RFC 8446 - TLS 1.3 : QUE FAUT-IL ATTENDRE DE CETTE NOUVELLE VERSION ?

Olivier LEVILLAIN– olivier.levillain@telecom-sudparis.eu
Maître de conférences à Télécom SudParis

TLS 1.3 a été publié en août 2018, après une longue gestation. Cette nouvelle version arrive après une suite de vulnérabilités affectant le protocole SSL/TLS. Quelles réponses TLS 1.3 apporte-t-il à la sécurité de nos communications ?

mots-clés : *SSL/TLS / TLS 1.3*

1. BREF HISTORIQUE DE SSL/TLS

SSL (*Secure Socket Layer*) est un protocole inventé par Netscape au milieu des années 1990 pour sécuriser les transactions bancaires sur HTTP. Le premier protocole publié, SSLv2, présentait de nombreux défauts de conception, et a rapidement été corrigé pour donner SSLv3.

En 1999, l'IETF reprend officiellement la maintenance du protocole, en le renommant TLS 1.0 (*Transport Layer Security*). TLS connaîtra une révision minime (TLS 1.1) en 2004, puis un premier pas timide vers la cryptographie moderne (TLS 1.2) en 2006 et finalement une révolution avec la RFC 8446, objet du présent article, publiée en août 2018.

L'objectif initial de SSLv2 était de sécuriser les communications HTTP pour permettre le commerce en ligne. Près de 25 après, TLS est devenu le protocole de sécurité le plus utilisé dans le monde. Dans son fonctionnement classique, TLS offre les propriétés suivantes : authentification du serveur à l'aide d'un certificat ; protection en confidentialité et en intégrité des données applicatives échangées ; anti-rejeu des messages.

NOTE

Pour résoudre les défauts de SSLv2, SSLv3 a rapidement été publié par Netscape (un an après), mais le mal était fait. En effet, bien que l'usage de SSLv2 soit aujourd'hui minuscule, on rencontre encore des serveurs l'utilisant… Cela donne une idée des délais de mise à jour des piles SSL/TLS sur Internet.

Une connexion TLS classique (pour les versions SSLv3 à TLS 1.2) suit le schéma donné à la figure 1. Tout d'abord, le client propose au serveur un certain nombre d'algorithmes cryptographiques et de paramètres dans le `ClientHello`. Si la proposition est compatible avec la configuration du serveur, celui-ci choisit la version du protocole et les algorithmes qui serviront ensuite (`ServerHello`).

Ensuite a lieu l'échange de clé pour que le client et le serveur construisent un secret commun ; ici, on suppose qu'il s'agit de l'échange de clé par chiffrement RSA. Le serveur envoie au client son certificat (message `Certificate`), puis le client utilise la clé publique issue du certificat pour chiffrer une valeur aléatoire et l'envoyer au serveur (`ClientKeyExchange`).

À partir de ce moment, le client et le serveur sont d'accord sur les algorithmes à utiliser, et partagent un secret. Ils peuvent donc déduire de

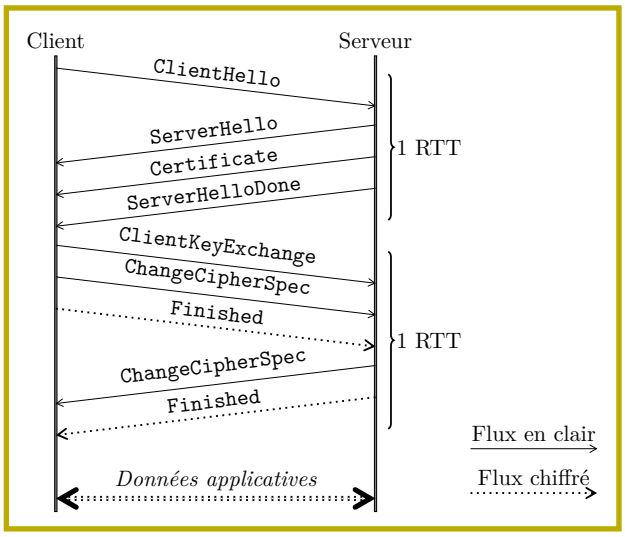

Fig. 1 : Transcript d'une connexion TLS classique (versions SSLv3 à TLS 1.2).

ce secret des clés de session et les utiliser pour protéger les flux applicatifs. Les messages **ChangeCipherSpec** activent la protection des flux, et les messages **Finished** permettent de valider l'échange (on parle de « confirmation de clé » en cryptographie).

