RFC 9147 : The Datagram Transport Layer Security (DTLS) Protocol Version 1.3

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 22 avril 2022

Date de publication du RFC : Avril 2022

https://www.bortzmeyer.org/9147.html

Pour sécuriser vos applications TCP sur l'Internet, vous utilisez TLS, pas vrai ? Et si vos applications préfèrent UDP, par exemple pour faire de la voix sur IP ? La solution est alors DTLS, dont la nouvelle version, la 1.3, est normalisée dans ce RFC 9147 [1]. Elle a vocation à remplacer l'ancienne version 1.2, qui était dans le RFC 6347.

DTLS fournit normalement les mêmes services de sécurité que TLS, notamment la confidentialité (via un chiffrement du trafic) et l'authentification (via une signature). Les seuls services TLS que DTLS ne peut pas rendre sont la protection de l'ordre des messages (ce qui est logique pour UDP) et, selon les options choisies, la protection contre le rejeu. DTLS a été conçu pour être aussi proche que possible de TLS, pour pouvoir réutiliser le code et s'appuyer sur des propriétés de sécurité déjà établies. DTLS 1.0 et 1.2 (il n'y a jamais eu de 1.1) avaient été normalisés sous forme de différences avec la version de TLS correspondante. De même, DTLS 1.3, objet de notre RFC, est défini en décrivant les différences avec TLS 1.3. Il faut donc lire le RFC 8446 avant. À propos de lectures, il faut aussi lire le RFC 9146, pour le concept d'identificateur de connexion ("*Connection ID*").

La section 3 du RFC pose le problème : on veut un truc comme TLS mais qui marche sur un service de datagrammes, en général UDP (RFC 768). Un service de datagrammes ne garantit pas l'ordre d'arrivée, ni même que les datagrammes arrivent. DTLS ajoute de la sécurité au service de datagrammes mais ne change pas ses propriétés fondamentales : une application qui utilise DTLS ne doit pas s'étonner que les messages n'arrivent pas tous, ou bien arrivent dans le désordre. C'est un comportement connu des applications de diffusion vidéo ou de jeu en ligne, qui préfèrent sauter les parties manquantes plutôt que de les attendre.

Pourquoi un protocole distinct, DTLS, plutôt que de reprendre TLS ? C'est parce que TLS ne marche que si le service de transport sous-jacent garantit certaines propriétés qu'UDP ne fournit pas. TLS a besoin de :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9147.txt

— Une numérotation croissante des données. Ces numéros ne sont pas explicites dans TLS, ils sont déduits de l'ordre d'arrivée. DTLS les rend explicites.
— L'établissement d'une association TLS nécessite un échange fiable, où les messages arrivent dans l'ordre d'émission. Là aussi, DTLS doit ajouter des numéros de séquence explicites pour pouvoir remettre dans l'ordre les messages d'établissement d'association (cf. section 4.2).
— Certains messages TLS n'ont pas d'accusé de réception explicite, TLS compte sur les messages suivants pour savoir si son message a été reçu. Avec DTLS, ces messages doivent avoir un accusé de réception puisque la couche transport ne nous dit plus si le message a été perdu.
— Certains messages TLS peuvent être de grande taille (chaines de certificats, par exemple), supérieure à celle du datagramme (qui est typiquement de 1 500 octets) et DTLS doit donc être capable de les fragmenter et de les réassembler (la fragmentation IP étant hélas souvent bloquée par des équipements réseau mal programmés et/ou mal configurés).
— Tout protocole utilisant les datagrammes doit se méfier des attaques par réflexion, où l'attaquant ment sur son adresse IP. TCP protège contre cela, mais DTLS, ne pouvant compter sur TCP, doit avoir son propre mécanisme de test de réversibilité <https://www.bortzmeyer.org/returnability.html>.

