Tampereen yliopisto

Linnea Viitanen

SECURITY PROPERTIES OF HTTP/3

Bachelor’s thesis
Faculty of Information Technology and Communication Sciences
Examiner: Matti Monnonen
May 2022

ABSTRACT

Linnea Viitanen: Security properties of HTTP/3
Bachelor's thesis
Tampere University
Bachelor's Programme in Information Technology
May 2022

As cyber-attacks become more prevalent, the importance of security in protocols today is much more prominent than ever before. As the Internet of Things brings society even closer to a more virtualized world, security must be the priority of today's protocols. Vulnerabilities and privacy-intrusive features in the protocols put their users at risk from hackers. HTTP/3 is an ambitious project that strives to achieve both strong security features and a solid framework for fast connection speed. Inevitably, the question arises, is it capable of achieving both?

The topic of this bachelor's thesis is to study how security has been invested in the developing HTTP/3 protocol, and how its defensive capabilities add up to the whole. By comparing it to its predecessor, HTTP/2, the work seeks to give its reader an overview of how the security of HTTP/3 is built.

The work is divided into two parts: theory and practical experiment. In theory, the security features of HTTP/3 are largely determined by following the IETF (Internet Engineering Task Force) draft standards (RFC). Such drafts are documents published by the IETF, in which the current technical characteristics and descriptions of the protocol are reported during its development.

According to the study, the biggest difference between the protocols brings the transport layer protocol change in HTTP/3. The application layer protocol builds on the QUIC protocol originally designed by Google. QUIC enables encryption of the connection at each step, thus minimizing the risk of side channel attacks. This switch also allows for the speed promised by the protocol, as it finally frees HTTP from the HOL shackles caused by TCP, which grants it the freedom to multiplex its connections.

The practical part of the work studied the data leakage caused by the side channels. HTTP/3 traffic was captured using Wireshark and compared to the corresponding HTTP/2 traffic. Based on the results, the practice supports the theory: HTTP/3 encryption has been built to withstand more than that of HTTP/2, and therefore does not allow a beneficial traffic analysis.

Keywords: HTTP/3, HTTP/2, QUIC, TCP, protocol comparison, side channel attack

TIIVISTELMÄ

Linnea Viitanen: HTTP/3:n turvaominaisuudet
Kandidaatintyö
Tampereen Yliopisto
Tietotekniikan tutkinto-ohjelma
Toukokuu 2022

Kyberhyökkäysten yleistyessä turvallisuuden merkitys korostuu protokollissa nykypäivänä huomattavasti enemmän mitä aikaisemmin. Esineiden internetin tuodessa yhteiskuntaa yhä lähemmäs virtualisoitua maailmaa tulee turvallisuuden olla nykypäivän protokollien ensimmäinen prioriteetti. Haavoittuvuudet sekä yksityisyyttä laiminlyövät ominaisuudet protokollissa asettavat niiden käyttäjät riskialttiiksi hakkereille. HTTP/3 on ominaisuuksiltaan kunnianhimoinen kokonaisuus, joka pyrkii saavuttamaan sekä vahvat turvaominaisuudet että parhaimmat lähtökohdat yhteyden nopeudelle. Välttämättäkin nousee esille kysymys: pystyykö se molempiin?

Tämän kandidaatintyön aiheena on tutkia, kuinka turvallisuuteen on panostettu kehitteillä olevassa HTTP/3-protokollassa ja miten sen puolustuskykyä on parannettu sitten HTTP/2:sen. Vertailemalla protokollia keskenään työ pyrkii luomaan lukijalleen kokonaiskuvan siitä, kuinka HTTP/3:sen turvallisuus rakentuu.

Työ jakaantuu kahteen osaan: teoriaan ja käytännön kokeeseen. Teoriaosuudessa HTTP/3:sen turvalliset ominaisuudet selviävät pitkälti IETF (Internet Engineering Task Force) standardiluonnoksia (eng. RFC) seuraamalla. Nämä luonnokset ovat IETF:n julkaisemia asiakirjoja, joihin protokollien ajankohtaiset tekniset ominaisuudet sekä kuvaukset raportoidaan niiden kehityksen aikana.

