RFC 8446 : The Transport Layer Security (TLS) Protocol Version 1.3

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 11 août 2018

Date de publication du RFC : Août 2018

https://www.bortzmeyer.org/8446.html

Après un très long processus, et d'innombrables polémiques, la nouvelle version du protocole de cryptographie TLS, la **1.3**, est enfin publiée. Les changements sont nombreux et, à bien des égards, il s'agit d'un nouveau protocole (l'ancien était décrit dans le RFC 5246 [1], que notre nouveau RFC remplace).

Vous pouvez voir l'histoire de ce RFC sur la Datatracker de l'IETF <https://datatracker.ietf.org/doc/draft-ietf-tls-tls13/>. Le premier brouillon a été publié en avril 2014, plus de trois années avant le RFC. C'est en partie pour des raisons techniques (TLS 1.3 est très différent de ses prédécesseurs) et en partie pour des raisons politiques. C'est que c'est important, la sécurité ! Cinq ans après les révélations de Snowden, on sait désormais que des acteurs puissants et sans scrupules, par exemple les États, espionnent massivement le trafic Internet. Il est donc crucial de protéger ce trafic, entre autres par la cryptographie. Mais dire « cryptographie » ne suffit pas ! Il existe des tas d'attaques contre les protocoles de cryptographie, et beaucoup ont réussi contre les prédécesseurs de TLS 1.3. Il était donc nécessaire de durcir le protocole TLS, pour le rendre moins vulnérable. Et c'est là que les ennuis ont commencé. Car tout le monde ne veut pas de la sécurité. Les États veulent continuer à espionner (le GCHQ britannique s'était clairement opposé à TLS 1.3 sur ce point <https://www.ncsc.gov.uk/blog-post/tls-13-better-individuals-harder-enterprises>). Les entreprises veulent espionner leurs employés (et ont pratiqué un "*lobbying*" intense contre TLS 1.3). Bref, derrière le désir de « sécurité », partagé par tout le monde, il y avait un désaccord de fond sur la surveillance. À chaque réunion de l'IETF, une proposition d'affaiblir TLS pour faciliter la surveillance apparaissait, à chaque fois, elle était rejetée et, tel le zombie des films d'horreur, elle réapparaissait, sous un nom et une forme différente, à la réunion suivante. Par exemple, à la réunion IETF de Prague en juillet 2017, l'affrontement a été particulièrement vif, alors que le groupe de travail TLS espérait avoir presque fini la version 1.3. Des gens se présentant comme "*enterprise networks*" ont critiqué les choix de TLS 1.3, notant qu'il

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc5246.txt

rendait la surveillance plus difficile (c'était un peu le but...) gênant notamment leur déboguage. Ils réclamaient un retour aux algorithmes n'ayant pas de sécurité persistante. Le début a suivi le schéma classique à l'IETF : « vous réclamez un affaiblissement de la sécurité » vs. « mais si on ne le fait pas à l'IETF, d'autres le feront en moins bien », mais, au final, l'IETF est restée ferme et n'a pas accepté de compromissions sur la sécurité de TLS. (Un résumé du débat est dans « "*TLS 1.3 in enterprise networks*" <https://www.cs.uic.edu/~s/musings/tls13-enterprises/> ».)

Pour comprendre les détails de ces propositions et de ces rejets, il faut regarder un peu en détail le protocole TLS 1.3.

Revenons d'abord sur les fondamentaux : TLS est un mécanisme permettant aux applications client/serveur de communiquer au travers d'un réseau non sûr (par exemple l'Internet) tout en empêchant l'écoute et la modification des messages. TLS suppose un mécanisme sous-jacent pour acheminer les bits dans l'ordre, et sans perte. En général, ce mécanisme est TCP. Avec ce mécanisme de transport, et les techniques cryptographiques mises en œuvre par dessus, TLS garantit :

— L'authentification du serveur (celle du client est facultative), authentification qui permet d'empêcher l'attaque de l'intermédiaire, et qui se fait en général via la cryptographie asymétrique,
— La confidentialité des données (mais attention, TLS ne masque pas la **taille** des données, permettant certaines analyses de trafic),
— L'intégrité des données (qui est inséparable de l'authentification : il ne servirait pas à grand'chose d'être sûr de l'identité de son correspondant, si les données pouvaient être modifiées en route).

Ces propriétés sont vraies même si l'attaquant contrôle complètement le réseau entre le client et le serveur (le modèle de menace est détaillé dans la section 3 - surtout la 3.3 - du RFC 3552, et dans l'annexe E de notre RFC).

TLS est un protocole gros et compliqué (ce qui n'est pas forcément optimum pour la sécurité). Le RFC fait 147 pages. Pour dompter cette complexité, TLS est séparé en deux composants :

— Le protocole de salutation ("*handshake protocol*"), chargé d'organiser les échanges du début, qui permettent de choisir les paramètres de la session (c'est un des points délicats de TLS, et plusieurs failles de sécurité ont déjà été trouvées dans ce protocole pour les anciennes versions de TLS),
— Et le protocole des enregistrements ("*record protocol*"), au plus bas niveau, chargé d'acheminer les données chiffrées.

