Cybersecurity option

Study and analysis of QUIC and study of an extension for VPNs

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

Professor :
M. Ahmed SERHROUCHNI

Contents

Introduction and analysis of the QUIC protocol

1.1 Motivations and general operation of the protocol

1.1.1 History and general ideas

QUIC (Quick UDP Internet Connections) is an experimental connection protocol developed by Google in 2013. It is estimated that on the Chrome search engine, more than half of the connections to Google servers use the QUIC protocol. The whole point of this protocol is that it drastically improves the performance of connection-oriented applications that therefore use the TCP protocol at the transport level of the OSI model.

Let us quickly present a brief history of what gave birth to the QUIC protocol.

HTTP, the main application-level protocol, was born in 1991 as HTTP/0.9, which became HTTP/1.1 in 1999 after standardization by the IETF (Internet Engineering Task Force). After a while, HTTP/1.1 was no longer able to meet the evolving needs of the Web and so HTTP/2 was born in 2015.

The HTTP protocol continued to be tested and improved on the Internet. Google in particular had started testing with a software called SPDY (pronounced "speedy"). This protocol was supposed to improve web browsing, which is the main purpose of HTTP. In late 2009, v1 of SPDY was announced, and it was quickly followed by v2 in 2010.

Overall, SPDY took the basic principles of HTTP and slightly modified the exchange format to make the necessary improvements. Eventually, SPDY was adopted in 2012 and became HTTP/2.0 in 2015, after various redesigns.

During this time, Google developed another project called gQUIC in parallel to SPDY. The SPDY binary syntax was compiled into gQUIC packets that could be sent as UDP datagrams. This was a historic departure from the TCP transport on which HTTP traditionally relied.

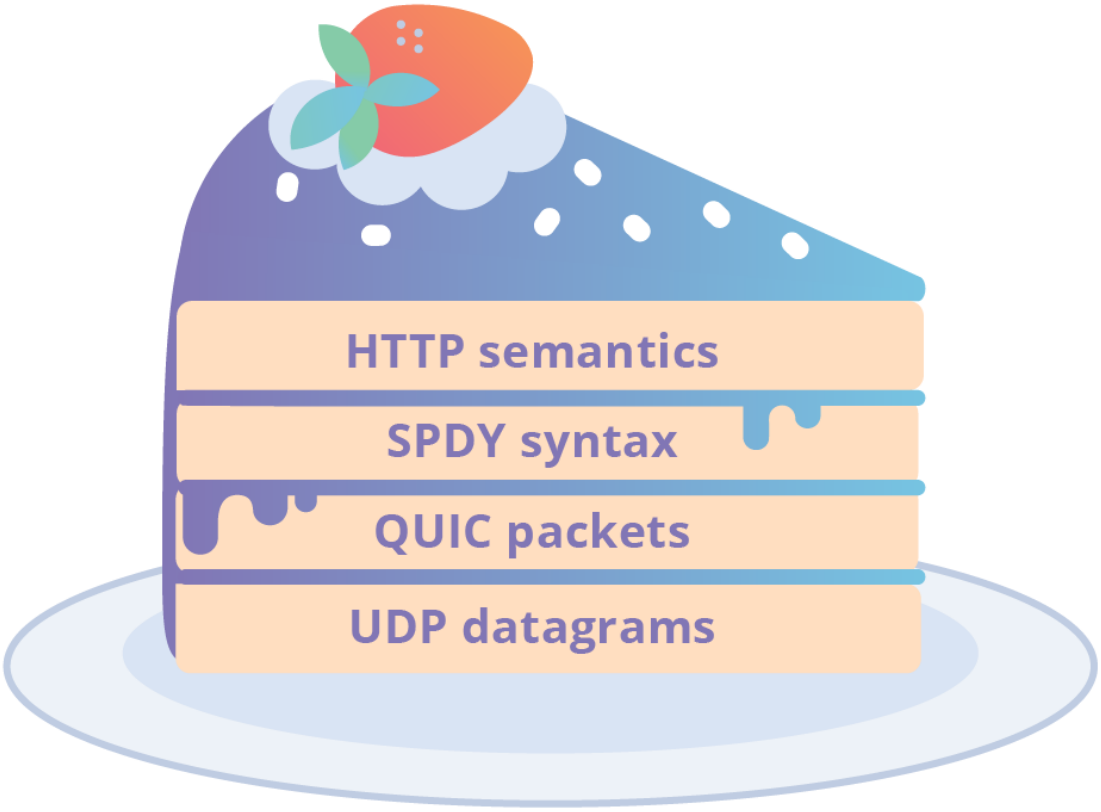


Figure 1.1: gQUIC Architecture

Google had developed a new method called QUIC Crypto, allowing for faster session-level security handshakes. (It was later superseded by TLS 1.3, which did not yet exist initially.) With QUIC, the connection and the encryption are indeed done in a single handshake, which saves a lot of time.

QUIC was able to solve the "HoL" problem: the "Head of line blocking". As HTTP/2 was based on the TCP protocol, which must guarantee the good order of arrival of the packets to their recipient, if a packet was lost, all the TCP connection had to be stopped while waiting for it to be retransmitted, and this blocked the various data flows on this connection. QUIC had the advantage of not using TCP and could make intelligent decisions about which packets to stop and which to let through in such a case, which greatly alleviated the traffic congestion problem.

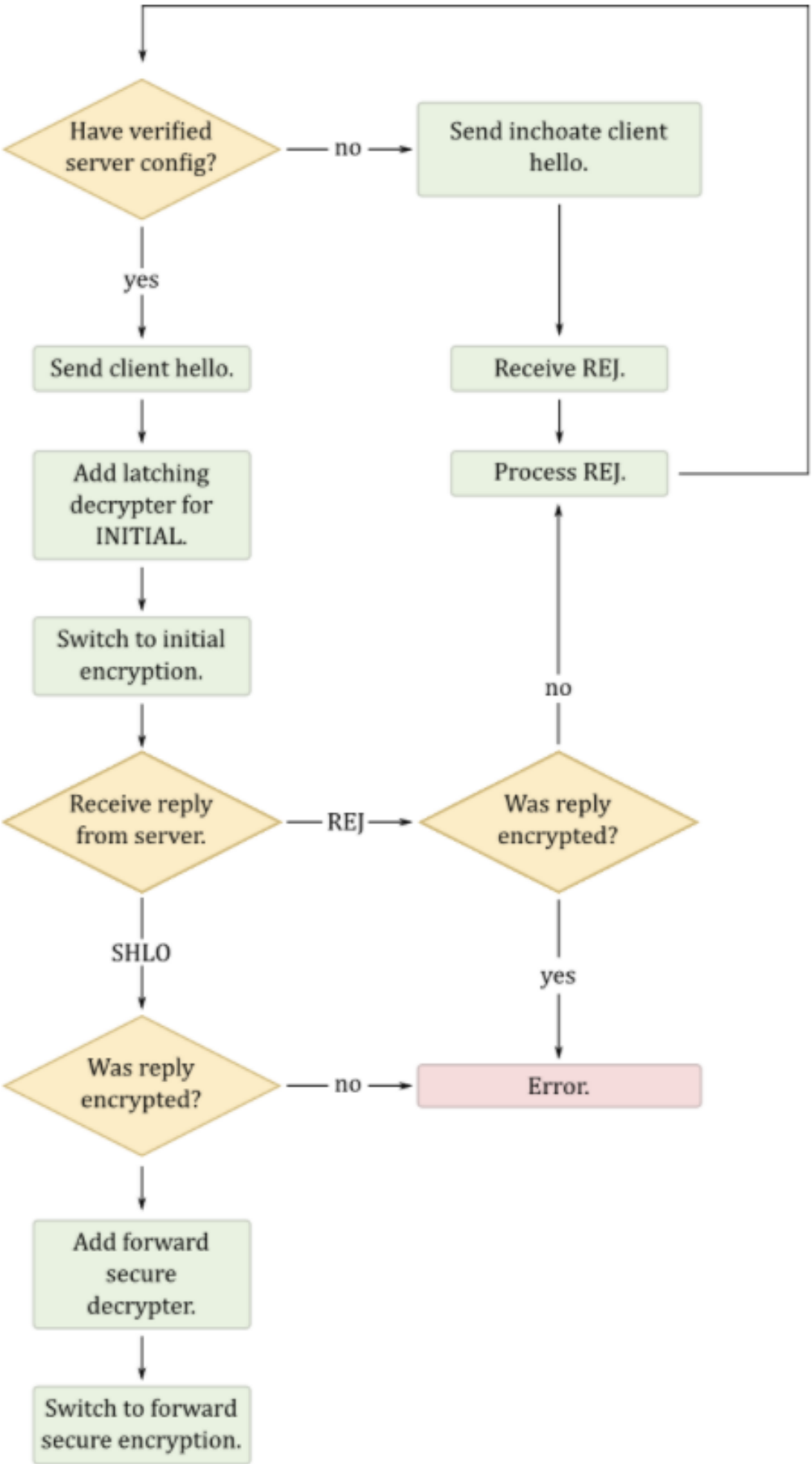
Another major idea enabled by QUIC is the "0-RTT" idea. The idea behind this is that when a client resumes a QUIC connection with a server with which it has already

established a TLS connection before, it does not need to establish a 3-way handshake again: information about the previous session is kept in memory, notably the encryption key used during the communication. In fact, the client's message can be directly encrypted with its private key while being directly understood by the server.

In November 2018, the IETF met in Bangkok and a new Internet project was adopted. The QUIC transport protocol, the successor to HTTP/2, was renamed to HTTP/3.

1.1.2 Detailed operation

Here is a diagram of what happens when a client initiates a handshake connection.


Have verified server config?
no
Send inchoate client hello.
yes
Send client hello.
Receive REJ.
Add latching decrypter for INITIAL.
Process REJ.
Switch to initial encryption.
no
Receive reply from server.
REJ
Was reply encrypted?
SHLO
yes
Was reply encrypted?
no
Error.
Add forward secure decrypter.
Switch to forward secure encryption.

At first, the client doesn't know anything about the server with which it will communicate. Before initiating the handshake, it sends an incomplete "client hello" request, which is in fact a "client hello" request with an empty payload but containing the DNS address of the server, a request for proof of certificate, and the version of the QUIC protocol supported by the client.

Following this request, the client receives a REJ (Reject) message from the server, which contains a proof of authenticity as well as its latest configuration. This configuration contains the server's public encryption key and the encryption algorithms it supports. The proof of authentication of the server (within the framework of the X.509 standard) is made up of the signature of the server's configuration using the client's public key, which the client can decode with its private key.

At this stage, the client has the knowledge of the server's configuration, and can therefore send it a complete "client hello" request, with a real payload. Note that this exchange of "client hello inchoate" and REJ messages can last a while, because the server does not necessarily want to reveal all its information to an unknown client.

Once the client sends real "client hello" requests, the server can either send an REJ message or accept the request and send back an encrypted "hello server" (SHLO on the diagram) containing one-time use keys, called "ephemeral". In the latter case, this means that the handshake was successful. Otherwise, in the case of an REJ response, the server gives the client information allowing it to perform a better handshake. For example, if the client has never connected to the server, the server must give the client a "source-address token", which is a kind of multiple-use passkey containing the client's IP address and a time stamp from the server. This token guarantees both the integrity and the confidentiality of the data. The client is then in possession of encryption keys called "initial keys", and it is with these that he will have to encrypt future packets.

Below, some of the main tags used by QUIC, according to the official documentation :

During the "hello inchoate client":

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.

When receiving the Rejection message (REJ flag) :

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

In the server configuration :

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.)

Finally, the tags in the "full client hello" :

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 Header Analysis

1.2.1 Introduction

A QUIC packet is the content of datagrams exchanged between 2 agents, called "QUIC Endpoints".

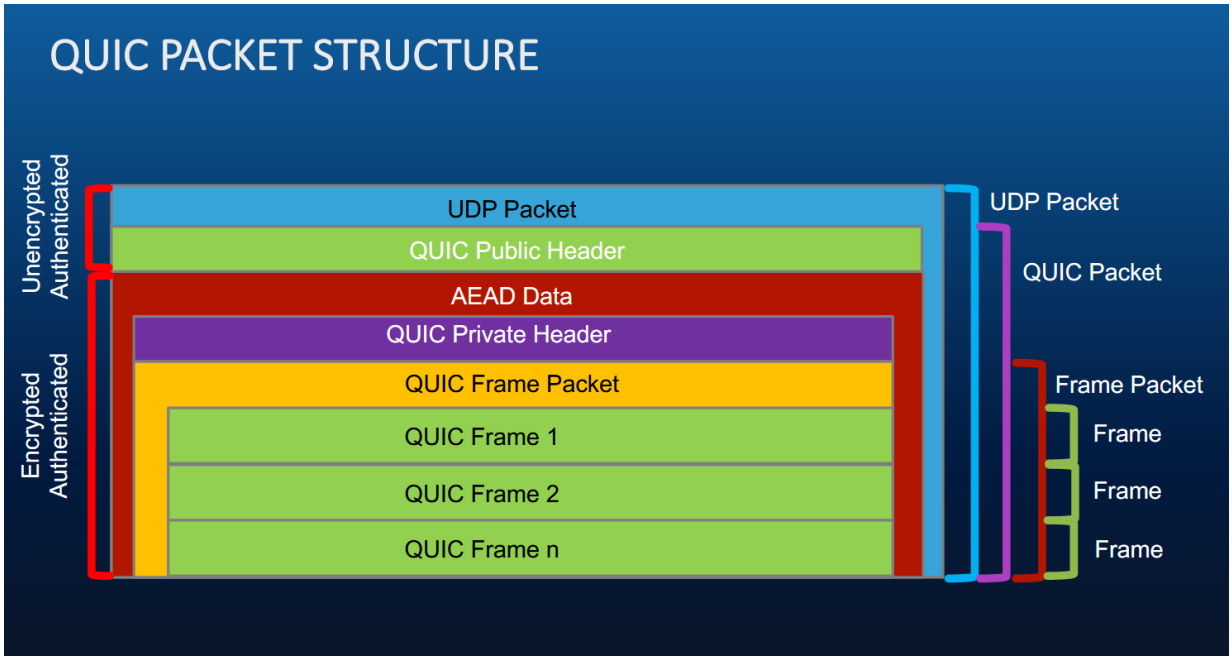


Figure 1.3: Structure of a QUIC package

The QUIC protocol defines two types of headers: long and short. The distinction between these two types of headers is found on the one hand in the differences in header size, but also the most significant bit which differs according to these types (0: short header, 1: long header). The data and lengths of QUIC packets depend on the version of QUIC used.

1.2.2 Long Header

Here is the representation of a QUIC "Long Header":

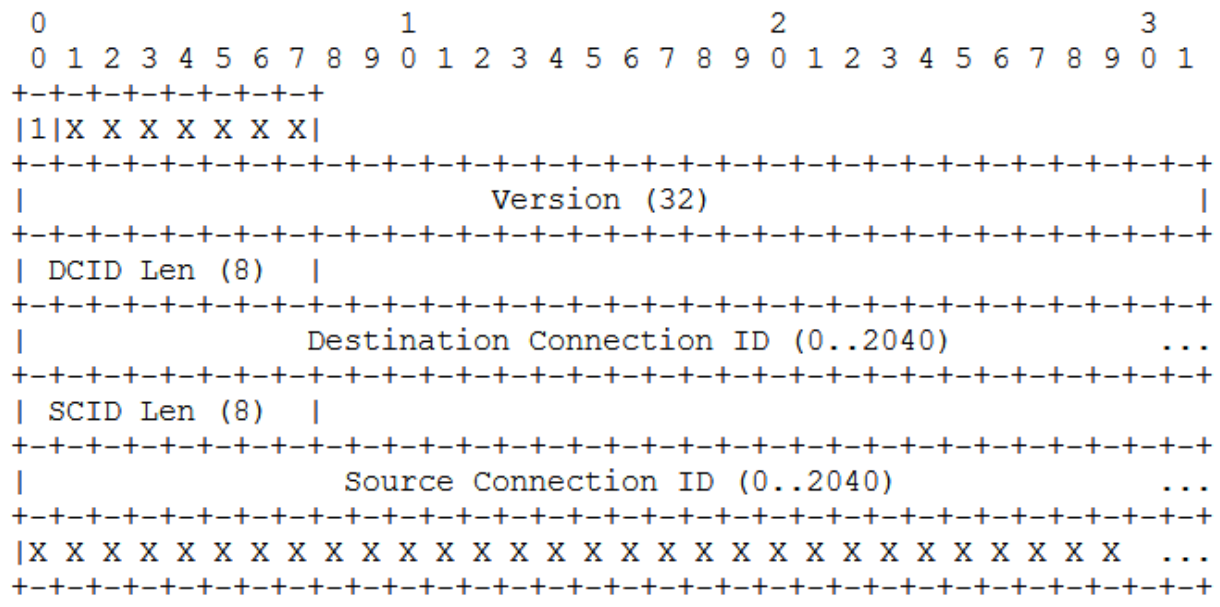


Figure 1.4: Datagram Long Header

As shown in this figure, a QUIC packet with a "long header" will have the most significant bit of the first byte set to 1. This header consists of The "Version" field, indicating the QUIC version, on 4 bytes; The DCID Length field, indicating on 1 byte the length in bytes of the DCID field; The DCID field (Destination Connection ID), whose length varies between 0 and 255 bytes; the SCID Length field, indicating on 1 byte the length of the SCID field; The SCID field (Source Connection ID), whose length also varies between 0 and 255 bytes. During the handshake, the long header is used to establish connection IDs in each direction. Each endpoint uses the SCID field to specify the DCID used in the packet addressed to it. Each endpoint uses the SCID field to specify the DCID used in the packet addressed to it. When a packet is received, each endpoint sets the DCID to match the SCID received.

1.2.3 Short Header

Here is the representation of a "Short Header":

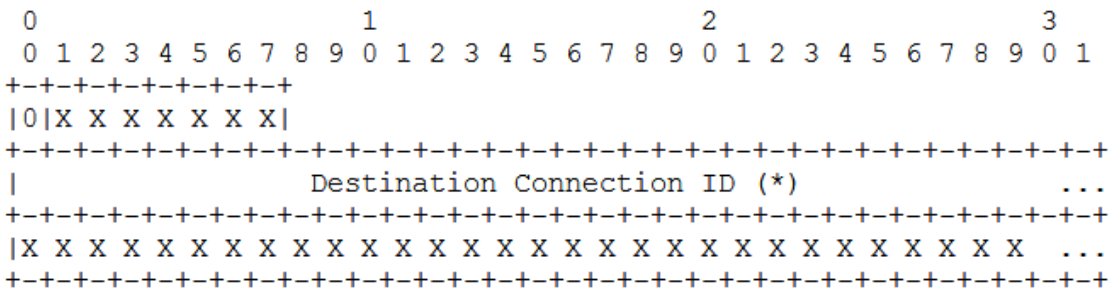


Figure 1.5: Datagram Short Header

As this figure shows, a "short header" is recognized with the most significant bit of the first byte set to 0. This header is similar to the long header, but shorter and simplified, in which only a DCID (Destination Connection ID) field is present. This type of header does not include the other fields existing in a "long header" (Version, DCID Length, SCID Length, SCID).

