fait que les champs d'extension soient inclus dans les entrées des hachages du message Terminé sera souvent suffisant, mais un soin extrême est nécessaire lorsque l'extension change la signification des messages envoyés dans la phase de prise de contact. Les concepteurs et les mises en œuvre doivent être conscients du fait que jusqu'à ce que la prise de contact ait été authentifiée, des attaquants actifs peuvent modifier les messages et insérer, retirer, ou remplacer les extensions.

- Il serait techniquement possible d'utiliser les extensions pour changer les aspects majeurs de la conception de TLS ; par exemple le concept de négociation de suite de chiffrement. Cela n'est pas recommandé ; il serait plus approprié de définir une nouvelle version de TLS – en particulier parce que les algorithmes de prise de contact de TLS ont une protection spécifique contre les attaques de régression de version fondées sur le numéro de version, et la possibilité de régression de version devrait être une considération significative dans tout changement de conception majeur.

7.4.1.4.1 Algorithmes de signature

Le client utilise l'extension "signature_algorithms" pour indiquer au serveur quelles paires signature/hachage peuvent être utilisées dans les signatures numériques. Le champ "extension_data" de cette extension contient une valeur "supported_signature_algorithms".

```
enum {
        none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),
        sha512(6), (255)
} HashAlgorithm;

enum { anonymous(0), rsa(1), dsa(2), ecdsa(3), (255) }
        SignatureAlgorithm;

struct {
        hachage HashAlgorithm ;
        signature SignatureAlgorithm ;
        } SignatureAndHashAlgorithm;

SignatureAndHashAlgorithm
        supported_signature_algorithms<2..2^16-2>;
```

Chaque valeur SignatureAndHashAlgorithm donne une seule paire hachage/signature que le client veut vérifier. Les valeurs sont indiquées en ordre de préférence décroissante.

Note : Comme tous les algorithmes de signature et de hachage peuvent n'être pas acceptés par une mise en œuvre (par exemple, DSA avec SHA-1, mais pas avec SHA-256), les algorithmes sont énumérés en paires.

hash
Ce champ indique l'algorithme de hachage qui peut être utilisé. Les valeurs indiquent la prise en charge des données non hachées, MD5 [MD5], SHA-1, SHA-224, SHA-256, SHA-384, et SHA-512 [SHS], respectivement. La valeur "aucune" est fournie pour les possibilités d'extension à venir, en cas d'algorithme de signature qui n'exigerait pas de hachage avant la signature.

signature
Ce champ indique l'algorithme de signature qui peut être utilisé. Les valeurs indiquent les signatures anonymes, RSASSA-PKCS1-v1_5 [PKCS1] et DSA [DSS], et ECDSA [ECDSA], respectivement. La valeur "anonyme" est sans signification dans ce contexte mais utilisée au paragraphe 7.4.3. Elle NE DOIT PAS apparaître dans cette extension.

La sémantique de cette extension est assez compliquée parce que la suite de chiffrement indique des algorithmes de signature permis mais pas d'algorithmes de hachage. Les paragraphes 7.4.2 et 7.4.3 décrivent les règles appropriées.

Si le client accepte seulement les algorithmes de hachage et de signature par défaut (dont la liste figure dans cette section), il PEUT omettre l'extension signature_algorithms. Si le client n'accepte pas les algorithmes par défaut, ou accepte d'autres algorithmes de hachage et de signature (et s'il veut les utiliser pour vérifier les messages envoyés par le serveur, c'est-à-dire, les certificats de serveur et les échanges de clés de serveur), il DOIT envoyer l'extension signature_algorithms, faisant la liste des algorithmes qu'il accepte.

Si le client n'envoie pas l'extension signature_algorithms, le serveur DOIT faire ce qui suit :

- Si l'algorithme d'échange de clés négocié est un de (RSA, DHE_RSA, DH_RSA, RSA_PSK, ECDH_RSA, ECDHE_RSA), se comporter comme si le client avait envoyé la valeur {sha1,rsa}.

- Si l'algorithme d'échange de clés négocié est un de (DHE_DSS, DH_DSS), se comporter comme si le client avait envoyé la valeur {sha1,dsa}.

- Si l'algorithme d'échange de clés négocié est un de (ECDH_ECDSA, ECDHE_ECDSA), se comporter comme si le client avait envoyé la valeur {sha1,ecdsa}.

Note : ceci est un changement par rapport à TLS 1.1 où il n'y avait pas de règles explicites, mais en pratique on peut supposer que l'homologue accepte MD5 et SHA-1.

Note : cette extension n'est pas significative pour les versions de TLS antérieures à 1.2. Les clients NE DOIVENT PAS l'offrir si ils ne proposent pas les versions antérieures. Cependant, même si les clients ne l'offrent pas, les règles spécifiées dans [TLSEXT] exigent que les serveurs ignorent les extensions qu'ils ne comprennent pas. Les serveurs NE DOIVENT PAS envoyer cette extension. Les serveurs TLS DOIVENT accepter de recevoir cette extension.

Quand on effectue une reprise de session, cette extension n'est pas incluse dans le Hello de serveur, et le serveur ignore l'extension dans le Hello du client (si elle est présenté).

7.4.2 Certificat de serveur

Quand est envoyé de message :
Le serveur DOIT envoyer un message Certificate chaque fois que la méthode d'échange de clés acceptée utilise des certificats pour l'authentification (ceci inclut toutes les méthodes d'échange de clés définies dans ce document excepté DH_anon). Ce message suivra toujours immédiatement le message ServerHello.

Signification de ce message :
Ce message convoie la chaîne de certificats du serveur au client.

Le certificat DOIT être approprié pour l'algorithme d'échange de clés de la suite de chiffrement négociée et toutes extensions négociées.

Structure de ce message :

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

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

certificate_list
C'est une séquence (chaîne) de certificats. Le certificat de l'envoyeur DOIT être le premier de la liste. Chaque certificat suivant DOIT directement certifier 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 de certificat racine PEUT être omis de la chaîne, dans l'hypothèse où l'extrémité distante doit déjà le posséder afin de le valider dans tous les cas.

Les mêmes type et structure 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 pour le vecteur du certificat parce que les certificats étendus PKCS n° 6 [PKCS6] ne sont pas utilisés. De même, PKCS n° 7 définit un ENSEMBLE plutôt qu'une SEQUENCE, rendant la tâche d'analyse de la liste plus difficile.

Les règles suivantes s'appliquent aux certificats envoyés par le serveur :

- Le type de certificat DOIT être X.509v3, sauf négociation explicite contraire (par exemple, [TLSPGP]).

- La clé publique du certificat de l'entité terminale (et les restrictions qui lui sont associées) DOIT être compatible avec l'algorithme d'échange de clés choisi.

Algorithme d'échange de clé	**Type de clé de certificat**
RSA RSA_PSK	Clé publique RSA ; le certificat DOIT permettre d'utiliser la clé pour le chiffrement (le bit keyEncipherment DOIT être mis si l'extension d'usage de la clé est présente). Note : RSA_PSK est défini dans [TLSPSK].
DHE_RSA ECDHE_RSA	Clé publique RSA ; le certificat DOIT permettre d'utilise la clé pour signer (le bit digitalSignature DOIT être mis si l'extension d'usage de la clé est présente) avec le schéma de signature et l'algorithme de hachage qui seront employés dans le message d'échange de clés du serveur. Note : ECDHE_RSA est défini dans [TLSECC].
DHE_DSS	Clé publique DSA ; le certificat DOIT permettre l'utilisation de la clé pour signer avec l'algorithme de hachage qui sera employé dans le message d'échange de clés du serveur.
DH_DSS DH_RSA	Clé publique Diffie-Hellman ; le bit keyAgreement DOIT être mis si l'extension d'usage de la clé est présente.
ECDH_ECDSA ECDH_RSA	Clé publique à capacité ECDH ; la clé publique DOIT utiliser un format de courbe et point accepté par le client, comme décrit dans [TLSECC].
ECDHE_ECDSA	Clé publique à capacité ECDSA ; le certificat DOIT permettre d'utiliser la clé pour signer avec l'algorithme de hachage qui sera employé dans le message d'échange de clés du serveur. La clé publique DOIT utiliser un format de courbe et point accepté par le client, comme décrit dans [TLSECC].

- Les extensions "server_name" et "trusted_ca_keys" [TLSEXT] sont utilisées pour guider le choix des certificats.

Si le client a fourni une extension "signature_algorithms", tous les certificats fournis alors par le serveur DOIVENT être signés par une paire d'algorithmes hachage/signature qui apparaît dans cette extension. Noter que cela implique qu'un certificat contenant une clé pour un algorithme de signature PEUT être signé en utilisant un algorithme de signature différent (par exemple, une clé RSA signée avec une clé DSA). Ceci est différent de TLS 1.1, qui exigeait que les algorithmes soient les mêmes. Noter que cela implique aussi que les algorithmes d'échange de clés DH_DSS, DH_RSA, ECDH_ECDSA, et ECDH_RSA ne restreignent pas les algorithmes à utiliser pour signer le certificat. Des certificats DH fixés PEUVENT être signés avec toute paire d'algorithme hachage/signature qui apparaît dans l'extension. Les noms DH_DSS, DH_RSA, ECDH_ECDSA, et ECDH_RSA sont historiques.

Si le serveur a plusieurs certificats, il en choisit un sur la base des critères susmentionnés (en plus des autres critères, tels que le point d'extrémité de couche transport, la configuration et les préférences locales, etc.). Si le serveur a un seul certificat, il DEVRAIT tenter de valider celui qui satisfait à ces critères.

Noter qu'il y a des certificats qui utilisent des algorithmes et/ou des combinaisons d'algorithmes qui ne peuvent pas être actuellement utilisés avec TLS. Par exemple, un certificat avec une clé de signature RSASSA-PSS (id-RSASSA-PSS OID dans SubjectPublicKeyInfo) ne peut pas être utilisé parce que TLS ne définit pas d'algorithme de signature correspondant.

Lorsque des suites de chiffrement qui spécifient de nouvelles méthodes d'échange de clés seront spécifiées pour le protocole TLS, elles impliqueront le format du certificat et les informations exigées pour le codage de clé.

7.4.3 Message d'échange de clés du serveur

Quand envoyer ce message :
Ce message sera envoyé immédiatement après le message Certificat du serveur (ou du message ServerHello, si c'est une négociation anonyme).

Le message ServerKeyExchange n'est envoyé par le serveur que lorsque le message Certificat du serveur (s'il est envoyé) ne contient pas assez de données pour permettre au client d'échanger un modèle initial de secret. Ceci est vrai pour les méthodes d'échange de clés suivantes :
DHE_DSS
DHE_RSA

DH_anon

Il n'est pas légal d'envoyer le message ServerKeyExchange pour les méthodes d'échange de clés suivantes :
RSA
DH_DSS
DH_RSA

Les autres algorithmes d'échange de clés, tels que ceux définis dans [TLSECC], DOIVENT spécifier si le message ServerKeyExchange est envoyé ou non ; et si le message est envoyé, son contenu.

Signification de ces message :
Ce message porte des informations cryptographiques pour permettre au client de communiquer le modèle initial de secret : une clé publique Diffie-Hellman avec laquelle le client peut achever un échange de clés (dont le résultat sera le modèle initial de secret) ou une clé publique pour certains autres algorithmes.

Structure de ce message :

```
enum { dhe_dss, dhe_rsa, dh_anon, rsa, dh_dss, dh_rsa     /* peut être étendue, par exemple, pour ECDH -- voir [TLSECC] */
       } KeyExchangeAlgorithm;

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
Le module de nombre premier utilisé pour l'opération Diffie-Hellman.

dh_g
Le générateur utilisé pour l'opération Diffie-Hellman.

dh_Ys
La valeur publique Diffie-Hellman du serveur (g^X mod p).

