Internet Engineering Task Force (IETF)
Request for Comments : 8446
RFC rendues obsolètes : 5077, 5246, 6961
RFC mises à jour : 5705, 6066
Catégorie : En cours de normalisation
ISSN: 2070-1721

E. Rescorla, Mozilla
août 2018

Traduction Claude Brière de L'Isle

Protocole de sécurité de la couche transport (TLS) version 1.3

Résumé
Le présent document spécifie la version 1.3 du protocole de sécurité de la couche Transport (TLS, *Transport Layer Security*). TLS permet aux applications client/serveur de communiquer sur l'Internet d'une façon qui est conçue pour empêcher l'espionnage, l'altération, et la falsification de message.

Le présent document met à jour les RFC 5705 et 6066, et rend obsolètes les RFC 5077, 5246, et 6961. Il spécifie aussi de nouvelles exigences pour les mises en œuvre de TLS 1.2.

Statut de ce mémoire
Ceci est un document de l'Internet en cours de normalisation.

Le présent document a été produit par l'équipe d'ingénierie de l'Internet (IETF). Il représente le consensus de la communauté de l'IETF. Il a subi une révision publique et sa publication a été approuvée par le groupe de pilotage de l'ingénierie de l'Internet (IESG). Plus d'informations sur les normes de l'Internet sont disponibles à la Section 2 de la [RFC7841].

Les informations sur le statut actuel du présent document, tout errata, et comment fournir des réactions sur lui peuvent être obtenues à http://www.rfc-editor.org/info/rfc8446

Table des Matières

1. Introduction

Le but principal de TLS est de fournir un canal sûr entre deux homologues communicants ; la seule exigence provenant du transport sous-jacent est celle d'un flux de données fiable, dans l'ordre. Précisément, le canal sûr devrait assurer les propriétés suivantes :

Authentification : le côté serveur du canal est toujours authentifié ; le côté client est facultativement authentifié. L'authentification peut se faire via un chiffrement asymétrique (par exemple, [RSA], l'algorithme de signature numérique à courbe elliptique [ECDSA], ou l'algorithme de signature numérique à courbe de Edwards (EdDSA) [RFC8032]) ou une clé pré partagée (PSK, *Pre-Shared Key*) symétrique.

Confidentialité : les données envoyées sur le canal après l'établissement sont seulement visibles aux points d'extrémité. TLS ne cache pas la longueur des données transmises, mais les points d'extrémité sont capables de bourrer les enregistrements TLS afin d'obscurcir les longueurs et améliorer la protection contre les techniques d'analyse du trafic.

Intégrité : les données envoyées sur le canal après l'établissement ne peuvent pas être modifiées sans détection par des attaquants.

Ces propriétés devraient être vraies même en face d'un attaquant qui a le contrôle complet du réseau, comme décrit dans la [RFC3552]. Voir à l'Appendice E une déclaration plus complète des propriétés de sécurité pertinentes.

TLS comporte deux composants principaux :

- Un protocole de prise de contact (Section 4) qui authentifie les partie communicantes, négocie les modes et paramètres de chiffrement, et établit le matériel de chiffrement partagé. Le protocole de prise de contact est conçu pour résister à l'altération ; un attaquant actif ne devrait pas être capable de forcer les homologues à négocier des paramètres différents de ceux qu'ils auraient si la connexion n'était pas attaquée.

- Un protocole d'enregistrement (Section 5) qui utilise les paramètres établis par le protocole de prise de contact pour protéger le trafic entre les homologues communicants. Le protocole d'enregistrement divise le trafic en une série d'enregistrements, dont chacun est protégé indépendamment en utilisant les clés de trafic.

TLS est indépendant du protocole d'application ; les protocoles de niveau supérieur peuvent se poser en toute transparence au dessus de TLS. La norme TLS ne spécifie cependant pas comment les protocoles augmentent leur sécurité avec TLS ; la façon d'initier la prise de contact TLS et comment interpréter les certificats d'authentification échangés sont laissés au jugement des concepteurs et développeurs de protocoles qui fonctionnent par dessus TLS.

Le présent document définit TLS version 1.3. Bien que TLS 1.3 ne soit pas directement compatible avec les versions précédentes, toutes les versions de TLS incorporent un mécanisme de prise en compte de la version qui permet aux clients et serveurs de négocier de façon interopérable une version commune si il en est une qui est prise en charge par les deux homologues.

Le présent document se substitue et rend obsolètes les versions précédentes de TLS, incluant la version 1.2 [RFC5246]. Il rend aussi obsolète le mécanisme de ticket TLS défini dans la [RFC5077] et le remplace par le mécanisme défini au paragraphe 2.2. Parce que TLS 1.3 change la façon dont les clés sont déduites, il met à jour la [RFC5705] comme décrit au paragraphe 7.5. Il change aussi comment sont portés les messages du protocole d'état de certificat en ligne (OCSP, *Online Certificate Status Protocol*) et met donc à jour la [RFC6066] et rend obsolète la [RFC6961] comme décrit au paragraphe 4.4.2.1.

1.1 Conventions et terminologie

Les mots clés "DOIT", "NE DOIT PAS", "EXIGE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS",

"RECOMMANDE", "PEUT", et "FACULTATIF" en majuscules dans ce document sont à interpréter comme décrit dans le BCP 14, [RFC2119], [RFC8174] quand, et seulement quand ils apparaissent tout en majuscules, comme montré ci-dessus.

Les termes suivants sont utilisés :

client : point d'extrémité qui initie la connexion TLS.

connexion : connexion de couche transport entre deux points d'extrémité.

point d'extrémité : le client ou le serveur de la connexion.

prise de contact : négociation initiale entre client et serveur qui établit les paramètres de leurs interactions ultérieures au sein de TLS.

homologue : point d'extrémité. Quand on discute d'un point d'extrémité particulier, "homologue" se réfère au point d'extrémité qui n'est pas le principal sujet de discussion.

receveur : point d'extrémité qui reçoit les enregistrements.

envoyeur : point d'extrémité qui transmet les enregistrements.

serveur: point d'extrémité qui n'a pas initié la connexion TLS.

1.2 Différences majeures avec TLS 1.2

Voici une liste des différences fonctionnelles majeures entre TLS 1.2 et TLS 1.3. Elle n'est pas destinée à être exhaustive, et il y a de nombreuses différences mineures.

- La liste des algorithmes de chiffrement symétrique pris en charge a été élaguée de tous les algorithmes qui sont considérés comme un héritage. Ceux qui restent sont tous des algorithmes de chiffrement authentifié avec données supplémentaires (AEAD, *Authenticated Encryption with Additional Data*). Le concept de suite de chiffrement a été changé pour séparer les mécanismes d'authentification et d'échange de clé de l'algorithme de protection d'enregistrement (incluant la longueur de la clé secrète) et un hachage à utiliser par la fonction de déduction de clé et le code d'authentification de message (MAC) de prise de contact.
- Un mode de délai d'aller retour de zéro (0-RTT) a été ajouté, épargnant un aller retour à l'établissement de la connexion pour certaines données d'application, au prix de certaines propriétés de sécurité.
- Les suites de chiffrement RSA et Diffie-Hellman statiques ont été supprimées ; tous les mécanismes d'échange de clé fondés sur la clé publique fournissent maintenant le secret vers l'avant.
- Tous les messages de prise de contact après le ServerHello sont maintenant chiffrés. Le message nouvellement introduit EncryptedExtensions permet à diverses extensions précédemment envoyées en clair dans le ServerHello de bénéficier aussi de la protection de la confidentialité.
- Les fonctions de déduction de clé ont été redessinées. La nouvelle conception permet une analyse plus facile par les cryptologues du fait de leurs propriétés de séparation de clé améliorées. La fonction de déduction de clé par extraction et expansion fondée sur HMAC (HKDF, *HMAC-based Extract-et-Expand Key Derivation Function*) est utilisée comme primitive sous-jacente.
- L'automate à états de prise de contact a été significativement restructuré pour être plus cohérent et pour supprimer des messages superflus tels que ChangeCipherSpec (sauf lorsque nécessaire pour la compatibilité des boîtiers de médiation).
- Les algorithmes de courbe elliptique sont maintenant dans la spécification de base, et de nouveaux algorithmes de signature, comme EdDSA, sont inclus. TLS 1.3 a supprimé la négociation de format de point en faveur d'un format à un seul point pour chaque courbe.
- D'autres améliorations cryptographiques ont été faites, incluant de changer le bourrage RSA pour utiliser le schéma de signature probabiliste RSA (RSASSA-PSS, *RSA Probabilistic Signature Scheme*), et la suppression de la compression, de l'algorithme de signature numérique (DSA, *Digital Signature Algorithm*), et des groupes Diffie-Hellman éphémères (DHE, *Ephemeral Diffie-Hellman*) personnalisés.
- Le mécanisme de négociation de version de TLS 1.2 a été déconseillé en faveur d'une liste de versions dans une extension. Cela augmente la compatibilité avec les serveurs existants qui mettent incorrectement en œuvre la négociation de version.
- La reprise de session avec et sans état du côté serveur ainsi que les suites de chiffrement fondées sur PSK des versions antérieures de TLS ont été remplacées par un seul nouvel échange PSK.
- Les références ont été mises à jour pour pointer sur les dernières versions des RFC, comme approprié (par exemple, la RFC 5280 plutôt que la RFC 3280).

1.3 Mises à jour affectant TLS 1.2

Le présent document définit plusieurs changements qui peuvent affecter les mises en œuvre de TLS 1.2, incluant celles qui ne prennent pas non plus en charge TLS 1.3 :
- un mécanisme de protection contre la dégradation de version est décrit au paragraphe 4.1.3 ;
- les schémas de signature RSASSA-PSS sont définis au paragraphe 4.2.3 ;
- l'extension de ClientHello "supported_versions" peut être utilisée pour négocier la version de TLS à utiliser, de préférence au champ legacy_version du ClientHello ;
- l'extension "signature_algorithms_cert" permet à un client d'indiquer quels algorithmes de signature il peut valider dans les certificats X.509.

De plus, le présent document précise certaines exigences de conformité pour les versions antérieures de TLS; voir au paragraphe 9.3.

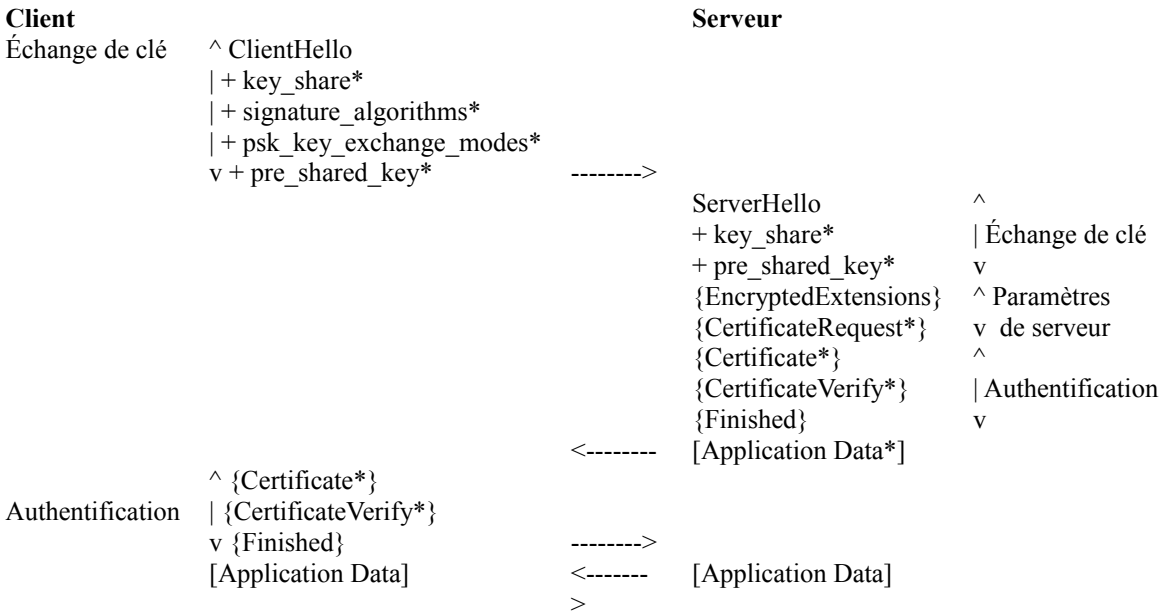
2. Vue d'ensemble du protocole

Les paramètres cryptographiques utilisés par le canal sûr sont produits par le protocole de prise de contact TLS. Ce sous protocole de TLS est utilisé par le client et le serveur lorsque ils communiquent ensemble pour la première fois. Le protocole de prise de contact permet aux homologues de négocier une version de protocole, choisir des algorithmes de chiffrement, facultativement de s'authentifier l'un l'autre, et d'établir le matériel de chiffrement de secret partagé. Une fois la prise de contact achevée, les homologues utilisent les clé établies pour protéger le trafic de couche application.

Un échec de la prise de contact ou une autre erreur de protocole déclenche la terminaison de la connexion, facultativement précédée par un message d'alerte (Section 6).

TLS prend en charge trois modes de base d'échange de clé :
- (EC)DHE (Diffie-Hellman sur champs finis ou sur courbes elliptiques)
- PSK seulement
- PSK avec (EC)DHE

La Figure 1 ci-dessous montre la prise de contact TLS de base complète :



\+ Indique les extensions notables envoyées dans le message noté précédemment.
* Indique les messages/extensions facultatifs ou qui dépendent de la situation qui ne sont pas toujours envoyés.
{} Indique les messages protégés en utilisant des clés déduites d'un [sender]_handshake_traffic_secret.
[] Indique les messages protégés en utilisant des clés déduites d'un [sender]_application_traffic_secret_N.

Figure 1 : Flux de messages pour la prise de contact TLS complète

La prise de contact peut être vue comme ayant trois phases (indiquées dans le diagramme ci-dessus) :

- Échange de clé : établir le matériel de chiffrement partagé et choisir les paramètres de chiffrement. Après cette phase, tout est chiffré.
- Paramètres de serveur : établir les autres paramètres de prise de contact (l'authentification du client, la prise en charge du protocole de couche application, etc.).
- Authentification : authentifie le serveur (et, facultativement le client) et assure la confirmation de clé et l'intégrité de la prise de contact.

Dans la phase d'échange de clé, le client envoie le message ClientHello (paragraphe 4.1.2) qui contient un nom occasionnel aléatoire (ClientHello.random) ; il offre des versions de protocole, une liste de paires de chiffrement/hachage HKDF symétriques, un ensemble de partages de clé Diffie-Hellman (dans l'extension "key_share" (paragraphe 4.2.8)), un ensemble d'étiquettes de clé pré partagée (dans l'extension "pre_shared_key" (paragraphe 4.2.11)), ou les deux, et éventuellement des extensions supplémentaires. D'autres champs et/ou messages peuvent aussi être présents pour la compatibilité des boîtiers de médiation *(middlebox)*.

Le serveur traite le ClientHello et détermine les paramètres de chiffrement appropriés pour la connexion. Il répond alors avec son propre ServerHello (paragraphe 4.1.3) qui indique les paramètres négociés de la connexion. La combinaison du ClientHello et du ServerHello détermine les clés partagées. Si l'établissement de clé (EC)DHE est utilisé, le ServerHello contient alors une extension "key_share" avec la part Diffie-Hellman éphémère du serveur ; la part du serveur DOIT être dans le même groupe qu'une des parts du client. Si l'établissement de clé PSK est utilisé, le ServerHello contient alors une extension "pre_shared_key" qui indique laquelle des PSK offertes par le client a été choisie. Noter que les mises en œuvre peuvent utiliser (EC)DHE et PSK ensemble, et dans ce cas les deux extensions seront fournies.

Le serveur envoie alors deux messages pour établir les paramètres de serveur :

EncryptedExtensions : les réponses aux extensions de ClientHello qui ne sont pas exigées pour déterminer les paramètres de chiffrement, autres que ceux qui sont spécifiques de certificats individuels (paragraphe 4.3.1).

CertificateRequest : si l'authentification du client fondée sur des certificats est désirée, les paramètres désirés pour ce certificat. Ce message est omis si l'authentification du client n'est pas désirée (paragraphe 4.3.2).

Finalement, le client et le serveur échangent les messages d'authentification. TLS utilise le même ensemble de messages chaque fois que l'authentification fondée sur le certificat est nécessaire. (L'authentification fondée sur la PSK se produit comme effet secondaire de l'échange de clés.) Précisément :

Certificate : le certificat du point d'extrémité et toutes les extensions par certificat. Ce message est omis par le serveur si il ne s'authentifie pas avec un certificat et par le client si le serveur n'a pas envoyé CertificateRequest (indiquant donc que le client ne devrait pas s'authentifier avec un certificat). Noter que si des clés publiques brutes [RFC7250] ou l'extension d'informations en mémoire tampon [RFC7924] sont utilisées, ce message ne contiendra alors pas de certificat mais plutôt une autre valeur correspondant à la clé à long terme du serveur (paragraphe 4.4.2).

CertificateVerify : une signature sur la prise de contact entière utilisant la clé privée correspondant à la clé publique dans le message Certificate. Ce message est omis si le point d'extrémité ne s'authentifie pas via un certificat (paragraphe 4.4.3).

Finished : un code d'authentification de message (MAC, *Message Authentification Code*) sur la prise de contact entière. Ce message donne la confirmation de clé, lie l'identité du point d'extrémité aux clés échangées, et en mode PSK authentifie aussi la prise de contact (paragraphe 4.4.4).

À réception des messages du serveur, le client répond avec ses messages Authentication, à savoir Certificate et CertificateVerify (si il est demandé), et Finished.

À ce point, la prise de contact est achevée, et client et serveur déduisent le matériel de chiffrement requis par la couche d'enregistrement pour échanger les données de couche application protégées par un chiffrement authentifié. Les données d'application NE DOIVENT PAS être envoyées avant d'envoyer le message Finished, sauf comme spécifié au paragraphe 2.3. Noter que alors que le serveur peut envoyer des données d'application avant de recevoir les messages d'authentification du client, toutes les données envoyées à ce moment sont, bien sûr, envoyées à un homologue non authentifié.

2.1 Partage de DHE incorrect

Si le client n'a pas fourni une extension "key_share" suffisante (par exemple, il a inclus seulement des groupes DHE ou ECDHE inacceptables ou non pris en charge par le serveur) le serveur corrige la discordance avec une HelloRetryRequest

et le client doit redémarrer la prise de contact avec une extension "key_share" appropriée, comme montré à la Figure 2. Si un paramètre de chiffrement commun ne peut être négocié, le serveur DOIT interrompre la prise de contact avec une alerte appropriée.

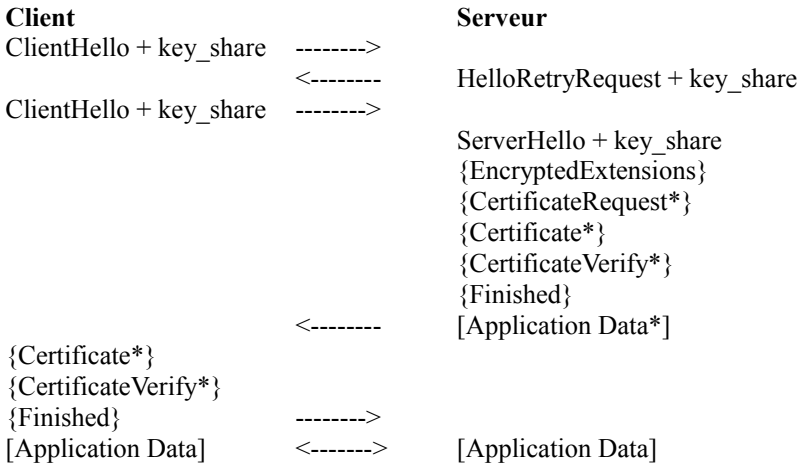


Figure 2 : Flux de messages pour une prise de contact complète avec des paramètres discordants

Note : La transcription de prise de contact incorpore l'échange initial ClientHello/HelloRetryRequest ; elle n'est pas réinitialisé avec le nouveau ClientHello.

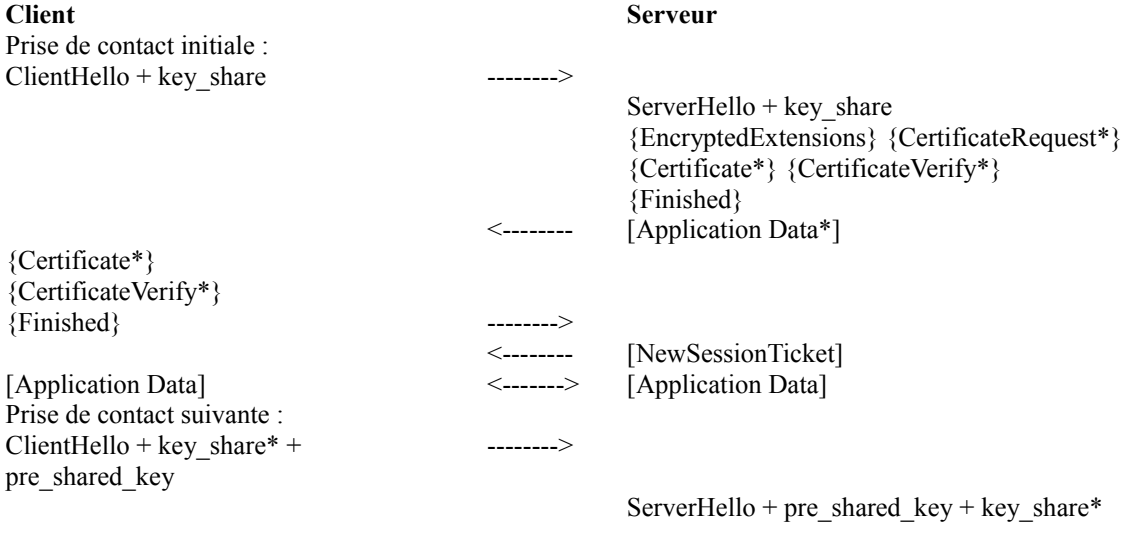
TLS permet aussi plusieurs variantes optimisées de la prise de contact de base, comme décrit dans les paragraphes suivants.

2.2 Reprise et clé pré partagée (PSK, *Pre-Shared Key*)

Bien que les PSK TLS puissent être établies hors bande, les PSK peuvent aussi être établies dans une connexion précédente et ensuite utilisées pour établir une nouvelle connexion ("reprise de session" ou "reprise" avec une PSK). Une fois la prise de contact achevée, le serveur peut envoyer au client une identité de PSK qui correspond à une clé unique déduite de la prise de contact initiale (voir au paragraphe 4.6.1). Le client peut alors utiliser cette identité de PSK dans de futures prises de contact pour négocier l'utilisation de la PSK associée. Si le serveur accepte la PSK, le contexte de sécurité de la nouvelle connexion est lié cryptographiquement à la connexion d'origine et la clé déduite de la prise de contact initiale est utilisée pour fixer l'état cryptographique plutôt qu'une pleine prise de contact. Dans TLS 1.2 et en dessous, cette fonctionnalité était fournie par les "identifiants de session" et les "tickets de session" [RFC5077]. Ces deux mécanismes sont obsolètes dans TLS 1.3.

Les PSK peuvent être utilisées avec l'échange de clés (EC)DHE afin d'assurer le secret vers l'avant en combinaison avec les clés partagées, ou peuvent être utilisées seules, au prix de la perte du secret vers l'avant pour les données d'application.

La Figure 3 montre une paire de prises de contact dans lesquelles la première prise de contact établit une PSK et la seconde prise de contact l'utilise :



```
                                    {EncryptedExtensions} {Finished}
                          <--------  [Application Data*]
{Finished}                -------->
[Application Data]        <------->  [Application Data]
```

Figure 3 : Flux de message pour reprise et PSK

Comme le serveur s'authentifie via une PSK, il n'envoie pas un message Certificate ou CertificateVerify. Lorsque un client offre la reprise via une PSK, il DEVRAIT aussi fournir une extension "key_share" au serveur pour lui permettre de refuser la reprise et revenir à une pleine prise de contact, si nécessaire. Le serveur répond avec une extension "pre_shared_key" pour négocier l'utilisation de l'établissement de clé PSK et peut (comme on le montre ici) répondre avec une extension "key_share" pour faire l'établissement de clé (EC)DHE, assurant donc le secret de transmission.

Lorsque les PSK sont provisionnées hors bande, l'identité de PSK et l'algorithme de hachage KDF à utiliser avec la PSK DOIVENT aussi être provisionnés.

Note : lorsque on utilise un secret pré partagé provisionné hors bande, une considération critique est d'utiliser une entropie suffisante durant la génération de clé, comme discuté dans la [RFC4086]. Déduire un secret partagé d'un mot de passe ou autre source à faible entropie n'est pas sûr. Un secret, ou mot de passe à faible entropie est sujet à des attaques de dictionnaire sur la base du liant de PSK. L'authentification de PSK spécifiée n'est pas un échange de clé authentifié sur la base d'un mot de passe fort même lorsque elle est utilisée avec l'établissement de clé Diffie-Hellman. Précisément, elle n'empêche pas un attaquant qui peut observer la prise de contact d'effectuer une attaque en force brute sur le mot de passe/clé pré partagée.

2.3 Données 0-RTT

Lorsque clients et serveurs partagent une PSK (obtenue de l'extérieur ou via une prise de contact antérieure), TLS 1.3 permet aux clients d'envoyer des données dès le premier vol ("données précoces"). Le client utilise la PSK pour authentifier le serveur et chiffrer les données précoces.
Comme le montre la Figure 4, les données 0-RTT sont juste ajoutées à la prise de contact 1-RTT dans le premier vol. Le reste de la prise de contact utilise les mêmes messages que pour une prise de contact 1-RTT avec reprise de PSK.

```
Client                                  Serveur
ClientHello + early_data
 + key_share*+ psk_key_exchange_modes
+ pre_shared_key
(Application Data*)        -------->
                                        ServerHello
                                        + pre_shared_key + key_share*
                                        {EncryptedExtensions}
                                        + early_data*
                                        {Finished}
                           <--------    [Application Data*]
(EndOfEarlyData)
      {Finished}           --------->
      [Application Data]   <-------->   [Application Data]
```

\+ Indique les extensions notables envoyées dans le message noté précédemment.
* Indique les messages/extensions facultatifs ou qui dépendent de la situation qui ne sont pas toujours envoyés.
() Indique les messages protégés avec des clés déduites d'un client_early_traffic_secret.
{} Indique les messages protégés en utilisant des clés déduites d'un [sender]_handshake_traffic_secret.
[] Indique les messages protégés en utilisant des clés déduites d'un [sender]_application_traffic_secret_N.

Figure 4 : Flux de messages pour une prise de contact 0-RTT

Note importante : les propriétés de sécurité pour des données 0-RTT sont plus faibles que pour d'autres sortes de données TLS. Précisément : 1. Ces données ne sont pas transmises secrètement, car elles ne sont chiffrées que sous des clés déduites en utilisant la PSK offerte. 2. Il n'y a pas de garantie de non répétition entre des connexions. La protection contre la répétition pour les données TLS 1.3 ordinaires de 1-RTT est fournie via la valeur Random du serveur, mais les données 0-RTT ne dépendent pas du ServerHello et ont donc des garanties plus faibles. C'est particulièrement pertinent lorsque les données sont authentifiées soit avec l'authentification client TLS, soit dans le protocole d'application. Le même avertissement s'applique à toute utilisation de early_exporter_master_secret.

Les données 0-RTT ne peuvent pas être dupliquées au sein d'une connexion (c'est-à-dire, le serveur ne va pas traiter les mêmes données deux fois pour la même connexion) et un attaquant ne sera pas capable de faire apparaître les données 0-RTT comme étant des données 1-RTT (parce qu'elles sont protégées avec des clés différentes). L'Appendice E.5 contient une description des attaques potentielles, et la Section 8 décrit les mécanismes que le serveur peut utiliser pour limiter l'impact de la répétition.

3. Langage de présentation

Le présent document traite du formatage des données dans une représentation externe. On utilise la syntaxe de présentation très basique et définie assez grossièrement.

3.1 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 de plusieurs octets sont des enchaînements d'octets, de gauche à droite, du haut vers le bas. Dans le flux d'octets, un élément multi octets (numérique dans l'exemple qui suit) 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 courant des octets du réseau ou format gros boutien.

3.2 Divers

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

Les composants facultatifs sont notés en les enclosant entre "[[ ]]" (double crochets).

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

Un alias de type T' pour un type existant T est défini par: T T';

3.3 Nombres

Le type de données numériques de base est un octet non signé (uint8). Tous les plus grands types de données numériques sont construits à partir d'une série de longueur fixe d'octets enchaînés comme décrit au paragraphe 3.1 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 transmises dans l'ordre des octets du réseau (gros boutien) ; la valeur uint32 représentée par les octets hexadécimaux 01 02 03 04 est équivalente à la valeur décimale 16 909 060.

3.4 Vecteurs

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

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

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

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

Des vecteurs de longueur variable sont définis en spécifiant une sous gamme de longueurs légales, inclusivement, en utilisant la notation <plancher...plafond>. Lorsque ils sont codés, 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 de zéro est appelé un vecteur vide.

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

Dans l'exemple suivant, "obligatoire" est 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 le paragraphe 3.3). De même, "longer" peut représenter jusqu'à 800 octets de données, ou 400 éléments uint16, et il peut être vide. Son codage va inclure un champ de longueur réelle de deux octets ajouté en tête du vecteur. La longueur d'un vecteur codé doit être un multiple exact de la longueur d'un seul élément (par exemple, un vecteur de 17 octets de uint16 serait illégal).

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

3.5 Enumerated

Un type supplémentaire de données épars, appelé "enum" ou "enumerated", est disponible. Chaque définition est un type différent. Seuls des énumérés de même type peuvent être alloués ou comparés. Chaque élément d'une énumération doit recevoir une valeur, comme le montre l'exemple suivant. Comme les éléments de l'énumération ne sont pas ordonnés, ils peuvent recevoir toute valeur unique, dans n'importe quel ordre.

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

Des futures extensions ou ajouts au protocole pourront définir de nouvelles valeurs. Les mises en œuvre doivent être capables d'analyser et ignorer les valeurs inconnues sauf si la définition du champ en décide autrement.

Un énuméré occupe autant d'espace dans le flux d'octets que le ferait sa valeur maximale définie en ordinal. La définition suivante causerait l'utilisation d'un octet 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 qui suit, Goût va consommer deux octets dans le flux de données mais peut seulement assumer les valeurs 1, 2, ou 4 dans la version courante du protocole.

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

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

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

Les noms alloués pour énumérer n'ont pas besoin d'être uniques. La valeur numérique peut décrire une gamme sur laquelle le même nom s'applique. La valeur inclut les valeurs minimum et maximum incluses dans cette gamme, séparées par deux caractères "point". Ceci est principalement utile pour réserver des régions de l'espace.

```
enum { triste(0), gai(1..254), heureux(255) } Humeur;
```

3.6 Types construits

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

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

Les champs de vecteur de longueur fixe et variable sont permis en utilisant la syntaxe standard de vecteur. Les structures V1 et V2 dans les variantes d'exemple (paragraphe 3.8) le démontrent.

Les champs dans une structure peuvent être qualifiés en utilisant le nom du type, avec une syntaxe assez semblable à celle disponible pour les énumérés. Par exemple, T.f2 se réfère au second champ de la déclaration précédente.

3.7 Constantes

Les champs et variables peuvent recevoir une valeur fixe en utilisant "=", comme dans :

```
struct {
   T1 f1 = 8;     /* T.f1 doit toujours être 8 */
   T2 f2;
} T;
```

3.8 Variantes

Des structures définies peuvent avoir des variantes sur la base d'une certaine connaissance disponible dans l'environnement. Le sélecteur doit être d'un type énuméré qui définit les variantes possibles que définit la structure. Chaque branche de la sélection (ci-dessous) spécifie le type du champ de cette variante et une étiquette de champ facultative. Le mécanisme par lequel la variante est choisie au moment du démarrage n'est pas prescrit par le langage de présentation.

```
struct {
   T1 f1;
   T2 f2;
   ....
   Tn fn;
   select (E) {
      cas e1: Te1 [[fe1]];
      cas e2: Te2 [[fe2]];
      ....
      cas en: Ten [[fen]];
   };
} Tv;
```

Par exemple :

```
enum { pomme(0), orange(1) } VariantTag;

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

struct {
   uint32 number;
   opaque string[10];    /* longueur fixe */
} V2;

struct {
```

```
    VariantTag type;
    select (VariantRecord.type) {
      cas pomme:  V1;
      cas orange: V2;
    };
  } VariantRecord;
```

4. Protocole de prise de contact

Le protocole de prise de contact est utilisé pour négocier les paramètres de sécurité d'une connexion. Les messages de prise de contact sont fournis à la couche d'enregistrement TLS, où ils sont encapsulés dans une ou plusieurs structures TLSPlaintext ou TLSCiphertext qui sont traitées et transmises comme spécifié par l'état de la connexion actuellement active.

```
enum {
    client_hello(1),              (hello du client)
    server_hello(2),              (hello du serveur)
    new_session_ticket(4),        ticket de nouvelle session)
    end_of_early_data(5),         (fin des données précoces)
    encrypted_extensions(8),      (extensions chiffrées)
    certificate(11),              (certificat)
    certificate_request(13),      (demande de certificat)
    certificate_verify(15),       (vérification de certificat)
    finished(20),                 (terminé)
    key_update(24),               (mise à jour de clé)
    message_hash(254),            (hachage du message)
    (255)
 } HandshakeType;

    struct {
      HandshakeType msg_type;                 /* type de prise de contact */
      uint24 length;                          /* octets restants dans le message */
      select (Handshake.msg_type) {
        cas client_hello:          ClientHello;
        cas server_hello:          ServerHello;
        cas end_of_early_data:     EndOfEarlyData;
        cas encrypted_extensions:  EncryptedExtensions;
        cas certificate_request:   CertificateRequest;
        cas certificate:           Certificate;
        cas certificate_verify:    CertificateVerify;
        cas finished:              Finished;
        cas new_session_ticket:    NewSessionTicket;
        cas key_update:            KeyUpdate;
      };
    } Handshake;
```

Les messages de protocole DOIVENT être envoyés dans l'ordre défini au paragraphe 4.4.1 et montré dans les diagrammes de la Section 2. Un homologue qui reçoit un message de prise de contact dans un ordre inattendu DOIT interrompre la prise de contact avec une alerte "message inattendu".

Les nouveaux types de message de prise de contact sont alloués par l'IANA comme décrit à la Section 11.

4.1 Messages d'échange de clé

Les messages d'échange de clé sont utilisés pour déterminer les capacités de sécurité du client et du serveur et pour établir les secrets partagés, incluant les clés de trafic utilisées pour protéger le reste de la prise de contact et les données.

4.1.1 Négociation du chiffrement

Dans TLS, la négociation du chiffrement se fait avec le client qui offre les quatre ensembles d'options suivants dans son

ClientHello :
- Une liste de suites de chiffrement qui indique les paires d'algorithme AEAD et de hachage HKDF que le client accepte.
- Une extension "supported_groups" (paragraphe 4.2.7) qui indique les groupes (EC)DHE que le client accepte et une extension "key_share" (paragraphe 4.2.8) qui contient les parts (EC)DHE pour certains de ces groupes ou tous.
- Une extension "signature_algorithms" (paragraphe 4.2.3) qui indique les algorithmes de signature que le client peut accepter. Une extension "signature_algorithms_cert" (paragraphe 4.2.3) peut aussi être ajoutée pour indiquer les algorithmes de signature spécifiques des certificats.
- Une extension "pre_shared_key" (paragraphe 4.2.11) qui contient une liste d'identités de clés symétriques connues du client et une extension "psk_key_exchange_modes" (paragraphe 4.2.9) qui indique les modes d'échange de clé qui peuvent être utilisés avec les PSK.

Si le serveur ne choisit pas une PSK, les trois premières de ces options sont alors entièrement orthogonales : le serveur choisit indépendamment une suite de chiffrement, un groupe (EC)DHE et une part de clé pour l'établissement de clé, et une paire algorithme de signature/certificat pour s'authentifier auprès du client. Si il n'y a pas de recouvrement entre les "supported_groups" reçus et ceux acceptés par le serveur, le serveur DOIT alors interrompre la prise de contact avec une alerte "échec de prise de contact" ou "sécurité insuffisante".

Si le serveur choisit une PSK, il DOIT alors aussi choisir un mode d'établissement de clé dans l'ensemble indiqué par l'extension "psk_key_exchange_modes" du client (à présent, PSK seul ou avec (EC)DHE). Noter que si la PSK peut être utilisée sans (EC)DHE, le non recouvrement des paramètres "supported_groups" n'a pas besoin d'être fatal, comme ce l'est dans le cas non PSK exposé à l'alinéa précédent.

Si le serveur choisit un groupe (EC)DHE et si le client n'a pas offert une extension "key_share" compatible dans le ClientHello initial, le serveur DOIT répondre avec un message HelloRetryRequest (paragraphe 4.1.4).