1.2.4 Connection ID (SCID, DCID)

Each connection between two endpoints has a set of connection identifiers, called "Connection ID", each of which can identify a connection. These are independently selected by the endpoints, which must select the one the pair should use.

As mentioned above, this field is of variable length, the primary function of this field is to ensure that addressing changes in lower layer protocols (UDP, IP,...) do not return QUIC packets to the wrong destination. A "Connection ID" is used by the endpoints (source and destination of the packet), as well as the intermediaries ensuring the correct routing of the packet. A connection ID must not contain information that can be used by an external observer to correlate with other connection IDs. A Length Header contains a Source Connection ID (SCID) field and a Destination Connection ID (DCID) field. These fields are used to set up a Connection ID for new connections. The DCID is chosen by the receiver of the packet to provide consistent routing, while the SCID is used to set the DCID used by the pair.

1.2.5 Version negotiation

QUIC versions are identified with a 32-bit ID. 0 is reserved for version negotiation. An endpoint receiving a packet with a long header with a version that is not understood or supported may send a version negotiation packet in response, which will not be the case

for packets with a short header.

Here is a datagram of a version negotiation packet:

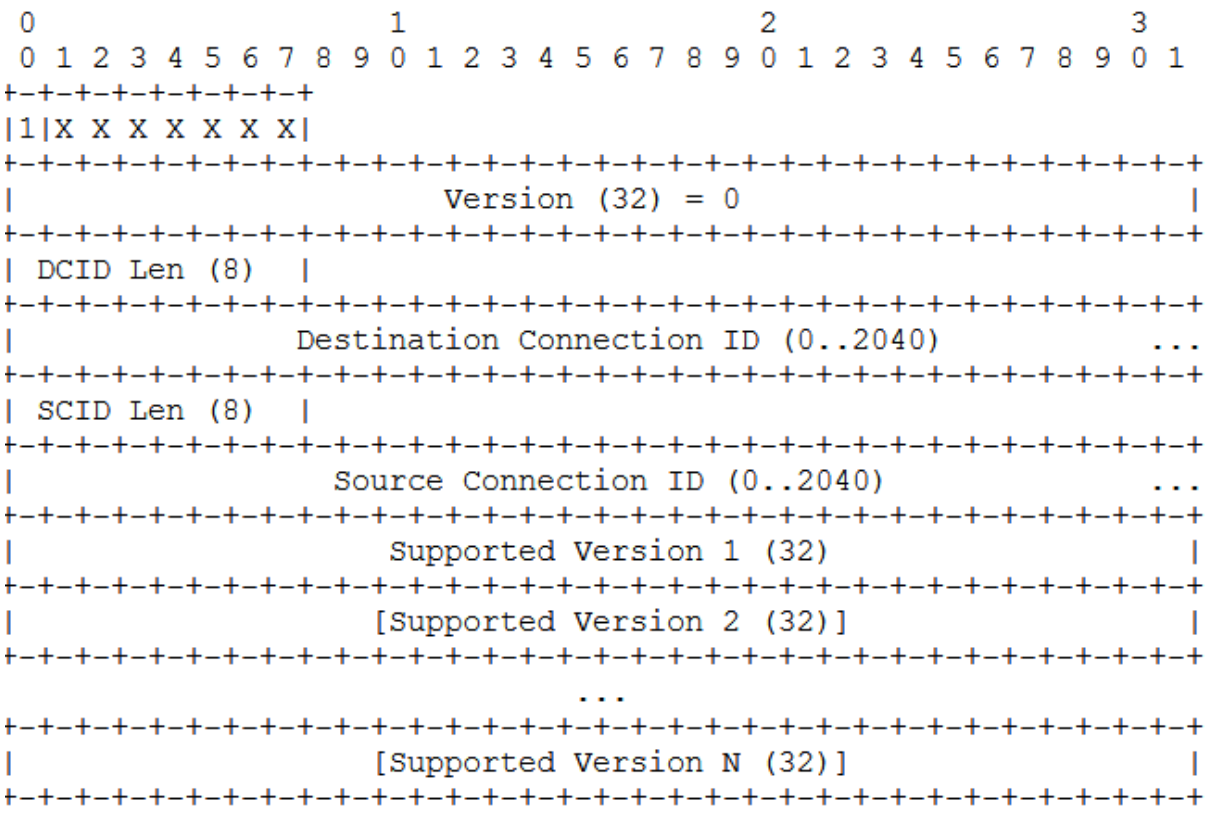


Figure 1.6: Version negotiation diagram

A version negotiation packet contains a list of supported versions, each identifying a version that an issuer supports. This list of supported versions follows the "Version" field. A version negotiation package will not contain any other fields. If a packet does not contain a supported version field, or a truncated version field, an endpoint should ignore it. No integrity or confidentiality protection is provided on version negotiation packets. A specific version of QUIC could authenticate the packet as part of its connection establishment process. An endpoint must include the value from the SCID field of the received packet in the DCID field. The SCID value must be copied from the DCID of the initially randomly selected received packet. Listening to the 2 Connections gives the assurance that the server has received the packet on the one hand, and that the version was not generated by an attacker on the other. An endpoint receiving a version negotiation packet could change the version for the subpackets. The conditions under which an endpoint will change its QUIC

version will depend on the version chosen.

1.3 Presentation of different scenarios - QUIC traffic analysis

1.3.1 Implementing a local QUIC client and server with ngtcp2

1.3.1.1 Local QUIC server setup

ngtcp2 is a draft implementation of the QUIC protocol implemented in C. This implementation contains a server program whose syntax is the following:

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

ADDR contains the server address. In this case it will be localhost. PORT is the port on which the server will be listening. Let's put 4433 to avoid a possible redundancy with port 443 which is already taken by Google. PRIVATE_KEY_FILE is the private key of the server in pem format, for example an rsa key. CERTIFICATE_FILE is the server certificate in x509 format. Be careful however, the commonName of the certificate must be 'localhost', otherwise the certificate does not correspond to the server address and will be rejected.

A multitude of options for the server are available, including:

–ciphers=<CIPHERS> : To specify the list of encryption algorithms you want to use

–groups=<GROUPS> : To specify the different groups supported (Diffie-Hellman groups)

–verify-client: To require a client certificate

Then we have various useful parameters for flow management, including:

–max-data=<SIZE> : The initial size of the flow control window

We can now launch the QUIC server locally with the command : examples/server localhost 4433 ssl_key.pem ssl_cert.pem

SSL_CERT being a leaf certificate having for identity 'localhost' delivered by the root authority 'our-ca-server'.

1.3.1.2 Implementing the QUIC client locally

Still with ngtcp2, we use the client program with the syntax:

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

ADDR being the destination address, localhost in our case. PORT the port number of the server, 4433 in our case URI the resource we want to retrieve from the server.

Among the options we count:

-d, –data=<PATH>: read the data in PATH and send it to the server as STREAM

-v, –version=<HEX> : the QUIC protocol version to use in hexadecimal format.

–ciphers=<CIPHERS>, –groups=<GROUPS> : the encryption algorithms and groups supported by the client.

–session-file=<PATH> : To read a TLS session ticket and resume the session.

–key-update=<DURATION> : To update the keys after a time defined here

–key=<PATH>, –cert=<PATH> : The private key and certificate of the client in PEM format.

–qlog-file=<PATH> : To retrieve the connection logs in QLOG format which can then be viewed using QVIZ.

Finally, we have transport parameters such as the initial size of the flow management window, as with the server.

We can then launch the client with the command :

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

1.3.2 Setting up scenarios

1.3.2.1 Connection to an unknown server, study of QUIC traffic with the chrome://net-export/ tool

To analyze a connection to an unknown server, we can simply use the log capture tool integrated to the google-chrome browser. To do this, we go to the chrome://net-export/ page. We choose to remove personal information from log captures, including cookies, passwords, etc... After launching the capture, we go to youtube.com. (On the kali virtual machine used, it was the first connection to the youtube server). Once the page is loaded, we can close it and stop the capture. Once the capture is saved in JSON format, we want to analyze it to observe the QUIC traffic. To do this, we use the chrome extension: Chromium NetLog dump viewer. We can upload the logs and click on the QUIC tab to see a menu showing the properties of the QUIC connections that have been opened. We can then click on "view all QUIC sessions", then choose the domain name youtube.com to display the QUIC traffic to youtube.

First we have a CHLO:

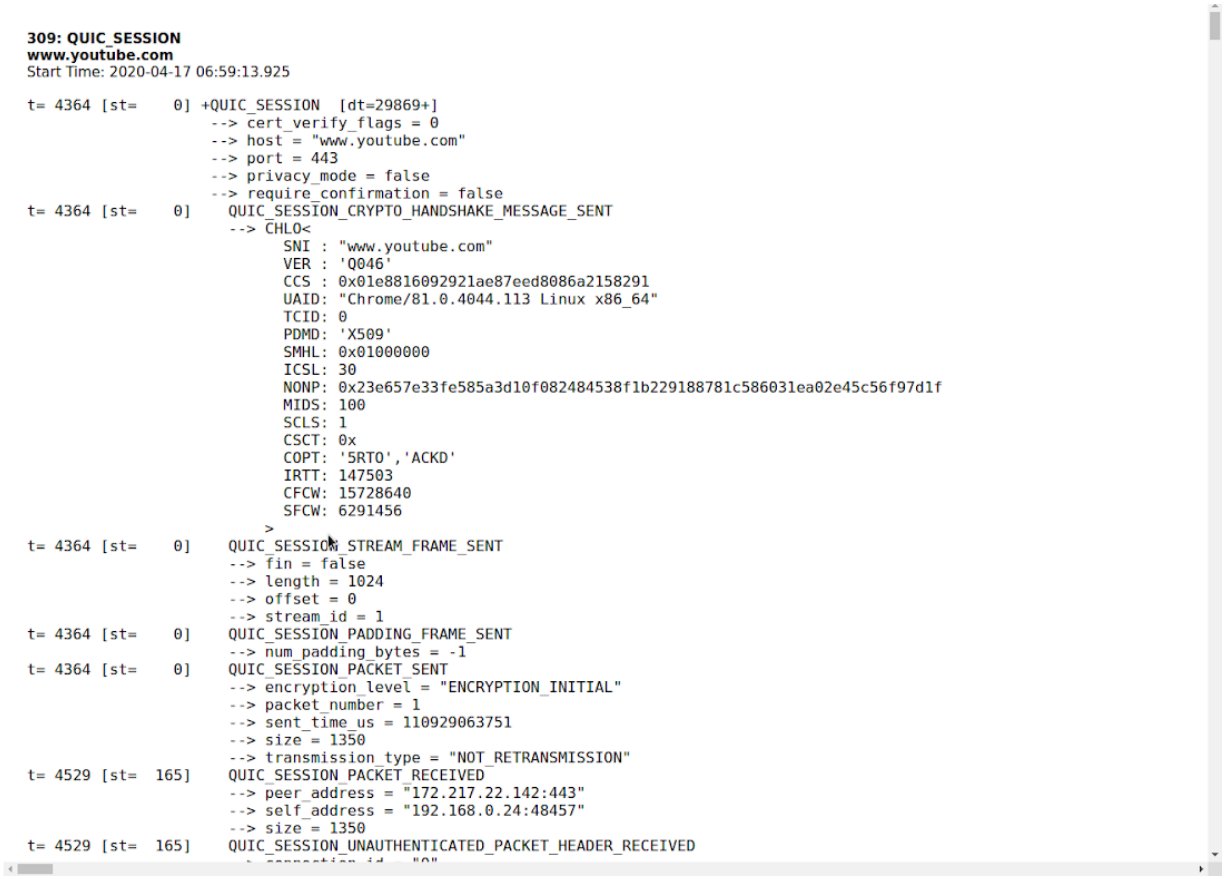


Figure 1.7: CHLO

Let's take a look at the CHLO:

- SNI: the DNS name of the server 'www.youtube.com'
- STK: (Source-address token) absent, because server unknown for the client
- VER: the QUIC protocol version used 'Q046'.
- CCS: Common certificate sets, a 64-bit series containing hashes in FNV-1a format of certificate sets that the client owns
- The CCRT (Cached certificates) flag is absent because the client has no certificates linked to this new server.
- PDMD: Proof Demand, types of proofs acceptable to the client, here 'x509'.

To this we add padding. We first receive a QUIC_SESSION_PACKET indicating that the version has been negotiated at 'Q046'.

```
                          ICSL: 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"
                      --> 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 = "Q046"
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
t= 4544 [st=  180]    QUIC_SESSION_PACKET_RECEIVED
```

Figure 1.8: QUIC SESSION PACKET

We also receive the REJ from the server:

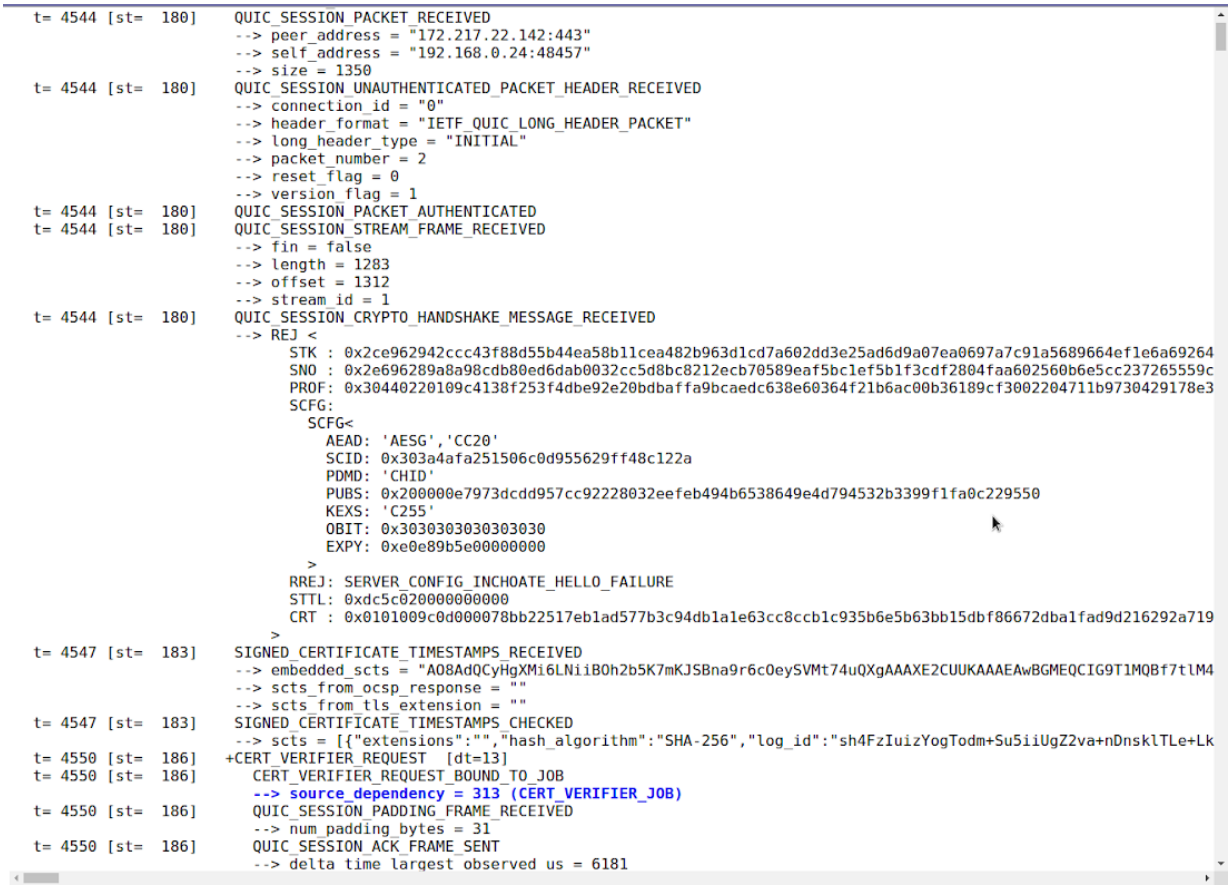


Figure 1.9: QUIC SESSION PACKET

Let's analyze some of the flags of the REJ:

- STK : Source address token, a string of bytes that the client will have to display during future client hello messages.
- SNO: Server nonce, that the client will have to show in each complete hello client. This is to prevent replay attacks
- PROF: Proof of authenticity, contains the signature of the server configuration by the public key of the leaf certificate. The format of the signature depends on the format of the public key.
- SCFG: Server config: server configuration, containing:
  - AEAD: Authentication encryption algorithms, with here AESG in preference

  - SCID: Server config ID
  - KEXS: key exchange algorithm, here C255 (Curve 25519)
  - PUBS: list of public values of the server
  - EXPY: expiration date of the server configuration
- STTL: Validity time of the server configuration in seconds

We can see that the server also sends its x509 certificate, which the client must verify afterwards.