Pour comprendre le rôle de ces deux protocoles, imaginons un protocole fictif simple, qui n'aurait qu'un seul algorithme de cryptographie symétrique, et qu'une seule clé, connue des deux parties (par exemple dans leur fichier de configuration). Avec un tel protocole, on pourrait se passer du protocole de salutation, et n'avoir qu'un protocole des enregistrements, indiquant comment encoder les données chiffrées. Le client et le serveur pourraient se mettre à communiquer immédiatement, sans salutation, poignée de mains et négociation, réduisant ainsi la latence <https://www.bortzmeyer.org/latence.html>. Un tel protocole serait très simple, donc sa sécurité serait bien plus facile à analyser, ce qui est une bonne chose. Mais il n'est pas du tout réaliste : changer la clé utilisée serait complexe (il faudrait synchroniser exactement les deux parties), remplacer l'algorithme si la cryptanalyse en venait à bout (comme c'est arrivé à RC4, cf. RFC 7465) créerait un nouveau protocole incompatible avec l'ancien, communiquer avec un serveur qu'on n'a jamais vu serait impossible (puisque on ne partagerait pas de clé commune), etc. D'où la nécessité du protocole de salutation, où les partenaires :

— S'authentifient avec leur clé publique (ou, si on veut faire comme dans le protocole fictif simple, avec une clé secrète partagée),
— Sélectionnent l'algorithme de cryptographie symétrique qui va chiffrer la session, ainsi que ses paramètres divers,
— Choisir la clé de la session TLS (et c'est là que se sont produites les plus grandes bagarres lors de la conception de TLS 1.3).

Notez que TLS n'est en général pas utilisé tel quel mais via un protocole de haut niveau, comme HTTPS pour sécuriser HTTP. TLS ne suppose pas un usage particulier : on peut s'en servir pour HTTP, pour SMTP (RFC 7672), pour le DNS (RFC 7858), etc. Cette intégration dans un protocole de plus haut niveau pose parfois elle-même des surprises en matière de sécurité, par exemple si l'application utilisatrice ne fait pas attention à la sécurité (Voir mon exposé à Devoxx <http://blog.soat.fr/2014/04/devoxx-2014-utiliser-tls-sans-se-tromper-une-conference-animee-par-stephane-bortzmeyer/>, et ses transparents (en ligne sur https://www.bortzmeyer.org/files/bortzmeyer-tls-devoxx.odp).)

TLS 1.3 est plutôt un nouveau protocole qu'une nouvelle version, et il n'est pas directement compatible avec son prédécesseur, TLS 1.2 (une application qui ne connait que 1.3 ne peut pas parler avec une application qui ne connait que 1.2.) En pratique, les bibliothèques qui mettent en œuvre TLS incluent en général les différentes versions, et un mécanisme de négociation de la version utilisée permet normalement de découvrir la version maximum que les deux parties acceptent (historiquement, plusieurs failles sont venues de ce point, avec des pare-feux stupidement configurés qui interféraient avec la négociation).

La section 1.3 de notre RFC liste les différences importantes entre TLS 1.2 (qui était normalisé dans le RFC 5246) et 1.3 :

- La liste des algorithmes de cryptographie symétrique acceptés a été violemment réduite. Beaucoup trop longue en TLS 1.2, offrant trop de choix, comprenant plusieurs algorithmes faibles, elle ouvrait la voie à des attaques par repli. Les « survivants » de ce nettoyage sont tous des algorithmes à chiffrement intègre.
- Un nouveau service apparait, 0-RTT ("*zero round-trip time*", la possibilité d'établir une session TLS avec un seul paquet, en envoyant les données tout de suite), qui réduit la latence <https://www.bortzmeyer.org/latence.html> du début de l'échange. Attention, rien n'est gratuit en ce monde, et 0-RTT présente des nouveaux dangers, et ce nouveau service a été un des plus controversés <https://github.com/tlswg/tls13-spec/issues/1001> lors de la mise au point de TLS 1.3, entrainant de nombreux débats à l'IETF.
- Désormais, la sécurité future est systématique, la compromission d'une clé secrète ne permet plus de déchiffrer les anciennes communications. Plus de clés publiques statiques, tout se fera par clés éphémères. C'était le point qui a suscité le plus de débats à l'IETF, car cela complique sérieusement la surveillance (ce qui est bien le but) et le déboguage <https://www.bortzmeyer.org/crypto-debug.html>. L'ETSI, représentante du patronat, a même normalisé son propre TLS délibérement affaibli, eTLS <https://www.etsi.org/news-events/news/1358-2018-11-press-ets
- Plusieurs messages de négociation qui étaient auparavant en clair sont désormais chiffrés. Par contre, l'indication du nom du serveur (SNI, section 3 du RFC 6066) reste en clair et c'est l'une des principales limites de TLS en ce qui concerne la protection de la vie privée. Le problème est important, mais très difficile à résoudre (voir par exemple la proposition ESNI, ""<https://datatracker.ietf.org/doc/draft-ietf-tls-esni/> et son cahier des charges dans le RFC 8744.)
- Les fonctions de dérivation de clé ont été refaites.
- La machine à états utilisée pour l'établissement de la connexion également (elle est détaillée dans l'annexe A du RFC).
- Les algorithmes asymétriques à courbes elliptiques font maintenant partie de la définition de base de TLS (cf. RFC 7748), et on voit arriver des nouveaux comme ed25519 (cf. RFC 8422).
- Par contre, DSA a été retiré.
- Le mécanisme de négociation du numéro de version (permettant à deux machines n'ayant pas le même jeu de versions TLS de se parler) a changé. L'ancien était très bien mais, mal implémenté, il a suscité beaucoup de problèmes d'interopérabilité. Le nouveau est censé mieux gérer les innombrables systèmes bogués qu'on trouve sur l'Internet (la bogue ne provenant pas tant de la bibliothèque TLS utilisée que des pare-feux mal programmés et mal configurés qui sont souvent mis devant). J'en profite pour vous recommander l'article « HTTPS : de SSL à TLS 1.3 <https://openweb.eu.org/articles/https-de-ssl-a-tls-1-3> », sur ce sujet de la négociation de version.

— La reprise d'une session TLS précédente fait l'objet désormais d'un seul mécanisme, qui est le même que celui pour l'usage de clés pré-partagées. La négociation TLS peut en effet être longue, en terme de latence, et ce mécanisme permet d'éviter de tout recommencer à chaque connexion. Deux machines qui se parlent régulièrement peuvent ainsi gagner du temps.

Un bon résumé de ce nouveau protocole est dans l'article de Mark Nottingham <https://blog.apnic.net/2017/12/12/internet-protocols-changing/>.