Si le serveur choisit avec succès les paramètres et n'exige pas un HelloRetryRequest, il indique les paramètres choisis dans le ServerHello comme suit :
- Si une PSK est utilisée, le serveur va alors envoyer une extension "pre_shared_key" indiquant la clé choisie.
- Lorsque (EC)DHE est utilisé, le serveur va aussi fournir une extension "key_share". Si une PSK n'est pas utilisée, (EC)DHE et l'authentification fondée sur le certificat sont alors toujours utilisés.
- Lorsque il s'authentifie via un certificat, le serveur va envoyer les messages Certificate (paragraphe 4.4.2) et CertificateVerify (paragraphe 4.4.3). Dans TLS 1.3 comme défini par le présent document, une PSK ou un certificat est toujours utilisé, mais pas les deux. De futurs documents pourront définir comment les utiliser ensemble.

Si le serveur est dans l'incapacité de négocier un ensemble de paramètres pris en charge (c'est-à-dire, si il n'y a pas de recouvrement entre les paramètres du client et ceux du serveur) il DOIT interrompre la prise de contact avec une alerte fatale "échec de prise de contact" ou "sécurité insuffisante" (voir la Section 6).

4.1.2. Hello du client

Lorsque un client se connecte à un serveur, il est EXIGÉ qu'il commence par envoyer le ClientHello comme son premier message TLS. Le client va aussi envoyer un ClientHello lorsque le serveur a répondu à son ClientHello avec une HelloRetryRequest. Dans ce cas, le client DOIT envoyer le même ClientHello sans modification, sauf comme suit :
- Si une extension "key_share" a été fournie dans la HelloRetryRequest, remplaçant la liste des partages par une liste contenant une seule KeyShareEntry provenant du groupe indiqué.
- En supprimant l'extension "early_data" (paragraphe 4.2.10) si il en était une présente. Les données précoces ne sont pas permises après une HelloRetryRequest.
- En incluant une extension "cookie" si une a été fournie dans la HelloRetryRequest.
- En mettant à jour l'extension "pre_shared_key" si elle est présente, en recalculant les valeurs de "obfuscated_ticket_age" et de "binder" *(liant)* et (facultativement) en supprimant toutes les PSK qui sont incompatibles avec la suite de chiffrement indiquée par le serveur.
- En ajoutant, supprimant ou changeant facultativement la longueur de l'extension "bourrage" [RFC7685].
- D'autres modifications qui peuvent être permises par une extension définie à l'avenir et présente dans la HelloRetryRequest.

Parce que TLS 1.3 interdit la renégociation, si un serveur a négocié TLS 1.3 et reçoit un ClientHello à tout autre moment, il DOIT terminer la connexion avec une alerte "message inattendu".

Si un serveur a établi une connexion TLS avec une version antérieure de TLS et reçoit un ClientHello TLS 1.3 dans une renégociation, il DOIT conserver la version de protocole précédente. En particulier, il NE DOIT PAS négocier TLS 1.3.

Structure de ce message :

```
uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];                    /* sélecteur de suite de chiffrement */
struct {
ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
Random random;
opaque legacy_session_id<0..32>;
CipherSuite cipher_suites<2..2^16-2>;
opaque legacy_compression_methods<1..2^8-1>;
Extension extensions<8..2^16-1>;
} ClientHello;
```

legacy_version : Dans les versions précédentes de TLS, ce champ était utilisé pour la négociation de version et représentait le plus haut numéro de version supporté par le client. L'expérience a montré que de nombreux serveurs ne mettent pas correctement en œuvre la négociation de version, conduisant à une "intolérance de version" dans laquelle le serveur rejette un ClientHello par ailleurs acceptable avec un numéro de version supérieur à celui qu'il prend en charge. Dans TLS 1.3, le client indique ses préférences de version dans l'extension "supported_versions" (paragraphe 4.2.1) et le champ legacy_version DOIT être réglé à 0x0303, qui est le numéro de version pour TLS 1.2. Les ClientHello de TLS 1.3 sont identifiés comme ayant une extension legacy_version de 0x0303 et supported_versions présente avec 0x0304 comme plus haute version indiquée ainsi. (Voir à l'Appendice D les détails sur la rétro compatibilité.)

random : 32 octets générés par un générateur de nombres aléatoires sûr. Voir à l'Appendice C des informations supplémentaires.

legacy_session_id : Les versions de TLS avant TLS 1.3 prenaient en charge une caractéristique "session resumption" qui a été fusionnée avec les clés pré partagées dans cette version (voir au paragraphe 2.2). Un client qui a un identifiant de session en antémémoire établi par un serveur pré TLS 1.3 DEVRAIT régler ce champ à cette valeur. En mode compatibilité (voir l'Appendice D.4), ce champ DOIT être non vide, de sorte qu'un client qui n'offre pas une session pré TLS 1.3 DOIT générer une nouvelle valeur de 32 octets. Cette valeur n'a pas besoin d'être aléatoire mais DEVRAIT être imprévisible pour éviter que les mises en œuvre se fixent sur une valeur spécifique (aussi appelée une ossification). Autrement, elle DOIT être réglée à un vecteur de longueur zéro (c'est-à-dire, un champ d'un seul octet de valeur zéro).

cipher_suites : Liste des options de chiffrement symétrique prises en charge par le client, et précisément l'algorithme de protection d'enregistrement (incluant la longueur de la clé secrète) et un hachage à utiliser avec HKDF, en ordre décroissant des préférences du client. Les valeurs sont définies dans l'Appendice B.4. Si la liste contient des suites de chiffrement que le serveur ne reconnaît pas, ne prend pas en charge, ou ne veut pas utiliser, le serveur DOIT ignorer ces suites de chiffrement et traiter celles qui restent comme d'habitude. Si le client tente un établissement de clé PSK, il DEVRAIT annoncer au moins une suite de chiffrement en indiquant un hachage associé à la PSK.

legacy_compression_methods : Les versions de TLS antérieures à 1.3 prenaient en charge la compression avec la liste des méthodes de compression prises en charge qui était envoyée dans ce champ. Pour chaque ClientHello TLS 1.3, ce vecteur DOIT contenir exactement un octet, réglé à zéro, qui correspond à la méthode de compression "null" dans les versions antérieures de TLS. Si un ClientHello TLS 1.3 est reçu avec une autre valeur dans ce champ, le serveur DOIT interrompre la prise de contact avec une alerte "paramètre illégal". Noter que les serveurs TLS 1.3 peuvent recevoir des ClientHello TLS 1.2 ou antérieurs qui contiennent d'autres méthodes de compression et (si on négocie une telle version antérieure) DOIVENT suivre les procédures pour la version antérieure appropriée de TLS.

extensions : Les clients demandent aux serveurs des fonctionnalités étendues en envoyant de données dans le champ "extensions". Le format réel de "Extension" est défini au paragraphe 4.2. Dans TLS 1.3, l'utilisation de certaines extensions est obligatoire, car la fonctionnalité est passée dans les extensions pour préserver la compatibilité du ClientHello avec les versions précédentes de TLS. Les serveurs DOIVENT ignorer les extensions non reconnues.

Toutes les versions de TLS permettent qu'un champ "extensions" suive facultativement le champ "méthodes de compression". Les messages ClientHello TLS 1.3 contiennent toujours des extensions (au minimum les "supported_versions", autrement, ils seront interprétés comme des messages ClientHello TLS 1.2). Cependant, les serveurs TLS 1.3 peuvent recevoir des messages ClientHello sans un champ "extensions" provenant de versions antérieures de TLS. La présence des extensions peut être détectée en déterminant si il y a des octets à la suite du champ compression_methods à la fin du ClientHello. Noter que cette méthode de détection de données facultatives diffère de la méthode TLS normale d'avoir un champ de longueur variable, mais elle était utilisée pour la compatibilité avec TLS avant que les extensions soient définies. Les serveurs TLS 1.3 auront besoin d'effectuer cette vérification en premier et seulement tenter de négocier TLS 1.3 si l'extension "supported_versions" est présente. Si il négocie une version de TLS antérieure à 1.3, un serveur DOIT vérifier que le message soit ne contient pas de données après legacy_compression_methods, soit qu'il contient un

bloc d'extensions valide sans données à suivre. Sinon, il DOIT alors interrompre la prise de contact avec une alerte "erreur de décodage".

Dans le cas où un client demande une fonctionnalité supplémentaire en utilisant "extensions" et que cette fonctionnalité n'est pas fournie par le serveur, le client PEUT interrompre la prise de contact.

Après l'envoi du message ClientHello, le client attend un message ServerHello ou HelloRetryRequest. Si des données précoces sont utilisées, le client peut transmettre des données d'application précoces (paragraphe 2.3) tout en attendant le prochain message de prise de contact.

4.1.3 Hello du serveur

Le serveur va envoyer ce message en réponse à un message ClientHello pour la poursuite de la prise de contact si il est capable de négocier un ensemble acceptable de paramètres de prise de contact sur la base du ClientHello.

Structure de ce message :

```
struct {
    ProtocolVersion legacy_version = 0x0303;        /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

legacy_version : Dans les versions antérieures de TLS, ce champ était utilisé pour la négociation de version et représentait le numéro de version choisie pour la connexion. Malheureusement, certains boîtiers de médiation sont mis en échec lorsque on leur présente de nouvelles valeurs. Dans TLS 1.3, le serveur TLS indique sa version en utilisant l'extension "supported_versions" (paragraphe 4.2.1), et le champ legacy_version DOIT être réglé à 0x0303, qui est le numéro de version pour TLS 1.2. (Voir à l'Appendice D les détails sur la rétro compatibilité.)

random : 32 octets générés par un générateur de nombres aléatoires sûr. Voir à l'Appendice C des informations supplémentaires. Les 8 derniers octets DOIVENT être écrasés comme décrit ci-dessous si on négocie TLS 1.2 ou TLS 1.1, mais les octets restants DOIVENT être aléatoires. Cette structure est générée par le serveur et DOIT être générée indépendamment du ClientHello.random.

legacy_session_id_echo : contenu du champ legacy_session_id du client. Noter que ce champ est en écho même si la valeur du client correspondait à une session pré TLS 1.3 en antémémoire que le serveur a choisi de ne pas reprendre. Un client qui reçoit un champ legacy_session_id_echo qui ne correspond pas à ce qu'il a envoyé dans le ClientHello DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

cipher_suite : la seule suite de chiffrement choisie par le serveur sur la liste de ClientHello.cipher_suites. Un client qui reçoit une suite de chiffrement qui n'a pas été offerte DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

legacy_compression_method : un seul octet qui DOIT avoir la valeur 0.

extensions : liste des extensions. Le ServerHello DOIT seulement inclure les extensions qui sont nécessaires pour établir le contexte cryptographique et négocier la version de protocole. Tous les messages TLS 1.3 ServerHello DOIVENT contenir l'extension "supported_versions". Les messages courants ServerHello contiennent soit l'extension "pre_shared_key", soit l'extension "key_share", soit les deux (lorsque on utilise une PSK avec l'établissement de clé (EC)DHE). D'autres extensions (voir le paragraphe 4.2) sont envoyées séparément dans le message EncryptedExtensions.

Pour des raisons de rétro compatibilité avec les boîtiers de médiation (voir l'Appendice D.4) le message HelloRetryRequest utilise la même structure que le ServerHello, mais avec Random réglé à la valeur spéciale SHA-256 de "HelloRetryRequest" :

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

À réception d'un message de type server_hello, les mises en œuvre DOIVENT d'abord examiner la valeur Random et, si elle correspond à cette valeur, la traiter comme décrit au paragraphe 4.1.4).

TLS 1.3 a un mécanisme de protection de repli incorporé dans la valeur aléatoire du serveur. Les serveurs TLS 1.3 qui négocient TLS 1.2 ou en dessous en réponse à un ClientHello DOIVENT régler spécialement les 8 derniers octets de leur valeur Random dans leur ServerHello.

Si on négocie TLS 1.2, les serveurs TLS 1.3 DOIVENT régler les 8 derniers octets de leur valeur Random aux octets :

44 4F 57 4E 47 52 44 01

Si on négocie TLS 1.1 ou en dessous, les serveurs TLS 1.3 DOIVENT, et les serveurs TLS 1.2 DEVRAIENT, régler les 8 derniers octets de leur valeur ServerHello.Random aux octets :

44 4F 57 4E 47 52 44 00

Les clients TLS 1.3 qui reçoivent un ServerHello qui indique TLS 1.2 ou en dessous DOIVENT vérifier que les 8 derniers octets ne sont pas égaux à l'une de ces valeurs. Les clients TLS 1.2 DEVRAIENT aussi vérifier que les 8 derniers octets ne sont pas égaux à la seconde valeur si le ServerHello indique TLS 1.1 ou en dessous. Si une correspondance est trouvée, le client DOIT interrompre la prise de contact avec une alerte "paramètre_illégal". Ce mécanisme donne une protection limitée contre les attaques en dégradation au dessus et en dessous de ce qui est fourni par l'échange fini : à cause de ServerKeyExchange, un message présent dans TLS 1.2 et en dessous, inclut une signature sur les deux valeurs aléatoires, il n'est pas possible à un attaquant actif de modifier les valeurs aléatoires sans être détecté tant que des chiffrements éphémères sont utilisés. Cela ne fournit pas de protection contre la dégradation quand RSA statique est utilisé.

Note : Ceci est un changement par rapport à la [RFC5246], de sorte qu'en pratique de nombreux clients et serveurs TLS 1.2 ne vont pas se comporter comme spécifié ci-dessus.

Un client TLS traditionnel qui effectue une renégociation avec TLS 1.2 ou antérieur et qui reçoit un ServerHello TLS 1.3 durant la renégociation DOIT interrompre la prise de contact avec une alerte "protocol_version". Noter que la renégociation n'est pas possible quand TLS 1.3 a été négocié.

4.1.4 Demande de réessai de Hello

Le serveur va envoyer ce message en réponse à un message ClientHello si il est capable de trouver un ensemble acceptable de paramètres mais si le ClientHello ne contient pas des informations suffisantes pour poursuivre la prise de contact. Comme exposé au paragraphe 4.1.3, la HelloRetryRequest a le même format qu'un message ServerHello, et les champs legacy_version, legacy_session_id_echo, cipher_suite, et legacy_compression_method ont la même signification. Cependant, par facilité, on parlera dans ce document de "HelloRetryRequest" comme si c'était un message distinct.

Les extensions de serveur DOIVENT contenir "supported_versions". De plus, elles DEVRAIENT contenir l'ensemble minimal d'extensions nécessaire pour que le client génère une paire correcte de ClientHello. Comme avec le ServerHello, une HelloRetryRequest NE DOIT PAS contenir d'extensions qui n'aient été d'abord offertes par le client dans son ClientHello, à l'exception de l'extension facultative "cookie" (voir au paragraphe 4.2.2).

À réception d'une HelloRetryRequest, le client DOIT vérifier legacy_version, legacy_session_id_echo, cipher_suite, et legacy_compression_method comme spécifié au paragraphe 4.1.3 et traiter ensuite les extensions, en commençant par déterminer la version en utilisant "supported_versions". Les clients DOIVENT interrompre la prise de contact avec une alerte "paramètre_illégal" si la HelloRetryRequest ne résulterait pas en un changement dans le ClientHello. Si un client reçoit une seconde HelloRetryRequest sur la même connexion (c'est-à-dire, où le ClientHello était lui-même en réponse à une HelloRetryRequest) il DOIT interrompre la prise de contact avec une alerte "message inattendu".

Autrement, le client DOIT traiter toutes les extensions dans la HelloRetryRequest et envoyer un second ClientHello mis à jour. Les extensions de HelloRetryRequest définies dans la présente spécification sont :
- supported_versions (voir le paragraphe 4.2.1)
- cookie (voir le paragraphe 4.2.2)
- key_share (voir le paragraphe 4.2.8)

Un client qui reçoit une suite de chiffrement qui n'était pas offerte DOIT interrompre la prise de contact. Les serveurs DOIVENT s'assurer qu'ils négocient la même suite de chiffrement lorsque ils reçoivent un ClientHello mis à jour conforme (si le serveur choisit la suite de chiffrement comme première étape de la négociation, puis cela va se produire automatiquement). À réception du ServerHello, les clients DOIVENT vérifier que la suite de chiffrement fournie dans le

ServerHello est la même que dans la HelloRetryRequest et autrement interrompre la prise de contact avec une alerte "paramètre_illégal".

De plus, dans son ClientHello mis à jour, le client NE DEVRAIT PAS offrir de clé pré partagée associée à un hachage autre que celui de la suite de chiffrement choisie. Cela permet au client d'éviter d'avoir à calculer des transcriptions de hachage partiel pour plusieurs hachages dans le second ClientHello.

La valeur de selected_version dans l'extension "supported_versions" de la HelloRetryRequest DOIT être conservée dans le ServerHello, et un client DOIT interrompre la prise de contact avec une alerte "paramètre_illégal" si la valeur change.

4.2 Extensions

Un certain nombre de messages TLS contiennent des structures d'extensions codées en étiquette-longueur-valeur.

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

enum {
    server_name(0),                             (nom de serveur)                        /* RFC 6066 */
    max_fragment_length(1),                     (longueur maximum de fragment)          /* RFC 6066 */
    status_request(5),                          (demande d'état)                        /* RFC 6066 */
    supported_groups(10),                       (groupes pris en charge)                /* RFC 8422, 7919 */
    signature_algorithms(13),                   (algorithmes de signature)              /* RFC 8446 */
    use_srtp(14),                               (utiliser SRTP)                         /* RFC 5764 */
    heartbeat(15),                              (battement de cœur)                     /* RFC 6520 */
    application_layer_protocol_negotiation(16), (négociation protocole couche appli.)   /* RFC 7301 */
    signed_certificate_timestamp(18),           (horodatage de certificat signé)        /* RFC 6962 */
    client_certificate_type(19),                (type de certificat de client)          /* RFC 7250 */
    server_certificate_type(20),                type de certificat de serveur)          /* RFC 7250 */
    padding(21),                                (bourrage)                              /* RFC 7685 */
    pre_shared_key(41),                         (clé pré partagée)                      /* RFC 8446 */
    early_data(42),                             (données précoces)                      /* RFC 8446 */
    supported_versions(43),                     (versions prises en charge)             /* RFC 8446 */
    cookie(44),                                 (mouchard)                              /* RFC 8446 */
    psk_key_exchange_modes(45),                 (modes d'échange de clé pré partagée)   /* RFC 8446 */
    certificate_authorities(47),                (autorités de certification)            /* RFC 8446 */
    oid_filters(48),                            (filtres d'OID)                         /* RFC 8446 */
    post_handshake_auth(49),                    (authentification post prise de contact) /* RFC 8446 */
    signature_algorithms_cert(50),              (certificats d'algorithmes de signature) /* RFC 8446 */
    key_share(51),                              (partage de clé)                        /* RFC 8446 */
    (65535)
 } ExtensionType;
```

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

La liste des types d'extension est tenue par l'IANA comme décrit à la Section 11.

Les extensions sont généralement structurées par demande/réponse, bien que certaines extensions soient juste des indications sans réponse correspondante. Le client envoie ses demandes d'extension dans le message ClientHello, et le serveur envoie ses réponses d'extension dans les messages ServerHello, EncryptedExtensions, HelloRetryRequest, et Certificate. Le serveur envoie les demandes d'extension dans le message CertificateRequest auquel un client PEUT répondre avec un message Certificate. Le serveur PEUT aussi envoyer des extensions non sollicitées dans un NewSessionTicket, bien que le client n'y réponde pas directement.

Les mises en œuvre NE DOIVENT PAS envoyer de réponse d'extension si le point d'extrémité distant n'a pas envoyé de demande d'extension correspondante, à l'exception de l'extension "cookie" dans la HelloRetryRequest. À réception d'une telle extension, un point d'extrémité DOIT interrompre la prise de contact avec une alerte "extension non acceptée".

Le tableau ci-dessous indique les messages où une certaine extension peut apparaître, en utilisant la notation suivante : CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket), et HRR (HelloRetryRequest). Si une mise en œuvre reçoit une extension qu'elle ne reconnaît pas et qui n'est pas spécifiée pour le message dans laquelle elle apparaît, elle DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

Extension	**TLS 1.3**
server_name [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
status_request [RFC6066]	CH, CR, CT
supported_groups [RFC7919]	CH, EE
signature_algorithms (RFC 8446)	CH, CR
use_srtp [RFC5764]	CH, EE
heartbeat [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE
signed_certificate_timestamp [RFC6962]	CH, CR, CT
client_certificate_type [RFC7250]	CH, EE
server_certificate_type [RFC7250]	CH, EE
padding [RFC7685]	CH
key_share (RFC 8446)	CH, SH, HRR
pre_shared_key (RFC 8446)	CH, SH
psk_key_exchange_modes (RFC 8446)	CH
early_data (RFC 8446)	CH, EE, NST
cookie (RFC 8446)	CH, HRR
supported_versions (RFC 8446)	CH, SH, HRR
certificate_authorities (RFC 8446)	CH, CR
oid_filters (RFC 8446)	CR
post_handshake_auth (RFC 8446)	CH
signature_algorithms_cert (RFC 8446)	CH, CR

Lorsque plusieurs extensions de différents types sont présentes, les extensions PEUVENT apparaître dans n'importe quel ordre, à l'exception de "pre_shared_key" (paragraphe 4.2.11) qui DOIT être la dernière extension dans le ClientHello (mais peut apparaître n'importe où dans le bloc d'extensions ServerHello). Il NE DOIT PAS y avoir plus d'une extension du même type dans un bloc d'extension.

Dans TLS 1.3, à la différence de TLS 1.2, les extensions sont négociées pour chaque prise de contact même quand on est dans la PSK en mode reprise. Cependant, les paramètres 0-RTT sont ceux qui sont négociés dans la précédente prise de contact ; des discordances peuvent exiger de rejeter 0-RTT (voir au paragraphe 4.2.10).

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

Dans certains cas où un serveur n'est pas d'accord pour une extension, il y a des conditions d'erreur (par exemple, la prise de contact ne peut pas continuer) et dans d'autres c'est simplement un refus de prendre en charge des caractéristiques particulières. En général, les alertes d'erreur devraient être utilisées dans les premiers cas, et un champ dans la réponse d'extension du serveur dans les derniers.

- Les extensions devraient, autant que possible, être conçues pour empêcher toute attaque qui force l'utilisation (ou la non utilisation) d'une caractéristique particulière en manipulant les messages de prise de contact. Ce principe devrait être suivi sans considérer si la caractéristique est estimée poser un problème de sécurité. Souvent, le fait que les champs d'extension sont inclus dans les entrées des hachages du message Finished sera suffisant, mais une attention extrême est nécessaire lorsque l'extension change la signification des messages envoyés dans la phase de prise de contact. Les concepteurs de mises en œuvres devraient 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, supprimer, ou remplacer les extensions.

4.2.1 Versions prise en charge

```
struct {
    select (Handshake.msg_type) {
        cas client_hello:
            ProtocolVersion versions<2..254>;
        cas server_hello:                    /* et HelloRetryRequest */
            ProtocolVersion selected_version;
    };
} SupportedVersions;
```

L'extension "supported_versions" est utilisée par le client pour indiquer quelles versions de TLS il prend en charge et par le serveur pour indiquer quelle version il utilise. L'extension contient une liste des versions prises en charge dans l'ordre de préférence, avec la version préférée en premier. Les mises en œuvre de la présente spécification DOIVENT envoyer cette extension dans le ClientHello qui contient toutes les versions de TLS qu'elles sont prêtes à négocier (pour la présente spécification, cela signifie au minimum 0x0304, mais si il est permis de négocier des versions antérieures de TLS, elles DOIVENT aussi être présentes).

Si cette extension n'est pas présente, les serveurs qui se conforment à la présente spécification et qui prennent aussi en charge TLS 1.2 DOIVENT négocier TLS 1.2 ou antérieur comme spécifié dans la [RFC5246], même si ClientHello.legacy_version est 0x0304 ou ultérieur. Les serveurs PEUVENT interrompre la prise de contact à réception d'un ClientHello avec legacy_version 0x0304 ou ultérieur.

Si cette extension est présente dans le ClientHello, les serveurs NE DOIVENT PAS utiliser la valeur de ClientHello.legacy_version pour la négociation de version et DOIVENT utiliser seulement l'extension "supported_versions" pour déterminer les préférences du client. Les serveurs DOIVENT seulement choisir une version de TLS présente dans cette extension et DOIVENT ignorer toute version inconnue présente dans cette extension. Noter que ce mécanisme rend possible de négocier une version antérieure à TLS 1.2 si un des côtés prend en charge une gamme éparse. Les mises en œuvre de TLS 1.3 qui choisissent de prendre en charge des versions antérieures de TLS DEVRAIENT prendre en charge TLS 1.2. Les serveurs DOIVENT être prêts à recevoir des ClientHello qui incluent cette extension mais n'incluent pas 0x0304 dans la liste des versions.

Un serveur qui négocie une version de TLS antérieure à TLS 1.3 DOIT établir ServerHello.version et NE DOIT PAS envoyer l'extension "supported_versions". Un serveur qui négocie TLS 1.3 DOIT répondre en envoyant une extension "supported_versions" contenant la valeur de version choisie (0x0304). Il DOIT régler le champ ServerHello.legacy_version à 0x0303 (TLS 1.2). Les clients DOIVENT vérifier la présence de cette extension avant de traiter le reste du ServerHello (bien qu'ils doivent analyser le ServerHello afin de lire l'extension). Si cette extension est présente, les clients DOIVENT ignorer la valeur de ServerHello.legacy_version et DOIVENT utiliser seulement l'extension "supported_versions" pour déterminer la version choisie. Si l'extension "supported_versions" dans le ServerHello contient une version non offerte par le client ou contient une version antérieure à TLS 1.3, le client DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

4.2.2 Mouchard

```
struct {
    opaque cookie<1..2^16-1>;
} Cookie;
```

Les mouchards *(cookies)* servent deux objets principaux :

- Permettre au serveur de forcer le client à démontrer son accessibilité à son adresse réseau apparente (fournissant donc une mesure de protection contre le déni de service). Ceci est principalement utile pour les transports qui ne sont pas en mode connexion (voir un exemple de cela dans la [RFC6347]).

- Permettre au serveur de transmettre l'état au client, lui permettant ainsi d'envoyer une HelloRetryRequest sans mémoriser d'état. Le serveur peut faire cela en mémorisant le hachage du ClientHello dans le mouchard de HelloRetryRequest (protégé avec un algorithme de protection d'intégrité convenable).

Lorsque il envoie une HelloRetryRequest, le serveur PEUT fournir une extension "cookie" au client (c'est une exception à la règle usuelle que les seules extensions qui peuvent être envoyées sont celles qui apparaissent dans le ClientHello). Lors de l'envoi d'un nouveau ClientHello, le client DOIT copier le contenu de l'extension reçue dans la HelloRetryRequest dans une extension "cookie" dans le nouveau ClientHello. Les clients NE DOIVENT PAS utiliser de mouchards dans leur ClientHello initial dans les connexions suivantes.

Lorsque un serveur fonctionne sans état, il peut recevoir un enregistrement non protégé de type change_cipher_spec entre le premier et le second ClientHello (voir la Section 5). Comme le serveur ne mémorise aucun état, cela va apparaître comme si c'était le premier message reçu. Les serveurs qui fonctionnent avec conservation d'état DOIVENT ignorer ces enregistrements.

4.2.3 Algorithmes de signature

TLS 1.3 fournit deux extensions pour indiquer quels algorithmes de signature peuvent être utilisés dans les signatures numériques. L'extension "signature_algorithms_cert" s'applique aux signatures dans les certificats, et l'extension "signature_algorithms", qui apparaissait à l'origine dans TLS 1.2, s'applique aux signatures dans les messages CertificateVerify. Les clés qui se trouvent dans les certificats DOIVENT aussi être du type approprié pour les algorithmes de signature avec lesquels elles sont utilisées. C'est un problème particulier pour les clés RSA et les signatures PSS, comme décrit ci-dessous. Si aucune extension "signature_algorithms_cert" n'est présente, l'extension "signature_algorithms" s'applique alors aussi aux signatures qui apparaissent dans les certificats. Les clients qui désirent que le serveur s'authentifie via un certificat DOIVENT envoyer l'extension "signature_algorithms". Si un serveur s'authentifie via un certificat et si le client n'a pas envoyé une extension "signature_algorithms", le serveur DOIT alors interrompre la prise de contact avec une alerte "extension manquante" (voir au paragraphe 9.2).

L'extension "signature_algorithms_cert" a été ajoutée pour permettre aux mises en œuvre qui prennent en charge différents ensembles d'algorithmes pour les certificats et dans TLS lui-même pour signaler clairement leurs capacités. Les mises en œuvre TLS 1.2 DEVRAIENT aussi traiter cette extension. Les mises en œuvre qui ont la même politique dans les deux cas PEUVENT omettre l'extension "signature_algorithms_cert".

Le champ "extension_data" de ces extensions contient une valeur de SignatureSchemeList :

```
enum {
/* algorithmes RSASSA-PKCS1-v1_5 */
        rsa_pkcs1_sha256(0x0401),
        rsa_pkcs1_sha384(0x0501),
        rsa_pkcs1_sha512(0x0601),

/* algorithmes ECDSA */
        ecdsa_secp256r1_sha256(0x0403),
        ecdsa_secp384r1_sha384(0x0503),
        ecdsa_secp521r1_sha512(0x0603),

/* algorithmes RSASSA-PSS avec OID de clé publique rsaEncryption */
        rsa_pss_rsae_sha256(0x0804),
        rsa_pss_rsae_sha384(0x0805),
        rsa_pss_rsae_sha512(0x0806),

/* algorithmes EdDSA */
        ed25519(0x0807),
        ed448(0x0808),

/* algorithmes RSASSA-PSS avec OID de clé publique RSASSA-PSS */
        rsa_pss_pss_sha256(0x0809),
        rsa_pss_pss_sha384(0x080a),
        rsa_pss_pss_sha512(0x080b),

/*algorithmes traditionnels */
        rsa_pkcs1_sha1(0x0201),
        ecdsa_sha1(0x0203),

/* Codets réservés */
        private_use(0xFE00..0xFFFF),
        (0xFFFF)
    } SignatureScheme;

    struct {
        SignatureScheme supported_signature_algorithms<2..2^16-2>;
```

} SignatureSchemeList;

Note : Cette énumération est nommée "SignatureScheme" parce que il y a déjà un type "SignatureAlgorithm" dans TLS 1.2, que ceci remplace. On utilise le terme "algorithme de signature" tout au long du texte.

Chaque valeur de SignatureScheme donne un seul algorithme de signature que le client veut vérifier. Les valeurs sont indiquées en ordre décroissant de préférence. Noter qu'un algorithme de signature prend en entrée un message de longueur arbitraire, plutôt qu'un résumé. Les algorithmes qui agissent traditionnellement comme un résumé devraient être définis dans TLS comme hachant d'abord l'entrée avec un algorithme de hachage spécifié et ensuite poursuivre comme d'habitude. Les groupes de codets dont la liste figure ci-dessous ont la signification suivante :

algorithmes RSASSA-PKCS1-v1_5 : indique un algorithme de signature utilisant RSASSA-PKCS1-v1_5 [RFC8017] avec l'algorithme de hachage correspondant comme défini dans [SHS]. Ces valeurs se réfèrent seulement aux signatures qui apparaissent dans les certificats (voir le paragraphe 4.4.2.2) et ne sont pas définis pour être utilisés dans les messages de prise de contact TLS signés, bien qu'ils PUISSENT apparaître dans "signature_algorithms" et "signature_algorithms_cert" pour la rétro compatibilité avec TLS 1.2.

algorithmes ECDSA : indique un algorithme de signature qui utilise ECDSA, la courbe correspondante comme définie dans ANSI X9.62 [ECDSA] et FIPS 186-4 [DSS], et l'algorithme de hachage correspondant comme défini dans [SHS]. La signature est représentée comme une structure ECDSA-Sig-Value codée en DER [X690].

algorithmes RSASSA-PSS RSAE : indique un algorithme de signature utilisant RSASSA-PSS [RFC8017] avec la fonction 1 de génération de gabarit. Le résumé utilisé dans la fonction de génération de gabarit et le résumé à signer sont tous deux l'algorithme de hachage correspondant comme défini dans [SHS]. La longueur du sel DOIT être égale à la longueur du résultat de l'algorithme de résumé. Si la clé publique est portée dans un certificat X.509, il DOIT utiliser l'OID rsaEncryption [RFC5280].

algorithmes EdDSA : indique un algorithme de signature utilisant EdDSA comme défini dans la [RFC8032] ou ses successeurs. Noter que cela correspond aux algorithmes "PureEdDSA" et non aux variantes "prehash".

algorithmes RSASSA-PSS PSS : indique un algorithme de signature utilisant RSASSA-PSS [RFC8017] avec la fonction 1 de génération de gabarit. Le résumé utilisé dans la fonction de génération de gabarit et le résumé à signer sont tous deux l'algorithme de hachage correspondant comme défini dans [SHS]. La longueur du sel DOIT être égale à la longueur de l'algorithme de résumé. Si la clé publique est portée dans un certificat X.509, elle DOIT utiliser l'OID RSASSA-PSS [RFC5756]. Lorsque utilisé dans les signatures de certificat, les paramètres de l'algorithme DOIVENT être codés en DER. Si les paramètres correspondants de clé publique sont présents, les paramètres dans la signature DOIVENT alors être identiques à ceux de la clé publique.

algorithmes traditionnels : indique les algorithmes qui sont déconseillés parce qu'ils utilisent des algorithmes qui ont des faiblesses connues, spécifiquement SHA-1 qui est utilisé dans ce contexte avec (1) RSA utilisant RSASSA-PKCS1-v1_5, ou (2) ECDSA. Ces valeurs se réfèrent seulement aux signatures qui apparaissent dans les certificats (voir le paragraphe 4.4.2.2) et ne sont pas définies pour être utilisées dans les messages de prise de contact TLS, bien qu'elles PUISSENT apparaître dans "signature_algorithms" et "signature_algorithms_cert" pour la rétro compatibilité avec TLS 1.2. Les points d'extrémité NE DEVRAIENT PAS négocier ces algorithmes mais il leur est permis de le faire seulement pour les besoins de rétro compatibilité. Les clients qui offrent ces valeurs DOIVENT les mentionner comme plus basse priorité (mentionnés après tous les autres algorithmes dans SignatureSchemeList). Les serveurs TLS 1.3 NE DOIVENT PAS offrir un certificat SHA-1 signé sauf si aucune chaîne valide de certificats ne peut être produite sans lui (voir le paragraphe 4.4.2.2).

Les signatures sur les certificats qui sont auto signés ou les certificats qui sont des ancres de confiance ne sont pas validés, car ils commencent un chemin de certification (voir la [RFC5280], paragraphe 3.2). Un certificat qui commence un chemin de certification PEUT utiliser un algorithme de signature qui n'est pas annoncé comme étant pris en charge dans l'extension "signature_algorithms".

Noter que TLS 1.2 définit cette extension différemment. Les mises en œuvre de TLS 1.3 qui veulent négocier TLS 1.2 DOIVENT se comporter conformément aux exigences de la [RFC5246] pour négocier cette version. En particulier :

- Les ClientHello TLS 1.2 PEUVENT omettre cette extension.
- Dans TLS 1.2, l'extension contenait des paires de hachage/signature. Les paires sont codées sur deux octets, de sorte que les valeurs de SignatureScheme ont été allouées pour s'aligner sur le codage TLS 1.2. Certaines paires traditionnelles restent non allouées. Ces algorithmes sont déconseillés par TLS 1.3. Elles NE DOIVENT être offertes ou négociées par aucune mise en œuvre. En particulier, MD5 [SLOTH], SHA-224, et DSA NE DOIVENT PAS être utilisées.
- Les schémas de signature ECDSA s'alignent sur les paires de hachage/signature ECDSA de TLS 1.2. Cependant,

l'ancienne sémantique n'imposait pas de contrainte à la courbe de signature. Si TLS 1.2 est négociée, les mises en œuvre DOIVENT être prêtes à accepter une signature qui utilise toute courbe qu'elles ont annoncé dans l'extension "supported_groups".

- Les mises en œuvre qui annoncent qu'elles prennent en charge RSASSA-PSS (qui est obligatoire dans TLS 1.3) DOIVENT être prêtes à accepter une signature qui utilise ce schéma même quand TLS 1.2 est négocié. Dans TLS 1.2, RSASSA-PSS est utilisé avec les suites de chiffrement RSA.

4.2.4 Autorités de certificat

L'extension "certificate_authorities" est utilisée pour indiquer les autorités de certification (CA) qu'un point d'extrémité prend en charge et qui DEVRAIENT être utilisées par le point d'extrémité receveur pour guider le choix du certificat.

Le corps de l'extension "certificate_authorities" consiste en une structure de CertificateAuthoritiesExtension.

```
opaque DistinguishedName<1..2^16-1>;
 struct {
   DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;
```

authorities : liste des noms distinctifs [X501] d'autorités de certification acceptables, représentées en format codé en DER [X690]. Ces noms distinctifs spécifient un nom distinctif désiré pour une ancre de confiance ou une CA subordonné ; donc, ce message peut être utilisé pour décrire des ancres de confiance connues ainsi qu'un espace d'autorisation désiré.

Le client PEUT envoyer l'extension "certificate_authorities" dans le message ClientHello. Le serveur PEUT l'envoyer dans le message CertificateRequest.

