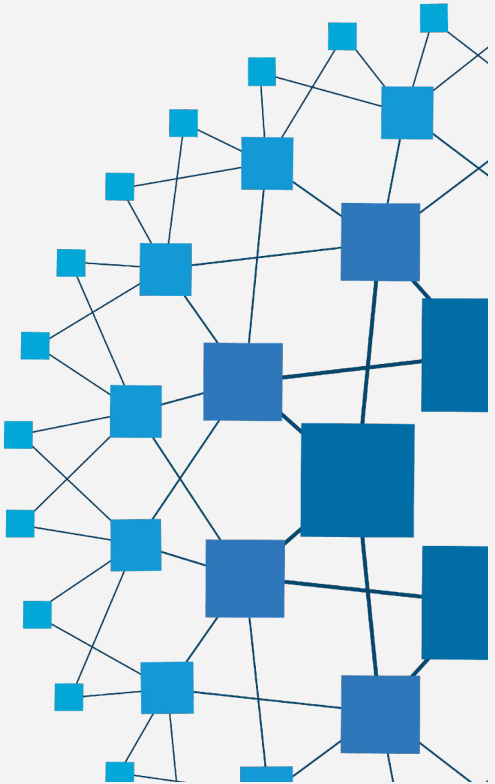
HTTP/3 - QUIC: impacts sur la sécurité

FRnOG 37 - 2023-04-14
Amaury Denoyelle - Willy Tarreau
HAPROXY

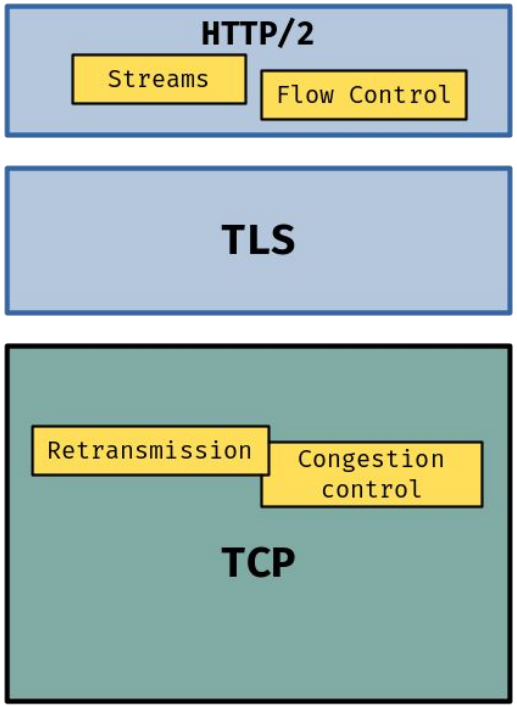


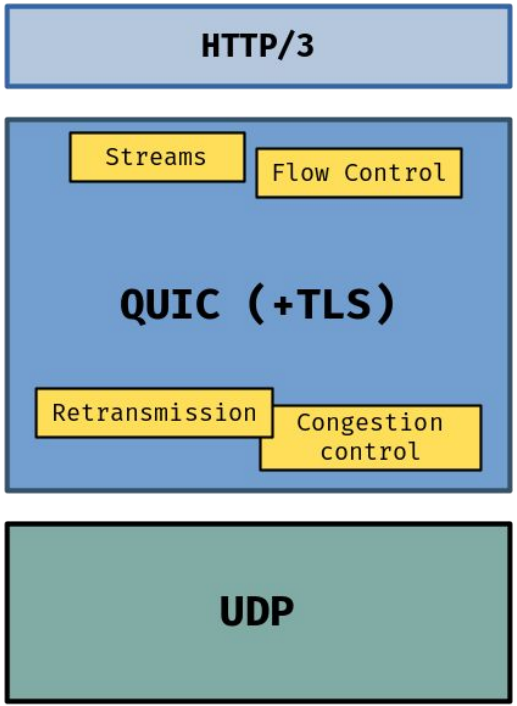
Petit rappel sur QUIC et HTTP/3

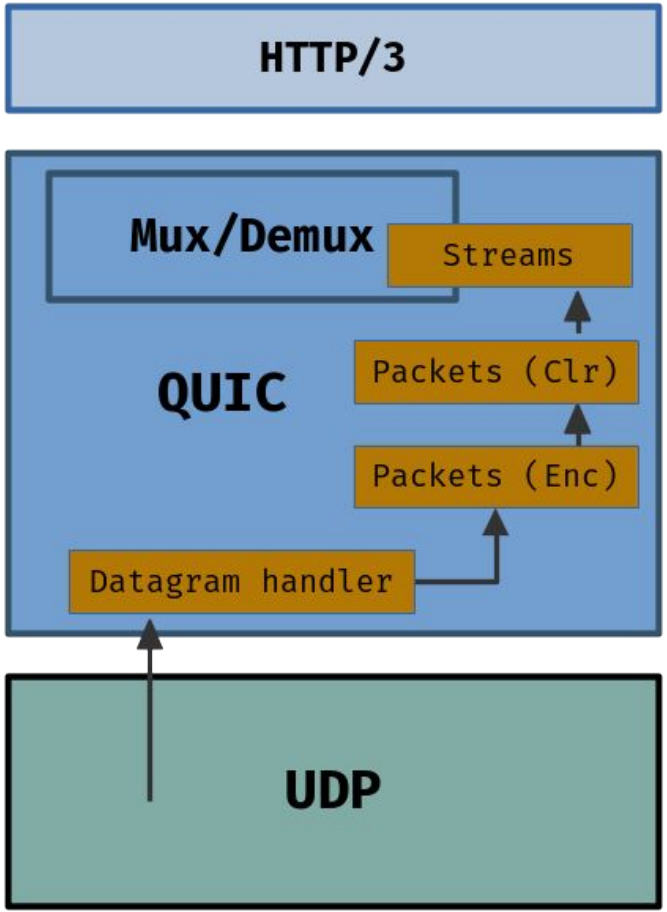
QUIC ? HTTP/3 ? h3 ?

- HTTP/3 (h3 pour les intimes) est la troisième version du protocole HTTP:
  - Binaire (comme HTTP/2)
  - Basé sur des trames (comme HTTP/2)
  - Repose sur des streams fournis par un protocole de transport
- QUIC est un moyen de transport fiable et sécurisé multiplexé:
  - Utilise des datagrammes **UDP**
  - Fiable (contrôle de congestion, ack, retransmits etc)
  - Sécurisé (TLS, entêtes masqués etc)
  - Fournit des streams indépendants (pratique pour HTTP/3)

QUIC ? HTTP/3 ? h3 ?






HTTP/3
Mux/Demux
Streams
QUIC
Packets (Clr)
Packets (Enc)
Datagram handler
UDP

Réticences parfois entendues

- UDP (spoofing, amplification, floods, fuites de données)
- Ports arbitraires (443 ou autres)
- Prise en compte par les outils de protection existants ?
- Logging sur les firewalls ?
- Conso CPU

Attaques de tiers

Amplification

- Scénario:

  Un attaquant se fait passer pour une IP victime et envoie du trafic à un serveur QUIC dans le but de *flooder* cette victime, comme on fait d'habitude avec DNS ou NTP