```
struct {
       select (KeyExchangeAlgorithm) {
         cas dh_anon:
           paramètres ServerDHParams;
         cas dhe_dss:
         cas dhe_rsa:
           paramètres ServerDHParams ;
           structure digitally-signed {
             client_random[32] opaque ;
             server_random[32] opaque ;
             paramètres ServerDHParams;
           } signed_params;
         cas rsa:
         cas dh_dss:
         cas dh_rsa:
           struct {} ;
/* le message est omis pour rsa, dh_dss, et dh_rsa */
/* il peut être étendu, par exemple, pour ECDH -- voir [TLSECC] */
       };
} ServerKeyExchange;
```

paramètres
Les paramètres d'échange de clés du serveur.

signed_params
Pour les échanges de clés non anonymes, une signature sur les paramètres d'échange de clés du serveur.

Si le client a offert l'extension "signature_algorithms", l'algorithme de signature et l'algorithme de hachage DOIVENT être une

paire figurant sur la liste de cette extension. Noter qu'il y a ici une possibilité d'incohérence. Par exemple, le client pourrait offrir l'échange de clés DHE_DSS mais omettre les paires DSA dans son extension "signature_algorithms". Afin de négocier correctement, le serveur DOIT vérifier toutes les suites de chiffrement candidates par rapport à l'extension "signature_algorithms" avant de faire son choix. Cela est assez inélégant mais c'est un compromis conçu pour minimiser les changements de la suite de chiffrement conçue à l'origine.

De plus, les algorithmes de hachage et de signature DOIVENT être compatibles avec la clé dans le certificat d'entité terminale du serveur. Les clés RSA PEUVENT être utilisées avec tout algorithme de hachage permis, sous réserve des restrictions du certificat, s'il en est.

Parce que les signatures DSA ne contiennent aucune indication sûre d'algorithme de hachage, il y a un risque de substitution de hachage si plusieurs hachages peuvent être utilisés avec une clé quelconque. Actuellement, DSA [DSS] ne peut être utilisé qu'avec SHA-1. Des révisions futures de DSS [DSS-3] sont attendues pour permettre l'utilisation d'autres algorithmes de résumé avec DSA, ainsi que des lignes directrices sur les algorithmes de hachage qui devraient être utilisés avec chaque taille de clé. De plus, des révisions futures de [PKIX] pourraient spécifier des mécanismes pour que les certificats indiquent quels algorithmes de résumé sont à utiliser avec DSA.

Comme des suites de chiffrement supplémentaires sont définies pour TLS, et qu'elles comportent 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 modèle initial de secret.

7.4.4 Demande de certificat

Quand est envoyé ce message :
Un serveur non anonyme peut facultativement demander un certificat au client, si c'est approprié pour la suite de chiffrement choisie. Ce message, s'il est envoyé, va suivre immédiatement le message ServerKeyExchange (s'il est envoyé ; autrement, ce message suit le message Certificate du serveur).

Structure de ce message :

```
enum {
        rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
        rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
        fortezza_dms_RESERVED(20), (255)
} ClientCertificateType;

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

struct {
     ClientCertificateType certificate_types<1..2^8-1>;
     SignatureAndHashAlgorithm
        supported_signature_algorithms<2^16-1>;
     DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
```

certificate_types
Liste des types de certificat que le client peut offrir.

rsa_sign	certificat contenant une clé RSA
dss_sign	certificat contenant une clé DSA
rsa_fixed_dh	certificat contenant une clé DH statique
dss_fixed_dh	certificat contenant une clé DH statique

supported_signature_algorithms
Liste des paires d'algorithmes hachage/signature que le serveur est capable de vérifier, en ordre de préférence décroissante.

certificate_authorities
Liste des noms distinctifs [X501] des certificate_authorities acceptables, représentés en format codé en DER. 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 pour décrire des racines connues aussi bien qu'un espace d'autorisation désiré. Si la liste certificate_authorities est vide, le client PEUT alors envoyer n'importe quel certificat du ClientCertificateType approprié, sauf s'il existe un arrangement externe qui s'y oppose.

L'interaction des champs certificate_types et supported_signature_algorithms est assez compliquée. Les certificate_types ont

été présents dans TLS depuis SSLv3, mais étaient quelque peu sous spécifiés. Beaucoup de leur fonctionnalité est remplacée par supported_signature_algorithms. Les règles suivantes s'appliquent :

- Tout certificat fourni par le client DOIT être signé en utilisant une paire d'algorithmes hachage/signature trouvée dans supported_signature_algorithms.

- Le certificat d'entité d'extrémité fourni par le client DOIT contenir une clé compatible avec certificate_types. Si la clé est une clé de signature, elle DOIT être utilisable avec une paire d'algorithmes hachage/signature dans supported_signature_algorithms.

- Pour des raisons historiques, les noms de certains types de certificat de client comportent l'algorithme utilisé pour signer le certificat. Par exemple, dans les premières versions de TLS, rsa_fixed_dh signifiait un certificat signé avec RSA et contenant une clé DH statique. Dans TLS 1.2, cette fonctionnalité a été rendue obsolète par supported_signature_algorithms, et le type de certificat ne restreint plus les algorithmes utilisés pour signer le certificat. Par exemple, si le serveur envoie le type de certificat dss_fixed_dh et les types de signature {{sha1, dsa}, {sha1, rsa}}, le client PEUT répondre avec un certificat contenant une clé DH statique, signée avec RSA-SHA1.

De nouvelles valeurs de ClientCertificateType sont allouées par l'IANA comme indiqué à la Section 12.

Note : Il n'est pas permis d'utiliser les valeurs marquées RÉSERVÉ. Elles ont été utilisées dans SSLv3.

Note : La demande d'authentification d'un client par un serveur anonyme est une alerte handshake_failure fatale.

7.4.5 Hello Done du serveur

Quand est envoyé ce message :
Le message ServerHelloDone est envoyé par le serveur pour indiquer la fin du ServerHello et des messages associés. Après l'envoi de ce message, le serveur va attendre une réponse du client.

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

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

Structure de ce message :

struct { } ServerHelloDone;

7.4.6 Certificat du client

Quand est envoyé ce message :
C'est le premier message que le client peut envoyer après avoir reçu un message ServerHelloDone. Ce message n'est envoyé que si le serveur demande un certificat. Si aucun certificat convenable n'est disponible, le client DOIT envoyer un message "certificate" ne contenant aucun certificat. C'est à dire que la structure certificate_list a une longueur de zéro. Si le client n'envoie aucun certificat, le serveur PEUT à sa discrétion soit continuer la prise de contact sans authentification du client, soit répondre par une alerte fatale handshake_failure. Aussi, si certains aspects de la chaîne de certificat ne sont pas acceptables (par exemple, il n'est pas signé par un CA connu, de confiance), le serveur PEUT à sa discrétion soit continuer la prise de contact (en considérant le client comme non authentifié) soit envoyer une alerte fatale.

Les certificats de client sont envoyés en utilisant la structure de certificat définie au paragraphe 7.4.2.

Signification de ce message :
Ce message porte la chaîne de certificats du client au serveur ; le serveur va l'utiliser lorsqu'il vérifie le message CertificateVerify (lorsque l'authentification du client est fondée sur la signature) ou en calculant le modèle initial de secret (pour le Diffie-Hellman non éphémère). Le certificat DOIT être approprié pour l'algorithme d'échange de clés de la suite de chiffrement négociée, et toutes les extensions négociées.

En particulier :
- Le type de certificat DOIT être X.509v3, sauf négociation explicite contraire (par exemple, [TLSPGP]).
- La clé publique de bout en bout du certificat (et les restrictions qui y sont associées) doit être compatible avec les types de

certificat énumérés dans CertificateRequest:

Type de certificat client	Type de clé de certificat
rsa_sign	Clé publique RSA ; le certificat DOIT permettre que la clé soit utilisée pour signer avec le schéma de signature et l'algorithme de hachage qui sera employé dans le message de vérification du certificat.
dss_sign	Clé publique DSA ; le certificat DOIT permettre que la clé soit utilisée pour signer avec l'algorithme de hachage qui sera employé dans le message de vérification du certificat.
ecdsa_sign	Clé publique capable de ECDSA ; le certificat DOIT permettre que la clé soit utilisée pour signer avec l'algorithme de hachage qui sera employé dans le message de vérification du certificat ; la clé publique DOIT utiliser un format de courbe et point accepté par le serveur.
rsa_fixed_dh dss_fixed_dh	Clé publique Diffie-Hellman ; DOIT utiliser le même paramètre que la clé du serveur.
rsa_fixed_ecdh ecdsa_fixed_ecdh	Clé publique à capacité ECDH ; DOIT utiliser la même courbe que la clé du serveur, et DOIT utiliser un format de point accepté par le serveur.

- Si la liste certificate_authorities dans le message demande de certificat était non vide, un des certificats de la chaîne de certificats DEVRAIT être produit par une des CA énumérées.

- Les certificats DOIVENT être signés en utilisant une paire d'algorithmes hachage/signature acceptable, comme décrit au paragraphe 7.4.4. Noter que cela relâche les contraintes sur les algorithmes de signature de certificat qui se trouvaient dans les précédentes versions de TLS.

Noter que, comme avec le certificat de serveur, il y a des certificats qui utilisent des combinaisons d'algorithmes/algorithme qui ne peuvent pas être actuellement utilisées avec TLS.

7.4.7 Message d'échange de clés de client

Quand est envoyé ce message :
Ce message est toujours envoyé par le client. Il DOIT immédiatement suivre le message de certificat de client, si il est envoyé. Autrement, il DOIT être le premier message envoyé par le client après qu'il a reçu le message ServerHelloDone.

Signification de ce message :
Avec ce message, le modèle initial de secret est établi, soit par transmission directe du secret chiffré en RSA, soit par la transmission des paramètres Diffie-Hellman qui vont permettre à chaque côté de s'accorder sur le même modèle initial de secret.

Lorsque le client utilise un exposant Diffie-Hellman éphémère, ce message contient alors la valeur publique Diffie-Hellman du client. Si le client envoie un certificat qui contient un exposant DH statique (c'est-à-dire, si il faut l'authentification client fixed_dh), ce message DOIT alors être envoyé mais DOIT être vide.

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

