RFC 9001 : Using TLS to Secure QUIC

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 28 mai 2021

Date de publication du RFC : Mai 2021

https://www.bortzmeyer.org/9001.html

Le protocole de transport QUIC <https://www.bortzmeyer.org/quic.html> est toujours sécurisé par la cryptographie. Il n'y a pas de communication en clair avec QUIC. Cette sécurisation se fait actuellement par TLS mais QUIC utilise TLS d'une manière un peu spéciale, documentée dans ce RFC.

Fini d'hésiter entre une version avec TLS ou une version sans. Avec QUIC <https://www.bortzmeyer.org/quic.html>, c'est forcément chiffré et, pour l'instant, avec TLS (dans le futur, QUIC pourra utiliser d'autres protocoles, mais ce n'est pas encore défini). QUIC impose en prime au minimum TLS 1.3 (RFC 8446 [1]), qui permet notamment de diminuer la latence <https://www.bortzmeyer.org/latence.html>, la communication pouvant s'établir dans certains cas dès le premier paquet. Petit rappel de TLS, pour commencer : TLS permet d'établir un canal sûr, fournissant authentification du serveur et confidentialité de la communication, au-dessus d'un media non-sûr (l'Internet). TLS est normalement constitué de deux couches, la couche des enregistrements ("*record layer*") et celle de contenu ("*content layer*"), qui inclut notamment le mécanisme de salutation initial. On établit une session avec le protocole de salutation ("*handshake protocol*"), puis la couche des enregistrements chiffre les données ("*application data*") avec les clés issues de cette salutation. Les clés ont pu être créées par un échange Diffie-Helman, ou bien être pré-partagées (PSK, pour "*Pre-Shared Key*"), ce dernier cas permettant de démarrer la session immédiatement (« 0-RTT »).

QUIC utilise TLS d'une façon un peu spéciale. Certains messages TLS n'existent pas comme `ChangeCipherSpec` ou `KeyUpdate` (QUIC ayant ses propres mécanismes pour changer la cryptographie en route, cf. section 6), et, surtout, la couche des enregistrements disparait, QUIC faisant le chiffrement selon son format, mais avec les clés négociées par TLS.

La poignée de mains qui établit la session TLS peut donc se faire de deux façons :

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc8446.txt`

— « 1-RTT » où le client et le serveur peuvent envoyer des données après un aller-retour (RTT). Rappelez-vous que c'est ce temps d'aller-retour qui détermine la latence <https://www.bortzmeyer.org/latence.html>, donc la « vitesse » perçue par l'utilisateur, au moins pour les transferts de faible taille (comme le sont beaucoup de ressources HTML).
— « 0-RTT », où le client peut envoyer des données dès le premier datagramme transmis. Cette façon de faire nécessite que client et serveur se soient parlés avant, et que le client ait stocké un jeton généré par le serveur (sept jours maximum, dit le RFC 8446), qui permettra au serveur de trouver tout de suite le matériel cryptographique nécessaire. Attention, le 0-RTT ne protège pas contre le rejeu, mais ce n'est pas grave pour des applications comme HTTP avec la méthode `GET`, qui est idempotente. Et le 0-RTT pose également problème avec la PFS (la confidentialité même en cas de compromission ultérieure des clés).

La section 3 du RFC fait un tour d'horizon général du protocole TLS tel qu'utilisé par QUIC. Comme vu plus haut, la couche des enregistrements ("*record layer*") telle qu'utilisée avec TCP n'est plus présente, les messages TLS comme `Handshake` et `Alert` sont directement transportés sur QUIC (qui, contrairement à TCP, permet d'assurer confidentialité et intégrité). Avec TCP, la mise en couches était stricte, TLS étant entièrement au-dessus de TCP, avec QUIC, l'intégration est plus poussée, QUIC et TLS coopèrent, le premier chiffrant avec les clés fournies par le second, et QUIC transportant les messages de TLS. De même, quand on fait tourner un protocole applicatif sur QUIC, par exemple HTTP/3 (RFC 9113), celui-ci est directement placé sur QUIC, TLS s'effaçant complètement. Les applications vont donc confier leurs données à QUIC, pas à TLS. En simplifiant (beaucoup…), on pourrait dire que TLS ne sert qu'au début de la connexion. Pour citer Radia Perlman, « "*It is misleading to regard this as a specification of running QUIC over TLS. It is related to TLS in the same way that DTLS is related to TLS : it imports much of the syntax, but there are many differences and its security must be evaluated largely independently. My initial reaction to this spec was to wonder why it did not simply run QUIC over DTLS . I believe the answer is that careful integration improves the performance and is necessary for some of the address agility/transition design.*" ».

La section 4 explique plus en détail comment les messages TLS sont échangés via QUIC. Les messages cryptographiques sont transportés dans des trames de type `CRYPTO`. Par exemple, le `ClientHello` TLS sera dans une trame `CRYPTO` elle-même située dans un paquet QUIC de type `Initial`. Les `Alert` sont dans des trames `CONNECTION_CLOSE` dont le code d'erreur est l'alerte TLS. Ce sont les seuls messages que QUIC passera à TLS, il fait tout le reste lui-même.