- Parade:
  - *Initial Packet* gros (1200 bytes) donc coûteux pour l'attaquant
  - Pas plus de 3x plus de data émises que ce qui est reçu
  - *Retry* stateless tout petit (cf SYN cookies)
  - Les paquets autres que *Initial* resteront sans réponse car sans session

Relayage de trafic

- Scénario:

  Un attaquant veut envoyer du trafic à une machine depuis une autre pour contourner des règles de filtrage par exemple, comme on fait en utilisant les services echo, rpcss etc.

- Parade:
  - La réponse à un *Initial Packet* commence par petit header avec 1 octet, version (4 oct), CID (20 oct max)
    => **25 octets non choisissables par l'attaquant**

Attaques sur l’infrastructure

Conso CPU par flooding de paquets

- Scénario:
  - Un attaquant cherche à saturer le CPU en stressant la stack qui est en userland, donc qui présente un coût de traversée élevé par paquet
- Parades:
  - les **petits paquets** sont soumis au **contrôle de CID** (très rapide); on absorbe sans difficulté 15 Mpps et 150 Gbps sur un 2x24 cores
  - Les paquets coûteux sont les *Initial Packet*, **assez gros** (~1.2 kB = 10kb) et **limitent** donc les traitements à seulement 1 Mpps @10Gbps en théorie
  - En pratique bien moins grâce au **retry stateless**
  - Filtrages stateful/stateless implémentables au plus bas niveau (XDP etc)