```
struct {
      select (KeyExchangeAlgorithm) {
          cas rsa:
              EncryptedPreMasterSecret;
          cas dhe_dss:
          cas dhe_rsa:
          cas dh_dss:
          cas dh_rsa:
          cas dh_anon:
              ClientDiffieHellmanPublic;
          } exchange_keys;
} ClientKeyExchange;
```

7.4.7.1 Message modèle initial de secret chiffré en RSA

Signification de ce message :
Si RSA est utilisé pour l'accord de clés et l'authentification, le client génère un modèle initial de secret de 48 octets, le chiffre en utilisant la clé publique tirée du certificat du serveur, et envoie le résultat dans un message modèle initial de secret chiffré.

Cette structure est une variante du message ClientKeyExchange et n'est pas un message en lui-même.

Structure de ce message :

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

client_version
La dernière (plus récente) version acceptée par le client. Ceci est utilisé pour détecter les attaques de régression de version.

random
46 octets aléatoires générés de façon sûre.

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

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

Note : Le numéro de version dans le PreMasterSecret *(modèle initial de secret)* est la version offerte par le client dans le ClientHello.client_version, et non la version négociée pour la connexion. Ce dispositif est conçu pour empêcher les attaques de régression. Malheureusement, certaines mises en œuvre anciennes utilisent à la place la version négociée, et vérifier le numéro de version peut conduire à interopérer avec de telles mises en œuvre client incorrectes.

Les mises en œuvre client DOIVENT toujours envoyer le numéro de version correct dans PreMasterSecret. Si ClientHello.client_version est TLS 1.1 ou supérieur, les mises en œuvre de serveur DOIVENT vérifier le numéro de version comme décrit dans la note ci-dessous. Si le numéro de version est TLS 1.0 ou antérieur, les mises en œuvre de serveur DEVRAIENT vérifier le numéro de version, mais PEUVENT avoir une option de configuration pour désactiver la vérification. Noter que si la vérification échoue, le PreMasterSecret DEVRAIT être rendu aléatoire de la façon décrite ci-dessous.

Note : Les attaques découvertes par Bleichenbacher [BLEI] et Klima et al. [KPR03] peuvent être utilisées pour attaquer un serveur TLS qui révèle si un message particulier, lorsqu'il est déchiffré, est correctement formaté PKCS#1, contient une structure PreMasterSecret valide, ou a le numéro de version correct.

Comme décrit par Klima [KPR03], ces faiblesses peuvent être évitées en traitant les blocs de message incorrectement formatés et/ou les discordances de numéro de version d'une manière indistingable des celle des blocs RSA correctement formatés. En d'autres termes :

1. Générer une chaîne R de 46 octets aléatoires
2. Déchiffrer le message pour récupérer le libellé M
3. Si le bourrage PKCS#1 n'est pas correct, ou si la longueur de message M n'est pas exactement 48 octets :
   pre_master_secret = ClientHello.client_version || R
   autrement Si ClientHello.client_version ≤ TLS 1.0, et vérification de numéro de version est explicitement désactivée :
   pre_master_secret = M
   autrement :
   pre_master_secret = ClientHello.client_version || M[2..47]

Noter que construire explicitement le pre_master_secret avec le ClientHello.client_version produit un master_secret invalide si le client a envoyé la mauvaise version dans le pre_master_secret d'origine.

Une autre approche est de traiter une discordance de numéro de version comme une erreur de format PKCS-1 et de rendre compètement aléatoire le modèle initial de secret :

1. Générer une chaîne R de 48 octets aléatoires
2. Déchiffrer le message pour récupérer le libellé M
3. Ici le bourrage PKCS#1 n'est pas correct, ou si la longueur du message M n'est pas exactement de 48 octets :
   pre_master_secret = R
   autrement Si ClientHello.client_version ≤ TLS 1.0, et vérification de numéro de version est explicitement désactivé
   modèle initial de secret = M
   autrement Si M[0..1] != ClientHello.client_version:
   modèle initial de secret = R
   autrement :
   modèle initial de secret = M

Bien qu'aucune attaque pratique ne soit connue contre cette construction, Klima et al. [KPR03] décrivent des attaques théoriques, et donc la première construction décrite est RECOMMANDÉE.

Dans tous les cas, un serveur TLS NE DOIT PAS générer une alerte si le traitement d'un message de modèle initial de secret chiffré en RSA échoue, ou si le numéro de version n'est pas celui attendu. Il DOIT au lieu de cela continuer la prise de contact avec un modèle initial de secret généré de façon aléatoire. Il peut être utile d'enregistrer la cause réelle de défaillance pour les besoins de dépannage ; cependant, il faut prendre soin d'éviter les fuites d'informations au profit d'un attaquant (par exemple, à travers une étude des distributions, des fichiers d'enregistrement, ou d'autres canaux.)
Le schéma de chiffrement RSAES-OAEP défini dans [PKCS1] est plus sûr contre l'attaque de Bleichenbacher. Cependant, pour une compatibilité maximale avec les versions antérieures de TLS, la présente spécification utilise le schéma de RSAES-PKCS1-v1_5. Aucune variante de l'attaque de Bleichenbacher n'est connue lorsque les recommandations suivantes ont été suivies.

Note de mise en œuvre : Les données chiffrées à l'aide d'une clé publique sont représentées comme un vecteur opaque <0..2^16-1> (voir au paragraphe 4.7). Et donc, le PreMasterSecret chiffré en RSA d'un ClientKeyExchange est précédé par deux octets de longueur. Ces octets sont redondants dans le cas de RSA parce que EncryptedPreMasterSecret sont les seules données dans ClientKeyExchange et sa longueur peut donc être déterminée sans ambiguïté. La spécification SSLv3 n'était pas claire sur le codages des données chiffrées avec une clé publique, et donc de nombreuses mises en œuvre SSLv3 ne comportent pas les octets de longueur – elles codent directement les données chiffrées en RSA dans le message ClientKeyExchange.

La présente spécification exige un codage correct du EncryptedPreMasterSecret complet avec les octets de longueur. La PDU résultante est incompatible avec de nombreuses mises en œuvre de SSLv3. Les développeurs qui mettent à jour leurs mises en œuvre de SSLv3 DOIVENT la modifier de façon à générer et accepter le codage correct. Les développeurs qui souhaitent être compatibles à la fois à SSLv3 et TLS devraient rendre le comportement de leur mise en œuvre dépendant de la version du protocole.

Note de mise en œuvre : On sait maintenant que les attaques à distance fondées sur l'étude des distributions temporelles contre TLS sont possibles, au moins lorsque client et serveur sont sur le même LAN. En conséquence, les mises en œuvre qui utilisent des clés RSA statiques DOIVENT utiliser "l'aveuglage" de RSA ou quelque autre technique capables de s'opposer à l'étude des distributions, telles que décrites dans [TIMING].

7.4.7.2 Valeur Diffie-Hellman publique du client

Signification de ce message :Cette structure porte la valeur publique (Yc) Diffie-Hellman du client si elle n'était pas déjà incluse dans le certificat du client. Le codage utilisé pour Yc est déterminé par la liste des PublicValueEncoding énumérés. Cette structure est une variante du message échange de clés du client, et n'est pas un message en lui-même.

Structure de ce message :
enum { implicit, explicit } PublicValueEncoding;

implicit
Si le client a envoyé un certificat qui contient une clé Diffie-Hellman convenable (pour l'authentification de client fixed_dh), lorsque Yc est implicite et n'a pas besoin d'être envoyé à nouveau. Dans ce cas, le message d'échange de clés client sera envoyé, mais il DOIT être vide.

explicit
Yc doit être envoyé.

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

dh_Yc
La valeur publique Diffie-Hellman du client (Yc).

7.4.8 Vérification de certificat

Quand est envoyé ce message :
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 la capacité de signature (c'est-à-dire que tous les certificats excepté ceux qui contiennent des paramètres Diffie-Hellman fixés). Lorsqu'il est envoyé, il DOIT suivre immédiatement le message d'échange de clés du client.

Structure de ce message :

```
struct {
      structure à signature numérique {
      handshake_messages[handshake_messages_length] opaque ;
      }
} CertificateVerify;
```

Ici, handshake_messages se réfère à tous les messages de prise de contact envoyés ou reçus, en commençant par le hello du client et se terminant, sans l'inclure, à ce message, y compris les champs type et longueur des messages de prise de contact. C'est l'enchaînement de toutes les structures de prise de contact (telles que définies au paragraphe 7.4) échangées jusqu'alors. Noter que cela exige que les deux côtés mettent les messages en mémoire tampon ou calculent les hachages en cours pour tous les algorithmes de hachage potentiels jusqu'au moment du calcul de CertificateVerify. Les serveurs peuvent minimiser le coût de ce calcul en offrant un ensemble restreint d'algorithmes de résumé dans le message CertificateRequest.

Les algorithmes de hachage et de signature utilisés dans la signature DOIVENT être présents dans le champ supported_signature_algorithms du message CertificateRequest. De plus, les algorithmes de hachage et de signature DOIVENT être compatibles avec la clé du certificat end-entity du client. Les clés RSA PEUVENT être utilisées avec tout algorithme de hachage permis, sous réserve des restrictions du certificat, s'il en est.

Parce que les signatures DSA ne contiennent aucune indication sûre d'algorithme de hachage, il y a un risque de substitution de hachage si plusieurs hachages peuvent être utilisés avec n'importe quelle clé. Actuellement, DSA [DSS] peut seulement être utilisé avec SHA-1. Des révisions futures de DSS [DSS-3] sont attendues pour permettre l'utilisation d'autres algorithmes de résumé avec DSA, ainsi que des directives sur les algorithmes de résumé qui devraient être utilisés avec chaque taille de clé. De plus, des révisions futures de [PKIX] pourront spécifier des mécanismes pour que les certificats indiquent quels algorithmes de résumé sont à utiliser avec DSA.

7.4.9 Terminé

Quand est envoyé ce message :
Un message Terminé est toujours envoyé immédiatement après un message de 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 de 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 juste d'être négociés. Les receveurs des messages Terminé DOIVENT vérifier que le 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.
Structure de ce message :

```
struct {
      verify_data[verify_data_length] opaque ;
} Terminé;
```

verify_data
PRF(master_secret, finished_label, Hash(handshake_messages)) [0..verify_data_length-1];

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

Hash note un hachage des messages de prise de contact. Pour le PRF défini à la Section 5, le hachage DOIT être le hachage utilisé comme base pour le PRF. Toute suite de chiffrement qui définit un PRF différent DOIT aussi définir le hachage à utiliser dans le calcul de Terminé.

Dans les précédentes versions de TLS, la longueur de verify_data était toujours de 12 octets. Dans la version actuelle de TLS, elle dépend de la suite de chiffrement. Toute suite de chiffrement qui ne spécifie pas explicitement verify_data_length a une longueur verify_data_length égale à 12. Cela inclut toutes les suites de chiffrement existantes. Noter que cette représentation a le même codage que dans les versions précédentes. De futures suites de chiffrement PEUVENT spécifier d'autre longueurs mais une telle longueur DOIT être d'au moins 12 octets.

handshake_messages
Toutes les données provenant de tous les messages dans cette prise de contact (n'incluant aucun message HelloRequest) jusqu'à ce message, non inclus. Ce sont seulement les données visibles à la couche prise de contact et cela n'inclut pas les en-têtes de couche enregistrement. C'est la concaténation de toutes les structures de la prise de contact définies au paragraphe 7.4, échangées jusque là.

Un message Terminé non précédé d'un message ChangeCipherSpec au moment approprié de la prise de contact est une erreur fatale.

La valeur handshake_messages inclut tous les messages de prise de contact commençant au ClientHello jusqu'à ce message Terminé, mais celui-ci non inclus. Cela peut être différent des handshake_messages du paragraphe 7.4.8 parce que cela inclurait le message CertificateVerify (s'il est envoyé). Aussi, les handshake_messages pour le message Terminé envoyé par le client seront différents de ceux du message Terminé envoyé par le serveur, parce que celui envoyé en second va inclure le précédent.

Note : Les messages ChangeCipherSpec, alertes, et tous autres types d'enregistrements ne sont pas des messages de prise de contact et ne sont pas inclus dans les calculs de hachage. Aussi, les messages HelloRequest sont 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 des 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 ServerHello. L'algorithme de compression est négocié dans les messages hello, et les valeurs aléatoires sont échangées dans les messages hello. Il ne reste plus qu'à 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 pre_master_secret en master_secret. Le pre_master_secret devrait être effacé de la mémoire une fois le calcul du master_secret effectué.

master_secret = PRF("modèle initial de secret", "secret maître", ClientHello.random + ServerHello.random) [0..47];

Le secret maître fait toujours exactement 48 octets. La longueur du modèle initial de secret va varier selon la méthode d'échange de clés.

8.1.1 RSA

Lorsque RSA est utilisé pour l'authentification du serveur et l'échange de clés, un modèle initial de secret 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 modèle initial de secret. Les deux parties convertissent alors le modèle initial de secret en secret maître, comme spécifié ci-dessus.

8.1.2 Diffie-Hellman

On effectue un calcul Diffie-Hellman conventionnel. La clé négociée (Z) est utilisée comme modèle initial de secret, et est convertie en master_secret, comme spécifié ci-dessus. Les octets de gauche de Z qui ne contiennent que des bits à zéro sont enlevés avant qu'il soit utilisé comme modèle initial de secret.

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

9. Suites de chiffrement obligatoires

En l'absence d'un profil d'application standard spécifiant autre chose, une application conforme à TLS DOIT mettre en œuvre la suite de chiffrement TLS_RSA_WITH_AES_128_CBC_SHA (voir la définition à l'Appendice A.5).

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 en cours. Les messages sont traités comme des données transparentes pour la couche enregistrement.

11. Considérations sur la sécurité

Les questions de sécurité sont discutées tout au long du présent mémoire, et particulièrement dans les Appendices D, E, et F.

12. Considérations relatives à l'IANA

Le présent document utilise plusieurs registres qui ont été créés à l'origine dans [TLS1.1]. L'IANA les a mis à jour pour faire référence au présent document. Les registres et leurs politiques d'allocation (inchangées par rapport à [TLS1.1]) sont énumérés ci-dessous.

- Registre TLS des identifiants ClientCertificateType : Des valeurs futures dans la gamme 0-63 (décimal) inclus sont allouées via une action de normalisation [RFC2434]. Les valeurs dans la gamme 64-223 (décimal) inclus sont allouées via l'exigence d'une spécification [RFC2434]. Les valeurs de 224 à 255 (décimal) inclus sont réservées pour utilisation privée [RFC2434].

- Registre TLS des suites de chiffrement : Des valeurs futures avec le premier octet dans la gamme 0 à 191 (décimal) inclus sont allouées via une action de normalisation [RFC2434]. Les valeurs avec le premier octet dans la gamme 192-254 (décimal) sont allouées via l'exigence d'une spécification [RFC2434]. Les valeurs avec le premier octet 255 (décimal) sont réservées pour utilisation privée [RFC2434].

- Le présent document définit plusieurs suites de chiffrement fondées sur HMAC-SHA256, dont les valeurs (à l'Appendice A.5) ont été allouées à partir du registre TLS des suites de chiffrement.

- Registre TLS ContentType : Les valeurs futures seront allouées via une action de normalisation [RFC2434].

- Registre TLS des alertes : Les valeurs futures seront allouées via une action de normalisation [RFC2434].

- Registre TLS HandshakeType : Les valeurs futures seront allouées via une action de normalisation [RFC2434].

Le présent document utilise aussi un registre créé à l'origine dans la [RFC4366]. L'IANA l'a mis à jour pour faire référence au présent document. Le registre et sa politique d'allocation (inchangée par rapport à la [RFC4366]) est cité ci-dessous :

- Registre TLS ExtensionType : Les valeurs futures seront allouées via consensus de l'IETF [RFC2434]. L'IANA a mis à jour ce registre pour y inclure l'extension signature_algorithms et sa valeur correspondante (voir au paragraphe 7.4.1.4).

De plus, le présent document définit deux nouveaux registres qui seront tenus par l'IANA :

- Registre TLS SignatureAlgorithm : Le registre a été initialement rempli par les valeurs décrites au paragraphe 7.4.1.4.1. Les valeurs futures dans la gamme 0-63 (décimal) inclus seront allouées via une action de normalisation [RFC2434]. Les valeurs dans la gamme 64-223 (décimal) inclus seront allouées via l'exigence d'une spécification [RFC2434]. Les valeurs de 224 à 255 (décimal) inclus sont réservées pour utilisation privée [RFC2434].

- Registre TLS HashAlgorithm : Le registre a été initialement rempli par les valeurs décrites au paragraphe 7.4.1.4.1. Les valeurs futures dans la gamme 0-63 (décimal) inclus seront allouées via une action de normalisation [RFC2434]. Les valeurs dans la gamme 64-223 (décimal) inclus seront allouées via l'exigence d'une spécification [RFC2434]. Les valeurs de 224 à 255 (décimal) inclus sont réservées pour utilisation privée [RFC2434].

Le présent document utilise aussi le registre TLS des identifiants de méthode de compression, défini dans la [RFC3749]. L'IANA a alloué la valeur 0 pour la méthode de compression "null".

Appendice A. Structures des données et valeurs constantes du protocole

Le présent appendice décrit les types et constantes du protocole.

A.1 Couche d'enregistrement

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

version ProtocolVersion = { 3, 3 };              /* TLS v1.2*/

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

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

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

struct {
       type ContentType ;
       version ProtocolVersion ;
       longueur uint16 ;
       select (SecurityParameters.cipher_type) {
           cas flux : GenericStreamCipher;
           cas bloc :  GenericBlockCipher;
           cas aead :   GenericAEADCipher;
       } fragment;
} TLSCiphertext;

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

struct {
       IV[SecurityParameters.record_iv_length] opaque ;
       block-ciphered struct {
           content[TLSCompressed.length] opaque ;
           MAC[SecurityParameters.mac_length] opaque ;
           uint8 padding[GenericBlockCipher.padding_length];
           uint8 padding_length;
       };
} GenericBlockCipher;

struct {
       nonce_explicit[SecurityParameters.record_iv_length] opaque ;
```

```
    aead-ciphered struct {
        content[TLSCompressed.length] opaque ;
    };
} GenericAEADCipher;
```