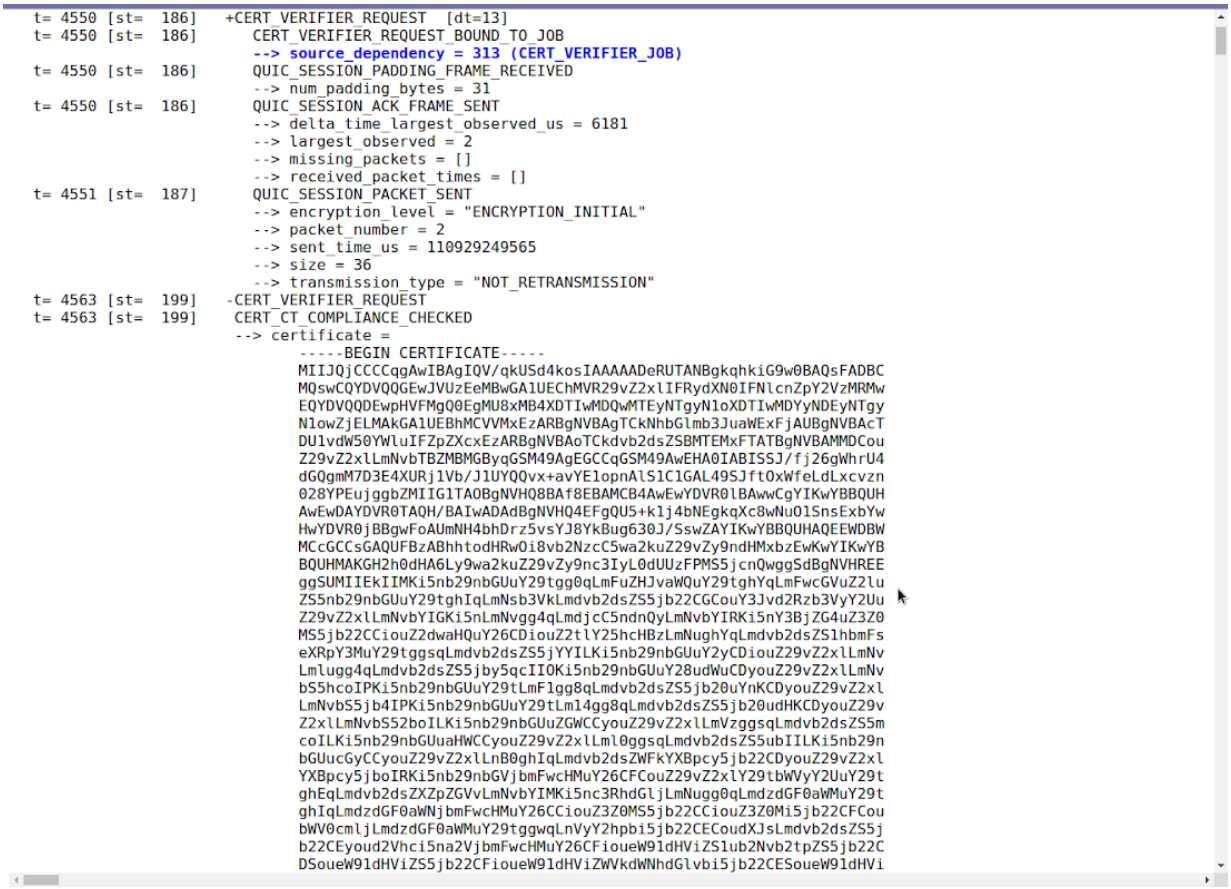


Figure 1.10: certificate x509

Once this is done, the client can then send a complete hello client, unlike the initial hello client which was incomplete because the server was unknown. Now the hello client sent is of the form:

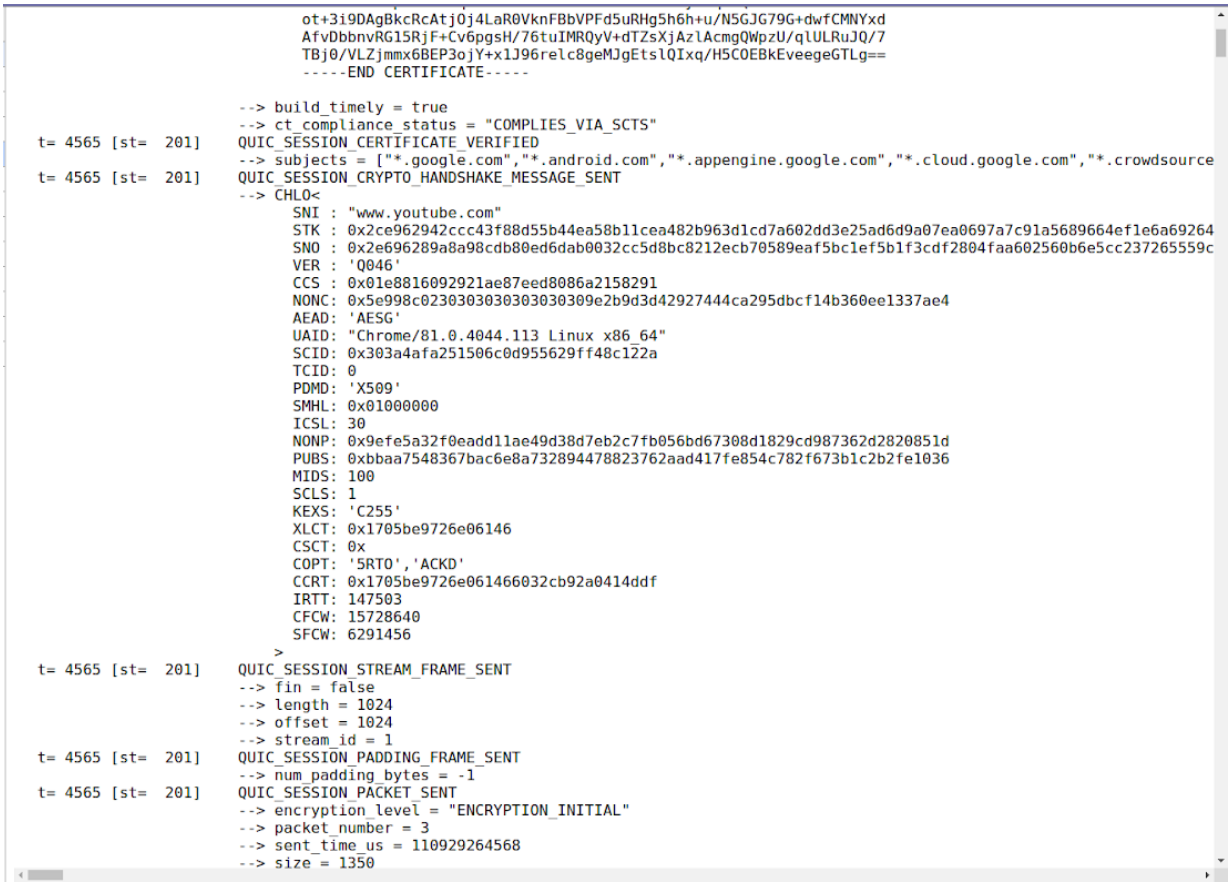


Figure 1.11: Client Hello complete

In addition to the basic fields of the original CHLO, there are additional fields, including:

- SCID (Server config ID)
- AEAD (the algorithm chosen to encrypt)
- KEXS (the chosen key exchange algorithm)
- NONC (client-announcement encoded on 32bytes made of 4 bytes of timestamp, 8 of server orbit and 20 of pseudo random data)
- SNO (Server nonce, if the server has given one, we can see that it is indeed the one which appears higher in the REJ)
- PUBS (Client public value for the key exchange algorithm)

This being, done, the client can then communicate to the server by sending it application data:

```
                        --> 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,applicati
                            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: Application data

1.3.2.2 Connection to an already known server

Let's study the case where the client wishes to reconnect to a server that it already knows (PUBS of the server, certificates, SCID, ...). The first step of the client is to check the server's cached certificate to make sure it is still valid:

Figure 1.13: Cache server check

Once the certificate is verified (certification chain, signatures, revocation,...), the client then sends the full client hello directly:

```
                        --> 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
                        --> CHLO<
                              SNI : "www.google.com"
                              STK : 0x658e11223665e8dc98d86305f33fc0b564f21cf67b410223a5051e34afc7cbdf4d812f60a5d7fc96e0e095d2f2aa43c7ab1b967ab929
                              VER : 'Q046'
                              CCS : 0x01e8816092921ae87eed8086a2158291
                              NONC: 0x5e9a21ef30303030303030302808a6c7733f4da0e62d1e208ca18b262e5be60e
                              AEAD: 'AESG'
                              UAID: "Chrome/80.0.3987.163 Windows NT 10.0; Win64; x64"
                              SCID: 0x303a4afa251506c0d955629ff48c122a
                              TCID: 0
                              PDMD: 'X509'
                              SMHL: 0x01000000
                              ICSL: 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
                              IRTT: 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/ddllog?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.
```

Figure 1.14: Full client hello

We then recognize the different parameters of the full client hello exposed above in the case of the connection to an unknown server. Once this is done, the client does not wait for a response from the server! It sends directly its https application packets whose content is encrypted:

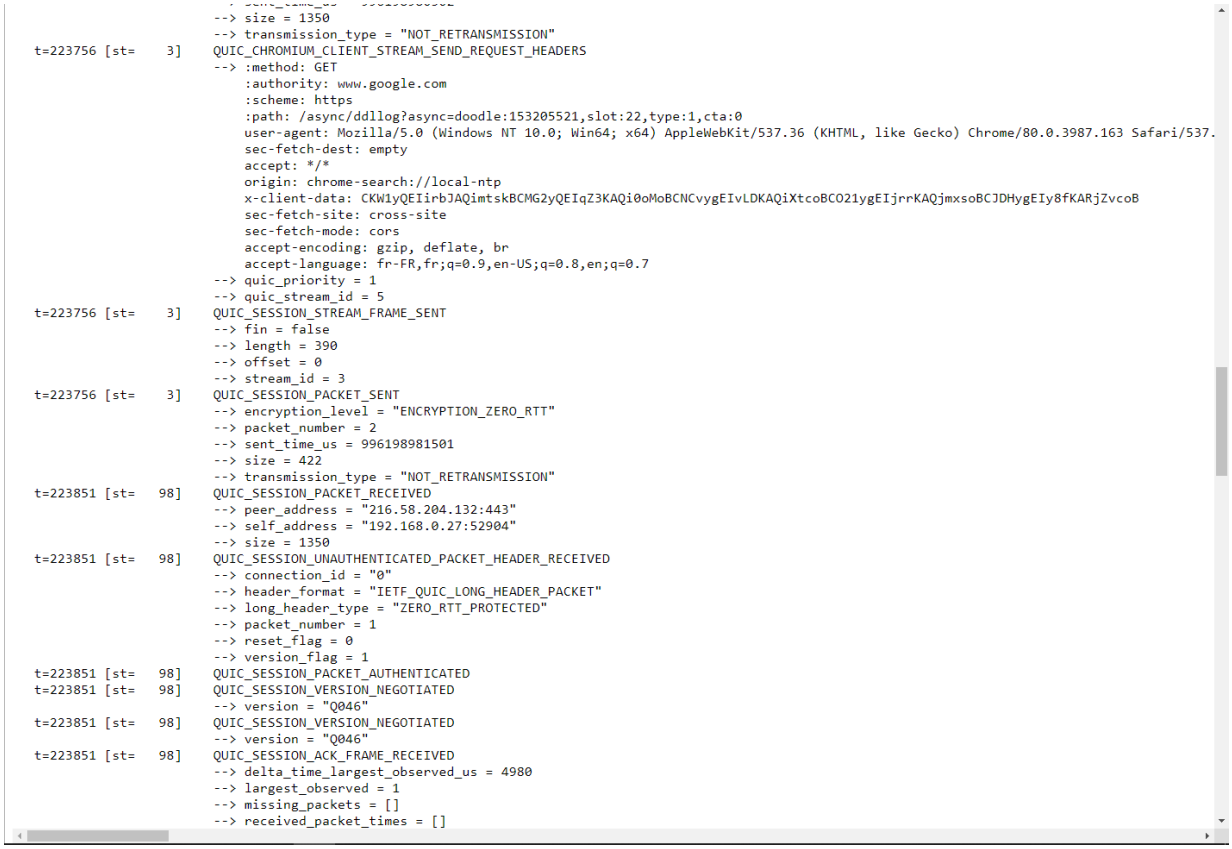


Figure 1.15: Encrypted application packages

The client and server are already talking in encrypted http. It was just necessary to: recheck the certificate and send the full client hello. This is a huge time saving compared to the classic TCP+TLS.

1.3.3 Packet loss scenario

QUIC is a UDP-based protocol. However, UDP does not have mechanisms for recovering lost packets. It is then the role of QUIC to add these necessary services. The scenario is as follows: we try to connect to a server that does not support the QUIC protocol. The client will try to connect, but the server will not respond because it does not know how to handle QUIC requests. Let's set up our client using ngtcp2. We type :

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

Then we notice on the console that the client tries to return the initial CRYPTO/-PADDING request several times. We manually stop the program and upload our log file

on qviz to have a diagram of the situation:

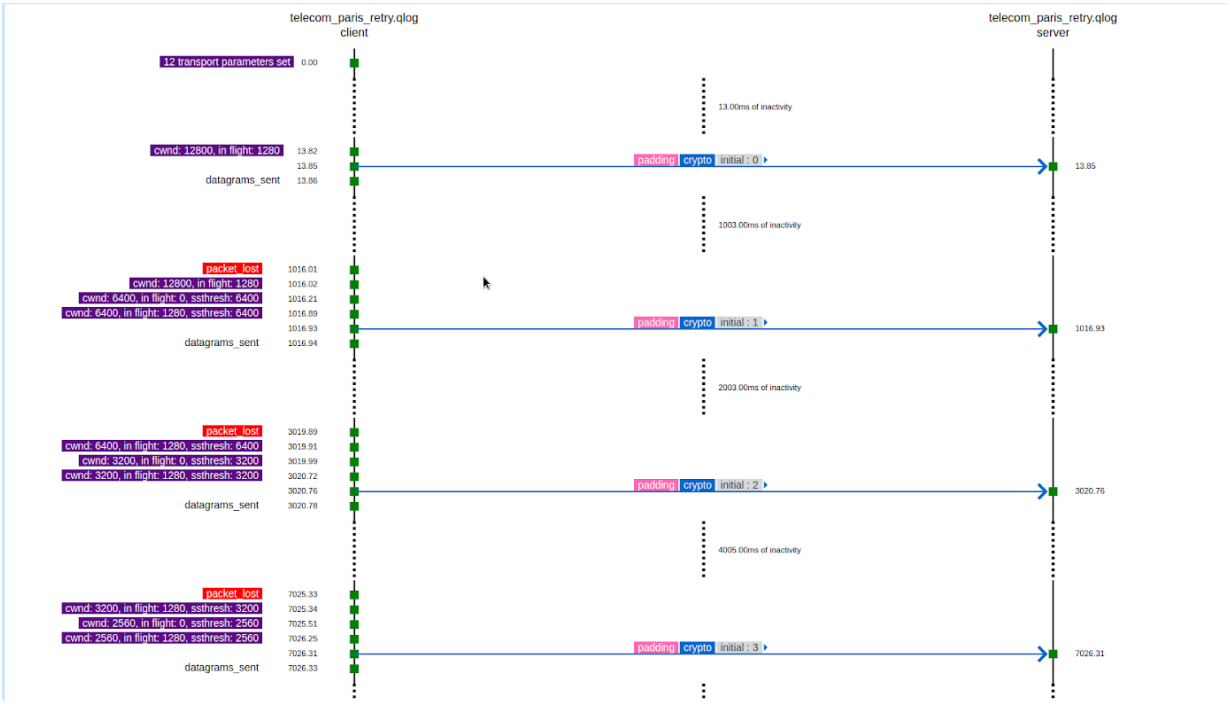


Figure 1.16: Situational Diagram

We can then see that it tries to send the request, it waits one second, no answer, it sends again. It waits this time 2 seconds, still nothing. Then it waits 4 seconds, etc... Each time, it doubles the timeout to hope to have a response from the server. We can also read on the left in purple that the size of the window decreases: maybe the packet did not arrive because the size of the window is too big. So it decreases gradually. This scenario is important because it shows how QUIC allows to manage layer 4 parameters while we were mainly concerned with layer 5 features.

1.3.3.1 Version negotiation failure

After sending the first CHLO to the server, the latter replies with the chosen version, but only if the server has a version corresponding to that of the client. Still with ngtcp2, let's try to contact youtube.com. We execute : $ examples/client –qlog-file logs.qlog youtube.com 443

The program stops with an error ERR_RECV_VERSION_NEGOCIATION. Indeed, the QUIC version of ngtcp2 is 'build-draft27' and not 'Q046'. But what happened to the

exchanges? Let's see it with Wireshark:

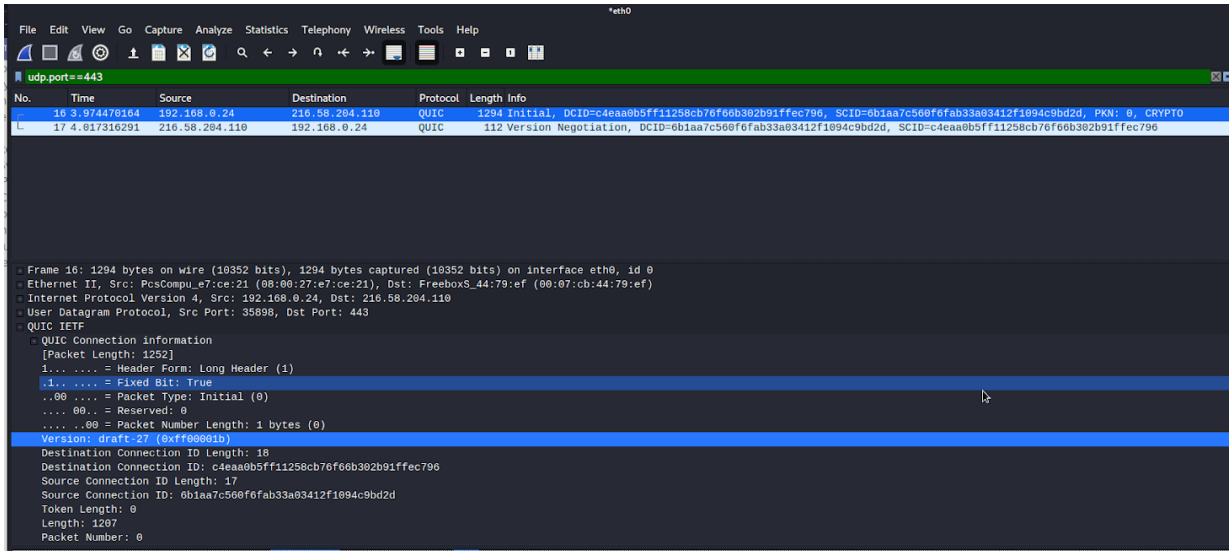


Figure 1.17: Capture wireshark - version

Our ngtcp2 client sends the CHLO with the draft-27 version. It then receives a Version Negociation packet containing the different versions supported by the server:

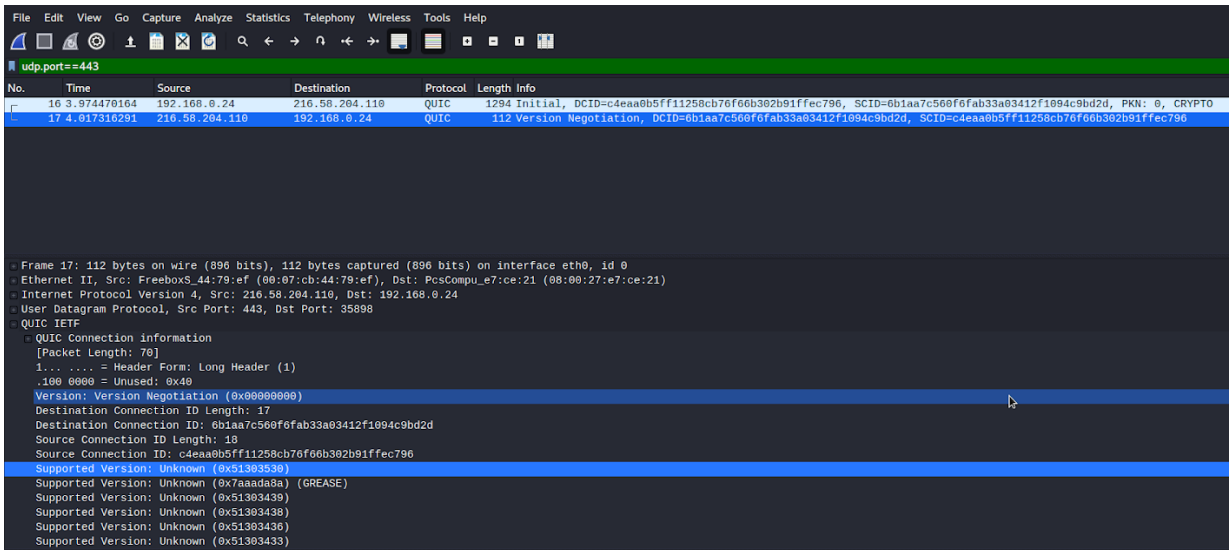


Figure 1.18: Capture wireshark - negotiation version

None of them correspond to the build-draft27 version. The client then stops with the error ERR_RECV_VERSION_NEGOCIATION. You can manually change the Version field sent to the server with the -v option. However, this is of no use: the server reads the version, expects Q046, but reads build-draft27 which it does not recognize. It throws the package away and does not even respond. That's why on the client side, the package loss mechanism is activated.

1.4 Performance and quality of service comparison with TLS and TCP

1.4.1 Similarities

QUIC and TLS+TCP are often compared, as are the application protocols that implement them, HTTP/3 and HTTP/2 respectively. Despite all the differences in implementation and speed that separate them, there are some structural similarities.

For one thing, the headers in HTTP/2 and HTTP/3 have very similar contents. The headers, QPACK for QUIC and HPACK for HTTP/2 are very similar in design, and are both the result of IP packet header compression mechanisms.

Another similarity at the communication level is that both protocols offer "server push" communication, i.e. communication initiated by the server, after prior authorization from the client. This is a fundamental mode of communication, also used in instant messaging: as soon as a member of a conversation sends a message, the central server sends it to all the other members of the conversation, who have therefore chosen to participate.

The last notable similarity is in the area of flow management: both protocols offer flow management, characterized in particular by the multiplexing of several sessions and associated flows on a single connection. As soon as two machines communicate, a session is created and processed on the same link as all other sessions. However, there is priority management for the flows of different sessions, and this is taken into account by both HTTP/2 and HTTP/3.

1.4.2 Differences

If QUIC has become popular, it is because it offers a number of improvements over its "rival" TCP + TLS. In particular, QUIC allows HTTP traffic to flow more smoothly, while making it secure.

One of the main deviations of QUIC from TCP is that it includes encryption in addition to the transport level and thus secure data transport, which TCP alone cannot manage

since it uses TLS, located higher in the OSI model since it makes the connection between the transport layer and the application layer to secure the data.

In addition to that, at the service level, the main difference is that QUIC is based on the UDP transport protocol, which is not connection-oriented, unlike TCP, and therefore allows a more efficient and less rigid communication than TCP.

Here are two protocol diagrams highlighting the difference between a handshake managed by QUIC and a classic handshake with TCP:

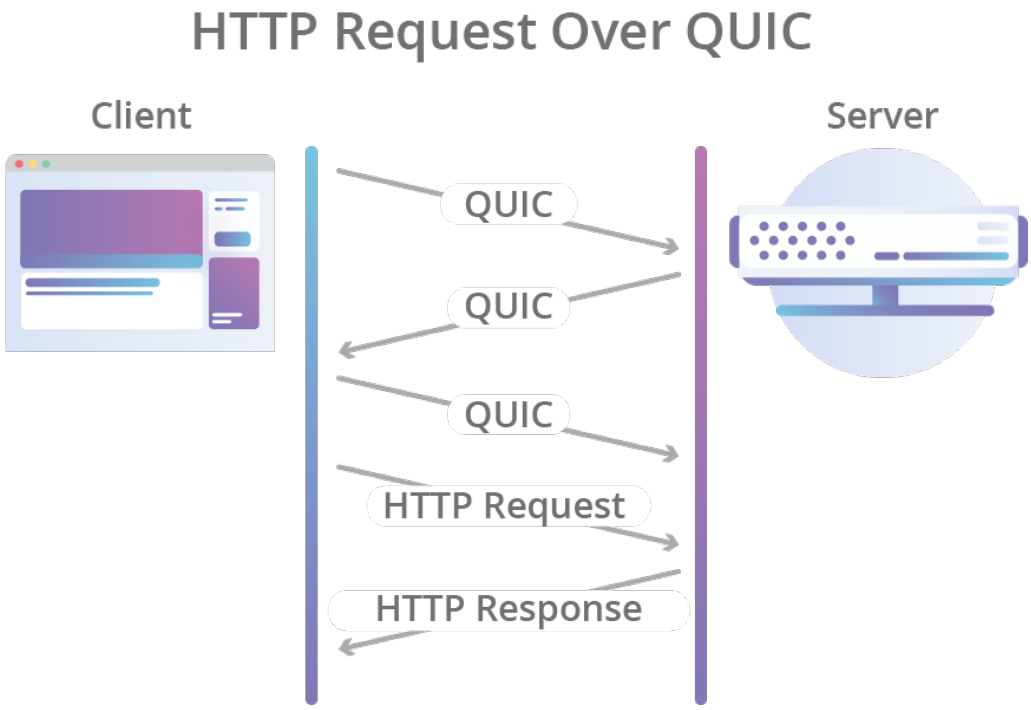


Figure 1.19: Handshake with QUIC protocol

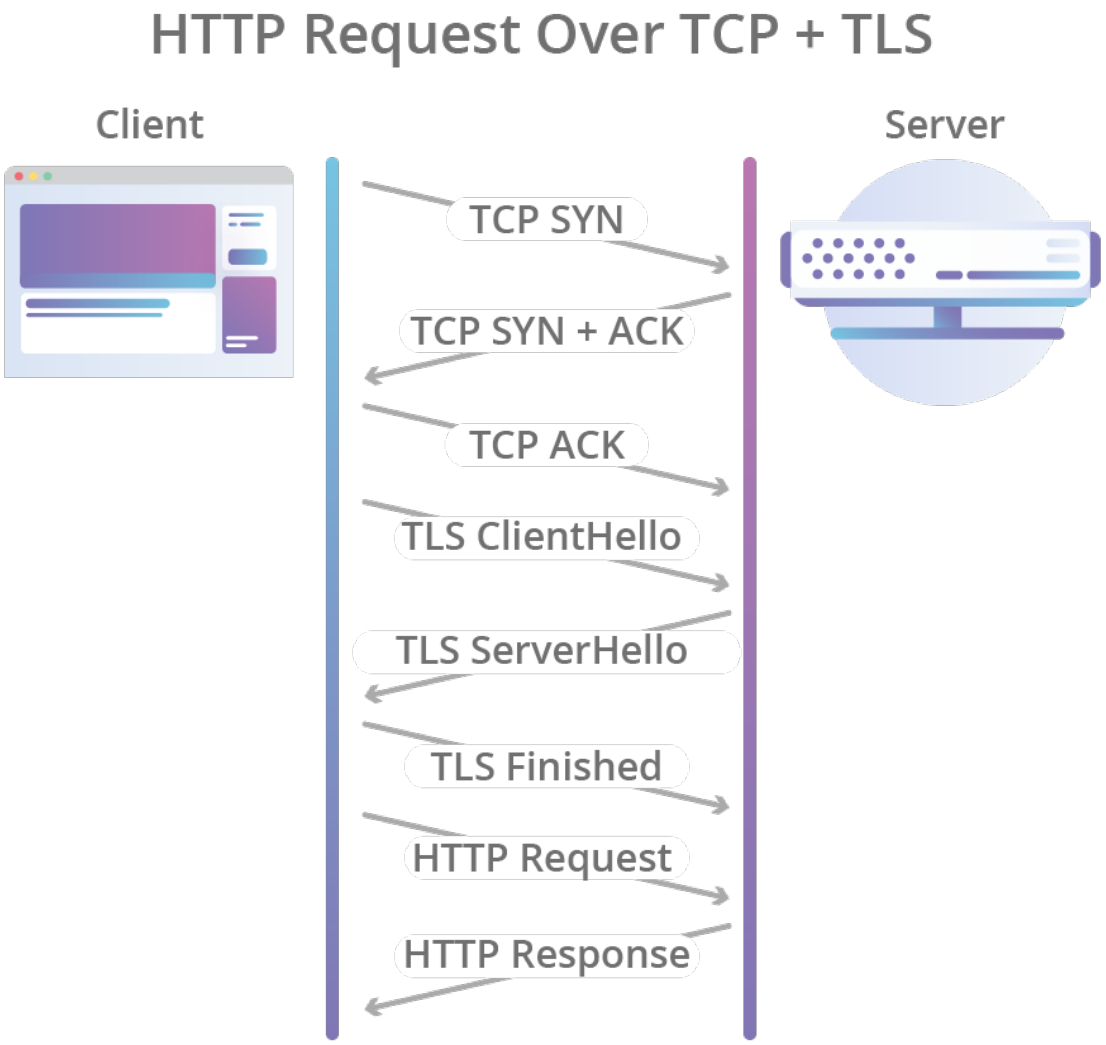


Figure 1.20: Handshake with TCP + TLS

As we can see, to initiate the communication, QUIC combines a classic "three-way handshake" and a TLS 1.3 handshake. This allows both the authentication and the negotiation of the cryptographic parameters which will be used afterwards during the communication. Thus, QUIC replaces the TLS layer with its own data encryption protocol, and thus allows a connection initiation twice as fast as a classic TCP + TLS.

In addition to that, QUIC goes further, by adding an encryption of the connection metadata, allowing to avoid a "man in the middle" attack. By encrypting the packet numbers, QUIC ensures that they cannot be used by a third party to find out information about their activity: only the client and the server can know about their communication.

1.5 Advantages and disadvantages of the QUIC protocol

1.5.1 Advantages

If the QUIC protocol is so widely used nowadays, it is because it has a large majority of advantages. We will present the main ones.

First of all, the QUIC protocol is based on the UDP protocol at the transport level. Thus, it makes it possible to free itself from all the mechanisms of acknowledgement of the transmitted data that TCP requires, and thus to bring new solutions when it comes to the guarantee of reception of the data which would not have been possible with the requirements of TCP, which is connection-oriented contrary to UDP.

Thus, the main advantage of QUIC over TCP is that the connection is much faster. Indeed, there is no need for the "three-way handshake" to initiate the connection, with QUIC, the connection is initiated with only one packet (two if it is the first connection with the server), and on top of that, the encryption via TLS/SSL is provided at the same time. A two-in-one solution that allows you to gain a lot in terms of speed and therefore in terms of decongestion of global traffic.

Another advantage is in the error detection: QUIC implements a "Forward Error Correction" mechanism, which is a technique allowing to control transmission errors especially on a noisy or damaged channel. The principle consists in the transmitter sending the encoded message in a redundant way, which will allow the receiver to detect transmission errors. Even better, QUIC also allows error correction thanks to an error correction code (ECC), not only for inverted/erroneous data, but also for missing data: the message can be completely reconstructed. All this is much more advantageous than a simple system where the server would ask the client to repeat the message: there is only one way to send the message and the traffic flows more smoothly. Because it is also one of the main goals which gave birth to QUIC: to face the increasing HTTP traffic on the net.

1.5.2 Disadvantages

The main disadvantage of QUIC follows from its advantages: it is the security. Indeed, although QUIC guarantees, and in a more efficient way than TCP, the encryption as well as the authentication of the data, the headers of the packets are less clear, and tasks such

as the regulation of traffic/network or the troubleshooting become more difficult by using QUIC.

Furthermore, some performance studies have shown that QUIC shows its limitations when transferring large amounts of data in very high bandwidth networks.

1.6 Study of the different vulnerabilities

1.6.1 Introduction

QUIC packets are sensitive to simple attacks, especially during the handshake, introducing latency, and thus countering one of the main advantages and goals sought by QUIC: the establishment of a connection with a 0-RTT. Several types of attacks are possible: we can quote one exploiting the public information coming from the server or the client, and another exploiting the unprotected fields of packets exchanged during the handshake.

1.6.2 Replay attack

As long as at least one client establishes a session with a particular server, an attacker can learn the public values (SCFG) of the server as well as the token value of the source address (STK), thus corresponding to the client during the validity period. An attacker can thus replay the SCFG corresponding to the server to the client, and replay the STK to the server, and will thus compromise both parties.

Replay attack using the SCFG:

An attacker can replay the SCFG with each client sending a connection initialization request with the server, while keeping the server unaware of these requests. Thus, those clients establishing an initial connection with this server without the knowledge of this server will have their packets subsequently rejected by this server because the server will not be able to recognize them. Although personal data is not affected, a client will suffer from latency and wasted resources.

Replay attack using STK:

An attacker can replay the STK from a client to the server, having already used its token with the client multiple times to establish additional connections. This action will force the server to establish keys as well as security keys for each knowledge without the client being aware of them. Each additional step in the handshake will then fail, but an attacker will be able to initiate a DoS attack, creating a large number of connections from multiple clients, and thus exhausting the server's computation and memory resources.

We can see that these attacks are allowed thanks to the parameters that were supposed

to reduce the latency. This kind of attack seems impossible unless we limit the use of these STK and SCFG tokens to a single use, but however would prohibit the establishment of a 0-RTT connection.

1.6.3 Packet manipulation attack

Not all fields in a QUIC packet are necessarily protected from possible adversarial manipulation. An attacker with access to the communication channel used by the client to establish a connection with the server can change the bits of the unprotected fields, notably the connection id CID, as well as the token of the source address STK. This could lead the server and the client to drift to different keys, and thus cause the connection to fail. In order to succeed in an attack, an attacker must ensure that all the changed parameters appear consistent in each packet sent and received independently, but inconsistent when considering both endpoints (client and server). These attacks do not compromise the confidentiality and authenticity of the communication that is encrypted by the initial key, since even if the initial keys are different, they remain unknown to the attacker. If client and server do not agree on the initial key, a QUIC session key cannot be established because the SHLO (Server Hello) packet is encrypted by the initial key.

These packet manipulation attacks are more complex than simply compromising and defeating the handshake because client and server can progress through the handshake while having an inconsistent conversation, resulting in inconsistent key establishment.

One way to mitigate this type of attack would be to sign each of the modifiable fields in its s_rejects and s_hello packets. However, these signatures would increase the computational cost of these various signatures, and leave the opportunity for a DoS attack, in which an adversary, using IP Spoofing, could send a large number of connection initiation requests on behalf of as many clients as desired.

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: Attacks Discoveries and Properties

QUIC adaptation to VPNs