Tout était beau et bon sur Internet avec TLS, jusqu'au début des années 2010...

2. MOTIVATIONS POUR UNE NOUVELLE VERSION

2.1 De nombreuses attaques sur SSL/TLS depuis 2011

Il n'aura pas échappé au lecteur curieux que SSL/TLS a été l'objet de nombreuses failles de sécurité ces dernières années.

Au-delà des failles d'implémentation, telles que la célèbre vulnérabilité *Heartbleed* (un bête dépassement de tampon en lecture dans le tas), des problèmes plus structurels ont été révélés. Cette section présente rapidement quelques exemples de tels problèmes. Le lecteur intéressé pourra consulter des publications complémentaires **[SSTIC12, SSTIC15, THESE (chapitre 1)]** pour plus de détails.

Tout d'abord, une bonne partie des primitives cryptographiques utilisées dans les anciennes versions de SSL/TLS s'est montrée vulnérable à des attaques diverses.

Du point de vue des primitives symétriques, le constat est assez terrible. En effet, le mode CBC, tel qu'utilisé jusqu'à TLS 1.0 utilisait le dernier bloc d'un paquet donné comme vecteur d'initialisation (IV dans le jargon cryptographique) pour le chiffrement du paquet suivant : cela rend l'IV prédictible pour un observateur réseau et conduit à une attaque à texte clair choisi, réalisable sous conditions dans un navigateur web **[BEAST]**. Au-delà de ce problème, corrigé avec TLS 1.1, l'implémentation du mode CBC dans SSL/TLS a souvent été vulnérable à des attaques de type *padding oracle* **[LUCKY13, POODLE]**. Ce problème était rendu possible par le paradigme utilisé pour combiner les primitives de chiffrement et d'intégrité dans SSL/TLS : *MAC-then-Pad-then-Encrypt*.

Avant TLS 1.2, la seule autre manière de chiffrer les paquets, si l'on souhaite éviter le mode CBC, était l'algorithme de chiffrement par flot RC4. Or des chercheurs ont montré en 2013 qu'il existait des biais statistiques exploitables dans le cadre de SSL/TLS **[BIAISRC4]**.

Un dernier exemple en cryptographie symétrique est la fonction de hachage MD5, qui est vulnérable à la recherche de collisions, et qui peut être utilisée dans TLS 1.2. Des attaques exploitant des collisions de transcript ont ainsi été réalisées **[SLOTH]**.

De même, certaines primitives asymétriques utilisées dans SSL/TLS ont montré des faiblesses exploitables en pratique. En tête de ces vulnérabilités, on peut citer l'attaque de Bleichenbacher sur le chiffrement RSA tel que décrit dans PKCS#1 v1.5. Ce mode peut en effet présenter un oracle de *padding* qui révèle de l'information lors du traitement du message chiffré. Bien que la vulnérabilité soit connue depuis 1998 **[BLEICHENBACHER]**, elle resurgit régulièrement sous des formes plus ou moins subtiles **[DROWN, ROBOT]**.

Un autre problème classique avec la cryptographie asymétrique dans TLS est l'absence de contrôle sur la taille des paramètres. Or une clé RSA de 512 bits peut être cassée facilement aujourd'hui. Il en est de même pour un échange Diffie-Hellman sur un corps fini de la même taille.

