



Filière SR2I

Etude et analyse de QUIC et étude d'une extension pour les VPN

Auteurs :

M. Sohaïb OUZINEB
M. Harith PROIETTI
M. Marvin ASSOMANY

Professeur :
M. Ahmed SERHROUCHNI

Version du 13 mai 2020

Table des matières

1 Présentation et analyse du protocole QUIC	1
1.1 Motivations et fonctionnement général du protocole	1
1.1.1 Historique et idées générales	1
1.1.2 Fonctionnement détaillé	3
1.2 Analyse du header	7
1.2.1 Introduction	7
1.2.2 Long Header	8
1.2.3 Short Header	9
1.2.4 Connection ID (SCID, DCID)	10
1.2.5 Négociation de Versions	11
1.3 Présentation de différents scénarios - Analyse du trafic QUIC	12
1.3.1 Mise en place d'un client et d'un serveur QUIC en local avec ngtcp2 .	12
1.3.2 Mise en place de scénarios	13
1.4 Comparaison de performance et de qualité de service avec TLS et TCP .	25
1.4.1 Similitudes	25
1.4.2 Différences	26
1.5 Avantages et inconvénients du protocole QUIC	29
1.5.1 Avantages	29
1.5.2 Inconvénients	30
1.6 Etude des différentes vulnérabilités	30
1.6.1 Introduction	30
1.6.2 Attaque de Rejeu	30
1.6.3 Attaque par manipulation de paquet	31

2 Adaptation de QUIC aux VPNs	33
2.1 Généralités sur les VPNs	33
2.1.1 Principe Général	33
2.1.2 Types de VPN	34
2.1.3 VPN Site-to-Site	35
2.1.4 Protocoles de Tunneling	36
2.1.5 Mode de fonctionnement	37
2.1.6 Protocole IKE	41
2.2 Architecture des VPN SSL/TLS	44
2.2.1 Bref historique et intérêt	44
2.2.2 Fonctionnement SSL	45
2.2.3 Gestion de l'authentification	49
2.2.4 Vérification des certificats	50
2.2.5 Gestion de la confidentialité	50
2.2.6 Gestion de l'intégrité	50
2.2.7 Gestion du non-rejet	51
2.2.8 SSL appliqué aux VPNs : exemple d'OpenVPN	51
2.3 Implémentation du protocole QUIC VPN en python	52
2.3.1 Descriptif du protocole	52
2.3.2 Implémentation	53
2.3.3 Tests	56
2.4 Propositions d'extensions pour QUIC	64
2.4.1 Authentification par clé publique	64
2.4.2 Authentification avec STK	66
2.4.3 Authentification par x509	66

Présentation et analyse du protocole QUIC

1.1 Motivations et fonctionnement général du protocole

1.1.1 Historique et idées générales

QUIC (Quick UDP Internet Connections) est un protocole expérimental de connexion développé par Google en 2013. On estime que sur le moteur de recherche Chrome, plus de la moitié des connexions aux serveurs de Google utilisent le protocole QUIC. Tout l'intérêt de ce protocole est qu'il améliore de manière drastique les performances d'applications orientées connexion qui utilisent donc le protocole TCP au niveau transport du modèle OSI.

Présentons rapidement un bref historique de ce qui a donné naissance au protocole QUIC.

HTTP, le principal protocole de niveau applicatif, a vu le jour en 1991 sous le nom de protocole HTTP/0.9, qui est devenu HTTP/1.1 en 1999 après normalisation par l'IETF (Internet Engineering Task Force). Au bout d'un certain temps, HTTP/1.1 n'a plus été en mesure de combler l'évolution des besoins du Web et c'est ainsi que HTTP/2 a vu le jour en 2015.

Le protocole HTTP a ainsi continué d'être testé et amélioré sur Internet. Google en particulier avait entamé des essais avec un logiciel du nom de SPDY (prononcer « speedy »). Ce protocole était censé améliorer la navigation sur le Web, ce qui constitue la principale raison d'être du protocole HTTP. Fin 2009, la v1 de SPDY fut annoncée, et elle fut rapidement suivie par la v2 en 2010.

Globalement, SPDY a repris les principes de base du HTTP et modifié légèrement le format d'échange pour y apporter les améliorations nécessaires. Finalement, SPDY a été adopté en 2012 et est devenu HTTP/2.0 en 2015, après divers remaniements.

Pendant cette période, Google a développé en parallèle de SPDY un autre projet appelé gQUIC. La syntaxe binaire SPDY a été compilée en paquets gQUIC pouvant être envoyés sous forme de datagrammes UDP. Cela constituait une rupture historique par rapport au

transport TCP sur lequel HTTP s'appuyait traditionnellement.

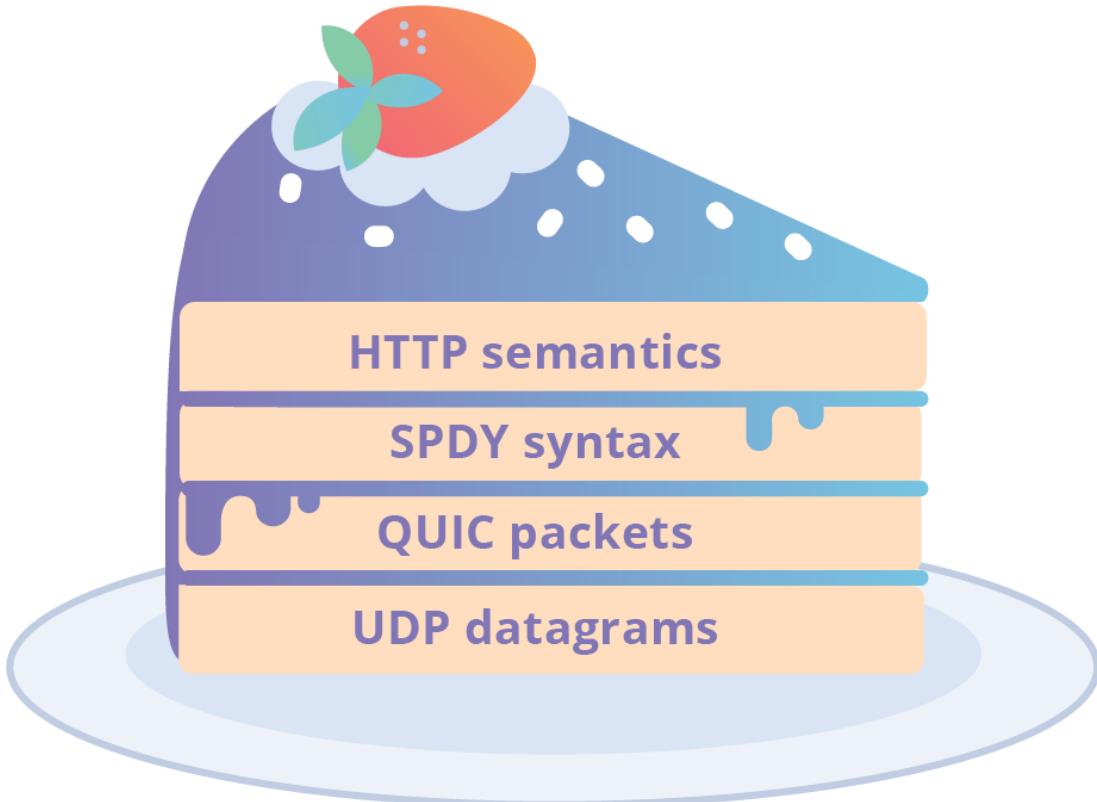


FIGURE 1.1 – Architecture gQUIC

Google avait élaboré une nouvelle méthode appelée QUIC Crypto, permettant d'accélérer les handshakes de sécurité au niveau session. (Il a ensuite été remplacé par TLS 1.3, qui n'existe pas encore initialement). Avec QUIC, la connexion et le chiffrement sont en effet réalisés en un seul handshake, ce qui permet un gain de temps considérable.

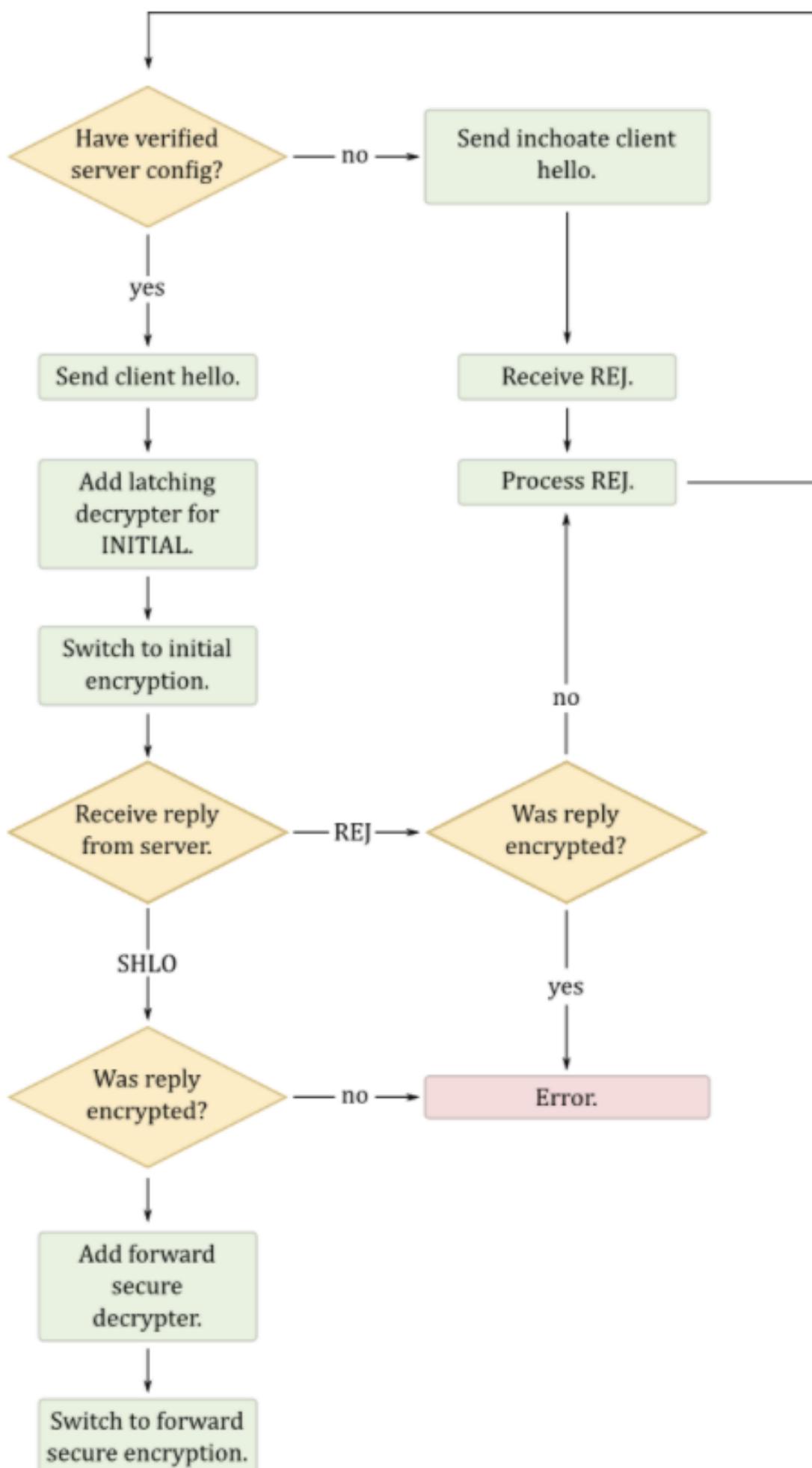
QUIC a notamment pu résoudre le problème du “HoL” : le “Head of line blocking”. Comme HTTP/2 reposait sur le protocole TCP, qui doit garantir le bon ordre d'arrivée des paquets à leur destinataire, si un paquet était perdu, toute la connexion TCP devait être stoppée en attendant qu'il soit retransmis, et cela bloquait ainsi les différents flux de données sur cette connexion. QUIC avait l'avantage de ne pas utiliser TCP et pouvait effectuer des décisions intelligentes sur les paquets à arrêter et ceux à laisser passer dans un tel cas de figure, ce qui a permis de pallier fortement le problème de saturation du trafic.

Une autre idée majeure permise par QUIC est celle du “ 0-RTT”. L’idée derrière est que lorsqu’un client reprend une connexion QUIC avec un serveur avec lequel il a déjà établi une connexion TLS auparavant, il n’a pas besoin d’établir à nouveau un 3-way handshake : des informations sur la session précédente sont gardées en mémoire, notamment la clé de chiffrement utilisée lors de la communication. De fait, le message du client peut directement être chiffré avec sa clé privée tout en étant directement compris par le serveur.

En novembre 2018, l’IETF s’est réunie à Bangkok et un nouveau projet Internet a été adopté. Le protocole de transport QUIC, le successeur de HTTP/2, a été renommé en HTTP/3.

1.1.2 Fonctionnement détaillé

Voici un diagramme retracant la situation lorsqu’un client initie une connexion handshake.



Dans un premier temps, le client ne connaît rien du serveur avec lequel il va communiquer. Avant d'initier le “handshake”, il envoie une requête “client hello” incomplète, qui est en fait une requête “client hello” au payload vide mais contenant notamment l’adresse DNS du serveur, une demande de preuve de certificat, et la version du protocole QUIC supportée par le client.

A la suite de cette requête, le client reçoit un message REJ (Reject) de la part du serveur, dans lequel sont notamment contenues une preuve d’authenticité ainsi que sa dernière configuration en date. Cette configuration contient notamment la clé publique de chiffrement du serveur ainsi que les algorithmes de chiffrement qu’il supporte. La preuve d’authentification du serveur, quant à elle (dans le cadre de la norme X.509), est constituée de la signature de la configuration du serveur à l’aide de la clé publique du client, qui pourra la décoder avec sa clé privée.

A ce stade, le client dispose donc de la connaissance de la configuration du serveur, et peut donc lui envoyer une requête “client hello” complète, avec un vrai payload. A noter que cet échange de “client hello inchoate” et de messages REJ peut durer un certain temps, car le serveur ne souhaite pas forcément dévoiler toutes ses informations à un client inconnu.

Une fois que le client envoie de vraies requêtes “client hello”, le serveur peut soit envoyer une message REJ ou bien accepter la requête et renvoyer un “serveur hello” crypté (SHLO sur le diagramme) contenant des clés à usage unique, dites “éphémères”. Dans ce dernier cas, cela signifie que le “handshake” s’est déroulé correctement. Autrement, en cas de réponse REJ, le serveur donne au client des informations lui permettant de réaliser un meilleur handshake. Par exemple, si le client ne s’est jamais connecté au serveur, ce dernier doit lui donner un “token d’adresse source” (“source-address token”), qui est une sorte de passe droit à usage multiple contenant l’adresse IP du client et un horodatage de la part du serveur. Ce token garantit à la fois l’intégrité et la confidentialité des données. Le client est alors en possession de clés de chiffrement dites “clés initiales”, et ce sont avec celles-ci qu’il devra chiffrer les futurs paquets.

Ci-dessous, quelques-uns des principaux tags utilisés par QUIC, d’après la documentation officielle :

Lors du “client hello inchoate” :

SNI

Server Name Indication (optional) : the fully qualified DNS name of the server, canonicalised to lowercase with no trailing period. Internationalized domain names need to be encoded as A-labels defined in RFC 5890. The value of the SNI tag must not be an IP address literal.

STK

Source-address token (optional) : the source-address token that the server has previously provided, if any.

PDMD

Proof demand : a list of tags describing the types of proof acceptable to the client, in preference order. Currently only X509 is defined.

Lors de la réception du message Rejection (tag REJ) :

SCFG

Server config (optional) : a message containing the server's serialised config. (Described below.)

STK

Source-address token (optional) : an opaque byte string that the client should echo in future client hello messages.

STTL

The duration, in seconds, that the server config is valid for.

PROF

Proof of authenticity (optional) : in the case of X.509, a signature of the server config by the public key in the leaf certificate. The format of the signature is currently fixed by the type of public key : RSA, RSA-PSS-SHA256, ECDSA, ECDSA-SHA256

Dans la configuration du serveur :

KEXS

Key exchange algorithms : a list of tags, in preference order, specifying the key exchange algorithms that the server supports. The following tags are defined : C255, Curve25519, P256, P-256

AEAD

Authenticated encryption algorithms : a list of tags, in preference order, specifying the AEAD primitives supported by the server. The following tags are defined :

AESG

AES-GCM with a 12-byte tag and IV. The first four bytes of the IV are taken from the key derivation and the last eight are the packet sequence number.

S20P

Salsa20 with Poly1305. (Provisional and not yet implemented.)

Enfin, les tags contenus dans le “full client hello” du client :

SCID

Server config ID : the ID of the server config that the client is using.

AEAD

Authenticated encryption : the tag of the AEAD algorithm to be used.

KEXS

Key exchange : the tag of the key exchange algorithm to be used.

NONC

Client nonce : 32 bytes consisting of 4 bytes of timestamp (big-endian, UNIX epoch seconds), 8 bytes of server orbit and 20 bytes of random data.

SNO

Server nonce (optional) : an echoed server nonce, if the server has provided one.

PUBS

Public value : the client's public value for the given key exchange algorithm.

CETV

Client encrypted tag-values (optional) : a serialised message, encrypted with the AEAD algorithm specified in this client hello and with keys derived in a manner specified in the CETV section. This message will contain further, encrypted tag-value pairs that specify client certificates, Channel IDs etc.

1.2 Analyse du header

1.2.1 Introduction

Un paquet QUIC est le contenu des datagrammes échangés par entre 2 agents, appelés “QUIC Endpoints”.

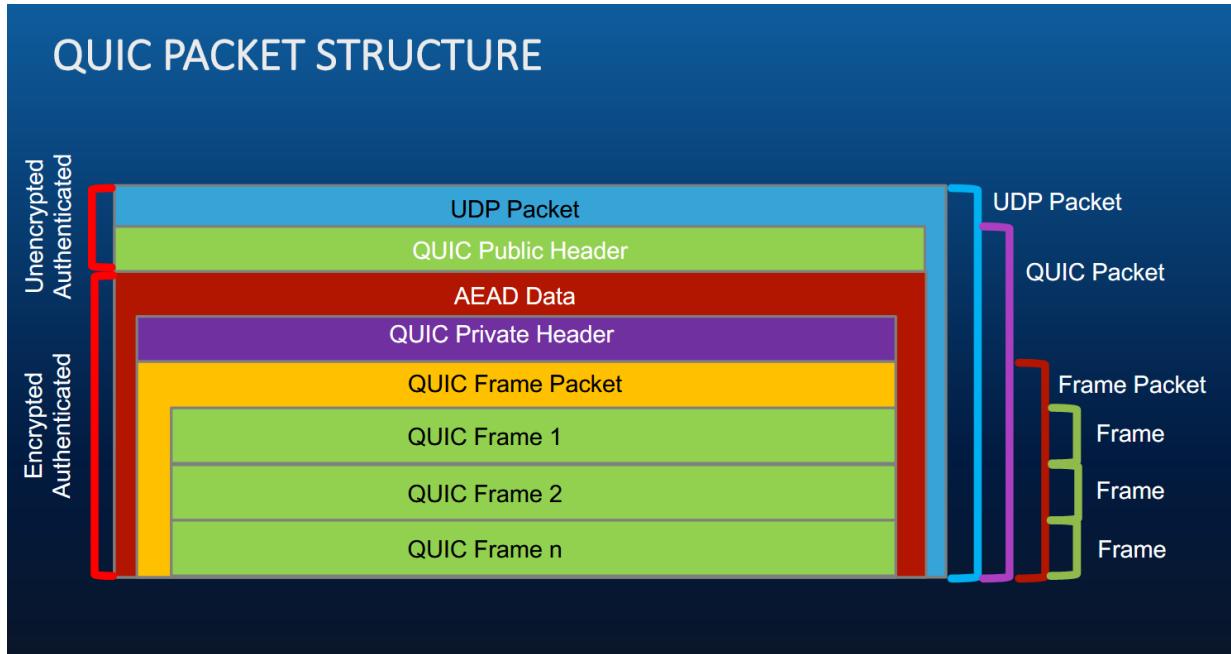


FIGURE 1.3 – Structure d'un paquet QUIC

Le protocole QUIC définit deux types de Headers : long et court. La distinction entre ces deux types de headers se retrouve d'une part dans la différence de taille de header, mais également le bit le plus significatif qui diffère selon ces types (0 : short header, 1 : long header). Les données et longueurs des paquets QUIC dépendent de la version de QUIC utilisée.

1.2.2 Long Header

Voici la représentation d'un “Long Header” QUIC :

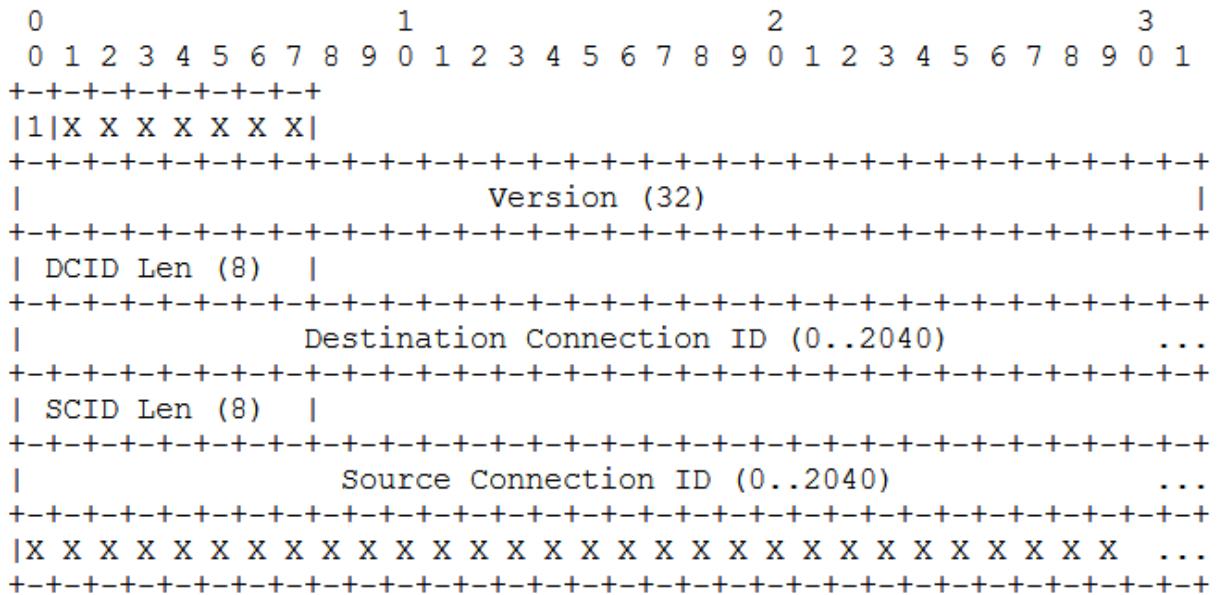


FIGURE 1.4 – Datagramme Long Header

Comme l'indique cette figure, un paquet QUIC avec un “long Header” aura le bit le plus significatif du premier octet à 1. Ce header se compose de Le champ “Version”, indiquant la version de QUIC, sur 4 octets ; Le champ DCID Length, indiquant sur 1 octet la longueur en octets du champ DCID ; Le champ DCID (Destination Connection ID), dont la longueur varie entre 0 et 255 octets; le champ SCID Length ,indiquant sur 1 octet la longueur du champ SCID ; Le champ SCID (Source Connection ID), dont la longueur également varie entre 0 et 255 octets. Lors du “handshake”, le “long header” est utilisé afin d'établir des connection ID dans chaque direction.Chaque endpoint utilise le champ SCID afin de spécifier le DCID utilisé dans le paquet qui lui est adressé. A la réception d'un paquet, chaque endpoint établit le DCID afn qu'il corresponde au SCID recu.