2.1 Generalities about VPNs

A VPN (Virtual Private Network) is a network technology allowing to build a private network inside a public infrastructure. This one allows to create a direct link between distant computers. It is now a major component of computer networks and distributed computing.

2.1.1 General principle

A VPN can be of point to point type (client-concentrator), but also in the form of a tight and distributed private network of MPLS type. It must allow :

- User authentication, authorizing access to the VPN for a certain number of users
- Address management, in which each user has a private and confidential address,
- Data encryption, allowing to protect data transiting on the public network,
- Key management, allowing the client and server to generate and regenerate keys,
- Multi-protocol support.

A VPN network is based on a tunneling protocol, which allows information from a sender to flow from one end of the tunnel to the other in an encrypted manner. Thus, a user will feel as if he/she is connecting directly to a remote network. In order for this data to be readable from one end to the other, the tunneling protocol used by all components of a VPN must be the same. Plusieurs types de protocoles de tunneling existent, dont :

- PPTP : Point to Point Tunneling Protocol ;
- L2TP : Layer Two Tunneling Protocol ;
- IPSec .

The principle of tunneling is to establish a virtual path after identifying the sender and the recipient. Then the source encrypts the data and routes it using this virtual path.

The data to be transmitted can be processed via protocols other than IP. In this case, the tunneling protocol encapsulates the data by adding a header. Tunneling is the entire process of encapsulation, transmission and decapsulation.

2.1.2 Types of VPN

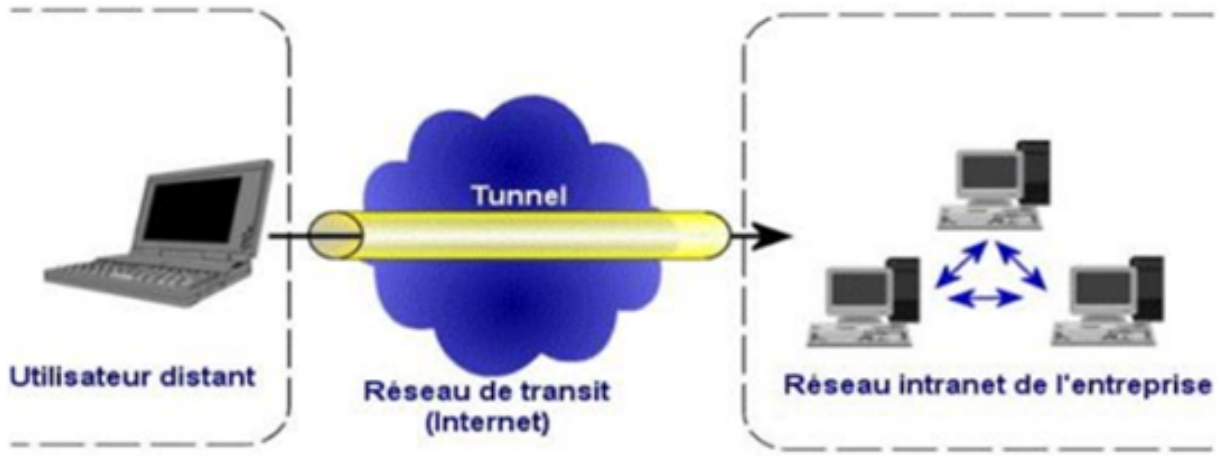


Figure 2.1: VPN Access

This type of VPN allows roaming users to access a private network. These users must use an internet connection to initiate a VPN connection. 2 cases are possible:

- The user asks the ISP to establish an encrypted connection to the remote server: he communicates with the ISP's NAS (network access server), and the NAS establishes an encrypted connection.

- Users have their own VPN client software. In this case, they directly establish communication with the private network in an encrypted way.

There are several advantages and disadvantages to these different methods:

- If the user goes through his ISP to establish a VPN connection with the remote network, he will be able to communicate through multiple networks by creating multiple tunnels, but this requires the NAS to be compatible with the solution adopted by the remote network, and the customer is also exposed to security risks, as the communication with the NAS is not encrypted;

- Conversely, this security problem does not arise in the second solution, since all communication is encrypted. On the other hand, it requires that each client carries the

software.

2.1.3 VPN Site-to-Site

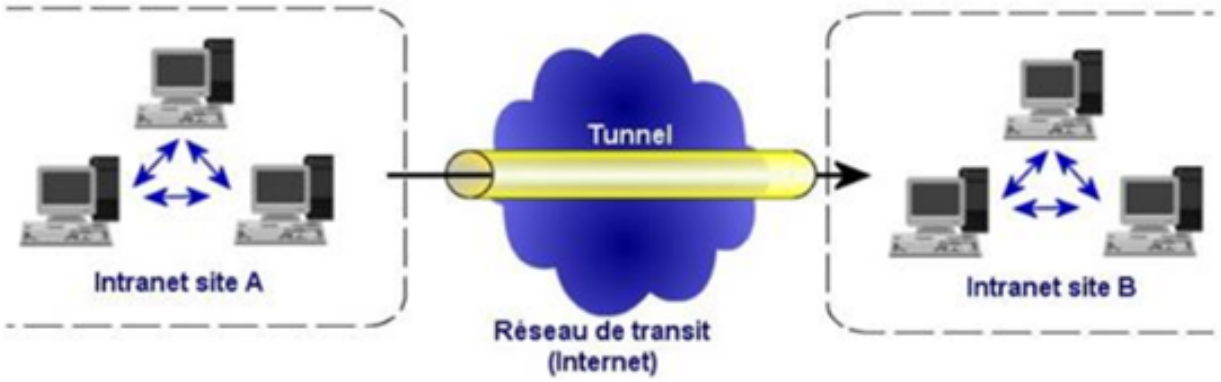


Figure 2.2: VPN Site-to-Site

This type of VPN makes it possible to connect 2 networks between them, and proves to be concretely useful for the companies having several remote sites. Given the very discrete use of this one, two major components correspond to the security and the integrity of the data, since sensitive and confidential data can be brought to transit on this network. Authentication at the packet level will ensure the validity of the data, its non-repudiation, as well as the identification of the sender of the data. The IP encapsulation as well as the cryptographic algorithms (and their signature are added to the packet) allow to ensure a sufficient level of security.

2.1.4 Tunneling protocols

2.1.4.1 IPSec

The IPSec protocol corresponds to a set of protocols defined by the IETF, and allowing a high level of security of the network layer. This protocol is based on 2 data protection mechanisms (applicable on IPv4 and IPv6), which are :

- AH, Authentication Header : ensures the integrity and authenticity of IP datagrams, without encrypting the data (Protocol ID : 51)

- ESP, Encapsulating Security Protocol: authenticates data while encrypting them. (Protocol ID: 50)

2.1.4.1.1 Authentication Header This one allows to ensure the authentication of the data sent via an IP datagram, by checking the identity of the 2 ends of the tunnel, but also allows to ensure the integrity of the data. It is inserted directly after the IP header.

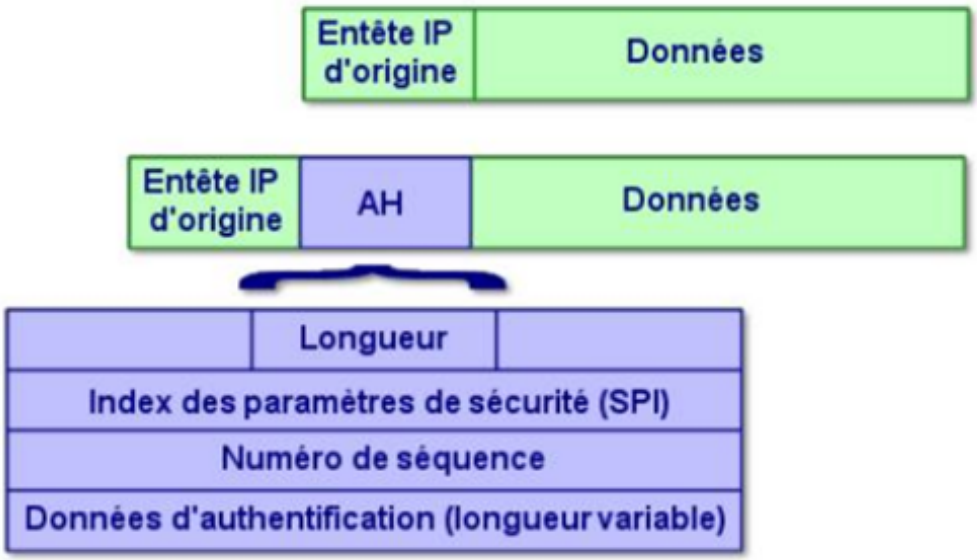


Figure 2.3: Authentication Header

The lack of confidentiality of this mechanism will ensure that this standard can be widely used on the Internet. The authenticity and integrity of the data is ensured by the "Authentication Data" block on the schema as well as the signature of the packet, or more commonly called ICV (Integrity Check Value), whose hash will allow to verify these two parameters. Moreover, the sequence number allows to check in reception if the anti-replay mechanism is well and truly activated. Finally, the SPI field allows to identify the security association to use for the packet. The fields " SPI ", " Sequence number " and " Authentication data " are all 3 coded on 32 bits.

2.1.4.1.2 Encapsulating Security Payload (ESP) In addition to ensuring data integrity and authentication, ESP also ensures data confidentiality.

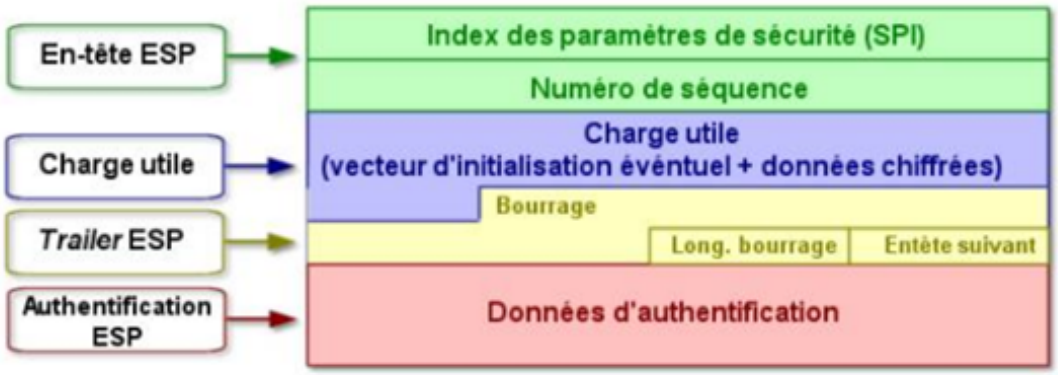


Figure 2.4: Encapsulating Security Payload

Confidentiality via is ensured as follows:

1) The sender encapsulates in the "payload" the data carried by the original datagram, or even the IP header in tunnel mode;

2) He then adds a stuffing (to align with a 4 bytes format) if necessary;

3) The result (load, stuffing, length and next header) will then be encrypted,

4) Adds cryptographic synchronization data if necessary.

Data integrity and authentication are ensured by fields similar to the fields presented in the AH mechanism.

2.1.5 Mode of operation

These 2 mechanisms can be used in 2 different modes:

- Tunnel mode: the entire IP packet is encrypted/authenticated.

- Transport" mode: only the transferred data is encrypted/authenticated

2.1.5.1 Tunnel Mode

In this mode, the data sent by the application crosses the protocol stack up to and including IP, then is sent through the IPSec module. This encapsulation by the IPSec module thus allows address masking. In this default mode, the IP packet is encrypted, then a new IP header is added and sent to the other end of the VPN (VPN pair). This is mainly used for communications between routers, or between terminals and routers. It allows to create

new virtual networks.

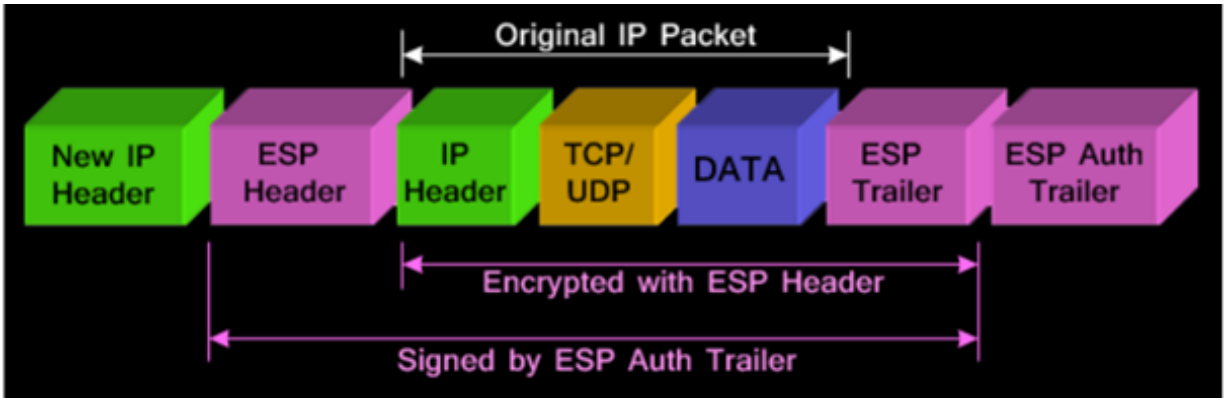


Figure 2.5: IPSec Tunnel Mode with ESP

As you can see on this datagram, the whole packet is encrypted, and a new external header is added.

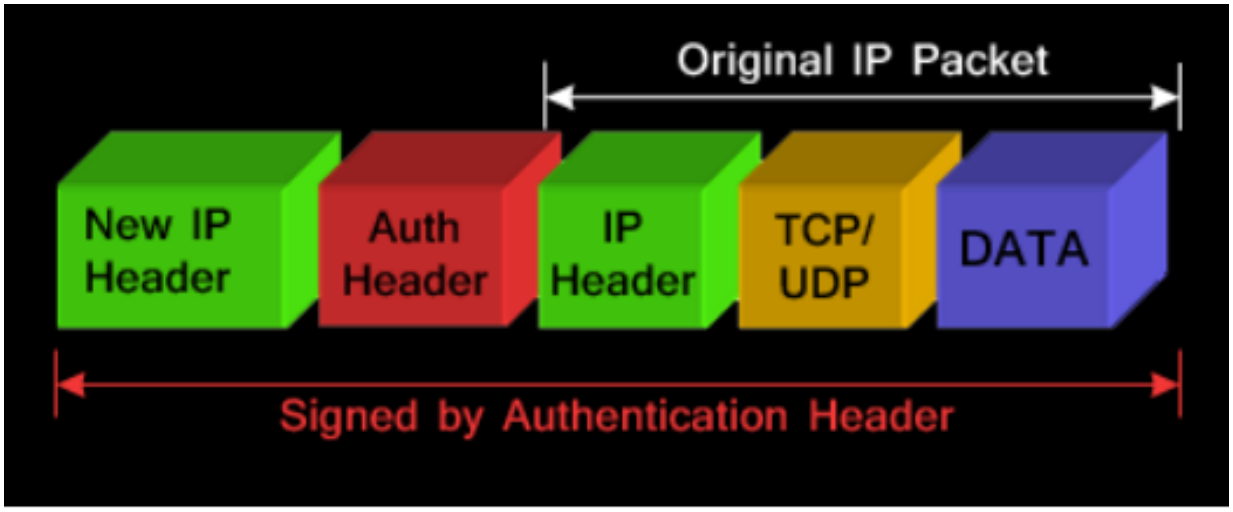


Figure 2.6: IPSec Tunnel Mode with AH

The use of AH in tunnel mode is to authenticate the entire IP packet. On the other hand, since confidentiality is not ensured, ESP remains privileged over AH.

2.1.5.2 Transport mode

In this mode, the security mechanisms are applied at the transport layer (signature, encryption), and the resulting packet is transmitted to the IP layer. The insertion of IPSec is thus transparent between TCP and IP. TCP sends its data to IPSec, which in turn sends it to IP. This mode does not allow address masking, but is relatively simple to implement. This mode is mainly used for communication between 2 terminals (client-server), but also when another "tunneling" protocol (notably GRE) is used to encapsulate the IP data packet. Data protection is ensured via this mode, consisting of a TCP or UDP header and the data, authenticated and/or encrypted by AH or ESP. The original IP header is not modified, except for the IP protocol (50: ESP, 51: AH), and the original protocol is included in the ESP trailer, which will be used during decryption.

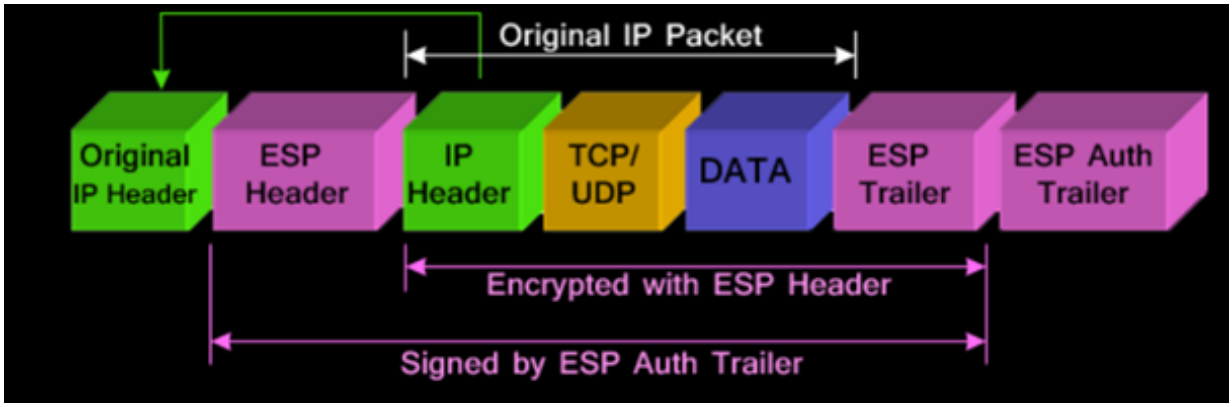


Figure 2.7: IPSec mode Transport with ESP

As we can see on this datagram, the original IP header is placed at the beginning of the packet, before the ESP header. This implies that this header is not encrypted, and therefore does not provide address masking. The use of ESP in transport mode therefore only allows the encryption of IP payloads. The protocol ID of ESP is 50.

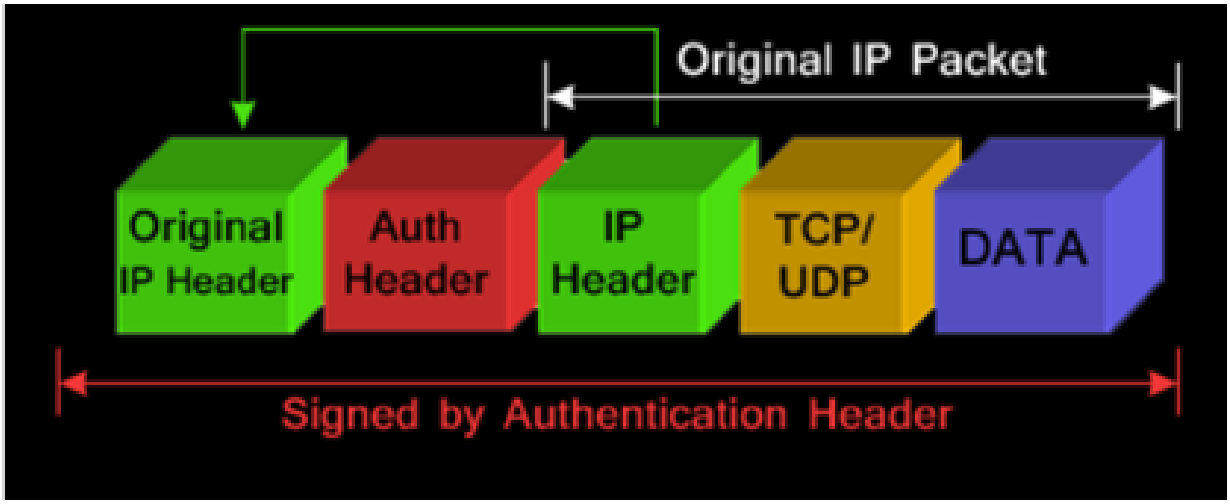


Figure 2.8: IPSec mode Transport with AH

The purpose of AH is to protect the whole packet. However, it places a copy of the original header at the beginning of the packet instead of creating a new one, and thus does not provide protection for the sensitive fields of the IP header (notably IP Source and IP Destination). The protocol ID of AH is 51. To manage these security mechanisms, the IPSec protocol must provide key management (generation, distribution, storage and deletion). To ensure this need, the IKE (Internet Key Exchange) system is used to provide authentication and key exchange mechanisms on the Internet.

2.1.6 IKE Protocol

IP-independent, the IKE protocol uses UDP on port 500, and is used for:

- Negotiation of security protocols;

- Authentication;

- Key generation/exchange.

It uses ISAKMP (Internet Security Association and Key Management Protocol), which is used to negotiate, modify and delete security associations and their attributes.

A Security Association (SA) is a connection that provides security services to the transported traffic, and is identified by 3 components:

- Destination address ;

- Protocol ID (AH or ESP);

- SPI, corresponding to the SA identifier.

Each SA is unidirectional: thus, the 2 directions of communication are protected by a bundle of SAs (corresponding to a set of SAs), 1 SA for each direction of communication, but also 1 per communication protocol. Each SA is stored in a database of security associations (SAD), consulted each time the IPSec layer receives data to send, thus making it possible to provide the security mechanism to be used (if the packet is previously accepted by the SPD, a database of security policies, indicating whether a packet must be accepted or rejected). ISAKMP has 3 components:

- Definition of a way to proceed: in 2 steps (phase 1 and 2), it will first (phase 1) define a set of security parameters specific to ISAKMP in order to establish a protected channel, then in a second step (phase 2) will define the security protocol to use (AH or ESP)

- Definition of message formats,

- Typical exchanges, allowing negotiations: PFS (Perfect Forward Secrecy), thanks to which keys used before or after cannot derive from the key in use.

IKE for its part uses ISAKMP in order to :

1) Establish a 1st tunnel (administrative tunnel, having for role to manage the other tunnels) between the 2 entities;

2) Establish a 2nd tunnel (or several a posteriori), corresponding to the IPSec tunnel (or data tunnel).

Several modes for IKE exist:

- Main mode ;

- Aggressive mode ;

- Fast mode;

The "Main" and "Aggressive" modes are used in Phase 1, while the "Fast" mode is used in the Phase 2 exchange process.

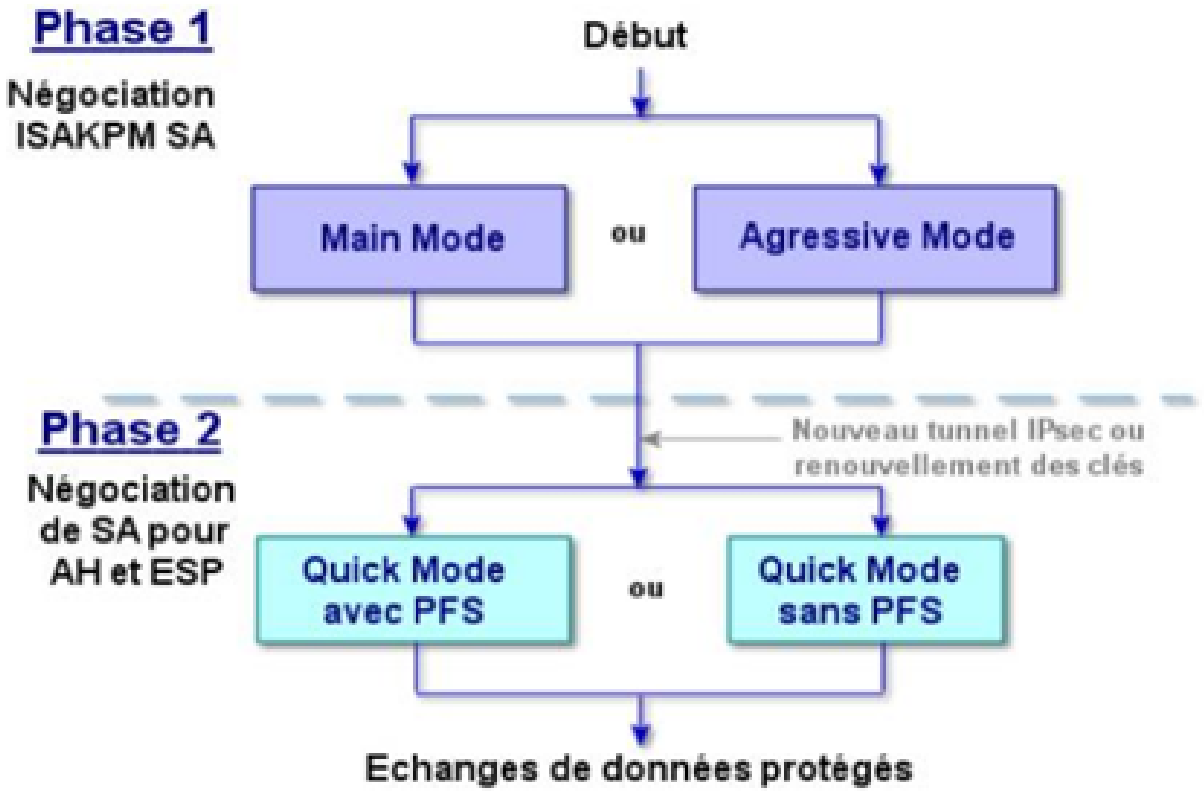


Figure 2.9: Phases IKEH

2.1.6.1 Phase1

Phase 1 allows via ISAKPM to negotiate a set of parameters:

- Encryption algorithm;
- Hash function ;
- Identification method
- Diffie-Hellman Group

The main difference between the "Main" mode and the "Aggressive" mode lies in the number of messages sent during this phase. Indeed, in the "Main" mode, the negotiation packets (Cipher Algorithm, SA, Identification Method) are sent one by one, while they are concatenated in the "Aggressive" mode, allowing a much faster negotiation.

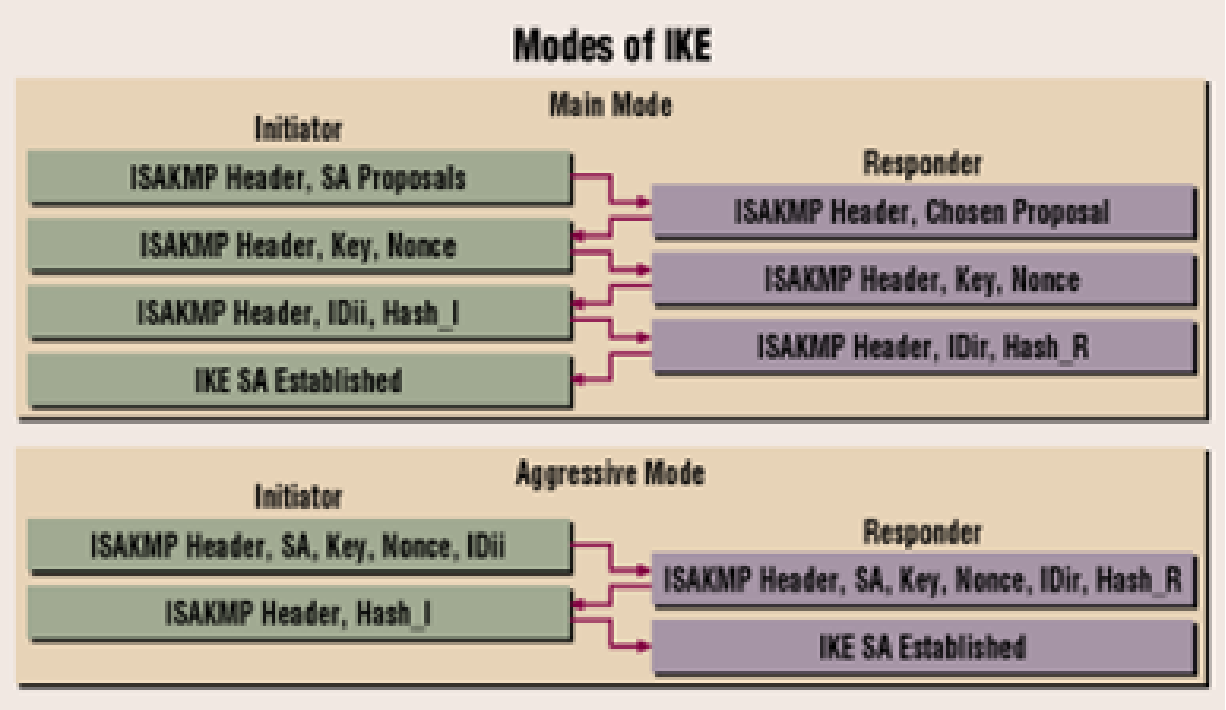


Figure 2.10: IKE Methods Phase 1

At the end of this phase, 3 keys are generated: one for encryption, another for authentication and a last one for the derivation of future keys. This makes it possible to establish an ISAKMP SA.

2.1.6.2 Phase 2

The parameters negotiated in phase 1 ensure the authenticity and confidentiality of the elements exchanged and the messages sent during phase 2. The "Hash" block following the ISAKMP header ensures authenticity, and confidentiality is ensured thanks to the encryption of all the blocks in the message. This phase makes it possible to negotiate SAs for other security protocols (notably IPSec). This negotiation allows the negotiation of SAs for security protocols, at least one for each communication direction (resulting in the negotiation of at least 2 SAs). The exchanges are done in the "Fast" mode, and have several roles:

- Negotiate a set of IPSec parameters ;
- Exchange random numbers, allowing to generate new keys derived from the secret generated in phase 1 using Diffie-Hellman.

IPSec is a layer 3 protocol, and is nowadays widely used in connection with VPNs. However, although very used, other "tunneling" protocols are also used: we can quote 2 categories:

- Layer 2 protocols: PPTP and L2TP

- Layer 3 protocols (including IPSec): MPLS.

Here we will focus on another protocol used, and which allowed the creation of a new type of VPN: SSL, which allowed the advent of VPN-SSL.

2.2 Architecture of SSL/TLS VPNs

After presenting what a VPN is and enumerating several types of them, it is appropriate to focus on SSL (Secure Socket Layer) VPNs, of which OpenVPN is an example. Our goal will be to adapt an SSL VPN by integrating QUIC to manage both the encryption of data and their transmission simultaneously. To do so, it is necessary to understand the principle of an SSL/TLS VPN.

While most VPNs are based on the IPsec protocol, the SSL VPN uses SSL to encrypt data. This protocol works on classic TCP/UDP ports and is generally configured on port 443, which is the "HTTPS" port, associated with the HTTP protocol secured by SSL. Note that on UDP, we speak about "DTLS".

2.2.1 Brief history and interest

SSL (Secure Socket Layer) is an encryption and security protocol for exchanges on the Internet, created in 1994 by the Netscape organization. It gave rise to 3 successive versions in 1994, 1995 and 1996, then the third version SSL 3.0 was taken over by the IETF and replaced in 1999 by TLS (Transport Layer Security), the successor of SSL. SSL or TLS or SSL/TLS are commonly used to designate the encryption protocol widely used in current communications. The version currently in use is SSL 3.0/TLS 3.1 .