A.2 Message de changement des spécifications 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_RESERVED(21),
    record_overflow(22),
    decompression_failure(30),
    handshake_failure(40),
    no_certificate_RESERVED(41),
    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_RESERVED(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    user_canceled(90),
    no_renegotiation(100),
    unsupported_extension(110),            /* nouveau */
    (255)
} AlertDescription;

struct {
    niveau AlertLevel ;
    description AlertDescription ;
} 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;
       longueur uint24 ;
       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:              Terminé;
       } body;
} Handshake;
```

A.4.1 Messages Hello

```
struct { } HelloRequest;

struct {
       uint32 gmt_unix_time;
       random_bytes[28] opaque ;
} 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..2^16-2>;
       CompressionMethod compression_methods<1..2^8-1>;
       select (extensions_present) {
            cas faux:
              struct {};
            cas vrai:
              Extension extensions<0..2^16-1>;
       };
} ClientHello;

struct {
       ProtocolVersion server_version;
       Random random;
       SessionID session_id;
       CipherSuite cipher_suite;
       CompressionMethod compression_method;
       select (extensions_present) {
            cas faux:
                 struct {};
            cas vrai:
                 Extension extensions<0..2^16-1>;
       };
} ServerHello;

struct {
```

```
        ExtensionType extension_type;
        extension_data<0..2^16-1> opaque ;
} Extension;

enum {
        signature_algorithms(13), (65535)
} ExtensionType;

enum{
        none(0), md5(1), sha1(2), sha224(3), sha256(4), sha384(5),
        sha512(6), (255)
} HashAlgorithm;
enum {
        anonymous(0), rsa(1), dsa(2), ecdsa(3), (255)
} SignatureAlgorithm;

struct {
            HashAlgorithm hash;
            SignatureAlgorithm signature;
} SignatureAndHashAlgorithm;

SignatureAndHashAlgorithm
supported_signature_algorithms<2..2^16-1>;
```

A.4.2 Messages d'authentification de serveur et d'échange de clés

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

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

enum { dhe_dss, dhe_rsa, dh_anon, rsa,dh_dss, dh_rsa
            /* peut être étendu, par exemple, pour ECDH -- voir [TLSECC] */
        } KeyExchangeAlgorithm;

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

struct {
          select (KeyExchangeAlgorithm) {
          cas dh_anon:
          paramètres ServerDHParams ;
          cas dhe_dss:
          cas dhe_rsa:
          paramètres ServerDHParams ;
          structure digitally-signed {
          client_random[32] opaque ;
          server_random[32] opaque ;
          paramètres ServerDHParams ;
          } signed_params;
          cas rsa:
          cas dh_dss:
          cas dh_rsa:
          struct {} ;
            /* le message est omis pour rsa, dh_dss, et dh_rsa */
            /* peut être étendu, par exemple, pour ECDH -- voir [TLSECC] */
} ServerKeyExchange;

enum {
```

```
        rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
        rsa_ephemeral_dh_RESERVED(5), dss_ephemeral_dh_RESERVED(6),
        fortezza_dms_RESERVED(20),
        (255)
} ClientCertificateType;

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

struct {
        ClientCertificateType certificate_types<1..2^8-1>;
        DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;

struct { } ServerHelloDone;
```

A.4.3 Messages d'authentification de client et d'échange de clés

```
struct {
    select (KeyExchangeAlgorithm) {
        cas rsa:
            EncryptedPreMasterSecret;
        cas dhe_dss:
        cas dhe_rsa:
        cas dh_dss:
        cas dh_rsa:
        cas dh_anon:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;

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

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

enum { implicit, explicit } PublicValueEncoding;

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

struct {
      digitally-signed struct {
          handshake_messages[handshake_messages_length] opaque ;
      }
} CertificateVerify;
```

A.4.4 Message de finalisation de prise de contact

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

A.5 Suite de chiffrement

Les valeurs suivantes définissent les codes des suites de chiffrement utilisées dans les messages ClientHello et ServerHello.

Une suite de chiffrement définit une spécification de chiffrement prise en charge dans TLS version 1.2.

TLS_NULL_WITH_NULL_NULL est spécifié et 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 fournit pas plus de protection qu'une connexion non sécurisée.

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

Les définitions de CipherSuite suivantes exigent que le serveur fournisse un certificat RSA qui puisse être utilisé pour l'échange de clés. Le serveur peut demander tout certificat capable de signature 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_WITH_NULL_SHA256             = { 0x00,0x3B };
CipherSuite TLS_RSA_WITH_RC4_128_MD5             = { 0x00,0x04 };
CipherSuite TLS_RSA_WITH_RC4_128_SHA             = { 0x00,0x05 };
CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA        = { 0x00,0x0A };
CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA         = { 0x00,0x2F };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA         = { 0x00,0x35 };
CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256      = { 0x00,0x3C };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA256      = { 0x00,0x3D };
```

Les définitions de suites 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, dans lequel les paramètres Diffie-Hellman sont signés par un certificat capable de signature, qui a été signé par la CA. L'algorithme de signature utilisé par le serveur est spécifié après le composant DHE du nom de la CipherSuite. Le serveur peut demander tout certificat capable de signature au client pour son authentification, 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_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x0D };
CipherSuite TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00,0x10 };
CipherSuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA    = { 0x00,0x13 };
CipherSuite TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA    = { 0x00,0x16 };
CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA      = { 0x00,0x30 };
CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA      = { 0x00,0x31 };
CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA     = { 0x00,0x32 };
CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA     = { 0x00,0x33 };
CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA      = { 0x00,0x36 };
CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA      = { 0x00,0x37 };
CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA     = { 0x00,0x38 };
CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA     = { 0x00,0x39 };
CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA256   = { 0x00,0x3E };
CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA256   = { 0x00,0x3F };
CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA256  = { 0x00,0x40 };
CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA256  = { 0x00,0x67 };
CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA256   = { 0x00,0x68 };
CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA256   = { 0x00,0x69 };
CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA256  = { 0x00,0x6A };
CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA256  = { 0x00,0x6B };
```

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. Utiliser ce mode est donc d'usage limité : ces suites de chiffrement NE DOIVENT PAS être utilisées par les mises en œuvre de TLS 1.2 à moins que la couche d'application n'ait spécifiquement demandé de permettre un échange de clés anonyme. (Les échanges de clés anonymes peuvent parfois être acceptables, par exemple, pour prendre en charge un chiffrement opportuniste lorsque aucun dispositif n'est en place pour l'authentification, ou lorsque TLS est utilisé au titre de protocoles de sécurité plus complexes qui ont d'autres moyens pour assurer l'authentification.)

```
CipherSuite TLS_DH_anon_WITH_RC4_128_MD5            = { 0x00,0x18 };
CipherSuite TLS_DH_anon_WITH_3DES_EDE_CBC_SHA       = { 0x00,0x1B };
CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA        = { 0x00,0x34 };
CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA        = { 0x00,0x3A };
CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA256     = { 0x00,0x6C };
CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA256     = { 0x00,0x6D };
```

Noter qu'utiliser un échange de clés non anonyme sans réellement vérifier l'échange de clés est essentiellement équivalent à un échange de clés anonyme, et les mêmes précautions s'appliquent. Bien que l'échange de clés non anonyme implique généralement un coût de calcul et de communication plus élevé qu'un échange de clés anonyme, il peut être dans l'intérêt de l'interopérabilité de ne pas désactiver l'échange de clés non anonyme lorsque la couche application permet l'échange de clés anonyme.

De nouvelles valeurs de suite de chiffrement ont été allouées par l'IANA comme décrit à la Section 12.

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 Les paramètres de sécurité

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

```
enum { null(0), (255) } CompressionMethod;
enum { server, client } ConnectionEnd;
enum { tls_prf_sha256 } PRFAlgorithm;
enum { null, rc4, 3des, aes } BulkCipherAlgorithm;
enum { stream, block, aead } CipherType;
enum { null, hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, hmac_sha512} MACAlgorithm;

/* d'autres valeurs peuvent être ajoutées aux algorithmes spécifiés dans CompressionMethod, PRFAlgorithm,
BulkCipherAlgorithm, et MACAlgorithm. */

struct {
ConnectionEnd         entity;
PRFAlgorithm          prf_algorithm;
BulkCipherAlgorithm   bulk_cipher_algorithm;
CipherType            cipher_type;
uint8                 enc_key_length;
uint8                 block_length;
uint8                 fixed_iv_length;
uint8                 record_iv_length;
MACAlgorithm          mac_algorithm;
uint8                 mac_length;
uint8                 mac_key_length;
CompressionMethod     compression_algorithm;
opaque                master_secret[48];
opaque                client_random[32];
opaque                server_random[32];
} SecurityParameters;
```

A.7 Changements par rapport à la RFC 4492

La RFC 4492 [TLSECC] a ajouté les suites de chiffrement à courbe elliptique à TLS. Le présent document change certaines des structures utilisées. Ce paragraphe précise les modifications demandées pour les mises en œuvre à la fois de la RFC 4492 et de TLS 1.2. Les mises en œuvre de TLS 1.2 qui n'appliquent pas la RFC 4492 n'ont pas besoin de le lire.