Ce RFC concerne TLS 1.3 mais il contient aussi quelques changements pour la version 1.2 (section 1.4 du RFC), comme un mécanisme pour limiter les attaques par repli portant sur le numéro de version, et des mécanismes de la 1.3 « portés » vers la 1.2 sous forme d'extensions TLS.

La section 2 du RFC est un survol général de TLS 1.3 (le RFC fait 147 pages, et peu de gens le liront intégralement). Au début d'une session TLS, les deux parties, avec le protocole de salutation, négocient les paramètres (version de TLS, algorithmes cryptographiques) et définissent les clés qui seront utilisées pour le chiffrement de la session. En simplifiant, il y a trois phases dans l'établissement d'une session TLS :

— Définition des clés de session, et des paramètres cryptographiques, le client envoie un `ClientHello`, le serveur répond avec un `ServerHello`,
— Définition des autres paramètres (par exemple l'application utilisée au-dessus de TLS, ou bien la demande `CertificateRequest` d'un certificat client), cette partie est chiffrée, contrairement à la précédente,
— Authentification du serveur, avec le message `Certificate` (qui ne contient pas forcément un certificat, cela peut être une clé brute - RFC 7250 ou une clé d'une session précédente - RFC 7924).

Un message `Finished` termine cette ouverture de session. (Si vous êtes fana de futurisme, notez que seule la première étape pourrait être remplacée par la distribution quantique de clés, les autres resteraient indispensables. Contrairement à ce que promettent ses promoteurs, la QKD ne dispense pas d'utiliser les protocoles existants.)

Comment les deux parties se mettent-elles d'accord sur les clés ? Trois méthodes :

— Diffie-Hellman sur courbes elliptiques qui sera sans doute la plus fréquente,
— Clé pré-partagée,
— Clé pré-partagée avec Diffie-Hellman,
— Et la méthode RSA, elle, disparait de la norme (mais RSA peut toujours être utilisé pour l'authentification, autrement, cela ferait beaucoup de certificats à jeter…)

Si vous connaissez la cryptographie, vous savez que les PSK, les clés partagées, sont difficiles à gérer, puisque devant être transmises de manière sûre avant l'établissement de la connexion. Mais, dans TLS, une autre possibilité existe : si une session a été ouverte sans PSK, en n'utilisant que de la cryptographie asymétrique, elle peut être enregistrée, et resservir, afin d'ouvrir les futures discussions plus rapidement. TLS 1.3 utilise le même mécanisme pour des « vraies » PSK, et pour celles issues de cette reprise de sessions précédentes (contrairement aux précédentes versions de TLS, qui utilisaient un mécanisme séparé, celui du RFC 5077, désormais abandonné).

Si on a une PSK (gérée manuellement, ou bien via la reprise de session), on peut même avoir un dialogue TLS dit « 0-RTT ». Le premier paquet du client peut contenir des données, qui seront acceptées et traitées par le serveur. Cela permet une importante diminution de la latence <https://www.bortzmeyer.org/latence.html>, dont il faut rappeler qu'elle est souvent le facteur limitant des performances. Par contre, comme rien n'est idéal dans cette vallée de larmes, cela se fait au détriment de la sécurité :

— Plus de confidentialité persistante, si la PSK est compromise plus tard, la session pourra être déchiffrée,
— Le rejeu devient possible, et l'application doit donc savoir gérer ce problème.

La section 8 du RFC et l'annexe E.5 détaillent ces limites, et les mesures qui peuvent être prises.

Le protocole TLS est décrit avec un langage spécifique, décrit de manière relativement informelle dans la section 3 du RFC. Ce langage manipule des types de données classiques :

— Scalaires (`uint8`, `uint16`),
— Tableaux, de taille fixe - `Datum[3]` ou variable, avec indication de la longueur au début - `uint16 longer<0..800>`,
— Énumérations (`enum { red(3), blue(5), white(7) } Color;`),
— Enregistrements structurés, y compris avec variantes (la présence de certains champs dépendant de la valeur d'un champ).

Par exemple, tirés de la section 4 (l'annexe B fournit la liste complète), voici, dans ce langage, la liste des types de messages pendant les salutations, une énumération :

```
 enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;
```

Et le format de base d'un message du protocole de salutation :

```
struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:    NewSessionTicket;
        case key_update:            KeyUpdate;
    };
} Handshake;
```

La section 4 fournit tous les détails sur le protocole de salutation, notamment sur la délicate négociation des paramètres cryptographiques. Notez que la renégociation en cours de session `<https://www.bortzmeyer.org/tls-renego.html>` a disparu, donc un `ClientHello` ne peut désormais plus être envoyé qu'au début.

Un problème auquel a toujours dû faire face TLS est celui de la négociation de version, en présence de mises en œuvre boguées, et, surtout, en présence de boitiers intermédiaires encore plus bogués (pare-feux ignorants, par exemple, que des DSI ignorantes placent un peu partout). Le modèle original de

TLS pour un client était d'annoncer dans le `ClientHello` le plus grand numéro de version qu'on gère, et de voir dans `ServerHello` le maximum imposé par le serveur. Ainsi, un client TLS 1.2 parlant à un serveur qui ne gère que 1.1 envoyait `ClientHello(client_version=1.2)` et, en recevant `ServerHello(server_version=1.1)`, se repliait sur TLS 1.1, la version la plus élevée que les deux parties gèraient. En pratique, cela ne marche pas aussi bien. On voyait par exemple des serveurs (ou, plus vraisemblablement, des pare-feux bogués) qui raccrochaient brutalement en présence d'un numéro de version plus élevé, au lieu de suggérer un repli. Le client n'avait alors que le choix de renoncer, ou bien de se lancer dans une série d'essais/erreurs (qui peut être longue, si le serveur ou le pare-feu bogué ne répond pas).