L'extension "trusted_ca_keys" [RFC6066], qui sert un objet similaire mais plus compliqué, n'est pas utilisée dans TLS 1.3 (bien qu'elle puisse apparaître dans les messages ClientHello provenant de clients qui offrent des versions antérieures de TLS).

4.2.5 Filtres d'OID

L'extension "oid_filters" permet aux serveurs de fournir un ensemble de paires OID/valeur qu'ils aimeraient que respecte le certificat du client. Cette extension, si elle est fournie par le serveur, DOIT seulement être envoyée dans le message CertificateRequest.

```
struct {
   opaque certificate_extension_oid<1..2^8-1>;
   opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
   OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;
```

filters : liste d'OID d'extension de certificat [RFC5280] avec leurs valeurs permises et représentées en format codé en DER [X690]. Certains OID d'extension de certificat permettent plusieurs valeurs (par exemple, Extended Key Usage). Si le serveur a inclus une liste de filtres non vide, le certificat de client inclus dans la réponse DOIT contenir tous les OID d'extension spécifiés que le client reconnaît. Pour chaque OID d'extension reconnu par le client, toutes les valeurs spécifiées DOIVENT être présentes dans le certificat de client (mais le certificat PEUT aussi avoir d'autres valeurs). Cependant, le client DOIT ignorer et sauter tous les OID d'extension de certificat non reconnus. Si le client a ignoré certains des OID d'extension de certificat requis et fourni un certificat qui ne satisfait pas la demande, le serveur PEUT à sa discrétion soit continuer la connexion sans authentification du client, soit interrompre la prise de contact avec une alerte "certificat non accepté". Aucun OID NE DOIT PAS apparaître plus d'une fois dans la liste des filtres.

Les RFC sur PKIX définissent divers OID d'extension de certificat et leurs types de valeur correspondants. Selon le type, les valeurs d'extension de certificat correspondantes ne sont pas nécessairement égales au bit près. Il est prévu que les mises en œuvre de TLS s'appuient sur leurs propres bibliothèques de PKI pour effectuer le choix de certificat en utilisant les OID d'extension de certificat.

Le présent document définit les règles de correspondance pour deux extensions de certificat standard défini dans la [RFC5280] :

- L'extension KeyUsage dans un certificat correspond à la demande lorsque tous les bits d'usage de clé mentionnés dans la demande sont aussi mentionnés dans l'extension de certificat Key Usage.
- L'extension ExtendedKeyUsage dans un certificat correspond à la demande quand tous les OID en rapport avec les clés présents dans la demande se trouvent aussi dans l'extension de certificat ExtendedKeyUsage. L'OID spécial anyExtendedKeyUsage NE DOIT PAS être utilisé dans la demande.

Des spécifications particulières peuvent définir des règles de correspondance pour d'autres extensions de certificat.

4.2.6 Authentification de client après prise de contact

L'extension "post_handshake_auth" est utilisée pour indiquer qu'un client veut effectuer une authentification post prise de contact (paragraphe 4.6.2). Les serveurs NE DOIVENT PAS envoyer de CertificateRequest post prise de contact aux clients qui n'offrent pas cette extension. Les serveurs NE DOIVENT PAS envoyer cette extension.

```
struct {} PostHandshakeAuth;
```

Le champ "extension_data" de l'extension "post_handshake_auth" est de longueur zéro.

4.2.7 Groupes pris en charge

Lorsque envoyée par le client, l'extension "supported_groups" indique les groupes désignés que le client prend en charge pour l'échange de clé, ordonnés par préférence décroissante.

Note : Dans les versions de TLS antérieures à TLS 1.3, cette extension était nommée "courbes_elliptiques" et contenait seulement des groupes de courbes elliptiques (voir les [RFC8422] et [RFC7919]. Cette extension a aussi été utilisée pour négocier des courbes ECDSA. Les algorithmes de signature sont maintenant négociés indépendamment (voir au paragraphe 4.2.3).

Le champ "extension_data" de cette extension contient une valeur "NamedGroupList" :

```
enum {
/* Groupes de courbes elliptiques (ECDHE) */
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019), x25519(0x001D), x448(0x001E),

/* Groupes de champ fini (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102), ffdhe6144(0x0103), ffdhe8192(0x0104),

/* Codets réservés */
    ffdhe_private_use(0x01FC..0x01FF), ecdhe_private_use(0xFE00..0xFEFF), (0xFFFF)
 } NamedGroup;

 struct {
    NamedGroup named_group_list<2..2^16-1>;
 } NamedGroupList;
```

Groupes de courbes elliptiques (ECDHE) : Indique la prise en charge de la courbe désignée correspondante, définie dans FIPS 186-4 [DSS] ou la [RFC7748]. Les valeurs de 0xFE00 à 0xFEFF sont réservées pour utilisation privée [RFC8126].

Groupes de champ fini (DHE) : Indique la prise en charge du groupe de champ fini correspondant, défini dans la [RFC7919]. Les valeurs de 0x01FC à 0x01FF sont réservées pour utilisation privée.

Les éléments dans named_group_list sont ordonnés selon les préférences de l'envoyeur (le choix préféré en premier).

Dans TLS 1.3, il est permis aux serveurs d'envoyer l'extension "supported_groups" au client. Les clients NE DOIVENT PAS agir sur des informations trouvées dans "supported_groups" avant la réussite complète de la prise de contact mais PEUVENT utiliser les informations apprises d'une prise de contact terminée avec succès pour changer les groupes qu'ils utilisent dans leur extension "key_share" dans les connexions suivantes. Si le serveur a un groupe qu'il préfère à ceux de l'extension "key_share" mais veut quand même accepter le ClientHello, il DEVRAIT envoyer "supported_groups" pour mettre à jour la vue du client de ses préférences ; cette extension DEVRAIT contenir tous les groupes que le serveur prend

en charge, sans considérer si ils sont actuellement pris en charge par le client.

4.2.8 Partage de clés

L'extension "key_share" contient les paramètres de chiffrement du point d'extrémité.

Les clients PEUVENT envoyer un vecteur client_shares vide afin de demander un choix de groupes par le serveur, au prix d'un aller-retour supplémentaire (voir le paragraphe 4.1.4).

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

group : le groupe désigné pour la clé qui est changée.

key_exchange : informations d'échange de clé. Le contenu de ce champ est déterminé par le groupe spécifié et sa définition correspondante. Les paramètres Field Diffie-Hellman [DH76] finis sont décrits au paragraphe 4.2.8.1 ; les paramètres Diffie-Hellman de courbe elliptique sont décrits au paragraphe 4.2.8.2.

Dans le message ClientHello, le champ "extension_data" de cette extension contient une valeur "KeyShareClientHello" :

```
struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;
```

client_shares : une liste de valeurs KeyShareEntry offertes en ordre décroissant des préférences du client.

Ce vecteur PEUT être vide si le client fait une HelloRetryRequest. Chaque valeur de KeyShareEntry DOIT correspondre à un groupe offert dans l'extension "supported_groups" et DOIT apparaître dans le même ordre. Cependant, les valeurs PEUVENT être un sous ensemble non contigu de l'extension "supported_groups" et PEUVENT omettre les groupes préférés. Une telle situation pourrait se produire si les groupes préférés sont nouveaux et ont peu de chances d'être pris en charge dans suffisamment de lieux pour que soit efficace une pré génération de partage de clés pour eux.

Les clients peuvent offrir autant de valeurs de KeyShareEntry que le nombre de groupes pris en charge qu'ils offrent, chacun représentant un seul ensemble de paramètres d'échange de clé. Par exemple, un client pourrait offrir des parts pour plusieurs courbes elliptiques ou plusieurs groupes FFDHE. Les valeurs de key_exchange pour chaque KeyShareEntry DOIVENT être générées indépendamment. Les clients NE DOIVENT PAS offrir plusieurs valeurs de KeyShareEntry pour le même groupe. Les clients NE DOIVENT PAS offrir de valeur de KeyShareEntry pour des groupes qui ne figurent pas sur la liste de l'extension "supported_groups" du client. Les serveurs PEUVENT vérifier si il a des violations de ces règles et interrompre la prise de contact avec une alerte "paramètre_illégal" si une d'elles est violée.

Dans un message HelloRetryRequest, le champ "extension_data" de cette extension contient une valeur KeyShareHelloRetryRequest :

```
struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;
```

selected_group : groupe mutuellement pris en charge que le serveur entend négocier et pour lequel il demande un réessai de ClientHello/KeyShare.

À réception de cette extension dans une HelloRetryRequest, le client DOIT vérifier que (1) le champ selected_group correspond à un groupe qui était fourni dans l'extension "supported_groups" dans le ClientHello d'origine et (2) que le champ selected_group ne correspond pas à un groupe qui était fourni dans l'extension "key_share" dans le ClientHello d'origine. Si une de ces vérifications échoue, le client DOIT alors interrompre la prise de contact avec une alerte "paramètre_illégal". Autrement, lors de l'envoi d'un nouveau ClientHello, le client DOIT remplacer l'extension originale "key_share" par une qui contient seulement une nouvelle KeyShareEntry pour le groupe indiqué dans le champ selected_group de la HelloRetryRequest déclencheuse.

Dans un message ServerHello, le champ "extension_data" de cette extension contient une valeur KeyShareServerHello :

```
struct {
   KeyShareEntry server_share;
} KeyShareServerHello;
```

server_share : une seule valeur de KeyShareEntry qui est dans le même groupe qu'une des part du client.

Si on utilise l'établissement de clé (EC)DHE, les serveurs offrent exactement une KeyShareEntry dans le ServerHello. Cette valeur DOIT être dans le même groupe que la valeur de KeyShareEntry offerte par le client que le serveur a choisie pour l'échange de clé négocié. Les serveurs NE DOIVENT PAS envoyer de KeyShareEntry pour un groupe non indiqué dans l'extension "supported_groups" du client et NE DOIVENT PAS envoyer de KeyShareEntry en utilisant le PskKeyExchangeMode "psk_ke". Si on utilise l'établissement de clé (EC)DHE et si une HelloRetryRequest contenant une extension "key_share" a été reçue par le client, le client DOIT vérifier que le NamedGroup choisi dans le ServerHello est le même que celui de la HelloRetryRequest. Si cette vérification échoue, le client DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

4.2.8.1 Paramètres Diffie-Hellman

Les paramètres Diffie-Hellman [DH76] pour les clients et les serveurs sont codés dans le champ opaque key_exchange d'une KeyShareEntry dans une structure KeyShare. La valeur opaque contient la valeur publique Diffie-Hellman (Y = g^X mod p) pour le groupe spécifié (voir dans la [RFC7919] les définitions de groupe) codée comme un entier gros boutien et bourré à gauche avec des zéros jusqu'à la taille de p en octets.

Note : pour un groupe Diffie-Hellman donné, le bourrage résulte en ce que toutes les clés publiques ont la même longueur.

Les homologues DOIVENT valider la clé publique Y de l'autre en s'assurant que 1 < Y < p-1. Cette vérification assure que l'homologue distant se comporte bien et ne force pas le système local dans un petit sous groupe.

4.2.8.2 Paramètres ECDHE

Les paramètres ECDHE pour les clients et les serveurs sont codés dans le champ opaque key_exchange d'une KeyShareEntry dans une structure KeyShare.

Pour secp256r1, secp384r1, et secp521r1, le contenu est la valeur en série de la structure suivante :

```
struct {
   uint8 legacy_form = 4;
   opaque X[coordinate_length];
   opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

X et Y sont, respectivement, la représentation binaire des valeurs x et y dans l'ordre des octets du réseau. Il n'y a pas de marqueur de longueur interne, de sorte que chaque représentation de nombre occupe autant d'octets qu'impliqué par les paramètres de la courbe. Pour P-256, cela signifie que chacun de X et Y utilise 32 octets, bourrés à gauche par des zéros si nécessaire. Pour P-384, ils prennent 48 octets chacun. Pour P-521, ils prennent 66 octets chacun.

Pour les courbes secp256r1, secp384r1, et secp521r1, les homologues DOIVENT valider chacun la valeur publique Q de l'autre en s'assurant que le point est un point valide sur la courbe elliptique. Les procédures de validation appropriées sont définies au paragraphe 4.3.7 de [ECDSA] et aussi au paragraphe 5.6.2.3 de [KEYAGR]. Ce processus comporte trois étapes : (1) vérifier que Q n'est pas le point infini (O), (2) vérifier que pour Q = (x, y) les deux entiers x et y sont dans l'intervalle correct, et (3) s'assurer que (x, y) est une solution correcte le l'équation de la courbe elliptique. Pour ces courbes, les mises en œuvre n'ont pas besoin de vérifier l'appartenance au sous groupe correct.

Pour X25519 et X448, le contenu de la valeur publique est la chaîne binaire en entrée et en sortie les fonctions correspondantes définies dans la [RFC7748] : 32 octets pour X25519 et 56 octets pour X448.

Note : Les versions de TLS antérieures à 1.3 permettent des négociations de format de point ; TLS 1.3 supprime cette caractéristique en faveur d'un seul format de point pour chaque courbe.

4.2.9 Modes d'échange de clé pré partagées

Afin d'utiliser les PSK, les clients DOIVENT aussi envoyer une extension "psk_key_exchange_modes". La sémantique de cette extension est que le client prend seulement en charge l'utilisation des PSK avec ces modes, ce qui restreint à la fois

l'utilisation des PSK offertes dans ce ClientHello et celles que le serveur peut fournir via NewSessionTicket.

Un client DOIT fournir une extension "psk_key_exchange_modes" si il offre une extension "pre_shared_key". Si les clients offrent une "pre_shared_key" sans une extension "psk_key_exchange_modes", les serveurs DOIVENT interrompre la prise de contact. Les serveurs NE DOIVENT PAS choisir un mode d'échange de clé qui n'est pas sur la liste du client. Cette extension restreint aussi les modes à utiliser avec la reprise de PSK. Les serveurs NE DEVRAIENT PAS envoyer de NewSessionTicket avec des tickets qui ne sont pas compatibles avec les modes annoncés ; cependant, si un serveur fait ainsi, l'impact sera juste que la tentative de reprise du client va échouer.

Le serveur NE DOIT PAS envoyer une extension "psk_key_exchange_modes".

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
        PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

psk_ke : établissement de clé seulement PSK. Dans ce mode, le serveur NE DOIT PAS fournir une valeur "key_share".

psk_dhe_ke : PSK avec établissement de clé (EC)DHE. Dans ce mode, le client et le serveur DOIVENT fournir des valeurs de "key_share" comme décrit au paragraphe 4.2.8.

Toutes les futures valeurs qui seront allouées doivent s'assurer que les messages de protocole transmis identifient sans ambiguïté quel mode a été choisi par le serveur ; à présent, ceci est indiqué par la présence de la "key_share" dans le ServerHello.

4.2.10 Indication de données précoces

Lorsque une PSK est utilisée et que les données précoces sont permises pour cette PSK, le client peut envoyer des données d'application dans son premier envoi de messages. Si le client opte pour cette solution, il DOIT fournir les deux extensions "pre_shared_key" et "early_data".

Le champ "extension_data" de cette extension contient une valeur "EarlyDataIndication".

```
 struct {} Vide ;

 struct {
    select (Handshake.msg_type) {
      cas new_session_ticket :   uint32 max_early_data_size;
      cas client_hello :         vide ;
      cas encrypted_extensions : vide ;
    };
 } EarlyDataIndication;
```

Voir au paragraphe 4.6.1 les détails concernant l'utilisation du champ max_early_data_size.

Les paramètres pour les données 0-RTT (version, suite de chiffrement symétrique, négociation de protocole de couche application (ALPN, *Application-Layer Protocol Negotiation*) [RFC7301] protocole, etc.) sont ceux associés à la PSK utilisée. Pour les PSK provisionnées en externe, les valeurs associées sont celles provisionnées avec la clé. Pour les PSK établies via un message NewSessionTicket, les valeurs associées sont celles qui ont été négociées dans la connexion qui a établi la PSK. La PSK utilisée pour chiffrer les données précoces DOIT être la première PSK sur la liste dans l'extension "pre_shared_key" du client.

Pour les PSK provisionnées via NewSessionTicket, un serveur DOIT valider que l'âge du ticket pour l'identité de PSK choisie (calculée en soustrayant ticket_age_add de PskIdentity.obfuscated_ticket_age modulo 2^32) est dans une petite tolérance de l'heure de production du ticket (voir la Section 8). Si il ne l'est pas, le serveur DEVRAIT poursuivre la prise de contact mais rejeter 0-RTT, et NE DEVRAIT PAS prendre d'autre mesure qui suppose que ce ClientHello est frais.

Les messages 0-RTT envoyés dans le premier envoi ont les mêmes types de contenu (chiffré) que les messages de même type envoyés dans les autres envois (prise de contact et données d'application) mais sont protégés sous des clés différentes. Après avoir reçu le message Finished du serveur, si le serveur a accepté des données précoces, un message EndOfEarlyData sera envoyé pour indiquer le changement de clé. Ce message sera chiffré avec les clé de trafic 0-RTT.

Un serveur qui reçoit une extension "early_data" DOIT se comporter d'une de trois façons :

- Ignorer l'extension et retourner une réponse 1-RTT régulière. Le serveur saute alors les données précoces passées en tentant de déprotéger les enregistrements reçus en utilisant la clé de trafic de prise de contact, en éliminant les enregistrements qui échouent à la déprotection (jusqu'à la max_early_data_size configurée). Une fois qu'un enregistrement a réussi a être déprotégé, il est traité comme le début du second envoi du client et le serveur poursuit comme avec une prise de contact 1-RTT ordinaire.

- Demander que le client envoie un autre ClientHello en répondant avec une HelloRetryRequest. Un client NE DOIT PAS inclure l'extension "early_data" dans son ClientHello de suite. Le serveur ignore alors les données précoces en sautant tous les enregistrements avec un type de contenu externe de "application_data" (indiquant qu'elles sont chiffrées), jusqu'à la max_early_data_size configurée.

- Retourner sa propre extension "early_data" dans EncryptedExtensions, indiquant qu'il entend traiter les données précoces. Il n'est pas possible au serveur d'accepter seulement un sous ensemble des messages de données précoces. Même si le serveur envoie un message acceptant les données précoces, les données précoces réelles elles-mêmes peuvent être déjà envoyées au moment où le serveur génère ce message.

Afin d'accepter les données précoces, le serveur DOIT avoir accepté une suite de chiffrement PSK et choisi la première clé offerte dans l'extension "pre_shared_key" du client. De plus, il DOIT vérifier que les valeurs suivantes sont les mêmes que celles associées à la PSK choisie :
- le numéro de version TLS,
- la suite de chiffrement choisie,
- le protocole ALPN [RFC7301] choisi, si il en est un.

Ces exigences sont un sur ensemble de celles nécessaires pour effectuer une prise de contact 1-RTT en utilisant la PSK en question. Pour les PSK établies en externe, les valeurs associées sont celles provisionnées avec la clé. Pour les PSK établies via un message NewSessionTicket, les valeurs associées sont celles négociées dans la connexion durant laquelle le ticket a été établi.

Les futures extensions DOIVENT définir leur interaction avec 0-RTT.

Si une de ces vérifications échoue, le serveur NE DOIT PAS répondre avec l'extension et doit éliminer toutes les données du premier envoi en utilisant un des deux premiers mécanismes cités ci-dessus (revenant donc à 1-RTT ou 2-RTT). Si le client tente une prise de contact à 0-RTT mais si le serveur la rejette, le serveur n'aura généralement pas les clés de protection d'enregistrement 0-RTT et devra plutôt utiliser à la place le déchiffrement d'essai (soit avec les clés de prise de contact 1-RTT, soit en cherchant un ClientHello en clair dans le cas d'une HelloRetryRequest) pour trouver le premier message non 0-RTT.

Si le serveur choisit d'accepter l'extension "early_data", il DOIT alors se conformer aux mêmes exigences de traitement d'erreur que spécifiées pour tous les enregistrements lors du traitement d'enregistrements de données précoces. Précisément, si le serveur échoue à déchiffrer un enregistrement 0-RTT qui suit une extension "early_data" acceptée, il DOIT terminer la connexion avec une alerte "mauvais mac d'enregistrement" conformément au paragraphe 5.2.

Si le serveur rejette l'extension "early_data", l'application cliente PEUT opter pour retransmettre les données d'application envoyées précédemment en données précoces une fois que la prise de contact s'est terminée. Noter que la retransmission automatique des données précoces pourrait résulter en hypothèses incorrectes concernant l'état de la connexion. Par exemple, lorsque la connexion négociée choisit un protocole ALPN différent de ce qui a été utilisé pour les données précoces, une application peut avoir besoin de construire des messages différents. De même, si les données précoces supposent quelque chose sur l'état de la connexion, elle peut être envoyée par erreur après l'achèvement de la prise de contact.

Une mise en œuvre de TLS NE DEVRAIT PAS renvoyer automatiquement les données précoces ; les applications sont dans une meilleure position pour décider quand la retransmission est appropriée. Une mise en œuvre de TLS NE DOIT PAS renvoyer automatiquement les données précoces sauf si la connexion négociée choisit le même protocole ALPN.

4.2.11 Extension de clé pré partagée

L'extension "pre_shared_key" est utilisée pour négocier l'identité de la clé pré partagée à utiliser avec une certaine prise de contact en association avec établissement de clé PSK.

Le champ "extension_data" de cette extension contient une valeur "PreSharedKeyExtension" :

```
struct {
   opaque identity<1..2^16-1>;
   uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
   PskIdentity identities<7..2^16-1>;
   PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
   select (Handshake.msg_type) {
      cas client_hello: OfferedPsks;
      cas server_hello: uint16 selected_identity;
   };
} PreSharedKeyExtension;
```

identity : étiquette pour une clé. Par exemple, un ticket (comme défini dans l'Appendice B.3.4) ou une étiquette pour une clé pré partagée établie en externe.

obfuscated_ticket_age : une version obscurcie de l'âge de la clé. Le paragraphe 4.2.11.1 décrit comment former cette valeur pour les identités établies via le message NewSessionTicket. Pour les identités établies en externe, un obfuscated_ticket_age de 0 DEVRAIT être utilisé, et les serveurs DOIVENT ignorer la valeur.

Identities : une liste des identités que le client veut négocier avec le serveur. Si elle est envoyée avec l'extension "early_data" (voir le paragraphe 4.2.10) la première identité est celle utilisée pour les données 0-RTT.

Binders : série de valeurs HMAC, une pour chaque valeur dans la liste des identités et dans le même ordre, calculées comme décrit ci-dessous.

selected_identity : identité choisie par le serveur, exprimée comme un index (de base 0) dans les identités de la liste du client.

Chaque PSK est associée à un seul algorithme de hachage. Pour les PSK établies via le mécanisme de ticket (paragraphe 4.6.1) c'est l'algorithme de hachage KDF sur la connexion où le ticket a été établi. Pour les PSK établies en externe, l'algorithme de hachage DOIT être réglé lors de l'établissement de la PSK ou par défaut à SHA-256 si un tel algorithme n'est pas défini. Le serveur DOIT s'assurer qu'il choisit une PSK (si il en est) et une suite de chiffrement compatibles.

Dans les versions de TLS antérieures à TLS 1.3, la valeur d'identification de nom de serveur (SNI, *Server Name Identification*) était destinée à être associée à la session (Section 3 de la [RFC6066]), le serveur étant obligé de vérifier que la valeur de SNI associée à la session correspond à celle spécifiée dans la reprise de prise de contact. Cependant, dans la réalité, les mises en œuvre n'étaient pas cohérentes sur laquelle des deux valeurs de SNI fournies utiliser, ce qui conduit à l'exigence de cohérence appliquée de facto par les clients. Dans TLS 1.3, la valeur de SNI est toujours explicitement spécifiée dans la reprise de prise de contact, et il n'est pas besoin que le serveur associe une valeur de SNI au ticket. Les clients DEVRAIENT cependant mémoriser le SNI avec la PSK pour satisfaire aux exigences du paragraphe 4.6.1.

Note de mise en œuvre : lorsque la reprise de session est le principal cas d'utilisation des PSK, la façon la plus directe de mettre en œuvre la PSK/suite de chiffrement correspondant aux exigences est de négocier d'abord la suite de chiffrement et ensuite d'exclure toutes les PSK incompatibles. Toutes les PSK inconnues (par exemple, celles qui ne sont pas dans la base de données de PSK ou qui sont chiffrées avec une clé inconnue) DEVRAIENT simplement être ignorées. Si aucune PSK acceptable n'est trouvée, le serveur DEVRAIT effectuer une prise de contact sans PSK, si possible. Si la rétro compatibilité est importante, une PSK fournie par le client, établie en externe, DEVRAIT influencer le choix de la suite de chiffrement.

Avant d'accepter l'établissement de clé PSK, le serveur DOIT valider la valeur de liant *(binder)* correspondante (voir le paragraphe 4.2.11.2 ci-dessous). Si cette valeur n'est pas présente ou n'est pas validée, le serveur DOIT interrompre la prise de contact. Les serveurs NE DEVRAIENT PAS tenter de valider plusieurs liants ; ils DEVRAIENT plutôt choisir une seule PSK et valider seulement le liant qui correspond à cette PSK. Voir au paragraphe 8.2 et à l'Appendice E.6 les questions de

sécurité pour cette exigence. Afin d'accepter l'établissement de clé PSK, le serveur envoie une extension "pre_shared_key" qui indique l'identité choisie.

Les clients DOIVENT vérifier que l'identité choisie du serveur est dans la gamme fournie par le client, que le serveur a choisi une suite de chiffrement indiquant un hachage associé à la PSK, et qu'une extension "key_share" de serveur est présente si exigé par l'extension "psk_key_exchange_modes" du ClientHello. Si ces valeurs ne sont pas cohérentes, le client DOIT interrompre la prise de contact avec une alerte "paramètre_illégal". Si le serveur fournit une extension "early_data", le client DOIT vérifier que l'identité choisie par le serveur est 0. Si une autre valeur est retournée, le client DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

L'extension "pre_shared_key" DOIT être la dernière extension dans le ClientHello (cela facilite la mise en œuvre comme décrit ci-dessous). Les serveurs DOIVENT vérifier que c'est la dernière extension et autrement faire échouer la prise de contact avec une alerte "paramètre_illégal".

4.2.11.1 Âge de ticket

La vue du client de l'âge d'un ticket est le temps passé depuis la réception du message NewSessionTicket. Les clients NE DOIVENT PAS tenter d'utiliser des tickets qui ont des âges supérieurs à la valeur de "ticket_lifetime" qui était fournie avec le ticket. Le champ "obfuscated_ticket_age" de chaque PskIdentity contient une version obscurcie de l'âge du ticket formée en prenant l'âge en millisecondes et en ajoutant la valeur de "ticket_age_add" qui était incluse avec le ticket (voir le paragraphe 4.6.1) modulo 2^32. Cette addition empêche des observateurs passifs de corréler des connexions sauf si les tickets sont réutilisés. Noter que le champ "ticket_lifetime" dans le message NewSessionTicket est en secondes mais que le "obfuscated_ticket_age" est en millisecondes. Parce que les durées de vie de ticket se limitent à une semaine, 32 bits suffisent à représenter tout âge plausible, même en millisecondes.

4.2.11.2 Liant PSK

La valeur de liant de PSK forme un lien entre une PSK et la prise de contact en cours, ainsi qu'un lien entre la prise de contact dans laquelle la PSK a été générée (si c'est via un message NewSessionTicket) et la prise de contact en cours. Chaque entrée dans la liste des liants est calculée comme un HMAC sur un hachage de transcription (voir le paragraphe 4.4.1) contenant un ClientHello partiel jusqu'au champ inclus PreSharedKeyExtension.identities. C'est-à-dire qu'il inclut tout le ClientHello mais pas la liste des liants elle-même. Les champs de longueur pour le message (incluant la longueur globale, la longueur du bloc des extensions, et la longueur de l'extension "pre_shared_key") sont tous réglés comme si les liants des longueurs correctes étaient présents.

PskBinderEntry est calculée de la même façon que le message Finished (paragraphe 4.4.4) mais avec la BaseKey étant la binder_key déduite via le programme de clé provenant de la PSK correspondante qui est offerte (voir le paragraphe 7.1).

Si la prise de contact inclut une HelloRetryRequest, les ClientHello et HelloRetryRequest initiaux sont inclus dans la transcription avec le nouveau ClientHello. Par exemple, si le client envoie ClientHello1, son liant sera calculé sur :

```
Transcript-Hash(Truncate(ClientHello1))
```

où Truncate() supprime la liste des liants du ClientHello.

Si le serveur répond avec une HelloRetryRequest et si le client envoie alors ClientHello2, son liant sera calculé sur :

```
Transcript-Hash(ClientHello1, HelloRetryRequest, Truncate(ClientHello2))
```

Le ClientHello1/ClientHello2 complet est inclus dans tous les autres calculs de hachage de prise de contact. Noter que dans le premier envoi, Truncate(ClientHello1) est haché directement, mais dans le second envoi, ClientHello1 est haché et ensuite réinjecté comme message "message_hash", comme décrit au paragraphe 4.4.1.

4.2.11.3 Ordre de traitement

Il est permis aux clients "d'écouler" les données 0-RTT jusqu'à ce qu'ils reçoivent le Finished du serveur, envoyant seulement alors le message EndOfEarlyData, suivi par le reste de la prise de contact. Afin d'éviter une impasse, lorsque ils acceptent des "early_data", les serveurs DOIVENT traiter le ClientHello du client et ensuite envoyer immédiatement leur envoi de messages, plutôt que d'attendre le message EndOfEarlyData du client avant d'envoyer le ServerHello.

4.3 Paramètres du serveur

Les deux messages suivants du serveur, EncryptedExtensions et CertificateRequest, contiennent des informations du serveur qui déterminent le reste de la prise de contact. Ces messages sont chiffrés avec les clés déduites du server_handshake_traffic_secret.

4.3.1 Extensions chiffrées

Dans toutes les prises de contact, le serveur DOIT envoyer le message EncryptedExtensions immédiatement après le message ServerHello. C'est le premier message chiffré avec les clés déduites du server_handshake_traffic_secret.

Le message EncryptedExtensions contient les extensions qui peuvent être protégées, c'est-à-dire, toutes celles qui ne sont pas nécessaires pour établir le contexte cryptographique mais qui ne sont pas associées à des certificats individuels. Le client DOIT vérifier dans les EncryptedExtensions la présence de toute extension interdite et si il en trouve DOIT interrompre la prise de contact avec une alerte "paramètre_illégal".

Structure de ce message :

```
struct {
   Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

extensions : liste des extensions. Pour plus d'informations, voir le tableau du paragraphe 4.2.

4.3.2 Demande de certificat

Un serveur qui s'authentifie avec un certificat PEUT facultativement demander un certificat au client. Ce message, si il est envoyé, DOIT suivre EncryptedExtensions.

Structure de ce message :

```
struct {
   opaque certificate_request_context<0..2^8-1>;
   Extension extensions<2..2^16-1>;
} CertificateRequest;
```

certificate_request_context : chaîne opaque qui identifie la demande de certificat et qui recevra un écho dans le message Certificate du client. Le certificate_request_context DOIT être unique sur la portée de cette connexion (empêchant ainsi la répétition des messages CertificateVerify du client). Ce champ DEVRA être de longueur zéro sauf si il est utilisé pour les échanges d'authentification post prise de contact décrits au paragraphe 4.6.2. Lorsque il demande une authentification post prise de contact, le serveur DEVRAIT rendre le contexte imprévisible pour le client (par exemple, en le générant au hasard) afin d'empêcher un attaquant qui a un accès temporaire à la clé privée du client de pré calculer des messages CertificateVerify valides.

Extensions : ensemble d'extensions décrivant les paramètres du certificat demandé. L'extension "signature_algorithms" DOIT être spécifiée, et d'autres extensions peuvent facultativement être incluses si elles sont définies pour ce message. Les clients DOIVENT ignorer les extensions non reconnues.

Dans les versions antérieures de TLS, le message CertificateRequest portait une liste d'algorithmes de signature et d'autorités de certification que le serveur accepterait. Dans TLS 1.3, la première est exprimée par l'envoi de l'extension "signature_algorithms" et facultativement de "signature_algorithms_cert". La seconde est exprimée par l'envoi de l'extension "certificate_authorities" (voir le paragraphe 4.2.4).

Les serveurs qui s'authentifient avec une PSK NE DOIVENT PAS envoyer de message CertificateRequest dans la prise de contact principale, bien qu'ils PUISSENT l'envoyer dans une authentification post prise de contact (voir le paragraphe 4.6.2) pourvu que le client ait envoyé l'extension "post_handshake_auth" (voir le paragraphe 4.2.6).

4.4 Messages d'authentification

Comme exposé à la Section 2, TLS utilise généralement un ensemble commun de messages pour l'authentification, la confirmation de clé, et l'intégrité de la prise de contact : Certificate, CertificateVerify, et Finished. (Les liants de PSK effectuent aussi la confirmation de clé, d'une façon similaire.) Ces trois messages sont toujours envoyés comme les derniers

messages dans l'envoi de prise de contact. Les messages Certificate et CertificateVerify ne sont envoyés que dans certaines circonstances, comme défini ci-dessous. Le message Finished est toujours envoyé au titre du bloc Authentification. Ces messages sont chiffrés avec des clés déduites du [sender]_handshake_traffic_secret.

Les calculs pour les messages Authentification prennent tous uniformément les entrées suivantes :
- La clé de certificat et de signature à utiliser.
- Un contexte de prise de contact consistant en l'ensemble de messages à inclure dans le hachage de transcription.
- Une clé de base à utiliser pour calculer une clé de MAC.

Sur la base de ces entrées, les messages contiennent alors :

Certificate : le certificat à utiliser pour l'authentification, et tous les certificats de soutien dans la chaîne. Noter que l'authentification du client sur la base du certificat n'est pas disponible dans les flux de prise de contact avec PSK (incluant 0-RTT).

CertificateVerify : signature sur la valeur Transcript-Hash(Handshake Context, Certificate).

Finished : un MAC sur la valeur Transcript-Hash(Handshake Context, Certificate, CertificateVerify) utilisant une clé MAC déduite de la clé de base.

Le tableau suivant définit le contexte de prise de contact et la clé MAC de base pour chaque scénario :

Mode	**Contexte de prise de contact**	**Clé de base**
Serveur	ClientHello ... le dernier de EncryptedExtensions/ CertificateRequest	server_handshake_traffic_secret
Client	ClientHello ... le dernier de serveur Finished/EndOfEarlyData	client_handshake_traffic_secret
Post-Handshake	ClientHello ... client Finished + CertificateRequest	client_application_traffic_secret_N

4.4.1 Hachage transcrit

Beaucoup des calculs cryptographiques dans TLS utilisent un hachage de transcription. Cette valeur est calculée en hachant l'enchaînement de chaque message inclus dans la prise de contact, incluant l'en-tête de message de prise de contact portant les champs de type et de longueur de message de prise de contact, mais non inclus les en-têtes de couche d'enregistrement. C'est-à-dire,

Transcript-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)

En exception à cette règle générale, quand le serveur répond à un ClientHello par une HelloRetryRequest, la valeur du ClientHello1 est remplacée par un message synthétique spécial de prise de contact du type de prise de contact "message_hash" contenant Hash(ClientHello1). C'est-à-dire,

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
    Hash(message_hash ||              /* type de prise de contact */
       00 00 Hash.length  ||          /* longueur du message de prise de contact (octets) */
       Hash(ClientHello1) ||          /* hachage du ClientHello1 */
       HelloRetryRequest  || ... || Mn)
```

La raison de cette construction est de permettre au serveur de faire une HelloRetryRequest sans état en mémorisant juste le hachage du ClientHello1 dans le mouchard, plutôt que d'exiger qu'il exporte l'état de hachage intermédiaire entier (voir le paragraphe 4.2.2).

Pour être concret, le hachage de transcription est toujours pris à partir de la séquence suivante de messages de prise de contact, commençant par le premier ClientHello et incluant seulement les messages qui ont été envoyés : ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, CertificateRequest de serveur, Certificate de serveur, CertificateVerify de serveur, Finished de serveur, EndOfEarlyData, Certificate de client, CertificateVerify de client, Finished de client.

En général, les mises en œuvre peuvent appliquer la transcription en conservant une valeur du hachage de transcription courante sur la base du hachage négocié. Noter, cependant, que les authentifications post prise de contact suivantes ne s'incluent pas les unes les autres, juste les messages jusqu'à la fin de la prise de contact principale.

4.4.2 Certificate

Ce message porte la chaîne de certificats du point d'extrémité à l'homologue.

Le serveur DOIT envoyer un message Certificate chaque fois que la méthode d'échange de clé acceptée utilise des certificats pour l'authentification (cela inclut toutes les méthodes d'échange de clé définies dans ce document sauf PSK).

Le client DOIT envoyer un message Certificate si et seulement si le serveur a demandé l'authentification du client via un message CertificateRequest (paragraphe 4.3.2). Si le serveur demande l'authentification du client mais qu'aucun certificat convenable n'est disponible, le client DOIT envoyer un message Certificat ne contenant pas de certificat (c'est-à-dire, avec le champ "certificate_list" de longueur 0). Un message Finished DOIT être envoyé sans considérer si le message Certificate est vide.

Structure de ce message :

```
enum {
    X509(0),
   RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
      cas RawPublicKey:                                  /* d'après la RFC 7250 ASN.1_subjectPublicKeyInfo */
         opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
      cas X509:
         opaque cert_data<1..2^24-1>;
   };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

certificate_request_context : si ce message est en réponse à une CertificateRequest, c'est la valeur de certificate_request_context dans ce message. Autrement (dans le cas d'une authentification de serveur) ce champ DEVRA être de longueur zéro.

certificate_list : séquence (chaîne) de structures CertificateEntry, contenant chacune un seul certificat et ensemble d'extensions.