The clear advantage of SSL/TLS is its ergonomics: indeed, it integrates very well with the protocols commonly used on the net, in particular HTTP, which only needs to be modified very little to allow the connection to be established with SSL. The latter allows the connection between the transport layer and the application layer. It is commonly considered to be part of the session level (level 5) of the OSI model. As explained above, the fusion between HTTP and SSL gives him the HTTPS protocol.

2.2.2 SSL operation

The architecture of the TLS/SSL encryption protocol is of client-server type. The initiation of a communication proceeds in the following way:

1) The client and the server agree on the version of the protocol to use

2) The cryptographic algorithm for future communications is chosen

3) The client and the server authenticate each other by exchanging and validating digital certificates.

4) Asymmetric encryption is used to generate and share the shared secret key, in order to secure its transmission

5) Once this key is shared, the client and the server can now communicate by encrypting their messages with the key shared in 4). The encryption is therefore symmetrical this time, which is faster than an asymmetrical encryption. Here is a detailed diagram showing how a handshake using this protocol works:

Here is a detailed diagram showing how a handshake using this protocol works:

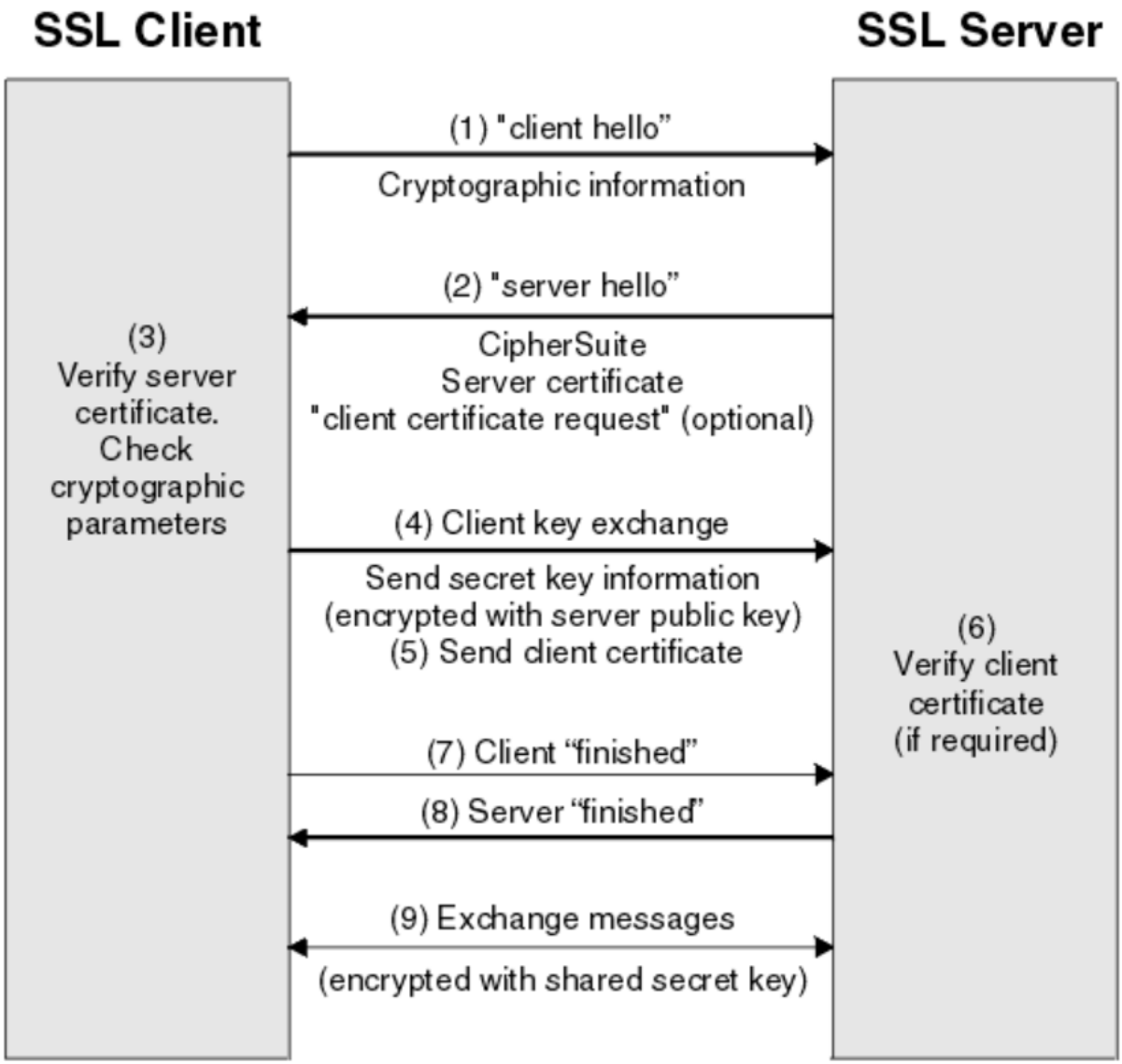


Figure 2.11: SSL Handshake

Step 1: The SSL/TLS client sends a "client hello" message to the server, in which it lists information such as the version of SSL/TLS it supports as well as the list of cryptographic suites (set of supported encryption algorithms) in order of preference, from the most robust to the weakest. It also sends the compression methods it supports. Moreover, it sends a string of random bytes which will allow to attack a "man in the middle" type attack (see step 2)

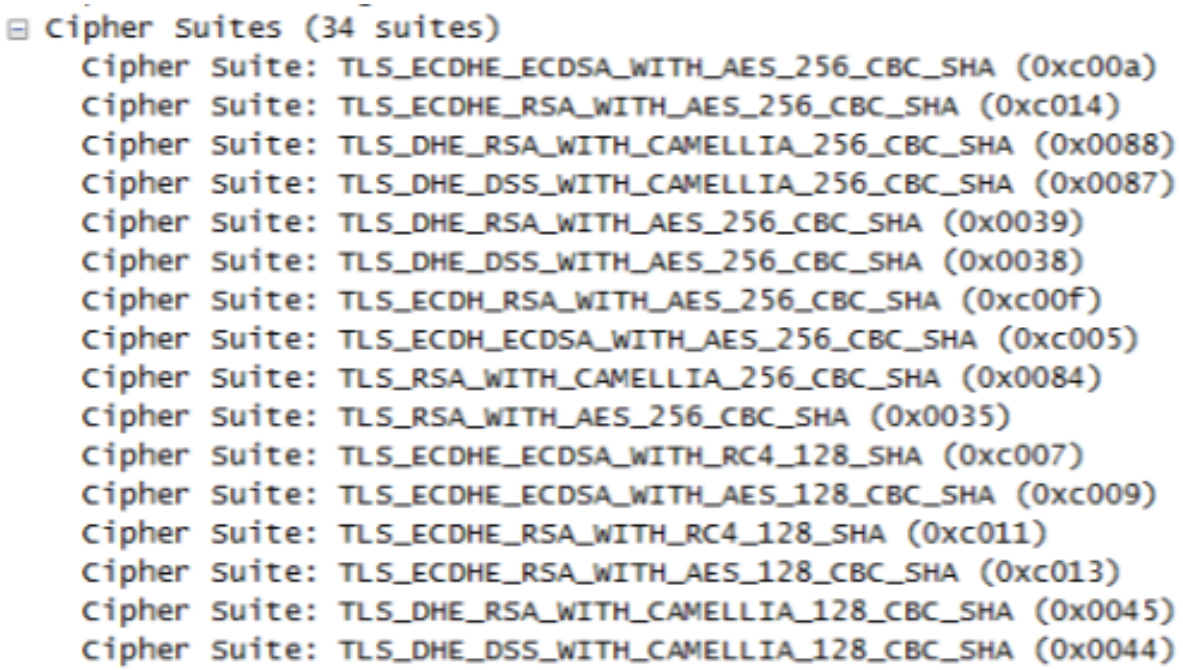


Figure 2.12: Example of Cypher Suites proposed during a "Client hello"

Step 2 : The server answers with a "server hello" which contains the cryptographic suite which it chose among those proposed by the client, as well as the session ID and a random string of 32 bytes. These random strings are important because they will allow to verify that the client is talking to the server and vice versa, as we will explain later. The server also sends its digitized certificate (X.509), and can request one from the client to authenticate itself by sending a "certificate request", in which it specifies the certificate formats it supports. He also sends his public key to the client, which will be needed for the server key exchange. Finally, it sends a "server Hello Done" indicating that the ServerHello and the associated messages are finished.

Steps 3, 4 and 5: The client checks the digitized certificate sent by the server. If the server has sent a client certificate request, the client also sends a random byte string encrypted with its private key and containing its digital certificate. (or a "no digital certificate alert"). In some handshakes, the absence of this client authentication will prevent the communication from being established. In addition, the client sends another string of random bytes ("pre master secret") which will be used, combined with the "client random" and "server random" (steps 1 and 2), to generate the "master secret" which will be the symmetrical key used during future communications. This "pre secret master" string is encrypted with the server's public key. This whole step is called "client key exchange".

We understand then the role of "random client" and "random server" at the beginning.

If they had not taken place, all the generation of the final secret key would have been done by the client, which sends the "pre master secret". Thus, if the server generated messages with an encryption set by the client, in a "man in the middle" type attack, one could imagine an attacker who collects the "pre master secret" key sent by the client and pretends to be the server without this being detected by the client (replay attack). For this, it is important that there is a construction of the key based on both sides, and different at each communication: we talk about "random generation". If the server sends a random string to the client at the beginning, the latter can verify later that it is indeed the same server that had sent the random string at the beginning, since it is used in the design of the final "secret master" key and that moreover only the server could decode the "pre master secret" with its private key.

Step 6: The SSL/TLS server verifies the client certificate if requested. It decodes the "pre master secret" with its private key (it is the only one to be able to do it. The master secret can then be calculated on both sides: master_secret = PRF(pre_master_secret, "master secret", ClientHello.random + ServerHello.random) where PRF is a pseudo-random function. Below is a diagram explaining in detail the calculation of this "master secret":

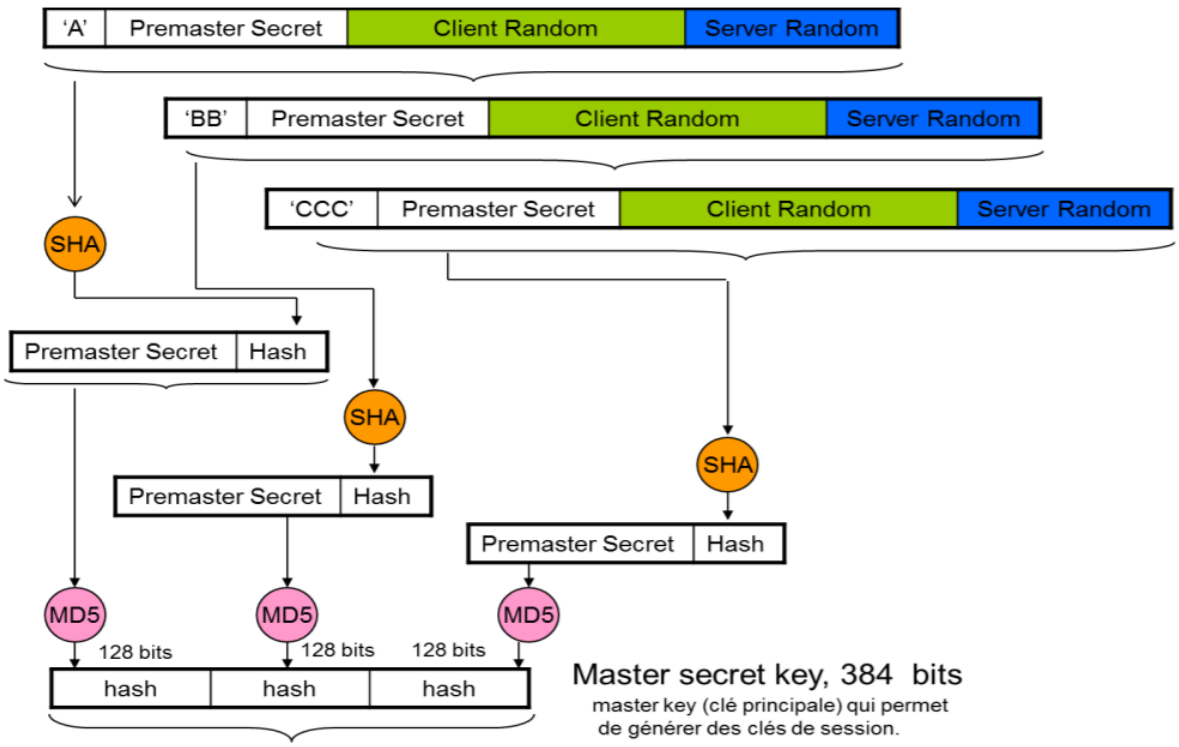


Figure 2.13: Generation of the Secret Master

Step 7: The client then sends the server a "final" message ("client finished"), which is encrypted with the new secret key calculated as described above ("change cipher spec"). With this step, the client indicates that the handshake is finished on its side, and will be able to verify that the server was able to decode the message with their common secret key.

Step 8: The server sends to the client a "final" message ("server finished") which is also encrypted with the new common secret key ("change cipher spec"), and indicates in this way that the handshake is also completed on its side.

Step 9: For the rest of the conversation, the client and the server can now exchange symmetrically encrypted messages with the shared secret key.

During the next connections, new keys can be derived from the "Master Secret" using, for example, the MD5 hash function combined with new random strings generated on the server and client sides.

2.2.3 Authentication management

Let's detail how authentication was possible in the above communication.

In the case of the server authentication : the client uses the public key that the server gave him during the "server hello" in order to encrypt the data that will be (notably the pre master secret) used to generate the secret key. Thus, the server can generate the secret key with the client if and only if it can decode these data with its private key. (that he is the only one to have).

In the case of client authentication: the server uses the public key provided in the client's certificate to decode the data that the client has sent to it during step 5. The principle is the same, authentication is then assured.

The exchanges of "finished" messages with the shared key ensure that the authentication is complete. Moreover, the authentication is complete if and only if each step of the handshake has taken place correctly.

The role of certificates also allows to guarantee authentication even more. Their management is quite complex, but the important thing to remember is that SSL certificates play a very similar role to client certificates. However, in the case of a client, the certificate is used to identify the client/individual itself, while in the case of a server the certificate authenticates the owner of the site.

If we call CA Client the client's certificate authority and CA Server the server's certificate authority, in the case of client and server authentication, the server needs :

a personal certificate provided by CA Server its private key a client certificate provided by CA Client The client needs :

a personal certificate provided by CA Client his private key a server certificate provided by CA Server

We observe that in terms of authentication, the exchanges are done in a similar way for the client and the server.

The certificates contain various information such as a serial number, a validity date, personal data on the owner (name, address, e-mail...)

2.2.4 Certificate verification

There are 4 steps in certificate verification:

verification of the digital signature the "chain of certificates" is verified: normally the client and the server must have had intermediate certificates the expiration date, activation date and validity period are checked the revocation status of the certificate

2.2.5 Privacy management

On the one hand, it can be observed that the exchange is based on alternating symmetrical and asymmetrical encryption, which reinforces the security of the communication.

In order to minimize the risk in case of theft or alteration of the secret key, the latter can be renegotiated periodically, and the old one is then no longer valid. The same security principle is used when sending random strings as explained above, which avoid "man in the middle" attacks.

During the handshake, the client and the server agree on the cryptographic primitive used as well as the secret key which will be used only for the current session. All messages during the session will therefore be encrypted with both the chosen primitive and the shared secret key, which ensures that the message remains private and will not be intercepted. Random strings only further reduce the already low risk of interception.

Moreover, the fact that the encryption used to transmit the private key is asymmetric avoids any problem of key distribution (no risk of interception).

2.2.6 Integrity management

The integrity is mainly allowed by the calculation of condensates (hash) of messages with secret key (hmac). The hash functions used are provided in the list of cryptographic suites used by the client, and it is then the server that will make the final choice, which it will communicate to the client. The hash function used is therefore clearly defined on both sides. For this, it is however important that the list of cryptographic algorithms supported

by the client is not of type "None", otherwise the integrity is strongly compromised.

2.2.7 Non-replay management

Non-replay is ensured by the sequence numbers contained in the packets.

2.2.8 SSL applied to VPNs: OpenVPN example

A VPN implementing the SSL/TLS encryption protocol described above is called an SSL VPN (Secure Sockets Layer Virtual Private Network).

In this type of VPN, the tunnel is implemented on top of SSL.

The general principle of a VPN then applies: users can connect, even more securely than a VPN without SSL, to a remote network such as their corporate network.

Figure 2.14: OpenVPN

A classic example of SSL VPN is OpenVPN: it is an open source VPN that uses SSL/TLS data encryption unlike most Vpn (based on IPsec). Indeed, OpenVPN is based on the OpenSSL cryptographic toolkit which allows to implement SSL/TLS. The encryption is usually 256-bit.

Moreover, one of its advantages is that it can work on any port on the server side, so you can run transport level protocols (TCP or UDP) on port 80 (or 443) in particular. OpenSSL VPN traffic is also quite similar to HTTPS traffic, which makes it difficult to identify and therefore block. The fact that all ports are available also allows OpenVPN to bypass almost all existing firewalls.

Regarding the cryptographic primitive used, OpenVPN mainly works on an AES256 type encryption, which is considered very robust.

Moreover, OpenVPN is multi-platform and can therefore be implemented on all operating systems. However, an important point to note is that OpenVPN requires a third-party application to install the client on one's computer: it is not natively supported by operating systems.

OpenVPN is known to be one of the most reliable VPNs, especially thanks to its additional SSL layer that makes communications even more secure. It can also

Note that OpenVPN works best on UDP according to its official website, so the OpenVPN access server tries to establish UDP connections first. Only if these connections fail, TCP will be used. UDP is therefore chosen by default when installing OpenVPN. This allows to gain in speed compared to TCP, especially since the strength of OpenSSL (strong encryption) also makes it relatively slow.

2.3 Implementation of the QUIC VPN protocol in python

2.3.1 Protocol description

The goal of this part is to implement in python the QUIC VPN. The protocol chosen for the programming is the following:

Initialization of virtual interfaces on both sides

The client initiates a QUIC handshake with the server.

Then, the client is in front of an interface where he has to type his login and password. These are sent via a QUIC stream in base64, the confidentiality being already assured by QUIC.

The server reads the stream and checks that the login/password combination is authorized. If it is the case, it sends to the client a message "Authentication successful", otherwise, "Authentication failed".

Then, if the authentication is successful, the server starts a thread that will listen to everything that is sent by the kernel on the virtual interface to encapsulate the received IP packets in a QUIC stream and send this to the client. The server also listens to all the QUIC events sent by the client to decapsulate them and rewrite them on the virtual interface.

On the client side, if it receives an "Authentication successful" from the server, it listens to the IP packets sent by the kernel on its virtual interface to encapsulate them in QUIC and send them to the server. It also listens to the QUIC streams sent by the server to decapsulate them and write on the virtual interface.

We can add the necessary routes afterwards.

In our implementation, we have made a client-to-server VPN.

2.3.2 Implementation

We have based ourselves on the implementation of QUIC of aioquic.

(https://github.com/aiortc/aioquic). We have chosen this implementation because it offers an API for programming applications based on QUIC. For the processing of virtual interfaces, we used pytun (https://github.com/montag451/pytun). Our code is available in the appendix.

Let us present the main parts of the 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()
```