Le présent document ajoute un champ "signature_algorithm" à l'élément à signature numérique afin d'identifier les algorithmes de signature et de résumé utilisés pour créer une signature. Cette modification s'applique aux signatures numériques formées aussi bien à l'aide de ECDSA, permettant ainsi aux signatures ECDSA d'être utilisées avec des algorithmes de résumé autres que SHA-1, pourvu que cet usage soit compatible avec le certificat et avec toutes restrictions imposées par de futures révisions de [PKIX].

Comme décrit aux paragraphes 7.4.2 et 7.4.6, les restrictions portant sur les algorithmes de signature utilisés pour signer les certificats ne sont plus liées à la suite de chiffrement (quand elle est utilisée par le serveur) ou le ClientCertificateType (quand elle est utilisée par le client). Et donc, les restrictions portant sur l'algorithme utilisé pour signer les certificats spécifiés aux sections 2 et 3 de la RFC 4492 sont aussi levées. Comme dans le présent document, les restrictions sur les clés subsistent sur le certificat d'entité d'extrémité.

Appendice B. Glossaire

AES, Norme de chiffrement évolué *(Advanced Encryption Standard)*
AES [AES] est un algorithme de chiffrement symétrique largement utilisé. AES est un chiffrement de bloc avec une clé de 128, 192, ou 256 bits et une taille de bloc de 16 octets. TLS ne prend actuellement en charge que les tailles de clé de 128 et 256 bits.

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

chiffrement asymétrique
Voir à cryptographie à clé publique.

chiffrement authentifié avec données additionnelles (AEAD)
Algorithme de chiffrement symétrique qui fournit simultanément la confidentialité et l'intégrité du message.

authentification
L'authentification 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 opère sur le libellé dans des groupes de bits, appelés blocs. 64 bits était, et 128 bits est, une taille de bloc courante.

chiffrement en vrac *(bulk cipher)*
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 libellé chiffré avec un chiffrement de bloc est d'abord combiné par opérateur OUX 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 opérateur OUX avec le bloc de texte chiffré précédent (ou le VI).

certificat
Au titre du protocole X.509 (aussi appelé cadre d'authentification ISO), les certificats sont alloués par une autorité de certificat de confiance et fournissent un lien fort 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 initie la connexion de transport sous-jacente. La principale différence de fonctionnement entre le serveur et le client est que le serveur est généralement authentifié, alors que le client n'est authentifié que facultativement.

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

clé MAC d'écriture client
Les données secrètes utilisées pour authentifier les données écrites par le client.

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

Norme de chiffrement de données (DES, *Data Encryption Standard*)
DES [DES] est toujours un algorithme de chiffrement symétrique très largement utilisé bien qu'il soit considéré maintenant

comme assez faible. DES 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 de clés, DES est traité comme ayant une longueur de clé de 8 octets (64 bits), mais qu'il ne donne que 56 bits de protection. (le bit de moindre poids de chaque octet de la clé est présumé être mis pour produire une imparité dans cet octet de clé.) DES peut aussi être utilisé en mode [3DES] 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 de TLS) et donne l'équivalent de 112 bits de sécurité.

Norme de signature numérique (DSS, *Digital Signature Standard*)
Norme de signature numérique, qui inclut l'algorithme de signature numérique, approuvé par le National Institute of Standards et Technology (NIST), défini dans NIST FIPS PUB 186-2, "Digital Signature Standard", publié en janvier 2000 par le Ministère du Commerce US [DSS]. Un projet de mise à jour significative [DSS-3] a été publié en mars 2006.

signatures numériques
Les signatures numériques utilisent la cryptographie à clé publique et des fonctions de hachage unidirectionnelles pour produire une signature des données qui peuvent être authentifiées, et sont difficiles à contrefaire ou à répudier.

prise de contact *(handshake)*
Négociation initiale entre client et serveur, qui établit les paramètres de leurs transactions.

Vecteur d'initialisation (IV)
Lorsqu'un chiffrement de bloc est utilisé en mode CBC, le vecteur d'initialisation est combiné par opérateur OUX avec le premier bloc de libellé avant le chiffrement.

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

secret maître
Données secrètes sécurisées pour générer les clés de chiffrement, les MAC secrets, et les vecteurs d'initialisation.

MD5
MD5 [MD5] est une fonction de hachage qui convertit un flux de données de longueur arbitraire en un hachage de taille fixe (16 octets). Du fait des progrès significatifs de l'analyse cryptographique, au moment de la publication du présent document, MD5 ne peut plus être considéré comme une fonction de hachage "sûre".

cryptographie de clé publique
Classe de techniques cryptographiques employant 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 peuvent être vérifiés avec la clé publique.

fonction de hachage unidirectionnelle
Transformation unidirectionnelle qui convertit une quantité arbitraire de données en un hachage de longueur fixe. Il est difficile d'inverser la transformation par le calcul ou de trouver des collisions. MD5 et SHA sont des exemples de fonctions de hachage unidirectionnelles.

RC4
Chiffrement de flux inventé par Ron Rivest. Un chiffrement compatible est décrit dans [SCH].

RSA
Algorithme de clé publique très largement utilisé aussi bien pour le chiffrement que pour les signatures numériques. [RSA]

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 créées par le protocole de prise de contact. Les sessions définissent un ensemble de paramètres de sécurité cryptographiques qui peuvent être partagés entre plusieurs connexions. Les sessions sont utilisées pour éviter la négociation coûteuse de nouveaux paramètres de sécurité pour chaque connexion.

identifiant de session
Un identifiant de session est une valeur générée par un serveur qui identifie une session particulière.

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

clé MAC d'écriture serveur
Données secrètes utilisées pour authentifier les données écrites par le serveur.

SHA
L'algorithme de hachage sécurisé (SHA, *Secure Hash Algorithm*) [SHS] est défini dans FIPS PUB 180-2. Il produit une sortie de 20 octets. Noter que toutes les références à SHA (sans suffixe numérique) utilisent en réalité l'algorithme modifié SHA-1.

SHA-256
L'algorithme de hachage sécurisé à 256 bits est défini dans FIPS PUB 180-2. Il produit une sortie de 32 octets.

SSL
Protocole de couche de connexion sécurisée (SSL, *Secure Socket Layer*) [SSL3] de Netscape. TLS se fonde sur SSL version 3.0.

chiffrement de flux
Algorithme de chiffrement qui convertit une clé en un flux de clé cryptographiquement fort, qui est alors combiné par l'opérateur OUX avec le libellé.

chiffrement symétrique
Voir chiffrement en vrac.

Sécurité de la couche Transport (TLS, *Transport Layer Security*)
Le présent protocole ; aussi, le groupe de travail Transport Layer Security de l'équipe d'ingénierie de l'Internet (IETF, *Internet Engineering Task Force*). Voir "Informations sur le groupe de travail" à la fin de ce document.

Appendice C. Définitions des suites de chiffrement

Suite de chiffrement	Échange de clé	Chiffrement	Mac
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_WITH_NULL_SHA256	RSA	NULL	SHA256
TLS_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	AES_128_CBC	SHA
TLS_RSA_WITH_AES_256_CBC_SHA	RSA	AES_256_CBC	SHA
TLS_RSA_WITH_AES_128_CBC_SHA256	RSA	AES_128_CBC	SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256	RSA	AES_256_CBC	SHA256
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
TLS_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4_128	MD5
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA	DH_DSS	AES_128_CBC	SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA	DH_RSA	AES_128_CBC	SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	DHE_DSS	AES_128_CBC	SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	DHE_RSA	AES_128_CBC	SHA
TLS_DH_anon_WITH_AES_128_CBC_SHA	DH_anon	AES_128_CBC	SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA	DH_DSS	AES_256_CBC	SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA	DH_RSA	AES_256_CBC	SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	DHE_DSS	AES_256_CBC	SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	DHE_RSA	AES_256_CBC	SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA	DH_anon	AES_256_CBC	SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	DH_DSS	AES_128_CBC	SHA256
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	DH_RSA	AES_128_CBC	SHA256

TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	DHE_DSS	AES_128_CBC	SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	DHE_RSA	AES_128_CBC	SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA256	DH_anon	AES_128_CBC	SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	DH_DSS	AES_256_CBC	SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	DH_RSA	AES_256_CBC	SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	DHE_DSS	AES_256_CBC	SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	DHE_RSA	AES_256_CBC	SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256	DH_anon	AES_256_CBC	SHA256

Chiffrement	Type de clé	Matériel	Taille d'IV	Taille de bloc
NULL	Flux	0	0	N/A
RC4_128	Flux	16	0	N/A
3DES_EDE_CBC	Bloc	24	8	8
AES_128_CBC	Bloc	16	16	16
AES_256_CBC	Bloc	32	16	16

MAC	Algorithme	mac_length	mac_key_length
NULL	N/A	0	0
MD5	HMAC-MD5	16	16
SHA	HMAC-SHA1	20	20
SHA256	HMAC-SHA256	32	32

Type
Indique si il s'agit d'un chiffrement de flux ou d'un chiffrement de bloc fonctionnant en mode CBC.

Matériel de clé
Le nombre d'octets pour le key_block qui sont utilisés pour générer les clés d'écriture.

Taille d'IV
Quantité de données qu'il est nécessaire de générer pour le vecteur d'initialisation. Zéro pour les chiffrements de flux ; égal à la taille de bloc pour les chiffrements de bloc (ce qui est égal à SecurityParameters.record_iv_length).

Taille de bloc
Quantité de données qu'un chiffrement de bloc chiffre dans un seul tronçon ; un chiffrement de bloc fonctionnant en mode CBC ne peut chiffrer qu'un multiple pair de sa taille de bloc.

Appendice D. Notes de mise en œuvre

Le protocole TLS ne peut pas empêcher de nombreuses fautes de sécurité courantes. Le présent appendice donne plusieurs recommandations pour aider les développeurs.

D.1 Génération et germination de nombre aléatoire

TLS exige un générateur de nombres pseudo aléatoires (PRNG, *pseudorandom number generator*) cryptographiquement sûr. La conception et le germe des PRNG doivent être soigneusement étudiés. Les PRNG fondés sur des opérations de hachage sécurisées, en particulier SHA-1, sont acceptables, mais ne peuvent pas fournir plus de sécurité que la taille de l'état du générateur de nombres aléatoires.

Pour estimer la quantité de germe produite, ajouter le nombre de bits d'information imprévisible dans chaque octet de germe. Par exemple, les valeurs de cadence de frappe tirées d'une horloge à 18,2 Hz d'un micro ordinateur fournissent 1 ou 2 bits sûr chacune, même si la taille totale de la valeur du compteur est 16 bits ou plus. Faire un germe de PRNG de 128 bits exigerait approximativement 100 valeurs de cette sorte.

[RANDOM] donne des indications sur la génération de valeurs aléatoires.

D.2 Certificats et authentification

Les mises en œuvre sont responsables de la vérification de l'intégrité des certificats et devraient généralement prendre en charge 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

doivent être faits avec beaucoup de soin. Les usagers devraient être capables de voir les informations sur le CA de certificat et de racine.

D.3 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 prendra probablement pas beaucoup de suites de chiffrement. Par exemple, le Diffie-Hellman anonyme est fortement déconseillé parce qu'il ne peut pas empêcher les attaques par interpositions. Les applications devraient aussi mettre en application les 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.