Extensions : ensemble de valeurs d'extension pour CertificateEntry. Le format "Extension" est défini au paragraphe 4.2. Les extensions valides pour les certificats de serveur incluent à présent l'extension État OCSP [RFC6066] et l'extension SignedCertificateTimestamp [RFC6962] ; de futures extensions pourront être définies aussi pour ce message. Les extensions dans le message Certificate du serveur DOIVENT correspondre à celles du message ClientHello. Les extensions dans le message Certificate du client DOIVENT correspondre aux extensions dans le message CertificateRequest du serveur. Si une extension s'applique à la chaîne entière, elle DEVRAIT être incluse dans la première CertificateEntry.

Si l'extension de type de certificat correspondante ("server_certificate_type" ou "client_certificate_type") n'a pas été négociée dans EncryptedExtensions, ou si le type de certificat X.509 a été négocié, chaque CertificateEntry contient un certificat X.509 codé en DER. Le certificat de l'envoyeur DOIT venir dans le premier CertificateEntry dans la liste. Chaque certificat suivant DEVRAIT directement certifier celui qui le précède immédiatement. Parce que la validation de certificat exige que des ancres de confiance soient distribuées de façon indépendante, un certificat qui spécifie une ancre de confiance PEUT être omis de la chaîne, pourvu que les homologues pris en charge soient connus pour posséder un des certificats omis.

Note : Avant TLS 1.3, l'ordre de "certificate_list" exigeait que chaque certificat certifie celui qui le précède immédiatement ; cependant, certaines mises en œuvre permettaient une certaine souplesse. Les serveurs envoyaient parfois à la fois un intermédiaire courant et un déconseillé pour des besoins de transition, et d'autres étaient simplement configurés de façon incorrecte, mais ces cas ne peuvent néanmoins être validés. Pour une compatibilité

maximale, toutes les mises en œuvre DEVRAIENT être prêtes à traiter des certificats potentiellement étrangers et des ordres arbitraires de toute version TLS, à l'exception du certificat d'entité d'extrémité qui DOIT être le premier.

Si le type de certificat RawPublicKey a été négocié, la certificate_list ne DOIT alors contenir pas plus de une CertificateEntry, qui contient une valeur de ASN1_subjectPublicKeyInfo comme défini à la Section 3 de la [RFC7250].

Le type de certificat OpenPGP [RFC6091] NE DOIT PAS être utilisé avec TLS 1.3.

La certificate_list du serveur DOIT toujours être non vide. Un client va envoyer une certificate_list vide si il n'a pas un certificat approprié à envoyer en réponse à la demande d'authentification du serveur .

4.4.2.1 État OCSP et extensions SCT

Les [RFC6066] et [RFC6961] fournissent des extensions pour négocier que le serveur envoie des réponses OCSP au client. Dans TLS 1.2 et en dessous, le serveur répond avec une extension vide pour indiquer la négociation de cette extension et les informations OCSP sont portées dans un message CertificateStatus. Dans TLS 1.3, les informations OCSP du serveur sont portées dans une extension dans la CertificateEntry contenant le certificat associé. Précisément, le corps de l'extension "status_request" provenant du serveur DOIT être une structure CertificateStatus comme défini dans la [RFC6066], qui est interprétée comme défini dans la [RFC6960].

Note : l'extension status_request_v2 [RFC6961] est déconseillée. Les serveurs TLS 1.3 NE DOIVENT PAS agir sur sa présence ou ses informations quand ils traitent les messages ClientHello ; en particulier, ils NE DOIVENT PAS envoyer d'extension status_request_v2 dans les messages EncryptedExtensions, CertificateRequest, ou Certificate. Les serveurs TLS 1.3 DOIVENT être capables de traiter les messages ClientHello qui l'incluent, car ils PEUVENT être envoyés par des clients qui souhaitent l'utiliser dans des versions de protocole antérieures.

Un serveur PEUT demander qu'un client présente une réponse OCSP avec son certificat en envoyant une extension "status_request" vide dans son message CertificateRequest. Si le client opte pour envoyer une réponse OCSP, le corps de son extension "status_request" DOIT être une structure CertificateStatus comme défini dans la [RFC6066].

De façon similaire, la [RFC6962] fournit un mécanisme pour qu'un serveur envoie un horodatage de certificat signé (SCT, *Signed Certificate Timestamp*) comme une extension dans le ServerHello dans TLS 1.2 et en dessous. Dans TLS 1.3, les informations de SCT du serveur sont portées dans une extension dans la CertificateEntry.

4.4.2.2 Choix du certificat de serveur

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

- Le type de certificat DOIT être X.509v3 [RFC5280], sauf négociation contraire explicite (par exemple, [RFC7250]).
- La clé publique d'entité d'extrémité du serveur (et les restrictions associées) DOIT être compatible avec l'algorithme d'authentification choisi à partir de l'extension "signature_algorithms" du client (actuellement RSA, ECDSA, ou EdDSA).
- Le certificat DOIT permettre que la clé soit utilisée pour signer (c'est-à-dire, le bit digitalSignature DOIT être établi si l'extension KeyUsage est présente) avec un schéma de signature indiqué dans les extensions "signature_algorithms"/"signature_algorithms_cert" du client (voir le paragraphe 4.2.3).
- Les extensions "server_name" [RFC6066] et "certificate_authorities" sont utilisées pour guider le choix du certificat. Comme les serveurs PEUVENT exiger la présence de l'extension "server_name", les clients DEVRAIENT envoyer cette extension, lorsque applicable.

Tous les certificats fournis par le serveur DOIVENT être signés par un algorithme de signature annoncé par le client si il est capable de fournir une telle chaîne (voir le paragraphe 4.2.3). Les certificats qui sont auto signés ou les certificats qui sont supposés être des ancres de confiance ne sont pas validés au titre de la chaîne et donc PEUVENT être signés avec tout algorithme.

Si le serveur ne peut pas produire une chaîne de certificats qui est signée seulement via les algorithmes dont la prise en charge est indiquée, il DEVRAIT alors continuer la prise de contact en envoyant au client une chaîne de certificats de son choix qui peut inclure des algorithmes qui ne sont pas connus pour être pris en charge par le client. Cette chaîne de repli NE DEVRAIT PAS utiliser l'algorithme déconseillé SHA-1 en général, mais PEUT le faire si l'annonce du client le permet, et NE DOIT PAS le faire autrement.

Si le client ne peut pas construire une chaîne acceptable en utilisant les certificats fournis et décide d'interrompre la prise de contact, il DOIT alors interrompre la prise de contact avec une alerte appropriée en rapport avec le certificat (par défaut, "certificat non accepté" ; voir le paragraphe 6.2 pour plus d'informations).

Si le serveur a plusieurs certificats, il choisit l'un d'eux sur la base des critères susmentionnés (en plus d'autres critères, comme un point d'extrémité de couche transport, une configuration locale, et des préférences).

4.4.2.3 Choix du certificat du client

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

- Le type de certificat DOIT être X.509v3 [RFC5280], sauf négociation contraire explicite (par exemple, [RFC7250]).
- Si l'extension "certificate_authorities" était présente dans le message CertificateRequest, au moins un des certificats dans la chaîne de certificats DEVRAIT être produit par une des CA de la liste.
- Les certificats DOIVENT être signés en utilisant un algorithme de signature acceptable, comme décrit au paragraphe 4.3.2. Noter que ceci assouplit les contraintes sur les algorithmes de signature de certificat qu'on trouvait dans les précédentes versions de TLS.
- Si le message CertificateRequest contenait une extension "oid_filters" non vide, le certificat d'entité d'extrémité DOIT correspondre aux OID d'extension qui sont reconnus par le client, comme décrit au paragraphe 4.2.5.

4.4.2.4 Réception d'un message de certificat

En général, les procédures détaillées de validation de certificat sortent du domaine d'application de TLS (voir la [RFC5280]). Ce paragraphe donne les exigences spécifiques de TLS.

Si le serveur fournit un message Certificate vide, le client DOIT interrompre la prise de contact avec une alerte "erreur de décodage".

Si le client n'envoie aucun certificat (c'est-à-dire, si il envoie un message Certificate vide) le serveur PEUT à sa discrétion soit continuer la prise de contact sans authentification du client, soit interrompre la prise de contact avec une alerte "certificat exigé". Aussi, si un aspect de la chaîne de certificats est inacceptable (par exemple, il n'est pas signé par une CA de confiance connue) le serveur PEUT à sa discrétion soit continuer la prise de contact (en considérant que le client n'est pas authentifié) soit interrompre la prise de contact.

Tout point d'extrémité qui reçoit un certificat dont il aurait besoin pour valider en utilisant tout algorithme de signature utilisant un hachage MD5 DOIT interrompre la prise de contact avec une alerte "mauvais certificat". SHA-1 est déconseillé, et il est RECOMMANDÉ que tout point d'extrémité recevant un certificat qu'il aurait besoin de valider en utilisant tout algorithme de signature utilisant un hachage SHA-1 interrompe la prise de contact avec une alerte "mauvais certificat". Pour être clair, cela signifie que les points d'extrémité peuvent accepter ces algorithmes pour les certificats qui sont auto signés ou sont des ancres de confiance.

Il est RECOMMENDÉ que tous les points d'extrémité passent à SHA-256 ou mieux aussitôt que possible pour conserver l'interopérabilité avec les mises en œuvre actuellement dans le processus de remplacement de la prise en charge de SHA-1.

Noter qu'un certificat qui contient 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 ECDSA).

4.4.3 CertificateVerify

Ce message est utilisé pour fournir une preuve explicite qu'un point d'extrémité possède la clé privée correspondant à son certificat. Le message CertificateVerify assure aussi l'intégrité pour la prise de contact jusqu'à ce point. Les serveurs DOIVENT envoyer ce message lors de l'authentification via un certificat. Les clients DOIVENT envoyer ce message chaque fois qu'ils s'authentifient via un certificat (c'est-à-dire, lorsque le message Certificat n'est pas vide). Lorsque il est envoyé, ce message DOIT apparaître immédiatement après le message Certificate et immédiatement avant le message Finished.

Structure de ce message :

```
struct {
   SignatureScheme algorithm;
   opaque signature<0..2^16-1>;
} CertificateVerify;
```

Le champ “algorithm” spécifie l'algorithme de signature utilisé (voir au paragraphe 4.2.3 la définition de ce type). La signature est une signature numérique utilisant cet algorithme. Le contenu couvert par la signature est le résultat du hachage comme décrit au paragraphe 4.4.1, à savoir : Transcript-Hash(Handshake Context, Certificate)

La signature numérique est alors calculée sur l'enchaînement de :
- une chaîne qui consiste en l'octet 32 (0x20) répété 64 fois,
- la chaîne de contexte,
- un seul octet 0 qui sert de séparateur,
- le contenu à signer.

Cette structure est destinée à empêcher une attaque sur des versions antérieures de TLS dans lesquelles le format ServerKeyExchange signifiait que des attaquants pouvaient obtenir une signature d'un message avec un préfixe choisi de 32 octets (ClientHello.random). Le bourrage initial de 64 octets supprime ce préfixe avec le ServerHello.random contrôlé par le serveur.

La chaîne de contexte pour une signature de serveur est "TLS 1.3, server CertificateVerify". La chaîne de contexte pour une signature de client est "TLS 1.3, client CertificateVerify". Elle est utilisée pour fournir une séparation entre les signatures faites dans différents contextes, comme aide contre de potentielles attaques inter protocoles.

Par exemple, si le hachage de transcription était 32 octets de 01 (cette longueur aurait un sens pour SHA-256) le contenu couvert par la signature numérique pour un serveur CertificateVerify serait :

```
20202020202020202020202020202020202020202020202020202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c207365727665722043657274696669636174655665726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

Du côté envoyeur, le procès de calcul du champ de signature du message CertificateVerify prend en entrée :
- le contenu couvert par la signature numérique,
- la clé de signature privée correspondant au certificat envoyé dans le message précédent.

Si le message CertificateVerify est envoyé par un serveur, l'algorithme de signature DOIT être un de ceux offerts dans l'extension "signature_algorithms" du client sauf si aucune chaîne de certificat valide ne peut être produite sans des algorithmes non pris en charge (voir le paragraphe 4.2.3).

Si il est envoyé par un client, l'algorithme de signature utilisé dans la signature DOIT être un de ceux présents dans le champ supported_signature_algorithms de l'extension "signature_algorithms" dans le message CertificateRequest.

De plus, l'algorithme de signature DOIT être compatible avec la clé dans le certificat d'entité d'extrémité de l'envoyeur. Les signatures RSA DOIVENT utiliser un algorithme RSASSA-PSS, sans considérer si des algorithmes RSASSA-PKCS1-v1_5 apparaissent dans "signature_algorithms". L'algorithme SHA-1 NE DOIT PAS être utilisé dans les signatures de messages CertificateVerify.

Tous les algorithmes de signature SHA-1 dans la présente spécification sont définis pour le seul usage dans les certificats traditionnels et ne sont pas valides pour les signatures CertificateVerify.

Le receveur d'un message CertificateVerify DOIT vérifier le champ Signature. Le procès de vérification prend en entrée :
- le contenu couvert par la signature numérique,
- la clé publique contenue dans le certificat d'entité d'extrémité trouvé dans le message Certificate associé,
- la signature numérique reçue dans le champ Signature du message CertificateVerify.

Si la vérification échoue, le receveur DOIT terminer la prise de contact avec une alerte "erreur de déchiffrement".

4.4.4 Finished

Le message Finished est le message final dans le bloc d'authentification. Il est essentiel pour assurer l'authentification de la prise de contact et des clés calculées.

Les receveurs de messages Finished DOIVENT vérifier que le contenu est correct et si il est incorrect DOIVENT terminer la connexion avec une alerte "erreur de déchiffrement".

Une fois qu'un côté a envoyé son message Finished et a reçu et validé le message Finished de son homologue, il peut commencer à envoyer et recevoir des données d'application sur la connexion. Il y a deux réglages dans lesquels il est permis d'envoyer des données avant de recevoir le Finished de l'homologue :

1. Les clients envoyant des données 0-RTT comme décrit au paragraphe 4.2.10.
2. Les serveurs PEUVENT envoyer des données après l'émission de leur premier envoi, mais parce que la prise de contact n'est pas encore achevée, ils n'ont pas l'assurance de l'identité de l'homologue ni de sa vivacité (c'est-à-dire, le ClientHello pourrait avoir été répété).

La clé utilisée pour calculer le message Finished est calculée à partir de la clé de base définie au paragraphe 4.4 en utilisant HKDF (voir le paragraphe 7.1). Précisément :

```
finished_key = HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Structure de ce message :

```
struct {
   opaque verify_data[Hash.length];
} Finished;
```

La valeur verify_data est calculée comme suit :

```
verify_data =
    HMAC(finished_key,
       Transcript-Hash(Handshake Context,
                  Certificate*, CertificateVerify*))
```

* Seulement inclus si présent.

HMAC [RFC2104] utilise l'algorithme de hachage pour la prise de contact. Comme noté ci-dessus, l'entrée HMAC peut généralement être mise en œuvre par un hachage courant, c'est-à-dire, juste le hachage de prise de contact à ce point.

Dans les versions antérieures de TLS, les verify_data font toujours 12 octets. Dans TLS 1.3, elles font la taille du résultat HMAC pour le hachage utilisé pour la prise de contact.

Note : les alertes et tous les autres types d'enregistrements non liés à la prise de contact ne sont pas des messages de prise de contact et ne sont pas inclus dans les calculs de hachage.

Tous les enregistrements qui suivent un message Finished DOIVENT être chiffrés avec la clé de trafic appropriée, comme décrit au paragraphe 7.2. En particulier, cela inclut toutes les alertes envoyées par le serveur en réponse aux messages Certificate et CertificateVerify du client.

4.5 Fin des données précoces

```
struct {} EndOfEarlyData;
```

Si le serveur a envoyé une extension "early_data" dans EncryptedExtensions, le client DOIT envoyer un message EndOfEarlyData après avoir reçu le Finished du serveur. Si le serveur n'a pas envoyé une extension "early_data" dans EncryptedExtensions, alors le client NE DOIT PAS envoyer de message EndOfEarlyData. Ce message indique que tous les messages application_data 0-RTT, si il en est, ont été transmis et que les enregistrements suivants sont protégés avec les clés de trafic de la prise de contact. Les serveurs NE DOIVENT PAS envoyer ce message, et les clients qui le reçoivent DOIVENT terminer la connexion avec une alerte "message inattendu". Ce message est chiffré avec les clés déduites du client_early_traffic_secret.

4.6 Messages après prise de contact

TLS permet aussi que d'autres messages soient envoyés après la prise de contact principale. Ces messages utilisent un type de contenu de prise de contact et sont chiffrés avec la clé de trafic approprié.

4.6.1 Message de ticket de nouvelle session

À tout moment après que le serveur a reçu le message Finished du client, il PEUT envoyer un message NewSessionTicket. Ce message crée une association unique entre la valeur du ticket et une PSK secrète déduite du secret maître de reprise (voir la Section 7).

Le client PEUT utiliser cette PSK pour de futures prises de contact en incluant la valeur du ticket dans l'extension "pre_shared_key" dans son ClientHello (paragraphe 4.2.11). Les serveurs PEUVENT envoyer plusieurs tickets sur une seule connexion, soit immédiatement après chaque autre ou après des événements spécifiques (voir l'Appendice C.4). Par exemple, le serveur peut envoyer un nouveau ticket après l'authentification post prise de contact afin d'encapsuler l'état supplémentaire d'authentification du client. Plusieurs tickets sont utiles aux clients pour divers objets, incluant :

- l'ouverture de plusieurs connexions HTTP en parallèle,
- effectuer une compétition de connexion entre les interfaces et familles d'adresses via (par exemple) Happy Eyeballs [RFC8305] ou des techniques qui s'y rapportent.

Un ticket DOIT seulement être repris avec une suite de chiffrement qui a le même algorithme de hachage de KDF que celui qui a été utilisé pour établir la connexion d'origine.

Les clients DOIVENT seulement reprendre si la nouvelle valeur de SNI est valide pour le certificat de serveur présenté dans la session d'origine et DEVRAIT seulement reprendre si la valeur de SNI correspond à celle utilisée dans la session d'origine. Le premier cas est une optimisation de performances : normalement, il n'y a pas de raison d'attendre de différents serveurs couverts par un seul certificat qu'ils soient capables d'accepter les tickets de l'autre ; donc, tenter la reprise dans ce cas gaspillerait un ticket à usage unique. Si une telle indication est fournie (en externe ou par tout autre moyen) les clients PEUVENT reprendre avec une valeur de SNI différente.

À la reprise, si elles rapportent une valeur de SNI à l'application appelante, les mises en œuvre DOIVENT utiliser la valeur envoyée dans le ClientHello de reprise plutôt que la valeur envoyée dans la session précédente. Noter que si une mise en œuvre de serveur décline toutes les identités de PSK avec différentes valeurs de SNI, ces deux valeurs sont toujours les mêmes.

Note : Bien que le secret maître de reprise dépende du second envoi du client, un serveur qui ne demande pas l'authentification du client PEUT calculer le reste de la transcription de façon indépendante et ensuite envoyer un NewSessionTicket immédiatement à l'envoi du Finished plutôt que d'attendre le Finished du client. Cela peut être approprié dans les cas où le client est supposé ouvrir plusieurs connexions TLS en parallèle et bénéficierait d'une réduction des frais généraux d'une reprise de prise de contact, par exemple.

```
struct {
   uint32 ticket_lifetime;
   uint32 ticket_age_add;
   opaque ticket_nonce<0..255>;
   opaque ticket<1..2^16-1>;
   Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

ticket_lifetime : indique la durée de vie en secondes comme un entier non signé de 32 bits dans l'ordre des octets du réseau à partir de l'heure de production du ticket. Les serveurs NE DOIVENT PAS utiliser de valeur supérieure à 604 800 secondes (7 jours). La valeur zéro indique que le ticket devrait être éliminée immédiatement. Les clients NE DOIVENT PAS mettre en antémémoire des tickets pour plus de 7 jours, sans considération de la durée de vie du ticket, et PEUVENT supprimer les tickets avant, sur la base de la politique locale. Un serveur PEUT traiter un ticket comme valide pour une période plus courte que ce qui était déclaré dans la ticket_lifetime.

ticket_age_add : valeur générée de façon sûre et aléatoire de 32 bits qui est utilisée pour obscurcir l'âge du ticket que le client inclut dans l'extension "pre_shared_key". L'âge du ticket côté client est ajouté à cette valeur modulo 2^32 pour obtenir la valeur qui est transmise par le client. Le serveur DOIT générer une valeur fraîche pour chaque ticket qu'il envoie.

ticket_nonce : valeur par ticket qui est unique pour tous les tickets produits sur cette connexion.

ticket : valeur du ticket à utiliser comme identité de PSK. Le ticket lui-même est une étiquette opaque. Il PEUT être soit une clé de recherche dans une base de données, soit une valeur auto chiffrée et auto authentifiée.

extensions : ensemble de valeurs d'extension pour le ticket. Le format "Extension" est défini au paragraphe 4.2. Les clients DOIVENT ignorer les extensions non reconnues.

La seule extension actuellement définie pour NewSessionTicket est "early_data", indiquant que le ticket peut être utilisé pour envoyer des données 0-RTT (paragraphe 4.2.10). Il contient la valeur suivante :

max_early_data_size : quantité maximum de données 0-RTT qu'il est permis au client d'envoyer quand il utilise ce ticket, en octets. Seule la charge utile de données d'application (c'est-à-dire, le texte en clair mais sans bourrage ou l'octet

de type de contenu interne) est compté. Un serveur qui reçoit plus que max_early_data_size octets de données de 0-RTT DEVRAIT terminer la connexion avec une alerte "message inattendu". Noter que les serveurs qui rejettent les données précoces du fait d'un manque de matériel cryptographique seront incapables de différencier le bourrage du contenu, de sorte que les clients NE DEVRAIENT PAS dépendre de la capacité des enregistrements de données précoces d'envoyer de grandes quantités de bourrage.

La PSK associée au ticket est calculée comme :

HKDF-Expand-Label(resumption_master_secret, "resumption", ticket_nonce, Hash.length)

Parce que la valeur ticket_nonce est distincte pour chaque message NewSessionTicket, une PSK différente sera déduite pour chaque ticket.

Noter qu'en principe, il est possible de continuer de produire de nouveaux tickets qui étendent indéfiniment la durée de vie du matériel de chiffrement déduit à l'origine d'une prise de contact initiale non PSK (qui était très probablement liée au certificat de l'homologue). Il est RECOMMANDÉ que les mises en œuvre placent des limites à la durée de vie totale d'un tel matériel de chiffrement ; ces limites devraient tenir compte de la durée de vie du certificat de l'homologue, de la probabilité d'une révocation intermédiaire, et du temps écoulé depuis la signature du CertificateVerify en ligne de l'homologue.

4.6.2 Authentification après prise de contact

Lorsque le client a envoyé l'extension "post_handshake_auth" (voir le paragraphe 4.2.6) un serveur PEUT demander l'authentification du client à tout moment après l'achèvement de la prise de contact par l'envoi d'un message CertificateRequest. Le client DOIT répondre avec les messages d'authentification appropriés (voir le paragraphe 4.4). Si le client choisit de s'authentifier, il DOIT envoyer Certificate, CertificateVerify, et Finished. Si il refuse, il DOIT envoyer un message Certificate ne contenant pas de certificat suivi par Finished. Tous les messages du client pour une réponse donnée DOIVENT apparaître consécutivement sur le réseau sans message intermédiaire d'autres types.

Un client qui reçoit un message CertificateRequest sans avoir envoyé l'extension "post_handshake_auth" DOIT envoyer une alerte fatale "message inattendu".

Note : Parce que l'authentification du client pourrait impliquer d'inviter l'utilisateur, les serveurs DOIVENT être prêts à un certain délai, incluant de recevoir un nombre arbitraire d'autres messages entre l'envoi de la CertificateRequest et la réception d'une réponse. De plus, les clients qui reçoivent de multiples CertificateRequest qui se suivent de près PEUVENT leur répondre dans un ordre différent de celui de leur réception (la valeur de certificate_request_context permet au serveur de distinguer les réponses).

4.6.3 Mise à jour de clé et de valeur d'initialisation

Le message de prise de contact KeyUpdate est utilisé pour indiquer que l'envoyeur est en train de mettre à jour ses clés de chiffrement d'envoi. Ce message peut être envoyé par l'un ou l'autre homologue après l'envoi d'un message Finished. Les mises en œuvre qui reçoivent un message KeyUpdate avant de recevoir un message Finished DOIVENT terminer la connexion avec une alerte "message inattendu". Après l'envoi d'un message KeyUpdate, l'envoyeur DEVRA envoyer tout son trafic avec la prochaine génération de clés, calculée comme décrit au paragraphe 7.2. À réception d'une KeyUpdate, le receveur DOIT mettre à jour ses clés de réception.

```
enum {
   update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
   KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update : indique si le receveur de la KeyUpdate devrait répondre avec sa propre KeyUpdate. Si une mise en œuvre reçoit toute autre valeur, elle DOIT terminer la connexion avec une alerte "paramètre_illégal".

Si le champ request_update est réglé à "update_requested", le receveur DOIT alors envoyer de lui-même une KeyUpdate avec request_update réglé à "update_not_requested" avant d'envoyer son prochain enregistrement de données d'application. Ce mécanisme permet aux deux côtés de forcer à une mise à jour de la connexion entière, mais conduit une mise en œuvre qui reçoit plusieurs KeyUpdate alors qu'elle est silencieuse à répondre avec une seule mise à jour. Noter que les mises en

œuvre peuvent recevoir un nombre arbitraire de messages entre l'envoi d'une KeyUpdate avec request_update réglé à "update_requested" et la réception du KeyUpdate de l'homologue, parce que ces messages peuvent être déjà en chemin. Cependant, parce que les clés d'envoi et de réception sont déduites de secrets de trafic indépendants, conserver le secret du trafic de réception ne menace pas le secret des données transmises avant que l'envoyeur ait changé les clés.

Si les mises en œuvre ont envoyé indépendamment leurs propres KeyUpdate avec request_update réglé à "update_requested" et qu'elles se croisent en chemin, chaque côté va aussi envoyer une réponse, avec comme résultat que chaque côté augmente de deux générations.

L'envoyeur et le receveur DOIVENT tous deux chiffrer leurs messages KeyUpdate avec les vieilles clés. De plus, les deux côtés DOIVENT appliquer qu'un KeyUpdate avec la vieille clé est reçu avant d'accepter de message chiffré avec la nouvelle clé. Manquer à faire comme cela peut permettre des attaques en troncature de message.

5. Protocole d'enregistrement

Le protocole d'enregistrement TLS prend les messages à transmettre, fragmente les données en blocs gérables, protège les enregistrements, et transmet le résultat. Les données reçues sont vérifiées, déchiffrées, réassemblées, et ensuite livrées aux clients de niveau supérieur.

Les enregistrements TLS sont typés, ce qui permet que plusieurs protocoles de niveau supérieur soient multiplexés sur la même couche d'enregistrement. Le présent document spécifie quatre types de contenu : prise de contact, données d'application, alerte, et changement de la spécification de chiffrement. L'enregistrement change_cipher_spec est seulement utilisé pour des besoins de compatibilité (voir l'Appendice D.4).

Une mise en œuvre peut recevoir un enregistrement non chiffré de type change_cipher_spec consistant en la valeur d'un seul octet 0x01 à tout moment après que le premier message ClientHello a été envoyé ou reçu et avant que le message Finished de l'homologue ait été reçu et DOIT simplement l'éliminer sans autre traitement. Noter que cet enregistrement peut apparaître à la prise de contact lorsque la mise en œuvre attend des enregistrements protégés, et donc il est nécessaire de détecter cette condition avant de tenter de déprotéger l'enregistrement. Une mise en œuvre qui reçoit toute autre valeur de change_cipher_spec ou qui reçoit un enregistrement change_cipher_spec protégé DOIT interrompre la prise de contact avec une alerte "message inattendu". Si une mise en œuvre détecte un enregistrement change_cipher_spec reçu avant le premier message ClientHello ou après le message Finished de l'homologue, il DOIT être traité comme un type d'enregistrement inattendu (bien que les serveurs sans état puissent n'être pas capables de distinguer ces cas de ceux qui sont permis).

Les mises en œuvre NE DOIVENT PAS envoyer de types d'enregistrements non définis dans le présent document sauf négociés par une extension. Si une mise en œuvre de TLS reçoit un type d'enregistrement inattendu, elle DOIT terminer la connexion avec une alerte "message inattendu". De nouvelles valeurs de type de contenu d'enregistrement sont allouées par l'IANA dans le registre de type de contenu TLS comme décrit à la Section 11.

5.1 Couche d'enregistrement

La couche d'enregistrement fragmente les blocs d'informations en enregistrements TLSPlaintext portant des données en tronçons de 2^14 octets ou moins. Les limites de message sont traitées différemment selon le ContentType sous-jacent. Tout futur type de contenu DEVRA spécifier les règles appropriées. Noter que ces règles sont plus strictes que ce qui était appliqué dans TLS 1.2.

Les messages Handshake PEUVENT être regroupés en un seul enregistrement TLSPlaintext ou fragmentés sur plusieurs enregistrements, pourvu que :

- Les messages NE DOIVENT PAS être entrelacés avec d'autre types d'enregistrement. C'est-à-dire, si un message de prise de contact est partagé entre deux enregistrements ou plus, il NE DOIT PAS y avoir d'autre enregistrement entre eux.
- Les messages de prise de contact NE DOIVENT PAS s'étendre au delà d'un changement de clés. Les mises en œuvre DOIVENT vérifier que tous les messages précédant immédiatement un changement de clé s'alignent avec une limite d'enregistrement ; sinon, elles DOIVENT alors terminer la connexion avec une alerte "message inattendu". Parce que les messages ClientHello, EndOfEarlyData, ServerHello, Finished, et KeyUpdate peuvent précéder immédiatement un changement de clé, les mises en œuvre DOIVENT envoyer ces messages alignés sur une limite d'enregistrement.

Les mises en œuvre NE DOIVENT PAS envoyer de fragments de longueur zéro de type Handshake, même si ces fragments contiennent un bourrage.

Les messages d'alerte (Section 6) NE DOIVENT PAS être fragmentés entre des enregistrements, et plusieurs messages d'alerte NE DOIVENT PAS être regroupés en un seul enregistrement TLSPlaintext. En d'autres termes, un enregistrement avec un type Alert DOIT contenir exactement un message.

Les messages de données d'application contiennent des données qui sont opaques pour TLS. Les messages de données d'application sont toujours protégés. Des fragments de longueur zéro de données d'application PEUVENT être envoyés, car ils sont potentiellement utiles comme contre mesures d'analyse de trafic. Des fragments de données d'application PEUVENT être étalés sur plusieurs enregistrements ou regroupés en un seul enregistrement.

```
enum {
   invalid(0),
   change_cipher_spec(20),
   alert(21),
   handshake(22),
   application_data(23),
  (255)
} ContentType;

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

type : protocole de niveau supérieur utilisé pour traiter le fragment inclus.

legacy_record_version : DOIT être réglé à 0x0303 pour tous les enregistrements générés par une mise en œuvre TLS 1.3 autre qu'un ClientHello initial (c'est-à-dire, un qui n'est pas généré après une HelloRetryRequest) où il PEUT aussi être 0x0301 pour des besoins de compatibilité. Ce champ est déconseillé et DOIT être ignoré pour tous les objets. Les précédentes versions de TLS auraient utilisé d'autres valeurs dans ce champ dans certaines circonstances.

length : longueur (en octets) du TLSPlaintext.fragment suivant. La longueur NE DOIT PAS excéder 2^14 octets. Un point d'extrémité qui reçoit un enregistrement qui excède cette longueur DOIT terminer la connexion avec une alerte "débordement d'enregistrement".

fragment : données à transmettre. Cette valeur est transparente et est traitée comme un bloc indépendant à traiter avec un protocole de couche supérieure spécifié par le champ de type.

Le présent document décrit TLS 1.3, qui utilise la version 0x0304. Cette valeur de version est historique, dérivée de l'utilisation de 0x0301 pour TLS 1.0 et 0x0300 pour SSL 3.0. Afin de maximiser la rétro compatibilité, un enregistrement contenant un ClientHello initial DEVRAIT avoir une version 0x0301 (reflétant TLS 1.0) et un enregistrement contenant un second ClientHello DEVRAIT avoir une version 0x0301 (reflétant TLS 1.0) et un enregistrement contenant un second ClientHello ou un ServerHello DOIT avoir la version 0x0303 (reflétant TLS 1.2). Lors de la négociation de versions antérieures de TLS, les points d'extrémité suivent les procédures et exigences données dans l'Appendice D.

Lorsque la protection d'enregistrement n'a pas encore été engagée, les structures TLSPlaintext sont écrites directement sur le réseau. Une fois que la protection d'enregistrement a commencé, les enregistrements TLSPlaintext sont protégés et envoyés comme décrit au paragraphe suivant. Noter que les enregistrements de données d'application NE DOIVENT PAS être écrits sur le réseau sans protection (voir les détails à la Section 2).

5.2 Protection de charge utile d'enregistrement

Les fonctions de protection d'enregistrement traduisent une structure TLSPlaintext en une structure TLSCiphertext. Les fonctions de déprotection inversent le processus. Dans TLS 1.3, à la différence des versions antérieures de TLS, tous les chiffrements sont modélisés comme "chiffrement authentifié avec des données associées" (AEAD, *Authenticated Encryption with Associated Data*) [RFC5116]. Les fonctions AEAD fournissent une opération unifiée de chiffrement et d'authentification qui transforme le texte source en texte chiffré et authentifié et inversement. Chaque enregistrement chiffré consiste en un en-tête de texte source suivi d'un corps chiffré, qui lui-même contient un type et un bourrage facultatif.

```
struct {
   opaque content[TLSPlaintext.length];
   ContentType type;
   uint8 zeros[longueur_de_bourrage];
} TLSInnerPlaintext;

struct {
   ContentType opaque_type = application_data; /* 23 */
   ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
   uint16 length;
   opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

content : valeur de TLSPlaintext.fragment, contenant le codage des octets d'un message de prise de contact ou d'alerte, ou les octets bruts des données d'application à envoyer.

Type : valeur de TLSPlaintext.type contenant le type de contenu de l'enregistrement.

Zeros : portée de longueur arbitraire d'octets de valeur zéro qui peut apparaître dans le texte en clair après le champ Type. Cela donne aux envoyeurs l'opportunité de bourrer tout enregistrement TLS d'une quantité choisie pour autant que le total reste dans les limites de taille d'enregistrement. Voir les détails au paragraphe 5.4.

opaque_type : le champ externe opaque_type d'un enregistrement TLSCiphertext est toujours réglé à la valeur 23 (application_data) pour la compatibilité avec les boîtiers de médiation qui sont accoutumés à analyser des versions antérieures de TLS. Le type de contenu réel de l'enregistrement se trouve dans TLSInnerPlaintext.type après déchiffrement.

legacy_record_version : le champ legacy_record_version est toujours 0x0303. Les TLSCiphertext TLS 1.3 ne sont pas générés tant que TLS 1.3 n'a pas été négocié, de sorte qu'il n'y a pas de problème de compatibilité historique si d'autres valeurs devaient être reçues. Noter que le protocole de prise de contact, incluant les messages ClientHello et ServerHello, authentifie la version de protocole, de sorte que cette valeur est redondante.

Length : longueur (en octets) du TLSCiphertext.encrypted_record suivant, qui est la somme des longueurs du contenu et du bourrage, plus un pour le type de contenu interne, plus toute expansion ajoutée par l'algorithme AEAD. La longueur NE DOIT PAS excéder 2^14 + 256 octets. Un point d'extrémité qui reçoit un enregistrement qui excède cette longueur DOIT terminer la connexion avec une alerte "débordement d'enregistrement".

encrypted_record : forme chiffrée en AEAD de la structure TLSInnerPlaintext mise en série.

Les algorithmes AEAD prennent en entrée une seule clé, un nom occasionnel *(nonce)*, un texte source *(plaintext)*, et des "données supplémentaires" à inclure dans la vérification d'authentification, comme décrit au paragraphe 2.1 de la [RFC5116]. La clé est soit la client_write_key, soit la serveur_write_key, le nom occasionnel est déduit du numéro de séquence et du client_write_iv ou server_write_iv (voir le paragraphe 5.3), et l'entrée de données supplémentaires est l'en-tête d'enregistrement. C'est-à-dire,

additional_data = TLSCiphertext.opaque_type || TLSCiphertext.legacy_record_version || TLSCiphertext.length

L'entrée de texte source dans l'algorithme AEAD est la structure TLSInnerPlaintext codée. La déduction des clés de trafic est définie au paragraphe 7.3.