TLS 1.3 change donc complètement le mécanisme de négociation. Le client annonce toujours la version 1.2 (en fait 0x303, pour des raisons historiques), et la vraie version est mise dans une extension, `supported_versions` (section 4.2.1), dont on espère qu'elle sera ignorée par les serveurs mal gérés. (L'annexe D du RFC détaille ce problème de la négociation de version.) Dans la réponse `ServerHello`, un serveur 1.3 doit inclure cette extension, autrement, il faut se rabattre sur TLS 1.2.

En parlant d'extensions, concept qui avait été introduit originellement dans le RFC 4366, notre RFC reprend des extensions déjà normalisées, comme le SNI ("*Server Name Indication*") du RFC 6066, le battement de cœur du RFC 6520, le remplissage du `ClientHello` du RFC 7685, et en ajoute dix, dont `supported_versions`. Certaines de ces extensions doivent être présentes dans les messages `Hello`, car la sélection des paramètres cryptographiques en dépend, d'autres peuvent être uniquement dans les messages `EncryptedExtensions`, une nouveauté de TLS 1.3, pour les extensions qu'on n'enverra qu'une fois le chiffrement commencé. Le RFC en profite pour rappeler que les messages `Hello` ne sont pas protégés cryptographiquement, et peuvent donc être modifiés (le message `Finished` résume les décisions prises et peut donc protéger contre ce genre d'attaques).

Autrement, parmi les autres nouvelles extensions :
— Le petit gâteau ("*cookie*"), pour tester la joignabilité,
— Les données précoces ("*early data*"), extension qui permet d'envoyer des données dès le premier message (« "*O-RTT*" »), réduisant ainsi la latence <`https://www.bortzmeyer.org/latence.html`>, un peu comme le fait le "*TCP Fast Open*" du RFC 7413,
— Liste des AC ("*certificate authorities*"), qui, en indiquant la liste des AC connues du client, peut aider le serveur à choisir un certificat qui sera validé (par exemple en n'envoyant le certificat CAcert <`https://www.bortzmeyer.org/cacert.html`> que si le client connait cette AC).

La section 5 décrit le protocole des enregistrements ("*record protocol*"). C'est ce sous-protocole qui va prendre un flux d'octets, le découper en enregistrements, les protéger par le chiffrement puis, à l'autre bout, déchiffrer et reconstituer le flux...Notez que « protégé » signifie à la fois confidentialité et intégrité puisque TLS 1.3, contrairement à ses prédécesseurs, impose AEAD (RFC 5116).

Les enregistrements sont typés et marqués "*handshake*" (la salutation, vue dans la section précédente), "*change cipher spec*", "*alert*" (pour signaler un problème) et "*application data*" (les données elle-mêmes) :

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

Le contenu des données est évidemment incompréhensible, en raison du chiffrement (voici un enregistrement de type 23, données, vu par tshark) :

```
TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 6316
    Encrypted Application Data: eb0e21f124f82eee0b7a37a1d6d866b075d0476e6f00cae7...
```

Et décrite par la norme dans son langage formel :

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

(Oui, le numéro de version reste à TLS 1.2 pour éviter d'énerver les stupides "*middleboxes*".) Notez que des extensions à TLS peuvent introduire d'autres types d'enregistrements.

Une faiblesse classique de TLS est que la taille des données chiffrées n'est pas dissimulée. Si on veut savoir à quelle page d'un site Web un client HTTP a accédé, on peut parfois le déduire de l'observation de cette taille <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6234422>. D'où la possibilité de faire du remplissage pour dissimuler cette taille (section 5.4 du RFC). Notez que le RFC ne suggère pas de politique de remplissage spécifique (ajouter un nombre aléatoire ? Tout remplir jusqu'à la taille maximale ?), c'est un choix compliqué. Il note aussi que certaines applications font leur propre remplissage, et qu'il n'est alors pas nécessaire que TLS le fasse.

La section 6 du RFC est dédiée au cas des alertes. C'est un des types d'enregistrements possibles, et, comme les autres, il est chiffré, et les alertes sont donc confidentielles. Une alerte a un niveau et une description :

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

Le niveau indiquait si l'alerte est fatale mais n'est plus utilisé en TLS 1.2, où il faut se fier uniquement à la description, une énumération des problèmes possibles (message de type inconnu, mauvais certificat, enregistrement non décodable - rappelez-vous que TLS 1.3 n'utilise que du chiffrement intègre, problème interne au client ou au serveur, extension non acceptée, etc). La section 6.2 donne une liste des erreurs fatales, qui doivent mener à terminer immédiatement la session TLS.

La section 8 du RFC est entièrement consacrée à une nouveauté délicate, le « 0-RTT ». Ce terme désigne la possibilité d'envoyer des données dès le premier paquet, sans les nombreux échanges de paquets qui sont normalement nécessaires pour établir une session TLS. C'est très bien du point de vue des performances, mais pas forcément du point de vue de la sécurité puisque, sans échanges, on ne peut plus vérifier à qui on parle. Un attaquant peut réaliser une attaque par rejeu en envoyant à nouveau un paquet qu'il a intercepté. Un serveur doit donc se défendre en se souvenant des données déjà envoyées et en ne les acceptant pas deux fois. (Ce qui peut être plus facile à dire qu'à faire ; le RFC contient une

bonne discussion très détaillée des techniques possibles, et de leurs limites. Il y en a des subtiles, comme d'utiliser des systèmes de mémorisation ayant des faux positifs, comme les filtres de Bloom, parce qu'ils ne produiraient pas d'erreurs, ils rejetteraient juste certains essais 0-RTT légitimes, cela ne serait donc qu'une légère perte de performance.)

La section 9 de notre RFC se penche sur un problème difficile, la conformité des mises en œuvres de TLS. D'abord, les algorithmes obligatoires. Afin de permettre l'interopérabilité, **toute** mise en œuvre de TLS doit avoir la suite de chiffrement `TLS_AES_128_GCM_SHA256` (AES en mode GCM avec SHA-256). D'autres suites sont recommandées (cf. annexe B.4). Pour l'authentification, RSA avec SHA-256 et ECDSA sont obligatoires. Ainsi, deux programmes différents sont sûrs de pouvoir trouver des algorithmes communs. La possibilité d'authentification par certificats PGP du RFC 6091 a été retirée.