La section 4 présente la couche Enregistrements de DTLS, c'est-à-dire le format des paquets sur le réseau. Les messages sont transportés dans des enregistrements ("*record*") TLS, et il peut y avoir plusieurs messages dans un seul datagramme UDP. Le format des messages est différent de celui de TLS (ajout d'un numéro de séquence) et de celui de DTLS 1.2 :

```
struct {
           ContentType type;
           ProtocolVersion legacy_record_version;
           uint16 epoch = 0
           uint48 sequence_number;
           uint16 length;
           opaque fragment[DTLSPlaintext.length];
       } DTLSPlaintext;

struct {
           opaque unified_hdr[variable];
           opaque encrypted_record[length];
} DTLSCiphertext;
```

La première structure est en clair, la seconde contient des données chiffrées. Un message dans un datagramme est, soit un `DTLSPlaintext` (ils sont notamment utilisés pour l'ouverture de connexion, quand on ne connait pas encore le matériel cryptographique à utiliser) ou bien un `DTLSCiphertext`. La partie chiffrée du `DTLSCiphertext` est composée d'un :

```
struct {
            opaque content[DTLSPlaintext.length];
            ContentType type;
            uint8 zeros[length_of_padding];
} DTLSInnerPlaintext;
```

Comme pour TLS 1.3, le champ `legacy_record_version` est ignoré (il n'est là que pour tromper les "*middleboxes*"). Le `unified_hdr` contient notamment l'identificateur de connexion ("*Connection ID*"), concept expliqué dans le RFC 9146 (comme avec QUIC <https://www.bortzmeyer.org/quic.html>, ils peuvent augmenter la traçabilité <https://www.bortzmeyer.org/quic-tracking.html>). Les détails de la structure des messages figurent dans l'annexe A du RFC.

À l'arrivée d'un datagramme, c'est un peu compliqué, vu la variété de formats. La section 4.1 suggère un mécanisme de démultiplexage, permettant de déterminer rapidement si le message était du `DTLSPlaintext`,

du `DTLSCipherext`, ou une erreur. Un datagramme invalide doit être silencieusement ignoré (c'est peut-être une tentative d'un méchant pour essayer d'injecter des données ; couper la connexion au premier datagramme invalide ouvrirait une facile voie d'attaque par déni de service).

Qui dit datagramme dit problèmes de MTU, une des plaies de l'Internet. Normalement, c'est l'application qui doit les gérer, après tout le principe d'un service de datagrammes, c'est que l'application fait tout. Mais DTLS ne lui facilite pas la tâche, car le chiffrement augmente la taille des données, « dans le dos » de l'application. Et, avant même que l'application envoie ses premières données, la poignée de main DTLS peut avoir des datagrammes dépassant la MTU, par exemple en raison de la taille des certificats. Et puis dans certains cas, le système d'exploitation ne transmet pas à l'application les signaux indispensables, comme les messages ICMP "*Packet Too Big*". Bref, pour aider, DTLS devrait transmettre à l'application, s'il la connait, la PMTU (la MTU du chemin complet, que la couche Transport a peut-être indiqué à DTLS). Et DTLS doit gérer la découverte de la PMTU tout seul pour la phase initiale de connexion.

La section 5 du RFC décrit le protocole d'établissement de l'association entre client et serveur (on peut aussi dire connexion, si on veut, mais pas session, ce dernier terme devrait être réservé au cas où on reprend une même session sur une connexion différente). Il est proche de celui de TLS 1.3 mais il a fallu ajouter tout ce qu'UDP ne fournit pas, la détection de la MTU ("*Path MTU*", la MTU du chemin complet), des accusés de réception explicites et la gestion des cas de pertes de paquets. Voici l'allure d'un message DTLS d'établissement de connexion :

```
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;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    uint16 message_seq;        /* DTLS-required field */
    uint24 fragment_offset;    /* DTLS-required field */
    uint24 fragment_length;    /* DTLS-required field */
    select (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;
    } body;
} Handshake;
```

Le message `ClientHello`, comme en TLS 1.3, a un champ `legacy_version` qui sert à faire croire aux "*middleboxes*" (et aux serveurs bogués qui ne gèrent pas correctement la négociation de version) qu'il s'agit d'un TLS ancien.

Un établissement de connexion DTLS typique est :
— Le client envoie un `ClientHello`,
— le serveur répond avec un `HelloRetryRequest` qui contient le "*cookie*",
— le client renvoie le `ClientHello`, cette fois avec le "*cookie*" (le serveur est désormais certain que le client ne ment pas sur son adresse IP),
— le serveur peut alors transmettre son `ServerHello`,
— le client peut alors envoyer un message `Finished` et transmettre des données.