Le résultat de AEAD consiste en le résultat de texte chiffré de l'opération de chiffrement AEAD. La longueur de texte source est supérieure à la TLSPlaintext.length correspondante du fait de l'inclusion de TLSInnerPlaintext.type et de tout bourrage fourni par l'envoyeur. La longueur du résultat de AEAD va généralement être supérieure à celle du texte source, mais d'une quantité qui varie avec l'algorithme AEAD.

Comme les chiffrements peuvent incorporer un bourrage, la quantité de surdébit peut varier avec les différentes longueurs de texte source. De façon symbolique,

AEADEncrypted = AEAD-Encrypt(write_key, nonce, additional_data, plaintext)

Le champ encrypted_record de TLSCiphertext est réglé à AEADEncrypted.

Afin de déchiffrer et vérifier, le chiffre prend en entrée la clé, le nom occasionnel, les données supplémentaires, et la valeur

de AEADEncrypted. Le résultat est soit le texte source, soit une erreur indiquant que le déchiffrement a échoué. Il n'y a pas de vérification d'intégrité séparée. De façon symbolique,

texte source de encrypted_record = AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)

Si le déchiffrement échoue, le receveur DOIT terminer la connexion avec une alerte "mauvais mac d'enregistrement".

Un algorithme AEAD utilisé dans TLS 1.3 NE DOIT PAS produire une expansion supérieure à 255 octets. Un point d'extrémité qui reçoit un enregistrement de son homologue avec une TLSCiphertext.length supérieure à 2^14 + 256 octets DOIT terminer la connexion avec une alerte "débordement d'enregistrement". Cette limite est déduite de la longueur maximum de TLSInnerPlaintext de 2^14 octets + 1 octet pour ContentType + l'expansion maximum de AEAD de 255 octets.

5.3 Nom occasionnel par enregistrement

Un numéro de séquence de 64 bits est conservé séparément pour les enregistrement de lecture et d'écriture. Le numéro de séquence approprié est incrémenté de un après la lecture ou l'écriture de chaque enregistrement. Chaque numéro de séquence est réglé à zéro au début d'une connexion et chaque fois que la clé est changée ; le premier enregistrement transmis sous une clé de trafic particulière DOIT utiliser le numéro de séquence 0.

Parce que la taille du numéro de séquence est de 64 bits, ils ne devraient pas revenir à zéro. Si une mise en œuvre de TLS devait avoir besoin de faire revenir à zéro un numéro de séquence, elle DOIT soit changer de clé (paragraphe 4.6.3) soit terminer la connexion.

Chaque algorithme AEAD va spécifier une gamme de longueurs possibles pour le nom occasionnel par enregistrement, de N_MIN octets à N_MAX octets d'entrée [RFC5116]. La longueur du nom occasionnel par enregistrement TLS (iv_length) est réglée au plus grand de 8 octets et N_MIN pour l'algorithme AEAD (voir la Section 4 de la [RFC5116]). Un algorithme AEAD où N_MAX fait moins de 8 octets NE DOIT PAS être utilisé avec TLS. Le nom occasionnel par enregistrement pour la construction AEAD est formé comme suit :

1. Le numéro de séquence d'enregistrement de 64 bits est codé dans l'ordre des octets du réseau et bourré à gauche avec des zéros jusqu'à iv_length.
2. Le numéro de séquence avec son bourrage est OUixé avec soit le client_write_iv, soit le server_write_iv (selon le rôle) statique.

La quantité résultante (la longueur iv_length) est utilisée comme nom occasionnel par enregistrement.

Note : C'est une construction différente de celle de TLS 1.2, qui spécifiait un nom occasionnel partiellement explicite.

5.4 Bourrage d'enregistrement

Tous les enregistrements TLS chiffrés peuvent être bourrés pour gonfler la taille du TLSCiphertext. Cela permet à l'envoyeur de cacher la taille du trafic à un observateur.

Lorsque elles génèrent un enregistrement TLSCiphertext, les mises en œuvre PEUVENT choisir de le bourrer. Un enregistrement non bourré est juste un enregistrement avec une longueur de bourrage de zéro. Le bourrage est une chaîne d'octets de valeur zéro ajoutée au champ ContentType avant le chiffrement. Les mises en œuvre DOIVENT régler les octets de bourrage tout à zéro avant le chiffrement.

Les enregistrements de données d'application peuvent contenir un TLSInnerPlaintext.content de longueur zéro si l'envoyeur le désire. Cela permet la génération de trafic couvert de taille plausible dans des contextes où la présence ou l'absence d'activité peut être sensible. Les mises en œuvre NE DOIVENT PAS envoyer d'enregistrements Handshake et Alert qui aient un TLSInnerPlaintext.content de longueur zéro ; si un tel message est reçu, la mise en œuvre receveuse DOIT terminer la connexion avec une alerte "message inattendu".

Le bourrage envoyé est automatiquement vérifié par le mécanisme de protection d'enregistrement ; lorsque un TLSCiphertext.encrypted_record est déchiffré avec succès, la mise en œuvre receveuse examine le champ de la fin au début jusqu'à ce qu'elle trouve un octet non à zéro. Cet octet non à zéro est le type de contenu du message. Ce schéma de 0 bourrage a été choisi parce qu'il permet le bourrage de tout enregistrement TLS bourré d'une taille arbitraire (de zéro jusqu'à la limite de taille d'enregistrement TLS) sans introduire de nouveaux types de contenu. Cette conception applique aussi le bourrage d'octets tout à zéro, ce qui permet la détection rapide des erreurs de bourrage.

Les mises en œuvre DOIVENT limiter leur examen au texte en clair retourné du déchiffrement AEAD. Si une mise en

œuvre receveuse ne trouve pas un octet non zéro dans le texte en clair, elle DOIT terminer la connexion avec une alerte "message inattendu".

La présence de bourrage ne change pas les limitations globales de taille d'enregistrement : le TLSInnerPlaintext complet codé NE DOIT PAS excéder 2^14 + 1 octets. Si la longueur maximum de fragment est réduite – comme, par exemple, par l'extension record_size_limit de la [RFC8449] – la limite réduite s'applique alors à tout le texte source, incluant le type de contenu et le bourrage.

Choisir une politique de bourrage qui suggère quand et de combien bourrer est un sujet complexe qui sort du domaine d'application de la présente spécification. Si la couche protocole d'application par dessus TLS a son propre bourrage, il peut être préférable de bourrer les enregistrements TLS de données d'application au sein de la couche Application. Le bourrage pour les enregistrements chiffrés Handshake ou Alert doit cependant toujours être traité à la couche TLS. Des documents ultérieurs pourraient définir des algorithmes de choix de bourrage ou définir un mécanisme de politique de bourrage au moyen d'extensions à TLS ou par d'autres moyens.

5.5 Limites à l'usage de clé

Il y a des limites cryptographiques à la quantité de texte source qui peut être chiffré en toute sécurité sous un ensemble de clés donné. [AEAD-LIMITS] donne une analyse de ces limites sous l'hypothèse que la primitive sous-jacente (AES ou ChaCha20) n'a pas de faiblesses. Les mises en œuvre DEVRAIENT faire une mise à jour de clé comme décrit au paragraphe 4.6.3 avant d'atteindre ces limites.

Pour AES-GCM, jusqu'à 2^24,5 enregistrements de pleine taille (environ 24 millions) peuvent être chiffrés sur une connexion tout en conservant une marge de sécurité d'approximativement 2^-57 pour la sécurité de chiffrement authentifié (AE, *Authenticated Encryption*). Pour ChaCha20/Poly1305, le numéro de séquence d'enregistrement va revenir à zéro avant que la limite de sécurité soit atteinte.

6. Protocole d'alerte

TLS fournit un type de contenu Alert pour indiquer des informations de clôture et des erreurs. Comme les autres messages, les messages d'alerte sont chiffrés comme spécifié par l'état de connexion actuel.

Les messages d'alerte portent une description de l'alerte et un champ hérité qui portait le niveau de sévérité du message dans les versions antérieures de TLS. Les alertes sont divisées en deux classes : alertes de clôture et alertes d'erreur. Dans TLS 1.3, la sévérité est implicite dans le type d'alerte envoyée, et le champ "niveau" peut être ignoré en toute sécurité. L'alerte "close_notify" est utilisée pour indiquer la clôture ordonnée d'une direction de la connexion. À réception d'une telle alerte, la mise en œuvre TLS DEVRAIT indiquer la fin des données à l'application.

Les alertes d'erreur indiquent une clôture interruptive de la connexion (voir le paragraphe 6.2). À réception d'une alerte d'erreur, la mise en œuvre TLS DEVRAIT indiquer une erreur à l'application et NE DOIT PAS permettre que d'autres données soient envoyées ou reçues sur la connexion. Les serveurs et clients DOIVENT oublier les valeurs et clés secrètes établies dans les connexions échouées, à l'exception des PSK associées à des tickets de session, qui DEVRAIENT être éliminés si possible.

Toutes les alertes citées au paragraphe 6.2 DOIVENT être envoyées avec AlertLevel=fatal et DOIVENT être traitées comme des alertes d'erreur lorsque reçues sans considération du AlertLevel dans le message. Les types d'alerte inconnus DOIVENT être traités comme des alertes d'erreur.

Note : TLS définit deux alertes génériques à utiliser sur un échec d'analyse d'un message. Les homologues qui reçoivent un message qui ne peut pas être analysé conformément à la syntaxe (par exemple, avoir une longueur qui s'étend au delà des limites du message ou contient une longueur hors gamme) DOIVENT terminer la connexion avec une alerte "erreur de décodage".

Les homologues qui reçoivent un message qui est syntaxiquement correct mais sémantiquement invalide (par exemple, une part DHE de p - 1, ou une énumération invalide) DOIVENT terminer la connexion avec une alerte "paramètre_illégal".

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

enum {
   close_notify(0),                    (notification de clôture)
```

```
   unexpected_message(10),              (message inattendu)
   bad_record_mac(20),                  (mauvais MAC d'enregistrement)
   record_overflow(22),                 (enregistrement débordant)
   handshake_failure(40),               (échec de prise de contact)
   bad_certificate(42),                 (mauvais certificat)
   unsupported_certificate(43),         (certificat non accepté)
   certificate_revoked(44),             (certificat révoqué)
   certificate_expired(45),             (certificat expiré)
   certificate_unknown(46),             (certificat inconnu)
   illegal_parameter(47),               (paramètre illégal)
   unknown_ca(48),                      (CA inconnue)
   access_denied(49),                   (accès refusé)
   decode_error(50),                    (erreur de décodage)
   decrypt_error(51),                   (erreur de déchiffrement)
   protocol_version(70),                (version de protocole)
   insufficient_security(71),           (sécurité insuffisante)
   internal_error(80),                  (erreur interne)
   inappropriate_fallback(86),          (repli inapproprié)
   user_canceled(90),                   (utilisateur annulé)
   missing_extension(109),              (extension manquante)
   unsupported_extension(110),          (extension non acceptée)
   unrecognized_name(112),              (nom non reconnu)
   bad_certificate_status_response(113), (mauvaise réponse d'état de certificat)
   unknown_psk_identity(115),           (identité PSK inconnue)
   certificate_required(116),           (certificat exigé)
   no_application_protocol(120),        (pas de protocole d'application)
   (255)
} AlertDescription;

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

6.1 Alertes de clôture

Le client et le serveur doivent partager la connaissance que la connexion est en train de fermer afin d'éviter l'attaque de troncature.

close_notify : cette alerte notifie au receveur que l'envoyeur ne va plus envoyer de message sur cette connexion. Toutes les données reçues après une alerte de clôture DOIVENT être ignorées à réception.

user_canceled : cette alerte notifie au receveur que l'envoyeur annule la prise de contact pour des raisons sans relation avec un échec de protocole. Si un usager annule une opération après l'achèvement de la prise de contact, il est plus approprié de juste clore la connexion en envoyant un "close_notify". Cette alerte DEVRAIT être suivie d'un "close_notify". Cette alerte a généralement AlertLevel=warning *(niveau d'alerte = avertissement)*.

L'une ou l'autre partie PEUT initier une clôture de son côté écriture de la connexion en envoyant une alerte "close_notify". Toutes les données reçues après la réception d'une alerte de clôture DOIVENT être ignorées. Si une clôture de niveau transport est reçue avant un "close_notify", le receveur ne peut pas savoir si toutes les données envoyées ont été reçues.

Chaque partie DOIT envoyer une alerte "close_notify" avant de clore son côté écriture de la connexion, sauf si elle a déjà envoyé une alerte d'erreur. Ceci n'a pas d'effet sur son côté lecture de la connexion. Noter que ceci est un changement par rapport aux versions de TLS antérieures à TLS 1.3 dans lesquelles il était exigé des mises en œuvre de réagir à un "close_notify" par l'élimination des écritures en cours et en l'envoi immédiat d'une alerte "close_notify". Cette exigence pourrait causer la troncature sur le côté lecture. Les deux parties n'ont pas besoin d'attendre de recevoir une alerte "close_notify" avant de clore leur propre côté de la connexion, bien que le faire introduise la possibilité de troncature.

Si le protocole d'application qui utilise TLS exige que toutes les données puissent être portées sur le transport sous-jacent après la fermeture de la connexion TLS, la mise en œuvre TLS DOIT recevoir une alerte "close_notify" avant d'indiquer la fin des données à la couche application. Aucune partie de la présente norme ne devrait être considérée comme dictant la manière dont un profil d'utilisation pour TLS gère son transport de données, incluant quand les connexions sont ouvertes ou

fermées.

Note : on suppose que la fermeture du côté écriture d'une connexion livre de façon fiable les données en cours avant de détruire le transport.

6.2 Alertes d'erreur

Le traitement d'erreur dans TLS est très simple. Lorsque une erreur est détectée, la partie qui détecte envoie un message à son homologue. Sur transmission ou réception d'un message d'alerte fatale, les deux parties DOIVENT immédiatement clore la connexion.

Chaque fois qu'une mise en œuvre rencontre une condition d'erreur fatale, elle DEVRAIT envoyer une alerte fatale appropriée et DOIT clore la connexion sans envoyer ou recevoir de données supplémentaires. Dans le reste de cette spécification, quand les phrases "terminer la connexion" et "interrompre la prise de contact" sont utilisées sans une alerte spécifique, cela signifie que la mise en œuvre DEVRAIT envoyer l'alerte indiquée par les descriptions ci-dessous. Les phrases "termine la connexion avec une alerte X" et "interrompre la prise de contact avec une alerte X" signifient que la mise en œuvre DOIT envoyer l'alerte X si elle envoie une alerte. Toutes les alertes définies ci-dessous dans cette section, ainsi que toutes les alertes inconnues, sont universellement considérées comme fatales en ce qui concerne TLS 1.3. La mise en œuvre DEVRAIT fournir un moyen de faciliter l'enregistrement de l'envoi et la réception des alertes.

Les alertes d'erreur suivantes sont définies :

unexpected_message : un message inapproprié (par exemple, un mauvais message de prise de contact, des données d'application prématurées, etc.) a été reçu. Cette alerte ne devrait jamais être observée dans des communications entre des mises en œuvre appropriées.

bad_record_mac : cette alerte est retournée si un enregistrement reçu ne peut être déprotégé. Parce que les algorithmes AEAD combinent le déchiffrement et la vérification, et aussi pour éviter des attaques sur canal latéral, cette alerte est utilisée pour tous les échecs de déprotection. Cette alerte ne devrait jamais être observée dans des communications entre des mises en œuvre appropriées, sauf quand les messages ont été corrompus dans le réseau.

record_overflow : un enregistrement TLSCiphertext a été reçu avec une longueur de plus de 2^14 + 256 octets, ou un enregistrement déchiffré en enregistrement TLSPlaintext avec plus de 2^14 octets (ou quelque autre limite négociée). Cette alerte ne devrait jamais être observée dans une communication entre des mises en œuvre appropriées, sauf quand les messages ont été corrompus dans le réseau.

handshake_failure : réception d'un message d'alerte "échec de prise de contact" qui indique que l'envoyeur a été incapable de négocier un ensemble acceptable de paramètres de sécurité étant données les options disponibles.

bad_certificate : un certificat était corrompu, contenait des signatures qui ne se sont pas vérifiées correctement, etc.

unsupported_certificate : un certificat était d'un type non pris en charge.

certificate_revoked : un certificat était révoqué par son signataire.

certificate_expired : un certificat a expiré ou n'est actuellement pas valide.

certificate_unknown : certains autres problèmes (non spécifiés) sont apparus dans le traitement du certificat, le rendant inacceptable.

illegal_parameter : un champ dans la prise de contact était incorrect ou incohérent avec les autres champs. Cette alerte est utilisée pour les erreurs qui se conforment à la syntaxe formelle du protocole mais sont par ailleurs incorrectes.

unknown_ca : une chaîne de certificat valide ou une chaîne partielle a été reçue, mais le certificat n'a pas été accepté parce que le certificat de CA n'a pas pu être localisé ou n'a pas pu être mis en correspondance avec une ancre de confiance connue.

access_denied : un certificat ou une PSK valide a été reçu, mais lorsque le contrôle d'accès a été appliqué, l'envoyeur a décidé de ne pas procéder à la négociation.

decode_error : un message n'a pas pu être décodé parce que un champ était hors de la gamme spécifiée ou que la longueur du message était incorrecte. Cette alerte est utilisée pour des erreurs où le message ne se conforme pas à la syntaxe

formelle du protocole. Cette alerte ne devrait jamais être observée dans une communication entre des mises en œuvre appropriées, sauf lorsque les messages ont été corrompus dans le réseau.

decrypt_error : une opération cryptographique de prise de contact (pas de couche d'enregistrement) incluant d'être incapable de vérifier correctement une signature ou valider un message Finished ou un liant PSK.

protocol_version : la version de protocole que l'homologue a tenté de négocier est reconnue mais pas prise en charge (voir l'Appendice D).

insufficient_security : retourné à la place de "échec de prise de contact" lorsque une négociation a échoué, spécifiquement parce que le serveur exige des paramètres plus sûrs que ceux acceptés par le client.

internal_error : une erreur interne sans relation avec l'homologue ou la correction du protocole (comme un échec d'allocation de mémoire) rend impossible de continuer.

inappropriate_fallback : envoyé par un serveur en réponse à une tentative invalide de réessai de connexion de la part d'un client (voir la [RFC7507]).

missing_extension : envoyé par les points d'extrémité qui reçoivent un message de prise de contact qui ne contient pas une extension dont l'envoi est obligatoire pour la version de TLS offerte ou autres paramètres négociés.

unsupported_extension : envoyé par les points d'extrémité qui reçoivent un message de prise de contact qui contient une extension dont il est connu que son inclusion est interdite dans un certain message de prise de contact, ou qui inclut des extensions dans un ServerHello ou un certificat non offertes d'abord dans le ClientHello ou la CertificateRequest correspondant.

unrecognized_name : envoyé par les serveurs quand il n'existe pas de serveur identifié par le nom fourni par le client via l'extension "server_name" (voir la [RFC6066]).

bad_certificate_status_response : envoyé par les clients quand une réponse OSSP invalide ou inacceptable est fournie par le serveur via l'extension "status_request" (voir la [RFC6066]).

unknown_psk_identity : envoyé par les serveurs quand l'établissement de clé PSK est désiré mais qu'une identité de PSK non acceptable est fournie par le client. L'envoi de cette alerte est FACULTATIF ; les serveurs PEUVENT à la place choisir d'envoyer une alerte "decrypt_error" pour simplement indiquer une identité de PSK invalide.

certificate_required : envoyé par les serveurs quand un certificat de client est désiré mais qu'aucun n'a été fourni par le client.

no_application_protocol : envoyé par les serveurs quand une extension de client "application_layer_protocol_negotiation" annonce seulement les protocoles que le serveur ne prend pas en charge (voir la [RFC7301]).

De nouvelles valeurs d'alerte seront allouées par l'IANA comme décrit à la Section 11.

7. Calculs cryptographiques

La prise de contact TLS établit un ou plusieurs secrets d'entrée qui sont combinés pour créer le matériel de chiffrement réel, comme détaillé ci-dessous. Le processus de déduction de clé incorpore à la fois les secrets d'entrée et la transcription de prise de contact. Noter que parce que la transcription de prise de contact inclut des valeurs aléatoires provenant des messages Hello, chaque prise de contact va avoir des secrets de trafic différents, même si les mêmes secrets d'entrée sont utilisés, comme c'est le cas lorsque la même PSK est utilisée pour plusieurs connexions.

7.1 Programmation de clé

Le processus de déduction de clé utilise les fonctions HKDF-Extract et HKDF-Expand comme défini pour HKDF [RFC5869], ainsi que les fonctions définies ci-dessous :

HKDF-Expand-Label(Secret, Étiquette, Contexte, Longueur) = HKDF-Expand(Secret, ÉtiquetteHkdf, Longueur)

où ÉtiquetteHkdf est spécifié comme :

```
struct {
   uint16 longueur = Longeur;
   étiquette opaque <7..255> = "tls13 " + Étiquette;
   contexte opaque <0..255> = Contexte;
} ÉtiquetteHkdf;
```

Derive-Secret(Secret, Étiquette, Messages) = HKDF-Expand-Label(Secret, Étiquette, Transcript-Hash(Messages), Hash.longueur)

La fonction de hachage utilisée par Transcript-Hash et HKDF est l'algorithme de hachage de suite de chiffrement. Hash.longueur est sa longueur de résultat en octets. Messages est l'enchaînement des messages de prise de contact indiqués, incluant les champs de type et longueur de message de prise de contact, mais non inclus les en-têtes de couche enregistrement. Noter que dans certains cas, un contexte de longueur zéro (indiqué par "") est passé à HKDF-Expand-Label. Les étiquettes spécifiées dans ce document sont toutes des chaînes ASCII et n'incluent pas d'octet NUL en queue.

Note : avec les fonctions courantes de hachage, toute étiquette plus longue que 12 caractères exige une itération supplémentaire de la fonction de hachage pour le calcul. Les étiquettes dans cette spécification doivent toutes être choisies pour tenir dans cette limite.

Les clés sont déduites de deux secrets d'entrée utilisant les fonctions HKDF-Extract et Derive-Secret. Le schéma général pour ajouter un nouveau secret est d'utiliser HKDF-Extract avec le sel comme état de secret courant et le matériel de chiffrement d'entrée (IKM, *Input Keying Material*) étant le nouveau secret à ajouter. Dans cette version de TLS 1.3, les deux secrets d'entrée sont :

- PSK (une clé pré partagée établie en externe ou déduite de la valeur resumption_master_secret *(secret maître de reprise)* provenant d'une connexion précédente)
- secret partagé (EC)DHE (paragraphe 7.4)

Cela produit un programme complet de déduction de clé montré dans le diagramme ci-dessous. Dans ce diagramme, les conventions de formatage suivantes s'appliquent :

- HKDF-Extract est dessiné comme prenant l'argument Sel du sommet et l'argument IKM de la gauche, avec son résultat en bas et le nom du résultat sur la droite.
- L'argument secret de Derive-Secret est indiqué par la flèche entrante. Par exemple, le secret Early est le secret pour générer le client_early_traffic_secret.
- "0" indique une chaîne de Hash.length octets réglés à zéro.

```
          0
          |
          v
PSK ->  HKDF-Extract = Early Secret
          |
          +-----> Derive-Secret(., "ext binder" | "res binder", "") = binder_key
          |
          +-----> Derive-Secret(., "c e traffic", ClientHello) = client_early_traffic_secret
          |
          +-----> Derive-Secret(., "e exp master", ClientHello) = early_exporter_master_secret
          |
          v
    Derive-Secret(., "derived", "")
          |
          v
(EC)DHE -> HKDF-Extract = Handshake Secret
          |
          +-----> Derive-Secret(., "c hs traffic",  ClientHello...ServerHello) = client_handshake_traffic_secret
          |
          +-----> Derive-Secret(., "s hs traffic", ClientHello...ServerHello) = server_handshake_traffic_secret
          |
          v
    Derive-Secret(., "derived", "")
          |
          v
0 -> HKDF-Extract = Master Secret
          |
```

```
+-----> Derive-Secret(., "c ap traffic", ClientHello...serveur Finished) = client_application_traffic_secret_0
|
+-----> Derive-Secret(., "s ap traffic", ClientHello...serveur Finished)  = server_application_traffic_secret_0
|
+-----> Derive-Secret(., "exp master",  ClientHello...serveur Finished) = exporter_master_secret
|
+-----> Derive-Secret(., "res master", ClientHello...client Finished) = resumption_master_secret
```

Le schéma général est ici que les secrets montrés en bas du côté gauche du diagramme sont juste l'entropie brute sans contexte, tandis que les secrets en bas du côté droit incluent le contexte de prise de contact et peuvent donc être utilisés pour déduire les clés de travail sans contexte supplémentaire. Noter que les différents appels à Derive-Secret peuvent prendre des arguments de messages différents, même avec le même secret. Dans un échange 0-RTT, Derive-Secret est appelé avec quatre transcriptions distinctes ; dans un échange 1-RTT seulement, il est appelé avec trois transcriptions distinctes.

Si un certain secret n'est pas disponible, alors la valeur 0 consistant en une chaîne de Hash.length octets réglés à zéro est utilisée. Noter que ceci ne signifie pas de sauter des tours, de sorte que si PSK n'est pas utilisé, Early Secret sera quand même HKDF-Extract(0, 0). Pour le calcul de binder_key, l'étiquette est "ext binder" pour les PSK externes (celles qui sont provisionnées en dehors de TLS) et "res binder" pour les PSK de reprise (celles provisionnées comme secret maître de reprise d'une prise de contact précédente). Les différentes étiquettes empêchent la substitution d'un type de PSK à un autre.

Il y a plusieurs valeurs potentielles de Early Secret, selon la PSK que le serveur choisit en fin de compte. Le client va avoir besoin d'en calculer une pour chaque PSK potentielle ; si aucune PSK n'est choisie, il devra alors calculer le Early Secret correspondant à la PSK zéro.

Une fois que toutes les valeurs qui sont à déduire d'un certain secret ont été calculées, ce secret DEVRAIT être détruit.

7.2 Mise à jour des secrets de trafic

Une fois la prise de contact achevée, il est possible à l'un et l'autre côté de mettre à jour ses clés d'envoi de trafic en utilisant le message de prise de contact KeyUpdate défini au paragraphe 4.6.3. La prochaine génération de clés de trafic est calculée en générant client_/server_application_traffic_secret_N+1 à partir de client_/server_application_traffic_secret_N comme décrit dans cette section et ensuite en re-déduisant les clés de trafic comme décrit au paragraphe 7.3.

Le application_traffic_secret de prochaine génération est calculé comme suit :

```
application_traffic_secret_N+1 = HKDF-Expand-Label(application_traffic_secret_N, "traffic upd", "", Hash.length)
```

Une fois que client_/server_application_traffic_secret_N+1 et ses clés de trafic associées ont été calculés, les mises en œuvre DEVRAIENT supprimer client_/server_application_traffic_secret_N et ses clés de trafic associées.

7.3 Calcul de clé de trafic

Le matériel de chiffrement du trafic est généré à partir des valeurs d'entrée suivantes :
- une valeur secrète
- une valeur ad hoc indiquant la valeur spécifique générée
- la longueur de la clé générée.

Le matériel de chiffrement du trafic est généré à partir d'une valeur d'entrée de secret du trafic utilisant :

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
[sender]_write_iv  = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

[sender] note le côté envoyeur. La valeur du secret pour chaque type d'enregistrement est montrée dans le tableau ci-dessous.

Type d'enregistrement	**Secret**
Application 0-RTT	client_early_traffic_secret
Prise de contact	[sender]_handshake_traffic_secret
Données d'application	[sender]_application_traffic_secret_N |

Tout le matériel de chiffrement du trafic est recalculé chaque fois que le secret sous-jacent change (par exemple, quand on passe des clés de prise de contact à celles des données d'application ou lors d'une mise à jour de clé).

7.4 Calcul de secret partagé (EC)DHE

7.4.1 Diffie-Hellman de champ fini

Pour les groupes de champs finis, un calcul conventionnel Diffie-Hellman [DH76] est effectué. La clé négociée (Z) est convertie en une chaîne d'octets en la codant en format gros boutien et en la bourrant à gauche avec des zéros jusqu'à la taille du nombre premier. Cette chaîne d'octets est utilisée comme secret partagé dans le programme de clé comme spécifié ci-dessus.

Noter que cette construction diffère des versions antérieures de TLS qui supprimaient les zéros de tête.

7.4.2 Diffie-Hellman de courbe elliptique

Pour secp256r1, secp384r1, et secp521r1, les calculs ECDH (incluant la génération de paramètre et de clé ainsi que le calcul de secret partagé) sont effectués conformément à [IEEE1363] en utilisant le schéma ECKAS-DH1 avec l'identité transposée en fonction de déduction de clé (KDF, *key derivation function*) de sorte que le secret partagé est la coordonnée x du point de la courbe elliptique du secret partagé ECDH représentée comme chaîne d'octets. Noter que cette chaîne d'octets ("Z" dans la terminologie IEEE 1363) telle que sortie par la primitive de conversion d'élément de champ en chaîne d'octet (FE2OSP, *Field Element to Octet String Conversion Primitive*) a une longueur constante pour tout champ ; les zéros qui se trouvent en tête de cette chaîne d'octets NE DOIVENT PAS être tronqués.

(Noter que cette utilisation de l'identité KDF est technique. La peinture complète est que ECDH est employé avec une KDF non triviale parce que TLS n'utilise pas directement ce secret pour autre chose que le calcul d'autres secrets.)

Pour X25519 et X448, les calculs d'ECDH sont les suivants :

- La clé publique à mettre dans la structure KeyShareEntry.key_exchange est le résultat de l'application de la fonction de multiplication scalaire ECDH à la clé secrète de longueur appropriée (dans l'entrée scalaire) et le point de base standard public (dans l'entrée du point de coordonnée u).

- Le secret partagé ECDH est le résultat de l'application de la fonction de multiplication scalaire ECDH à la clé secrète (dans l'entrée scalaire) et la clé publique de l'homologue (dans l'entrée du point de coordonnée u). Le résultat est utilisé brut, sans traitement.

Pour ces courbes, les mises en œuvre DEVRAIENT utiliser l'approche spécifiée dans la [RFC7748] pour calculer le secret partagé Diffie-Hellman. Les mises en œuvre DOIVENT vérifier si le secret partagé Diffie-Hellman calculé est la valeur toute de zéros, et interrompre si c'est le cas, comme décrit à la Section 6 de la [RFC7748]. Si les mises en œuvre utilisent une autre mise en œuvre de ces courbes elliptiques, elles DEVRAIENT effectuer les vérifications supplémentaires spécifiées à la Section 7 de la [RFC7748].

7.5 Exporteurs

La [RFC5705] définit les exporteurs de matériel de chiffrement pour TLS dans les termes de la fonction pseudo aléatoire (PRF, *pseudorandom function*) de TLS. Le présent document remplace la PRF par HKDF, exigeant donc une nouvelle construction. L'interface d'exporteur reste la même.

La valeur d'exporteur est calculée par :

```
TLS-Exporter(label, context_value, key_length) = HKDF-Expand-Label(Derive-Secret(Secret, label, ""), "exporter",
    Hash(context_value), key_length)
```

Où Secret est soit le early_exporter_master_secret, soit le exporter_master_secret. Les mises en œuvre DOIVENT utiliser le exporter_master_secret sauf spécification contraire explicite par l'application. Le early_exporter_master_secret est défini pour être utilisé dans des réglages où un exporteur est nécessaire pour les données 0-RTT. Une interface séparée pour l'exporteur précoce est RECOMMANDÉE ; cela évite que l'utilisateur de l'exporteur utilise accidentellement un exporteur précoce quand un régulier est désiré ou vice versa.

Si aucun contexte n'est fourni, la context_value est de longueur zéro. Par conséquent, ne fournir aucun contexte calcule la

même valeur que de fournir un contexte vide. Ceci est un changement par rapport aux versions antérieures de TLS où un contexte vide produisait un résultat différent qu'un contexte absent. Au moment de la publication du présent document, aucune étiquette d'exporteur allouée n'est utilisée à la fois avec et sans un contexte. De futures spécifications NE DOIVENT PAS définir une utilisation des exporteurs qui permette à la fois un contexte vide et pas de contexte avec la même étiquette. De nouvelles utilisations des exporteurs DEVRAIENT fournir un contexte dans tous les calculs d'exporteur, bien que la valeur puisse être vide.

Les exigences pour le format des étiquettes d'exporteur sont définies à la Section 4 de la [RFC5705].

8. 0-RTT et anti-répétition

Comme noté au paragraphe 2.3 et à l'Appendice E.5, TLS ne fournit pas de protection inhérente contre la répétition pour les données 0-RTT. Deux menaces potentielles posent problème :

- Des attaquants du réseau qui montent une attaque en répétition en dupliquant simplement un envoi de données 0-RTT.

- Des attaquants du réseau qui tirent parti d'un comportement de réessai du client pour s'arranger pour que le serveur reçoive plusieurs copies d'un message d'application. Cette menace existe déjà dans une certaine mesure parce que les clients qui privilégient la robustesse répondent aux erreurs du réseau en tentant des demandes de réessai. Cependant, 0-RTT ajoute une dimension supplémentaire à tout système de serveur qui ne conserve pas un état de serveur globalement cohérent. Précisément, si un système de serveur a plusieurs zones où les tickets provenant de la zone A ne seront pas acceptés dans la zone B, un attaquant peut alors dupliquer un ClientHello et des données précoces destinées à A pour A et B. En A, les données seront acceptées dans 0-RTT, mais en B le serveur va rejeter les données 0-RTT et forcer plutôt à une pleine prise de contact. Si l'attaquant bloque le ServerHello provenant de A, le client va alors achever la prise de contact avec B et probablement réessayer la demande, conduisant à la duplication sur le système serveur comme un tout.

La première classe d'attaque peut être empêchée en partageant l'état pour garantir que les données 0-RTT seront acceptées au plus une fois. Les serveurs DEVRAIENT fournir ce niveau de protection contre la répétition en mettant en œuvre une des méthodes décrites dans cette section ou par des moyens équivalents. On comprend néanmoins que du fait des problèmes de fonctionnement, tous les déploiements ne conserveront pas l'état à ce niveau. Donc, en fonctionnement normal, les clients ne vont pas savoir lequel de ces mécanismes, s'il en est, les serveurs mettent réellement en œuvre, et donc DOIVENT seulement envoyer des données précoces dont ils estiment qu'elles peuvent être répétées sans danger.

En plus des effets directs des répétitions, il y a une classe d'attaques où même des opérations normalement considérées comme idempotentes pourraient être exploitées par un grand nombre de répétitions (attaques de synchronisation, épuisement des limites de ressource et autres, comme décrit à l'appendice E.5). Elles peuvent être atténuées en s'assurant que chaque charge utile 0-RTT ne peut être répétée qu'un nombre limité de fois. Le serveur DOIT s'assurer que toute instance (que ce soit une machine, une trame, ou toute autre entité au sein de l'infrastructure de service pertinente) va accepter 0-RTT pour la même prise de contact 0-RTT au plus une fois ; cela limite le nombre de répétitions au nombre d'instances de serveur dans le système. Une telle garantie peut être obtenue en enregistrant en local les données provenant des ClientHello récemment reçus et en rejetant les répétitions, ou par toute autre méthode qui fournit une garantie égale ou plus forte. La garantie "au plus une fois par instance de serveur" est une exigence minimale ; les serveurs DEVRAIENT plus limiter les répétitions de 0-RTT lorsque c'est faisable.

La seconde classe d'attaques ne peut pas être empêchée à la couche TLS et DOIT être traitée par toute application. Noter que toute application dont les clients mettent en œuvre un comportement de réessai a d'ores et déjà besoin de mettre en œuvre une certaine forme de défense anti répétition.

8.1 Tickets à usage unique

La plus simple forme de défense anti répétition est que le serveur permette seulement que chaque ticket de session soit utilisé une fois. Par exemple, le serveur peut tenir une base de données de tous les tickets valides en cours, supprimant chaque ticket de la base lorsque il est utilisé. Si un ticket inconnu est fourni, le serveur va alors revenir à une pleine prise de contact.

Si les tickets ne sont pas contenus eux-mêmes mais sont plutôt des clés d'une base de données, et si les PSK correspondantes sont supprimées après usage, les connexions établies en utilisant les PSK jouissent alors du secret vers l'avant. Cela améliore la sécurité pour toutes les données 0-RTT et l'usage de PSK lorsque une PSK est utilisée sans (EC)DHE.

Parce que ce mécanisme exige de partager la base de données de session entre les nœuds serveurs dans des environnements avec plusieurs serveurs répartis, il peut être difficile de réaliser de forts taux de connexions 0-RTT à PSK réussies comparés à celui des tickets auto chiffrés. À la différence des bases de données de session, les tickets de session peuvent réussir l'établissement de session fondé sur la PSK même sans mémorisation consistante, bien que quand 0-RTT est permis ils puissent encore exiger une mémorisation consistante pour l'anti répétition des données 0-RTT, comme précisé au paragraphe suivant.