De plus, certaines extensions à TLS sont obligatoires, un pair TLS 1.3 ne peut pas les refuser :
— `supported_versions`, nécessaire pour annoncer TLS 1.3,
— `cookie`,
— `signature_algorithms`, `signature_algorithms_cert`, `supported_groups` et `key_share`,
— `server_name`, c'est à dire SNI ("*Server Name Indication*"), souvent nécessaire pour pouvoir choisir le bon certificat (cf. section 3 du RFC 6066).

La section 9 précise aussi le comportement attendu des équipements intermédiaires. Ces dispositifs (pare-feux, par exemple, mais pas uniquement) ont toujours été une plaie pour TLS. Alors que TLS vise à fournir une communication sûre, à l'abri des équipements intermédiaires, ceux-ci passent leur temps à essayer de s'insérer dans la communication, et souvent la cassent. Normalement, TLS 1.3 est conçu pour que ces interférences ne puissent pas mener à un repli (le repli est l'utilisation de paramètres moins sûrs que ce que les deux machines auraient choisi en l'absence d'interférence).

Il y a deux grandes catégories d'intermédiaires, ceux qui tripotent la session TLS sans être le client ou le serveur, et ceux qui terminent la session TLS de leur côté. Attention, dans ce contexte, « terminer » ne veut pas dire « y mettre fin », mais « la sécurité TLS se termine ici, de manière à ce que l'intermédiaire puisse accéder au contenu de la communication ». Typiquement, une "*middlebox*" qui « termine » une session TLS va être serveur TLS pour le client et client TLS pour le serveur, s'insérant complètement dans la conversation. Normalement, l'authentification vise à empêcher ce genre de pratiques, et l'intermédiaire ne sera donc accepté que s'il a un certificat valable. C'est pour cela qu'en entreprise, les machines officielles sont souvent installées avec une AC contrôlée par le vendeur du boitier intermédiaire, de manière à permettre l'interception.

Le RFC ne se penche pas sur la légitimité de ces pratiques, uniquement sur leurs caractéristiques techniques. (Les boitiers intermédiaires sont souvent programmés avec les pieds <`https://www.bortzmeyer.org/killed-by-proxy.html`>, et ouvrent de nombreuses failles <`https://www.bortzmeyer.org/https-interception.html`>.) Le RFC rappelle notamment que l'intermédiaire qui termine une session doit suivre le RFC à la lettre (ce qui devrait aller sans dire…)

Depuis le RFC 4346, il existe plusieurs registres IANA pour TLS, décrits en section 11, avec leurs nouveautés. En effet, plusieurs choix pour TLS ne sont pas « câblés en dur » dans le RFC mais peuvent évoluer indépendamment. Par exemple, le registre de suites cryptographiques <`https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-4`> a une politique d'enregistrement « spécification nécessaire » (cf. RFC 8126, sur les politiques d'enregistrement). La cryptographie fait régulièrement des progrès, et il faut donc pouvoir modifier la liste des suites acceptées (par exemple lorsqu'il faudra y ajouter les algorithmes post-quantiques <`https://www.bortzmeyer.org/pas-sage-en-seine-quantique.html`>) sans avoir à toucher au RFC (l'annexe B.4 donne la liste **actuelle**). Le registre des types de contenu <`https://www.iana.org/assignments/`

tls-parameters/tls-parameters.xml#tls-parameters-5>, lui, a une politique d'enregistrement bien plus stricte, « action de normalisation ». On crée moins souvent des types que des suites cryptographiques. Même chose pour le registre des alertes <https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-6> ou pour celui des salutations <https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-7>.

L'annexe C du RFC plaira aux programmeurs, elle donne plusieurs conseils pour une mise en œuvre correcte de TLS 1.3 (ce n'est pas tout d'avoir un protocole correct, il faut encore qu'il soit programmé correctement). Pour aider les développeurs à déterminer s'ils ont correctement fait le travail, un futur RFC fournira des vecteurs de test.

Un des conseils les plus importants est évidemment de faire attention au générateur de nombres aléatoires, source de tant de failles de sécurité en cryptographie. TLS utilise des nombres qui doivent être imprévisibles à un attaquant pour générer des clés de session. Si ces nombres sont prévisibles, toute la cryptographie s'effondre. Le RFC conseille fortement d'utiliser un générateur existant (comme /dev/urandom sur les systèmes Unix) plutôt que d'écrire le sien, ce qui est bien plus difficile qu'il ne semble. (Si on tient quand même à le faire, le RFC 4086 est une lecture indispensable.)

Le RFC conseille également de vérifier le certificat du partenaire par défaut (quitte à fournir un moyen de débrayer cette vérification). Si ce n'est pas le cas, beaucoup d'utilisateurs du programme ou de la bibliothèque oublieront de le faire. Il suggère aussi de ne pas accepter certains certificats trop faibles (clé RSA de seulement 1 024 bits, par exemple).

Il existe plusieurs moyens avec TLS de ne pas avoir d'authentification du serveur : les clés brutes du RFC 7250 (à la place des certificats), ou bien les certificats auto-signés. Dans ces conditions, une attaque de l'homme du milieu est parfaitement possibe, et il faut donc prendre des précautions supplémentaires (par exemple DANE, normalisé dans le RFC 6698, que le RFC oublie malheureusement de citer).

Autre bon conseil de cryptographie, se méfier des attaques fondées sur la mesure du temps de calcul, et prendre des mesures appropriées (par exemple en vérifiant que le temps de calcul est le même pour des données correctes et incorrectes).