Tutkimuksen perusteella protokollien välille tuo eniten eroavaisuutta kuljetuskerroksen protokollan vaihdos HTTP/3:ssa. Uusi protokolla rakentuu Googlen alun perin suunnitteleman QUIC protokollan päälle. QUIC mahdollistaa yhteyden salauksen sen jokaisessa vaiheessa, ja täten minimoi HTTP/2:sen sivukanavahyökkäysten riskin pois. Tämä vaihdos mahdollistaa myös protokollan lupaaman nopeuden, sillä sen avulla HTTP vapautuu vihdoinkin TCP:n aiheuttamista HOL-kahleista, ja pystyy vapaasti multipleksaamaan yhteyksiään.

Työn käytännönosuudessa tutkittiin sivukanavien aiheuttamaa tietovuotoa. HTTP/3 liikennettä kuunneltiin Wiresharkin avulla, ja vertailtiin vastaavaan HTTP/2 liikenteeseen. Tulosten perusteella käytäntö tukee teoriaa: HTTP/3:n salaus on vahvempi kuin HTTP/2:n, eikä se mahdollista hyödyttävää verkkoliikenneanalyysiä.

Avainsanat: HTTP/3, HTTP/2, QUIC, TCP, protokollavertailu, sivukanavahyökkäys

LIST OF SYMBOLS AND ABBREVIATIONS

TCP	Transmission control protocol
HOL	Head of line (blocking)
UDP	User datagram protocol
QUIC	Quick UDP Internet Connections
HTTP	Hypertext transfer protocol
HTTPS	Hypertext transfer protocol secure
HTML	Hypertext markup language
TLS	Transport layer security
SSL	Secure sockets layer
DoS	Denial of service attack
DDoS	Distributed denial of service attack
IETF	Internet engineering taskforce

CONTENTS

1. INTRODUCTION

Throughout its development, Hypertext Transfer Protocol, or HTTP, has been used to establish the web we know today. It is a request-response protocol necessary for clients and servers, typically browsers and webpages, to be able to communicate with each other. Initially, it was a simple protocol with no security features, but as time went on and the web developed, the lack of security became a problem. Before moving to HTTP/2, the protocol relied on an additional cryptographic layer for security, forming HTTPS. This worked quite well, despite its security flaws, until web traffic and content size started growing. HTTP/2 was developed as a binary protocol in response to this, as well as to further enforce the payload security. Even with proper encryption, the transferred packets would bleed sensitive information through side channels. To this point, the protocol had used TCP for payload transportation, but it turned out to be a limiting factor for the protocols multiplexing features, resulting in non-ideal connection speeds. These security flaws combined with the increasing demand for capacity in network connections inspired the initial idea of HTTP/3. The protocol would utilize a new transport layer protocol, QUIC, which would not only give the protocol a robust encryption, but also enable proper multiplexing.

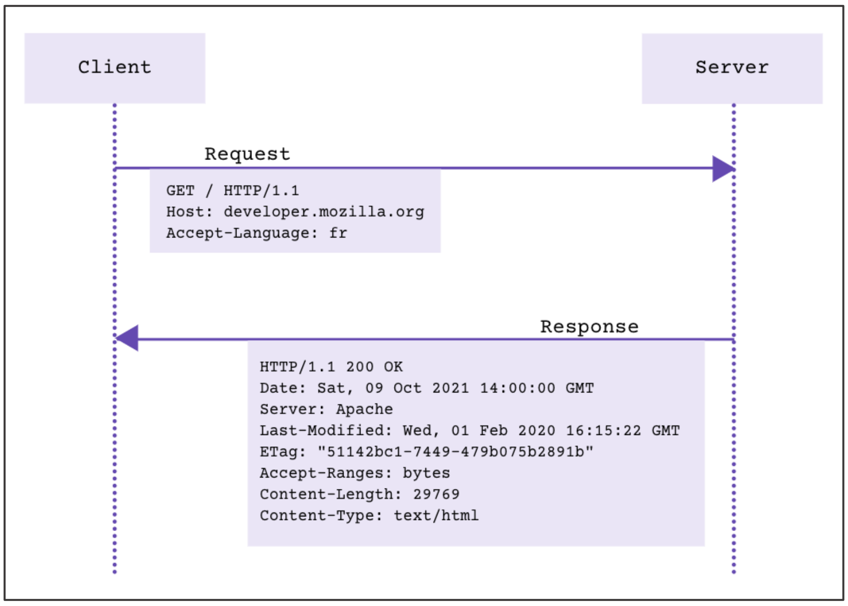
The objective of this thesis is to generally address the security features of HTTP/3, without providing a deeper analysis. The thesis observes how HTTP properties have been implemented in this version, and what has been added to the security side since HTTP/2. Chapter 2 lays the groundwork for understanding the differences HTTP/2 security and the basics of the transportation layer protocol QUIC, and chapter 3 addresses them. Chapter 4 goes into a practical analysis of the visible differences, by dissecting both protocols in Wireshark. The analysis is done in a laboratory client-server environment, where dissecting and analyzing the network traffic is transparent.