On a vu que le principal travail de TLS est de fournir du matériel cryptographique à QUIC. Plus précisément, TLS fournit, après sa négociation avec son pair :

— Un secret, d'où la clé sera dérivée,
— une fonction de chiffrement intègre, par exemple `AEAD_AES_128_GCM` (AES avec GCM) ou `AEAD_CHACHA20_POLY1305` (ChaCha20, RFC 8439),
— une fonction de dérivation des clés.

QUIC se servira de tout cela pour chiffrer.

QUIC impose une version minimale de TLS : la 1.3, normalisée dans le RFC 8446. Les versions ultérieures sont acceptées mais elles n'existent pas encore.

Ah et, comme toujours avec TLS, le client doit authentifier le serveur, typiquement via son certificat. Le serveur ne doit pas utiliser les possibilités TLS de ré-authentification ultérieure (message `CertificateRequest`) car le multiplexage utilisé par QUIC empêcherait de corréler cette demande d'authentification avec une requête précise du client.

Outre le « 0-RTT » de QUIC, qui permet au client d'envoyer des données applicatives dès le premier paquet, QUIC+TLS fournit un autre mécanisme pour gagner du temps à l'établissement de la connexion, la reprise de session ("*session resumption*", RFC 8446, section 2.2). Si le client a enregistré les informations nécessaires depuis une précédente session avec ce serveur, il peut attaquer directement avec un `NewSessionTicket` dans une trame `CRYPTO` et abréger ainsi l'établissement de session TLS.

Les erreurs TLS, comme `bad_certificate`, `unexpected_message` ou `unsupported_extension`, sont définies dans le RFC 8446, section 6. Dans QUIC, elles sont transportées dans des trames de type `CONNECTION_CLOSE`, et mises dans le champ d'erreur (`Error Code`, RFC 9000, section 19.19). Notez que ces trames mènent forcément à la coupure de toute la session QUIC, et il n'y a donc pas moyen de transporter un simple avertissement TLS.

Bien, maintenant qu'on a vu le rôle de TLS, comment QUIC va-t-il utiliser les clés pour protéger les paquets ? La section 5 répond à cette question. QUIC va utiliser les clés fournies par TLS (je simplifie, car QUIC effectue quelques dérivations avant) comme clés de chiffrement intègre (RFC 5116). Il utilisera l'algorithme de chiffrement symétrique indiqué par TLS. Tous les paquets ne sont pas protégés (par exemple ceux de négociation de version, inutiles pour l'instant puisque QUIC n'a qu'une version, ne bénéficient pas de protection puisqu'il faudrait connaitre la version pour choisir les clés de protection). Le cas des paquets `Initial` est un peu plus subtil puisqu'ils sont chiffrés, mais avec une clé dérivée du "*connection ID*", qui circule en clair. Donc, en pratique, seule leur intégrité est protégée, par leur confidentialité (cf. section 7 pour les conséquences).

J'ai dit que QUIC n'utilisait pas directement les clés fournies par TLS. Il applique en effet une fonction de dérivation, définie dans la section 7.1 du RFC 8446, elle-même définie à partir des fonctions du RFC 5869.

Il existe plein de pièges et de détails à prendre en compte quand on met en œuvre QUIC+TLS. Par exemple, en raison du réordonnancement des datagrammes dans le réseau, et des pertes de datagrammes, un paquet chiffré peut arriver avant le matériel cryptographique qui permettrait de le déchiffrer, ou bien avant que les affirmations du pair aient été validées. La section 5.7 du RFC explique comment gérer ce cas (en gros, jeter les paquets qui sont « en avance », ou bien les garder pour déchiffrement ultérieur mais ne surtout pas tenter de les traiter). Autre piège, QUIC ignore les paquets dont la vérification d'intégrité a échoué, alors que TLS ferme la connexion. Cela a pour conséquences qu'avec QUIC un attaquant peut essayer plusieurs fois. Il faut donc compter les échecs et couper la connexion quand un nombre maximal a été atteint (section 6.6). Bien sûr, pour éviter de faciliter une attaque par déni de service (où l'attaquant enverrait plein de mauvais paquets dans l'espoir de fermer la connexion), ces limites doivent être assez hautes ($2\hat{}\{\}23$ paquets pour `AEAD_AES_128_GCM`), voir « "*Limits on Authenticated Encryption Use in TLS*" `<http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>` » ou « "*Robust Channels : Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3*" `<https://eprint.iacr.org/2020/718>` », ainsi que l'annexe B du RFC.

Encore question détails subtils, la poignée de mains de TLS n'est pas tout à fait la même quand elle est utilisée par QUIC (section 8). Ainsi, ALPN doit être utilisé et avec succès, autrement on raccroche avec l'erreur `no_application_protocol`.