Au-delà des problèmes liés à la cryptographie, des chercheurs ont mis au jour des incohérences dans le protocole, permettant à un attaquant de créer des connexions interprétées différemment par le client et par le serveur, ce qui est normalement impossible

(voir par exemple la RFC 5746 pour l'attaque sur la renégociation). Ces vulnérabilités proviennent des mécanismes d'intégrité mis en œuvre dans la négociation, et découlent essentiellement du fait que l'ensemble du transcript des messages échangés ne soit pas couvert par une signature électronique.

Enfin, le protocole SSL/TLS, avec ses 6 versions, plus de 300 suites cryptographiques, et ses extensions diverses et variées, est un système complexe. Cela a mené à de nombreuses failles d'implémentation, notamment sur la gestion de la machine à état **[SMACK]**.

Il est cependant intéressant de constater qu'une pile TLS 1.2 à jour et correctement configurée permet de répondre à la majorité des problèmes soulevés. Malheureusement, les contraintes à appliquer sont assez drastiques, et l'ensemble reste assez fragile. Quitte à payer le prix de l'incompatibilité, autant définir proprement une nouvelle version.

NOTE

Pour être précis, la faille *Heartbleed* peut être attribuée à une spécification mal écrite. En effet, la RFC 6520 n'est pas très claire quant à la manière de découper les messages *Heartbeat* dans les *records* TLS. Or, une bonne spécification est une description limpide ne laissant pas de doute dans son interprétation !

2.2 Objectifs du groupe de travail de l'IETF

En 2013, Eric Rescorla a jeté les premières bases d'une nouvelle version de TLS dans un *draft* IETF intitulé *New Handshake Flows for TLS 1.3*. Lorsque l'idée de réviser TLS s'est faite plus concrète, le groupe de travail correspondant à l'IETF s'est fixé un certain nombre d'objectifs pour cette nouvelle mouture.

Les premiers objectifs concernent naturellement la sécurité du protocole, en prônant l'utilisation de la cryptographie à l'état de l'art. De plus, le groupe de travail avait également pour objectif d'améliorer les performances du protocole, en offrant par défaut une phase de négociation en un aller-retour entre le client et le serveur (1 RTT, pour 1 *round-trip time*), et en proposant des connexions contenant des données chiffrées dès le premier paquet (mode dit 0 RTT).

Sans détailler l'ensemble des étapes qui ont mené au standard TLS 1.3 (il y a tout de même eu 28 brouillons intermédiaires, sur une période de près de 5 ans), intéressons-nous maintenant aux grandes évolutions apportées par cette nouvelle version.

3. PRÉSENTATION DE TLS 1.3 (MODE 1 RTT)

3.1 Échange de clé initial

Avec TLS 1.3, la phase de négociation a été complètement repensée, comme le montre la figure 2. Dans son déroulement standard (1 RTT ou *Full Handshake*), elle commence par une négociation des algorithmes cryptographiques et par un échange de clé non authentifié : le client propose dans son `ClientHello` des suites cryptographiques, des groupes sur lesquels un échange de clé est envisageable, et pour certains de ces groupes, une part de secret Diffie-Hellman (pour un corps fini de module n et de générateur g, il s'agit du g^x [modulo n], avec x la clé privée), dans l'extension `KeyShare`. Si les propositions conviennent au serveur, il répond avec la suite et le groupe sélectionnés, et inclut sa part de secret.

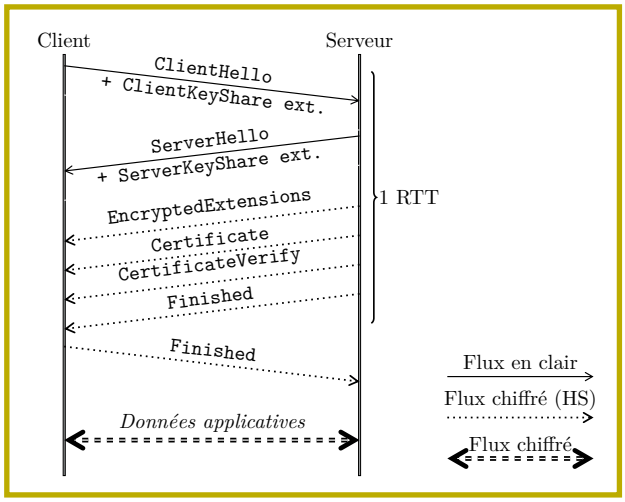

Fig. 2 : Transcript d'une connexion TLS 1.3 en mode 1 RTT (aussi appelé Full Handshake).