2. BACKGROUND

Before going into the secure properties of HTTP/3, it is important to comprehend the building blocks of HTTP-security. This chapter will go through the basic properties of HTTP, as well as the concept of HTTP/2. The security and vulnerabilities of the latter are discussed in the end of the chapter. Since QUIC is a major part of HTTP/3, the chapter will also explain its concept without going into many technical details.

2.1 Hypertext Transfer Protocol

Hypertext Transfer Protocol, or HTTP, is an application layer protocol, developed for transferring messages between two machines over a transport layer protocol. The protocol is based on request-respond operations, which allow back and forth hypertext communication between clients and servers. A client sends a request to a web server, and the server sends a response back. The following figure 1 shows an HTTP/1.1-request to an Apache server.



```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2021 14:00:00 GMT
Server: Apache
Last-Modified: Wed, 01 Feb 2020 16:15:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

Figure 1: A basic request-response operation in HTTP/1.1. The message body has been left out.

A client request is a message sent to the server, and its type is specified with an HTTP method. These methods are labeled as GET, POST and PUT. Using these labels determine, how the request will be processed. GET request will return a resource, POST transfers resource to the server and PUT generally replaces a resource. Server response, correspondingly, is a message sent from server to the client as a reply to the request they have sent. The response tells the client how the request has been handled by using HTTP status codes. The most common status code is 200, which means that the request succeeded. [1]

Although the protocol is commonly associated with HTML, it does not mean that the transferred content is only hypertext. In fact, the protocol can transfer all kinds of data, e.g., images, videos, or audio files. The data type is defined in each response header with the tag “content-type”. [2] In figure 1, the type has been assigned “text/html”.

HTTP is also responsible for transferring cookies to the browser during a client-server connection. Cookies are small fragments of user data, that are stored in the browser for connection state management. They can be used for tracking, identifying, and saving user data when visiting a website.

Another feature of HTTP are response caches, which aim to speed up the connection. Caches are storages filled with previously requested content and they have a goal of reducing network roundtrips during connections. Connections may exploit them either fully or partly. Loading content from cache does not require network connection. They contain only static components of a website, like index page content: text and images. Caches are stored same way as cookies: in the web browser. Both cache and cookies can be deleted by the browser client, when necessary.

2.1.1 HTTPS

In the earlier versions of HTTP, the transferred payload was a series of plain hypertext packets with no encryption in place. While this made it a human-friendly protocol to interpret, it also left the protocol vulnerable to different cyber threats [2]. This was later solved by adding a cryptographic layer TLS on top of TCP, creating a secure version of HTTP, called Hyper Text Transfer Protocol Secure, or HTTPS for short. It is not another version of HTTP, but rather a protocol stack, which encrypt the connection requests and responses. The transferred data is encrypted bi-directionally by using asymmetric encryption with the use of public-private keypairs. Encryption makes the communication impossible to decipher without the proper keys. [3] This allows the client-server communication to be authenticated and properly secured, and it prevents or mitigates most of the vulnerabilities HTTP has. In the past, HTTPS was mainly used to secure confidentiality in transactions like banking, but today, as it is compatible with all versions of HTTP, it has become a mandatory practice for reliable connection in both HTTP/1.1 and HTTP/2.