D.4 Pièges à éviter

L'expérience de la mise en œuvre a montré que certaines parties des spécifications TLS antérieures ne sont pas facilement compréhensibles, et ont été une source de problèmes d'interopérabilité et de sécurité. Beaucoup de ces problèmes ont été précisés dans le présent document, mais cet appendice contient une courte liste des choses les plus importantes qui exigent une attention particulière de la part des mises en œuvre.

Questions de protocole TLS :

- Traitez vous correctement les messages de prise de contact qui sont fragmentés en plusieurs enregistrement TLS (voir au paragraphe 6.2.1) ? Y compris le cas particulier d'un ClientHello séparé en plusieurs petits fragments ? Fragmentez vous les messages de prise de contact qui dépassent la taille de fragment maximum ? En particulier, les messages de prise de contact de certificat et demande de certificat peuvent être assez longs pour requérir une fragmentation.

- Ignorez vous le numéro de version de couche d'enregistrement TLS dans tous les enregistrements TLS avant le ServerHello (voir l'Appendice E.1) ?

- traitez vous correctement les extensions TLS dans le ClientHello, y compris d'omettre complètement le champ extensions ?

- Acceptez vous la renégociation, initiée à la fois par le client et le serveur ? Bien que la renégociation soit une caractéristique facultative, il est vivement recommandé de la prendre en charge.

- Lorsque le serveur a demandé un certificat de client, mais qu'aucun certificat convenable n'est disponible, envoyez vous correctement un message Certificate vide, au lieu d'omettre tout le message (voir au paragraphe 7.4.6) ?

Détails cryptographiques :

- Dans le modèle initial de secret chiffré en RSA, envoyez vous et vérifiez vous correctement le numéro de version ? Lorsque vous rencontrez une erreur, continuez vous la prise de contact pour éviter l'attaque Bleichenbacher (voir au paragraphe 7.4.7.1) ?

- Quelles contre mesures utilisez vous pour empêcher les attaques de cadencement contre les opérations de déchiffrement et de signature RSA (voir au paragraphe 7.4.7.1) ?

- Lors de la vérification des signatures RSA, acceptez vous à la fois les paramètres manquants et nuls (voir paragraphe 4.7) ? Vérifiez vous que le bourrage RSA n'a pas de sonnées supplémentaires après la valeur du hachage ? [FI06]

- Dans l'échange de clés Diffie-Hellman, retirez vous correctement les octets à zéro en tête de la clé négociée (voir au paragraphe 8.1.2) ?

- Votre client TLS vérifie-t-il que les paramètres Diffie-Hellman envoyés par le serveur sont acceptables (voir au paragraphe F.1.1.3) ?

- Comment générez vous les IV imprévisibles pour les chiffrements en mode CBC (voir au paragraphe 6.2.3.2) ?

- Acceptez vous un long bourrage en mode CBC (jusqu'à 255 octets ; voir au paragraphe 6.2.3.2) ?

- Comment traitez vous les attaques de cadencement en mode CBC (paragraphe 6.2.3.2) ?

- Utilisez vous un générateur de nombres aléatoires fort, et plus important, avec un germe approprié (voir l'Appendice D.1) pour générer le modèle initial de secret (pour l'échange de clés RSA), les valeurs privées Diffie-Hellman, le paramètre DSA

"k", et les autres valeurs critiques pour la sécurité ?

Appendice E Rétro-compatibilité

E.1 Compatibilité avec TLS 1.0/1.1 et SSL 3.0

Comme il y a diverses versions de TLS (1.0, 1.1, 1.2, et de futures versions) et SSL (2.0 et 3.0), il est nécessaire d'avoir des moyens pour négocier la version spécifique du protocole à utiliser. Le protocole TLS fournit un mécanisme incorporé pour la négociation de version de façon à ne pas encombrer d'autres composants de protocole avec la complexité du choix de version.

Les versions 1.0, 1.1, et 1.2 de TLS, et SSL 3.0 sont très similaires, et utilisent des messages ClientHello compatibles ; et donc, les prendre toutes en charge est relativement facile. De même, les serveurs pourront facilement traiter des clients qui essayent d'utiliser les futures versions de TLS pour autant que les formats du ClientHello restent compatibles, et que le client accepte la plus récente version de protocole disponible chez le serveur.

Un client TLS 1.2 qui souhaite négocier avec de vieux serveurs va envoyer un ClientHello TLS 1.2 normal, contenant { 3, 3 } (TLS 1.2) dans ClientHello.client_version. Si le serveur ne prend pas en charge cette version, il va répondre par un ServerHello qui contient un numéro de version plus ancien. Si le client accepte d'utiliser cette version, la négociation va se poursuivre comme il convient pour le protocole négocié.

Si la version choisie par le serveur n'est pas acceptée par le client (ou pas acceptable), le client DOIT envoyer un message d'alerte "protocol_version" et clore la connexion.

Si un serveur TLS reçoit un ClientHello qui contient un numéro de version supérieur à la plus forte version acceptée par le serveur, il DOIT répondre en accord avec la plus forte version supportée par le serveur.

Un serveur TLS peut aussi recevoir un ClientHello contenant un numéro de version inférieur à la plus forte version supportée. Si le serveur souhaite négocier avec de vieux clients, il va passer comme il convient à la plus forte version supportée par le serveur qui n'est pas supérieure à ClientHello.client_version. Par exemple, si le serveur accepte TLS 1.0, 1.1, et 1.2, et si client_version est TLS 1.0, le serveur va passer à un ServerHello TLS 1.0. Si le serveur accepte (ou si il veut utiliser) seulement des versions supérieures à client_version, il DOIT envoyer un message d'alerte "protocol_version" et clore la connexion.

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

Note : certaines mises en œuvre de serveur sont connues pour mettre en œuvre de façon incorrecte la négociation de version. Par exemple, il y a des serveurs TLS 1.0 fautifs qui closent simplement la connexion lorsque le client offre une version plus récente que TLS 1.0. Il est aussi connu que certains serveurs vont refuser la connexion si des extensions TLS sont incluses dans le ClientHello. L'interopérabilité avec de tels serveurs fautifs est un sujet complexe qui sort du domaine d'application du présent document, et pourra exiger plusieurs tentatives de connexion de la part du client.

Les versions antérieures de la spécification TLS n'étaient pas parfaitement claires sur ce que le numéro de version de couche d'enregistrement (TLSPlaintext.version) devrait contenir lors de l'envoi du ClientHello (c'est-à-dire, avant qu'on sache quelle version du protocole sera employée). Et donc, les serveurs TLS conformes à la présente spécification DOIVENT accepter toute valeur {03,XX} comme numéro de version de couche d'enregistrement pour le ClientHello.

Les clients TLS qui souhaitent négocier avec de plus anciens serveurs PEUVENT envoyer n'importe quelle valeur {03,XX} comme numéro de version de couche d'enregistrement. Les valeurs normales devraient être {03,00}, le plus faible numéro de version supporté par le client, et la valeur de ClientHello.client_version. Aucune valeur unique ne garantira l'interopérabilité avec ces vieux serveurs, mais ceci est un sujet complexe qui va au delà de la portée du présent document.

E.2 Compatibilité avec SSL 2.0

Les clients TLS 1.2 qui souhaitent prendre en charge les serveurs SSL 2.0 DOIVENT envoyer les messages CLIENT-HELLO de version 2.0 définis dans [SSL2]. Le message DOIT contenir le même numéro de version que celui qui serait utilisé pour un ClientHello ordinaire, et DOIT coder les suites de chiffrement TLS prises en charge dans le champ CIPHER-SPECS-DATA comme décrit ci-dessous.

Avertissement : La capacité à envoyer des messages CLIENT-HELLO de version 2.0 messages sera supprimée graduellement avec toute la diligence nécessaire dans la mesure où le nouveau format de ClientHello offre de meilleurs mécanismes pour

passer aux versions plus récentes et négocier les extensions. Les clients TLS 1.2 NE DEVRAIENT PAS accepter SSL 2.0.

Cependant, même les serveurs TLS qui ne prennent pas en charge SSL 2.0 PEUVENT accepter les messages CLIENT-HELLO de version 2.0. Le message est présenté ci-dessous en détail suffisant pour les développeurs de serveur TLS ; la vraie définition est toujours censée être [SSL2].

Pour les besoins de la négociation, le CLIENT-HELLO 2.0 est interprété de la même façon qu'un ClientHello avec une méthode de compression "nulle" et pas d'extensions. Noter que ce message DOIT être envoyé directement sur le réseau, et non enveloppé comme un enregistrement TLS. Pour calculer Terminé et CertificateVerify, le champ msg_length n'est pas considéré comme faisant partie du message de prise de contact.

```
uint8 V2CipherSpec[3];
    struct {
        uint16 msg_length;
        uint8 msg_type;
        Version version;
        uint16 cipher_spec_length;
        uint16 session_id_length;
        uint16 challenge_length;
        V2CipherSpec cipher_specs[V2ClientHello.cipher_spec_length];
        session_id[V2ClientHello.session_id_length] opaque;
        challenge[V2ClientHello.challenge_length opaque;
} V2ClientHello;
```

msg_length
Le bit de plus fort poids DOIT être à 1 ; les bits restants contiennent la longueur des données suivantes en octets.

msg_type
Ce champ, conjointement avec le champ version, identifie un message ClientHello de version 2. Sa valeur DOIT être 1.

version
Égal à ClientHello.client_version.

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

session_id_length
Ce champ DOIT avoir une valeur de zéro pour un client qui revendique la prise en charge de TLS 1.2.

challenge_length
Longueur en octets de la mise en cause du client qui demande au serveur de s'authentifier. Historiquement, les valeurs permises sont entre 16 et 32 octets inclus. Lorsqu'on utilise la prise de contact rétrocompatible SSLv2, le client DEVRAIT utiliser une mise en cause de 32 octets.

cipher_specs
C'est une liste de tous les CipherSpecs que le client veut et est capable d'utiliser. En plus des spécifications de chiffrement de 2.0 définies dans [SSL2], cela comporte les suites de chiffrement TLS normalement envoyées dans les ClientHello.cipher_suites, chaque suite de chiffrement étant préfixée d'un octet de zéros. Par exemple, la suite de chiffrement TLS {0x00,0x0A} serait envoyée sous la forme {0x00,0x00,0x0A}.

session_id
Ce champ DOIT être vide.

challenge
Correspond au ClientHello.random. Si la longueur de la mise en cause est inférieure à 32, le serveur TLS bourrera les données avec des octets à zéro en tête (note : pas en queue) pour faire une longueur de 32 octets.

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

E.3 Éviter la régression de version vers l'attaque par interposition

Lorsqu'un client TLS retombe au mode de compatibilité de version 2.0, il DOIT utiliser le format de bloc spécial PKCS#1.

Ceci est fait de telle sorte que les serveurs TLS rejettent les sessions de version 2.0 avec des clients capables de TLS.

Lorsqu'un client négocie SSL 2.0 mais prend aussi en charge TLS, il DOIT régler les huit octet de droite (de moindre poids) du bourrage PKCS (non inclus la terminaison nulle du bourrage) pour le chiffrement RSA du champ ENCRYPTED-KEY-DATA de la CLIENT-MASTER-KEY à 0x03 (les autres octets de bourrage sont aléatoires).

Lorsqu'un serveur capable de TLS négocie SSL 2.0, il DEVRAIT, après déchiffrement du champ ENCRYPTED-KEY-DATA, vérifier que ces huit octets de bourrage sont 0x03. Si ils ne le sont pas, le serveur DEVRAIT générer une valeur aléatoire pour SECRET-KEY-DATA, et continuer la prise de contact (qui va éventuellement échouer si les clé ne correspondent pas). Noter que le rapport d'une situation d'erreur au client peut rendre le serveur vulnérable aux attaques décrites dans [BLEI].

Appendice 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 qu'ils ne peuvent obtenir les informations secrètes de sources extérieures au protocole. Les attaquants sont supposés avoir la capacité de capturer, modifier, supprimer, répéter, et altérer d'autre manière les messages envoyés sur le canal de communication. Le présent appendice souligne 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 comporte 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 clé

TLS accepte trois modes d'authentification : authentification des deux parties, authentification du serveur avec un client non authentifié, et anonymat total. Chaque fois que le serveur est authentifié, le canal est sûr 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, son message de certificat doit fournir une chaîne de certificat valide conduisant à une autorité de certificat acceptable. De même, les clients authentifiés doivent fournir 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 pas périmé ou révoqué.