Cette première étape permet donc de se mettre d'accord sur les primitives utilisées, et de réaliser un échange de clé. Il est alors possible de dériver un premier jeu de clés symétriques pour protéger la suite de la négociation.

Il est important de remarquer quelques changements importants sur les messages **Hello**. Tout d'abord, les suites cryptographiques négociées avec TLS 1.3 ne concernent que la protection symétrique des messages ; celle-ci se fait uniquement à l'aide d'algorithmes de chiffrement combinés (souvent appelés AEAD) à l'état de l'art comme AES-GCM ou Chacha20-Poly1035.

Les autres éléments autrefois négociés dans la suite cryptographique sont désormais traités dans des extensions. L'extension **supported_groups** liste l'ensemble des groupes Diffie-Hellman supportés ; la liste contient des références à des groupes nommés, ce qui ne permet plus l'utilisation de groupes de taille ridicule. Enfin, l'extension **signature_algorithms** permet au client d'annoncer les algorithmes de signature que le serveur peut utiliser pour s'authentifier.

En effet, après ces premiers échanges, il est possible de protéger les communications, mais sans garantie sur la personne au bout du fil !

NOTE

L'idée d'enclencher rapidement la protection des messages échangés (qu'on retrouve dans IKEv2) permet de ne pas faire circuler en clair certaines informations comme le certificat du serveur. La protection apportée aux messages de négociation est cependant limitée, puisqu'un attaquant actif peut facilement se faire passer pour le serveur.

3.2 Authentification du serveur

Il est donc essentiel de procéder ensuite à l'authentification du serveur (l'authentification du client, optionnelle, ne sera pas traitée ici). C'est l'objet des messages **Certificate** (qui contient comme son nom l'indique la chaîne de certificats du serveur) et **CertificateVerify** (qui inclut une signature de tous les messages de négociation précédemment échangés).

En particulier, il est important de comprendre que par rapport aux versions précédentes, cette signature du serveur couvre désormais systématiquement l'ensemble du transcript, et pas uniquement certains morceaux, comme c'était le cas auparavant. Cette meilleure couverture permet de contrer certaines attaques dites *cross-protocol*, qui créent une confusion en modifiant certains paramètres, tout en réutilisant des messages signés d'une connexion à l'autre.

Le message **Certificate** a fait l'objet de quelques modifications, puisqu'il peut désormais contenir des extensions pour chaque certificat, par exemple pour embarquer des informations de révocation (réponse OCSP agrafée) ou des preuves *Certificate Transparency*.

De plus, avant TLS 1.3, l'ordre des certificats dans le message devait en théorie être strict (chaque certificat devant être vérifié à l'aide du certificat suivant) ; cette contrainte, déjà largement bafouée sur Internet, est officiellement levée avec la nouvelle version du protocole : le premier certificat doit être celui du serveur, mais pour le reste, c'est au client de se débrouiller pour retrouver un ordre plausible.

3.3 Confirmation de clé

On retrouve enfin les messages **Finished**, qui contiennent essentiellement un MAC explicite des messages précédents, le tout protégé en confidentialité et en intégrité. Cette étape, appelée « confirmation de clé », participe aux garanties de sécurité apportées par l'échange de clé.

Lorsque ces messages ont été reçus et vérifiés, le client et le serveur activent de nouvelles clés cryptographiques, qui pourront servir à protéger le contenu applicatif. Ces clés sont dérivées de l'ensemble du transcript, et incluent en particulier les messages d'authentification du serveur.