Conso RAM par flooding de paquets

- Scénario:
  - Attaques de réassemblage visant à mettre en attente plein de paquets disjoints sans envoyer ceux qui les débloquent
- Parades:
  - **Flow control** par stream (16kB stream sur HAProxy)
  - Limite sur le nombre de streams ouverts
  - Streams **indépendants** et traitement out-of-order => les streams **progressent** sans être gênés par des streams incomplets

Conso RAM par connexions massivement spoofées

- Scénario:
  - Attaques de saturation visant à établir plein de connexions spoofées
- Parades:
  - Retry sur connexion non confirmée
  - Retry token et serveur CID non prédictibles => nécessiterait d'être sur le chemin
  - Paquets **chiffrés + signés** dans les deux sens => un tiers ne peut insérer que des rejeux dans une connexion établie.

Conso RAM par connexions réelles

- Scénario:
  - Attaque de saturation visant à établir plein de connexions réelles depuis une machine sacrifiable et de les maintenir le plus longtemps possible
- Parade:
  - Idle timeout borné par le serveur
- Scénario:
  - Maintien de la connexion grâce à des trames de PING
- Parade:
  - Timeouts streams HAProxy libèrent la connexion
  - Limite sur le nombre de retransmissions

Conso RAM par streams concurrents

- Scénario:
  - Attaque de saturation visant à établir plein de streams parallèles et à les maintenir (requêtes incomplètes, ou *slow read* des réponses)
- Parade:
  - Limite sur le nombre de streams concurrents
  - Comme H1/H2, limites à choisir par l'implémentation
  - Pour *SlowRead*, pas de taille minimum de fenêtre de flow control
    => à traîter par l'implémentation (limite de buffers en Tx sur haproxy + timeout client)

Abus répétés / flood de requêtes

- Scénario
  - HAProxy sait se protéger contre des attaques en TCP (stick-tables, ...)
  - Un attaquant peut-il passer outre en utilisant QUIC ?
- Parade:
  - Dans HAProxy, QUIC est un protocole de transport comme TCP, donc on travaille avec des *“sockets* QUIC” et les logs, les règles `tcp-request connection` et les `track-sc` s’appliquent pareil qu’en TCP. On peut donc bloquer automatiquement les abus très tôt.

Flux non autorisés

- Problèmes classiques:
  - UDP cible fréquente d'attaques de type fuite de données ou reverse shells
  - Parfois l'UDP est restreint voire bloqué sur les routeurs périphériques (donc stateless)
  - QUIC fonctionne sur UDP et nécessite de rouvrir ces flux

  Solutions:

  - il peut être judicieux de n’autoriser UDP que **depuis/vers des plages d’IP précises pour un ou des ports précis et dédiés à QUIC**
  - Le **filtrage stateful** peut être pertinent pour éviter les **fuites de données**

Attaques sur les données

Sniffing, Hijacking, Tracking etc

- Scénario:
  - Tentative d'interception de données
  - Tentative d'insertion dans une connexion
  - Traque des utilisateurs
- Parades:
  - Chiffrement obligatoire des entêtes des paquets
  - Chiffrement obligatoire des données en TLS 1.3 (pas d'algo nul, pas de MITM, PFS).
  - Attaques "*rebinding*" etc: impossible de forcer une migration de connexion sans la confirmation du client

Attaques sur les applications

Attaques applicatives (plutôt HTTP/3 que QUIC)

- Scénario:
  - Plantages, contournements de filtrage, désynchronisation
- Parades:
  - Une seule request/response par stream (comme H2)
  - Content-length optionnel mais contrôle obligatoire si présent (comme H2)
  - headers binaires donc risque d'injection de caractères moisis (comme H2)

    => rien de nouveau par rapport à HTTP/2, privilégier la réutilisation de code déjà validé.
  - **Note1: vérifier que HTTP/0.9 n'est pas activé (via ALPN)!**
  - **Pour plus d'infos, RFC 9000 section 21. Security considerations**

That’s all folks!

(Y’a plus qu’à déployer)

Contacts:

- Amaury Denoyelle <adenoyelle@haproxy.com>
- Willy Tarreau <wtarreau@haproxy.com>