Although it makes things far more secure, it does not participate in securing the connection side channels. This means that information like transferred data sizes, or clients time spent on a website can be interpreted from the traffic [3], creating an exposure to a side-channel attack. Side-channel attacks are basically attacks based on the information patterns that a protocol may use. Chen et al. studied in their paper [4] the severity of side-channel leaks. Their study found multiple different applications that were leaking exploitable private data despite using a HTTPS encryption. This data included health information, search queries, family income and investment secrets.

2.1.2 HTTP/2

As the websites developed into dynamic, complex structures, HTTP/1.1 became unsuitable for content-heavy websites that were still using its single response/request connection. HTTP/2 was developed as a binary protocol [5], and it introduced two major features: multiplexing and header compression. Multiplexing is a mechanism, that enables concurrent requests in the same connection, and it was used to counter the slowing connection speed of HTTP/1.1. By using multiplexing, the session establishment including a cryptographic handshake became faster, because more phases could be transferred in a single stream, as seen in figure 2.

The header compression was managed with a unique feature of HTTP/2 called HPACK [6]. It is an algorithm built to help the protocol reduce the size of the transmitted data. HPACK does this by compressing the otherwise repetitive headers of HTTP/2 messages. HTTP had previously allowed only the message body to be compressed, but since the traffic has increased, the uncompressed headers would affect the transfer speed. This can be seen in content-heavy websites, that need to send more requests to load the page.

In HTTP/2, multiplexing grants the freedom of out-of-order packet receiving, which means that the protocol allows the client to send multiple requests in a single connection without needing to wait for the responses. The effect on the speed was tested in a realistic network environment by Griffin, who concluded that multiplexing feature increases the protocol speed by 14% compared to HTTP/1.1 [7].

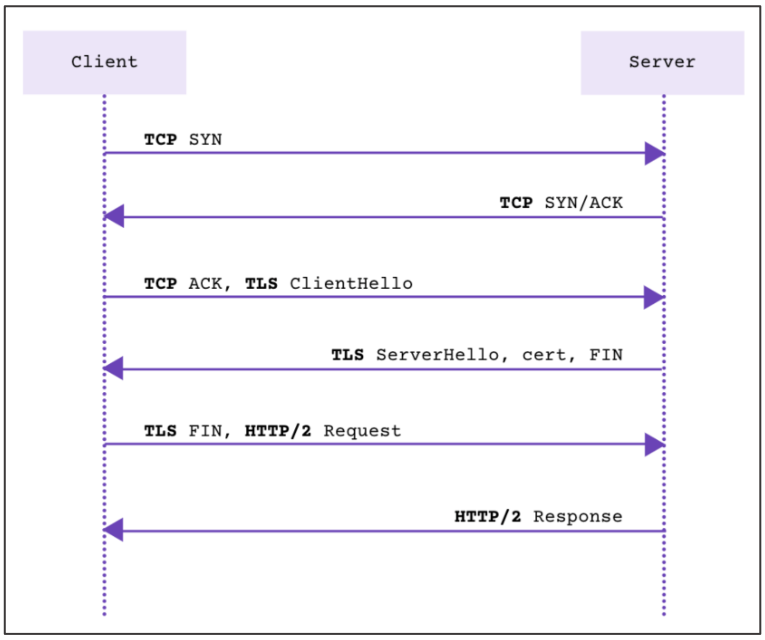


Figure 2: HTTP/2 session establishment

A lot of these listed properties create a good foundation for the protocol and makes it still relevant today. The connection speed, however, will become a problem for it in the future. Since its transportation level protocol TCP suffers from head-of-line blocking, multiplexing cannot be utilized ideally. This drawback comes up, when there are multiple connections for TCP to handle: one lost or out-of-order packet will stall the protocol, until the packet has been retransmitted successfully. TCP cannot distinguish multiple streams and crams them all into one. Since this problem remains outside of the structure of

HTTP/2, it cannot be solved without making modifications to the transport layer protocol.

The security aspect of HTTP/2 was a huge improvement from the previous version, and it still holds up. It was achieved mainly by forcing HTTP/2 implementations to use TLS version 1.2 or higher [5]. This denied the usage of TLS 1.0 and 1.1, which had both suffered from vulnerabilities such as Heartbleed and BEAST [8]. Additionally, the binary form creates another security layer to the protocol. The use of binary reduces errors and makes the protocol more complex.