8.2 Enregistrement de Hello de client

Une autre forme d'anti répétition est d'enregistrer une valeur unique déduite du ClientHello (généralement, soit la valeur aléatoire, soit le liant de PSK) et de rejeter les dupliqués. Enregistrer tous les ClientHello cause une croissance sans limite de l'état, mais un serveur peut plutôt enregistrer les ClientHello dans une certaine fenêtre de temps et utiliser le "obfuscated_ticket_age" pour s'assurer que les tickets ne sont pas réutilisés en dehors de cette fenêtre.

Afin de mettre cela en œuvre, quand un ClientHello est reçu, le serveur vérifie d'abord le liant de PSK, comme décrit au paragraphe 4.2.11. Puis il calcule le expected_arrival_time comme décrit au paragraphe suivant et rejette le 0-RTT si il est en dehors de la fenêtre d'enregistrement, revenant à la prise de contact 1-RTT.

Si le expected_arrival_time est dans la fenêtre, le serveur vérifie alors si il a enregistré un ClientHello correspondant. Si il en trouve un, soit il interrompt la prise de contact avec une alerte "paramètre_illégal", soit il accepte la PSK mais rejette le 0-RTT. Si aucun ClientHello correspondant n'est trouvé, il accepte alors 0-RTT et ensuite mémorise le ClientHello pendant aussi longtemps que le expected_arrival_time est à l'intérieur de la fenêtre. Les serveurs PEUVENT aussi mettre en œuvre les magasins de données avec des faux positifs, comme les filtres Bloom, auquel cas ils DOIVENT répondre à la répétition apparente en rejetant le 0-RTT mais NE DOIVENT PAS interrompre la prise de contact.

Le serveur DOIT déduire la clé de mémorisation seulement à partir des sections validées du ClientHello. Si le ClientHello contient plusieurs identités de PSK, un attaquant peut alors créer plusieurs ClientHello avec des valeurs différentes de liant pour l'identité la moins préférée dans l'hypothèse où le serveur ne va pas la vérifier (comme recommandé au paragraphe 4.2.11). C'est-à-dire que si le client envoie les PSK A et B mais que le serveur préfère A, l'attaquant peut changer le liant pour B sans affecter le liant pour A. Si le liant pour B fait partie de la clé de mémorisation, ce ClientHello ne va pas apparaître comme un dupliqué, ce qui va causer l'acceptation du ClientHello, et peut causer des effets collatéraux comme une pollution de l'antémémoire de répétition, bien que toutes les données 0-RTT ne soient pas déchiffrables parce qu'elles vont utiliser des clés différentes. Si le liant validé ou le ClientHello.random est utilisé comme clé de mémorisation, cette attaque n'est alors pas possible.

Parce que ce mécanisme n'exige pas de mémoriser tous les tickets en cours, il peut être plus facile de le mettre en œuvre dans des systèmes répartis avec de forts taux de reprise et 0-RTT, au prix d'une défense anti répétition potentiellement plus faible à cause de la difficulté de mémoriser de façon fiable et restituer les messages ClientHello reçus. Dans de nombreux systèmes comme ceux là, il est impraticable d'avoir une mémorisation globalement cohérente pour tous les ClientHello reçus. Dans ce cas, la meilleure protection anti répétition est fournie en ayant une seule zone de mémorisation d'autorité pour un certain ticket et en refusant 0-RTT pour ce ticket dans toute autre zone. Cette approche empêche la simple répétition par l'attaquant parce que seulement une zone va accepter les données 0-RTT. Une conception plus faible est de mettre en œuvre une mémorisation séparée pour chaque zone mais de permettre 0-RTT dans toute zone. Cette approche limite le nombre de répétitions à une fois par zone. La duplication de message d'application reste bien sûr possible avec l'une ou l'autre conception.

Lorsque les mises en œuvre viennent de démarrer, elles DEVRAIENT rejeter 0-RTT tant qu'une portion de leur fenêtre d'enregistrement se chevauche avec l'heure de démarrage. Autrement, elles courent le risque d'accepter des répétitions qui ont été à l'origine envoyées durant cette période.

Note : Si l'horloge du client fonctionne plus vite que celle du serveur, un ClientHello peut alors être reçu qui soit en dehors de la fenêtre à l'avenir, auquel cas il peut être accepté pour 1-RTT, causant un réessai du client, et ensuite acceptable ultérieurement pour 0-RTT. C'est une autre variante de la seconde forme d'attaque décrite à la Section 8.

8.3 Vérifications de fraîcheur

Parce que le ClientHello indique l'heure à laquelle le client l'a envoyé, il est possible de déterminer efficacement si un ClientHello a probablement été envoyé raisonnablement récemment et accepter seulement 0-RTT pour un tel ClientHello, revenant autrement à une prise de contact 1-RTT. Ceci est nécessaire pour le mécanisme de mémorisation de ClientHello décrit au paragraphe 8.2 parce que autrement le serveur aurait besoin de mémoriser un nombre illimité de ClientHello, et

c'est une optimisation utile pour les tickets auto contenus à usage unique parce qu'elle permet un rejet efficace des ClientHello qui ne peuvent pas être utilisés pour 0-RTT.

Afin de mettre en œuvre ce mécanisme, un serveur a besoin de mémoriser l'heure à laquelle le serveur a généré le ticket de session, décalé du temps d'aller-retour estimé entre client et serveur. C'est-à-dire,

heure de création ajustée = heure de création + RTT estimé

Cette valeur peut être codée dans le ticket, évitant ainsi d'avoir besoin de conserver l'état pour chaque ticket en cours. Le serveur peut déterminer la vue du client de l'âge du ticket en soustrayant la valeur "ticket_age_add" du ticket du paramètre "obfuscated_ticket_age" dans l'extension "pre_shared_key" du client. Le serveur peut déterminer l'heure d'arrivée attendue (expected_arrival_time) du ClientHello comme :

heure d'arrivée attendue = heure de création ajustée + âge du ticket du clients

Lorsque un nouveau ClientHello est reçu, l'heure d'arrivée attendue est alors comparée à l'heure actuelle de l'horloge du serveur et si elles diffèrent de plus d'une certaine valeur, 0-RTT est rejeté, bien que la prise de contact 1-RTT puisse être permise pour la compléter.

Il y a plusieurs sources d'erreur potentielles qui pourraient causer des discordances entre l'heure d'arrivée attendue et l'heure mesurée. Des variations des débits des horloges du client et du serveur seront probablement minimales, bien que potentiellement les temps absolus puissent différer de grandes valeurs. Les délais de propagation du réseau vont plus probablement causer une discordance des valeurs légitimes pour le temps écoulé. Les deux messages NewSessionTicket et ClientHello peuvent être retransmis et donc retardés, ce qui peut être caché par TCP. Pour les clients sur l'Internet, cela implique des fenêtres de l'ordre de dix secondes pour prendre en compte les erreurs d'horloge et le variations de mesures ; d'autres scénarios de déploiement peuvent avoir des besoins différents. Les distributions de biais d'horloge ne sont pas symétriques, de sorte que le compromis optimal peut impliquer une gamme asymétrique de discordances permissibles des valeurs.

Noter que les vérifications de fraîcheur ne sont pas suffisantes pour empêcher les répétitions parce qu'elles ne les détectent pas durant la fenêtre d'erreur, qui – selon la bande passante et les capacités su système – pourrait inclure des milliards de répétitions avec les réglages du monde réel. De plus, cette vérification de fraîcheur est seulement faite au moment où le ClientHello est reçu et non lorsque les enregistrements de données d'application précoces suivants sont reçus. Après que les données précoces sont acceptées, les enregistrements peuvent continuer d'être écoulés au serveur sur une plus longue période.

9. Exigences de conformité

9.1 Suites de chiffrement de mise en œuvre obligatoire

En l'absence d'une norme de profil d'application spécifiant autrement :

Une application conforme à TLS DOIT mettre en œuvre la suite de chiffrement TLS_AES_128_GCM_SHA256 [GCM] et DEVRAIT mettre en œuvre les suites de chiffrement TLS_AES_256_GCM_SHA384 [GCM] et TLS_CHACHA20_POLY1305_SHA256 [RFC8439] (voir l'Appendice B.4).

Une application conforme à TLS DOIT prendre en charge les signatures numériques avec rsa_pkcs1_sha256 (pour les certificats) rsa_pss_rsae_sha256 (pour CertificateVerify et les certificats) et ecdsa_secp256r1_sha256. Une application conforme à TLS DOIT prendre en charge l'échange de clés avec secp256r1 (NIST P-256) et DEVRAIT prendre en charge l'échange de clés avec X25519 [RFC7748].

9.2 Extensions de mise en œuvre obligatoire

En l'absence d'une norme de profil d'application spécifiant autrement, une application conforme à TLS DOIT mettre en œuvre les extensions TLS suivantes :

- Versions prises en charge ("supported_versions"; paragraphe 4.2.1)
- Mouchard ("cookie"; paragraphe 4.2.2)
- Algorithmes de signature ("signature_algorithms"; paragraphe 4.2.3)
- Certificat d'algorithmes de signature ("signature_algorithms_cert"; paragraphe 4.2.3)
- Groupes négociés ("supported_groups"; paragraphe 4.2.7)
- Partage de clé ("key_share"; paragraphe 4.2.8)

- Indication du nom du serveur ("server_name"; Section 3 de la [RFC6066])

Toutes les mises en œuvre DOIVENT envoyer et utiliser ces extensions lors de l'offre des caractéristiques applicables :
- "supported_versions" est EXIGÉ pour tous les messages ClientHello, ServerHello, et HelloRetryRequest.
- "signature_algorithms" est EXIGÉ pour l'authentification de certificat.
- "supported_groups" est EXIGÉ pour les messages ClientHello utilisant l'échange de clés DHE ou ECDHE.
- "key_share" est EXIGÉ pour l'échange de clés DHE ou ECDHE.
- "pre_shared_key" est EXIGÉ pour l'accord de clés PSK.
- "psk_key_exchange_modes" est EXIGÉ pour l'accord de clés PSK.

Un client est considéré comme tentant de négocier l'utilisation de la présente spécification si le ClientHello contient une extension "supported_versions" avec 0x0304 contenu dans son corps. Un tel message ClientHello DOIT satisfaire les exigences suivantes :
- Si il ne contient pas une extension "pre_shared_key", il DOIT contenir à la fois une extension "signature_algorithms" et une extension "supported_groups".
- Si il contient une extension "supported_groups", il DOIT aussi contenir une extension "key_share", et vice versa. Un vecteur KeyShare.client_shares vide est permis.

Les serveurs qui reçoivent un ClientHello qui ne se conforme pas à ces exigences DOIVENT interrompre la prise de contact avec une alerte "extension manquante".

De plus, toutes les mises en œuvre DOIVENT prendre en charge l'utilisation de l'extension "server_name" avec les applications capables de l'utiliser. Les serveurs PEUVENT exiger que les clients envoient une extension "server_name" valide. Les serveurs qui exigent cette extension DEVRAIENT répondre à un ClientHello qui n'a pas une extension "server_name" en terminant la connexion avec une alerte "extension manquante".

9.3. Invariants du protocole

Ce paragraphe décrit les invariants que les points d'extrémité et boîtiers de médiation TLS DOIVENT suivre. Il s'applique aussi aux versions antérieures de TLS.

TLS est conçu pour être extensible de façon sûre et compatible. Les clients ou serveurs plus récents, lorsque ils communiquent avec des homologues récents, devraient négocier les paramètres communs préférés. La prise de contact TLS fournit la protection contre la dégradation : les boîtiers de médiation qui passent du trafic entre un client récent et un serveur récent sans terminer TLS devraient être rendus incapables d'influencer la prise de contact (voir l'Appendice E.1). En même temps, les déploiements sont mis à jour à des moments différents, de sorte qu'un client ou serveur récent PEUT continuer de prendre en charge de paramètres plus anciens, ce qui lui permettra d'interopérer avec les points d'extrémité plus anciens.

Pour cela, les mises en œuvre DOIVENT traiter correctement les champs extensibles :
- Un client qui envoie un ClientHello DOIT prendre en charge tous les paramètres qui y sont annoncés. Autrement, le serveur peut manquer à interopérer en choisissant un de ces paramètres.
- Un serveur qui reçoit un ClientHello DOIT correctement ignorer toutes les suites de chiffrement, extensions, et autres paramètres non reconnus. Autrement, il peut manquer à interopérer avec les clients les plus récents. Dans TLS 1.3, un client qui reçoit un CertificateRequest ou un NewSessionTicket DOIT aussi ignorer toutes les extensions non reconnues.
- Un boitier de médiation qui termine une connexion TLS DOIT se comporter comme un serveur conforme à TLS (à l'égard du client d'origine) incluant d'avoir un certificat que le client accepte, et aussi comme un client conforme à TLS (à l'égard du serveur d'origine) incluant de vérifier le certificat du serveur d'origine. En particulier, il DOIT générer son propre ClientHello contenant seulement des paramètres qu'il comprend, et il DOIT générer une valeur fraîche de ServerHello aléatoire, plutôt que de transmettre la valeur du point d'extrémité.

Noter que les exigences et analyses de sécurité de protocole de TLS s'appliquent seulement aux deux connexions séparément. Le déploiement sûr d'une terminaison TLS exige des considérations de sécurité supplémentaires qui sortent du domaine d'application du présent document.

- Un boîtier de médiation qui transmet des paramètres de ClientHello qu'il ne comprend pas NE DOIT PAS traiter de messages au delà de ce ClientHello. Il DOIT transmettre tout le trafic suivant non modifié. Autrement, il peut manquer à interopérer avec des clients et serveurs plus récents.

Les ClientHello transmis peuvent contenir des annonces pour des caractéristiques non prises en charge par le boîtier de médiation, de sorte que la réponse peut inclure de futurs ajouts à TLS que le boîtier de médiation ne reconnaît pas. Ces

ajouts PEUVENT changer arbitrairement tout message au delà du ClientHello. En particulier, les valeurs envoyées dans le ServerHello peuvent changer, le format du ServerHello peut changer, et le format du TLSCiphertext peut changer.

La conception de TLS 1.3 a été contrainte par le large déploiement de boîtiers de médiation non conformes à TLS (voir l'Appendice D.4) ; cependant, elle n'assouplit pas les invariants. Ces boîtiers de médiation continuent d'être non conformes.

10. Considérations sur la sécurité

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

11. Considérations relatives à l'IANA

Le présent document utilise plusieurs registres qui ont été à l'origine créés dans la [RFC4346] et mis à jour dans la [RFC8447]. L'IANA a mis à jour ces référence avec le présent document. Les registres et leurs politiques d'allocation sont ci-dessous :

- Registre des suites de chiffrement TLS : les valeurs avec le premier octet dans la gamme 0 à 254 (en décimal) sont allouées via une spécification exigée [RFC8126]. Les valeurs avec le premier octet à 255 (décimal) sont réservées pour utilisation privée [RFC8126]. L'IANA a ajouté les suites de chiffrement mentionnées à l'Appendice B.4 au registre. Les colonnes "Valeur" et "Description" sont tirées de ce tableau. Les colonnes "DTLS-OK" et "Recommandé" sont toutes deux marquées "Y" pour chaque nouvelle suite de chiffrement.

- Registre ContentType TLS : les valeurs futures seront allouées via action de normalisation [RFC8126].

- Registre des alertes TLS : les valeurs futures seront allouées via action de normalisation [RFC8126]. L'IANA a rempli ce registre avec les valeurs de l'Appendice B.2. La colonne "DTLS-OK" est marquée "Y" pour toutes ces valeurs. Les valeurs marquées "_RESERVED" ont des commentaires qui décrivent leur utilisation précédente.

- Registre HandshakeType TLS : les valeurs futures seront allouées via action de normalisation [RFC8126]. L'IANA a mis à jour ce registre pour changer le nom de l'élément 4 de "NewSessionTicket" en "new_session_ticket" et rempli ce registre avec les valeurs de l'Appendice B.3. La colonne "DTLS-OK" est marquée "Y" pour toutes ces valeurs. Les valeurs marquées "_RESERVED" ont des commentaires qui décrivent leur utilisation précédente ou temporaire.

Le présent document utilise aussi le registre des valeurs de ExtensionType TLS créé à l'origine dans la [RFC4366]. L'IANA l'a mis à jour pour référencer le présent document. Les changements au registre sont :

- L'IANA a mis à jour la politique d'enregistrement comme suit : les valeurs avec le premier octet dans la gamme 0 à 254 (décimal) sont allouées via spécification exigée [RFC8126]. Les valeurs avec le premier octet à 255 (décimal) sont réservées pour utilisation privée [RFC8126].

- L'IANA a mis à jour ce registre pour inclure les extensions "key_share", "pre_shared_key", "psk_key_exchange_modes", "early_data", "cookie", "supported_versions", "certificate_authorities", "oid_filters", "post_handshake_auth", et "signature_algorithms_cert" avec les valeurs définies dans le présent document et la valeur "Recommandée" de "Y".

- L'IANA a mis à jour ce registre pour inclure une colonne "TLS 1.3" qui fait la liste des messages dans lesquels l'extension peut apparaître. Cette colonne a été initialement remplie à partir du tableau du paragraphe 4.2, avec toute extension non mentionnée marquée par "-" pour indiquer qu'elle n'est pas utilisée par TLS 1.3.

Le présent document met à jour une entrée dans le registre Types de certificat TLS créé à l'origine dans la [RFC6091] et mis à jour dans la [RFC8447]. L'IANA a mis à jour l'entrée pour la valeur 1 comme ayant pour nom "OpenPGP_RESERVED", la valeur "Recommandée" "N", et le commentaire "Utilisé dans les versions de TLS antérieures à 1.3".

Le présent document met à jour une entrée dans le registre TLS Types d'état de certificat créé à l'origine dans la [RFC6961]. L'IANA a mis à jour l'entrée pour la valeur 2 comme ayant pour nom "ocsp_multi_RESERVED" et le commentaire "Utilisé dans les versions de TLS antérieures à 1.3".

Le présent document met à jour deux entrées dans le registre TLS Groupes pris en charge (créé sous un nom différent dans

la [RFC4492] ; tenu maintenant par la [RFC8422]) et mis à jour par les [RFC7919] et [RFC8447]. Les entrées pour les valeurs 29 et 30 (x25519 et x448) ont été mises à jour pour se référer aussi au présent document.

De plus, ce document définit deux nouveaux registres tenus par l'IANA :

- Registre TLS SignatureScheme : les valeurs avec le premier octet dans la gamme de 0 à 253 (décimal) sont alloués via spécification exigée [RFC8126]. Les valeurs avec le premier octet à 254 ou 255 (décimal) sont réservées pour utilisation privée [RFC8126]. Les valeurs avec le premier octet dans la gamme 0 à 6 ou avec le second octet dans la gamme 0 à 3 qui ne sont pas actuellement allouées sont réservés pour la rétro compatibilité. Ce registre a une colonne "Recommandé". Le registre a été rempli initialement avec les valeurs décrites au paragraphe 4.2.3. Les valeurs suivantes sont marquées "Recommandé" : ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256, rsa_pss_pss_sha384, rsa_pss_pss_sha512, et ed25519. La colonne "Recommandé" a reçu une valeur de "N" sauf demande explicite, et l'ajout d'une valeur "Recommandée" de "Y" exige une action de normalisation [RFC8126]. L'approbation de l'IESG est EXIGÉE pour une transition de Y à N.

- Registre TLS PskKeyExchangeMode : Les valeurs dans la gamme de 0 à 253 (décimal) sont allouées via spécification exigée [RFC8126]. Les valeurs 254 et 255 (décimal) sont réservées pour utilisation privée [RFC8126]. Ce registre a une colonne "Recommandé". Le registre a été initialement rempli avec psk_ke (0) et psk_dhe_ke (1). Tous deux sont marqués "Recommandé". La colonne "Recommandé" a reçu une valeur de "N" sauf demande explicite, et l'ajout d'une valeur de "Recommandé" de "Y" exige une action de normalisation [RFC8126]. L'approbation de l'IESG est EXIGÉE pour une transition de Y à N.

12. Références

12.1. Références normatives

[DH76] Diffie, W. et M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory, Vol. 22 No. 6, pp. 644-654, DOI 10.1109/TIT.1976.1055638, novembre 1976.

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

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) et GMAC", NIST Special Publication 800-38D, DOI 10.6028/NIST.SP.800-38D, novembre 2007.

[RFC2104] H. Krawczyk, M. Bellare et R. Canetti, "HMAC : Hachage de clés pour l'authentification de message", février 1997, DOI 10.17487/RFC2104.

[RFC2119] S. Bradner, "Mots clés à utiliser dans les RFC pour indiquer les niveaux d'exigence", BCP 14, mars 1997. *(MàJ par* RFC8174), DOI 10.17487/RFC2119.

[RFC5116] D. McGrew, "Interface et algorithmes pour le chiffrement authentifié", janvier 2008. *(P.S.)*, DOI 10.17487/RFC5116.

[RFC5280] D. Cooper et autres, "Profil de certificat d'infrastructure de clé publique X.509 et de liste de révocation de certificat (CRL) pour l'Internet", mai 2008. (*Remplace les* RFC3280, RFC4325, RFC4630) *(P.S. ; MàJ par* RFC8398, 8399), DOI 10.17487/RFC5280.

[RFC5705] E. Rescorla; "Exporteurs de matériel de clé de chiffrement pour la sécurité de la couche Transport (TLS)", mars 2010. *(P.S. ; MàJ par* RFC8446), DOI 10.17487/RFC5705.

[RFC5756] S. Turner, D. Brown, K. Yiu, R. Housley, T. Polk, "Mise à jour des paramètres d'algorithmes RSAES-OAEP et RSASSA-PSS", janvier 2010. (*MàJ* RFC4055). *(P. S.)*, DOI 10.17487/RFC5756.

[RFC5869] H. Krawczyk, P. Eronen, "Fonction de déduction de clé par extraction et expansion fondée sur HMAC (HKDF)", mai 2010. *(Information)*, DOI 10.17487/RFC5869.

[RFC6066] D. Eastlake 3rd, " Extensions à la sécurité de la couche Transport (TLS) : Définitions d'extension", janvier 2011. (*Remplace la* RFC4366 *; MàJ par* RFC8446, RFC8449), DOI 10.17487/RFC6066.

[RFC6655] D. McGrew, D. Bailey, "Suites de chiffrement AES-CCM pour la sécurité de la couche Transport (TLS)", juillet 2012. *(P.S.)*, DOI 10.17487/RFC6655.

[RFC6960] S. Santesson et autres, "Protocole d'état de certificat en ligne (OCSP) pour l'infrastructure de clé publique Internet X.509", juin 2013. (*Remplace* RFC2560, 6277) (*MàJ* RFC5912) *(P.S.)*, DOI 10.17487/RFC6960.

[RFC6961] Y. Pettersen, "Extension Demande d'état de certificat multiple pour la sécurité de la couche Transport (TLS)", juin 2013. *(P.S. ; rendue obsolète par la* RFC8446), DOI 10.17487/RFC6961.

[RFC6962] B. Laurie, A. Langley, E. Kaspe, "Transparence de certificat", juin 2013. *(Exp.)*, DOI 10.17487/RFC6962.

[RFC6979] T. Pornin, "Utilisation déterministe de l'algorithme de signature numérique (DSA) et de l'algorithme de signature numérique à courbe elliptique (ECDSA)", août 2013. *(Information)*, DOI 10.17487/RFC6979.

[RFC7001] M. Kucherawy, "Champ d'en-tête de message pour indiquer l'état d'authentification", septembre 2013. (*Remplace* RFC5451, RFC6577, *Remplacée par* RFC7601) *(P.S.)*, DOI 10.17487/RFC7301.

[RFC7507] B. Moeller, A. Langley, "Valeur de suite de chiffrement de signalisation de repli pour TLS pour empêcher les attaques de dégradation de protocole", avril 2015. *(P.S. ; MàJ* 2246, 4346, 5246, 6347*)*, DOI 10.17487/RFC7507.

[RFC7748] A. Langley, et autres, "Courbes elliptiques pour la sécurité", janvier 2016. *(Information)*, DOI 10.17487/RFC7748.

[RFC7919] D. Gilmor, "Négociation de paramètres éphémères de champs Diffie-Hellman finis pour TLS", août 2016. *(P.S. ; MàJ* RFC2246*,* 4346*,* 4492*,* 5246*)*, DOI 10.17487/RFC7919.

[RFC8017] K. Moriarty, et autres, "PKCS n° 1 : Spécifications de cryptographie RSA version 2.2", novembre 2016. *(Info. remplace* RFC3447), DOI 10.17487/RFC8017.

[RFC8032] S. Josefsson, I. Liusvara, "Algorithme de signature numérique Edwards-Curve", janvier 2017. *(Information)*, DOI 10.17487/RFC8032.

[RFC8126] M. Cotton, B. Leiba, T. Narten, "Lignes directrices pour la rédaction d'une section de considérations relatives à l'IANA dans les RFC", juin 2017. BCP 26. *(Remplace* RFC5226), DOI 10.17487/RFC8126.

[RFC8439] Y. Nir, A. Langley, "Chiffrement ChaCha20 et authentificateur Poly1305 pour les protocoles de l'IETF", juin 2018. *(Information, remplace* RFC 7539), DOI 10.17487/RFC8439.

[SHS] Dang, Q., "Secure Hash Standard (SHS)", National Institute of Standards et Technology report, DOI 10.6028/NIST.FIPS.180-4, août 2015.

[X690] UIT-T, "Technologies de l'information – Règles de codage ASN.1 : Spécification des règles de codage de base (BER), règles de codage canoniques (CER) et règles de codage distinctives (DER)", ISO/CEI 8825-1:2015, novembre 2015.

12.2. Références pour information

[AEAD-LIMITS] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", août 2017, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf >.

[BBFGKZ16] Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., and S. Zanella-Beguelin, "Downgrade Resilience in Key-Exchange Protocols", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.37, mai 2016.

[BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models et Reference Les mises en œuvre for the TLS 1.3 Standard Candidate", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2017.26, mai 2017.

[BDF] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings

of IEEE Symposium on Security and Privacy (San Jose), mai 2017, <https://eprint.iacr.org/2016/1178 >.

[Ben17a] Benjamin, D., "Presentation before the TLS WG at IETF 100", novembre 2017, <https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/ >.

[Ben17b] Benjamin, D., "Additional TLS 1.3 results from Chrome", message to the TLS mailing list, 18 décembre 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html >.

[Blei98] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Proceedings of CRYPTO '98, 1998.

[BMMRT15] Badertscher, C., Matt, C., Maurer, U., Rogaway, P., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015, septembre 2015, <https://eprint.iacr.org/2015/394>.

[BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016, juillet 2016, < https://eprint.iacr.org/2016/564 >.

[CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-compromise Security", IEEE Computer Security Foundations Symposium, DOI 10.1109/CSF.2016.19, juillet 2015.

[CHECK] Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., and H. Shacham, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS '16, octobre 2016. DOI 10.1145/2976749.2978395.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., Scott, S., and T. van der Merwe, "Awkward Handshake: Possible mismatch of client/server view on client authentification in post-handshake mode in Revision 18", message to the TLS mailing list, 10 février 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentification", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.35, mai 2016, < https://ieeexplore.ieee.org/document/7546518/ >.

[CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Proceedings of Eurocrypt 2001, DOI 10.1007/3-540-44987-6_28, avril 2001.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies, pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates", Proceedings of ACM CCS 2015, octobre 2015, < https://eprint.iacr.org/2015/914 >.

[DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Full and Pre-shared Key Handshake Protocol", TRON 2016, février 2016, < https://eprint.iacr.org/2016/081 >.

[DOW92] Diffie, W., van Oorschot, P., and M. Wiener, "Authentification and authenticated key exchanges", Designs, Codes and Cryptography, DOI 10.1007/BF00124891, juin 1992.

[DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard (DSS)", NIST FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, juillet 2013.

[FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of EuroS&P 2017, avril 2017, < https://eprint.iacr.org/2017/082 >.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.34, mai 2016, < https://ieeexplore.ieee.org/document/7546517/ >.

[FW15] Weimer, F., "Factoring RSA Keys With TLS Perfect Forward Secrecy", septembre 2015.

[HCJC16] Husak, M., Cermak, M., Jirsik, T., and P. Celeda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security, Vol. 2016, DOI

10.1186/s13635-016-0030-7, février 2016.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora's Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies, août 2015.

[IEEE1363] IEEE, "IEEE Standard Specifications for Public Key Cryptography", IEEE Std. 1363-2000, DOI 0.1109/IEEESTD.2000.92292.

[JSS15] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of ACM CCS 2015, DOI 10.1145/2810103.2813657, octobre 2015, <https://www.nds.rub.de/media/nds/veroeffentlichungen/2015/08/21/Tls13QuicAttacks.pdf >.

[KEYAGR] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology, DOI 10.6028/NIST.SP.800-56Ar3, avril 2018.

[Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010, août 2010, < https://eprint.iacr.org/2010/264 >.

[Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentification Compiler for Key Exchange (avec Applications to Client Authentification in TLS 1.3", Proceedings of ACM CCS 2016, octobre 2016, <https://eprint.iacr.org/2016/711 >.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of EuroS&P 2016, mars 2016, <https://eprint.iacr.org/2015/978 >.

[LXZFH16] Li, X., Xu, J., Zhang, Z., Feng, D., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Proceedings of IEEE Symposium on Security et Privacy (San Jose), DOI 10.1109/SP.2016.36, mai 2016, <https://ieeexplore.ieee.org/document/7546519/ >.

[Mac17] MacCarthaigh, C., "Security Review of TLS1.3 0-RTT", mars 2017, < https://github.com/tlswg/tls13-spec/issues/1001 >.

[PS18] Patton, C. and T. Shrimpton, "Partially specified channels: The TLS 1.3 record layer without elision", 2018, <https://eprint.iacr.org/2018/634 >.

[PSK-FINISHED] Scott, S., Cremers, C., Horvat, M., and T. van der Merwe, "Revision 10: possible attack if client authentification is allowed during PSK", message à la liste de diffusion TLSt, 31octobre 2015, <https://www.ietf.org/mail-archive/web/tls/current/msg18215.html >.

[REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT 2000, DOI 10.1007/3-540-44448-3_42, octobre 2000.

[Res17a] Rescorla, E., "Preliminary data on Firefox TLS 1.3 Middlebox experiment", message à la liste de diffusion TLS, 5 décembre 2017, < https://www.ietf.org/mail-archive/web/tls/current/msg25091.html >.

[Res17b] Rescorla, E., "More compatibility measurement results", message à la liste de diffusion TLS, 22 décembre 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html >.

[RFC3552] E. Rescorla, B. Korver, "Lignes directrices pour la rédaction d'une section de considérations sur la sécurité dans les RFC", juillet 2003. (BCP0072), DOI 10.17487/RFC3552.

[RFC4086] D. Eastlake 3rd, J. Schiller, S. Crocker, "Exigences d'aléa pour la sécurité", juin 2005. (*Remplace* RFC1750) (BCP0106), DOI 10.17487/RFC4086.

[RFC4346] T. Dierks et E. Rescorla, "Protocole de sécurité de la couche Transport (TLS) version 1.1", avril 2006. *(Remplace* RFC2246 *; Remplacée par* RFC5246 *; MàJ par* RFC4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919), DOI 10.17487/RFC4346.

[RFC4366] S. Blake-Wilson et autres, "Extensions de sécurité de la couche Transport (TLS)", avril 2006. (*Obsolète,* RFC5246) *(P.S.)*, DOI 10.17487/RFC4366.

[RFC4492] S. Blake-Wilson et autres, "Suites de chiffrement de cryptographie à courbe elliptique (ECC) pour la sécurité de la couche Transport (TLS)", mai 2006. (*MàJ par* RFC5246, 7919), *(Information ; rendue obsolète par* RFC8422), DOI 10.17487/RFC4492.

[RFC5077] J. Salowey et autres, "Reprise de session de sécurité de la couche Transport (TLS) sans état côté serveur", janvier 2008. (*Remplace* RFC4507) *(P.S. ; rendue obsolète par la* RFC8446), DOI 10.17487/RFC5077.

[RFC5246] T. Dierks, E. Rescorla, "Version 1.2 du protocole de sécurité de la couche Transport (TLS)", août 2008. *(P.S. ; rendue obsolète par la* RFC8446), DOI 10.17487/RFC5246.

[RFC5764] D. McGrew, E. Rescorla, "Extension à la sécurité de la couche de transport de datagrammes (DTLS) pour établir des clés pour le protocole sécurisé de transport en temps réel (SRTP)", mai 2010. *(P. S.)*, DOI 10.17487/RFC5764.

[RFC5929] J. Altman, N. Williams, L. Zhu, "Liaisons de canaux pour TLS", juillet 2010, DOI 10.17487/RFC5929. *(P. S.)*

[RFC6091] N. Mavrogiannopoulos, D. Gillmor, "Utilisation de clés OpenPGP pour l'authentification de sécurité de la couche Transport (TLS)", février 2011. (*Remplace la* RFC5081) *(Information)*, DOI 10.17487/RFC6091.

[RFC6101] A. Freier, P. Karlton, P. Kocher "Protocole de couche de connexion sécurisée (SSL) version 3.0", août 2011. *(Historique)*, DOI 10.17487/RFC6101.

[RFC6176] S. Turner, T. Polk. "Interdiction de l'utilisation de la couche de prises sécurisées (SSL) version 2.0", mars 2011. (*MàJ* RFC2246, RFC4346, RFC5246) *(P. S.)*, DOI 10.17487/RFC6176.

[RFC6347] E. Rescorla, N. Modadugu, "Sécurité de la couche transport de datagrammes, version 1.2", janvier 2012. *(Remplace la* RFC4347) *(P.S. ; MàJ par* RFC7905), DOI 10.17487/RFC6347.

[RFC6520] R. Seggelmann, M. Tuexen, M. Williams, "Extension Heartbeat à la sécurité de la couche Transport (TLS) et à la sécurité de la couche transport de datagrammes (DTLS)", février 2012. *(P.S.)*, DOI 10.17487/RFC6520.

[RFC7230] R. Fielding, et J. Reschke, "Protocole de transfert Hypertexte (HTTP/1.1) : syntaxe et acheminement de message", juin 2014. *(P.S.)*, DOI 10.17487/RFC7230.

[RFC7250] P. Wouters, et autres, "Utilisation de clés publiques brutes dans la sécurité de la couche transport (TLS) et la sécurité de la couche de transport de datagrammes (DTLS)", juin 2014. *(P.S.)*, DOI 10.17487/RFC7250.

[RFC7465] A. Popov, "Interdiction des suites de chiffrement RC4", février 2015. *(P.S. ; MàJ RFC* 5246*,* 4346*,* 2246*)*, DOI 10.17487/RFC7465.

[RFC7568] R. Barnes, et autres, "La couche de prises sûres version 3.0 est déconseillée", juin 2015. *(P.S. ; MàJ* RFC5246*)*, DOI 10.17487/RFC7568.

[RFC7627] K. Bhargavan, et autres, "Extension au secret maître étendu et au hachage de session de la sécurité de couche transport (TLS)", septembre 2015.*(P.S. ; MàJ* RFC5246*)*, DOI 10.17487/RFC7627.

[RFC7685] A. Langley, "Extension de bourrage de message d'accueil de TLS", octobre 2015, *(P.S., MàJ* RFC5246*)*, DOI 10.17487/RFC7685.

[RFC7924] S. Santesson, H. Tschofenig, "Extension d'informations mises en antémémoire dans TLS", juillet 2016. *(P.S.)*, DOI 10.17487/RFC7924.

[RFC8305] D. Schinazi, "Algorithme Happy Eyeballs version 2 : meilleure connectivité en utilisant la concurrence", décembre 2017. *(P.S. ; remplace la* RFC6555*)*, DOI 10.17487/RFC8305.

[RFC8422] Y. Nir, et autres, "Suites de chiffrement de cryptographie à courbe elliptique pour TLS version 1.2 et antérieures", août 2018. *(P.S. ; remplace* RFC4492), DOI 10.17487/RFC8422.

[RFC8447] J. Salowey, S. Turner, "Mise à jour des registres de l'IANA pour TLS/DTLS", août 2018. *(P.S. ; MàJ* RFC 3749, 4680 5077, 5246, 5705, 5878, 6520, 7301), DOI 10.17487/RFC8447.

[RFC8449] M. Thomson, "Extension de limite de taille d'enregistrement pour TLS", août 2018. *(P.S. ; MàJ* RFC6066),

DOI 10.17487/RFC8449.

[RSA] Rivest, R., Shamir, A., L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 21 n° 2, pp. 120-126, DOI 10.1145/359340.359342, février 1978.

[SIGMA] Krawczyk, H., "SIGMA: The 'SIGn-et-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols", Proceedings of CRYPTO 2003, DOI 10.1007/978-3-540-45146-4_24, août 2003.

[SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentification in TLS, IKE, and SSH", Network and Distributed System Security Symposium (NDSS 2016), DOI 10.14722/ndss.2016.23418, février 2016.

[SSL2] Hickman, K., "The SSL Protocol", février 1995.

[TIMING] Boneh, D. and D. Brumley, "Remote Timing Attacks Are Practical", USENIX Security Symposium, août 2003.