Il n'y a aucune bonne raison d'utiliser certains algorithmes faibles (comme RC4, abandonné depuis le RFC 7465), et le RFC demande que le code pour ces algorithmes ne soit pas présent, afin d'éviter une attaque par repli (annexes C.3 et D.5 du RFC). De la même façon, il demande de ne jamais accepter SSL v3 (RFC 7568).

L'expérience a prouvé que beaucoup de mises en œuvre de TLS ne réagissaient pas correctement à des options inattendues, et le RFC rappelle donc qu'il faut ignorer les suites cryptographiques inconnues (autrement, on ne pourrait jamais introduire une nouvelle suite, puisqu'elle casserait les programmes), et ignorer les extensions inconnues (pour la même raison).

L'annexe D, elle, est consacrée au problème de la communication avec un vieux partenaire, qui ne connait pas TLS 1.3. Le mécanisme de négociation de la version du protocole à utiliser a complètement changé en 1.3. Dans la 1.3, le champ version du ClientHello contient 1.2, la vraie version étant dans l'extension supported_versions. Si un client 1.3 parle avec un serveur ¡= 1.2, le serveur ne connaitra pas cette extension et répondra sans l'extension, avertissant ainsi le client qu'il faudra parler en 1.2 (ou plus vieux). Ça, c'est si le serveur est correct. S'il ne l'est pas ou, plus vraisemblablement, s'il est derrière une "*middlebox*" boguée, on verra des problèmes comme par exemple le refus de répondre aux clients utilisant des extensions inconnues (ce qui sera le cas pour supported_versions), soit en rejettant ouvertement la demande soit, encore pire, en l'ignorant. Arriver à gérer des serveurs/"*middleboxes*"

incorrects est un problème complexe. Le client peut être tenté de re-essayer avec d'autres options (par exemple tenter du 1.2, sans l'extension `supported_versions`). Cette méthode n'est pas conseillée. Non seulement elle peut prendre du temps (attendre l'expiration du délai de garde, re-essayer…) mais surtout, elle ouvre la voie à des attaques par repli : l'attaquant bloque les `ClientHello` 1.3 et le client, croyant bien faire, se replie sur une version plus ancienne et sans doute moins sûre de TLS.

En parlant de compatibilité, le « 0-RTT » n'est évidemment pas compatible avec les vieilles versions. Le client qui envoie du « 0-RTT » (des données dans le `ClientHello`) doit donc savoir que, si la réponse est d'un serveur ¡= 1.2, la session ne pourra pas être établie, et il faudra donc réessayer sans 0-RTT.

Naturellement, les plus gros problèmes ne surviennent pas avec les clients et les serveurs mais avec les "*middleboxes*". Plusieurs études ont montré leur caractère néfaste (cf. présentation à l'IETF 100 <https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/>, mesures avec Chrome <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html> (qui indique également que certains serveurs TLS sont gravement en tort, comme celui installé dans les imprimantes Canon), mesures avec Firefox <https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>, et encore d'autres mesures <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>). Le RFC suggère qu'on limite les risques en essayant d'imiter le plus possible une salutation de TLS 1.2, par exemple en envoyant des messages `change_cipher_spec`, qui ne sont plus utilisés en TLS 1.3, mais qui peuvent rassurer la "*middlebox*" (annexe D.4).

Enfin, le RFC se termine par l'annexe E, qui énumère les propriétés de sécurité de TLS 1.3 : même face à un attaquant actif (RFC 3552), le protocole de salutation de TLS garantit des clés de session communes et secrètes, une authentification du serveur (et du client si on veut), et une sécurité persistante, même en cas de compromission ultérieure des clés (sauf en cas de 0-RTT, un autre des inconvénients sérieux de ce service, avec le risque de rejeu). De nombreuses analyses détaillées de la sécurité de TLS sont listées dans l'annexe E.1.6. À lire si vous voulez travailler ce sujet.

Quant au protocole des enregistrements, celui de TLS 1.3 garantit confidentialité et intégrité (RFC 5116).

TLS 1.3 a fait l'objet de nombreuses analyses de sécurité par des chercheurs, avant même sa normalisation, ce qui est une bonne chose (et qui explique en partie les retards). Notre annexe E pointe également les limites restantes de TLS :

— Il est vulnérable à l'analyse de trafic. TLS n'essaie pas de cacher la taille des paquets, ni l'intervalle de temps entre eux. Ainsi, si un client accède en HTTPS à un site Web servant quelques dizaines de pages aux tailles bien différentes, il est facile de savoir quelle page a été demandée, juste en observant les tailles. (Voir « "*I Know Why You Went to the Clinic : Risks and Realization of HTTPS Traffic Analysis*" <https://arxiv.org/abs/1403.0297> », de Miller, B., Huang, L., Joseph, A., et J. Tygar et « "*HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting*" <https://is.muni.cz/repo/1299983/https_client_identification-paper.pdf> », de Husak, M., [Caractère Unicode non montré [2]]ermak, M., Jirsik, T., et P. [Caractère Unicode non montré]eleda). TLS fournit un mécanisme de remplissage avec des données bidon, permettant aux applications de brouiller les pistes. Certaines applications utilisant TLS ont également leur propre remplissage (par exemple, pour le DNS, c'est le RFC 7858). De même, une mise en œuvre de TLS peut retarder les paquets pour rendre l'analyse des intervalles plus difficile. On voit que dans les deux cas, taille des paquets et intervalle entre eux, résoudre le problème fait perdre en performance (c'est pour cela que ce n'est pas intégré par défaut).

2. Car trop difficile à faire afficher par LaTeX

— TLS peut être également vulnérable à des attaques par canal auxiliaire. Par exemple, la durée des opérations cryptographiques peut être observée, ce qui peut donner des informations sur les clés. TLS fournit quand même quelques défenses : l'AEAD facilite la mise en œuvre de calculs en temps constant, et format uniforme pour toutes les erreurs, empêchant un attaquant de trouver quelle erreur a été déclenchée.