What is to consider, though, is that not all websites communicate with HTTP/2. This leads to a downgrade vulnerability, where a front-end server uses HTTP/2 with the clients but rewrites the requests to back-end using HTTP/1.1 [5]. Interestingly, this can be exploited in numerous ways, as James Kettle has done in his research. Kettle describes closed bug-bounty case studies, in which he exploits this vulnerability and gains access to different websites. One of the studies includes him inserting harmless JavaScript code into the requests. By replacing the code with a malicious script, an attacker could gain an access to user accounts. Another study involves a case, where he includes a redirection prefix to the request, which would lead the victim to a malicious version of the website. All the described cases can be redone by using the tools from Burp Suite, which has implemented native support for manipulating HTTP/2 requests from versions 2021.8 upwards. [9]

2.2 QUIC

QUIC stands for quick UDP internet connections, and it is a standalone secure transport protocol first introduced by Google in 2012. After years in development, it was officially standardized in May 2021 by the IETF. The protocol aims to provide security to transport connections by increasing network traffic performance. It serves as an encrypted tunnel for different application layer protocols, such as HTTP/3. Features that originally make TCP more reliable than UDP, for example loss recovery and congestion control, have been migrated over to QUIC. It has also been developed with security in mind, namely encrypting even the side-channel data. QUIC has migrated features from TLS, so it has also an extra layer of encryption. [10] This encryption allows QUIC packets to be encrypted at every stage of a connection. The comparison of QUICs role in HTTP/3 stack to HTTP/2 can be seen in figure 3.

QUIC has increased network traffic performance due to its UDP features, and it provides considerably reduced latency compared to TCP [11]. This is achieved by the connection establishment mechanism of QUIC, as well as the obvious absence of head-of-line blocking. By using features from UDP, the protocol eliminates the issue of head-of-line blocking that is slowing TCP down. This allows it to establish several multiplexed connections between two endpoints without high chance of interruption. The side of UDP allows uncontrolled dataflow, with no latency coming from retransmissions or packet loss recovery. Any retransmissions are done on the level of QUIC, which allows the UDP streams to keep going while a single multiplexed stream is being repaired. Each stream is delivered independently, so that packet loss in one case will not affect others. This also allows QUIC to be flexible regarding the network environment. A feature called connection migration allows a QUIC connection to adapt to the change in endpoint addresses, for example when switching from cellular data to Wi-Fi, without breaking. Each of the connections have a specific ID, which is carried by the QUIC packets, to prevent any collisions between migrating sessions. [12]

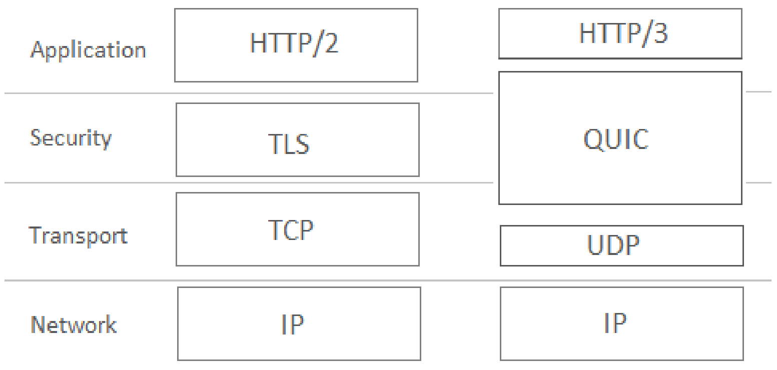


Figure 3: The role of QUIC in the stack

The connection establishment mechanism in QUIC incorporates the handshake of TLS 1.3 [13], by merging it with the transport handshake and replacing the TLS record layer with QUICs own framing format. This version of TLS is currently the newest version of the protocol with added features and has patched all the vulnerable ciphers and algorithm from past versions. With the key-exchange and selection of supported protocols embedded into the initial handshake, QUIC is seemingly twice more efficient in network roundtrips compared to the regular TCP/TLS stack. Typical TCP/TLS round-trip time, or RTT, of a session establishment is 300 milliseconds, whereas QUIC evidently achieves the same in 100 milliseconds [14].

QUIC has an additional feature for re-establishing a connection labeled 0-RTT. It minimizes the round-trip time to zero, which allows a re-established connection to pick up where it left off, without the need of redoing the handshakes. This allows the connection to resume sending encrypted application data right away. If the feature is enabled, QUIC will cache the parameters from the client-server negotiation and use them in a new session. [12]

3. SECURITY PROPERTIES OF HTTP/3