3.4 Quelques détails

Parfois, les choses ne se passent pas aussi bien : incompatibilité entre versions (le serveur va par exemple répondre avec un **ServerHello** TLS 1.2), incompatibilité sur les suites cryptographiques (le serveur va clore la connexion avec une alerte… mais cela ne devrait pas arriver, car certaines suites doivent toujours être implémentées)…

Un autre cas intéressant peut se produire : client et serveur sont d'accord sur la version et les algorithmes, mais le client n'a pas proposé de part Diffie-Hellman sur un groupe acceptable pour le serveur. Dans ce

cas, on retombe dans un mode 2 RTT, puisque le serveur répond au `ClientHello` avec un message `HelloRetryRequest` indiquant le groupe Diffie-Hellman à utiliser ; le client pourra alors produire un `ClientHello` acceptable et suivre la cinématique précédente.

Le lecteur perspicace aura par ailleurs remarqué sur le schéma précédent un message non décrit plus haut : `EncryptedExtensions`. Comme son nom l'indique, ce message est chiffré et contient un certain nombre d'extensions du protocole. Pour bénéficier de cette protection, les extensions en question ne doivent évidemment pas être utiles pour la phase d'échange de clés ; on y retrouve ainsi la réponse du serveur aux extensions *Server Name Indication* ou encore ALPN (*Application Layer Protocol Negotiation*).

4. PROPRIÉTÉS DE SÉCURITÉ DU MODE 1 RTT (LE BON)

4.1 Un grand nettoyage

La première avancée évidente avec TLS 1.3, c'est la disparition d'algorithmes, modes et options considérés comme dangereux. Comme indiqué plus haut, au niveau symétrique, seuls les algorithmes combinés (AEAD) sont conservés, ce qui met fin au mode CBC et à RC4.

Du point de vue asymétrique, le chiffrement RSA (qui n'offrait pas de *forward secrecy*, ou confidentialité persistante) disparaît : seuls les échanges de clés reposant sur Diffie-Hellman et signés par le serveur sont conservés. Ainsi, les secrets longs termes (la clé privée du serveur) et les secrets courts termes (les clés de session) sont proprement décorrélés (on obtient donc la propriété de *forward secrecy*). Au passage, l'absence de chiffrement RSA met également un terme aux oracles de *padding* à l'origine des attaques de Bleichenbacher !

NOTE

Si le chiffrement RSA n'est plus disponible dans TLS 1.3 pour l'échange de clé, il reste toujours possible d'utiliser la signature RSA pour authentifier le serveur. C'est d'ailleurs encore aujourd'hui la méthode la plus courante.

De plus, comme indiqué précédemment, les groupes sur lesquels les échanges de clés Diffie-Hellman sont réalisés appartiennent à une liste bien connue : d'une part des corps finis de taille raisonnable et générés à partir de la constante *e* (la base du logarithme népérien, voir RFC 7919 pour plus de détails), et d'autre part une liste de courbes elliptiques standard publiées par le NIST (p256r1, p384r1 et p521r1) ou par des chercheurs en cryptographie (x255219 et x448). Ce n'est plus une donnée imposée par le serveur, sans que le client puisse facilement en vérifier la qualité.

De manière plus anecdotique, la renégociation et la compression, qui ont été à l'origine de failles, ont été retirées du standard.

4.2 Un protocole prouvé

Au-delà de ce grand ménage, le standard a pris en compte certaines modifications pour rendre la preuve du protocole faisable. En effet, TLS 1.3 a été l'occasion pour de nombreuses équipes de recherche à travers le monde d'apporter des preuves sur les propriétés de sécurité du protocole, en parallèle de sa spécification.

Parmi les changements utiles à une meilleure preuve du protocole, la modification la plus importante est sans doute la fonction de dérivation de secrets. À la place d'un HMAC jusqu'à TLS 1.2, TLS 1.3 utilise la construction HKDF (*HMAC-based Extract-and-Expand Key Derivation Function*, voir RFC 5869), qui offre des propriétés intéressantes.