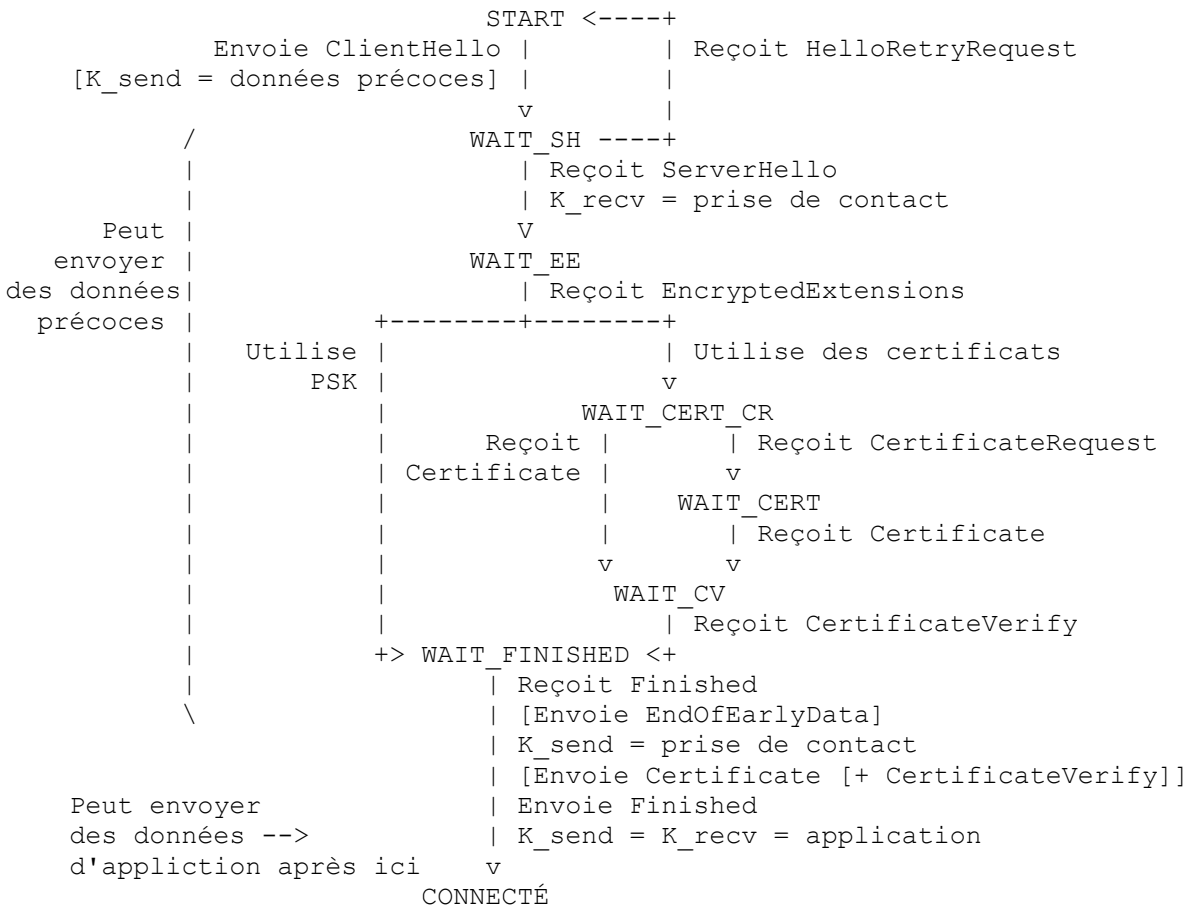
[TLS13-TRACES] Thomson, M., "Example Handshake Traces for TLS 1.3", Travail en cours, draft-ietf-tls-tls13-vectors-06 juillet 2018.

[X501] Recommandation UIT-T X.501, "Technologies de l'information - Interconnexion des systèmes ouverts – L'annuaire : modèles", octobre 2016, <https://www.itu.int/rec/T-REC-X.501/en >.

Appendice A. Automate à états

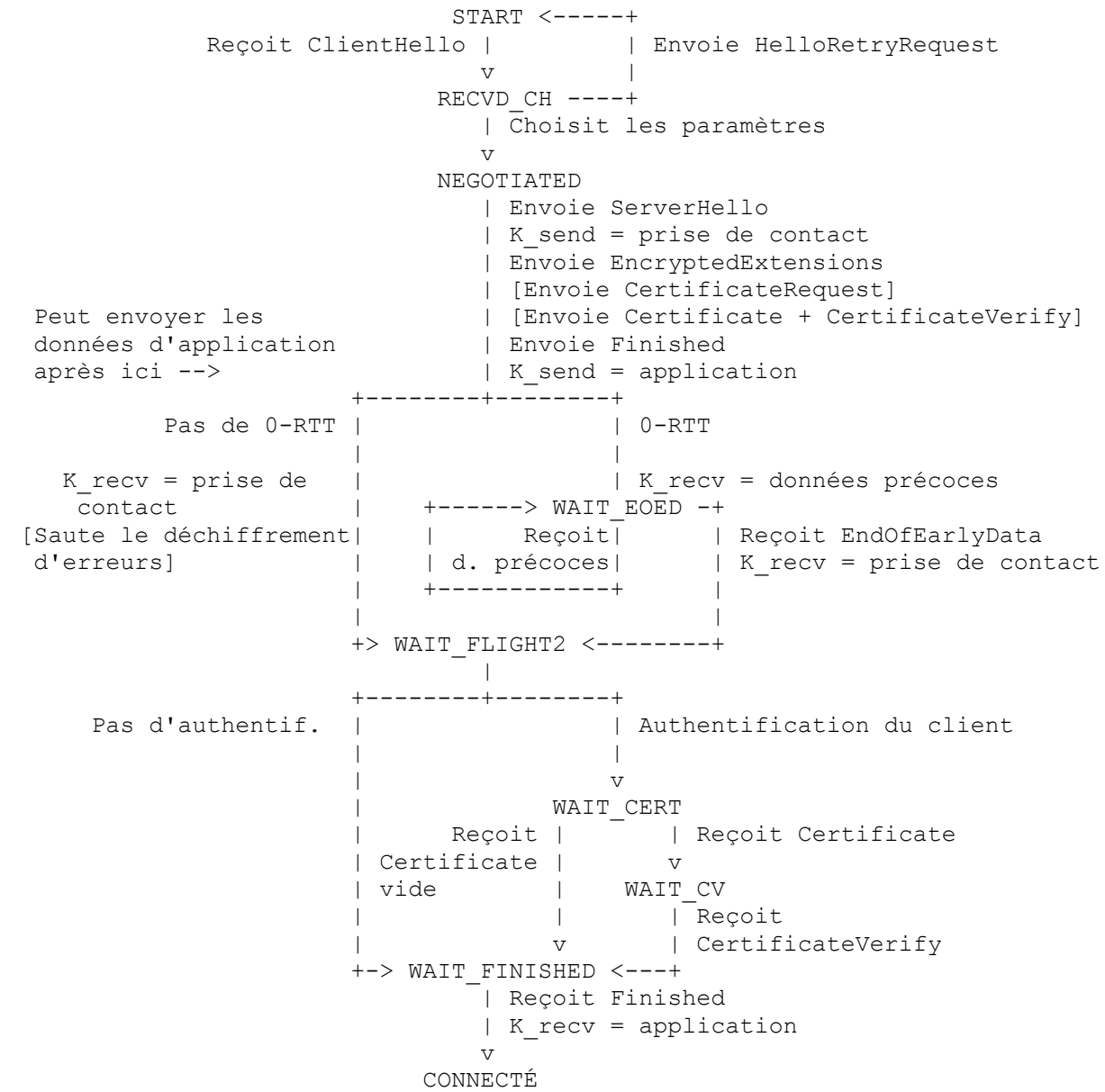
Le présent appendice donne un résumé des transitions d'état légales pour les prises de contact de client et de serveur. Les noms d'états (en majuscules, par exemple, START) n'ont pas de signification formelle mais sont fournis pour faciliter la compréhension. Les actions qui ne sont prises que dans certaines circonstances sont indiquées entre crochets []. La notation "K_{send,recv} = foo" signifie "régler la clé d'envoi/réception à la valeur donnée".

A.1 Client



Noter qu'avec les transitions montrées ci-dessus, les clients peuvent envoyer des alertes qui dérivent de messages post ServerHello en clair ou avec les clés de données précoces. Si les clients ont besoin d'envoyer de telles alertes, ils DEVRAIENT d'abord changer les clés pour les clés de prise de contact, si possible.

A.2. Serveur



Appendice B. Structures et valeurs constantes des données du protocole

Le présent appendice fournit les types normatifs de protocole et la définition des constantes. Les valeurs mentionnées comme "_RESERVED" étaient utilisées dans les versions antérieures de TLS et ne sont mentionnées ici que pour être complet. Les mises en œuvre de TLS 1.3 NE DOIVENT PAS les envoyer mais peuvent les recevoir d'anciennes mises en œuvre de TLS.

B.1 Couche d'enregistrement

```
enum {
  invalid(0),
```

```
      change_cipher_spec(20),
      alert(21),
      handshake(22),
      application_data(23),
      heartbeat(24),              /* RFC 6520 */
      (255)
    } ContentType;

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

    struct {
      opaque content[TLSPlaintext.length];
      ContentType type;
      uint8 zeros[length_of_padding];
    } TLSInnerPlaintext;

    struct {
      ContentType opaque_type = application_data;               /* 23 */
      ProtocolVersion legacy_record_version = 0x0303;           /* TLS v1.2 */
      uint16 length;
      opaque encrypted_record[TLSCiphertext.length];
    } TLSCiphertext;
```

B.2 Messages d'alerte

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

enum {
    close_notify(0),                    (notification de clôture)
    unexpected_message(10),             (message inattendu)
    bad_record_mac(20),                 (mauvais MAC d'enregistrement)
    record_overflow(22),                (enregistrement débordant)
    handshake_failure(40),              (échec de prise de contact)
    bad_certificate(42),                (mauvais certificat)
    unsupported_certificate(43),        (certificat non accepté)
    certificate_revoked(44),            (certificat révoqué)
    certificate_expired(45),            (certificat expiré)
    certificate_unknown(46),            (certificat inconnu)
    illegal_parameter(47),              (paramètre illégal)
    unknown_ca(48),                     (CA inconnue)
    access_denied(49),                  (accès refusé)
    decode_error(50),                   (erreur de décodage)
    decrypt_error(51),                  (erreur de déchiffrement)
    protocol_version(70),               (version de protocole)
    insufficient_security(71),          (sécurité insuffisante)
    internal_error(80),                 (erreur interne)
    inappropriate_fallback(86),         (repli inapproprié)
    user_canceled(90),                  (utilisateur annulé)
    missing_extension(109),             (extension manquante)
    unsupported_extension(110),         (extension non acceptée)
    unrecognized_name(112),             (nom non reconnu)
    bad_certificate_status_response(113), (mauvaise réponse d'état de certificat)
    unknown_psk_identity(115),          (identité PSK inconnue)
    certificate_required(116),          (certificat exigé)
    no_application_protocol(120),       (pas de protocole d'application)
    (255)
  } AlertDescription;
```

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

B.3 Protocole de prise de contact

```
enum {
        hello_request_RESERVED(0),
        client_hello(1),
        server_hello(2),
        hello_verify_request_RESERVED(3),
        new_session_ticket(4),
        end_of_early_data(5),
        hello_retry_request_RESERVED(6),
        encrypted_extensions(8),
        certificate(11),
        server_key_exchange_RESERVED(12),
        certificate_request(13),
        server_hello_done_RESERVED(14),
        certificate_verify(15),
        client_key_exchange_RESERVED(16),
        finished(20),
        certificate_url_RESERVED(21),
        certificate_status_RESERVED(22),
        supplemental_data_RESERVED(23),
        key_update(24),
        message_hash(254),
        (255)
} HandshakeType;

struct {
        HandshakeType msg_type;                 /* prise de contact type */
        uint24 length;                          /* octets dans le message */
        select (Handshake.msg_type) {
           cas client_hello:            ClientHello;
           cas server_hello:            ServerHello;
           cas end_of_early_data:       EndOfEarlyData;
           cas encrypted_extensions:    EncryptedExtensions;
           cas certificate_request:     CertificateRequest;
           cas certificate:             Certificate;
           cas certificate_verify:      CertificateVerify;
           cas finished:                Finished;
           cas new_session_ticket:      NewSessionTicket;
           cas key_update:              KeyUpdate;
        };
} Handshake;
```

B.3.1 Messages d'échange de clé

```
   uint16 ProtocolVersion;
   opaque Random[32];

   uint8 CipherSuite[2];                /* Sélecteur de suite de chiffrement */

   struct {
      ProtocolVersion legacy_version = 0x0303;        /* TLS v1.2 */
      Random random;
      opaque legacy_session_id<0..32>;
      CipherSuite cipher_suites<2..2^16-2>;
      opaque legacy_compression_methods<1..2^8-1>;
```

```
    Extension extensions<8..2^16-1>;
} ClientHello;

struct {
    ProtocolVersion legacy_version = 0x0303;            /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;

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

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
    supported_groups(10),                       /* RFC 8422, 7919 */
    signature_algorithms(13),                   /* RFC 8446 */
    use_srtp(14),                               /* RFC 5764 */
    heartbeat(15),                              /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18),           /* RFC 6962 */
    client_certificate_type(19),                /* RFC 7250 */
    server_certificate_type(20),                /* RFC 7250 */
    padding(21),                                /* RFC 7685 */
    RESERVED(40),                               /* Utilisé mais jamais alloué */
    pre_shared_key(41),                         /* RFC 8446 */
    early_data(42),                             /* RFC 8446 */
    supported_versions(43),                     /* RFC 8446 */
    cookie(44),                                 /* RFC 8446 */
    psk_key_exchange_modes(45),                 /* RFC 8446 */
    RESERVED(46),                               /* Utilisé mais jamais alloué */
    certificate_authorities(47),                /* RFC 8446 */
    oid_filters(48),                            /* RFC 8446 */
    post_handshake_auth(49),                    /* RFC 8446 */
    signature_algorithms_cert(50),              /* RFC 8446 */
    key_share(51),                              /* RFC 8446 */
    (65535)
} ExtensionType;

struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;

struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;

struct {
    KeyShareEntry server_share;
} KeyShareServerHello;

struct {
```

```
      uint8 legacy_form = 4;
      opaque X[coordinate_length];
      opaque Y[coordinate_length];
   } UncompressedPointRepresentation;

   enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

   struct {
      PskKeyExchangeMode ke_modes<1..255>;
   } PskKeyExchangeModes;

   struct {} Vide;

   struct {
      select (Handshake.msg_type) {
         cas new_session_ticket:          uint32 max_early_data_size;
         cas client_hello:                Vide ;
         cas encrypted_extensions:        Vide ;
      };
   } EarlyDataIndication;

   struct {
      opaque identity<1..2^16-1>;
      uint32 obfuscated_ticket_age;
   } PskIdentity;

   opaque PskBinderEntry<32..255>;

   struct {
      PskIdentity identities<7..2^16-1>;
      PskBinderEntry binders<33..2^16-1>;
   } OfferedPsks;

   struct {
      select (Handshake.msg_type) {
         cas client_hello:      OfferedPsks;
         cas server_hello:      uint16 selected_identity;
      };
   } PreSharedKeyExtension;
```

B.3.1.1 Extension de version

```
    struct {
       select (Handshake.msg_type) {
          cas client_hello:      ProtocolVersion versions<2..254>;
          cas server_hello:      ProtocolVersion selected_version;          /* et HelloRetryRequest */
       };
    } SupportedVersions;
```

B.3.1.2 Extension de mouchard

```
    struct {
       opaque cookie<1..2^16-1>;
    } Cookie;
```

B.3.1.3 Extension d'algorithme de signature

```
enum {
/* algorithmes RSASSA-PKCS1-v1_5 */
        rsa_pkcs1_sha256(0x0401),
        rsa_pkcs1_sha384(0x0501),
        rsa_pkcs1_sha512(0x0601),
```

```
/* algorithmes ECDSA */
        ecdsa_secp256r1_sha256(0x0403),
        ecdsa_secp384r1_sha384(0x0503),
        ecdsa_secp521r1_sha512(0x0603),

/* algorithmes RSASSA-PSS avec rsaEncryption d'OID de clé publique */
        rsa_pss_rsae_sha256(0x0804),
        rsa_pss_rsae_sha384(0x0805),
        rsa_pss_rsae_sha512(0x0806),

/* algorithmes EdDSA */
        ed25519(0x0807),
        ed448(0x0808),

/* algorithmes RSASSA-PSS avec RSASSA-PSS d'OID de clé publique */
        rsa_pss_pss_sha256(0x0809),
        rsa_pss_pss_sha384(0x080a),
        rsa_pss_pss_sha512(0x080b),

/* algorithms traditionnels */
        rsa_pkcs1_sha1(0x0201),
        ecdsa_sha1(0x0203),

/* Codets réservés */
    obsolete_RESERVED(0x0000..0x0200),
     dsa_sha1_RESERVED(0x0202),
     obsolete_RESERVED(0x0204..0x0400),
     dsa_sha256_RESERVED(0x0402),
     obsolete_RESERVED(0x0404..0x0500),
     dsa_sha384_RESERVED(0x0502),
     obsolete_RESERVED(0x0504..0x0600),
     dsa_sha512_RESERVED(0x0602),
     obsolete_RESERVED(0x0604..0x06FF),
     private_use(0xFE00..0xFFFF),
     (0xFFFF)
} SignatureScheme;

struct {
    SignatureScheme supported_signature_algorithms<2..2^16-2>;
} SignatureSchemeList;
```

B.3.1.4 Extension de groupes pris en charge

```
enum {
        unallocated_RESERVED(0x0000),

/* Groupes de courbe elliptique (ECDHE) */
        obsolete_RESERVED(0x0001..0x0016),
        secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
        obsolete_RESERVED(0x001A..0x001C),
        x25519(0x001D), x448(0x001E),

/* Groupes de champs finis (DHE) */
        ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
        ffdhe6144(0x0103), ffdhe8192(0x0104),

/* Codets réservés */
        ffdhe_private_use(0x01FC..0x01FF),
        ecdhe_private_use(0xFE00..0xFEFF),
        obsolete_RESERVED(0xFF01..0xFF02),
        (0xFFFF)
```

```
} NamedGroup;

struct {
      NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Les valeurs au sein des gammes "obsolete_RESERVED" sont utilisées dans des versions antérieures de TLS et NE DOIVENT PAS être offertes ou négociées par les mises en œuvre TLS 1.3. Les courbes obsolètes ont diverses faiblesses connues ou théoriques ou ont été très peu utilisées, dans certains cas seulement du fait de problèmes involontaires de configuration de serveur. Elles ne sont plus considérées appropriées pour l'usage général et devraient être considérées comme potentiellement non sûres. L'ensemble des courbes spécifié ici est suffisant pour l'interopérabilité avec toutes les mises en œuvre TLS actuellement déployées et proprement configurées.

B.3.2 Messages de paramètres de serveur

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

  struct {
     DistinguishedName authorities<3..2^16-1>;
  } CertificateAuthoritiesExtension;

  struct {
     opaque certificate_extension_oid<1..2^8-1>;
     opaque certificate_extension_valeurs<0..2^16-1>;
  } OIDFilter;

  struct {
     OIDFilter filters<0..2^16-1>;
  } OIDFilterExtension;

  struct {} PostHandshakeAuth;

  struct {
     Extension extensions<0..2^16-1>;
  } EncryptedExtensions;

  struct {
     opaque certificate_request_context<0..2^8-1>;
     Extension extensions<2..2^16-1>;
  } CertificateRequest;
```

B.3.3 Messages d'authentification

```
enum {
     X509(0),
     OpenPGP_RESERVED(1),
     RawPublicKey(2),
     (255)
  } CertificateType;

  struct {
     select (certificate_type) {
       cas RawPublicKey:                                  /* D'après la RFC 7250 ASN.1_subjectPublicKeyInfo */
         opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;
       cas X509:
         opaque cert_data<1..2^24-1>;
     };
     Extension extensions<0..2^16-1>;
  } CertificateEntry;

  struct {
     opaque certificate_request_context<0..2^8-1>;
```

```
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;

struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;

struct {
    opaque verify_data[Hash.length];
} Finished;
```

B.3.4 Établissement de ticket

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

B.3.5 Mise à jour de clé

```
struct {} EndOfEarlyData;

enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

B.4 Suites de chiffrement

Une suite de chiffrement symétrique définit la paire d'algorithmes AEAD et de hachage à utiliser avec HKDF. Les noms de suite de chiffrement suivent la convention : CipherSuite TLS_AEAD_HASH = VALUE;

Composant	Contenu
TLS	Chaîne "TLS"
AEAD	Algorithme AEAD utilisé pour la protection d'enregistrement
HASH	Algorithme de hachage utilisé avec HKDF
VALUE	Identifiant de deux octets alloué à cette suite de chiffrement

La présente spécification définit les suites de chiffrement ci-après avec TLS 1.3.

Description	Valeur
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

Les algorithmes AEAD correspondants AEAD_AES_128_GCM, AEAD_AES_256_GCM, et AEAD_AES_128_CCM sont définis dans la [RFC5116]. AEAD_CHACHA20_POLY1305 est défini dans la [RFC8439]. AEAD_AES_128_CCM_8 est défini dans la [RFC6655]. Les algorithmes de hachage correspondants sont définis dans [SHS].

Bien que TLS 1.3 utilise le même espace de suite de chiffrement que les versions antérieures de TLS, les suites de chiffrement TLS 1.3 sont définies différemment, spécifiant seulement les chiffrements symétriques, et elles ne peuvent pas être utilisées pour TLS 1.2. De même, les suites de chiffrement pour TLS 1.2 et en dessous ne peuvent pas être utilisées

avec TLS 1.3.

De nouvelles valeurs de suite de chiffrement sont allouées par l'IANA comme décrit à la Section 11.

Appendice C. Notes de mise en œuvre

Le protocole TLS ne peut pas empêcher de nombreuses fautes de sécurité courantes. Le présent appendice fournit plusieurs recommandations pour aider à la mise en œuvre. [TLS13-TRACES] fournit des vecteurs d'essais pour les prises de contact TLS 1.3.

C.1 Génération et germe de nombre aléatoire

TLS exige un générateur de nombres pseudo aléatoires cryptographiquement sûr (CSPRNG, *cryptographically secure pseudorandom number generator*). Dans la plupart des cas, le système d'exploitation fournit une facilité appropriée telle que /dev/urandom, qui devrait être utilisée en l'absence d'autres problèmes (par exemple, de performances). Il est RECOMMANDÉ d'utiliser une mise en œuvre existante de CSPRNG de préférence à en créer une nouvelle. De nombreuses bibliothèques cryptographiques adéquates sont déjà disponibles sous des conditions de licence favorables. Si on ne les trouve pas satisfaisantes, la [RFC4086] fournit de lignes directrices pour la génération de valeurs aléatoires.

TLS utilise des valeurs aléatoires (1) dans les champs de protocole public comme les valeurs aléatoires publiques dans le ClientHello et le ServerHello et (2) pour générer le matériel de chiffrement. Avec un CSPRNG, au fonctionnement approprié cela ne présente pas de problème de sécurité, car il est impossible de déterminer l'état du CSPRNG à partir du résultat. Cependant, avec un CSPRNG cassé, il serait possible à un attaquant d'utiliser le résultat public pour déterminer l'état interne du CSPRNG et par là de prédire le matériel de chiffrement, comme documenté dans [CHECK]. Les mises en œuvre peuvent fournir une sécurité supplémentaire contre cette forme d'attaque en utilisant des CSPRNG séparés pour générer des valeurs publiques et privées.

C.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 certificat. En l'absence d'une indication spécifique provenant d'un profil d'application, les certificats devraient toujours être vérifiés pour s'assurer d'une signature appropriée par une autorité de certification (CA, *Certificate Authority*) de confiance. Le choix et l'ajout d'ancres de confiance devrait faire l'objet d'une grande attention. Les utilisateurs devraient être capables de voir les informations sur le certificat et sur l'ancre de confiance. Les applications DEVRAIENT aussi appliquer les tailles de clés minimum et maximum. Par exemple, les chemins de certification qui contiennent des clés ou signatures plus faibles que RSA à 2048 bits ou ECDSA à 224 bits ne sont pas appropriées pour des applications sûres.

C.3 Pièges de mise en œuvre

L'expérience de mise en œuvre a montré que certaines parties des spécifications TLS antérieures ne sont pas faciles à comprendre et ont été la source de problèmes d'interopérabilité et de sécurité. Beaucoup d'entre eux 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 développeurs.

Problèmes du protocole TLS :

- Traitez vous correctement les messages de prise de contact qui sont fragmentés en plusieurs enregistrements TLS (voir au paragraphe 5.1) ? Traitez vous correctement le cas d'un ClientHello qui est étalé sur plusieurs petits fragments ? Fragmentez vous les messages de prise de contact qui excèdent la taille maximum de fragment ? En particulier, les messages Certificate et CertificateRequest de la prise de contact peuvent être assez grands pour exiger la fragmentation.

- Ignorez vous le numéro de version TLS de couche d'enregistrement dans tous les enregistrements TLS non chiffrés (voir l'Appendice D)?

- Vous êtes vous assuré que toutes les prises en charge de chiffrements SSL, RC4, EXPORT, et MD5 (via l'extension "signature_algorithms") sont complètement supprimées de toutes les configurations possibles qui prennent en charge TLS 1.3 ou ultérieur, et qui tentent d'utiliser ces capacités obsolètes échouent correctement (voir l'Appendice D)?

- Traitez vous correctement les extensions TLS dans les ClientHello, incluant les extensions inconnues ?

- 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 le message entier (voir le paragraphe 4.4.2)?

- Lors du traitement du fragment en texte source produit par AEAD-Decrypt et de l'examen de la fin à la recherche du ContentType, évitez vous d'examiner le début du texte source pour le cas où l'homologue aurait envoyé un texte source mal formé tout de zéros ?

- Ignorez vous correctement les suites de chiffrement non reconnues (paragraphe 4.1.2), les extensions de Hello (paragraphe 4.2), les groupes désignés (paragraphe 4.2.7), les partages de clé (paragraphe 4.2.8), les versions prises en charge (paragraphe 4.2.1), et les algorithmes de signature (paragraphe 4.2.3) dans le ClientHello?

- Comme serveur, envoyez vous une HelloRetryRequest aux clients qui prennent en charge un groupe compatible (EC)DHE mais ne le prévoient pas dans l'extension "key_share" ? Comme client, traitez vous correctement une HelloRetryRequest provenant du serveur ?

Détails cryptographiques :
- Quelles contre mesures utilisez vous pour empêcher les attaques de synchronisation [TIMING] ?
- Quand vous utilisez l'échange de clés Diffie-Hellman, préservez vous correctement les octets zéro en tête dans la clé négociée (paragraphe 7.4.1) ?
- Votre client TLS vérifie t-il que les paramètres Diffie-Hellman envoyés par le serveur sont acceptables (paragraphe 4.2.8.1)?
- Utilisez vous un générateur de nombres aléatoires fort et, plus important, muni d'un germe approprié (Appendice C.1) lorsque vous générez des valeurs privées Diffie-Hellman, le paramètre ECDSA "k", et les autres valeurs critiques pour la sécurité ? Il est RECOMMANDÉ que les mises en œuvre utilisent "ECDSA déterministe" comme spécifié dans la [RFC6979].
- Bourrez vous de zéros les valeurs de clé publique Diffie-Hellman et les secrets partagés jusqu'à la taille de groupe (paragraphe 4.2.8.1 et paragraphe 7.4.1) ?
- Vérifiez vous les signatures après les avoir faites, pour les protéger contre les fuites de clé RSA-CRT [FW15]?

C.4 Prévention du traçage de client

Les clients NE DEVRAIENT PAS réutiliser un ticket pour plusieurs connexions. La réutilisation d'un ticket permet à des observateurs passifs de corréler les différentes connexions. Les serveurs qui produisent des tickets DEVRAIT offrir au moins autant de tickets que le nombre de connexions qu'un client peut utiliser ; par exemple, un navigateur de la Toile qui utilise HTTP/1.1 [RFC7230] peut ouvrir six connexions à un serveur. Les serveurs DEVRAIENT produire de nouveaux tickets avec chaque connexion. Cela assure que les clients sont toujours capables d'utiliser un nouveau ticket lors de la création d'une nouvelle connexion.

C.5 Fonctionnement non authentifié

Les versions antérieures de TLS offraient des suites de chiffrement explicitement non authentifiées sur la base de Diffie-Hellmann anonyme. Ces modes ont été déconseillés dans TLS 1.3. Cependant, il est toujours possible de négocier des paramètres qui ne fournissent pas d'authentification vérifiable du serveur par plusieurs méthodes, incluant :
- des clés publiques brutes [RFC7250],
- d'utiliser une clé publique contenue dans un certificat mais sans validation de la chaîne de certificats ou de ses contenus.

L'une ou l'autre de ces techniques utilisée seule est vulnérable aux attaques par interposition et donc non sûre pour utilisation générale. Cependant, il est aussi possible de lier de telles connexions à un mécanisme externe d'authentification via une validation hors bande de la clé publique du serveur, la confiance à la première utilisation, ou un mécanisme comme des liens de canaux (bien que les liens de canaux décrits dans la [RFC5929] ne soient pas définis pour TLS 1.3). Si un tel mécanisme n'est pas utilisé, la connexion n'a alors pas de protection contre une attaque par interposition active ; les applications NE DOIVENT PAS utiliser TLS de cette façon en l'absence de configuration explicite ou d'un profil spécifique d'application.

Appendice D. Rétro compatibilité

Le protocole TLS fournit un mécanisme incorporé pour la négociation de version entre points d'extrémité prenant

potentiellement en charge des versions différentes de TLS.

TLS 1.x et SSL 3.0 utilisent des messages ClientHello compatibles. Les serveurs peuvent aussi traiter les clients qui essayent d'utiliser les futures versions de TLS tant que le format de ClientHello reste compatible et qu'il y a au moins une version de protocole prise en charge par le client et le serveur.

Les versions antérieures de TLS utilisaient le numéro de version de couche enregistrement (TLSPlaintext.legacy_record_version et TLSCiphertext.legacy_record_version) pour divers objets. En ce qui concerne TLS 1.3, ce champ est déconseillé. La valeur de TLSPlaintext.legacy_record_version DOIT être ignorée par toutes les mises en œuvre. La valeur de TLSCiphertext.legacy_record_version est incluse dans les données supplémentaires pour la déprotection mais PEUT autrement être ignorée ou PEUT être validée pour correspondre à la valeur constante fixée. La négociation de version est effectuée en utilisant seulement les versions de prise de contact (ClientHello.legacy_version et ServerHello.legacy_version, ainsi que les extensions "supported_versions" ClientHello, HelloRetryRequest, et ServerHello). Afin de maximiser l'interopérabilité avec les points d'extrémité plus anciens, les mises en œuvre qui négocient l'utilisation de TLS 1.0-1.2 DEVRAIENT régler le numéro de version de couche enregistrement à la version négociée pour le ServerHello et tous les enregistrements qui en découlent.

Pour une compatibilité maximum avec les comportements non standard et les déploiements mal configurés antérieurs, toutes les mises en œuvre DEVRAIENT prendre en charge la validation des chemins de certification sur la base des attentes du présent document, même lors du traitement de prises de contact de versions TLS antérieures (paragraphe 4.4.2.2).

TLS 1.2 et les versions antérieures prenaient en charge une extension "Secret maître étendu" [RFC7627] qui condensait de larges parties de la transcription de prise de contact dans le secret maître. Parce que TLS 1.3 hache toujours la transcription jusqu'au Finished du serveur, les mises en œuvre qui prennent en charge TLS 1.3 et les versions antérieures DEVRAIENT indiquer l'utilisation de l'extension Secret maître étendu dans leurs API chaque fois que TLS 1.3 est utilisé.

D.1 Négociation avec un ancien serveur

Un client TLS 1.3 qui souhaite négocier avec des serveurs qui ne prennent pas en charge TLS 1.3 va envoyer un ClientHello TLS 1.3 normal contenant 0x0303 (TLS 1.2) dans ClientHello.legacy_version mais avec la ou les versions correctes dans l'extension "supported_versions". Si le serveur ne prend pas en charge TLS 1.3, il va répondre avec un ServerHello contenant un numéro de version plus ancien. Si le client accepte d'utiliser cette version, la négociation va se poursuivre comme approprié pour le protocole négocié. Un client qui utilise un ticket pour la reprise DEVRAIT initier la connexion en utilisant la version qui a été négociée précédemment.

Noter que les données de 0-RTT ne sont pas compatibles avec les anciens serveurs et NE DEVRAIENT PAS être envoyées si on ne sait pas si le serveur accepte TLS 1.3 (voir l'Appendice D.3).

Si la version choisie par le serveur n'est pas acceptée par le client (ou n'est pas acceptable) le client DOIT interrompre la prise de contact avec une alerte "protocol_version".

Certaines mises en œuvre de serveur traditionnelles sont connues pour ne pas mettre en œuvre correctement la spécification TLS et peuvent interrompre les connexions quand elles rencontrent des extensions ou versions de TLS qu'elles ne connaissent pas. L'interopérabilité avec des serveurs défectueux est un sujet complexe qui sort du domaine d'application du présent document. Plusieurs tentatives de connexion peuvent être nécessaires afin de négocier une connexion rétro compatible ; cependant, cette pratique est vulnérable aux attaques en dégradation et N'EST PAS RECOMMANDÉE.

D.2 Négociation avec un ancien client

Un serveur TLS peut aussi recevoir un ClientHello qui indique un numéro de version inférieur à la plus haute version prise en charge. Si l'extension "supported_versions" est présente, le serveur DOIT négocier en utilisant cette extension comme décrit au paragraphe 4.2.1. Si l'extension "supported_versions" n'est pas présente, le serveur DOIT négocier le minimum de ClientHello.legacy_version et de TLS 1.2. Par exemple, si le serveur accepte TLS 1.0, 1.1, et 1.2, et si legacy_version est TLS 1.0, le serveur va poursuivre avec un ServerHello TLS 1.0. Si l'extension "supported_versions" est absente et si le serveur accepte seulement les versions supérieures à ClientHello.legacy_version, le serveur DOIT interrompre la prise de contact avec une alerte "protocol_version".

Noter que les versions antérieures de TLS ne spécifiaient pas clairement la valeur de numéro de version de couche enregistrement dans tous les cas (TLSPlaintext.legacy_record_version). Les serveurs vont recevoir diverses versions TLS 1.x dans ce champ, mais sa valeur DOIT toujours être ignorée.

D.3 Rétro compatibilité avec 0-RTT

Les données 0-RTT ne sont pas compatibles avec les serveurs plus anciens. Un serveur plus ancien va répondre au ClientHello avec un ancien ServerHello, mais il ne va pas correctement sauter les données 0-RTT et va échouer à achever la prise de contact. Cela peut causer des problèmes lorsque un client tente d'utiliser 0-RTT, en particulier avec des déploiements multi serveurs. Par exemple, un déploiement pourrait développer TLS 1.3 graduellement avec certains serveurs qui mettent en œuvre TLS 1.3 et certains autres qui mettent en œuvre TLS 1.2, ou un déploiement de TLS 1.3 pourrait être dégradé en TLS 1.2.

Un client qui tente d'envoyer des données 0-RTT DOIT faire échouer une connexion si il reçoit un ServerHello avec TLS 1.2 ou plus ancien. Il peut réessayer la connexion avec 0-RTT désactivé. Pour éviter une attaque en dégradation, le client NE DEVRAIT PAS désactiver TLS 1.3, seulement 0-RTT.

Pour éviter cette condition d'erreur, les déploiements multi serveurs DEVRAIENT s'assurer d'un déploiement uniforme et stable de TLS 1.3 sans 0-RTT avant d'activer 0-RTT.

D.4 Mode de compatibilité de boîtier de médiation

Les mesures de champs [Ben17a], [Ben17b], [Res17a], [Res17b] ont trouvé un nombre significatif de mauvais comportements de boîtiers de médiation lorsque une paire client/serveur TLS négocie TLS 1.3. Les mises en œuvre peuvent augmenter les chances d'établir des connexions à travers ces boîtiers de médiation en faisant ressembler la prise de contact TLS 1.3 à une prise de contact TLS 1.2 :

- le client fournit toujours un identifiant de session non vide dans le ClientHello, comme décrit dans la section legacy_session_id du paragraphe 4.1.2.
- Si il n'y a pas d'offre de données précoces, le client envoie un enregistrement change_cipher_spec factice (voir le troisième paragraphe de la Section 5) immédiatement avant son second vol. Cela peut être soit avant son second ClientHello, soit avant son envoi de prise de contact chiffré. Si il offre des données précoces, l'enregistrement est placé immédiatement après le premier ClientHello.
- Le serveur envoie un enregistrement change_cipher_spec factice immédiatement après son premier message de prise de contact. Ce peut être après un ServerHello ou une HelloRetryRequest.