1.2.3 Short Header

Voici la représentation d'un “Short Header” :

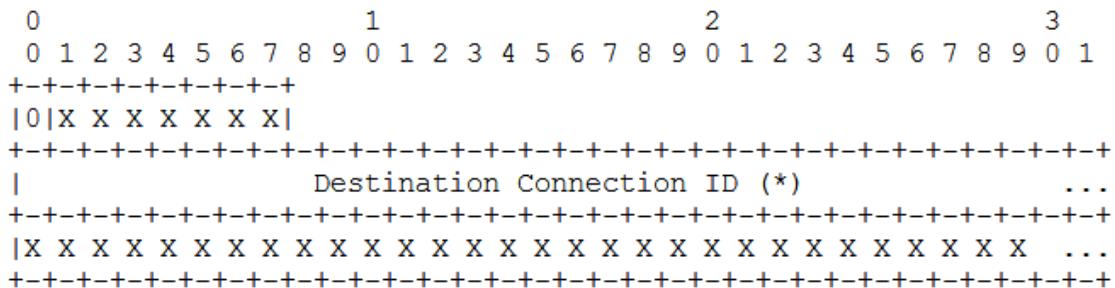


FIGURE 1.5 – Datagramme Short Header

Comme l'indique cette figure, un “short header” se reconnaît avec le bit plus significatif du premier octet à 0. Ce header est semblable au long header, mais plus court et simplifié, dans lequel seul un champ DCID (Destination Connection ID) est présent. Ce type de header n'inclut donc pas les autres champs existant dans un “long header” (Version ,DCID Length, SCID Length,SCID).

1.2.4 Connection ID (SCID, DCID)

Chaque connexion entre 2 endpoint possède un ensemble d'identifiants de connexion, appelés "Connexion ID", chacun pouvant identifier une connection. Celles ci sont indépendamment sélectionnées par les endpoints, qui doivent sélectionner celle que la paire doit utiliser.

Tel qu’indiqué ci-dessus, ce champ est de longueur variable, la fonction primaire de ce champ est de s’assurer que les changements d’adressage au niveau des protocoles de couches inférieur (UDP, IP,...) ne renvoient pas les paquets QUIC vers la mauvaise destination. Un “Connection ID“ est utilisé par les points finaux (source et destinataire du paquet), ainsi que les intermédiaires s’assurant du bon cheminement du paquet. Cet ID est également conçu afin de déterminer à quelle connexion QUIC le paquet devrait être redirigée. Un ‘connection ID’ ne doit pas contenir d’informations qui peut être utilisée par un observateur externe afin de les corréler avec d’autres ‘Connection ID’. Un “longueur Header” contient un “champ SCID (Source Connection ID)” ainsi qu’un ‘champ DCID (Destination Connection ID”)’. Ces champs sont utilisés afin de mettre en place un Connection ID pour de nouvelles connexions. Le DCID est choisi par le récepteur du paquet et permet ainsi de fournir un routage consistant, tandis que le SCID est utilisé afin de mettre en place le DCID utilisé par la paire.

1.2.5 Négociation de Versions

Les versions de QUIC sont identifiées avec un ID de 32 bits. La version 0 est réservé pour la négociation de version. Un endpoint recevant un paquet avec un long header muni d'une version non compris ou non supporté pourrait envoyer un paquet de négociation de version en réponse, ce qui ne sera pas le cas pour des paquets avec un short header.

Voici un le datagramme d'un paquet de négociation de version :

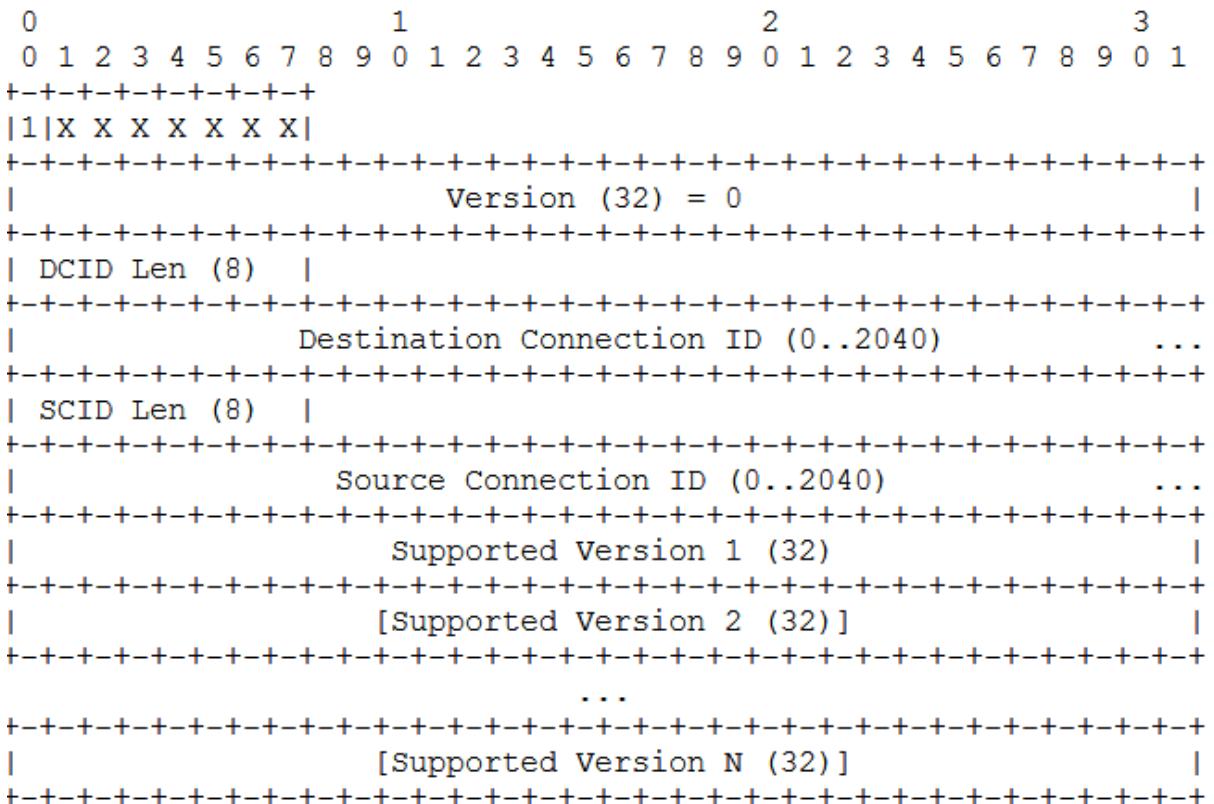


FIGURE 1.6 – Datagramme de négociation de version

Un paquet de négociation de version contient une liste de version supportée, chacune identifiant une version qu'un émetteur supporte. Cette liste de version supportées suit le champs “Version”. Un paquet de négociation de version de contiendra pas d'autre champs. Si un paquet ne contient pas de champs de version supportée, ou tronquée, un endpoint devra l'ignorer. Aucune protection d'intégrité ou de confidentialité n'est assurée sur les paquets de négociation de version. Une version spécifique de QUIC pourrait authentifier le paquet comme une partie de son procédé d'établissement de connexion. Un endpoint

doit inclure la valeur venant du champs SCID du paquet reçu dans le champ DCID. La valeur du SCID doit être copié du DCID du paquet reçu sélectionné initialement de façon aléatoire. Ecouter les 2 Connections donnent l'assurance que le serveur a reçu le paquet d'une part, et que la version n'a pas été générée par un attaquant. Un endpoint recevant un paquet de négociation de version pourrait changer de version pour les sous-paquets. Les conditions pour lesquelles un endpoint va changer de version QUIC dépendra de la version choisie.

1.3 Présentation de différents scénarios - Analyse du trafic QUIC

1.3.1 Mise en place d'un client et d'un serveur QUIC en local avec ngtcp2

1.3.1.1 Mise en place du serveur QUIC en local

ngtcp2 est un draft d'implémentation du protocole QUIC implémenté en C. Cette implémentation contient un programme serveur dont la syntaxe est la suivante :

```
examples/server [OPTIONS] <ADDR> <PORT> <PRIVATE_KEY_FILE> <CERTIFICATE_FILE>
```

ADDR contient l'adresse du serveur. Dans ce cas, ce sera localhost. PORT est le port sur lequel le serveur sera à l'écoute. Mettons 4433 pour éviter une éventuelle redondance avec le port 443 qui est déjà pris par Google. PRIVATE_KEY_FILE est la clé privée du serveur au format pem, par exemple une clé rsa. CERTIFICATE_FILE est le certificat du serveur au format x509. Attention cependant, le commonName du certificat doit être ‘localhost’, sinon le certificat ne correspond pas à l'adresse du serveur et sera rejeté.

Une multitude d'options pour le serveur est disponible, notamment :

`-ciphers=<CIPHERS>` : Pour préciser la liste des algorithmes de chiffrements que l'on souhaite utiliser

`-groups=<GROUPS>` : Pour préciser les différents groupes supportés (groupes de Diffie-Hellman)

`-verify-client` : Pour exiger un certificat client

Nous avons ensuite différents paramètres utiles pour la gestion de flux, notamment :

`-max-data=<SIZE>` : La taille initiale de la fenêtre de contrôle de flux

Nous pouvons maintenant lancer le serveur QUIC en local avec la commande : examples/server localhost 4433 ssl_key.pem ssl_cert.pem

SSL_CERT étant un certificat feuille ayant pour identité ‘localhost’ délivré par l’autorité racine ‘our-ca-server’.

1.3.1.2 Mise en place du client QUIC en local

Toujours avec ngtcp2, on utilise le programme client ayant pour syntaxe :

examples/client [OPTIONS] <ADDR> <PORT> <URI>

ADDR étant l’adresse de destination, localhost dans notre cas. PORT le numéro de port du serveur, 4433 dans notre cas URI la ressource que l’on souhaite récupérer auprès du serveur.

Parmis les options on compte :

-d, -data=<PATH> : lire les données dans PATH et les envoyer au serveur en tant que STREAM

-v, -version=<HEX> : la version du protocole QUIC à utiliser au format hexadécimal.

-ciphers=<CIPHERS>, -groups=<GROUPS> : les algorithmes de chiffrements et les groupes supportés par le client.

-session-file=<PATH> : Pour lire un ticket de session TLS et reprendre la session.

-key-update=<DURATION> : Pour mettre à jour les clés au bout d’un temps défini ici

-key=<PATH>, -cert=<PATH> : La clé privée et le certificat du client au format PEM.

-qlog-file=<PATH> : Pour récupérer les logs de connexions au format QLOG qui pourront par la suite être visualisés en utilisant QVIZ.

Enfin, on a des paramètres de transport tels que la taille initiale de la fenêtre de la gestion de flux, comme avec le serveur.

On peut alors lancer le client avec la commande :

examples/client -qlog-file logs.qlog localhost 4433

1.3.2 Mise en place de scénarii

1.3.2.1 Connexion à un serveur inconnu, étude du traffic QUIC avec l’outil chrome ://net-export/

Pour analyser une connexion à un serveur inconnu, nous pouvons simplement utiliser l’outils de capture de logs intégré au navigateur google-chrome. Pour cela, on se dirige à la page chrome ://net-export/. On choisit de retirer les informations personnelles des

captures de log, notamment les cookies, les mots de passe, etc... Après avoir lancé la capture, on se dirige vers youtube.com. (Sur la machine virtuelle kali employée, c'était la première connexion au serveur youtube). Une fois la page chargée, on peut la fermer et arrêter la capture. Une fois la capture enregistrée au format JSON, nous désirons l'analyser pour observer le traffic QUIC. Pour cela, nous utilisons l'extension chrome : Chromium NetLog dump viewer. Nous pouvons upload les logs et cliquer sur l'onglet QUIC pour voir apparaître un menu indiquant les propriétés sur les connexions QUIC qui ont été ouvertes. On peut alors cliquer sur "view all QUIC sessions", puis choisir le nom de domaine youtube.com pour afficher le traffic QUIC à youtube.

On a tout d'abord un CHLO :

```
309: QUIC_SESSION
www.youtube.com
Start Time: 2020-04-17 06:59:13.925

t= 4364 [st= 0] +QUIC_SESSION [dt=29869+]
    --> cert_verify_flags = 0
    --> host = "www.youtube.com"
    --> port = 443
    --> privacy_mode = false
    --> require_confirmation = false
t= 4364 [st= 0] QUIC_SESSION_CRYPTO_HANDSHAKE_MESSAGE_SENT
    --> CHLO<
        SNI : "www.youtube.com"
        VER : 'Q046'
        CCS : 0x01e8816092921ae87eed8086a2158291
        UAID: "Chrome/81.0.4044.113 Linux x86_64"
        TCID: 0
        PDMD: 'X509'
        SMHL: 0x01000000
        ICNL: 30
        NONP: 0x23e657e33fe585a3d10f082484538f1b229188781c586031ea02e45c56f97d1f
        MIDS: 100
        SCLS: 1
        CSCT: 0x
        COPT: '5RTO', 'ACKD'
        IRTT: 147503
        CFCW: 15728640
        SFCW: 6291456
    >
t= 4364 [st= 0] QUIC_SESSION_STREAM_FRAME_SENT
    --> fin = false
    --> length = 1024
    --> offset = 0
    --> stream_id = 1
t= 4364 [st= 0] QUIC_SESSION_PADDING_FRAME_SENT
    --> num_padding_bytes = -1
t= 4364 [st= 0] QUIC_SESSION_PACKET_SENT
    --> encryption_level = "ENCRYPTION_INITIAL"
    --> packet_number = 1
    --> sent_time_us = 110929063751
    --> size = 1350
    --> transmission_type = "NOT RETRANSMISSION"
t= 4529 [st= 165] QUIC_SESSION_PACKET RECEIVED
    --> peer_address = "172.217.22.142:443"
    --> self_address = "192.168.0.24:48457"
    --> size = 1350
t= 4529 [st= 165] QUIC_SESSION_UNAUTHENTICATED_PACKET_HEADER_RECEIVED
    --> connection_id = "0"
```

FIGURE 1.7 – CHLO

Penchons-nous sur le CHLO :

- SNI : le nom DNS du serveur 'www.youtube.com'
- STK : (Source-address token) absent, car serveur inconnu pour le client

- VER : la version du protocole QUIC utilisée ‘Q046’
 - CCS : Common certificate sets, une série de 64 bits contenant les hachés au format FNV-1a de sets de certificats que le client possède
 - Le flag CCRT (Cached certificates) est absent car le client ne possède pas de certificats liés à ce nouveau serveur.
 - PDMD : Proof Demand, types de preuves acceptables pour le client, ici ‘x509’
- A cela on rajoute du padding. On reçoit dans un premier temps un QUIC_SESSION_PACKET indiquant que la version a bien été négociée à ‘Q046’.

```

ICSL: 50
NONP: 0x23e657e33fe585a3d10f082484538f1b229188781c586031ea02e45c56f97d1f
MIDS: 100
SCLS: 1
CSCL: 0x
COPT: 'SRTO', 'ACKD'
IRTT: 147503
CFCW: 15728640
SFCW: 6291456
>
t= 4364 [st= 0] QUIC_SESSION_STREAM_FRAME_SENT
--> fin = false
--> length = 1024
--> offset = 0
--> stream_id = 1
t= 4364 [st= 0] QUIC_SESSION_PADDING_FRAME_SENT
--> num_padding_bytes = -1
t= 4364 [st= 0] QUIC_SESSION_PACKET_SENT
--> encryption_level = "ENCRYPTION_INITIAL"
--> packet_number = 1
--> sent_time_us = 110929063751
--> size = 1350
--> transmission_type = "NOT RETRANSMISSION"
t= 4529 [st= 165] QUIC_SESSION_PACKET_RECEIVED
--> peer_address = "172.217.22.142:443"
--> self_address = "192.168.0.24:48457"
--> size = 1350
t= 4529 [st= 165] QUIC_SESSION_UNAUTHENTICATED_PACKET_HEADER_RECEIVED
--> connection_id = "0"
--> header_format = "IETF_QUIC_LONG_HEADER_PACKET"
--> long_header_type = "INITIAL"
--> packet_number = 1
--> reset_flag = 0
--> version_flag = 1
t= 4529 [st= 165] QUIC_SESSION_PACKET_AUTHENTICATED
t= 4529 [st= 165] QUIC_SESSION_VERSION_NEGOTIATED
--> version = "Q046"
t= 4529 [st= 165] QUIC_SESSION_VERSION_NEGOTIATED
--> version = "0046"
t= 4529 [st= 165] QUIC_SESSION_ACK_FRAME RECEIVED
--> delta_time_largest_observed_us = 4910
--> largest_observed = 1
--> missing_packets = []
--> received_packet_times = []
t= 4529 [st= 165] QUIC_SESSION_STREAM_FRAME RECEIVED
--> fin = false
--> length = 1312
--> offset = 0
--> stream_id = 1
QUIC_SESSION_PACKET_RESETTING

```

FIGURE 1.8 – QUIC SESSION PACKET

Nous recevons aussi le REJ du serveur :

```

t= 4544 [st= 180] QUIC_SESSION_PACKET RECEIVED
--> peer_address = "172.217.22.142:443"
--> self_address = "192.168.0.24:48457"
--> size = 1350
t= 4544 [st= 180] QUIC SESSION_UNAUTHENTICATED_PACKET_HEADER RECEIVED
--> connection_id = "0"
--> header_format = "IETF_QUIC_LONG_HEADER_PACKET"
--> long_header_type = "INITIAL"
--> packet_number = 2
--> reset_flag = 0
--> version_flag = 1
t= 4544 [st= 180] QUIC_SESSION_PACKET_AUTHENTICATED
t= 4544 [st= 180] QUIC_SESSION_STREAM_FRAME RECEIVED
--> fin = false
--> length = 1283
--> offset = 1312
--> stream_id = 1
t= 4544 [st= 180] QUIC_SESSION_CRYPTO_HANDSHAKE_MESSAGE RECEIVED
--> REJ <
    STK : 0x2ce962942ccc43f88d55b44ea58b11cea482b963d1cd7a602dd3e25ad6d9a07ea0697a7c91a5689664ef1e6a69264
    SNO : 0x2e696289aba98cd80ed6ab0032cc5d8bc8212ecb70589ea1f5b1f3cdf2804faa602560b6e5cc23726559c
    PROF: 0x304402202109c4138f253f4dbe92e20bdbaffa9bcaedc638e60364f21b6ac00b36189cf3002204711b9730429178e3
    SCFG:
        SCFG<
            AEAD: 'AESG', 'CC20'
            SCID: 0x303a4afa251506c0d955629ff48c122a
            PDMD: 'CHID'
            PUBS: 0x200000e7973dcdd9573cc92228032eefeb494b6538649e4d794532b3399f1fa0c229550
            KEXS: 'C255'
            OBIT: 0x3030303030303030
            EXPY: 0xe0e89b5e00000000
        >
    RREJ: SERVER_CONFIG_INCHOATE_HELLO_FAILURE
    STTL: 0xd5c020000000000
    CRT: 0x0101009c0d000078bb22517eb1ad577b3c94db1a1e63cc8ccb1c935b6e5b63bb15dbf86672db1fad9d216292a719
>
t= 4547 [st= 183] SIGNED_CERTIFICATE_TIMESTAMPS RECEIVED
--> embedded_scts = "A08Ad0CyhgXMi6LNiiB0h2b5K7mKJSBna9r6c0eySVMt74uQXgAAAXE2CUKAAAEBGMEQCIG9T1MQBf7tM4
--> scts_from_ocsp_response = ""
--> scts_from_tls_extension = ""
t= 4547 [st= 183] SIGNED_CERTIFICATE_TIMESTAMPS CHECKED
--> scts = [{"extensions": "", "hash_algorithm": "SHA-256", "log_id": "sh4FzIuizYogTodm+Su5iiUgZ2va+nDnsklTLe+Lk
t= 4550 [st= 186] +CERT_VERIFIER REQUEST [dt=13]
t= 4550 [st= 186] CERT_VERIFIER_REQUEST_BOUND_TO_JOB
--> source_dependency = 313 (CERT_VERIFIER_JOB)
t= 4550 [st= 186] QUIC_SESSION_PADDING_FRAME RECEIVED
--> num_padding_bytes = 31
t= 4550 [st= 186] QUIC_SESSION_ACK_FRAME SENT
--> delta_time_largest_observed_us = 6181

```

FIGURE 1.9 – QUIC SESSION PACKET

Analysons quelques flags du REJ :

- STK : Source address token, une chaîne de bytes que le client devra afficher lors de futures messages client hello.
- SNO : Server nonce, que le client devra montrer dans chaque client hello complet. Ceci pour prévenir les attaques type replay
- PROF : Proof of authenticity, contient la signature de la configuration du serveur par la clé publique du certificat feuille. Le format de la signature dépendant de celui de la clé publique.
- SCFG : Server config : configuration du serveur, contenant :
 - AEAD : Authentication encryption algorithms, avec ici AESG en préférence
 - SCID : Server config ID
 - KEXS : algorithme d'échange de clé, ici C255 (Curve 25519)
 - PUBS : liste de valeurs publiques du server
 - EXPY : date d'expiration de la configuration du serveur