We set up a virtual interface 'mytunnel' by configuring the type of interface (tunnel), the source and destination IP addresses, the network mask, the MTU. We set the MTU to 1048 bytes because this is the maximum amount that can be transported on a QUIC stream. If we put more, there is fragmentation on the QUIC side.

Then we have the following method which will be executed to connect to the server via the QUIC handshake and then launch the authentication request (query method):

```
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()
```

The client will then, after the handshake, make an authentication request which follows:

```
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)
```

The client must then enter a login and password, these are encoded in base64 separated by a ":", then sent in a QUIC stream.

On the server side, we have a quic_event_received method which will be called as soon as a QUIC event is received from the 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))
```

First, the server receives the QUIC frame, it decodes it with the auth_check method:

```
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")
```

The server decodes the base64, separates according to the ":" and authorizes if it has for login root and password toor. It returns Authentication succeeded if it is the case, Authentication failed otherwise. It returns it encoded in bytes to prepare the writing in the QUIC stream.

The server then sends the answer to the client. If the authentication was successful, the server launches the thread which will listen to the virtual interface to encapsulate in the QUIC and send to the 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()
```

On the client side, once the Authentication succeeded, we launch the same type of thread. Once this is done, the two client and server points can start exchanging via the

tunnel. To indicate this change of state, the global variable COUNT is present to tell the client and the server how to handle the received streams. Before authentication it is 0, after authentication it is 1.

If the client sends a packet to the server via the virtual interface "mytunnel", it will be encapsulated in QUIC and sent by the tun_read thread to the server.

The latter will then decapsulate this stream and write the IP packet in the virtual interface, hence the part of code present in the quic_event_received method:

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

The same code is present on the client side so that it can decapsulate the QUIC streams and write them on its virtual interface.

2.3.3 Tests

To test this VPN, we decide to realize the following tests: ping, access to an html page present on the server via http, telnet, ssh.

First of all, we launch the server with the following command:

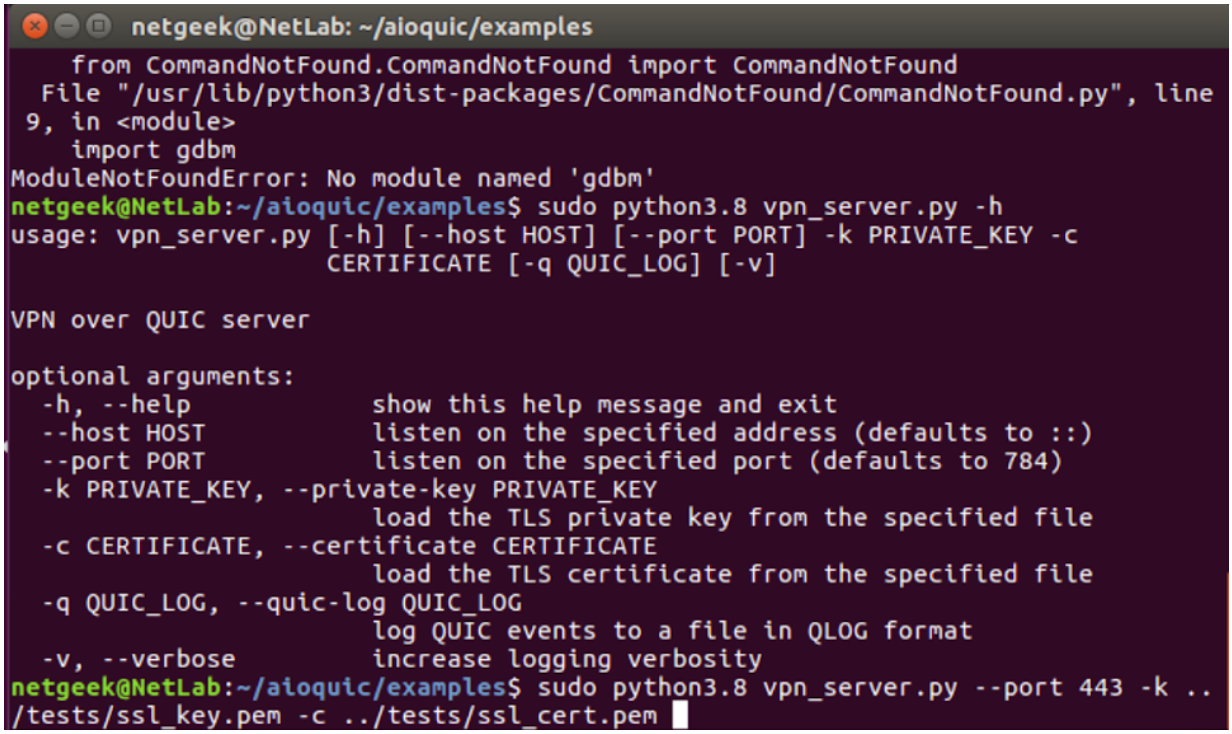


Figure 2.15: Server launch

We do the same on the client side:

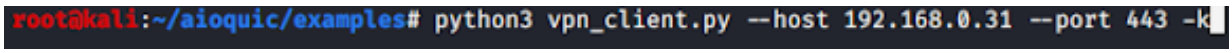


Figure 2.16: Client Launch

We have then, the authentication to realize:

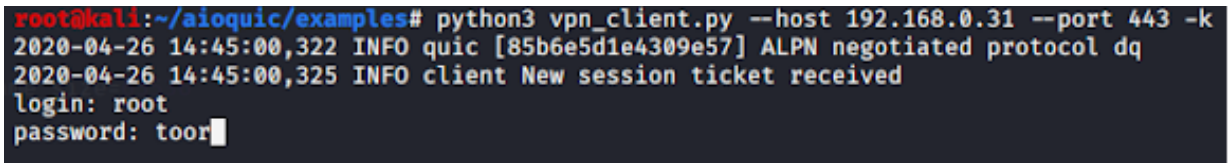


Figure 2.17: Authentication

Once this is done, the tunnel is well created, we can communicate with the server with the address of its virtual interface (10.10.10.2). We can see on the client side that the virtual interface has been initialized:

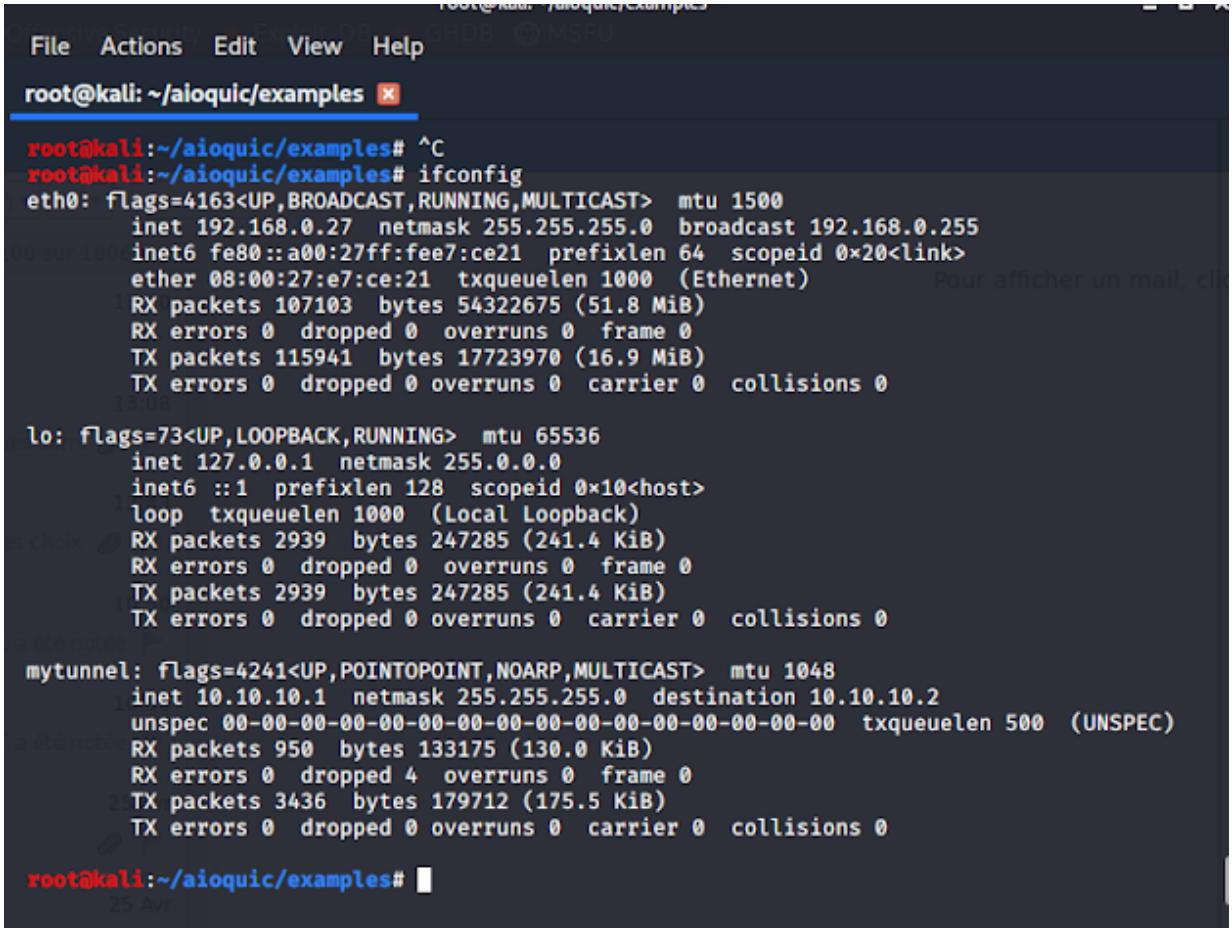


Figure 2.18: Virtual interface verification

We can now begin our tests:

2.3.3.1 ping

From the client, we try to ping 10.10.10.2. The traffic is supposed to go through the mytunnel virtual interface:

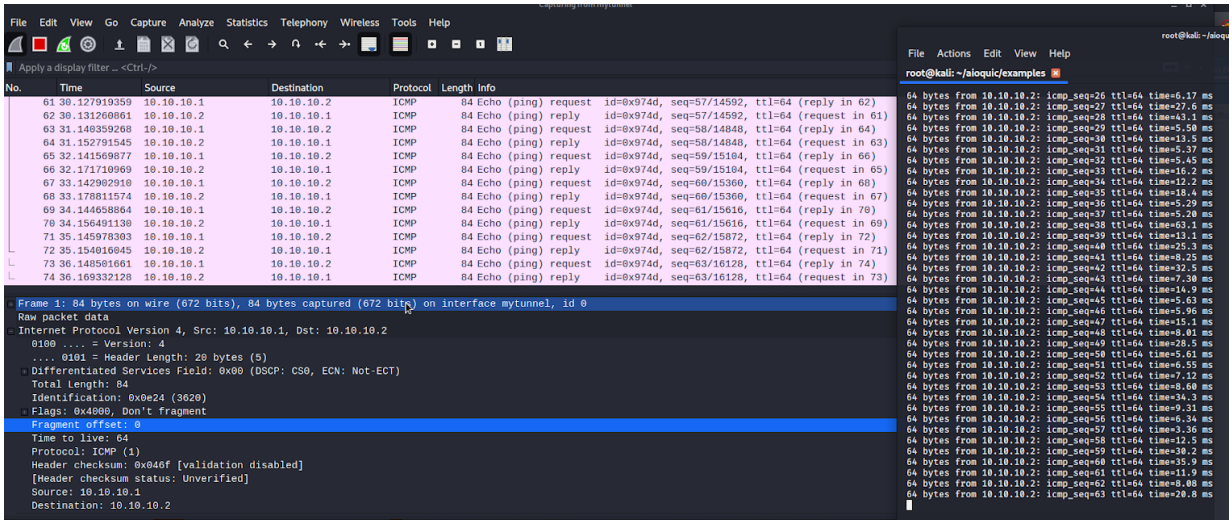


Figure 2.19: Visualization of a ping with wireshark

As the capture shows, the ping goes very well. The ICMP echo reply and echo request packets are well captured by Wireshark which listens on the client side virtual interface.