Par ailleurs, l'arbre de dérivation des clés a été modifié, afin de décorréler les secrets de session d'une connexion des secrets de session après une reprise de session.

Enfin, le schéma de signature utilisé avec RSA est désormais PSS (*Probabilistic Signature Scheme*, défini dans PKCS#1 v2.1), qui offre de meilleures garanties cryptographiques.

4.3 Simplification de la machine à état

Si on ne considère que le mode 1 RTT, la machine à état du protocole a été significativement simplifiée. De plus, au-delà de la négociation initiale,

les messages de type **Handshake** qui peuvent être échangés sont rares. Le serveur peut émettre des messages **NewSessionTicket** pour permettre au client une reprise de session ultérieure. Le serveur peut demander au client de s'authentifier ; on parle de *Post Handshake Authentication*, et cette fonctionnalité peut poser des problèmes (voir section 5.2). Enfin, le client ou le serveur peuvent demander à rafraîchir les clés symétriques utilisées pour protéger les messages (**KeyUpdate**).

De plus, la reprise de session a été fusionnée avec l'authentification par clé prépartagée (PSK, *Pre-Shared Key*) : les tickets fournis par le serveur sont ensuite utilisés comme des PSK. Les mécanismes de reprise de session précédents ont donc été supprimés au profit d'une fonctionnalité plus simple.

Tous ces changements permettent de produire des implémentations de TLS 1.3 beaucoup plus simples, si on s'en tient au mode 1 RTT et qu'on interdit l'authentification tardive du client. Et globalement, TLS 1.3 contraint de la sorte est propre, à l'état de l'art, et essentiellement prouvé.

Cependant, la spécification ne s'arrête pas là…

5. FONCTIONNALITÉS PLUS DISCUTABLES (LA BRUTE)

5.1 0-RTT

Avec TLS 1.3, un nouveau mode est apparu, permettant l'échange de données applicatives dès le premier paquet TLS. Il s'agit du mode 0 RTT, qui peut être utilisé dans le cas d'une reprise de session, et à condition que les tickets de session fournis par le serveur autorisent ce mode.

Concrètement l'utilisation d'une PSK dans le cas d'une reprise de session permet dès le premier paquet de disposer d'une clé commune. Cette clé est donc utilisée pour protéger des paquets applicatifs juste après le **ClientHello**.

L'avantage évident de ce mode est d'accélérer l'échange des données applicatives dans une connexion TLS. Il est cependant important de bien comprendre qu'a priori, ces données peuvent être rejouées par un attaquant, puisqu'il lui suffit de rejouer la première salve de messages envoyés par le client. L'impact de ce rejeu dépend bien entendu de l'application transportée par TLS, mais il est essentiel de bien comprendre que les garanties de sécurité apportées par le mode 0 RTT dans TLS 1.3 sont plus faibles que celles apportées (et prouvées) pour le mode 1 RTT.

Pour éviter les attaques par rejeu, la première solution (qui a sans surprise la préférence de l'auteur…) est de désactiver complètement 0 RTT.

La méthode prônée par les partisans du mode 0 RTT est de ne l'utiliser qu'avec des protocoles qui ont défini un « profil ». L'idée étant d'expliquer dans quelles situations il est possible de l'utiliser. Avec HTTP, une version naïve serait de n'autoriser que des requêtes idempotentes dans les données 0 RTT… ce qui semble assez audacieux quand on sait ce que peut contenir une requête GET, officiellement idempotente. Cette solution est vouée à l'échec puisqu'il sera dur de résister à ce gain en rapidité…

Après de nombreux débats sur la liste du groupe de travail, une section a été ajoutée pour décrire une solution pour détecter et rejeter les rejeux… mais au prix d'un état côté serveur. Cette solution a très peu de chances d'être mise en place, car l'utilisation de tickets pour la reprise de session repose au contraire sur l'idée de ne pas avoir d'état côté serveur (le ticket contient l'ensemble des paramètres chiffrés par une clé connue du serveur). Cette difficulté est amplifiée dans le cadre d'un service offert par une multitude de serveurs frontaux, qui peuvent partager les clés pour déchiffrer les tickets…