— STTL : Durée de validité de la configuration du serveur en secondes

On peut voir que le serveur envoie aussi son certificat x509, que le client doit vérifier par la suite.

```
t= 4550 [st= 186] +CERT VERIFIER REQUEST [dt=13]
t= 4550 [st= 186]   CERT_VERIFIER_REQUEST_BOUND_TO_JOB
    --> source_dependency = 313 (CERT_VERIFIER_JOB)
t= 4550 [st= 186]   QUIC_SESSION_PADDING_FRAME_RECEIVED
    --> num_padding_bytes = 31
t= 4550 [st= 186]   QUIC_SESSION_ACK_FRAME_SENT
    --> delta_time_largest_observed_us = 6181
    --> largest_observed = 2
    --> missing_packets = []
    --> received_packet_times = []
t= 4551 [st= 187]   QUIC_SESSION_PACKET_SENT
    --> encryption_level = "ENCRYPTION_INITIAL"
    --> packet_number = 2
    --> sent_time_us = 110929249565
    --> size = 36
    --> transmission_type = "NOT_RETRANSMISSION"
t= 4563 [st= 199] -CERT VERIFIER REQUEST
t= 4563 [st= 199] CERT_CT_COMPLIANCE_CHECKED
    --> certificate =
        -----BEGIN CERTIFICATE-----
        MIIJQjCCQCgAwIBAgIQV/qkUsd4kosIAAAAdeRUTANBgkqhkiG9w0BAoSFADBC
        M0swCQYDVQQGEwJVVzEaMBwGA1UECHMVR29vZ2xLIFRydxKN0IFN1cnZpY2VzMRMw
        E0YDVQ0DEwpHVFmG00EgMu8xMB4XD1wMDQwMTExNTgyN1oXDTIwMDYyNDEyNTgy
        N1owZjELMAKGA1UEBhMCVVMxKzARBgNVBAgTCkNhG1mb3JuawExFJAUBgNVBACt
        DU1vdW50YWhlIFZpZXcxEzARBgNVBAoTCkdvb2dsZSBMTEmxFTATBqNVBAMDCou
        Z29vZ2xLlmNvbTzBzMBMGByGSM49aGEGCgGSM49aWeH0IAIBISjJ/fj26gWhrU4
        dGQgmM7D3E4XURj1Vb/J1UYQ0vx+a+vYE1opnAlS1C1GAL49SJft0xWfLdLxcvzn
        028YPeujggbZMIGITA0BgNVH08BaF8EBAMCB4AwEwYDVR01BAwvCgYIKwYBBQUH
        AwEwDAYDVR0TAQH/BAIwADAdBgNVHQ4EFgQU5+k1j4bNEgkqc8wNu01SnsExbQyW
        HwYDVR0jBBgwFoAUmNH4bhDrz5vsYJBYkBug630j/SswZAYIKwYBBQUHAQEEWDBW
        McCGCCsGAQUFBzABhhhtdHRwO18vb2Nzc5wa2kuZ29vZy9ndHMxbzEwKwYIKwYB
        BOUHMAKGH20dHA6Ly9wa2kuZ29vZy9nc31yL0dUJzFPMS5jcn0wgg5dBgNVHREE
        ggSUMIIEKIMK15nb29nbGUuY29tLmF1gb8gLmdvb2dsZ55jb20uNKCdryuZ29vZ2xL
        Z55nb29nbGUuY29tghIdLnmsb2vLmdvb2dsZ55jb22CGcou3Jvd2Rzb3Vyy2Uu
        Z29vZ2xLlmNvbYIGKi5nLmNvgg4qlmdujcc5ndn0yLmNvbYIRKi5nY3BjZG4uZ3Z0
        MS5jb22CCiouZ2dwaHQuY26CDiouZ2t1Y25hcHBzLmNughYqlmdvb2dsZ51hbmf5
        eXRpY3MuY29tggsgLmdvb2dsZ55jYYIK15nb29nbGUuY2yCDiouZ29vZ2xLmNv
        Lmlugg4gLmdvb2dsZ55jb20y5qc1I0K15nb29nbGUuY28udWuCdrouZ29vZ2xLmNv
        b55hcoIPK15nb29nbGUuY29tLmF1gb8gLmdvb2dsZ55jb20uNKCdryuZ29vZ2xL
        LmNvb55jb41PKi5nb29nbGUuY29tLm14gg8qlmdvb2dsZ55jb20udHKCDrouZ29v
        Z2xLlmNvb552boILK15nb29nbGUuZGWCcYouZ29vZ2xLm10ggsqLmdvb2dsZ55m
        coILK15nb29nbGUuaHWCcYouZ29vZ2xLm10ggsqLmdvb2dsZ55ub1ILK15nb29n
        bgGlucGycCyouZ29vZ2xLmB0ghIqlmdvb2dsZWFKYXBpcy5jb22CDrouZ29vZ2xL
        YXBpcy5jboIRKi5nb29nbGUjbmFwcHMuY26CFcouZ29vZ2xLmVyy2bWyyY2UuY29t
        ghEqLmdvb2dsZKZpZGVvLmNbVbYIMK15nC3RhG1jLmNugg0qlmdzdGF0aMuY29t
        ghIqlmdzGF0aWNjbmFwcHMuY26CCiouZ3Z0MS5jb22CCiouZ3Z0M15jb22CFcou
        bwV0cm1jLmdzdGF0aWMyY29tggwqlnvYy2hpbi5jb22CEoudXjsLmdvb2dsZ55j
        b22CEyoud2Vhci5na2VjbmFwcHMuY26CFioueW91dHVizS1ub2Nvb2tpZ55jb22C
        DSoueW91dHVizS5jb22CFioueW91dHVizWVkdWnhdG1vb15jb22CESoueW91dHV
```

FIGURE 1.10 – certificat x509

Une fois ceci fait, le client peut alors envoyer un client hello complet, à la différence du client hello initial qui était incomplet car le serveur était inconnu. Maintenant, le client hello envoyé est de la forme :

```

ot+3i9DAgBkcRcAtj0j4LaR0VknFBbVPFd5uRHg5h6n+u/N5GJG79G+dwfCMNYxd
AfvDbbnvRG15RjF+Cv6pgsh/76tuIMRQyV+dTZsXjAzcLAcmgQWpzU/qlULRuJ0//7
TBj0/VLZjmmx6BEP3o)Y+x1J96reIc8geMjgEtslQIxq/H5COEBKEveegeGTlg==
-----END CERTIFICATE-----

--> build_timeley = true
--> ct_compliance_status = "COMPLIES_VIA_SCTS"
t= 4565 [st= 201] QUIC_SESSION_CERTIFICATE_VERIFIED
--> subjects = ["*.google.com","*.android.com","*.appengine.google.com","*.cloud.google.com","*.crowdsource
t= 4565 [st= 201] QUIC_SESSION_CRYPTO_HANDSHAKE_MESSAGE_SENT
--> CHLO<
    SNI : "www.youtube.com"
    STK : 0x2ce962942ccc43f88d55b44ea58b11cea482b963d1cd7a602dd3e25ad6d9a07ea0697a7c91a5689664ef1e6a69264
    SNO : 0x2e696289a8a98cdb80ed6dab0032cc5d8bc8212ecb70589eaf5bc1ef5b1f3cdf2804faa602560b6e5cc237265559c
    VER : '0046'
    CCS : 0x01e8816092921ae07eed8086a2158291
    NONC: 0x5e998c02303030303030309e2b9d3d42927444ca295dbcfc14b360ee1337ae4
    AEAD: 'AESG'
    UAIID: "Chrome/81.0.4044.113 Linux x86_64"
    SCID: 0x303a4afa251506c0d955629ff48c122a
    TCID: 0
    PDMD: 'X509'
    SMHL: 0x01000000
    ICSL: 30
    NONP: 0x9efe5a32f0eadd11ae49d38d7eb2c7fb056bd67308d1829cd987362d2820851d
    PUBS: 0xbbaa7548367bac6e8a732894478823762aad417fe854c782f673b1c2b2fe1036
    MIDS: 100
    SCLS: 1
    KEXS: 'C255'
    XLCT: 0x1705be9726e06146
    CSCT: 0x
    COPT: 'SRTO', 'ACKD'
    CCRT: 0x1705be9726e061466032cb92a0414ddf
    IRTT: 147503
    CFCW: 15728640
    SFCW: 6291456
    >
t= 4565 [st= 201] QUIC_SESSION_STREAM_FRAME_SENT
--> fin = false
--> length = 1024
--> offset = 1024
--> stream_id = 1
t= 4565 [st= 201] QUIC_SESSION_PADDING_FRAME_SENT
--> num_padding_bytes = -1
t= 4565 [st= 201] QUIC_SESSION_PACKET_SENT
--> encryption_level = "ENCRYPTION_INITIAL"
--> packet_number = 3
--> sent_time_us = 110929264568
--> size = 1350

```

FIGURE 1.11 – Client Hello complet

En plus des champs de base du CHLO initial, on compte des champs supplémentaires, notamment :

- SCID (Server config ID)
- AEAD (l'algorithme choisi pour crypter)
- KEXS (l'algorithme d'échange de clés choisi)
- NONC (client-nonce codé sur 32bytes constitué de 4 octets de timestamp, 8 de server orbit et 20 de données pseudo aléatoires)
- SNO (Server nonce, si le serveur en a donné un, on peut voir que c'est effectivement celui qui apparaît plus haut dans le REJ)
- PUBS (valeur publique du client pour l'algorithme d'échange de clés)

Ceci étant, fait, le client peut alors communiquer au serveur en lui envoyant des données applicatives :

```

--> size = 1350
--> transmission_type = "NOT_RETRANSMISSION"
t= 4568 [st= 204] QUIC_CHROMIUM_CLIENT_STREAM_SEND_REQUEST_HEADERS
--> :method: GET
:authority: www.youtube.com
:scheme: https
:path: /?hl=fr&gl=FR
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.113
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
purpose: prefetch
x-client-data: CI22yQEIPrbJAQipncoBCNCvygEIVLDKAQjttcoBCI66ygEI5sbKARjat8oB
sec-fetch-site: cross-site
sec-fetch-mode: navigate
sec-fetch-dest: document
referer: https://www.google.com/
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9
cookie: [49 bytes were stripped]
--> quic_priority = 4
--> quic_stream_id = 5
t= 4568 [st= 204] QUIC_SESSION_STREAM_FRAME_SENT
--> fin = false
--> length = 470
--> offset = 0
--> stream_id = 3
t= 4568 [st= 204] QUIC_SESSION_PACKET_SENT
--> encryption_level = "ENCRYPTION_ZERO_RTT"
--> packet_number = 4
--> sent_time_us = 110929267657
--> size = 502
--> transmission_type = "NOT_RETRANSMISSION"
t= 4668 [st= 304] QUIC_SESSION_PACKET RECEIVED
--> peer_address = "172.217.22.142:443"
--> self_address = "192.168.0.24:48457"
--> size = 1350
t= 4669 [st= 305] QUIC_SESSION_UNAUTHENTICATED_PACKET_HEADER RECEIVED
--> connection_id = "0"
--> header_format = "IETF_QUIC_LONG_HEADER_PACKET"
--> long_header_type = "INITIAL"
--> packet_number = 3
--> reset_flag = 0
--> version_flag = 1
t= 4669 [st= 305] QUIC_SESSION_PACKET_AUTHENTICATED
t= 4669 [st= 305] QUIC_SESSION_STREAM_FRAME RECEIVED
--> fin = false
--> length = 1312
--> offset = 0
--> stream_id = 1

```

FIGURE 1.12 – Données applicatives

1.3.2.2 Connexion à un serveur déjà connu

Etudions le cas où le client souhaite se reconnecter à un serveur qu'il connaît déjà (PUBS du serveur, certifs, SCID, ...). La première étape du client est de vérifier le certificat du serveur en cache pour vérifier qu'il est toujours valide :

140963: QUIC_SESSION
www.google.com
Start Time: 2020-04-17 23:38:55.283

```
t=223753 [st= 0] +QUIC_SESSION [dt=23582+]
    --> cert_verify_flags = 0
    --> host = "www.google.com"
    --> port = 443
    --> privacy_mode = true
    --> require_confirmation = false
t=223754 [st= 1] SIGNED_CERTIFICATE_TIMESTAMPS_RECEIVED
    --> embedded_scts = "AO8AdQcyhgXMI6LnIb0h2b5k7mKJSBna9r6c0eySVmt74uQxgAAAXE2CUKAAAElwBGMEQCIG9t1Mqbf7t1M46FKFuCcThOsPiyoIccCNU
    --> scts_from_ocsp_response = ""
    --> scts_from_tls_extension = ""
t=223754 [st= 1] SIGNED_CERTIFICATE_TIMESTAMPS_CHECKED
    --> scts = [{"extensions": "", "hash_algorithm": "SHA-256"}, {"log_id": "sh4FzIuizYogTodm+Su5iUgZ2va+nDnsk1tLe+LkF4=", "origin": "Embedded", "type": "SCT"}]
t=223754 [st= 1] CERT_CT_COMPLIANCE_CHECKED
    --> certificate =
        -----BEGIN CERTIFICATE-----
        MIIDjQjCCCoqAwIBAgIQW/qkU0d4k0sIAAAAdeRUTANBgkqhkiG9w0BAQsFADBC
        MQswCQYDVQQGEwJUUEoBaG1UEChMRw29vZ2x1IFRydXN0IFNlcnzpY2zMRMw
        E0YDVQDDEwphVFMqQ0EGeqIU0sXB4XD1IMDQmDTIyTgyNIoDXT1mWdyHNEyNTgy
        NIowZjELMAKgA1UEBhNCVNVixZARBgNVAgTCkhgbGlm3juwEf+jauBgIVBACt
        DU1vdW50YWluIFZpZXcxEzRaBgNVAoTCkdvb2dsZSBNTMxFtATBqIVBAMMdcou
        Z29vZ2x1LnVbTBZMBMgByGSM49AgE GCCqGSM49AvEHA0IAIBISjJ/fj26gwHrU4
        dGQmM7D3E4XURj1vB/jIUvQqv+avYElopnA1S1C1GAL495Jft0xWfde.lxvczn
        028YFUiujggZMII1G1Ta0BgHvNQH08Af8EBAMCB4AvwYDVR01BAmwCgYIKwYBQ0U
        AwEwDAYVR0TQAj/BAiwdADdBgNHQ4FFgQj5+k14bjlFgkjx8wNu01SnsExbYw
        HwYDVR0jBBgF0AuUmH4hbDrzVsY8jkBug630j/SswZAY1kwYBBQUHQAEWDBW
        MCcGCCsGAQUFBzABhhtodHRwOi8vb2zNzcCsWa2ku29vZy9ndHMkbzEvkwYIKwYB
        BQUHAKGH2h0dHAGLywa2ku29vZy9n3Iy10dUuZFPMS5jcnQwgg5dgBgfVHREE
        ggSU1IEK1IMK15inb29nbGUu29t9ggQmfuzHJvawQuY29tghYqlmfwcvGuZ21u
        Z55nb29nbGUuY29t9gqIqlmlhs3Vklmdvbd2sZ55jb2b2GcouY33vd2rb3Vyy2Uu
        Z29vZ2x1LnVbYIKKi5nlmHvgg4lmdjcC5ndnQylmVlvbYIRKi5nY3BjZG4uZ3Z0
        MS5jb22C1iouZzdwatQyU28C1iouZzt29chBzlmNughYqLndvb2dsZ1hbmFs
        eXpRj3MuV29t9ggsqLmdvb2dsZ55jYY1LK15nb29nbGUuY2z9vZ29vZ2x1LnVb
        Lnluggq4lmdvb2dsZ55jbYsqCIOKi5nb29nbGUuY28udlwCdoyuZ29vZ2x1LnVb
        b5ShcoIPKi5nb29nbGUuY29tLm1fg8qlmdvb2dsZ55jb20udHKCDyouZ29v
        LmVbS55jb4IPKi5nb29nbGUuY29tLm14gg8qlmdvb2dsZ55jb20udHKCDyouZ29v
        Z2x1LnVbS552b0lK15nb29nbGUuY29tLm10gsgqlmdvb2dsZ55m
        coILK15nb29nbGUuHWCCyoouZ29vZ2x1LnVbS552b0lK15nb29nbGUuHWCCyoouZ29v
        Z2x1LnVbS552b0lK15nb29nbGUuY29tLm10gsgqlmdvb2dsZ55ubILLk15nb29n
```

FIGURE 1.13 – Vérification du serveur en cache

Une fois le certificat vérifié (chaîne de certification, signatures, révocation,...), le client envoie alors directement le full client hello :

```

--> build_timely = true
--> ct_compliance_status = "COMPLIES_VIA_SCTS"
t=223755 [st= 2] QUIC_SESSION_CERTIFICATE_VERIFIED
--> subjects = ["*.google.com","*.android.com","*.appengine.google.com","*.cloud.google.com","*.crowdsource.google.com","*.g.co",
t=223755 [st= 2] QUIC_SESSION_CRYPTO_HANDSHAKE_MESSAGE_SENT
--> CHLOK
    SNI : "www.google.com"
    STK : 0x658e11223665e8dc98d86305f33fc0b564f21cf67b410223a5051e34afc7cbdf4d812f60a5d7fc96e0e095d2f2aa43c7ab1b967ab929
    VER : 'Q46'
    CCS : 0x1e8816092921ae87eed8086a2158291
    NONC: 0x5e9a21ef3030303030302808a6c7733f4da0e62d1e208ca18b262e5be60e
    AEAD: 'AESG'
    Uайд: "Chrome/80.0.3987.163 Windows NT 10.0; Win64; x64"
    SCID: 0x303ada4fa251506c0d955629ff48c122a
    TCID: 0
    PDMD: 'X509'
    SHML: 0x01000000
    ICNL: 30
    NONP: 0xcae4e6fe80e8893d886a146584dbb83058c8ec513c62289e6638567a2f40763f
    PUBS: 0xce2a3de88b691ebc4f7f84168230462f141a6911a95db0e906a195b0a02f7f27
    MIDS: 100
    SCLS: 1
    KEXS: 'C255'
    XLCT: 0x1705be9726e06146
    CSCT: 0x
    COPT: '1PTO','PTOS','PLE1','PVS1','PSDA','6PTO','ACKD'
    CCRT: 0x1705be9726e061466032cb92a0414ddf
    IRRT: 137429
    CFCW: 15728640
    SFCW: 6291456
    >
t=223755 [st= 2] QUIC_SESSION_STREAM_FRAME_SENT
--> fin = false
--> length = 1024
--> offset = 0
--> stream_id = 1
t=223755 [st= 2] QUIC_SESSION_PADDING_FRAME_SENT
--> num_padding_bytes = -1
t=223755 [st= 2] QUIC_SESSION_PACKET_SENT
--> encryption_level = "ENCRYPTION_INITIAL"
--> packet_number = 1
--> sent_time_us = 996198980502
--> size = 1350
--> transmission_type = "NOT_RETRANSMISSION"
t=223756 [st= 3] QUIC_CHROMIUM_CLIENT_STREAM_SEND_REQUEST_HEADERS
--> :method: GET
      :authority: www.google.com
      :scheme: https
      :path: /async/dlllog?async=doodle:153205521,slot:22,type:1,cta:0
      user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163 Safari/537.36

```

FIGURE 1.14 – Full client hello

On reconnaît alors les différents paramètres du full client hello exposés plus haut dans le cas de la connexion à un serveur non connu. Ceci, fait, le client n'attend pas de réponse du serveur ! Il envoie directement ses paquets applicatifs https dont le contenu est crypté :

```

        . . . . .
--> size = 1350
--> transmission_type = "NOT_RETRANSMISSION"
t=223756 [st= 3] QUIC_CHROMIUM_CLIENT_STREAM_SEND_REQUEST_HEADERS
--> :method: GET
:authority: www.google.com
:scheme: https
:path: /async/ddlllog?async=doodle:153205521,slot:22,type:1,cta:0
:user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.163 Safari/537.
:sec-fetch-dest: empty
:accept: */*
:origin: chrome-search://local-ntp
:x-client-data: CKWlyQEiirbJAQimtskBCMG2yQEiQz3KAQi0oMoBCNCvygEivLDKAQiXtcoBC021ygEijrrKAQjmxs0BCJDHygEiy8fKARjZvcoB
:sec-fetch-site: cross-site
:sec-fetch-mode: cors
:accept-encoding: gzip, deflate, br
:accept-language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7
--> quic_priority = 1
--> quic_stream_id = 5
t=223756 [st= 3] QUIC_SESSION_STREAM_FRAME_SENT
--> fin = false
--> length = 390
--> offset = 0
--> stream_id = 3
t=223756 [st= 3] QUIC_SESSION_PACKET_SENT
--> encryption_level = "ENCRYPTION_ZERO_RTT"
--> packet_number = 2
--> sent_time_us = 996198981501
--> size = 422
--> transmission_type = "NOT_RETRANSMISSION"
t=223851 [st= 98] QUIC_SESSION_PACKET RECEIVED
--> peer_address = "216.58.204.132:443"
--> self_address = "192.168.0.27:52904"
--> size = 1350
t=223851 [st= 98] QUIC_SESSION_UNAUTHENTICATED_PACKET_HEADER RECEIVED
--> connection_id = ""
--> header_format = "IETF_QUIC_LONG_HEADER_PACKET"
--> long_header_type = "ZERO_RTT_PROTECTED"
--> packet_number = 1
--> reset_flag = 0
--> version_flag = 1
t=223851 [st= 98] QUIC_SESSION_PACKET_AUTHENTICATED
t=223851 [st= 98] QUIC_SESSION_VERSION_NEGOTIATED
--> version = "Q046"
t=223851 [st= 98] QUIC_SESSION_VERSION_NEGOTIATED
--> version = "Q046"
t=223851 [st= 98] QUIC_SESSION_ACK_FRAME RECEIVED
--> delta_time_largest_observed_us = 4980
--> largest_observed = 1
--> missing_packets = []
--> received_packet_times = []

```

FIGURE 1.15 – Paquets applicatifs cryptés

Le client et le serveur sont déjà en train de parler en http crypté. Il a fallu juste : revérifier le certificat et envoyer le full client hello. C'est un gain de temps immense par rapport au classique TCP+TLS.

1.3.2.3 Scénario de pertes de paquets

QUIC est un protocole basé sur UDP. Cependant, UDP n'a pas de mécanismes de récupération de paquet perdu. C'est alors au rôle de QUIC de rajouter ces services nécessaires. Le scénario est le suivant : on essaie de se connecter à un serveur qui ne supporte pas le protocole QUIC. Le client va essayer de se connecter, mais le serveur ne va pas répondre car il ne sait pas traiter des requêtes QUIC. Mettons en place notre client en utilisant ngtcp2. On tape :

```
$ examples/client -qlog-file logs.qlog telecom-paris.fr 443
```

On s'aperçoit alors sur la console que le client tente de renvoyer à plusieurs reprises la

requête initiale CRYPTO/PADDING. On arrête manuellement le programme et on upload notre fichier log sur qviz pour avoir un diagramme de la situation :



FIGURE 1.16 – Diagramme situationnel

On peut alors voir qu'il tente d'envoyer la requête, il attend une seconde, pas de réponse, il renvoie. Il attend cette fois 2 secondes, toujours rien. Il attend alors 4 secondes, etc... A chaque fois, il double le timeout pour espérer avoir une réponse du serveur. On peut aussi lire à gauche en violet que la taille de la fenêtre diminue : peut-être que le paquet n'est pas arrivé car la taille de la fenêtre est trop grande. Donc il la baisse progressivement. Ce scénario est important car il montre comment QUIC permet de gérer des paramètres de couche 4 alors qu'on se préoccupait jusque là principalement de ces fonctionnalités de couche 5.