Le 0-RTT introduit un nouveau risque, celui de rejeu. (Et 0-RTT a sérieusement contribué aux délais qu'à connu le projet TLS 1.3, plusieurs participants à l'IETF protestant contre cette introduction risquée <http://bristolcrypto.blogspot.com/2017/03/pkc-2017-kenny-paterson-accepting-bets.html>.) Si l'application est idempotente, ce n'est pas très grave. Si, par contre, les effets d'une requête précédentes peuvent être rejoués, c'est plus embêtant (imaginez un transfert d'argent répété…) TLS ne promet rien en ce domaine, c'est à chaque serveur de se défendre contre le rejeu (la section 8 donne des idées à ce sujet). Voilà pourquoi le RFC demande que les requêtes 0-RTT ne soient pas activées par défaut, mais uniquement quand l'application au-dessus de TLS le demande. (Cloudflare, par exemple, n'active pas le 0-RTT par défaut.)

Voilà, vous avez maintenant fait un tour complet du RFC, mais vous savez que la cryptographie est une chose difficile, et pas seulement dans les algorithmes cryptographiques (TLS n'en invente aucun, il réutilise des algorithmes existants comme AES ou ECDSA), mais aussi dans les protocoles cryptographiques, un art complexe. N'hésitez donc pas à lire le RFC en détail, et à vous méfier des résumés forcément toujours sommaires, comme cet article.

À part le 0-RTT, le plus gros débat lors de la création de TLS 1.3 avait été autour du concept que ses partisans nomment « visibilité » et ses adversaires « surveillance ». C'est l'idée qu'il serait bien pratique si on (on : le patron, la police, le FAI…) pouvait accéder au contenu des communications TLS. « Le chiffrement, c'est bien, à condition que je puisse lire les données quand même » est l'avis des partisans de la visibilité. Cela avait été proposé dans les "*Internet-Drafts*" `draft-green-tls-static-dh-in-tls13` et `draft-rhrd-tls-tls13-visibility`. Je ne vais pas ici pouvoir capturer la totalité du débat, juste noter quelques points qui sont parfois oubliés dans la discussion. Côté partisans de la visibilité :

— Dans une entreprise capitaliste, il n'y pas de citoyens, juste un patron et des employés. Les ordinateurs appartiennent au patron, et les employés n'ont pas leur mot à dire. Le patron peut donc décider d'accéder au contenu des communications chiffrées.
— Il existe des règles (par exemple PCI-DSS dans le secteur financier ou HIPAA dans celui de la santé) qui requièrent de certaines entreprises qu'elles sachent en détail tout ce qui circule sur le réseau. Le moyen le plus simple de le faire est de surveiller le contenu des communications, même chiffrées. (Je ne dis pas que ces règles sont intelligentes, juste qu'elles existent. Notons par exemple que les mêmes règles imposent d'utiliser du chiffrement fort, sans faille connue, ce qui est contradictoire.)
— Enregistrer le trafic depuis les terminaux <https://www.bortzmeyer.org/terminal-host.html> est compliqué en pratique : applications qui n'ont pas de mécanisme de journalisation du trafic, systèmes d'exploitation fermés, boîtes noires…
— TLS 1.3 risque de ne pas être déployé dans les entreprises qui tiennent à surveiller le trafic, et pourrait même être interdit dans certains pays, où la surveillance passe avant les droits humains.

Et du côté des adversaires de la surveillance :

— La cryptographie, c'est compliqué et risqué. TLS 1.3 est déjà assez compliqué comme cela. Lui ajouter des fonctions (surtout des fonctions délibérement conçues pour affaiblir ses propriétés de sécurité) risque fort d'ajouter des failles de sécurité. D'autant plus que TLS 1.3 a fait l'objet de nombreuses analyses de sécurité avant son déploiement, et qu'il faudrait tout recommencer.
— Contrairement à ce que semblent croire les partisans de la « visibilité », il n'y a pas que HTTPS qui utilise TLS. Ils ne décrivent jamais comment leur proposition marcherait avec des protocoles autres que HTTPS.
— Pour HTTPS, et pour certains autres protocoles, une solution simple, si on tient absolument à intercepter tout le trafic, est d'avoir un relais explicite, configuré dans les applications, et combiné avec un blocage dans le pare-feu des connexions TLS directes. Les partisans de la visibilté ne sont en général pas enthousiastes pour cette solution car ils voudraient faire de la surveillance furtive, sans qu'elle se voit dans les applications utilisées par les employés ou les citoyens.

— Les partisans de la « visibilité » disent en général que l'interception TLS serait uniquement à l'intérieur de l'entreprise, pas pour l'Internet public. Mais, dans ce cas, tous les terminaux <https://www.bortzmeyer.org/terminal-host.html> sont propriété de l'entreprise et contrôlés par elle, donc elle peut les configurer pour copier tous les messages échangés. Et, si certains de ces terminaux sont des boîtes noires, non configurables et dont on ne sait pas bien ce qu'ils font, eh bien, dans ce cas, on se demande pourquoi des gens qui insistent sur leurs obligations de surveillance mettent sur leur réseau des machines aussi incontrôlables.
— Dans ce dernier cas (surveillance uniquement au sein d'une entreprise), le problème est interne à l'entreprise, et ce n'est donc pas à l'IETF, organisme qui fait des normes pour l'Internet, de le résoudre. Après tout, rien n'empêche ces entreprises de garder TLS 1.2.

Revenons maintenant aux choses sérieuses, avec les mises en œuvre de TLS 1.3. Il y en existe au moins une dizaine à l'heure actuelle mais, en général, pas dans les versions officiellement publiées des logiciels. Notons quand même que Firefox 61 sait faire du TLS 1.3. Les autres mises en œuvre sont prêtes, même si pas forcément publiées. Prenons l'exemple de la bibliothèque GnuTLS. Elle dispose de TLS 1.3 depuis la version 3.6.3. Pour l'instant, il faut compiler cette version avec l'option `./configure --enable-tls13-support`, qui n'est pas encore activée par défaut. Un bon article du mainteneur de GnuTLS <https://nikmav.blogspot.com/2018/05/gnutls-and-tls-13.html> explique bien les nouveautés de TLS 1.3.