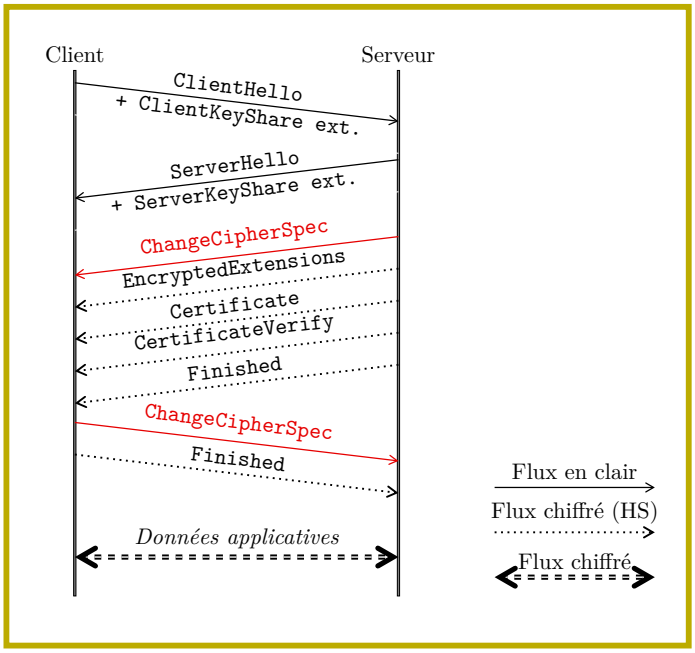

Fig. 3 : Transcript réel d'une connexion TLS 1.3 en mode 1 RTT (avec les messages ChangeCipherSpec artificiels).

5.2 Authentification tardive du client

Le serveur peut demander au client de s'authentifier lors de la négociation initiale, à l'aide d'un message **`CertificateRequest`**. Le client répondra alors avec des messages **`Certificate`** et **`CertificateVerify`**, comme le serveur.

Cependant, certains services nécessitent que le client s'authentifie après la négociation initiale (par exemple en réponse à une requête vers une ressource sensible). TLS 1.3 permet un tel mécanisme, n'importe quand après la négociation initiale, et un nombre arbitraire de fois… La machine à état induite par l'authentification tardive du client n'est donc pas bornée en pratique, ce qui peut être perçu comme une régression.

En pratique, suite à des discussions au sein du groupe de travail, un client peut annoncer qu'il ne supporte pas ce mécanisme, afin qu'il puisse conserver un automate simple. Comme la fonctionnalité devrait être peu utilisée en pratique, c'est sans doute une bonne idée de la désactiver.

6. LES AJOUTS TARDIFS POUR FAIRE FACE AUX MIDDLEBOXES (LE TRUAND)

6.1 Ajout de messages et champs inutiles

Même en désactivant les fonctionnalités douteuses évoquées dans la section 5, un échange TLS 1.3 réel se présente généralement plutôt comme le montre la figure 3. Les messages **`ChangeCipherSpec`**, qui déclenchaient jusqu'à TLS 1.2 le changement des clés utilisées pour la protection des paquets, qui avaient disparu assez tôt dans le travail sur TLS 1.3, sont réapparus !

La raison de cette présence est expliquée dans la section D.4 de la RFC, intitulée *Middlebox Compatibility Mode*. Pour que les communications TLS 1.3 ne soient pas trop souvent interrompues par des équipements réseau indiscrets (ou avec des piles TLS en carton), un mode de compatibilité a été prévu pour que TLS 1.3 ressemble au maximum à TLS 1.2.