1.3.2.4 Echec de négociation de version

Après avoir envoyé le premier CHLO au serveur, celui-ci répond en indiquant la version choisie, mais seulement si le serveur à une version correspondant à celle du client. Toujours avec ngtcp2, essayons de contacter youtube.com. On exécute : \$ examples/client -qlog-file logs.qlog youtube.com 443

Le programme s'arrête avec une erreur ERR_RECV_VERSION_NEGOCIATION. En effet, la version QUIC de ngtcp2 est ‘build-draft27’ et non ‘Q046’. Mais que s'est-il passé au niveau des échanges ? Voyons cela avec Wireshark :

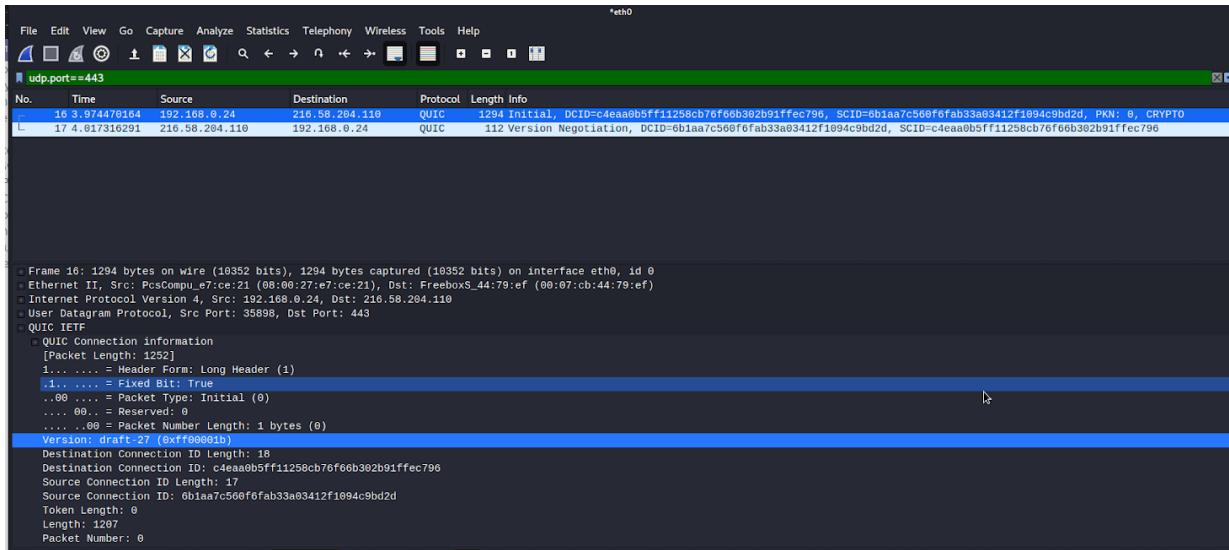


FIGURE 1.17 – Capture wireshark - version

Notre client ngtcp2 envoie le CHLO avec la version draft-27. Il reçoit alors un paquet de type Version Negociation contenant les différentes versions supportées par le serveur :

1.4. Comparaison de performance et de qualité de service avec TLS et TCP

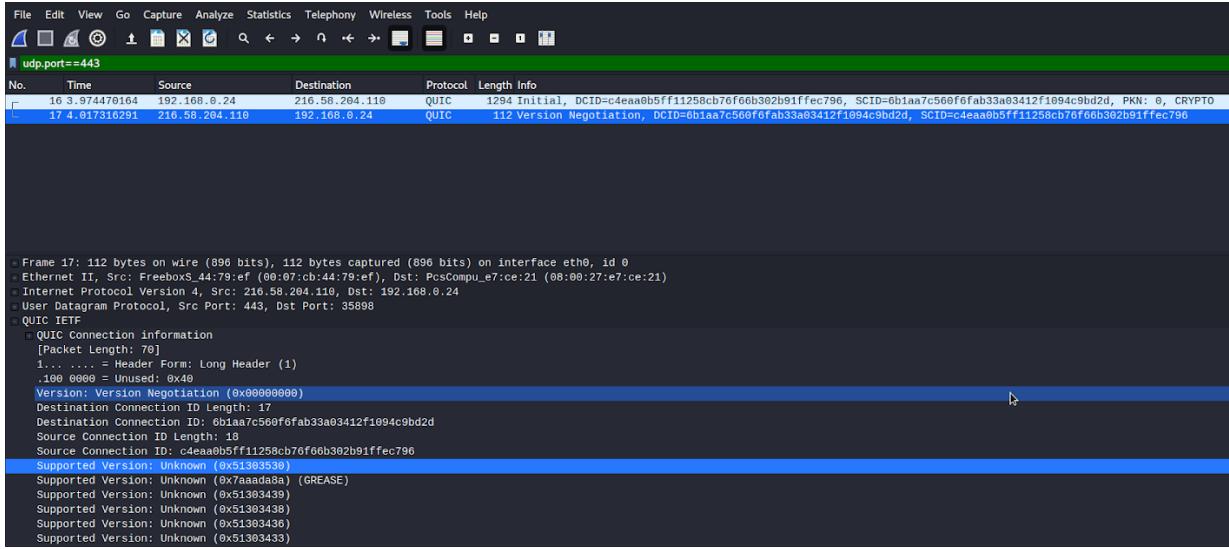


FIGURE 1.18 – Capture wireshark - version négociation

Aucune d'entre elles ne correspond à la version build-draft27. Le client s'arrête alors avec l'erreur ERR_RECV_VERSION_NEGOCIATION. On peut changer manuellement le champ Version envoyé au serveur avec l'option -v. Cependant, cela n'est d'aucune utilité : le serveur lit la version, s'attend à du Q046, mais lit du build-draft27 qu'il ne reconnaît pas. Il jette le paquet et ne répond même pas. C'est pour cela que côté client, le mécanisme de perte de paquets s'active.

1.4 Comparaison de performance et de qualité de service avec TLS et TCP

1.4.1 Similitudes

QUIC et TLS+TCP sont souvent comparés, au même titre que les protocoles applicatifs qui les mettent en oeuvre, respectivement HTTP/3 et HTTP/2 le sont aussi. Malgré toutes les différences d'implémentation et de rapidité qui les séparent, on peut toutefois noter quelques similitudes structurelles.

D'une part, les headers dans HTTP/2 et HTTP/3 ont des contenus très similaires. Les en-têtes, QPACK pour QUIC et HPACK pour HTTP/2 ont une conception très proche, et sont tous les deux des résultats de mécanismes de compression d'en-têtes pour paquets IP.

Une autre similitude au niveau de la communication est que les deux protocoles proposent une communication de type “Server push”, c'est-à-dire une communication initiée par le serveur, après autorisation préalable du client. C'est un mode de communication fondamental, utilisé notamment aussi dans les messageries instantanées : en effet, dès qu'un membre d'une conversation envoie un message, le serveur central l'envoie à tous les autres membres de la conversation, qui ont donc fait le choix d'y participer.

La dernière similitude notable se situe au niveau de la gestion de flux : les deux protocoles offrent une gestion de flux, notamment caractérisée par le multiplexage de plusieurs sessions et flux associés sur une seule connexion. Dès que deux machines communiquent, une session est créée et traitée sur le même lien que toutes les autres sessions. Toutefois, une gestion de priorité sur les flux des différentes sessions existe, et est prise en compte à la fois par HTTP/2 et HTTP/3.

1.4.2 Différences

Si QUIC est devenu populaire, c'est bien parce qu'il présente un certain nombre d'améliorations par rapport à son “rival” TCP + TLS. QUIC permet notamment de fluidifier le trafic HTTP, tout en le rendant sécurisé.

Une des déviations principales de QUIC vis-à-vis de TCP est qu'il inclut en plus du niveau transport le chiffrement et donc le transport sécurisé des données, ce que TCP seul ne peut gérer puisqu'il utilise TLS, situé plus haut dans le modèle OSI puisqu'il fait la liaison entre la couche transport et la couche applicative pour sécuriser les données.

En plus de cela, au niveau du service, la principale différence est que QUIC repose sur le protocole de transport UDP, non-orienté connexion, contrairement à TCP, et permet donc une communication de manière plus efficace et moins rigide que TCP.

Voici deux diagrammes protocolaires mettant en avant la différence entre un handshake géré par QUIC et un handshake classique avec TCP :

1.4. Comparaison de performance et de qualité de service avec TLS et TCP

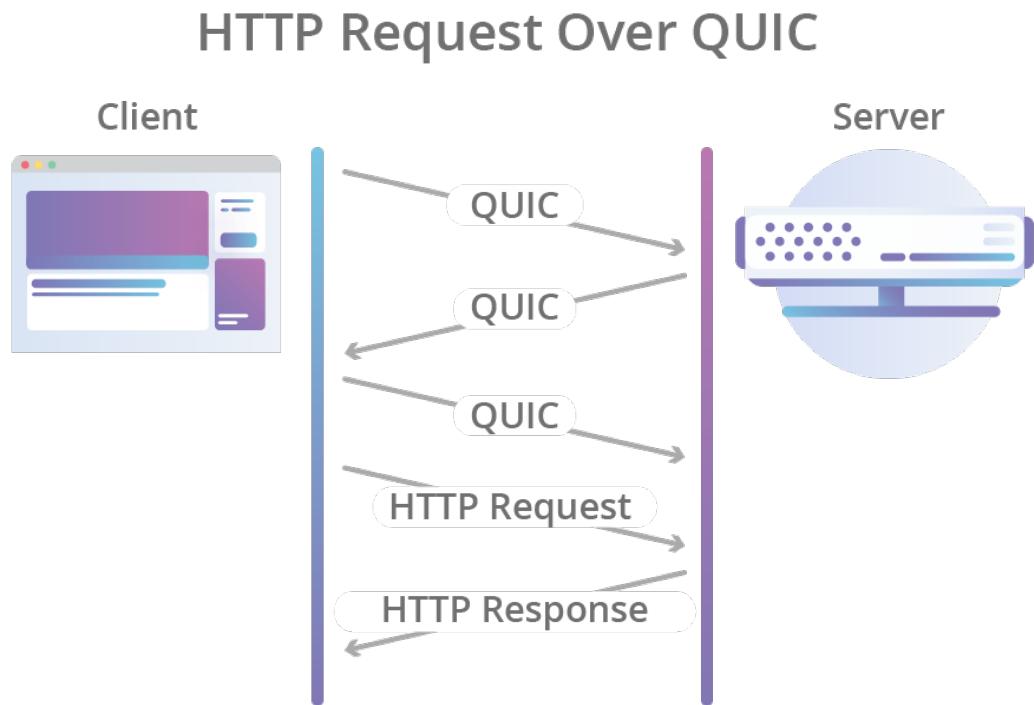


FIGURE 1.19 – Handshake avec le protocole QUIC

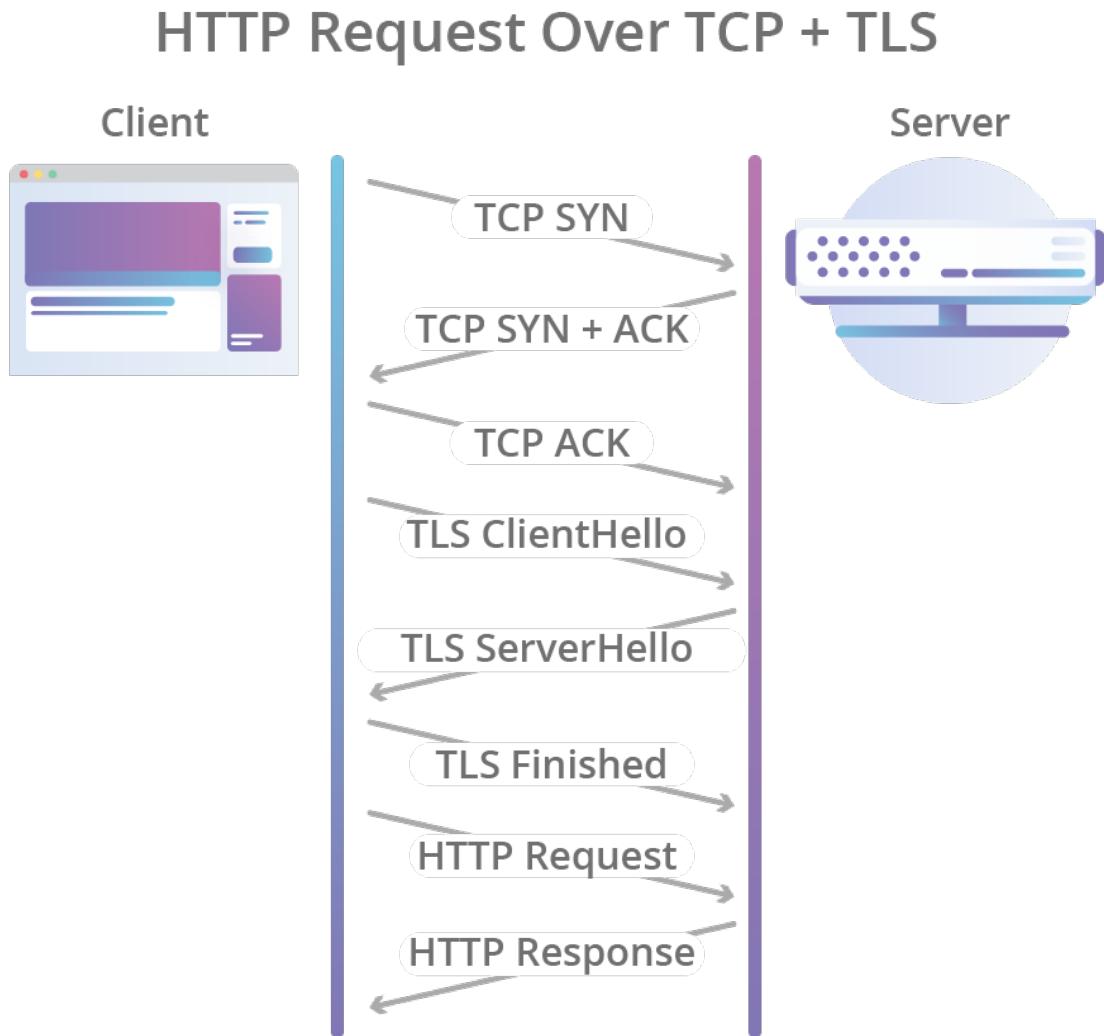


FIGURE 1.20 – Handshake avec le protocole TCP + TLS

Comme on peut le voir, pour initier la communication, QUIC combine un “three-way handshake” classique et un handshake TLS 1.3 . Cela permet à la fois l’authentification et la négociation des paramètres cryptographiques qui vont être utilisés par la suite durant la communication. Ainsi, QUIC remplace la couche TLS avec son propre protocole de chiffrement des données, et permet donc une initiation de la connexion deux fois plus

rapide qu'un TCP + TLS classique.

En plus de cela, QUIC va plus loin, en ajoutant un chiffrement des métadatas de connexion, permettant d'éviter une attaque du type "man in the middle". En chiffrant les numéros de paquets, QUIC permet de s'assurer qu'ils ne peuvent être utilisés par un tiers pour trouver des informations sur leur activité : seul le client et le serveur peuvent être au courant de leur communication.

1.5 Avantages et inconvénients du protocole QUIC

1.5.1 Avantages

Si le protocole QUIC est si utilisé actuellement, c'est parce qu'il présente une grande majorité d'avantages. Nous en présenterons les principaux.

Tout d'abord, le protocole QUIC est basé sur le protocole UDP au niveau transport. Ainsi, il permet de s'affranchir de tous les mécanismes d'acquittement des données transmises que nécessite TCP, et ainsi d'apporter de nouvelles solutions quand à la garantie de réception des données qui n'auraient été possibles avec les exigences de TCP, qui est orienté connexion contrairement à UDP.

Ainsi, le principal avantage de QUIC sur TCP est que la connexion est beaucoup plus rapide. En effet, pas besoin du "three-way handshake" pour initier la connexion, avec QUIC, la connexion est initiée avec un seul paquet (deux si c'est la première connexion avec le serveur), et en plus de cela, le chiffrement via TLS/SSL est assuré en même temps. Du deux en un qui permet de beaucoup gagner en rapidité donc en décongestion du trafic global.

Un autre avantage se situe au niveau de la détection d'erreurs : QUIC met en oeuvre un mécanisme de "Forward Error Correction", qui est une technique permettant de contrôler les erreurs de transmission notamment sur un canal bruyant ou endommagé. Le principe consiste à ce que l'émetteur envoie de manière redondante le message encodé, ce qui va permettre au receveur de pouvoir détecter des erreurs de transmission. Mieux encore, QUIC permet également la correction d'erreurs grâce à un code correcteur d'erreur (ECC), non seulement pour des données interverties/erronées, mais également pour des données manquantes : le message peut être totalement reconstruit. Tout ceci est bien plus avantageux qu'un simple système où le serveur demanderait au client de répéter le message : il n'y a qu'un seul aller du message et on y gagne en fluidité du trafic. Car c'est aussi un des principaux buts qui a donné naissance à QUIC : faire face au trafic HTTP croissant sur le net.

1.5.2 Inconvénients

Le principal inconvénient de QUIC découle de ses avantages : c'est la sécurité. En effet, bien que QUIC garantisse, et de manière plus efficace que TCP, le chiffrement ainsi que l'authentification des données, les en-têtes des paquets sont moins clairs, et des tâches telles que la régulation de trafic/réseau ou bien le dépannage deviennent plus difficiles en utilisant QUIC.

Par ailleurs, certaines études de performance ont montré que QUIC montre ses limites lors de transfert de grandes quantités de données dans des réseaux à très large bande passante.

1.6 Etude des différentes vulnérabilités

1.6.1 Introduction

Les paquets QUIC sont sensibles à de simples attaques, et notamment lors du handshake, introduisant ainsi de la latence, et donc vient contrer l'un des principaux avantages et buts recherché par QUIC : l'établissement d'une connection avec un 0-RTT. Plusieurs types d'attaques sont possibles : nous pouvons citer une exploitant les informations publiques provenant du serveur ou du client, et une autre exploitant les champs non protégés de paquet échangés lors du handshake.

1.6.2 Attaque de Rejeu

Tant qu'au moins un client établit une session avec un serveur en particulier, un attaquant peut apprendre les valeurs publiques (SCFG) du serveur ainsi que la valeur du token de l'adresse source (STK), correspondant ainsi au client pendant la période de validité. Un attaquant peut ainsi rejouer le SCFG correspondant au serveur au client, et rejouer le STK au serveur, et va ainsi compromettre les 2 parties.

Attaque de rejeu à l'aide du SCFG :

Un attaquant peut rejouer le SCFG avec chaque client envoyant une requête d'initialisation de connexion avec le serveur, tout en maintenant le serveur dans la méconnaissance de ces requêtes. Ainsi, ces clients établissant une connexion initiale avec ce serveur sans la connaissance de ce serveur vont voir leur paquets ensuite rejeté par celui ci car le serveur ne sera pas capable de les reconnaître. Bien que les données personnelles ne soient pas affectées ,un client souffrira ainsi de latence est d'un gaspillage de ressources.

Attaque de rejeu à l'aide du STK :

Un attaquant peut rejouer le STK d'un client vers le serveur, s'étant déjà servi de se token avec le client à de multiples reprises afin d'établir des connexions additionnelles. Cette action forcera ainsi le serveur à établir des clés ainsi mais également des clés de sécurisation pour chaque connaissance sans que le client en ait connaissance. Chaque étapes additionnelles dans le handshake vont ensuite échouer, mais un attaquant pourra ainsi initier une attaque DoS, en créant un grand nombre de connexions venant de multiples clients, et ainsi viendra épouser les ressources de calcul et de mémoire du serveur.

On peut voir que ces attaques sont permises grâce aux paramètres qui pourtant devait permettre la réduction de la latence. Ce genre d'attaque semble impossible à moins de limiter l'utilisation de ces jetons STK et SCFG à une utilisation unique, mais cependant interdirait l'établissement d'une connexion à 0-RTT.

1.6.3 Attaque par manipulation de paquet

Tous les champs d'un paquet QUIC ne sont pas forcément protégés contre une possible manipulation adverse. Un attaquant avec un accès au canal de communication utilisé par le client pour établir une connexion avec le serveur peut ainsi changer les bits des champs non protégés, notamment le connection id CID, ainsi que le token de l'adresse source STK. Cela pourrait ainsi mener le serveur ainsi que le client à dériver vers des clés différentes, et ainsi faire échouer l'établissement de la connexion. Afin de réussir une attaque, un attaquant doit s'assurer que tous les paramètres modifiés semblent cohérents dans chaque paquets envoyés et reçus de façon indépendante, mais incohérente lorsqu'on prend en compte les 2 endpoint (client et serveur). Ces attaques ne compromettent aucunement la confidentialité et l'authenticité de la communication qui est chiffré par la clé initiale, puisque même si les clés initiales soient différentes, elles restent inconnues par l'attaquant. Si client et serveur ne sont pas en accord sur la clé initiale, une clé de session QUIC ne peut pas être établie car le paquet SHLO (Server Hello) est chiffrée par la clé initiale.

Ces attaques de manipulation de paquet sont plus complexes que simplement compromettre et faire échouer le handshake car client et serveur peuvent ainsi progresser à travers le handshake tout en ayant une conversation incohérente, aboutissant ainsi à l'établissement de clés inconsistante.

Une façon d'atténuer ce type d'attaque serait de signer chacun des champs modifiables dans ses paquets `s_rejects` et `s_hello`. Cependant, ces signatures augmenteraient le coût de calcul de ces différentes signatures, et laissent l'opportunité à une attaque DoS, dans laquelle un adversaire, à l'aide du IP Spoofing, pourrait envoyer un grand nombre de requêtes d'initialisation de connexion au nom d'autant de client que souhaité.