Le but général du processus d'échange de clés est de créer un modèle initial de secret connu des parties à la communication et pas des attaquants. Le modèle initial de secret sera utilisé pour générer le secret maître (voir au paragraphe 8.1). Le secret maître est exigé pour générer les messages Terminé, les clés de chiffrement, et les clés de MAC (voir aux paragraphes 7.4.9 et 6.3). En envoyant un message Terminé correct, les parties prouvent donc qu'elles connaissent le modèle initial de secret correct.

F.1.1.1 Échange de clé anonyme

Les sessions complètement anonymes peuvent être établies en utilisant Diffie-Hellman pour l'échange de clés. Les paramètres publics du serveur sont contenus dans le message d'échange de clés du serveur, et ceux du client sont envoyés dans le message d'échange de clés 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 modèle initial de secret).

Avertissement : Les connexions complètement anonymes ne fournissent de protection que contre l'espionnage passif. Si un canal indépendant à l'abri des altérations n'est pas utilisé pour vérifier que les messages Terminé n'ont pas été remplacés par un attaquant, l'authentification du serveur est nécessaire dans les environnements où les attaques actives par interposition sont un problème.

F.1.1.2. Échange de clé RSA avec authentification

Avec RSA, échange de clés et authentification du serveur sont combinées. La clé publique est contenue dans le certificat du serveur. Noter que la compromission des résultats de clé RSA statique du serveur résulte en la perte de confidentialité pour toutes les sessions protégées par cette clé statique. Les utilisateurs de TLS qui désirent un secret parfait vers l'aval devraient utiliser les suites de chiffrement DHE. Les dommages causés par l'exposition d'une clé privée peuvent être limités en changeant fréquemment sa clé privée (et le certificat).

Après avoir vérifié le certificat du serveur, le client chiffre un modèle initial de secret avec la clé publique du serveur. En

réussissant à décoder le modèle initial de secret et en produisant un message Terminé correct, le serveur démontre qu'il connaît la clé privée correspondant au certificat du serveur.

Lorsque RSA est utilisé pour l'échange de clés, les clients sont authentifiés en utilisant le message Vérifier le certificat (voir au paragraphe 7.4.8). Le client signe une valeur déduite 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 clé 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 utiliser le message d'échange de clés du serveur pour envoyer un ensemble de paramètres Diffie-Hellman temporaires signés avec un certificat DSA ou RSA.
Les paramètres temporaires sont hachés avec les valeurs aléatoires du hello avant de signer pour assurer que des attaquants ne répètent pas de vieux paramètres. Dans l'un et 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 les paramètres Diffie-Hellman fixes, son certificat contient les informations exigées pour achever 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 modèle initial de secret) chaque fois qu'ils communiquent. Pour empêcher que le modèle initial de secret demeure en mémoire plus longtemps que nécessaire, il devrait être converti aussitôt que possible en secret maître. Les paramètres Diffie-Hellman client doivent être compatibles avec ceux fournis par le serveur pour l'échange de clés pour fonctionner.

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

Si la même paire de clés DH doit être utilisée pour plusieurs prises de contact, soit parce que le client ou le serveur a un certificat qui contient une paire de clé DH fixe, soit parce que le serveur réutilise les clés DH, il faut prendre soin d'empêcher les attaques de petit sous-groupe. Les mises en œuvre DEVRAIENT suivre les lignes directrices qui se trouvent dans [SUBGROUP].

Les attaques de petit sous-groupe sont très facilement évitées en utilisant une des suites de chiffrement DHE et en générant une clé privée DH fraîche (X) pour chaque prise de contact. Si une base convenable (telle que 2) est choisie, g^X mod p peut être calculé très rapidement ; donc, les coûts de performance sont minimisés. De plus, utiliser une clé fraîche pour chaque prise de contact donne un secret parfait vers l'aval. Les mises en œuvre DEVRAIENT générer un nouveau X pour chaque prise de contact lorsqu'elles utilisent des suites de chiffrement DHE.

Parce que TLS permet au serveur de fournir des groupes DH arbitraires, le client devrait vérifier que le groupe DH est de taille convenable comme défini par la politique locale. Le client DEVRAIT aussi vérifier que l'exposant public DH lui paraît être de la taille adéquate. [KEYSIZ] est un guide utile sur la force des diverses tailles de groupes. Le serveur PEUT choisir d'aider le client en fournissant un groupe connu, comme ceux définis dans [IKEALG] ou [MODP]. Cela peut être vérifié par simple comparaison.

F.1.2 Attaques de régression de version

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

Bien que la solution qui consiste à utiliser un bourrage non aléatoire du message de bloc de type 2 de PKCS n° 1 soit inélégante, elle donne un moyen raisonnablement sûr aux serveurs de version 3.0 pour détecter l'attaque. Cette solution n'est pas sûre contre les attaquants qui peuvent forcer la clé et substituer un nouveau message ENCRYPTED-KEY-DATA contenant la même clé (mais avec un bourrage normal) avant que le seuil d'attente spécifié par l'application ne soit arrivé à expiration. Altérer le bourrage des huit octets de moindre poids du bourrage 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 c'est essentiellement équivalent à augmenter la taille du bloc d'entrée de 8 octets.

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

Un attaquant pourrait essayer d'influencer l'échange de prise de contact pour amener les parties à choisir des algorithmes de chiffrement différents de ceux qu'ils auraient normalement choisi.

Pour cette attaque, un attaquant doit activement changer un ou plusieurs messages de prise de contact. Si cela arrive, le client et

le serveur calculeront des valeurs différentes pour les hachages de message de prise de contact. Il en résultera que chaque partie n'acceptera pas le message Terminé de l'autre. Sans le modèle initial de secret, l'attaquant ne peut pas réparer les messages Terminé, ainsi, l'attaque sera découverte.

F.1.4 Reprise des sessions

Lorsqu'une connexion est établie par la reprise d'une session, les nouvelles valeurs aléatoires ClientHello.random et 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 sécurisé utilisées pour produire les clés de chiffrement et les clés de MAC soient sûres, la connexion devrait être sûre et effectivement indépendante des connexions antérieures. Les attaquants ne peuvent pas utiliser de clés de chiffrement ou de secrets de MAC connus pour compromettre le secret maître sans casser les opérations de hachage sécurisé.

Les sessions ne peuvent être reprises qu'avec l'accord à la fois du client et du serveur. Si l'une des parties soupçonne que la session pourrait avoir été compromise, ou que des certificats pourraient avoir expiré ou être révoqués, il 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 fonctionnent dans des environnements d'insécurité relative ne devraient pas écrire les identifiants de session dans des mémoires permanentes.

F.2 Protection des données d'application

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

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

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

Note : Les clés MAC peuvent être plus grandes que les clés de chiffrement, de sorte que les messages peuvent rester à l'épreuve des altérations même si les clés de chiffrement sont cassées.

F.3 IV explicites

[CBCATT] décrit une attaque de libellé choisi contre TLS qui dépend de la connaissance du vecteur d'initialisation pour un enregistrement. Les versions précédentes de TLS [TLS1.0] utilisaient le résidu de CBC de l'enregistrement précédent comme vecteur d'initialisation et permettaient donc cette attaque. La présente version utilise un IV explicite afin de se protéger contre cette attaque.

F.4 Sécurité des modes de chiffrement composites

TLS sécurise les données d'application transmises via l'utilisation des fonctions de chiffrement symétrique et d'authentification définies dans la suite de chiffrement négociée. L'objectif est de protéger à la fois l'intégrité et la confidentialité des données transmises contre les actions malveillantes d'attaquants actifs à travers le réseau. Il se trouve que l'ordre dans lequel les fonctions de chiffrement et d'authentification sont appliquées aux données joue un rôle important pour atteindre cet objectif [ENCAUTH].

La méthode la plus robuste, appelée chiffrer puis authentifier, applique d'abord le chiffrement aux données puis applique un MAC au texte chiffré. Cette méthode assure que les buts d'intégrité et de confidentialité sont obtenus avec TOUTE paire de fonctions de chiffrement et de MAC, pourvu que celle là soit sûre contre les attaques de libellé choisi et que le MAC soit sûr contre les attaques de message choisi. TLS utilise une autre méthode, appelée authentifier puis chiffrer, dans laquelle on calcule d'abord un MAC sur le libellé, puis l'enchaînement du libellé et du MAC est chiffré. Cette méthode s'est révélée sûre pour CERTAINES combinaisons de fonctions de chiffrement et de fonctions de MAC, mais il n'est pas garanti qu'elle soit sûre en général.

En particulier, il a été montré qu'il existe des fonctions de chiffrement parfaitement sûres (sûres même au sens de la théorie de l'information) qui combinées à toute fonction MAC sûre, échoue à tenir l'objectif de confidentialité contre une attaque active. Donc, les nouvelles suites de chiffrement et modes de fonctionnement adoptés dans TLS doivent être analysés par la méthode authentifier puis chiffrer pour vérifier qu'elles satisfont aux objectifs déclarés d'intégrité et de confidentialité.

Actuellement, la sécurité de la méthode authentifier puis chiffrer a été prouvée pour certains cas importants. L'un est le cas des chiffrements de flux dans lesquels un bourrage, imprévisible par le calcul, de la longueur du message, plus la longueur de l'étiquette du MAC, est produit en utilisant un générateur pseudo aléatoire et ce bourrage est combiné par opérateur OUX avec l'enchaînement du libellé et de l'étiquette de MAC. L'autre est le cas du mode CBC utilisant un chiffrement de bloc sécurisé. Dans ce cas, la sécurité peut être démontrée si on applique un passage de chiffrement de CBC à l'enchaînement du libellé et du MAC et qu'on utilise un nouveau vecteur d'initialisation indépendant, et imprévisible pour chaque nouvelle paire de libellé et de MAC. Dans les versions de TLS antérieures à 1.1, le mode CBC était utilisé de façon appropriée EXCEPTÉ qu'il utilisait un vecteur d'initialisation prévisible sous la forme du dernier bloc du texte chiffré précédent. Cela rendait TLS ouvert aux attaques de libellé choisi. La présente version du protocole est immunisée contre ces attaques. Pour les détails exacts des modes de chiffrement prouvés sûrs, voir [ENCAUTH].

F.5 Déni de service

TLS est susceptible d'un certain nombre d'attaques de déni de service (DoS). En particulier, un attaquant qui initie un grand nombre de connexions TCP peut causer la consommation de grandes quantités de CPU par un serveur pour effectuer le déchiffrement RSA. Cependant, parce que TLS est généralement utilisé sur TCP, il est difficile à l'attaquant de cacher son point d'origine si un brassage aléatoire SYN TCP approprié est utilisé [SEQNUM] par la pile TCP.

Comme TLS fonctionne sur TCP, il est aussi susceptible de subir un certain nombre d'attaques de DoS sur les connexions individuelles. En particulier, les attaquants peuvent faire de faux RST, terminant ainsi les connexions, ou faire de faux enregistrements TLS partiels, causant par là le calage de la connexion. On ne peut pas en général se défendre contre ces attaques par un protocole utilisant TCP. Les développeurs ou utilisateurs qui ont des problèmes avec cette classe d'attaques devraient utiliser IPsec AH [AH] ou ESP [ESP].

F.6 Notes finales

Pour que TLS soit capable de fournir une connexion sûre, les systèmes client et serveur, les clés et les applications doivent être sûrs. De plus, la mise en œuvre doit être affranchie de toute erreur de sécurité.

La force du système n'est pas supérieure à celle du plus faible algorithme d'échange de clés et d'authentification pris en charge, et seules des fonctions cryptographiques dignes de confiance devraient être utilisées. Les clés publiques courtes et les serveurs anonymes devraient n'être utilisés qu'avec une grande prudence. Les mises en œuvre et les usagers doivent être prudents en décidant quels certificats et autorités de certificat sont acceptables ; une autorité de certificat malhonnête peut causer des dommages désastreux.

Références normatives

[AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)" FIPS 197. 26 novembre 2001.

[3DES] National Institute of Standards et Technology, "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", NIST Special Publication 800-67, mai 2004.