This chapter will go through the added security properties of HTTP/3. The text contains references to cryptography and transport layer security TLS, so the reader is advised to have a prior understanding of these topics. The chapter breaks down the session establishment of HTTP/3 and observes each part of it from the security aspect, as well as analyzes how the protocol has been modified from its prior version, HTTP/2. Additionally, the chapter goes through the security of QUIC from point of HTTP/3, and the defensive capabilities of them as a protocol stack.

3.1 HTTP properties implemented in HTTP/3

HTTP/3 is the upcoming version of HTTP and at the time of writing it remains an RFC draft. Originally, it was intended to be an extension of HTTP/2, but the use of TCP would not enable the wanted features. Therefore HTTP/3 properties are very similar to its previous version, with the biggest difference being the replacement of the TCP with the new transport protocol QUIC. As discussed in chapter 2.2, the change speeds up the client-server session by eliminating the head-of-line blocking of TCP. This means that using QUIC, HTTP/3 is fully capable of multiplexing its streams.

Since the protocol is heavily tied to the transportation layer protocol, its security also relies strongly on it. In HTTP/3, some of the previously HTTP responsible features have been delegated to QUIC. For example, while the header compression works similarly to the HPACKs in HTTP/2, it has now been readdressed to QUIC. This reworked version is not much different to the previous one other than it is now called QPACK. The reason for the rework was the compatibility issues: HPACK was built with TCP’s total ordering of the packets in mind, whereas QUIC being a protocol built on UDP enables out-of-order packet transfer. Aside from this, QPACK reuses much of the features of HPACK.

Some of the most basic HTTP protocol properties were discussed in chapter 2.1, like caches and cookies. In HTTP/3, caches and storing user data are implemented much like in the other versions. However, cacheability of pushed responses have been discussed in the draft section 10.4 under security considerations. This section briefly considers the situation, that a server has more than one tenant, in which case the server provider must be cautious about the authority of the tenants. The failure of proper authentication on the server could lead to the tenants requesting caches they would not have permission to. Similarly, using cookies has not changed. Only their transferring has been tweaked along with the hop from HPACK to QPACK. In QPACK, the cookie-field in HTTP messages may be split into separate lines for better compression efficiency, as described in the draft. [15]

3.2 HTTP/3 session establishment and lifecycle

When opening a new connection, the session establishment in HTTP/3 begins with QUIC initializing and encrypting the traffic, as seen in figure 4. The encryption happens by doing a cryptographic handshake, during which the use of HTTP/3 is indicated by selecting an Application-Layer Protocol Negotiation, or ALPN, token “h3”. The handshake lifecycle is the same as with TLS 1.3, where both client and server exchange their certificates and authenticate one another. The handshake has been built within QUIC, so the connection will always be secure with no possible configuration mishaps.

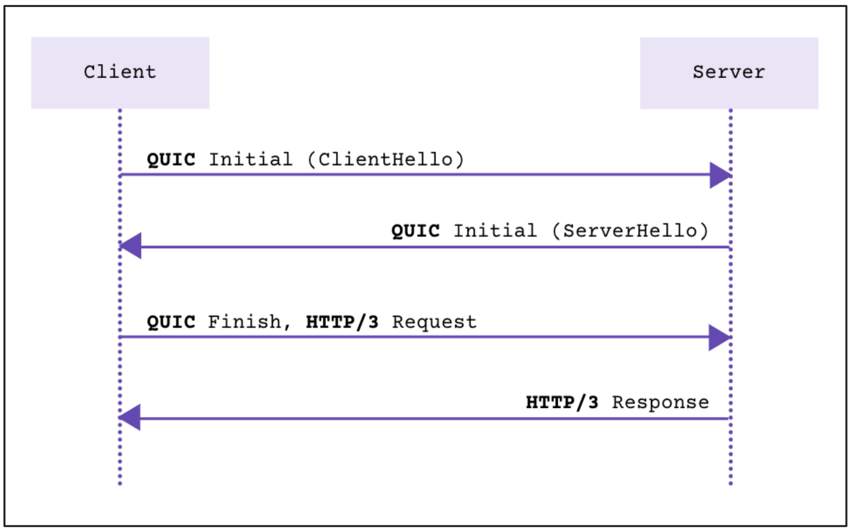

Client
Server
QUIC Initial (ClientHello)
QUIC Initial (ServerHello)
QUIC Finish, HTTP/3 Request
HTTP/3 Response

Figure 4: Using QUIC to establish 1-RTT connection

Each HTTP/3 packet, or stream, is built from HTTP framing layers. The protocol has three different stream types: control, request, and push stream. The frames of a packet can consist of DATA, HEADERS, CANCEL_PUSH, SETTINGS, PUSH_PROMISE, GOAWAY, MAX_PUSH_ID or Reserved. Once setting up the QUIC connection has finished, the first HTTP/3 message will send a SETTINGS frame with it, which contains the details regarding how the endpoints of the HTTP/3 session will communicate. The chosen settings will be used for the rest of the connection. After the control stream has been received, the application data stream will open, and remains so until one of the endpoints close the connection by either an idle timeout or a graceful shutdown.