Attack Name	Type	On-Path	Traffic Sniffing	IP Spoofing	Impact
Server Config Replay Attack	Replay	No	Yes	Yes	Connection Failure
Source-Address Token Replay Attack	Replay	No	Yes	Yes	Server DoS
Connection ID Manipulation Attack	Manipulation	Yes	No	No	Connection Failure; server load
Source-Address Token Manipulation Attack	Manipulation	Yes	No	No	Connection Failure; server load
Crypto Stream Offset Attack	Other	No	Yes	Yes	Connection Failure

FIGURE 1.21 – Attaques Découvertes et Propriétés

Adaptation de QUIC aux VPNs

2.1 Généralités sur les VPNs

Un VPN (Virtual Private Network) est une technologie réseau permettant de construire un réseau privé à l'intérieur d'une infrastructure publique. Celui-ci permet de créer un lien direct entre des ordinateurs distants. Celui-ci est aujourd'hui une composante majeure des réseaux informatiques et de l'informatique distribuée.

2.1.1 Principe Général

Un VPN peut être de type point à point (client-concentrateur), mais également sous la forme d'un réseau privé étanche et distribué de type MPLS. Celui-ci doit permettre :

- Une authentification utilisateur, autorisant l'accès au VPN à un certain nombre d'utilisateurs
- Une gestion d'adresses, dans laquelle chaque utilisateur dispose d'une adresse privée et confidentielle,
- Un cryptage des données, permettant de protéger les données transitant sur le réseau public,
- Une gestion de clés, permettant au client et serveur de générer et regénérer des clés,
- Une prise en charge multiprotocolaire.

Un réseau VPN est basé sur un protocole de tunneling, permettant de faire circuler les informations d'un émetteur d'un bout à l'autre du tunnel, et ce de façon cryptée. Ainsi, un utilisateur aura l'impression de directement se connecter à un réseau distant. Pour que ces données soient lisibles d'un bout à l'autre, le protocole de tunneling utilisé par toutes les composantes d'un VPN doit être le même. Plusieurs types de protocoles de tunneling existent, dont :

- PPTP : Point to Point Tunneling Protocol ;

- L2TP : Layer Two Tunneling Protocol ;
- IPSec .

Le principe du tunneling est d'établir un chemin virtuel après avoir identifié l'expéditeur et le destinataire. Ensuite, la source crypte les données et les achemine à l'aide de ce chemin virtuel. Les données à transmettre peuvent être traitées via d'autres protocoles qu'IP. Dans ce cas, le protocole de tunneling encapsule les données en ajoutant un en-tête. La transmission par tunnel est l'ensemble du processus d'encapsulation, de transmission et de décapsulation.

2.1.2 Types de VPN

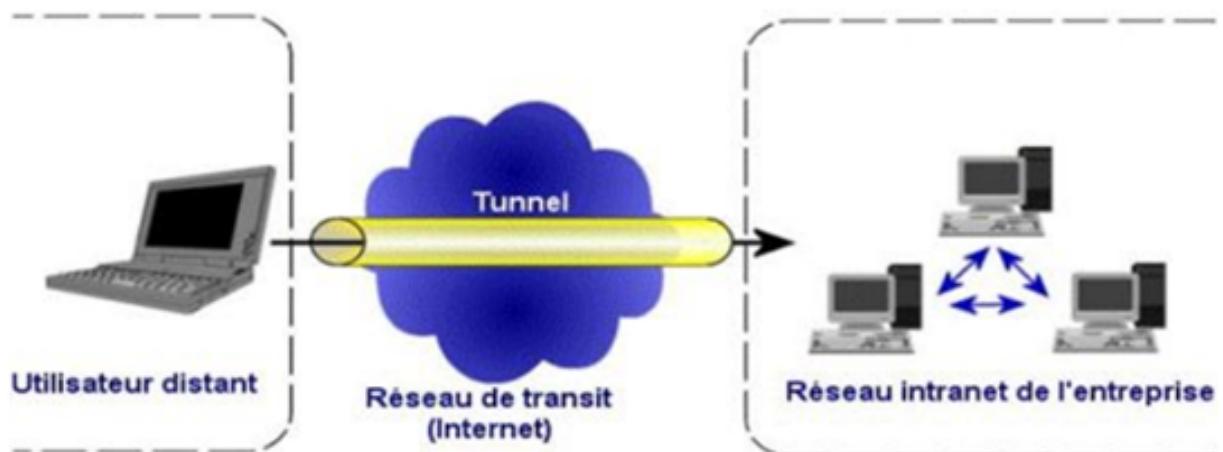


FIGURE 2.1 – VPN D'accès

Ce type de VPN permet à des utilisateurs itinérants d'accéder à un réseau privé. Ces utilisateurs doivent utiliser d'une connexion internet pour initier une connexion VPN. 2 cas sont possibles :

- L'utilisateur demande au fournisseur d'accès d'établir une connexion cryptée avec le serveur distant : il communique avec le NAS (serveur d'accès réseau) du fournisseur d'accès, et le NAS établit une connexion cryptée.
- Les utilisateurs disposent de leur propre logiciel client VPN. Dans ce cas, ils établissent directement la communication avec le réseau privé de manière cryptée.

Plusieurs avantages et inconvénients incombent à ces différentes méthodes :

- Si l'utilisateur passe par son fournisseur d'accès internet afin d'établir une connexion VPN avec le réseau distant, il pourra communiquer via plusieurs réseaux en créant plusieurs tunnels, mais cela exige le fait que le NAS soit compatible avec la solution adoptée par le réseau distant, et le client s'expose par ailleurs à des risques sécurité, étant donné le fait que la communication avec le NAS n'est pas cryptée ;

- A l'inverse, ce problème de sécurité n'apparaît pas dans la deuxième solution, puisque toute la communication est cryptée. En revanche, cela nécessite que chaque client transporte le logiciel.

2.1.3 VPN Site-to-Site

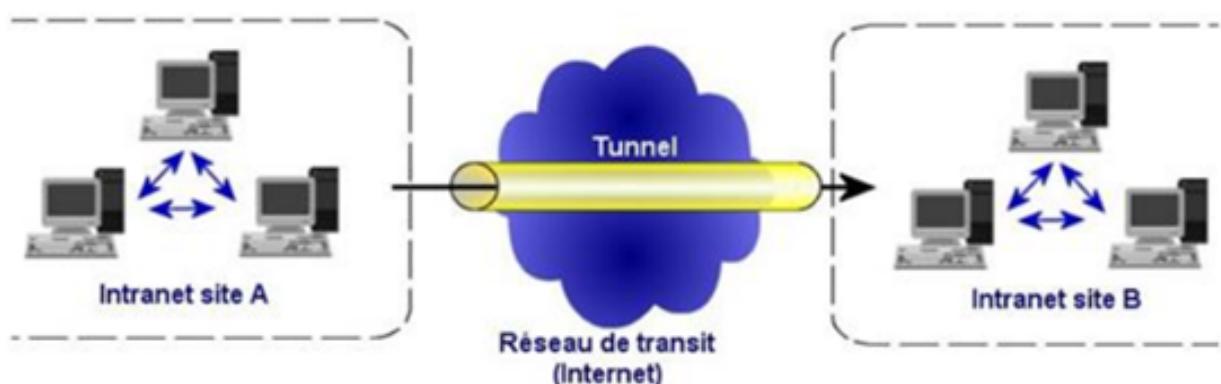


FIGURE 2.2 – VPN Site-to-Site

Ce type de VPN permet de relier 2 réseaux entre eux, et s'avère concrètement utile pour les entreprises ayant plusieurs sites distants. Etant donné l'utilisation très discrète de celle-ci, deux composantes majeures correspondent à la sécurité et l'intégrité des données, puisque des données sensibles et confidentielles peuvent être amenées à transiter sur ce réseau. L'authentification au niveau du paquet permettra d'assurer la validité des données, leur non-répudiation, ainsi que l'identification de l'émetteur des données. L'encapsulation IP ainsi que les algorithmes de cryptographie (et dont leur signature sont ajoutées au paquet) permettent d'assurer un niveau de sécurité suffisant.

2.1.4 Protocoles de Tunneling

2.1.4.1 IPSec

Le protocole IPSec correspond à un ensemble de protocoles définis par l'IETF, et permettant un haut niveau de sécurité de la couche réseau. Ce protocole est basé sur 2 mécanismes de protection des données (applicables sur IPv4 et IPv6), qui sont :

- AH, Authentication Header : assure l'intégrité et l'authenticité des datagrammes IP, sans chiffrer les données (Protocol ID : 51)
- ESP, Encapsulating Security Protocol : authentifie les données tout en les chiffrant. (Protocole ID : 50)

2.1.4.1.1 Authentication Header Celui-ci permet d'assurer l'authentification des données envoyées via un datagramme IP, en vérifiant l'identité des 2 extrémités du tunnel, mais également permet de s'assurer de l'intégrité des données. Elle vient directement s'insérer à la suite de l'entête IP.

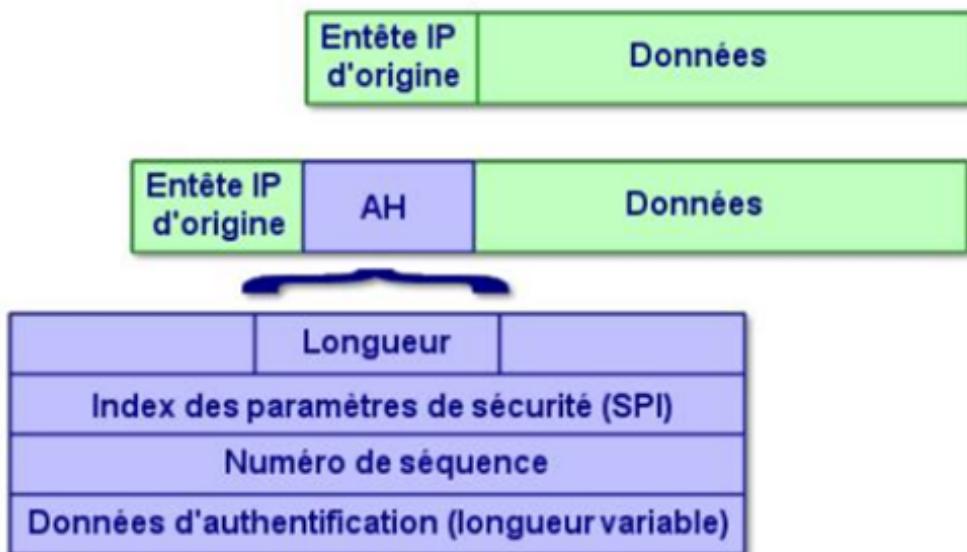


FIGURE 2.3 – Authentication Header

L'absence de confidentialité de ce mécanisme permettra de s'assurer que ce standard pourra être largement répandu sur internet. L'authenticité et l'intégrité des données est assurée par le bloc « Données d'authentification » sur le schéma ainsi que la signature du

paquet, ou plus communément appelé ICV (Integrity Check Value), dont le hash permettra de vérifier ces deux paramètres. De plus, le numéro de séquence permet de vérifier en réception si le mécanisme anti-rejet et bel et bien activé. Enfin, le champ SPI permet de d'identifier l'association de sécurité à utiliser pour le paquet. Les champs « SPI », « Numéro de séquence » et « Données d'authentification » sont tous les 3 codés sur 32 bits.

2.1.4.1.2 Encapsulating Security Payload (ESP) En plus d'assurer l'intégrité des données ainsi que l'authentification, ESP permet également d'assurer la confidentialité des données.

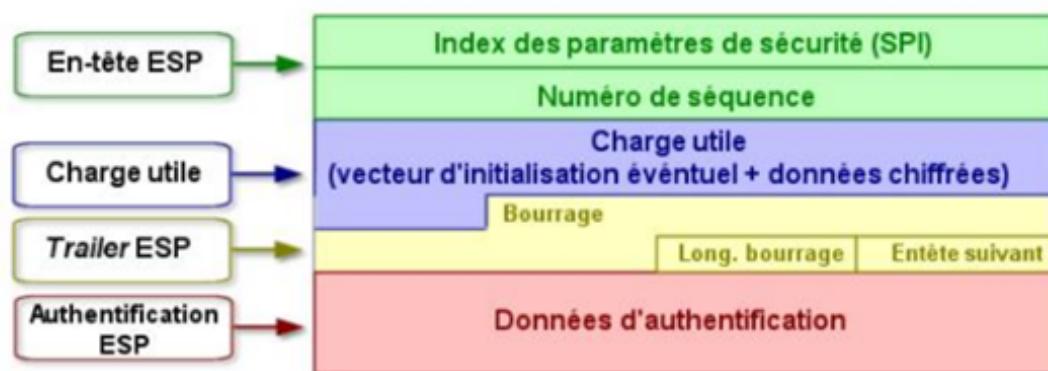


FIGURE 2.4 – Encapsulating Security Payload

La confidentialité via est assurée comme suit :

- 1) L'expéditeur vient encapsuler dans la « charge utile » les données transportées par le datagramme original, voir même l'entête IP dans le mode tunnel ;
- 2) Il vient ensuite ajouter un bourrage (pour s'aligner sur un format de 4 octets) si nécessaire ;
- 3) Le résultat (charge,bourrage,longueur et entête suivant) va ensuite être chiffré,
- 4) Ajoute éventuellement des données de synchronisation cryptographique.

L'intégrité et l'authentification des données sont assurés par les champs similaires au champs présentés dans le mécanisme AH.

2.1.5 Mode de fonctionnement

Ces 2 mécanismes peuvent être utilisés dans 2 modes différents :

- Mode « Tunnel » : la totalité du paquet IP est chiffrée/ authentifiée.
- Mode « Transport » : seules les données transférées sont chiffrées/authentifiées

2.1.5.1 Mode Tunnel

Dans ce mode, les données envoyées par l'application traversent la pile de protocole jusqu'à IP compris, puis sont envoyés dans le module IPSec. Cette encapsulation par le module IPSec permet ainsi le masquage d'adresse. Dans ce mode par défaut, le paquet IP est chiffré, puis un nouvel en-tête IP est ajouté puis est envoyé à l'autre bout du VPN (Paire VPN). Celui-ci est principalement utilisé pour les communications entre routeurs, ou entre terminaux et routeurs. Il permet ainsi de créer de nouveaux réseaux virtuels.

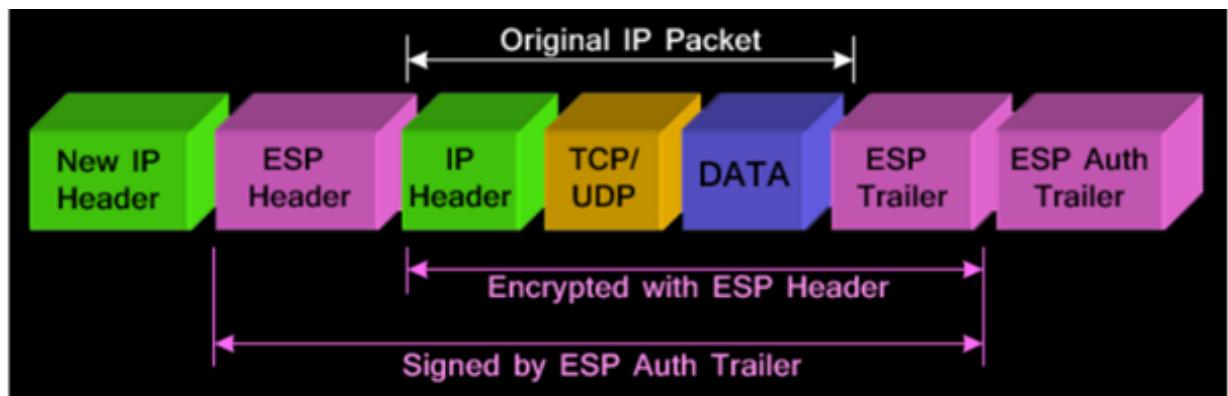


FIGURE 2.5 – IPSec Mode Tunnel avec ESP

Comme on peut le remarquer sur ce datagramme, le paquet entier est chiffré, et un nouvel en-tête externe est ajouté.

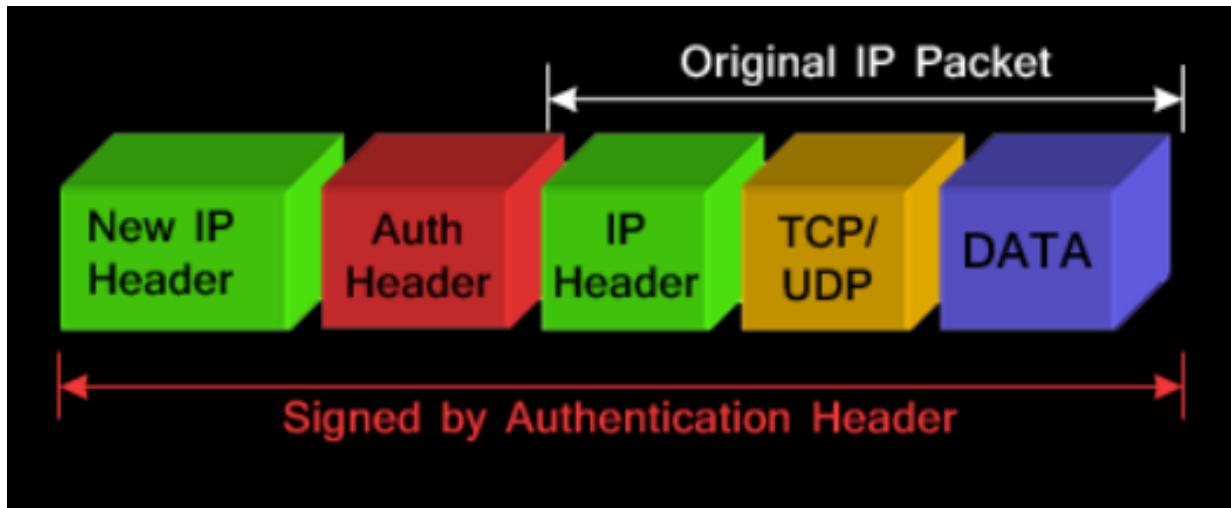


FIGURE 2.6 – IPSec Mode Tunnel avec AH

L'utilisation de AH en mode tunnel est d'authentifier l'intégralité du paquet IP. En revanche, puisque la confidentialité n'est pas assurée, ESP reste privilégié par rapport à AH.

2.1.5.2 Mode Transport

Dans ce mode, les mécanismes de sécurité sont appliqués au niveau de la couche transport (signature, chiffrement), dont le paquet résultant est transmis à la couche IP. L'insertion d'IPSec est ainsi transparente entre TCP et IP. TCP envoie ses données vers IPSec, envoyant à son tour vers IP. Celui-ci donc ne permet pas un masquage d'adresse, mais s'avère relativement simple à mettre en œuvre. Ce mode est principalement employé pour la communication entre 2 terminaux (client-serveur), mais aussi lorsqu'un autre protocole de « tunneling » (notamment GRE) est utilisé pour encapsuler le paquet de données IP. La protection des données est assurée via ce mode, composé d'un entête TCP ou UDP et des données, authentifiés et/ou chiffrés par AH ou ESP. L'entête IP original n'est pas modifié, sauf pour le protocole IP (50 : ESP, 51 : AH), et le protocole original est inclus dans le trailer ESP, qui sera utilisé lors du déchiffrement.

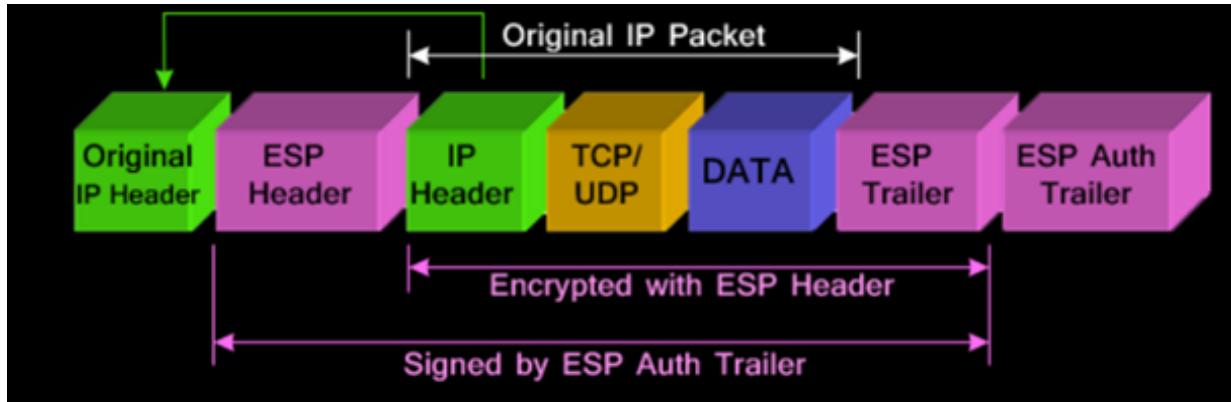


FIGURE 2.7 – IPSec mode Transport avec ESP

Comme on peut le voir sur ce datagramme, l'en-tête IP original est placé au début du paquet, avant l'en-tête ESP. Cela implique le fait que cet entête n'est pas chiffré, est n'assure donc pas le masquage d'adresses. L'utilisation d'ESP en mode transport permet donc uniquement de chiffrer les données utiles IP. Le protocole ID de ESP est 50.

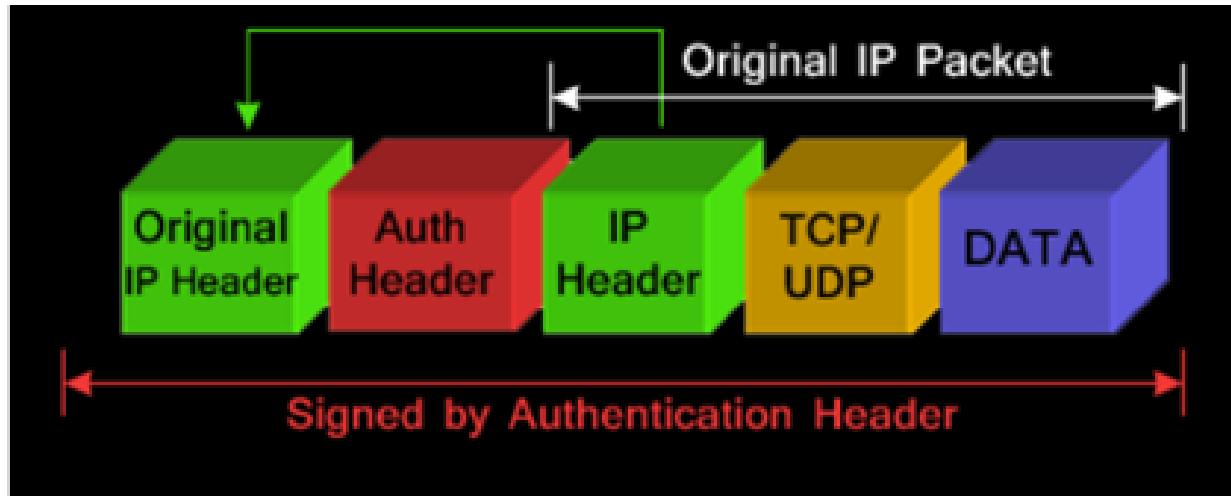


FIGURE 2.8 – IPSec mode Transport avec AH

Le but de AH est ici de protéger le paquet entier. Cependant, celui-ci place une copie de l'en-tête original au début du paquet au lieu d'en créer un nouveau, et ne fournit donc pas une protection des champs sensibles de l'en-tête IP (notamment Source IP et Destination IP). Le protocole ID de AH est le 51. Pour gérer ces mécanismes de sécurisation, une

gestion des clés doit être assurée par le protocole IPSec (génération, distribution, stockage et suppression). Afin d'assurer ce besoin, le système IKE (Internet Key Exchange) est employé afin de fournir des mécanismes d'authentification et d'échange de clés sur Internet.