Mis ensemble, ces changements font ressembler la prise de contact TLS 1.3 à la reprise de session TLS 1.2, qui améliore les chances de réussite de la connexion à travers un boîtier de médiation. Ce "mode de compatibilité" est partiellement négocié : le client peut opter pour la fourniture d'un identifiant de session ou non, et le serveur doit y faire écho. L'un ou l'autre côté peut envoyer une change_cipher_spec à tout moment durant la prise de contact, car elles doivent être ignorées par l'homologue, mais si le client envoie un identifiant de session non vide, le serveur DOIT envoyer le change_cipher_spec comme décrit dans cet appendice.

D.5 Restrictions de sécurité relatives à la rétro compatibilité

Les mises en œuvre qui négocient l'utilisation de plus anciennes versions de TLS DEVRAIENT préférer le secret vers l'avant et les suites de chiffrement AEAD, lorsque disponibles.

La sécurité des suites de chiffrement RC4 est considérée comme insuffisante pour les raisons citées dans la [RFC7465]. Les mises en œuvre NE DOIVENT PAS offrir ou négocier les suites de chiffrement RC4 pour toute version de TLS pour quelque raison que ce soit.

Les anciennes versions de TLS permettaient l'utilisation de chiffrements très faibles. Les chiffrements de moins de 112 bits NE DOIVENT PAS être offerts ou négociés pour toute version de TLS quelle qu'en soit la raison.

La sécurité de SSL 3.0 [RFC6101] est considérée comme insuffisante pour les raisons mentionnées dans la [RFC7568], et il NE DOIT être négocié pour aucune raison.

La sécurité de SSL 2.0 [SSL2] est considérée comme insuffisante pour les raisons mentionnées dans la [RFC6176], et il NE DOIT être négocié pour aucune raison.

Les mises en œuvre NE DOIVENT PAS envoyer un CLIENT-HELLO compatible SSL version 2.0. Les mises en œuvre NE DOIVENT PAS négocier TLS 1.3 ou ultérieur en utilisant un CLIENT-HELLO compatible SSL version 2.0. Il n'est pas RECOMMANDÉ aux mises en œuvre d'accepter un CLIENT-HELLO compatible SSL version 2.0 afin de négocier de plus anciennes versions de TLS.

Les mises en œuvre NE DOIVENT PAS envoyer un ClientHello.legacy_version ou ServerHello.legacy_version réglé à

0x0300 ou moins. Tout point d'extrémité qui reçoit un message Hello avec ClientHello.legacy_version ou ServerHello.legacy_version réglé à 0x0300 DOIT interrompre la prise de contact avec une alerte "protocol_version".

Les mises en œuvre NE DOIVENT PAS envoyer d'enregistrements avec une version de moins de 0x0300. Les mises en œuvre NE DEVRAIENT PAS accepter d'enregistrements avec une version de moins de 0x0300 (mais peuvent le faire par inadvertance si le numéro de version d'enregistrement est ignoré complètement).

Les mises en œuvre NE DOIVENT PAS utiliser l'extension HMAC tronqué, définie à la Section 7 de la [RFC6066], car elle n'est pas applicable aux algorithmes AEAD et il a été montré qu'elle n'est pas sûre dans certains scénarios.

Appendice E Vue d'ensemble des propriétés de sécurité

Il n'appartient pas au présent document de faire une analyse complète de la sécurité de TLS. Cet appendice donne une description informelle des propriétés désirées ainsi que des références à des travaux plus détaillés dans la littérature de la recherche qui fournissent des définitions plus formelles.

On traite les propriétés de la prise de contact séparément de celle de la couche enregistrement.

E.1 Prise de contact

La prise de contact TLS est un protocole d'échange de clé authentifié (AKE, *Authenticated Key Exchange*) qui est destiné à fournir une fonction d'authentification unidirectionnelle (serveur seulement) et mutuelle (client et serveur). À l'achèvement de la prise de contact, chaque côté produit sa vue des valeurs suivantes :

- un ensemble de "clés de session" (les divers secrets déduits du secret maître) à partir duquel peut être déduit un ensemble de clés de travail ;
- un ensemble de paramètres de chiffrement (algorithmes, etc.) ;
- les identités des parties communicantes.

On suppose que l'attaquant est un attaquant actif du réseau, ce qui signifie qu'il a le contrôle complet du réseau utilisé pour communiquer entre les parties [RFC3552]. Même dans ces conditions, la prise de contact devrait fournir les propriétés mentionnées ci-dessous. Noter que ces propriétés ne sont pas nécessairement indépendantes, mais reflètent les besoins des utilisateurs du protocole.

Établir les mêmes clés de session : La prise de contact doit produire en sortie le même ensemble de clés de session sur les deux côtés de la prise de contact, pourvu qu'elle s'achève avec succès sur chaque point d'extrémité (voir [CK01], définition 1, partie 1).

Secret des clé s de session : Les clés de session partagées ne devraient être connues que des parties communicantes et non de l'attaquant (voir [CK01], définition 1, partie 2). Noter que dans une connexion authentifiée unilatéralement, l'attaquant peut établir ses propres clés de session avec le serveur, mais ces clés de session sont distinctes de celles établies par le client.

Authentification des homologues : La vue du client de l'identité de l'homologue devrait refléter l'identité du serveur. Si le client est authentifié, la vue du serveur de l'identité de l'homologue devrait correspondre à l'identité du client.

Unicité des clés de session : Deux prises de contact distinctes devraient produire des clés de session distinctes, sans relation entre elles. Les clés de session individuelles produites par une prise de contact devraient aussi être distinctes et indépendantes.

Protection contre la dégradation : Les paramètres cryptographiques devraient être les mêmes sur les deux côtés et devraient être les mêmes que si les homologues avaient communiqué en l'absence d'une attaque (voir [BBFGKZ16], définitions 8 et 9).

Secret vers l'avant par rapport aux clés à long terme : Si le matériel de chiffrement à long terme (dans ce cas, les clé de signature dans les modes d'authentification fondés sur le certificat ou la PSK externe/de reprise dans PSK avec modes (EC)DHE) est compromis après l'achèvement de la prise de contact, cela ne compromet pas la sécurité de la clé de session (voir [DOW92]), pour autant que la clé de session elle-même ait été écrasée. La propriété de secret vers l'avant n'est pas satisfaite lorsque la PSK est utilisée dans le PskKeyExchangeMode "psk_ke".

Résistance à l'appropriation d'une clé compromise (KCI, *Key Compromise Impersonation*) : Dans une connexion

mutuellement authentifiée avec des certificats, la compromission du secret à long terme d'un acteur ne devrait pas casser l'authentification de l'homologue de l'acteur dans cette connexion (voir [HGFS15]). Par exemple, si la clé de signature d'un client est compromise, il ne devrait pas être possible de simuler des serveurs arbitraires pour ce client dans les prises de contact suivantes.

Protection des identités de point d'extrémité : L'identité du serveur (certificat) devrait être protégée contre les attaques passives. L'identité du client devrait être protégée contre les attaques passives et actives.

De façon informelle, les modes fondés sur la signature de TLS 1.3 assurent l'établissement d'une clé unique, secrète, partagée établie par un échange de clé (EC)DHE et authentifié par la signature du serveur sur la transcription de la prise de contact, ainsi que liée à l'identité du serveur par un MAC. Si le client est authentifié par un certificat, il signe aussi la transcription de prise de contact et fournit un MAC lié aux deux identités. [SIGMA] décrit la conception et l'analyse de ce type de protocole d'échange ce clé. Si des clés fraîches (EC)DHE sont utilisées pour chaque connexion, les clés produites sont secrètes vers l'avant.

La PSK externe et la PSK de reprise s'amorcent à partir d'un secret partagé à long terme en un unique ensemble par connexion de clés de session à court terme. Ce secret peut avoir été établi dans une prise de contact précédente. Si la PSK avec établissement de clé (EC)DHE est utilisée, ces clé de session seront aussi secrètes vers l'avant. La PSK de reprise a été conçue de façon à ce que le secret maître de reprise calculé par la connexion N et nécessaire pour former la connexion N+1 soit séparé des clés de trafic utilisées par la connexion N, fournissant ainsi le secret vers l'avant entre les connexions. De plus, si plusieurs tickets sont établis sur la même connexion, elles sont associées à des clés différentes, de sorte que la compromission de la PSK associée à un ticket ne conduit pas à la compromission des connexions établies avec les PSK associées à d'autres tickets. Cette propriété est très intéressante si les tickets sont mémorisés dans une base de données (et peuvent ainsi être supprimés) plutôt que si ils sont auto chiffrés.

La valeur de liant de PSK forme un lien entre une PSK et la prise de contact en cours, ainsi que entre la session où la PSK a été établie et la session en cours. Ce lien inclut transitivement la transcription originale de la prise de contact, parce que cette transcription est résumée dans les valeurs qui produisent le secret maître de reprise. Cela exige que le KDF utilisé pour produire le secret maître de reprise et le MAC utilisé pour calculer le liant soient tous deux résistants à la collision. Voir les détails sur ce point au paragraphe E.1.1. Noter que le liant ne couvre pas les valeurs de liant provenant des autres PSK, bien qu'elles soient incluses dans le MAC Finished.

TLS ne permet actuellement pas que le serveur envoie un message certificate_request dans une prise de contact non fondée sur le certificat (par exemple, PSK). Si cette restriction devait être supprimée à l'avenir, la signature du client ne couvrirait pas directement le certificat du serveur. Cependant, si la PSK est établie par un NewSessionTicket, la signature du client couvrirait transitivement le certificat du serveur à travers le liant de PSK. [PSK-FINISHED] décrit une attaque concrète sur des constructions qui ne lient pas le certificat du serveur (voir aussi [Kraw16]). Il n'est pas sûr d'utiliser une authentification de client fondée sur le certificat lorsque le client pourrait partager potentiellement la même paire de PSK/identifiant de clé dans deux points d'extrémité différents. Les mises en œuvre NE DOIVENT PAS combiner des PSK externes avec l'authentification fondée sur le certificat du client ou du serveur sauf si c'est négocié par une extension.

Si un exporteur est utilisé, il produit alors des valeurs qui sont uniques et secrètes (parce que générées à partir d'une clé de session unique). Les exporteurs calculés avec des étiquettes et contextes différents sont computationnellement indépendants, de sorte qu'il est impossible de calculer l'un à partir de l'autre ou le secret de session à partir de la valeur exportée. Noter que les exporteurs peuvent produire des valeurs de longueur arbitraire ; si les exporteurs doivent être utilisés comme liants de canal, la valeur exportée DOIT être assez grande pour assurer la résistance à la collision. Les exporteurs fournis dans TLS 1.3 sont déduits des mêmes contextes de prise de contact que respectivement les clés de trafic précoce et les clés de trafic d'application, et ont donc des propriétés de sécurité similaires. Noter qu'ils n'incluent pas le certificat de client ; de futures applications qui souhaiteraient se lier au certificat de client pourraient avoir besoin de définir un nouvel exporteur qui inclurait toute la transcription de prise de contact.

Pour tous les modes de prise de contact, le MAC Finished (et, lorsque présente, la signature) empêche les attaques en déclassement. De plus, l'utilisation de certains octets dans les noms occasionnels aléatoires comme décrit au paragraphe 4.1.3 permet la détection du déclassement sur des versions antérieures de TLS. Voir [BBFGKZ16] pour des détails sur le déclassement de TLS 1.3.

Aussitôt que le client et le serveur ont échangé assez d'informations pour établir des clés partagées, le reste de la prise de contact est chiffré, assurant ainsi la protection contre les attaques passives, même si la clé partagée calculée n'est pas authentifiée. Parce que le serveur s'authentifie avant le client, le client peut s'assurer que si il s'authentifie auprès du serveur, il révèle seulement son identité à un serveur authentifié. Noter que les mises en œuvre doivent utiliser le mécanisme de bourrage d'enregistrement fourni durant la prise de contact pour éviter de laisser fuir des informations sur les identités à cause de la longueur. Les identités de PSK proposées du client ne sont pas chiffrées, pas plus que ne l'est celle que le serveur choisit.

E.1.1 Déduction de clé et HKDF

Dans TLS 1.3 la déduction de clé utilise HKDF comme défini dans la [RFC5869] et ses deux composants, HKDF-Extract et HKDF-Expand. On trouvera des explications complètes sur les raisons de la construction HKDF dans [Kraw10] et les raison de la façon de l'utiliser dans TLS 1.3 dans [KW16]. Tout au long du présent document, chaque application de HKDF-Extract est suivie d'une ou plusieurs invocations de HKDF-Expand. Cet ordre devrait toujours être suivi (y compris dans les futures révisions du présent document) ; en particulier, on NE DEVRAIT PAS utiliser un résultat de HKDF-Extract comme entrée à d'autres applications de HKDF-Extract sans un HKDF-Expand entre eux. Plusieurs applications de HKDF-Expand à certaines des mêmes entrées sont permises pour autant qu'elles se différencient via la clé et/ou les étiquettes.

Noter que HKDF-Expand met en œuvre une fonction pseudo aléatoire (PRF) avec des entrées et des résultats de longueur variable. Dans certaines utilisations de HKDF de ce document (par exemple, pour générer les exporteurs et le resumption_master_secret) il est nécessaire que l'application de HKDF-Expand soit résistante à la collision ; à savoir qu'il devrait être impossible de trouver deux entrées différentes à HKDF-Expand qui donnent la même valeur. Cela exige que la fonction de hachage sous-jacente soit résistante à la collision et que la longueur du résultat de HKDF-Expand fasse au moins 256 bits (ou autant que nécessaire pour que la fonction de hachage empêche de trouver des collisions).

E.1.2 Authentification du client

Un client qui a envoyé des données d'authentification à un serveur, soit durant la prise de contact, soit dans une authentification post prise de contact, ne peut pas être sûr que le serveur considère après coup si le client est authentifié ou non. Si le client a besoin de déterminer si le serveur considère que la connexion est authentifiée unilatéralement ou mutuellement, ceci doit être provisionné par la couche application. Voir les détails dans [CHHSV17]. De plus, l'analyse de l'authentification post prise de contact d'après [Kraw16] montre que le client identifié par le certificat envoyé dans la phase post prise de contact possède les clés de trafic. Cette partie est donc le client qui a participé à la prise de contact originale ou un à qui le client original a délégué la clé de trafic (en supposant que la clé de trafic n'a pas été compromise).

E.1.3 0-RTT

Le mode de fonctionnement 0-RTT fournit généralement des propriétés de sécurité similaires à celles des données 1-RTT, avec deux exceptions qui sont que les clés de chiffrement 0-RTT ne fournissent pas un vrai secret vers l'avant et que le serveur n'est pas capable de garantir l'unicité de la prise de contact (non répétabilité) sans garder de potentiellement grosses quantités d'état indues. Voir à la Section 8 le mécanisme pour limiter l'exposition à la répétition.

E.1.4 Indépendance d'exporteur

Le exporter_master_secret et le early_exporter_master_secret sont déduits comme étant indépendants des clés de trafic et donc ne représentent pas une menace pour la sécurité du trafic chiffré avec ces clés. Cependant, comme ces secrets peuvent être utilisés pour calculer toute valeur d'exporteur, elles DEVRAIENT être supprimées aussitôt que possible. Si l'ensemble total des étiquettes d'exporteur est connu, les mises en œuvre DEVRAIENT alors pré calculer l'étape interne Derive-Secret du calcul de l'exporteur pour toutes ces étiquettes, puis éliminer le [early_]exporter_master_secret, suivi par chaque valeur interne aussitôt qu'il est certain qu'on en aura plus besoin.

E.1.5 Sécurité compromise a posteriori

TLS ne fournit pas de sécurité pour les prises de contact qui ont lieu après que le secret à long terme de l'homologue (clé de signature ou PSK externe) est compromise. Il ne fournit donc pas de sécurité post compromission [CCG16], parfois aussi appelée secret vers l'arrière ou secret futur. Ceci est à l'opposé de la résistance de KCI, qui décrit les garanties de sécurité qu'a une partie après que son propre secret à long terme a été compromis.

E.1.6 Références externes

Le lecteur devrait se reporter aux références suivantes pour l'analyse de la prise de contact TLS : [DFGS15], [CHSV16], [DFGS16], [KW16], [Kraw16], [FGSW16], [LXZFH16], [FG17], et [BBK17].

E.2 Couche d'enregistrement

La couche enregistrement dépend de la production par la prise de contact d'un secret du trafic fort qui puisse être utilisé pour déduire des clés de chiffrement bidirectionnelles et des noms occasionnels. En supposant que cela est vrai, et que les

clés ne sont pas utilisées pour plus de données qu'indiqué au paragraphe 5.5, la couche enregistrement devrait fournir les garanties suivantes :

Confidentialité : un attaquant ne devrait pas être capable de déterminer le contenu du texte source d'un enregistrement.

Intégrité : un attaquant ne devrait pas être capable de fabriquer un nouvel enregistrement différent d'un existant qui serait accepté par le receveur.

Protection de l'ordre/non répétabilité : un attaquant ne devrait pas être capable de faire que le receveur accepte un enregistrement qu'il a déjà accepté ou faire que le receveur accepte l'enregistrement N+1 sans avoir d'abord traité l'enregistrement N.

Dissimulation de la longueur : dans un enregistrement d'une certaine longueur externe, l'attaquant ne devrait pas être capable de déterminer ce qui dans l'enregistrement est du contenu et ce qui est du bourrage.

Secret vers l'avant après changement de clé : Si le mécanisme de mise à jour de clé de trafic décrit au paragraphe 4.6.3 a été utilisé et si la clé de génération précédente est supprimée, un attaquant qui compromet le point d'extrémité ne devrait pas être capable de décrypter le trafic chiffré avec la vieille clé.

De façon informelle, TLS 1.3 fournit ces propriétés en protégeant avec AEAD le texte source avec une clé forte. Le format de chiffrement AEAD [RFC5116] fournit la protection de la confidentialité et de l'intégrité pour les données. La non répétabilité est fournie en utilisant un nom occasionnel séparé pour chaque enregistrement, le nom occasionnel étant déduit du numéro de séquence de l'enregistrement (paragraphe 5.3) avec chaque côté qui tient ses numéros de séquence de façon indépendante ; donc, les enregistrements qui ne sont pas livrés dans l'ordre résultent en un échec de déprotection AEAD. Afin d'empêcher la cryptanalyse de masse lorsque le même texte source est chiffré de façon répétée par différents utilisateurs sous la même clé (comme c'est très souvent le cas pour HTTP) le nom occasionnel est formé en mêlant le numéro de séquence avec une valeur d'initialisation secrète par connexion déduite avec les clé de trafic. Voir dans [BT16] l'analyse de cette construction.

La technique de changement de clé dans TLS 1.3 (paragraphe 7.2) suit la construction du générateur en série discuté dans [REKEY], qui montre que le changement de clés peut permettre que les clés soient utilisées pour un plus grand nombre de chiffrements que sans le changement de clés. Cela s'appuie sur la sécurité de la fonction HKDF-Expand-Label comme fonction pseudo aléatoire (PRF). De plus, tant que cette fonction est vraiment unidirectionnelle, il n'est pas possible de calculer les clés de trafic avant un changement de clé (secret vers l'avant).

TLS n'assure pas la sécurité des données qui sont communiquées sur une connexion après la compromission du secret du trafic de cette connexion. C'est-à-dire, TLS ne fournit pas de sécurité post compromission, de secret vers le futur, de secret vers le passé par rapport au secret du trafic. Bien sûr un attaquant qui découvre un secret du trafic peut calculer tous les futurs secrets du trafic sur cette connexion. Les systèmes qui veulent de telles garanties doivent faire une prise de contact fraîche et établir une nouvelle connexion avec un échange (EC)DHE.

E.2.1 Références externes

Le lecteur devrait se reporter aux références suivantes pour l'analyse de la couche d'enregistrement TLS : [BMMRT15], [BT16], [BDF], [BBK17], et [PS18].

E.3 Analyse de trafic

TLS est susceptible de subir diverses attaque d'analyse de trafic sur la base de l'observation de la longueur et de la durée des paquets chiffrés [CLINIC], [HCJC16]. C'est particulièrement facile lorsque il y a un petit ensemble de messages possibles à distinguer, comme pour un serveur vidéo hébergeant un corpus fixe de contenu, mais cela fournit encore des informations utilisables même dans des scénarios plus compliqués.

TLS ne fournit pas de défenses spécifiques contre cette forme d'attaque mais inclut un mécanisme de bourrage à l'usage des applications : le texte source protégé par la fonction AEAD consiste en un contenu plus un bourrage de longueur variable, qui permet à l'application de produire des enregistrements chiffrés de longueur arbitraire ainsi qu'un bourrage couvrant seulement le trafic pour dissimuler les différences entre les périodes de transmission et les périodes de silence. Comme le bourrage est chiffré avec le contenu réel, un attaquant ne peut pas déterminer directement la longueur du bourrage mais peut être capable de le mesurer indirectement en utilisant le rythme des canaux exposés durant le traitement de l'enregistrement (c'est-à-dire, en regardant combien de temps cela prend pour traiter un enregistrement ou en observant les enregistrements un à un pour voir lesquels provoquent une réponse de la part du serveur). En général, on ne sait pas comment supprimer tous ces canaux parce qu'une fonction constante de suppression de bourrage va probablement fournir le

contenu selon des fonctions qui dépendent des données. Au minimum, un serveur ou un client complètement constant va exiger une étroite coopération avec la mise en œuvre de protocole de couche d'application, incluant de rendre ce protocole de couche supérieure constant dans le temps.

Note : Des défenses robustes contre l'analyse de trafic vont probablement conduire à des performances inférieures dues aux délais de transmission des paquets et de l'augmentation du volume de trafic.

E.4 Attaques côté canal

En général, TLS n'a pas de défenses spécifiques contre les attaques de canal latéral (c'est-à-dire, celles qui attaquent les communications via des canaux secondaires comme de synchronisation) laissant cela à la mise en œuvre des primitives cryptographiques pertinentes. Cependant, certaines caractéristiques de TLS sont conçues pour rendre plus facile d'écrire du code résistant au canal latéral :

- À la différence des versions antérieures de TLS qui utilisaient une structure composite de MAC puis chiffrement, TLS 1.3 utilise seulement des algorithmes AEAD, permettant aux mises en œuvre d'utiliser les applications auto contenues de ces primitives en temps constant.
- TLS utilise une alerte uniforme "mauvais_mac_d'enregistrement" pour toutes les erreurs de déchiffrement, qui est destinée à empêcher un attaquant d'obtenir une vue précise des portions du message. Une résistance supplémentaire est fournie par la clôture de la connexion sur de telles erreurs ; une nouvelle connexion aura du matériel de chiffrement différent, empêchant les attaques contre les primitives cryptographiques qui exigent plusieurs essais.

Les fuites d'information par les canaux latéraux peuvent se produire à des couches supérieures à TLS, dans les protocoles d'application et à l'application qui les utilise. La résistance aux attaques de canal latéral dépend des applications et des protocoles d'application qui doivent s'assurer séparément que des informations confidentielles ne sont pas divulguées par inadvertance.

E.5 Attaques en répétition sur 0-RTT

La répétition de données 0-RTT présente un certain nombre de menaces pour la sécurité pour les applications qui utilisent TLS, sauf si ces applications sont spécifiquement agencées pour être sûres en cas de répétition (au minimum, cela signifie qu'elles soient idempotentes, mais dans de nombreux cas, cela peut aussi exiger d'autres conditions plus fortes, comme un temps de réponse constant). Les attaques potentielles incluent :

- La duplication des actions qui causent des effets collatéraux (par exemple, l'achat d'un objet ou un transfert monétaire) à dupliquer, causant ainsi un dommage au site ou à l'usager.
- L'attaquant peut mémoriser et répéter les messages 0-RTT afin de les réorganiser par rapport à d'autres messages (par exemple, déplacer une suppression après une création).
- Exploitation du comportement de rythme de l'antémémoire pour découvrir le contenu des messages 0-RTT en répétant un message 0-RTT à un nœud d'antémémoire différent et en utilisant ensuite une connexion séparée pour mesurer la latence de demande, pour voir si les deux demandes s'adressent à la même ressource.

Si les données peuvent être répétées un grand nombre de fois, des attaques supplémentaires deviennent possibles, comme de faire des mesures répétées de la vitesse des opérations cryptographiques. De plus, elles peuvent être capables de surcharger les systèmes de limitation de débit. Pour une description plus complète de ces attaques, voir [Mac17].

En fin de compte, les serveurs ont la responsabilité de se protéger contre les attaques qui emploient la répétition de données 0-RTT. Les mécanismes décrits à la Section 8 sont destinés à empêcher la répétition à la couche TLS mais ne fournissent pas une complète protection contre la réception de multiples copies des données du client. TLS 1.3 revient à la prise de contact 1-RTT lorsque le serveur n'a pas d'informations sur le client, par exemple, parce qu'il est dans une grappe différente qui ne partage pas l'état ou parce que le ticket a été supprimé comme décrit au paragraphe 8.1. Si la couche protocole d'application retransmet les données dans ce réglage, il est alors possible à un attaquant d'introduire la duplication de message en envoyant le ClientHello à la fois à la grappe d'origine (qui traite les données immédiatement) et à une autre grappe qui va revenir à 1-RTT et traiter les données à la répétition de la couche application. La portée de cette attaque est limitée par la volonté du client de réessayer les transactions et ne permet donc seulement qu'une quantité limitée de duplication, chaque copie apparaissant comme une nouvelle connexion au serveur.

Si il est mis en œuvre correctement, le mécanisme décrit aux paragraphes 8.1 et 8.2 empêche qu'un ClientHello répété et ses données 0-RTT associées soient acceptés plusieurs fois par toute grappe qui a un état cohérent ; pour les serveurs qui limitent l'utilisation de 0-RTT à une grappe pour un seul ticket, et un ClientHello et ses données 0-TTT associées ne seront acceptés qu'une seule fois. Cependant, si l'état n'est pas entièrement cohérent, un attaquant peut alors être capable d'avoir plusieurs copies des données acceptées durant la fenêtre de duplication. Parce que les clients ne connaissent pas les détails exacts du comportement du serveur, ils NE DOIVENT PAS envoyer de messages dans les données précoces dont ils ne

sont pas sûrs qu'ils ne soient pas répétés et qu'ils ne voudraient pas les réessayer à travers de multiples connexions 1-RTT.

Les protocoles d'application NE DOIVENT PAS utiliser les données 0-RTT dans un profil définissant leur usage. Ce profil doit identifier quels messages ou interactions sont d'utilisation sûre avec 0-RTT et comment traiter la situation quand le serveur rejette 0-RTT et revient à 1-RTT.

De plus, pour éviter de mauvaises utilisations accidentelles, les mises en œuvre de TLS NE DOIVENT PAS activer 0-RTT (en envoi ou en l'acceptant) si ce n'est pas spécifiquement demandé par l'application et NE DOIVENT PAS automatiquement renvoyer les données 0-RTT si c'est rejeté par le serveur sauf instructions de l'application. Les applications côté serveur peuvent souhaiter mettre en œuvre un traitement spécial pour les données 0-RTT pour certaines sortes de trafic d'application (par exemple, interrompre la connexion, demander que les données soient renvoyées à la couche application, ou retarder le traitement jusqu'à l'achèvement de la prise de contact). Afin de permettre aux applications de mettre en œuvre de type de traitement, les mises en œuvre de TLS DOIVENT fournir un moyen pour que l'application détermine si la prise de contact est achevée.

E.5.1 Répétition et exporteurs

Les répétitions du ClientHello produisent le même exporteur précoce, exigeant donc une attention supplémentaire de la part des applications qui utilisent ces exporteurs. En particulier, si ces exporteurs sont utilisés comme un liant de canal d'authentification (par exemple, en signant le résultat de l'exporteur) un attaquant qui compromet la PSK peut transplanter les authentifiants entre les connexions sans compromettre la clé d'authentification.

De plus, l'exporteur précoce NE DEVRAIT PAS être utilisé pour générer des clés de chiffrement de serveur à client parce que cela permettrait la réutilisation de ces clés. Ceci est parallèle à l'utilisation des clés de trafic d'application précoce seulement dans la direction client à serveur.

E.6 Exposition de l'identité de PSK

Parce que les mises en œuvre répondent à un liant de PSK invalide en interrompant la prise de contact, il est possible à un attaquant de vérifier si une certaine identité de PSK est valide. Précisément, si un serveur accepte à la fois des prises de contact et des prises de contact fondées sur le certificat, une identité de PSK valide va résulter en un échec de prise de contact, tandis qu'une identité invalide va juste être sautée et résulter en une prise de contact avec certificat réussie. Les serveurs qui prennent seulement en charge les prises de contact avec PSK peuvent être capables de résister à cette forme d'attaque en traitant les cas où il n'y a pas d'identité de PSK valide et où il y a une identité mais qui a un liant identiquement invalide.

E.7 Partage des PSK

TLS 1.3 a une approche prudente des PSK en les liant à une KDF spécifique. À l'opposé, TLS 1.2 permet aux PSK d'être utilisées avec toute fonction de hachage et la PRF TLS 1.2. Donc, toute PSK qui est utilisée avec TLS 1.2 et TLS 1.3 doit être utilisée avec seulement un hachage dans TLS 1.3, ce qui est sous optimal si l'utilisateur veut provisionner une seule PSK. Les constructions sont différentes dans TLS 1.2 et TLS 1.3, bien que toutes deux se fondent sur HMAC. Bien qu'il n'y ait pas de moyen connu pour que la même PSK puisse produire un résultat en rapport dans les deux versions, une analyse limitée a seulement été faite. Les mises en œuvre peuvent s'assurer en toute sécurité contre les résultats en rapports avec les protocoles croisés en ne réutilisant pas les PSK entre TLS 1.3 et TLS 1.2.

E.8. Attaques sur RSA statique

Bien que TLS 1.3 n'utilise pas le transport de clé RSA et ne soit pas directement susceptible d'une attaque de type Bleichenbacher [Blei98], si les serveur TLS 1.3 prennent en charge aussi RSA statique dans le contexte de versions antérieures de TLS, il est alors possible de se faire passer pour le serveur pour les connexions TLS 1.3 [JSS15]. Les mises en œuvre de TLS 1.3 peuvent empêcher cette attaque en désactivant la prise en charge de RSA statique sur toutes les versions de TLS. En principe, les mises en œuvre peuvent aussi être capables de séparer les certificats avec des bits keyUsage différents pour le déchiffrement de RSA statique et les signatures RSA, mais cette technique s'appuie sur des clients qui refusent d'accepter les signatures qui utilisent des clés dans des certificats qui n'ont pas le bit digitalSignature établi, et de nombreux clients n'appliquent pas cette restriction.

Contributeurs

Martin Abadi, University of California, Santa Cruz, abadi@cs.ucsc.edu

Christopher Allen (co-éditeur de TLS 1.0), Alacrity Ventures, ChristopherA@AlacrityManagement.com
Richard Barnes, Cisco, rlb@ipv.sx
Steven M. Bellovin, Columbia University, smb@cs.columbia.edu
David Benjamin, Google, davidben@google.com
Benjamin Beurdouche, INRIA & Microsoft Research, benjamin.beurdouche@ens.fr
Karthikeyan Bhargavan, (éditeur de la [RFC7627]) INRIA, karthikeyan.bhargavan@inria.fr
Simon Blake-Wilson (co-auteur de la [RFC4492]) BCI, sblakewilson@bcisse.com
Nelson Bolyard (co-auteur de la [RFC4492]) Sun Microsystems, Inc., nelson@bolyard.com
Ran Canetti, IBM, canetti@watson.ibm.com
Matt Caswell, OpenSSL, matt@openssl.org
Stephen Checkoway, University of Illinois at Chicago, sfc@uic.edu
Pete Chown, Skygate Technology Ltd, pc@skygate.co.uk
Katriel Cohn-Gordon, University of Oxford, me@katriel.co.uk
Cas Cremers, University of Oxford, cas.cremers@cs.ox.ac.uk
Antoine Delignat-Lavaud, (co-auteur de la [RFC7627]) INRIA, antdl@microsoft.com
Tim Dierks (co-auteur de TLS 1.0, co-éditeur de TLS 1.1 et 1.2), Indépendant, tim@dierks.org
Roelof DuToit, Symantec Corporation, roelof_dutoit@symantec.com
Taher Elgamal, Securify, taher@securify.com
Pasi Eronen, Nokia, pasi.eronen@nokia.com
Cedric Fournet, Microsoft, fournet@microsoft.com
Anil Gangolli, anil@busybuddha.org
David M. Garrett, dave@nulldereference.com
Illya Gerasymchuk, Indépendant, illya@iluxonchik.me
Alessandro Ghedini, Cloudflare Inc., alessandro@cloudflare.com
Daniel Kahn Gillmor, ACLU, dkg@fifthhorseman.net
Matthew Green, Johns Hopkins University, mgreen@cs.jhu.edu
Jens Guballa, ETAS, jens.guballa@etas.com
Felix Guenther, TU Darmstadt, mail@felixguenther.info
Vipul Gupta (co-auteur de la [RFC4492]) Sun Microsystems Laboratories, vipul.gupta@sun.com
Chris Hawk, (co-auteur de la [RFC4492]) Corriente Networks LLC, chris@corriente.net
Kipp Hickman
Alfred Hoenes
David Hopwood, consultant indépendant, david.hopwood@blueyonder.co.uk
Marko Horvat, MPI-SWS, mhorvat@mpi-sws.org
Jonathan Hoyland, Royal Holloway, University of London, jonathan.hoyland@gmail.com
Subodh Iyengar, Facebook, subodh@fb.com
Benjamin Kaduk, Akamai Technologies, kaduk@mit.edu
Hubert Kario, Red Hat Inc., hkario@redhat.com
Phil Karlton, (co-auteur de SSL 3.0)
Leon Klingele, indépendant, mail@leonklingele.de
Paul Kocher, (co-auteur de SSL 3.0) Cryptography Research, paul@cryptography.com
Hugo Krawczyk, IBM, hugokraw@us.ibm.com
Adam Langley, (co-auteur de la [RFC7627]) Google, agl@google.com
Olivier Levillain, ANSSI, olivier.levillain@ssi.gouv.fr
Xiaoyin Liu, University of North Carolina at Chapel Hill, xiaoyin.l@outlook.com
Ilari Liusvaara, indépendant, ilariliusvaara@welho.com
Atul Luykx, K.U. Leuven, atul.luykx@kuleuven.be
Colm MacCarthaigh, Amazon Web Services, colm@allcosts.net
Carl Mehner, USAA, carl.mehner@usaa.com
Jan Mikkelsen, Transactionware, janm@transactionware.com
Bodo Moeller, (co-auteur de la [RFC4492]) Google, bodo@acm.org
Kyle Nekritz, Facebook, knekritz@fb.com
Erik Nygren, Akamai Technologies, erik+ietf@nygren.org
Magnus Nystrom, Microsoft, mnystrom@microsoft.com
Kazuho Oku, DeNA Co., Ltd., kazuhooku@gmail.com
Kenny Paterson, Royal Holloway, University of London, kenny.paterson@rhul.ac.uk
Christopher Patton, University of Florida, cjpatton@ufl.edu
Alfredo Pironti, (co-auteur de la [RFC7627]), INRIA, alfredo.pironti@inria.fr
Andrei Popov, Microsoft, andrei.popov@microsoft.com
Marsh Ray, (co-auteur de la [RFC7627]) Microsoft, maray@microsoft.com
Robert Relyea, Netscape Communications, relyea@netscape.com
Kyle Rose, Akamai Technologies, krose@krose.org
Jim Roskind, Amazon, jroskind@amazon.com

Michael Sabin
Joe Salowey, Tableau Software, joe@salowey.net
Rich Salz, Akamai, rsalz@akamai.com
David Schinazi, Apple Inc., dschinazi@apple.com
Sam Scott, Royal Holloway, University of London, me@samjs.co.uk
Thomas Shrimpton, University of Florida, teshrim@ufl.edu
Dan Simon, Microsoft, Inc., dansimon@microsoft.com
Brian Smith, indépendant, brian@briansmith.org
Brian Sniffen, Akamai Technologies, ietf@bts.evenmere.org
Nick Sullivan, Cloudflare Inc., nick@cloudflare.com
Bjoern Tackmann, University of California, San Diego, btackmann@eng.ucsd.edu
Tim Taubert, Mozilla, ttaubert@mozilla.com
Martin Thomson, Mozilla, mt@mozilla.com
Hannes Tschofenig, Arm Limited, Hannes.Tschofenig@arm.com
Sean Turner, sn3rd, sean@sn3rd.com
Steven Valdez, Google, svaldez@google.com
Filippo Valsorda, Cloudflare Inc., filippo@cloudflare.com
Thyla van der Merwe, Royal Holloway, University of London, tjvdmerwe@gmail.com
Victor Vasiliev, Google, vasilvv@google.com
Hoeteck Wee, École Normale Supérieure, Paris, hoeteck@alum.mit.edu
Tom Weinstein
David Wong, NCC Group, david.wong@nccgroup.trust
Christopher A. Wood, Apple Inc., cawood@apple.com
Tim Wright, Vodafone, timothy.wright@vodafone.com
Peter Wu, indépendant, peter@lekensteyn.nl
Kazu Yamamoto, Internet Initiative Japan Inc., kazu@iij.ad.jp

Adresse de l'auteur

Eric Rescorla
Mozilla
mél : ekr@rtfm.com