Une fois GnuTLS correctement compilé, on peut utiliser le programme en ligne de commande `gnutls-cli` avec un serveur qui accepte TLS 1.3 :

```
% gnutls-cli  gmail.com
...
- Description: (TLS1.3)-(ECDHE-X25519)-(RSA-PSS-RSAE-SHA256)-(AES-256-GCM)
- Ephemeral EC Diffie-Hellman parameters
 - Using curve: X25519
 - Curve size: 256 bits
- Version: TLS1.3
- Key Exchange: ECDHE-RSA
- Server Signature: RSA-PSS-RSAE-SHA256
- Cipher: AES-256-GCM
- MAC: AEAD
...
```

Et ça marche, on fait du TLS 1.3. Si vous préférez écrire le programme vous-même, regardez ce petit programme (en ligne sur `https://www.bortzmeyer.org/files/test-tls13.c`). Si GnuTLS est en `/local`, il se compilera avec `cc -I/local/include -Wall -Wextra -o test-tls13 test-tls13.c -L/local/lib -lgnutls` et s'utilisera avec :

```
% ./test-tls13 www.ietf.org
TLS connection using "TLS1.3 AES-256-GCM"

%  ./test-tls13 gmail.com
TLS connection using "TLS1.3 AES-256-GCM"

%  ./test-tls13 mastodon.gougere.fr
TLS connection using "TLS1.2 AES-256-GCM"

% ./test-tls13 www.bortzmeyer.org
TLS connection using "TLS1.2 AES-256-GCM"

% ./test-tls13 blog.cloudflare.com
TLS connection using "TLS1.3 AES-256-GCM"
```

Cela vous donne une petite idée des serveurs qui acceptent TLS 1.3.

Un pcap d'une session TLS 1.3 est disponible en (en ligne sur `https://www.bortzmeyer.org/files/tls13.pcap`). Notez que le numéro de version n'est pas encore celui du RFC (0x304). Ici, 0x7f1c désigne l'"*Internet-Draft*" numéro 28. Voici la session vue par tshark :

```
    1   0.000000 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TCP 94 36866 → https(443) [S
    2   0.003052 2400:cb00:2048:1::6814:55 → 2001:67c:370:1998:9819:4f92:d0c0:e94d TCP 86 https(443) → 36866 [S
    3   0.003070 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TCP 74 36866 → https(443) [A
    4   0.003354 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TLSv1 403 Client Hello
    5   0.006777 2400:cb00:2048:1::6814:55 → 2001:67c:370:1998:9819:4f92:d0c0:e94d TCP 74 https(443) → 36866 [A
    6   0.011393 2400:cb00:2048:1::6814:55 → 2001:67c:370:1998:9819:4f92:d0c0:e94d TLSv1.3 6496 Server Hello, Ch
    7   0.011413 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TCP 74 36866 → https(443) [A
    8   0.011650 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TLSv1.3 80 Change Cipher Spec
    9   0.012685 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TLSv1.3 148 Application Data
   10   0.015693 2400:cb00:2048:1::6814:55 → 2001:67c:370:1998:9819:4f92:d0c0:e94d TCP 74 https(443) → 36866 [A
   11   0.015742 2400:cb00:2048:1::6814:55 → 2001:67c:370:1998:9819:4f92:d0c0:e94d TLSv1.3 524 Application Data
   12   0.015770 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TCP 74 36866 → https(443) [R
   13   0.015788 2400:cb00:2048:1::6814:55 → 2001:67c:370:1998:9819:4f92:d0c0:e94d TCP 74 https(443) → 36866 [R
   14   0.015793 2001:67c:370:1998:9819:4f92:d0c0:e94d → 2400:cb00:2048:1::6814:55 TCP 74 36866 → https(443) [R
```

Et, complètement décodée par tshark :

```
Secure Sockets Layer [sic]
    TLSv1 Record Layer: Handshake Protocol: Client Hello
        Content Type: Handshake (22)
        Version: TLS 1.0 (0x0301)
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Version: TLS 1.2 (0x0303)
...
            Extension: supported_versions (len=9)
                Type: supported_versions (43)
                Length: 9
                Supported Versions length: 8
                Supported Version: Unknown (0x7f1c)
                Supported Version: TLS 1.2 (0x0303)
                Supported Version: TLS 1.1 (0x0302)
                Supported Version: TLS 1.0 (0x0301)
```

Le texte complet est en (en ligne sur `https://www.bortzmeyer.org/files/tls13.txt`). Notez bien que la négociation est en clair. D'autres exemples de traces TLS 1.3 figurent dans le RFC 8448.

Quelques autres articles à lire :

— L'annonce officielle de l'IETF <https://www.ietf.org/blog/tls13/>,
— Le bon résumé <https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/> de Cloudflare sur la version 1.3,
— Discussion Ycombinator sur la visibilité <https://news.ycombinator.com/item?id=16564935>,
— Liste complète des réponses aux partisans de la visbilité <https://github.com/sftcd/tinfoil>,
— Bon article historique très détaillé sur l'histoire de TLS <https://tlseminar.github.io/tls-13/>, jusqu'à la version 1.3,
— Un exemple d'un des ateliers où a été étudié la sécurité de TLS 1.3 <https://www.mitls.org/tls:div/>, à Paris en 2017, et un autre <https://www.ietfjournal.org/tron-workshop-connects-iet> à San Diego en 2016,
— Bon article sur la question des middleboxes <https://blog.cloudflare.com/why-tls-1-3-isnt-in-brow>, expliquant notamment les extensions « inutiles », mais permettant de tromper ces "*middleboxes*" pour ressembler à TLS 1.2.