2.1.6 Protocole IKE

Indépendant d'IP, le protocole IKE utilise UDP sur le port 500, et est utilisé pour :

- La négociation de protocoles de sécurisation ;
- L'authentification ;
- La génération/échange de clés.

Celui-ci se sert notamment de ISAKMP (Internet Security Association and Key Management Protocol), qui est utilisé pour la négociation, la modification et la suppression des associations de sécurité et de leurs attributs.

Une SA (Security Association) correspond à une connexion fournissant des services de sécurité au trafic transporté, et est identifiée par 3 composantes :

- Adresse de destination ;
- ID de protocole (AH ou ESP) ;
- SPI, correspondant à l'identifiant du SA.

Chaque SA est unidirectionnelle : ainsi, les 2 sens de communications sont protégés par un bundle de SA (correspondant à un ensemble de SA), 1 SA pour chaque sens de communication, mais également 1 par protocole de communication. Chaque SA est stockée dans une base de données des associations de sécurité (SAD), consultés à chaque fois que la couche IPSec reçoit des données à envoyer, permettant ainsi de fournir le mécanisme de sécurité à utiliser (si au préalable le paquet est accepté par le SPD, base de données des politiques de sécurité, indiquant si un paquet doit être accepté ou rejeté). ISAKMP comprend 3 composantes :

- Définition d'une façon de procéder : en 2 étapes (phases 1 et 2), il va tout d'abord (phase 1) définir un ensemble de paramètres de sécurité propre à ISAKMP afin d'établir un canal protégé, puis dans un second temps (phase 2) va définir le protocole de sécurité à utiliser (AH ou ESP)

- Définition des formats de message,
- Echanges types, permettant des négociations : PFS (Perfect Forward Secrecy), grâce à laquelle des clés utilisés antérieurement ou postérieurement ne peuvent dériver de la clé en cours d'utilisation.

IKE pour sa part utilise ISAKMP afin de pouvoir :

- 1) Etablir un 1er tunnel (tunnel administratif, ayant pour rôle de gérer les autres tunnels) entre les 2 entités ;
- 2) Etablir un 2nd tunnel (ou plusieurs a posteriori), correspondant au tunnel IPSec (ou tunnel de données).

Plusieurs modes pour IKE existent :

- Mode Principal ;
- Mode Agressif ;
- Mode Rapide ;

Les modes « Principal » et « Agressif » sont utilisés dans la phase 1, tandis que le mode « Rapide » est utilisé dans le processus d'échange de la phase 2.

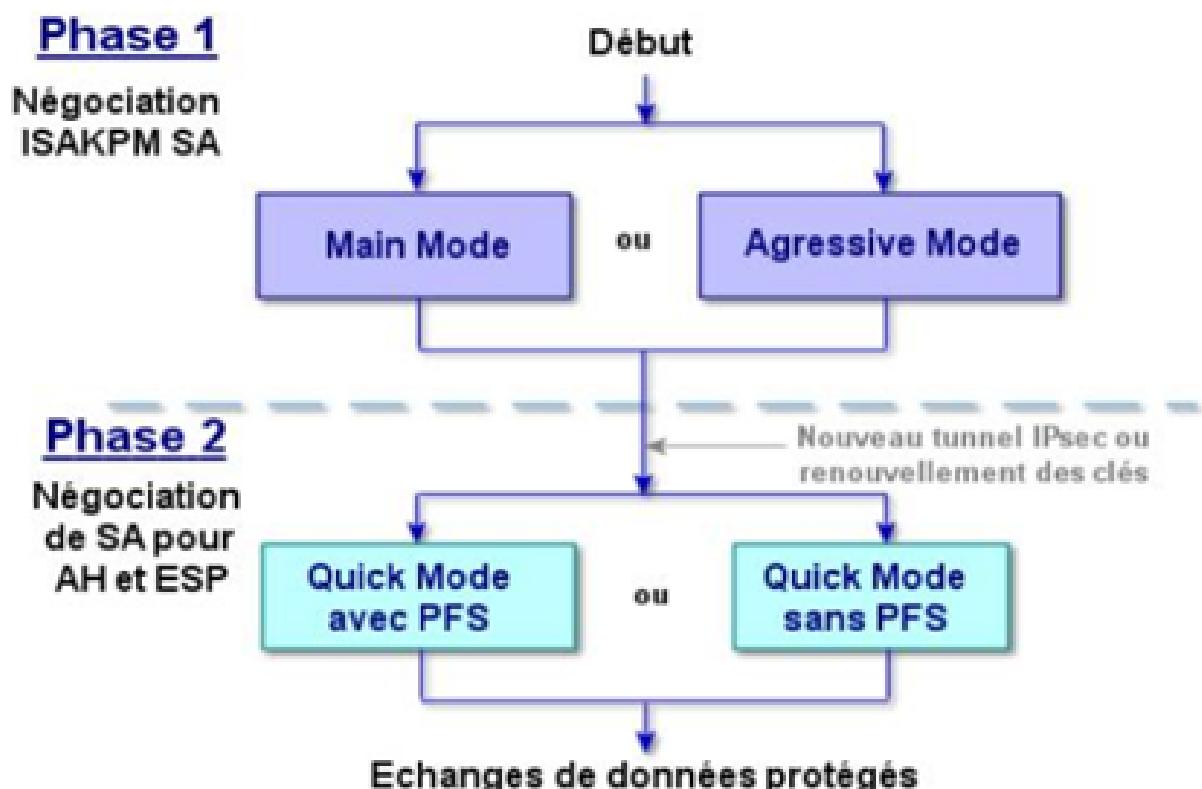


FIGURE 2.9 – Phases IKEH

2.1.6.1 Phase1

La phase 1 permet via ISAKMP de négocier un ensemble de paramètres :

- Algorithme de chiffrement ;
- Fonction de hachage ;
- Méthode d'identification
- Groupe Diffie-Hellman

La principale différence entre le mode « Principal » et le mode « Agressif » réside dans le nombre de messages envoyés durant cette phase. En effet, dans le mode « Principale », les paquets de négociations (Algorithme de chiffrement, SA, méthode d'identification) sont envoyés un à un, tandis que ceux-ci sont concaténés en dans le mode « Agressif », permettant une négociation beaucoup plus rapide.

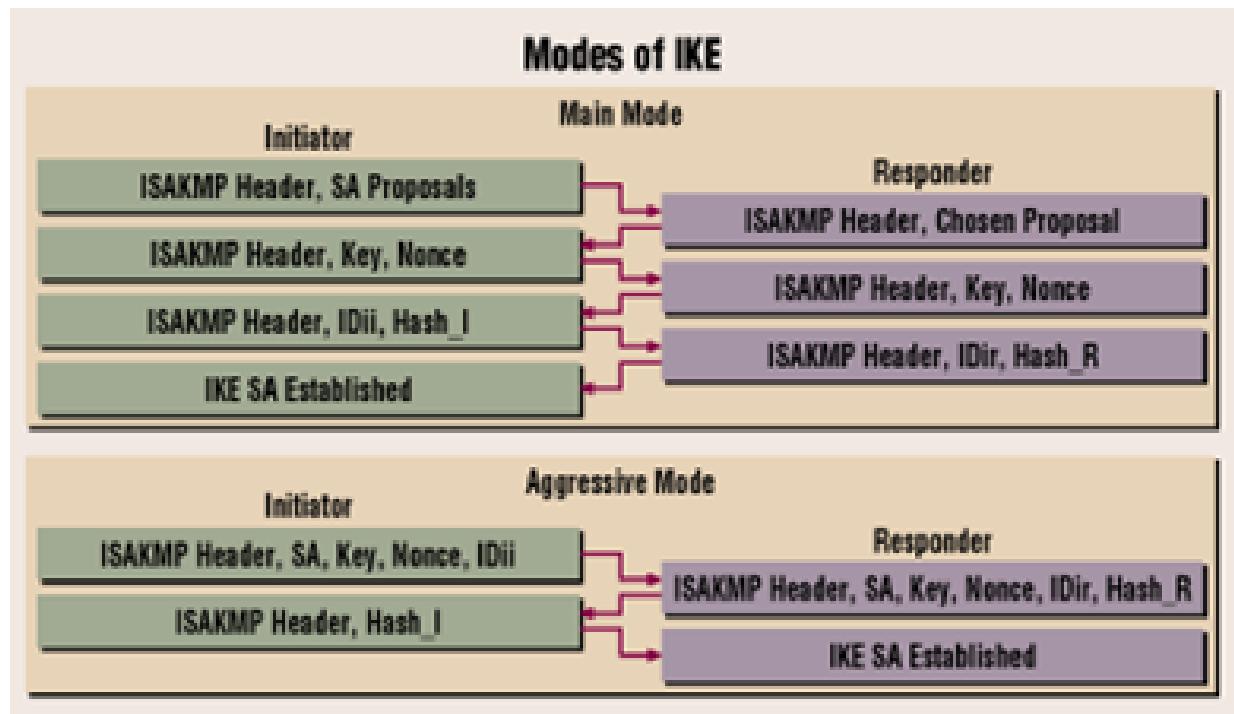


FIGURE 2.10 – Modes d'IKE Phase 1

A l'issue de cette phase, 3 clés sont générées : une pour le chiffrement, une autre pour l'authentification et une dernière pour la dérivation de futures clés. Cela permet ainsi d'établir une SA ISAKMP.

2.1.6.2 Phase 2

Les paramètres négociés dans la phase 1 permettent d'assurer l'authenticité et la confidentialité des éléments échangés et des messages envoyés lors de la phase 2. Le bloc « Hash » à la suite de l'entête ISAKMP permet d'assurer l'authenticité, et la confidentialité est assurée grâce au chiffrement de l'ensemble des blocs du message. Cette phase permet de négocier des SA pour d'autres protocoles de sécurité (notamment IPSec). Cette négociation permet la négociation de SA pour des protocoles de sécurité, au moins une pour chaque sens de communication (aboutissant à la négociation d'au moins 2 SA). Les échanges se font dans le mode « Rapide », et ont plusieurs rôles :

- Négocier un ensemble de paramètres IPSec ;
- Echanger des nombres aléatoires, permettant de générer de nouvelles clés dérivant du secret généré en phase 1 à l'aide de Diffie-Hellman.

IPSec est un protocole de couche 3, et est aujourd'hui très utilisé en lien avec les VPN. Cependant, bien que très utilisé, d'autres protocoles de « tunneling » sont également utilisés : on peut citer 2 catégories :

- Protocoles de couches 2 : PPTP et L2TP
- Protocoles de couche 3 (dont fait partie IPSec) : MPLS.

Nous allons ici nous intéresser à un autre protocole utilisé, et ayant permis la création d'un nouveau type de VPN : le SSL, ayant permis l'avènement du VPN-SSL.

2.2 Architecture des VPN SSL/TLS

Après avoir présenté ce qu'est un VPN et énuméré plusieurs types d'entre eux, il convient de s'attarder sur les VPN de type SSL (Secure Socket Layer), dont OpenVPN est un exemple. Notre but par la suite sera d'adapter un VPN de type SSL en y intégrant QUIC pour gérer à la fois l'encryption des données ainsi que leur transmission simultanément. Pour cela, il convient de comprendre le principe d'un VPN de type SSL/TLS.

Tandis que la plupart des VPN sont basés sur le protocole IPsec, le VPN SSL utilise comme son nom l'indique SSL pour chiffrer les données. Ce protocole fonctionne sur des ports classiques TCP/UDP et est généralement configuré sur le port 443, qui est le port “HTTPS”, associé au protocole HTTP sécurisé par SSL. A noter que sur UDP, on parle de “DTLS”.

2.2.1 Bref historique et intérêt

SSL (Secure Socket Layer) est un protocole de chiffrement et de sécurisation des échanges sur Internet, créé en 1994 par l'organisation Netscape. Il a donné lieu à 3 versions succes-

sives en 1994, 1995 et 1996, puis la troisième version SSL 3.0 a été reprise par l'IETF et remplacée en 1999 par TLS (Transport Layer Security), le successeur de SSL. Couramment, on utilise indifféremment SSL ou TLS ou encore SSL/TLS pour désigner le protocole de chiffrement largement utilisé par les communications actuelles. La version actuellement utilisée est SSL 3.0/TLS 3.1 .

Le clair avantage de SSL/TLS est son ergonomie : en effet, il s'intègre très bien aux protocoles couramment utilisés sur le net, notamment HTTP, qui n'a besoin que d'être très peu modifié pour permettre d'établir la connexion avec SSL. Ce dernier permet de faire la connexion entre la couche transport et la couche applicative. Couramment, on considère qu'il s'intègre au niveau session (niveau 5) du modèle OSI. Comme expliqué plus haut, la fusion entre HTTP et SSL donne lui au protocole HTTPS.

2.2.2 Fonctionnement SSL

L'architecture du protocole de chiffrement TLS/SSL est de type client-serveur. L'initiation d'une communication se déroule de la manière suivante :

- 1) Le client et le serveur s'accordent sur la version du protocole à utiliser
- 2) L'algorithme cryptographique lors des communications futures est choisi
- 3) Le client et le serveur s'authentifient réciproquement en s'échangeant puis validant des certificats digitaux.
- 4) Un chiffrement asymétrique est utilisé pour générer et partager la clé secrète partagée, afin de sécuriser sa transmission
- 5) Une fois cette clé partagée, le client et le serveur peuvent désormais communiquer en chiffrant leurs messages avec la clé partagée en 4). Le chiffrement est donc cette fois symétrique, ce qui est plus rapide qu'un chiffrement asymétrique. Voici un schéma détaillé montrant comment se déroule un handshake utilisant ce protocole :

Voici un schéma détaillé montrant comment se déroule un handshake utilisant ce protocole :

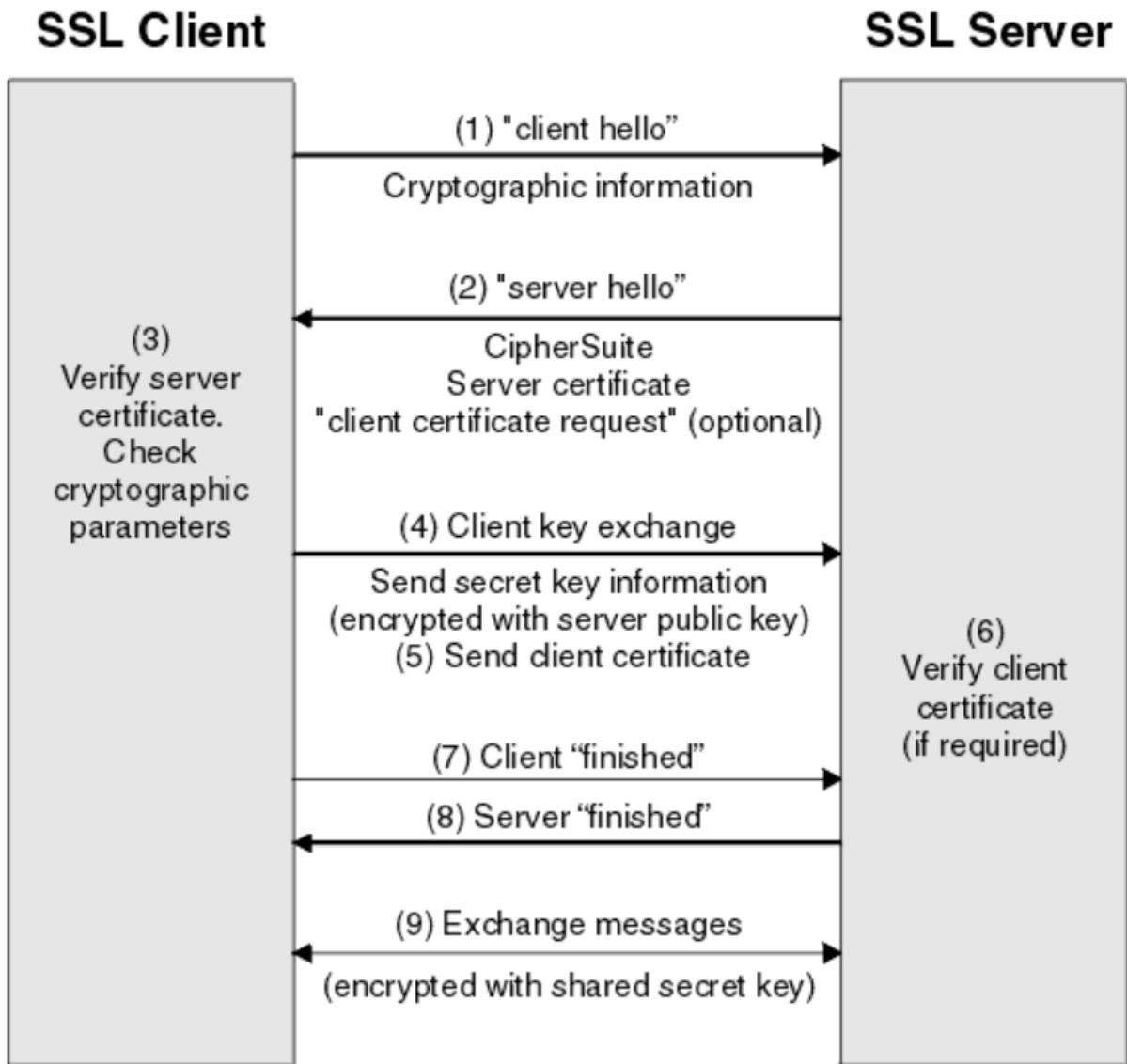


FIGURE 2.11 – Handshake SSL

Etape 1 : Le client SSL/TLS envoie un message “client hello” au serveur, dans lequel il liste des informations comme la version de SSL/TLS qu'il supporte ainsi que la listes des suites cryptographiques (ensemble des algorithmes de chiffrement supportés) par ordre de préférence, du plus robuste au plus faible. Il envoie aussi les méthodes de compression qu'il supporte. Par ailleurs, il envoie une chaîne d'octets aléatoires qui va permettre d'attaquer une attaque type “man in the middle” (cf étape 2)

- **Cipher Suites (34 suites)**

- Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)**
- Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)**
- Cipher Suite: TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0088)**
- Cipher Suite: TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA (0x0087)**
- Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)**
- Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)**
- Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)**
- Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)**
- Cipher Suite: TLS_RSA_WITH_CAMELLIA_256_CBC_SHA (0x0084)**
- Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)**
- Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)**
- Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)**
- Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)**
- Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)**
- Cipher Suite: TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x0045)**
- Cipher Suite: TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA (0x0044)**

FIGURE 2.12 – Exemple de Cypher Suites proposées lors d'un "Client hello"

Etape 2 : Le serveur répond avec un "serveur hello" qui contient la suite cryptographique qu'il a choisie parmi celles proposées par le client, ainsi que l'identifiant de session (session ID) et une chaîne aléatoire de 32 octets. Ces chaînes aléatoires sont donc importantes car elles vont permettre de vérifier que le client parle bien au serveur et réciproquement, comme nous l'expliquerons plus loin. Le serveur envoie aussi son certificat digitalisé (X.509), et peut en demander un de la part du client pour qu'il s'authentifie en envoyant une "requête de certificat du client" ("certificate request"), dans laquelle il précise les formats de certificats qu'il supporte. Il envoie également sa clé publique au client, qui sera nécessaire pour l'échange des clés ("server key exchange"). Il envoie enfin un "server Hello Done" indiquant que le ServerHello et les messages associés sont terminés.

Étapes 3, 4 et 5 : Le client vérifie le certificat digitalisé envoyé par le serveur. Si le serveur a envoyé une requête de certificat du client, le client envoie également une chaîne d'octets aléatoire chiffrée avec sa clé privée et contenant son certificat digital. (ou une "no digital certificate alert"). Dans certains handshakes, l'absence de cette authentification du client va empêcher l'établissement de la communication. Par ailleurs, le client envoie une autre chaîne d'octets aléatoires ("pre master secret") qui va servir, combinée au "client random" et "server random" (étapes 1 et 2), à générer le "master secret" qui sera la clé symétrique utilisée lors des communications futures. Cette chaîne "pre secret master" est chiffrée avec la clé publique du serveur. Toute cette étape est appelée "client key exchange".

On comprend alors le rôle des “random client” et “random server” au début. S’ils n’avaient pas eu lieu, toute la génération de la clé finale secrète serait revenue au client, qui envoie le “pre master secret”. Ainsi, si le serveur générât quant à lui des messages avec un chiffrement fixé par le client, dans une attaque type “man in the middle”, on pourrait imaginer un attaquant qui récolte la clé “pre master secret” envoyée par le client et se fait passer pour le serveur sans que cela soit détecté par le client (attaque “replay”). Pour cela, il est important qu’il y ait une construction de la clé basée des deux côtés, et différente à chaque communication : on parle de “génération d’aléa”. Si le serveur envoie une chaîne aléatoire au client au début, ce dernier peut vérifier par la suite que c’est bien le même serveur qui avait envoyé la chaîne random au début, puisque celle-ci est utilisée dans la conception de la clé finale “secret master” et que par ailleurs seul le serveur pouvait décoder le “pre master secret” avec sa clé privée.