Bien entendu, ces messages **`ChangeCipherSpec`** n'ont aucune signification et ne sont pas signés dans les transcripts, ce qui n'est pas pour rassurer les observateurs se souvenant des vulnérabilités mettant en œuvre ces messages…

Dans le même objectif de ressemblance de TLS 1.3 avec les versions précédentes, certains champs du message **`ServerHello`** ont finalement été conservés (**`CompressionMethod`** par exemple).

6.2 Impact réel sur les piles

Concrètement, ces modifications de dernières minutes peuvent paraître anecdotiques, et c'est certainement ce que pensent les développeurs des piles TLS. Cependant, ces verrues poseront à terme des problèmes, puisqu'il faut à la fois les ignorer dans la machine à état du protocole, tout en les acceptant sur le fil. Il y a fort à parier que ces messages sordides seront tôt ou tard à l'origine de comportements intéressants des piles TLS 1.3.

De plus, ces adaptations du protocole pour tolérer des équipements réseau perturbateurs risquent de durer longtemps, même si elles n'apparaissent que dans une annexe de la RFC.

CONCLUSION

Comme nous l'avons vu, TLS 1.3 est une révolution du protocole SSL/TLS, qui n'avait pas été fondamentalement modifié depuis 2006. Il faut reconnaître que cette nouvelle version est un pas en avant du point de vue cryptographique et méthodes formelles. Cependant, les concepteurs du protocole sont tombés dans un travers courant dans l'ingénierie logicielle. En effet, le standard contient des fonctionnalités complexes (mode 0 RTT, authentification tardive du client), ainsi que des bricolages pour s'accommoder d'équipements réseau déficients. On ne peut que conseiller l'utilisation d'une version du protocole n'incluant ni ces fonctionnalités ni ces artifices.

Par ailleurs, il faut garder à l'esprit qu'il faudra encore vivre avec les versions précédentes du protocole pendant encore quelques années (décennies ?).

Enfin, prouver un protocole et utiliser les bonnes primitives cryptographiques n'est que la partie émergée de l'iceberg. Il est en effet nécessaire de tester et de prouver les implémentations concrètes de la spécification !

REMERCIEMENTS

Merci à Jean-Sylvain pour sa relecture attentive. ■

RÉFÉRENCES

[BEAST] T. Duong et J. Rizzo, Here Come The XOR Ninjas, Ekoparty 2011.

[BIAISRC4] N. J. AlFardan et al., On the Security of RC4 in TLS, Usenix Security 2013.

[BLEICHENBACHER] D. Bleichenbacher, Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1, CRYPTO' 98.

[DROWN] N. Aviram, DROWN: Breaking TLS with SSLv2, Usenix Security 2016 : **https://drownattack.com/**.

[LUCKY13] N. AlFardan et K. Paterson, Lucky 13: Breaking the TLS and DTLS Record Protocols, S&P 2013.

[THESE] O. Levillain, A study of the TLS ecosystem : **https://paperstreet.picty.org/yeye/2016/phdthesis-Levillain16/**.

[POODLE] B. Möller, T. Duong et K. Kotowicz, This POODLE Bites - Exploiting The SSL 3.0 Fallback, Google Security Advisory 2014 : **https://www.openssl.org/~bodo/ssl-poodle.pdf**.

[ROBOT] H. Böck, J. Somorovsky et C. Young, Return Of Bleichenbacher's Oracle Threat, Usenix Security 2018 : **https://robotattack.org/**.

[SLOTH] K. Bhargavan et al., Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH, NDSS 2016 : **https://mitls.org/pages/attacks/SLOTH**.

[SMACK] B. Beurdouche et al., A Messy State of the Union: Taming the Composite State Machines of TLS, S&P 2015 : **https://mitls.org/pages/attacks/SMACK**.

[SSTIC12] O. Levillain, SSL/TLS : état des lieux et recommandations : **https://www.sstic.org/2012/presentation/ssl_tls_soa_recos/**.

[SSTIC15] O. Levillain, SSL/TLS, 3 ans plus tard : **https://www.sstic.org/2015/presentation/ssltls_soa_reloaded/**.