Le but de TLS est de fournir de la sécurité, notamment confidentialité et authentification, donc il est recommandé de bien lire la section 9, qui traite de la sécurité de l'ensemble du RFC. Ainsi, si on utilise les tickets de session de TLS (RFC 8446, section 4.6.1), comme ils sont transmis en clair, ils peuvent permettre à un observateur de relier deux sessions, même si les adresses IP sont différentes.

Le « 0-RTT » est formidable pour diminuer la latence, mais il diminue aussi la sécurité : il n'y a pas de protection contre le rejeu. Si QUIC lui-même n'est pas vulnérable au rejeu, l'application qui travaille au-dessus de QUIC peut l'être. Prenez un protocole applicatif qui aurait des services comme, en HTTP, « envoyez-moi une pizza » (sans doute avec la méthode `POST`), on voit bien que le rejeu serait problématique. Bref, les applications qui, contrairement au protocole QUIC, ne sont pas idempotentes, ont tout intérêt à désactiver le 0-RTT.

QUIC tournant sur UDP, qui ne protège pas contre l'usurpation d'adresse IP <https://www.bortzmeyer.org/usurpation-adresse-ip.html>, il existe en théorie un risque d'attaque par réflexion, avec amplification <https://www.bortzmeyer.org/attaques-reflexion.html>. Par exemple, la réponse à un `ClientHello` peut être bien plus grande que le `ClientHello` lui-même. Pour limiter les risques, QUIC impose que le premier paquet du client ait une taille minimale (pour réduire le facteur d'amplification), en utilisant le remplissage, et que le serveur ne réponde pas avec plus de trois fois la quantité de données envoyée par le client, tant que l'adresse IP de celui-ci n'a pas été validée.

Plus sophistiquées sont les attaques par canal auxiliaire. Par exemple, si une mise en œuvre de QUIC jette trop vite les paquets invalides, un attaquant qui mesure les temps de réaction pourra en déduire des informations sur ce qui n'allait pas exactement dans son paquet. Il faut donc que toutes les opérations prennent un temps constant.

Et puis, bien sûr, comme tout protocole utilisant la cryptographie, QUIC+TLS a besoin d'un générateur de nombres aléatoires correct (cf. RFC 4086).

Question mise en œuvre, notez qu'on ne peut pas forcément utiliser une bibliothèque TLS quelconque. Il faut qu'elle permette de séparer signalisation et chiffrement et qu'elle permette d'utiliser QUIC comme transport. (Et il n'y a pas d'API standard pour cela <https://daniel.haxx.se/blog/2019/01/21/quic-and-missing-apis/>.) C'est par exemple le cas de la bibliothèque picotls <https://github.com/h2o/picotls>. Pour OpenSSL, il faut attendre <https://www.openssl.org/blog/blog/2020/02/17/QUIC-and-OpenSSL/> (un patch existe <https://github.com/openssl/openssl/pull/8797>) et cela bloque parfois l'intégration <https://bugs.debian.org/932151> de certains logiciels.

Et question tests, à ma connaissance, on ne peut pas actuellement utiliser `openssl s_client` ou `gnutls-cli` avec un serveur QUIC. Même problème avec le fameux site de test TLS .

Pour terminer, voici l'analyse d'une communication QUIC+TLS, analyse faite avec Wireshark. D'abord, le premier paquet, de type QUIC `Initial`, qui contient le `ClientHello` dans une trame de type `CRYPTO` :

```
QUIC IETF
    1... .... = Header Form: Long Header (1)
    .1.. .... = Fixed Bit: True
    ..00 .... = Packet Type: Initial (0)
    .... 00.. = Reserved: 0
    .... ..11 = Packet Number Length: 4 bytes (3)
    Version: 1 (0x00000001)
    Destination Connection ID Length: 8
    Destination Connection ID: 345d144296b90cff
...
    Length: 1226
    Packet Number: 0
    TLSv1.3 Record Layer: Handshake Protocol: Client Hello
        Frame Type: CRYPTO (0x0000000000000006)
        Offset: 0
        Length: 384
        Crypto Data
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Length: 380
            ...
            Extension: quic_transport_parameters (len=85)
                Type: quic_transport_parameters (57)
                Length: 85
                Parameter: initial_max_stream_data_bidi_local (len=4) 2097152
```

```
            Type: initial_max_stream_data_bidi_local (0x05)
            Length: 4
            Value: 80200000
            initial_max_stream_data_bidi_local: 2097152
...
```

Dans les extensions TLS, notez l'extension spécifique à QUIC, `quic_transport_parameters`. QUIC « abuse » de TLS en s'en servant pour emballer ses propres paramètres. (La liste de ces paramètres de transport figure dans un registre IANA <https://www.iana.org/assignments/quic/quic.xml#quic-transport>.)

La réponse à ce paquet `Initial` contiendra le `ServerHello` TLS. La poignée de mains se terminera avec les paquets QUIC de type `Handshake`. TLS ne servira plus par la suite, QUIC chiffrera tout seul.