Etape 6 : Le serveur SSL/TLS vérifie le certificat du client si demandé. Il décode le “pre master secret” avec sa clé privée (il est le seul à pouvoir le faire). Le master secret peut alors être calculé des deux côtés : $\text{master_secret} = \text{PRF}(\text{pre_master_secret}, \text{"master secret"}, \text{ClientHello.random} + \text{ServerHello.random})$ où PRF est une fonction pseudo-aléatoire. Ci-dessous un schéma expliquant en détail le calcul de ce “Master secret” :

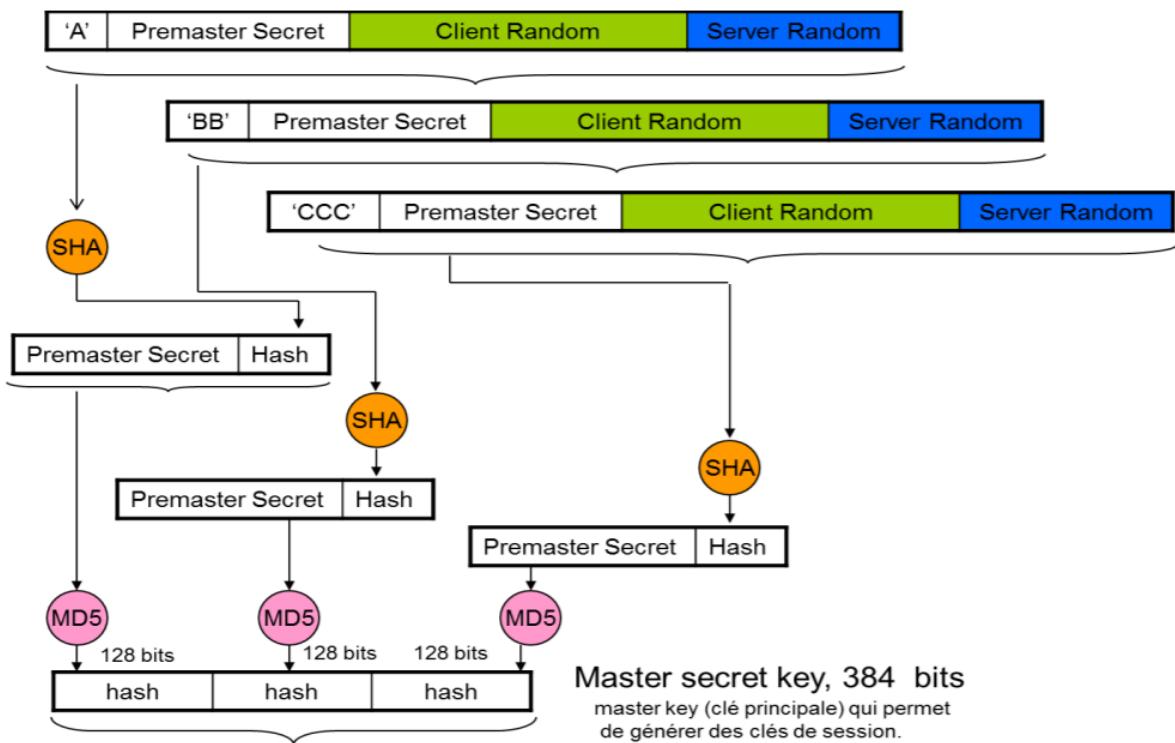


FIGURE 2.13 – Génération du Master secret

Etape 7 : Le client envoie alors au serveur un message “final” (“client finished”), qui est chiffré avec la nouvelle clé secrète calculée comme décrit ci-dessus (“change cipher spec”). Avec cette étape, le client indique que le handshake est terminé de son côté, et pourra vérifier que le serveur a bien pu décoder le message avec leur clé secrète commune.

Etape 8 : Le serveur envoie au client un message “final” (“server finished”) qui est aussi chiffré avec la nouvelle clé secrète commune (“change cipher spec”), et indique de cette manière que la handshake est également complété de son côté.

Etape 9 : Pour la suite de la conversation, le client et le serveur peuvent désormais échanger des messages chiffrés de manière symétrique avec la clé secrète partagée.

Lors des prochaines connexions, de nouvelles clés pourront être dérivées à partir du “Master Secret” en utilisant par exemple la fonction de hachage MD5 combinée à de nouvelles chaînes aléatoires générées côté serveur et côté client.

2.2.3 Gestion de l'authentification

Détaillons comment l'authentification a pu être possible lors de la communication sus-décrise.

Dans le cas de l'authentification du serveur : le client utilise la clé publique que le serveur lui a donnée lors du “server hello” afin de chiffrer les données qui seront (notamment le pre master secret) qui seront utilisées pour générer la clé secrète. Ainsi, le serveur peut générer la clé secrète avec le client si et seulement s'il peut décoder ces données avec sa clé privée. (qu'il est le seul à avoir).

Dans le cas de l'authentification du client : le serveur utilise la clé publique fournie dans le certificat du client afin de décoder les données que ce dernier lui a envoyées durant l'étape 5. Le principe est le même, l'authentification est alors assurée.

Les échanges des messages “finished” avec la clé partagée permettent de s'assurer que l'authentification est complète. De plus, l'authentification est complète si et seulement si chacune des étapes du handshake s'est déroulée correctement.

Le rôle des certificats permet aussi de garantir d'autant plus l'authentification. Leur gestion est assez complexe, mais ce qu'il faut retenir est que les certificats SSL jouent un rôle très similaire aux certificats client. Toutefois, dans le cas d'un client, le certificat est utilisé pour identifier le client/individu lui-même tandis que dans le cas d'un serveur le certificat authentifie le propriétaire du site.

Si on appelle CA Client l'autorité de certification du Client et CA Serveur l'autorité de certification du Serveur, dans le cas d'une authentification client et serveur, le serveur a besoin :

d'un certificat personnel fourni par CA Serveur de sa clé privée d'un certificat du client

fourni par CA Client Le client quant à lui a besoin :

d'un certificat personnel fourni par CA Client de sa clé privée d'un certificat du serveur fourni par CA Server

On observe qu'en termes d'authentification, les échanges se font à chaque fois de manière similaire pour le client et le serveur.

Les certificats contiennent diverses informations comme un numéro de série, une date de validité, des données personnelles sur le propriétaire (nom, adresse, e-mail...)

2.2.4 Vérification des certificats

Il existe 4 étapes lors des vérifications de certificats :

la vérification de la signature digitale la “chaîne de certificats” est vérifiée : normalement le client et le serveur ont dû avoir des certificats intermédiaires la date d'expiration, d'activation et la période de validité sont vérifiées l'état de révocation du certification

2.2.5 Gestion de la confidentialité

D'une part, on observe que l'échange est basé sur une alternance de chiffrements symétriques et asymétriques, ce qui renforce la sûreté de la communication.

Afin de minimiser le risque en cas de vol ou d'altération de la clé secrète, cette dernière peut être renégociée périodiquement, et l'ancienne n'est alors plus valide. Le même principe de sûreté est utilisée lors de l'envoi de chaînes aléatoires comme expliquée ci-dessus, qui permettent d'éviter les attaques “man in the middle”.

Durant le handshake, le client et le serveur s'accordent sur la primitive cryptographique utilisée ainsi que la clé secrète qui sera donc utilisée seulement pour la session en cours. Tous les messages au cours de la session seront donc chiffrés à la fois avec la primitive choisie et la clé secrète partagée, ce qui permet de s'assurer que le message reste privé et ne sera pas intercepté. Les chaînes aléatoires ne font que diminuer encore plus le risque déjà faible d'interception.

Par ailleurs, le fait que le chiffrement utilisé pour transmettre la clé privée soit asymétrique évite tout problème de distribution de la clé (pas de risque d'interception).

2.2.6 Gestion de l'intégrité

L'intégrité est principalement permise par le calcul de condensats (hash) des messages avec clé secrète (hmac). Les fonctions de hachage utilisées sont fournies dans la liste des suites cryptographiques utilisées par le client, et c'est ensuite le serveur qui va faire le choix

final, qu'il communiquera au client. La fonction de hachage utilisée est donc clairement définie de part et d'autre. Pour ceci, il est toutefois important que la liste d'algorithmes cryptographiques supportés par le client ne soit pas de type "None", autrement l'intégrité est fortement compromise.

2.2.7 Gestion du non-rejet

Le non-rejet est assuré par les numéros de séquence contenus dans les paquets.

2.2.8 SSL appliqué aux VPNs : exemple d'OpenVPN

Un VPN implémentant le protocole de chiffrement SSL/TLS décrit plus haut est appelé un SSL VPN (Secure Sockets Layer Virtual Private Network).

Dans ce type de VPN, le tunnel est implanté au dessus de SSL.

Le principe général d'un VPN s'applique alors : les utilisateurs peuvent se connecter, de manière encore plus sécurisée qu'un VPN sans SSL, à un réseau distant comme leur réseau de leur entreprise.



FIGURE 2.14 – OpenVPN

Un exemple classique de VPN SSL est OpenVPN : c'est un VPN open source qui utilise un cryptage des données de type SSL/TLS contrairement à la plupart des Vpn (basés sur IPsec). En effet, OpenVPN est basé sur la boîte à outils cryptographiques OpenSSL qui permet d'implémenter SSL/TLS. Le chiffrement est généralement réalisé sur 256 bits.

Par ailleurs, un de ses avantages est qu'il peut fonctionner sur n'importe quel port côté serveur, on peut donc faire fonctionner les protocoles de niveau transport (TCP ou UDP) sur le port 80 (ou 443) en particulier. Le trafic VPN OpenSSL est par ailleurs assez

similaire au trafic HTTPS, ce qui le rend difficile à identifier et donc à bloquer. Le fait que tous les ports soient disponibles permet également à OpenVPN de contourner quasi tous les pare-feux existants.

Concernant la primitive cryptographique utilisée, OpenVPN fonctionne principalement sur un chiffrement de type AES256, réputé très robuste.

De plus, OpenVPN est multi-plateformes et peut donc être implémenté sur tous les systèmes d'exploitation. Néanmoins, un point important à souligner est que OpenVPN nécessite une application tierce pour installer le client sur son ordinateur : il n'est pas nativement pris en charge par les systèmes d'exploitation.

OpenVPN est réputé comme étant l'un des VPN les plus fiables, notamment grâce à sa couche SSL supplémentaire qui sécurise encore plus les communications. Il peut par ailleurs

A noter que OpenVPN fonctionne mieux sur UDP d'après son site officiel, par conséquent le serveur d'accès OpenVPN essaie d'établir en priorité des connexions UDP. Ce n'est que si ces connexion échouent que TCP sera utilisé. UDP est donc choisi par défaut lors de l'installation d'OpenVPN. Cela permet de gagner en rapidité par rapport à TCP, d'autant plus que la force d'OpenSSL (cryptage fort) fait aussi sa relative lenteur.

2.3 Implémentation du protocole QUIC VPN en python

2.3.1 Descriptif du protocole

Le but de cette partie est d'implémenter en python le VPN QUIC. Le protocole retenu pour la programmation est le suivant :

Initialisation des interfaces virtuelles des deux côtés

Le client initie avec le serveur un handshake QUIC.

Ensuite, le client est face à une interface où il doit taper son login et mot de passe. Ceux-ci sont envoyés via un stream QUIC en base64, la confidentialité étant déjà assurée par QUIC.

Le serveur lit dans le stream et vérifie que la combinaison login/mot de passe est bien autorisée. Si c'est le cas, il envoie au client un message "Authentication successful", sinon, "Authentication failed".

Puis, si l'authentification est réussie, le serveur lance un thread qui va être à l'écoute de tout ce qui est envoyé par le noyau sur l'interface virtuelle pour encapsuler les paquets IP reçus dans un stream QUIC et envoyer ceci au client. Le serveur se met aussi à l'écoute de tous les événements QUIC envoyés par le client pour les décapsuler et réécrire sur l'interface virtuelle.

Côté client, s'il reçoit un "Authentication successful" venant du serveur, il se met à l'écoute des paquets IP envoyés par le noyau sur son interface virtuelle pour encapsuler dans du QUIC et l'envoyer au serveur. Il se met aussi à l'écoute des stream QUIC envoyés par le serveur pour les décapsuler et écrire sur l'interface virtuelle.

On peut rajouter par la suite les routes nécessaires.

Dans le cadre de notre implémentation, nous avons réalisé un VPN client-to-server.

2.3.2 Implémentation

Nous nous sommes basés sur l'implémentation de QUIC de aioquic.

(<https://github.com/aiortc/aioquic>). Nous avons retenu cette implémentation car elle propose une API pour programmer des applications basées sur QUIC. Pour le traitement des interfaces virtuelles, nous avons utilisé pytun (<https://github.com/montag451/pytun>). Notre code est disponible en annexe.

Présentons les parties principales du code :

```
#initialize virtual interface tun
tun=TunTapDevice(name='mytunnel', flags=pytun.IFF_TUN|pytun.IFF_NO_PI)
tun.addr='10.10.10.1'
tun.dstaddr='10.10.10.2'
tun.netmask='255.255.255.0'
tun.mtu=1048
tun.persist(True)
tun.up()
```

Nous mettons en place une interface virtuelle 'mytunnel' en configurant notamment le type d'interface (tunnel), les adresses ip source et destination, le masque de réseau, la MTU. Nous mettons la MTU à 1048 octets car c'est la quantité maximale qui peut être transporté sur un stream QUIC. Si on met plus, il y a fragmentation côté QUIC.

Nous avons ensuite la méthode suivante qui va être exécutée pour permettre de se connecter au serveur via le handshake QUIC puis de lancer la requête (méthode query) d'authentification :

```
async def run(
    configuration: QuicConfiguration,
    host: str,
    port: int,
) -> None:
    logger.debug(f"Connecting to {host}:{port}")
```

```

async with connect(
    host ,
    port ,
    configuration=configuration ,
    session_ticket_handler=save_session_ticket ,
    create_protocol=VPNClient ,
) as client :
    client = cast(VPNClient , client)
    logger.debug("Sending connection query")
    await client.query()

```

Le client va alors, après le handshake, effectuer une requête d'authentification qui suit :

```

async def query(self) -> None:
    #client authentication using login/password , clear text because
    #already encrypted by QUIC
    global STREAM_ID
    login = input("login: ")
    password = input("password: ")
    conc = login + ":" + password
    conc = conc.encode('utf-8')
    auth = base64.b64encode(conc)
    query = auth
    STREAM_id = self._quic.get_next_available_stream_id()
    end_stream = False
    self._quic.send_stream_data(STREAM_ID, bytes(query), end_stream)
    waiter = self._loop.create_future()
    self._ack_waiter = waiter
    self.transmit()

    return await asyncio.shield(waiter)

```

Le client doit alors rentrer un login et mot de passe, ceux-ci sont codés en base64 séparés par un “：“, puis envoyé dans un stream QUIC.

Côté serveur, nous avons une méthode quic_event_received qui va être appelée dès qu'un événement QUIC est reçu provenant du client.

```

def quic_event_received(self , event):
    global COUNT,tun , STREAM_ID
    if isinstance(event , StreamDataReceived):

        if(COUNT==0):
            #authentication check

```

```

        data = self.auth_check(event.data)
        end_stream = False
        STREAM_ID = event.stream_id
        self._quic.send_stream_data(event.stream_id, data, end_stream)
        self.transmit()

#if auth successful, start reading on local tun interface and
#prepare to receive QUIC|IP|QUIC
if(data==bytes("Authentication_succeeded","utf-8")):
    t=threading.Thread(target=self.tun_read)
    t.start()
    COUNT=1
else:
    #QUIC event received => decapsulate and write to local tun
    answer=event.data
    tun.write(bytes(answer))

```

Tout d'abord, le serveur reçoit la trame QUIC, il la décode avec la méthode auth_check :

```

def auth_check(self,payload) :
    decoded_auth = base64.b64decode(payload).decode("utf-8", "ignore")
    login=decoded_auth.partition(":")[0]
    password=decoded_auth.partition(":")[2]
    bool= (login=="root" and password=="toor")
    if(bool):
        return bytes("Authentication_succeeded","utf-8")
    else:
        return bytes("Authentication_failed","utf-8")

```

Le serveur décode la base64, sépare selon le “ : ” et autorise si il a pour login root et mot de passe toor. Il renvoie Authentication succeeded si c'est le cas, Authentication failed sinon. Il le renvoie codé en octets pour préparer l'écriture dans le stream QUIC.

Le serveur envoie alors la réponse au client. Si l'authentification s'est bien faite, le serveur lance le thread qui va être à l'écoute de l'interface virtuelle pour encapsuler dans du QUIC et envoyer au client :

```

def tun_read(self):
    global tun,STREAM_ID
    while True:
        #intercept packets that are about to be sent
        packet=tun.read(tun.mtu)
        end_stream = False

```

```
#send them through the appropriate QUIC Stream
self._quic.send_stream_data(STREAM_ID, bytes(packet), end_stream)
self.transmit()
```

Côté client une fois avoir reçu le Authentication succeeded, on lance le même type de thread. Une fois ceci fait, les 2 points clients et serveur peuvent commencer à s'échanger via le tunnel. Pour indiquer ce changement d'état, la variable globale COUNT est présente pour indiquer au client et au serveur la manière de traiter les streams reçus. Avant l'authentification, elle est 0, après, elle est à 1.

Si le client envoie un paquet au serveur via l'interface virtuelle “mytunnel”, il va être encapsulé dans du QUIC et envoyé par le thread tun_read au serveur.

Ce dernier va alors décapsuler ce stream et écrire le paquet IP dans l'interface virtuelle, d'où la partie de code présente dans la méthode quic_event_received :

```
else:
    #QUIC event received => decapsulate and write to local tun
    answer=event.data
    tun.write(answer)
```

Le même code est présent côté client pour qu'il puisse décapsuler les streams QUIC et les écrire sur son interface virtuelle.

2.3.3 Tests

Pour tester ce VPN, nous décidons de réaliser les tests suivants : ping, accéder à une page html présente sur le serveur via http, telnet, ssh.

Tout d'abord, on lance le serveur avec la commande suivante :

```
netgeek@NetLab:~/aioquic/examples
from CommandNotFound.CommandNotFound import CommandNotFound
File "/usr/lib/python3/dist-packages/CommandNotFound/CommandNotFound.py", line
9, in <module>
    import gdbm
ModuleNotFoundError: No module named 'gdbm'
netgeek@NetLab:~/aioquic/examples$ sudo python3.8 vpn_server.py -h
usage: vpn_server.py [-h] [--host HOST] [--port PORT] -k PRIVATE_KEY -c
                      CERTIFICATE [-q QUIC_LOG] [-v]

VPN over QUIC server

optional arguments:
-h, --help            show this help message and exit
--host HOST          listen on the specified address (defaults to ::)
--port PORT          listen on the specified port (defaults to 784)
-k PRIVATE_KEY, --private-key PRIVATE_KEY
                      load the TLS private key from the specified file
-c CERTIFICATE, --certificate CERTIFICATE
                      load the TLS certificate from the specified file
-q QUIC_LOG, --quic-log QUIC_LOG
                      log QUIC events to a file in QLOG format
-v, --verbose         increase logging verbosity
netgeek@NetLab:~/aioquic/examples$ sudo python3.8 vpn_server.py --port 443 -k ..
	tests/ssl_key.pem -c ..tests/ssl_cert.pem
```

FIGURE 2.15 – Lancement serveur

On fait de même côté client :

```
root@kali:~/aioquic/examples# python3 vpn_client.py --host 192.168.0.31 --port 443 -k
```

FIGURE 2.16 – Lancement client

On a alors ensuite, l'authentification à réaliser :

```
root@kali:~/aioquic/examples# python3 vpn_client.py --host 192.168.0.31 --port 443 -k
2020-04-26 14:45:00,322 INFO quic [85b6e5d1e4309e57] ALPN negotiated protocol dq
2020-04-26 14:45:00,325 INFO client New session ticket received
login: root
password: toor
```

FIGURE 2.17 – Authentification

Une fois ceci fait, le tunnel est bien créé, on peut communiquer avec le serveur avec l'adresse de son interface virtuelle (10.10.10.2). Nous pouvons voir côté client que l'interface virtuelle a bien été initialisée :

```

root@kali:~/aioquic/examples# ^C
root@kali:~/aioquic/examples# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.27 netmask 255.255.255.0 broadcast 192.168.0.255
        inet6 fe80::a00:27ff:fe7:ce21 prefixlen 64 scopeid 0x20<link>
            ether 08:00:27:e7:ce:21 txqueuelen 1000 (Ethernet)
                RX packets 107103 bytes 54322675 (51.8 MiB)
                RX errors 0 dropped 0 overruns 0 frame 0
                TX packets 115941 bytes 17723970 (16.9 MiB)
                TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    13:00.0: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
            loop txqueuelen 1000 (Local Loopback)
                RX packets 2939 bytes 247285 (241.4 KiB)
                RX errors 0 dropped 0 overruns 0 frame 0
                TX packets 2939 bytes 247285 (241.4 KiB)
                TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    25:0: flags=4241<UP,POINTOPOINT,NOARP,MULTICAST> mtu 1048
        inet 10.10.10.1 netmask 255.255.255.0 destination 10.10.10.2
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 txqueuelen 500 (UNSPEC)
            RX packets 950 bytes 133175 (130.0 KiB)
            RX errors 0 dropped 4 overruns 0 frame 0
            TX packets 3436 bytes 179712 (175.5 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@kali:~/aioquic/examples# 
25 Avr

```

FIGURE 2.18 – Vérification interface virtuelle

On peut maintenant commencer nos tests :

2.3.3.1 ping

A partir du client, on essaie de ping 10.10.10.2. Le trafic est censé passer par l'interface virtuelle mytunnel :

2.3. Implémentation du protocole QUIC VPN en python

59

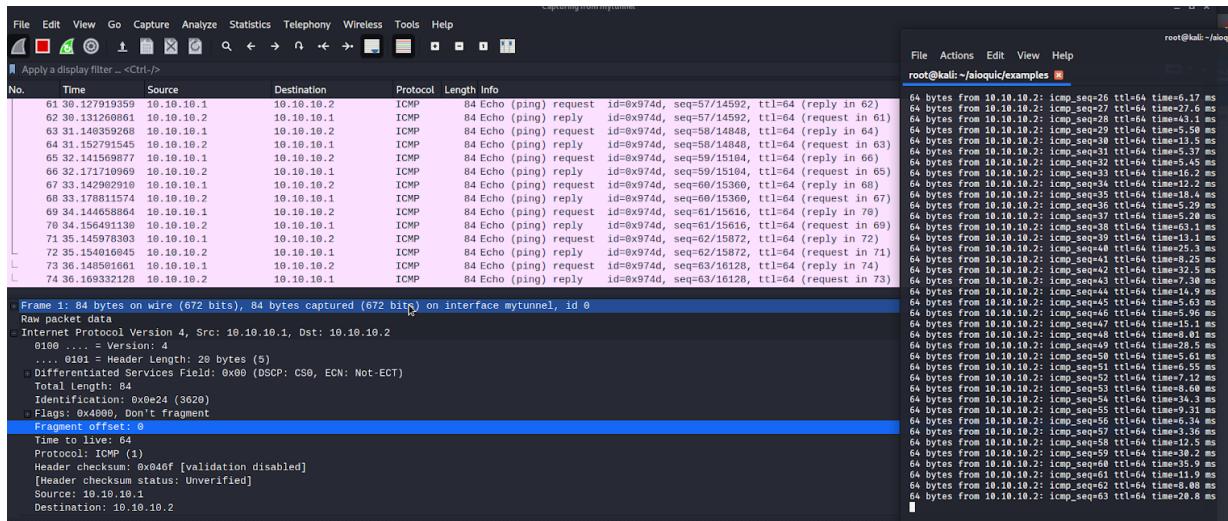


FIGURE 2.19 – Visualisation d'un ping avec wireshark

Comme le montre la capture, le ping se passe très bien. Les paquets ICMP echo reply et echo request sont bien capturés par Wireshark qui écoute sur l'interface virtuelle côté client.