2.3.3.2 HTTP

On the server side, we start the http server on port 80 with the command below:

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

Figure 2.20: Launch server (HTTP)

On the client side, in a browser we type "http://10.10.10.2:80" Still on the client side, we see that the page loads quickly and wireshark, the packets captured show that there was no need for retransmission:

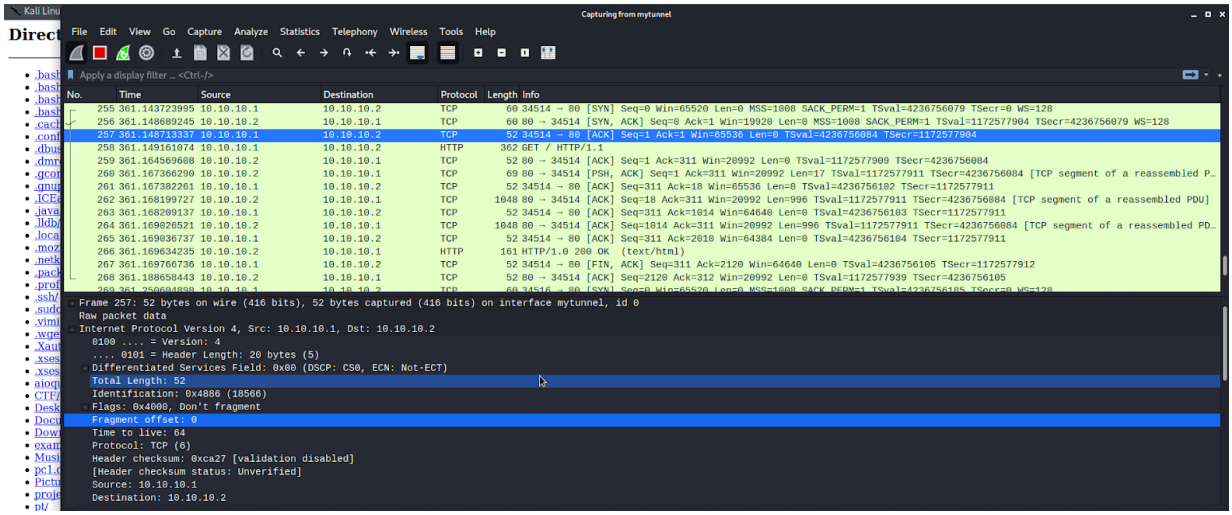


Figure 2.21: Client-side wireshark capture

2.3.3.3 ssh

Another test with ssh works:

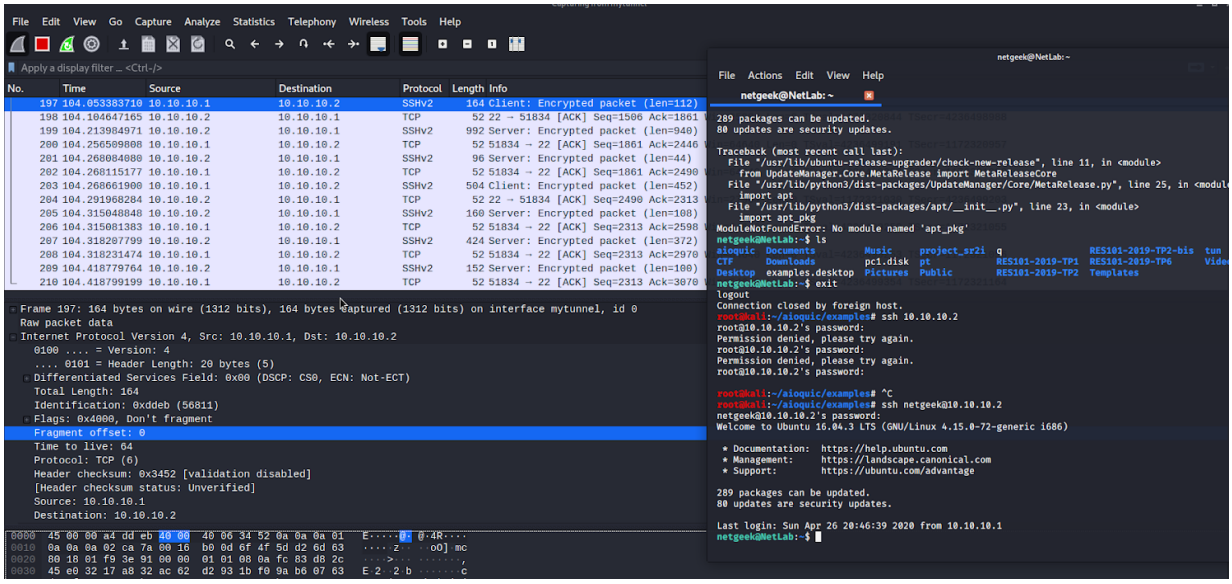


Figure 2.22: test with SSH

2.3.3.4 telnet

ssh is encrypted unlike telnet which sends traffic in clear. However, with the QUIC VPN, the traffic is encrypted no matter what the application! We can therefore use telnet to limit the use of the network and especially to have a faster remote connection:

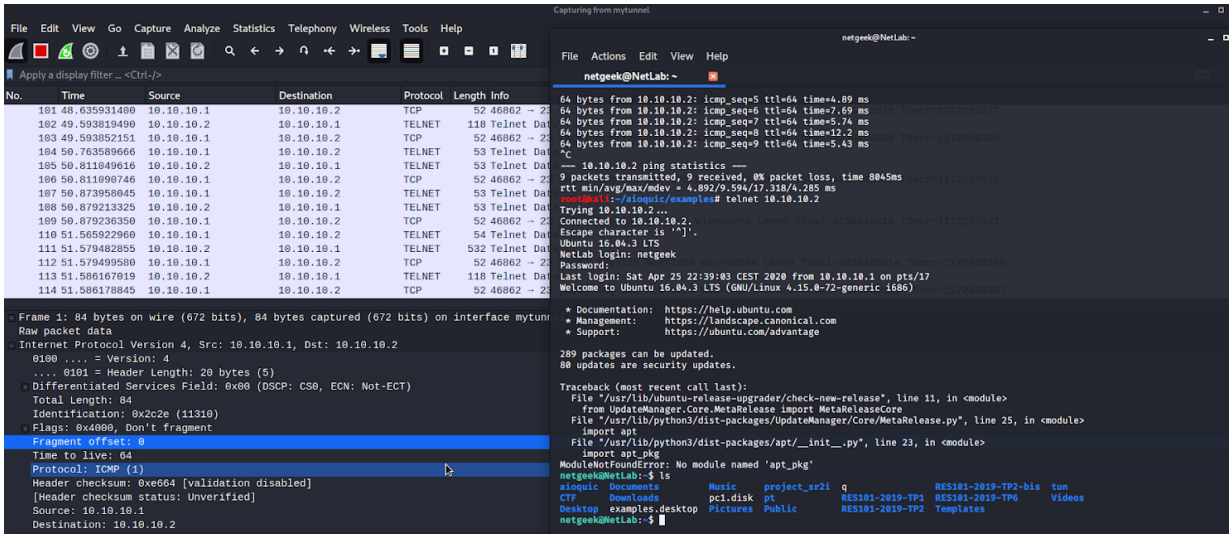


Figure 2.23: Test with Telnet

We must now check something important: verify that the traffic circulating in the network is encrypted by QUIC. To do this: let's listen on the eth0 port of the client. Indeed, the traffic is about to be sent via mytunnel. This traffic is intercepted by the VPN which will then send it via eth0. On the server side, it receives the QUIC stream on eth0, decapsulates it, then writes it on the server virtual interface.

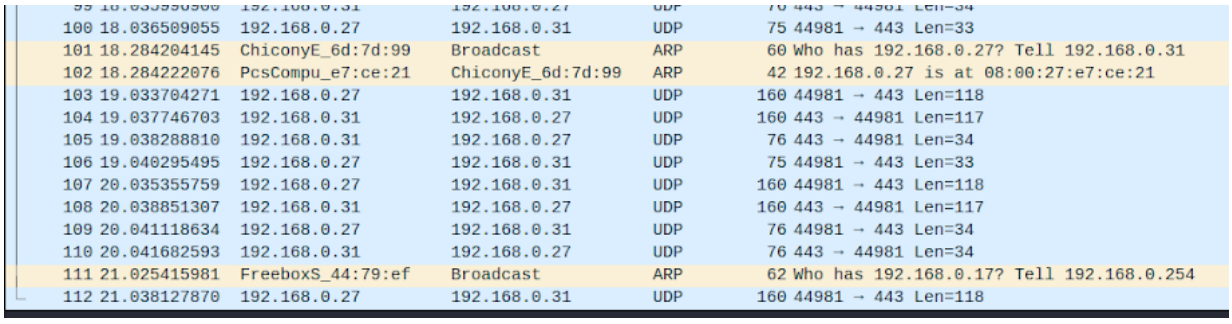


Figure 2.24: Circulating QUIC traffic (UDP detected)

We can see that the traffic intercepted in eth0 is interpreted as UDP, which is not shocking because QUIC uses UDP as a medium to send its payload. Still in Wireshark, let's go to Edit>Preferences>Protocols>QUIC and set the port to 443. We return to the capture, and now we see:

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)

[1cm]

Figure 2.25: Circulating QUIC traffic (detected as QUIC)

The traffic was indeed detected as QUIC and with the information "Protected Payload" indicating that the content is encrypted.

Let's compare the latency between without VPN and with VPN. Without VPN, we have the following ping:

```
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 without VPN

And with the VPN, we have the following ping:

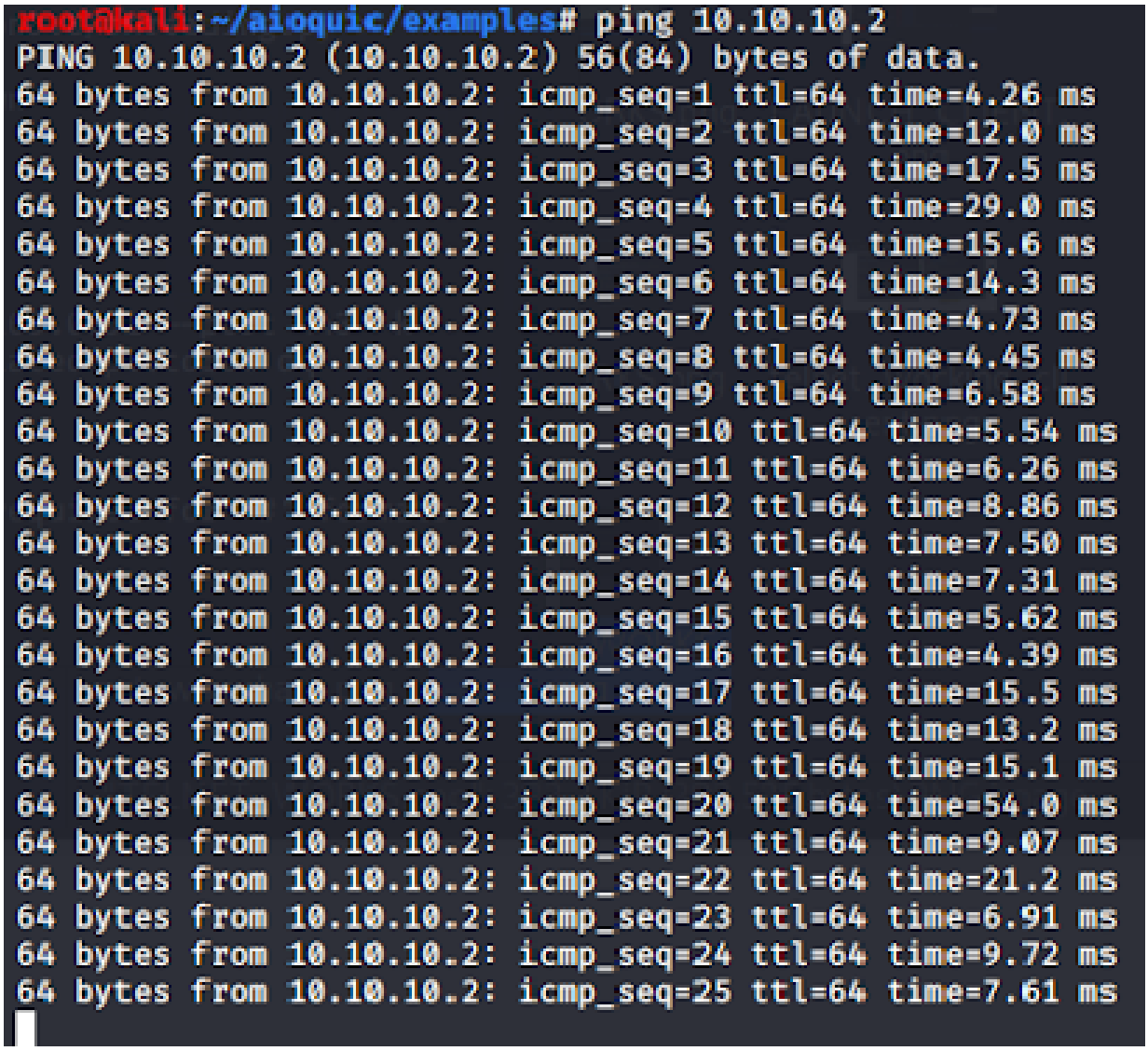


Figure 2.27: Ping aioquic with VPN

In addition to the moments when we have peaks due to the variable use of the network, we see that at least, the ping takes 3ms without VPN and minimum 4ms with VPN. It is normal that VPN has a slightly higher latency because it has to read, encapsulate, decapsulate, write. However, the performance is still very satisfactory. The ssh and telnet are very smooth when used in particular.

2.4 Proposed extensions for QUIC

QUIC is a protocol that was designed to accelerate web traffic. It is thus normal that this one does not force the authentication of the client: everyone has the right to connect to google.com, one does not need a certificate x509. On the other hand, the server must be certified because we want to be sure to be connected to the real google.com. However, for VPNs, the context is different: the client must be authenticated because after the VPN connection is established, the client can access the company's machines through the tunnel. So it is absolutely essential to authenticate the client. The question now is: how to authenticate the client? And more importantly, how to implement it in QUIC?

2.4.1 Public key authentication

For this, we take our inspiration from SSH authentication, which works as follows:

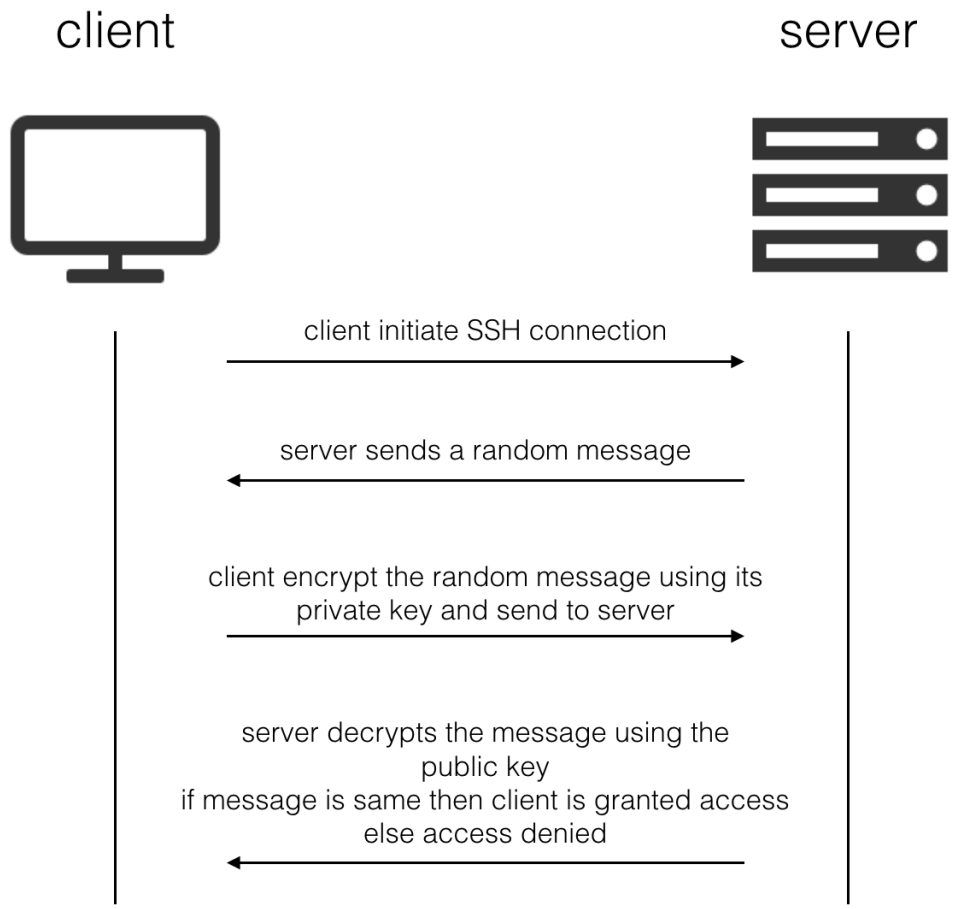


Figure 2.28: handshake SSH

In the case of QUIC, we could perform an authentication in the following way:

The server generates a pseudo random challenge message and sends it to the client. The client generates a pseudo random message 'nonce', and sends to the server: PBc, nonce, $S_{pkc}(challenge||nonce)$ PBc: client's public key, S_{pkc}: sign with client's private key pkc.

The server can then verify the signature with the public key. If this public key is accepted by the server, it authorizes the connection. Otherwise it refuses it. This can be very well integrated in the QUIC full handshake. During the hello server, it gives the challenge to the client. When the client answers, the full client hello, it indicates the message PBc, nonce, $S_{pkc}(challenge||nonce)$ to authenticate itself.

2.4.2 STK Authentication

During the first handshake, the server assigns the client an STK which allows to make sure that the client has the IP address he claims to have. Indeed, this STK contains the client's IP and a timestamp from the server encrypted by AES-GCM. The client will then have to continue to send this STK during client hello. However, this STK is vulnerable because it can be intercepted when the server assigns it to the client. An attacker can then replay it. In order to use this STK as a cookie, it must be protected from replay attacks. To do this, the client can add a client timestamp to the STK so that the server can detect if the time difference between this timestamp at the time of transmission and the time of arrival is small enough or not: if it is too large, it may be a replay. Once these modifications are made, the server can then use the STK to authenticate the client.

To prevent replay, we propose another solution than the timestamp :

-The client generates a PUBC/PKC key pair

-The client generates a pseudo-random nonce

-The client sends to the server: STK, PUBC,nonce,$S_{PKC}(STK, nonce)$

Server side:

- The server checks the signature

- The server checks if the nonce has not already been used

- The server checks the STK to verify that the source IP address matches the STK

- The server then authenticates the client with the STK

The signature allows integrity, the nonce allows non-replay, the STK allows identification. The whole forms the authentication. This system could be included in the full client hello of QUIC.

2.4.3 Authentication by x509

Although QUIC is web oriented, our design of a VPN program showed that it would be essential to have a possibility to authenticate a client via an x509 certificate. Indeed, this would be quite relevant in a site to site VPN connection: the 2 servers should authenticate each other via certificates. We could then add in the first hello client an optional field allowing to integrate the hash of the client certificate in FNV-1a format. The server can then, upon reception, verify the client certificate and identify it.