[DSS] NIST FIPS PUB 186-2, "Digital Signature Standard", National Institute of Standards and Technology, U.S. Department of Commerce, 2000.

[HMAC] H. Krawczyk, M. Bellare et R. Canetti, "HMAC : Hachage de clés pour l'authentification de message", RFC 2104, février 1997.

[MD5] R. Rivest, "Algorithme MD5 de résumé de message", RFC 1321, avril 1992.

[PKCS1] J. Jonsson et B. Kaliski, "Normes de cryptographie à clés publiques (PKCS) n° 1 : Spécifications de la cryptographie RSA, version 2.1", RFC 3447, février 2003.

[PKIX] R. Housley, W. Polk, W. Ford et D. Solo, "Profil de certificat d'infrastructure de clé publique X.509 et liste de révocation de certificat (CRL) pour l'Internet", RFC 3280, avril 2002. *(Rendue obsolète par la RFC 5280, mai*

2008)

[SCH] B. Schneier. "Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2ème éd.", publié par John Wiley & Sons, Inc. 1996.

[SHS] NIST FIPS PUB 180-2, "Secure Hash Standard", National Institute of Standards and Technology, U.S. Department of Commerce, August 2002.

[REQ] S. Bradner, "Mots clés à utiliser dans les RFC pour indiquer les niveaux d'exigence", BCP 14, RFC 2119, mars 1997.

[RFC2434] T. Narten et H. Alvestrand, "Lignes directrices pour la rédaction de la section Considérations relatives à l'IANA dans les RFC", BCP 26, RFC 2434, octobre 1998.

[X680] Recommandation UIT-T X.680 (2002) | ISO/CEI 8824-1 :2002, Technologies de l'information – Notation de syntaxe abstraite n° 1 (ASN.1) : Spécification de la notation de base.

[X690] Recommandation UIT-T X.690 (2002) | ISO/CEI 8825-1 :2002, Technologies de l'information – Règles de codage ASN.1 : Spécification des règles de codage de base (BER), règles de codage canonique (CER) et règles de codage distinctives (DER).

Références informatives

[AEAD] D. McGrew, "Interface et algorithmes pour le chiffrement authentifié", RFC 5116, janvier 2008.

[AH] S. Kent, "En-tête d'authentification IP", RFC 4302, décembre 2005.

[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.

[CBCATT] Moeller, B., "Security of CBC Ciphersuites in SSL/TLS: Problems et Countermeasures", http://www.openssl.org/~bodo/tls-cbc.txt.

[CBCTIME] Canvel, B., Hiltgen, A., Vaudenay, S., et M. Vuagnoux, "Password Interception in a SSL/TLS Channel", Advances in Cryptology -- CRYPTO 2003, LNCS vol. 2729, 2003.

[CCM] "NIST Special Publication 800-38C: The CCM Mode for Authentication et Confidentiality", http://csrc.nist.gov/publications/nistpubs/800-38C/ SP800-38C.pdf

[DES] National Institute of Standards and Technology, "Data Encryption Standard (DES)", FIPS PUB 46-3, octobre 1999.

[DSS-3] NIST FIPS PUB 186-3 Draft, "Digital Signature Standard", National Institute of Standards and Technology, U.S. Department of Commerce, 2006.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANS X9.62-2005, novembre 2005.

[ENCAUTH] Krawczyk, H., "The Order of Encryption et Authentication for Protecting Communications (Or: How Secure is SSL?)", Crypto 2001.

[ESP] S. Kent, "Encapsulation IP de charge utile de sécurité (ESP)", RFC 4303, décembre 2005.

[FI06] Hal Finney, "Bleichenbacher's RSA signature forgery based on implementation error", ietf-openpgp@imc.org mailing list, 27 août 2006, http://www.imc.org/ietf-openpgp/ mail-archive/msg14307.html.

[GCM] Dworkin, M., NIST Special Publication 800-38D, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) et GMAC", novembre 2007.

[IKEALG] J. Schiller, "Algorithmes cryptographiques à utiliser dans l'échange de clés Internet, version 2 (IKEv2)", RFC 4307, décembre 2005.

[KEYSIZ] H. Orman et P. Hoffman, "Détermination de la force des clés publiques utilisées pour l'échange de clés symétriques", BCP 86, RFC 3766, avril 2004.

[KPR03] V. Klima, O. Pokorny, T. Rosa, "Attacking RSA-based Sessions in SSL/TLS", http://eprint.iacr.org/2003/052/, mars 2003.

[MODP] T. Kivinen et M. Kojo, "Groupes modulaires exponentiels (MODP) Diffie-Hellman supplémentaires pour l'échange de clés sur Internet (IKE)", RFC 3526, mai 2003.

[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.

[RANDOM] D. Eastlake 3, J. Schiller et S. Crocker, "Exigences d'aléa pour la sécurité", BCP 106, RFC 4086, juin 2005.

[RFC3749] S. Hollenbeck, "Méthodes de compression du protocole de sécurité de la couche Transport", RFC 3749, mai 2004.

[RFC4366] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen et T. Wright, "Extensions de la sécurité de la couche Transport (TLS)", RFC 4366, avril 2006. *(Rendue obsolète par la RFC 5246, août 2008)*

[RSA] R. Rivest, A. Shamir, et L. M. Adleman, "A Method for Obtaining Digital Signatures et Public-Key Cryptosystems", Communications of the ACM, v. 21, n. 2, Feb 1978, pp. 120-126.

[SEQNUM] S. Bellovin, S., "Défense contre les attaques de numéro de séquence", RFC 1948, mai 1996.

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

[SSL3] A. Freier, P. Karlton, et P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.

[SUBGROUP] R. Zuccherato, "Méthodes pour éviter les attaques de "petit sous-groupe" dans la méthode d'accord de clés Diffie-Hellman pour S/MIME", RFC 2785, mars 2000.

[TCP] J. Postel, "Protocole de contrôle de Transmission ", STD 7, RFC 793, septembre 1981. *(Mise à jour par les RFC 1122 et 3168)*

[TIMING] D. Boneh, D. Brumley, "Remote timing attacks are practical", USENIX Security Symposium 2003.

[TLSAES] P. Chown, "Suites de chiffrement de la norme de chiffrement évolué (AES) pour la sécurité de la couche Transport (TLS)", RFC 3268, juin 2002. *(Rendue obsolète par la RFC 5246, août 2008)*

[TLSECC] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk et B. Moeller, "Suites de chiffrement de cryptographie de courbe elliptique (ECC) pour la sécurité de la couche Transport (TLS)", RFC 4492, mai 2006. *(Mise à jour par la RFC 5246, août 2008)*

[TLSEXT] D. Eastlake 3, "Extensions de sécurité de la couche Transport (TLS) : Définitions d'extensions", Travail en cours, février 2008.

[TLSPGP] N. Mavrogiannopoulos, "Utilisation des clés OpenPGP pour l'authentification de la sécurité de la couche Transport (TLS)", RFC 5081, novembre 2007.

[TLSPSK] P. Eronen, éd., et H. Tschofenig, éd., "Suites de chiffrement de clés pré partagées pour la sécurité de la couche Transport (TLS)", RFC 4279, décembre 2005.

[TLS1.0] T. Dierks et C. Allen, "Protocole TLS version 1.0", RFC 2246, janvier 1999.

[TLS1.1] T. Dierks et E. Rescorla, "Protocole de sécurité de la couche Transport (TLS) version 1.1", RFC 4346, avril 2006. *(Rendue obsolète par la RFC 5246, août 2008)*

[X501] Recommandation UIT-T X.501 : Technologies de l'information - Interconnexion des systèmes ouverts – L'annuaire : Modèles, 1993.

[XDR] M. Eisler, éd., "XDR : norme pour la représentation des données externes", STD 67, RFC 4506, mai 2006.

Informations sur le groupe de travail

La liste de diffusion du groupe de travail TLS de l'IETF est située à l'adresse de messagerie électronique <tls@ietf.org>. Les informations sur le groupe et la façon de s'inscrire sur la liste de diffusion figurent à l'adresse <https://www1.ietf.org/mailman/listinfo/tls>

Les archives de la liste se trouvent à <http://www.ietf.org/mail-archive/web/tls/current/index.html>

Contributeurs

Christopher Allen (coéditeur de TLS 1.0) Alacrity Ventures ; ChristopherA@AlacrityManagement.com
Martin Abadi, University of California, Santa Cruz ; abadi@cs.ucsc.edu
Steven M. Bellovin, Columbia University ; smb@cs.columbia.edu
Simon Blake-Wilson, BCI ; sblakewilson@bcisse.com
Ran Canetti, IBM ; canetti@watson.ibm.com
Pete Chown, Skygate Technology Ltd ; pc@skygate.co.uk
Taher Elgamal, taher@securify.com ; Securify
Pasi Eronen, pasi.eronen@nokia.com ; Nokia
Anil Gangolli, anil@busybuddha.org
Kipp Hickman
Alfred Hoenes
David Hopwood, Independent Consultant ; david.hopwood@blueyonder.co.uk
Phil Karlton (co-auteur de SSLv3)
Paul Kocher (co-auteur de SSLv3), Cryptography Research ; paul@cryptography.com
Hugo Krawczyk, IBM ; hugo@ee.technion.ac.il
Jan Mikkelsen, Transactionware ; janm@transactionware.com
Magnus Nystrom, RSA Security ; magnus@rsasecurity.com
Robert Relyea, Netscape Communications ; relyea@netscape.com
Jim Roskind, Netscape Communications ; jar@netscape.com
Michael Sabin
Dan Simon, Microsoft, Inc. ; dansimon@microsoft.com
Tom Weinstein
Tim Wright, Vodafone ; timothy.wright@vodafone.com

Adresse des éditeurs

Tim Dierks
Independent
mél : tim@dierks.org

Eric Rescorla
RTFM, Inc.
mél : ekr@rtfm.com

Déclaration de copyright

Copyright (C) The IETF Trust (2008).

Le présent document est soumis aux droits, licences et restrictions contenus dans le BCP 78, et sauf pour ce qui est mentionné ci-après, les auteurs conservent tous leurs droits.

Le présent document et les informations y contenues sont fournies sur une base "EN L'ÉTAT" et LE CONTRIBUTEUR, L'ORGANISATION QU'IL OU ELLE REPRÉSENTE OU QUI LE/LA FINANCE (S'IL EN EST), LA INTERNET SOCIETY, LE IETF TRUST ET LA INTERNET ENGINEERING TASK FORCE DÉCLINENT TOUTES GARANTIES, EXPRIMÉES OU IMPLICITES, Y COMPRIS MAIS NON LIMITÉES À TOUTE GARANTIE QUE L'UTILISATION DES INFORMATIONS CI-ENCLOSES NE VIOLENT AUCUN DROIT OU AUCUNE GARANTIE IMPLICITE DE COMMERCIALISATION OU D'APTITUDE À UN OBJET PARTICULIER.

Propriété intellectuelle

L'IETF ne prend pas position sur la validité et la portée de tout droit de propriété intellectuelle ou autres droits qui pourrait être revendiqués au titre de la mise en œuvre ou l'utilisation de la technologie décrite dans le présent document ou sur la mesure dans laquelle toute licence sur de tels droits pourrait être ou n'être pas disponible ; pas plus qu'elle ne prétend avoir accompli aucun effort pour identifier de tels droits. Les informations sur les procédures de l'ISOC au sujet des droits dans les documents de l'ISOC figurent dans les BCP 78 et BCP 79.

Des copies des dépôts d'IPR faites au secrétariat de l'IETF et toutes assurances de disponibilité de licences, ou le résultat de tentatives faites pour obtenir une licence ou permission générale d'utilisation de tels droits de propriété par ceux qui mettent en œuvre ou utilisent la présente spécification peuvent être obtenues sur répertoire en ligne des IPR de l'IETF à http://www.ietf.org/ipr.

L'IETF invite toute partie intéressée à porter son attention sur tous copyrights, licences ou applications de licence, ou autres droits de propriété qui pourraient couvrir les technologies qui peuvent être nécessaires pour mettre en œuvre la présente norme. Prière d'adresser les informations à l'IETF à ietf-ipr@ietf.org.