2.3.3.2 HTTP

Côté serveur, on lance le serveur http sur le port 80 avec la commande ci-dessous :

```
netgeek@NetLab:~$ sudo python2 -m SimpleHTTPServer 80
[sudo] password for netgeek:
Serving HTTP on 0.0.0.0 port 80 ...
```

FIGURE 2.20 – Lancement serveur (HTTP)

Sur le client, dans un navigateur on tape “http ://10.10.10.2 :80” Toujours côté client, on voit que la page se charge rapidement et wireshark, les paquets capturés montrent qu'il n'y a pas eu besoin de retransmission :

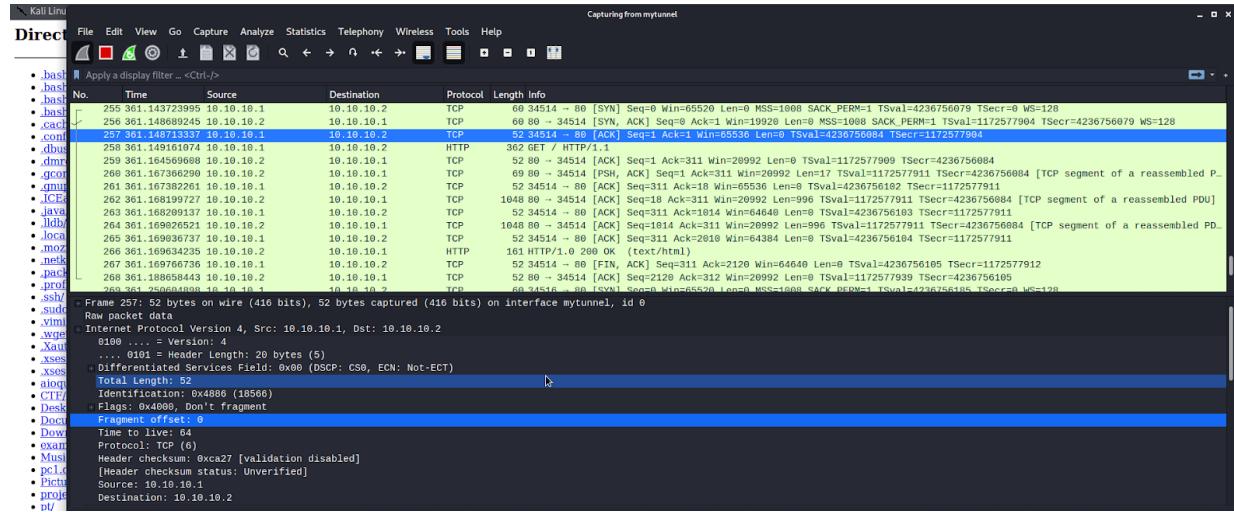


FIGURE 2.21 – Capture wireshark côté client

2.3.3.3 ssh

Un autre test avec ssh fonctionne :

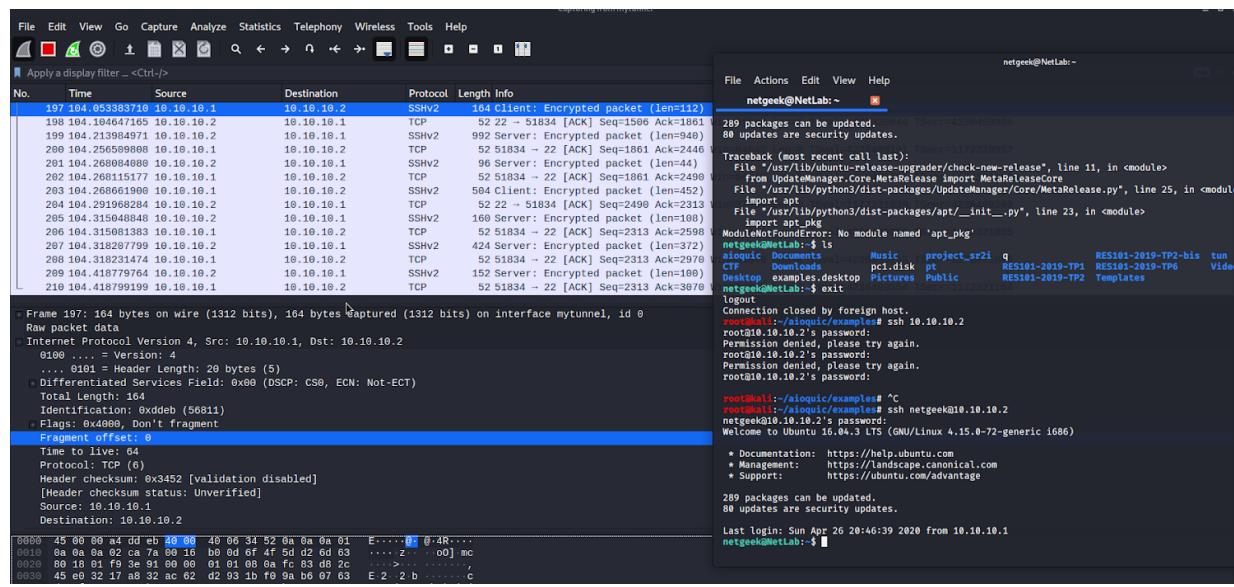


FIGURE 2.22 – test avec SSH

2.3.3.4 telnet

ssh est crypté contrairement à telnet qui envoie le trafic en clair. Cependant, avec le VPN QUIC, le trafic est crypté peu importe l'application ! On peut donc utiliser telnet pour limiter l'utilisation du réseau et surtout avoir une connexion distante plus rapide :

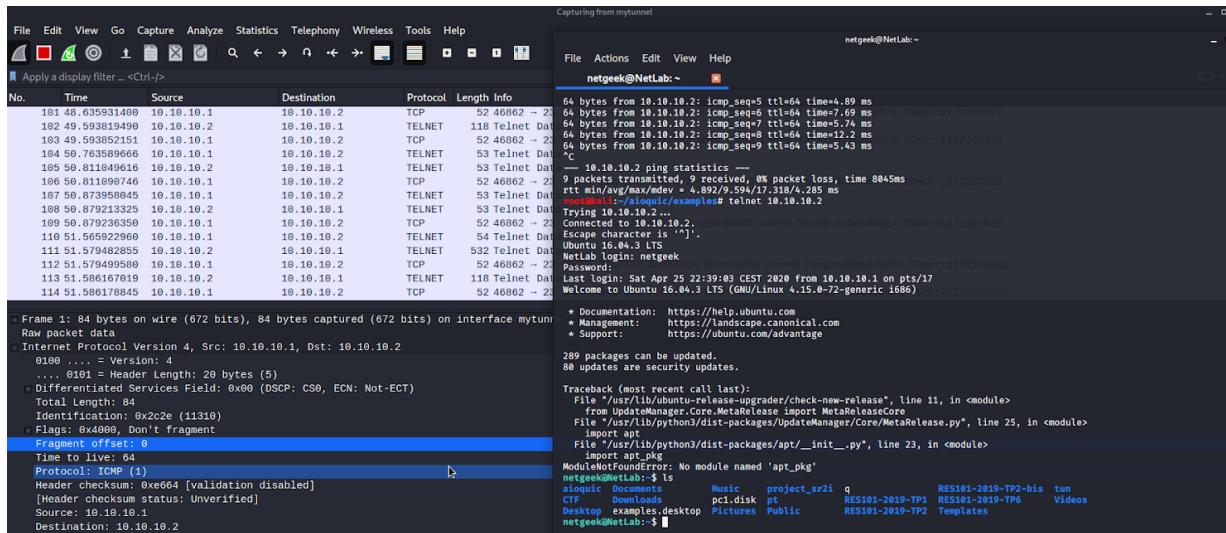


FIGURE 2.23 – Test avec Telnet

Il faut maintenant vérifier quelque chose d'important : vérifier que le trafic qui circule dans le réseau est bien crypté par QUIC. Pour cela : écoutons sur le port eth0 du client. En effet, le trafic est sur le point d'être envoyé via mytunnel. Celui-ci est intercepté par le VPN qui va alors l'envoyer via eth0. Côté serveur, il reçoit le stream QUIC sur eth0, décapsule, puis écrit sur l'interface virtuelle serveur.

55 10.0.0.55:55000	192.168.0.31	192.168.0.27	UDP	70 443 → 44981 Len=34
100 18.036509055	192.168.0.27	192.168.0.31	UDP	75 44981 → 443 Len=33
101 18.284204145	ChiconyE_6d:7d:99	Broadcast	ARP	60 Who has 192.168.0.27? Tell 192.168.0.31
102 18.284222076	PcsCompu_e7:ce:21	ChiconyE_6d:7d:99	ARP	42 192.168.0.27 is at 08:00:27:e7:ce:21
103 19.033704271	192.168.0.27	192.168.0.31	UDP	160 44981 → 443 Len=118
104 19.037746703	192.168.0.31	192.168.0.27	UDP	160 443 → 44981 Len=117
105 19.038288810	192.168.0.31	192.168.0.27	UDP	76 443 → 44981 Len=34
106 19.040295495	192.168.0.27	192.168.0.31	UDP	75 44981 → 443 Len=33
107 20.035355759	192.168.0.27	192.168.0.31	UDP	160 44981 → 443 Len=118
108 20.038851307	192.168.0.31	192.168.0.27	UDP	160 443 → 44981 Len=117
109 20.041118634	192.168.0.27	192.168.0.31	UDP	76 44981 → 443 Len=34
110 20.041682593	192.168.0.31	192.168.0.27	UDP	76 443 → 44981 Len=34
111 21.025415981	FreeboxS_44:79:ef	Broadcast	ARP	62 Who has 192.168.0.17? Tell 192.168.0.254
112 21.038127870	192.168.0.27	192.168.0.31	UDP	160 44981 → 443 Len=118

FIGURE 2.24 – Circulation trafic QUIC (détecté UDP)

On voit que le trafic intercepté en eth0 est interprété comme étant du UDP, ce qui n'est pas choquant car QUIC se base sur UDP comme support pour envoyer ses payload. Toujours dans Wireshark, allons dans Edit>Preferences>Protocols>QUIC et précisons le port à 443. On revient dans la capture, et maintenant on voit :

393	70.277100401	192.168.0.27	192.168.0.31	QUIC	76 Protected Payload (KP0)
394	70.277904109	192.168.0.31	192.168.0.27	QUIC	76 Protected Payload (KP0)
395	71.272934629	192.168.0.27	192.168.0.31	QUIC	160 Protected Payload (KP0)
396	71.285235211	192.168.0.31	192.168.0.27	QUIC	160 Protected Payload (KP0)
397	71.285265837	192.168.0.31	192.168.0.27	QUIC	76 Protected Payload (KP0)
398	71.287659793	192.168.0.27	192.168.0.31	QUIC	76 Protected Payload (KP0)
399	72.273920615	192.168.0.27	192.168.0.31	QUIC	160 Protected Payload (KP0)
400	72.277934811	192.168.0.31	192.168.0.27	QUIC	160 Protected Payload (KP0)
401	72.280384114	192.168.0.31	192.168.0.27	QUIC	76 Protected Payload (KP0)
402	72.281314662	192.168.0.27	192.168.0.31	QUIC	76 Protected Payload (KP0)
403	73.276393414	192.168.0.27	192.168.0.31	QUIC	160 Protected Payload (KP0)
404	73.292751309	192.168.0.31	192.168.0.27	QUIC	160 Protected Payload (KP0)
405	73.293207548	192.168.0.31	192.168.0.27	QUIC	76 Protected Payload (KP0)
406	73.295453310	192.168.0.27	192.168.0.31	QUIC	76 Protected Payload (KP0)

FIGURE 2.25 – Circulation trafic QUIC (détecté comme QUIC)

Le trafic a bien été détecté comme étant du QUIC et avec pour information le “Protected Payload” indiquant que le contenu est crypté.

Comparons la latence entre sans VPN et avec VPN. Sans VPN, nous avons le ping suivant :

```
root@kali:~/aioquic/examples# ping 192.168.0.31
PING 192.168.0.31 (192.168.0.31) 56(84) bytes of data.
64 bytes from 192.168.0.31: icmp_seq=1 ttl=64 time=3.77 ms
64 bytes from 192.168.0.31: icmp_seq=2 ttl=64 time=8.05 ms
64 bytes from 192.168.0.31: icmp_seq=3 ttl=64 time=2.58 ms
64 bytes from 192.168.0.31: icmp_seq=4 ttl=64 time=3.74 ms
64 bytes from 192.168.0.31: icmp_seq=5 ttl=64 time=15.9 ms
64 bytes from 192.168.0.31: icmp_seq=6 ttl=64 time=3.00 ms
64 bytes from 192.168.0.31: icmp_seq=7 ttl=64 time=12.6 ms
64 bytes from 192.168.0.31: icmp_seq=8 ttl=64 time=15.3 ms
64 bytes from 192.168.0.31: icmp_seq=9 ttl=64 time=14.1 ms
```

FIGURE 2.26 – Ping aioquic sans VPN

Et avec le VPN, nous avons le ping suivant :

```
root@kali:~/aioquic/examples# ping 10.10.10.2
PING 10.10.10.2 (10.10.10.2) 56(84) bytes of data.
64 bytes from 10.10.10.2: icmp_seq=1 ttl=64 time=4.26 ms
64 bytes from 10.10.10.2: icmp_seq=2 ttl=64 time=12.0 ms
64 bytes from 10.10.10.2: icmp_seq=3 ttl=64 time=17.5 ms
64 bytes from 10.10.10.2: icmp_seq=4 ttl=64 time=29.0 ms
64 bytes from 10.10.10.2: icmp_seq=5 ttl=64 time=15.6 ms
64 bytes from 10.10.10.2: icmp_seq=6 ttl=64 time=14.3 ms
64 bytes from 10.10.10.2: icmp_seq=7 ttl=64 time=4.73 ms
64 bytes from 10.10.10.2: icmp_seq=8 ttl=64 time=4.45 ms
64 bytes from 10.10.10.2: icmp_seq=9 ttl=64 time=6.58 ms
64 bytes from 10.10.10.2: icmp_seq=10 ttl=64 time=5.54 ms
64 bytes from 10.10.10.2: icmp_seq=11 ttl=64 time=6.26 ms
64 bytes from 10.10.10.2: icmp_seq=12 ttl=64 time=8.86 ms
64 bytes from 10.10.10.2: icmp_seq=13 ttl=64 time=7.50 ms
64 bytes from 10.10.10.2: icmp_seq=14 ttl=64 time=7.31 ms
64 bytes from 10.10.10.2: icmp_seq=15 ttl=64 time=5.62 ms
64 bytes from 10.10.10.2: icmp_seq=16 ttl=64 time=4.39 ms
64 bytes from 10.10.10.2: icmp_seq=17 ttl=64 time=15.5 ms
64 bytes from 10.10.10.2: icmp_seq=18 ttl=64 time=13.2 ms
64 bytes from 10.10.10.2: icmp_seq=19 ttl=64 time=15.1 ms
64 bytes from 10.10.10.2: icmp_seq=20 ttl=64 time=54.0 ms
64 bytes from 10.10.10.2: icmp_seq=21 ttl=64 time=9.07 ms
64 bytes from 10.10.10.2: icmp_seq=22 ttl=64 time=21.2 ms
64 bytes from 10.10.10.2: icmp_seq=23 ttl=64 time=6.91 ms
64 bytes from 10.10.10.2: icmp_seq=24 ttl=64 time=9.72 ms
64 bytes from 10.10.10.2: icmp_seq=25 ttl=64 time=7.61 ms
```

FIGURE 2.27 – Ping aioquic avec VPN

Outre les moments où l'on a des pics suite à l'utilisation variable du réseau, on voit qu'au minimum, le ping prend 3ms sans VPN et minimum 4ms avec VPN. Il est normal que le VPN ait une latence légèrement supérieure car il doit lire, encapsuler, décapsuler, écrire. Toutefois, les performances sont quand même très satisfaisantes. Le ssh et le telnet sont très fluides lors de l'utilisation en particulier.

2.4 Propositions d'extensions pour QUIC

QUIC est un protocole qui a été conçu pour accélérer le traffic web. Il est donc normal que celui-ci ne force pas l'authentification du client : tout le monde à priori à le droit de se connecter à google.com, on n'a pas besoin d'un certificat x509. Par contre le serveur, lui doit être certifié car on veut être sûr de bien être connecté au vrai google.com. Cependant, pour les VPN, le contexte est différent : il faut authentifier le client car après la connexion VPN établie, le client peut accéder aux machines de l'entreprise via le tunnel. Il est donc tout à fait essentiel d'authentifier le client. La question maintenant est : comment authentifier le client ? Et surtout, comment l'implémenter à QUIC ?

2.4.1 Authentification par clé publique

Pour cela, nous nous inspirons de l'authentification SSH, qui fonctionne de la manière suivante :

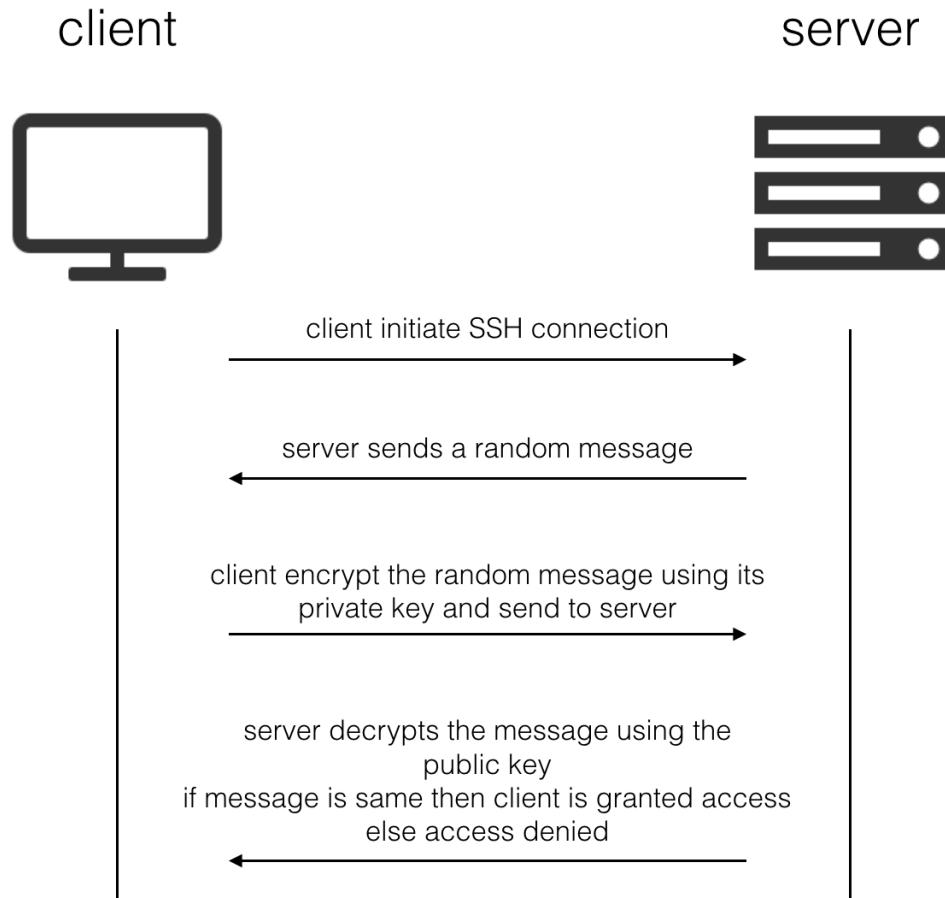


FIGURE 2.28 – handshake SSH

Dans le cas de QUIC, on pourrait réaliser une authentification de la manière suivante :

Le serveur génère un message pseudo aléatoire challenge et l'envoie au client. Le client génère un message pseudo aléatoire ‘nonce’, et envoie au serveur : $PBc, \text{nonce}, S_{pkc}(challenge||nonce)$
 PBc : clé publique du client, S_{pkc} : signer par la clé privée du client pkc.

Le serveur peut alors vérifier la signature avec la clé publique. Si cette clé publique est acceptée par le serveur, celui-ci autorise la connexion. Sinon il la refuse. Ceci peut être très bien intégré au full handshake QUIC. Lors du serveur hello, celui-ci donne le challenge au client. Lors de la réponse du client, le full client hello, celui-ci indique le message $PBc, \text{nonce}, S_{pkc}(challenge||nonce)$ pour s'authentifier.

2.4.2 Authentification avec STK

Lors du premier handshake, le serveur attribue au client un STK permettant par la suite de s'assurer que son détenteur possède bien l'adresse IP qu'il prétend avoir. En effet, ce STK contient l'ip du client ainsi qu'un timestamp issu du serveur crypté par AES-GCM. Le client devra par la suite continuer à envoyer ce STK lors des client hello. Cependant, ce STK est vulnérable car il peut être intercepté lorsque le serveur l'attribue au client. Un attaquant peut alors le rejouer. Pour que ce STK serve finalement de cookie, il faut le protéger d'attaques type replay. Pour cela, le client peut ajouter un timestamp client au STK pour que le serveur détecte si la différence de temps entre ce timestamp lors de l'émission et l'heure d'arrivée est suffisamment faible ou non : si il est trop importante, il s'agit peut-être d'un replay. Ces modifications faites, le serveur peut alors utiliser le STK comme authentification du client.

Pour empêcher le rejeu, nous proposons une autre solution que le timestamp :

- Le client génère une paire de clés PUBC/PKC
- Le client génère un nonce pseudo-aléatoire
- Le client envoie au serveur : STK, PUBC,nonce, $S_{PKC}(STK,nonce)$

Côté serveur :

- Le serveur vérifie la signature
- Le serveur vérifie si le nonce n'a pas déjà été utilisé
- Le serveur vérifie le STK pour vérifier que l'adresse IP source convient bien avec le STK
 - Le serveur authentifie alors le client par le STK

La signature permet l'intégrité, le nonce le non-rejeu, le STK l'identification. Le tout forme l'authentification. Ce système pourrait être inclus dans le full client hello de QUIC.

2.4.3 Authentification par x509

Bien que QUIC soit orienté web, notre conception d'un programme VPN a montré qu'il serait essentiel d'avoir une possibilité d'authentifier un client via un certificat x509. En effet, cela serait tout à fait pertinent dans une connexion VPN type site to site : les 2 serveurs devraient s'authentifier mutuellement via des certificats. On pourrait alors ajouter dans le premier client hello un champ optionnel permettant d'intégrer le haché du certificat client au format FNV-1a. Le serveur peut alors, à la réception, vérifier le certificat client et l'identifier ainsi.