Il existe également, comme en TLS 1.3, un mode rapide, quand le client a déjà contacté ce serveur et obtenu du matériel cryptographique qu'il peut ré-utiliser. (C'est ce qu'on appelle aussi le « 0-RTT » ou la « reprise de session », et ça existe aussi dans des protocoles comme QUIC <`https://www.bortzmeyer.org/quic.html`>.) Tous les paquets pouvant se perdre, DTLS doit avoir un mécanisme de réémission.

Les risques d'attaques par déni de service sont très élevés dans le cas où on utilise des datagrammes. Un méchant peut envoyer des demandes de connexion répétées, pour forcer le serveur à faire des calculs cryptographiques et surtout à allouer de la mémoire pour les connexions en attente. Pire, il peut utiliser un serveur DTLS pour des attaques par réflexion où le méchant ment sur son adresse IP pour que le serveur dirige ses réponses vers une victime innocente. Les attaques par réflexion sont encore pires lorsqu'elles sont combinées à une amplification, quand la réponse est plus grosse que la question.

Pour éviter ces attaques, DTLS reprend, comme vu plus haut, le principe des "*cookies*" sans état de Photuris (RFC 2522) et IKE (RFC 7296).

L'extension `connection_id` (cf. RFC 9146) peut être mise dans le `ClientHello`. Notez qu'une nouveauté par rapport au RFC 9146 est la possibilité de changer les identifiants de connexion pendant une association.

En section 7, une nouveauté de DTLS 1.3, et qui n'a pas d'équivalent dans TLS, les accusés de réception (`ACK`), nécessaires puisqu'on fonctionne au-dessus d'un service de datagrammes, qui ne garantit pas l'arrivée de tous les paquets. Un `ACK` permet d'indiquer les numéros de séquence qu'on a vu. Typiquement, on envoie des `ACK` quand on a l'impression que le partenaire est trop silencieux (ce qui peut vouloir dire que ses messages se sont perdus). Ces accusés de réception sont facultatifs, on peut décider que la réception des messages (par exemple un `ServerHello` quand on a envoyé un `ClientHello`) vaut accusé de réception. Ils servent surtout à exprimer son impatience « allô ? tu ne dis rien ? »

Dans sa section 11, notre RFC résume les points importants, question sécurité (en plus de celles communes à TLS et DTLS, qui sont traitées dans le RFC 8446). Le principal risque, qui n'a pas vraiment d'équivalent dans TLS, est celui de déni de service via une consommation de ressources déclenchée par un partenaire malveillant. Les "*cookies*" à la connexion ne sont pas obligatoires mais fortement recommandés. Pour être vraiment sûrs, ces "*cookies*" doivent dépendre de l'adresse IP du partenaire, et d'une information secrète pour empêcher un tiers de les générer.

À noter d'autre part que, si DTLS garantit plusieurs propriétés de sécurité identiques à celle de TLS (confidentialité et authentification du serveur), il ne garantit pas l'ordre d'arrivée des messages (normal, on fait du datagramme…) et ne protège pas parfaitement contre le rejeu (sections 3.4 et 4.5.1 du RFC si vous voulez le faire).

Les sections 12 et 13 résument les principaux changements depuis DTLS 1.2 (seulement les principaux car DTLS 1.3 est très différent de 1.2) :
— n'accepte désormais que du chiffrement intègre,
— le mécanisme de reprise de session est différent, et la terminologie a changé (on parle désormais de PSK, "*Pre-Shared Key*" pour désigner les données stockées chez le client),

— la négociation de version a changé, pour tenir compte des nombreuses "*middleboxes*" boguées,
— les numéros de séquence sont chiffrées (sans cela, corréler deux échanges utilisant des adresses IP différentes serait trivial). En fait, c'est plus compliqué que cela, il y a le numéro de séquence complet dans la partie chiffrée, et un numéro réduit à ses derniers bits en clair.

Si jamais vous vous lancez dans la programmation d'une bibliothèque DTLS, lisez l'annexe C, qui vous avertit sur quelques pièges typiques de DTLS.

En octobre 2021, il n'y avait pas encore de DTLS 1.3 dans GnuTLS (ticket en cours <https://gitlab.com/gnutls/gnutls/-/issues/1019>), BoringSSL ou dans OpenSSL (ticket en cours <https://github.com/openssl/openssl/issues/13900>).

Merci à Manuel Pégourié-Gonnard pour sa relecture attentive.