Establishing a new connection, as described above, goes by the name of "1-RTT", which illustrates the time spent on the cryptographic handshake. When re-establishing a connection, HTTP/3 can use the "0-RTT" feature of QUIC, that allows it to restore the previous connection state. In practice, this means that on client-side, there is no need to go through the handshake again. QUIC needs to send only an initial message, and HTTP/3 application data stream will open. The first incoming HTTP/3 message will contain the same SETTINGS frame that was used in the previous connection. When resuming the connection using 0-RTT, the parameters from the previous connections come from the QUIC server's cache. The server is responsible of storing the parameters for a certain period, before terminating the connection with idle timeout.

The connection shutdown happens, when one of the endpoints send a GOAWAY frame within a control stream. If at any point of the connection an erroneous frame is sent, a corresponding error code will be sent in a control stream, and the connection will be terminated. [15], [12]

3.3 Connection security with QUIC

This subsection goes through the very basics, only scratching the cryptographic side of QUIC. To support end-to-end encryption in a HTTP/3 connection, most of the cryptographic primitives QUIC uses come specifically from TLS v1.3 [16]. QUIC relies on these v1.3 features for authentication, security parameter negotiation, forward secrecy, and many other security properties. The TLS version is non-negotiable in HTTP/3, meaning that TLS downgrading and exposing the protocol to a vulnerable version is prevented [17].

QUIC intends to achieve a complete security for the transferred payload and implements several encryption keys to do so: the initial keys, early data keys, handshake keys, application data keys and packet protection keys. In session establishment, a client and server both send each other an initial packet, which contain the cryptographic messages. Initial keys, which are connection- and version specific, are used to encrypt these packets. After receiving the initial packets, both parties must discard the keys, to prevent an attacker from spoofing an initial packet. During the handshake, the parties agree on a set of handshake keys, that the connection will use to complete the cryptographic handshake. These will be discarded as well when the handshake is complete. After a handshake is ready, application data keys are used to encrypt and send a 1-RTT packet, which finalizes the connection establishment. With the session ongoing, after encrypting a packet with TLS handshake, QUIC re-encrypts its payload and header separately using its packet-header protection keys [14]. The two parts will be combined once re-encrypted, forming the actual payload. This creates a heightened security, as it also secures the payload metadata. [18], [19]

The early data keys are used, when a 0-RTT connection is established, only to protect acknowledgements of 0-RTT packets. These keys, however, are not used to protect the packet content in anyway.

3.4 Defensive capabilities and protocol vulnerabilities

The transport protocol QUIC, which is heavily tied to HTTP/3, has both advantages and vulnerabilities deriving from TLS and UDP. TLS 1.3 is designed so that the handshake process generates a key hash specific for it, which both parties need to confirm for the connection to continue. If a connection detects a faulty hash, the handshake will terminate. In QUIC itself, the initial packet includes a CertificateVerify message, which itself contains another hash. To verify the TLS version, the hashes can be checksummed. With using the ALPN extension in TLS as discussed in chapter 3.2, HTTP/3 connection endpoints will always use the same protocol versions, protecting the connection from a cross-protocol attack [15].

UDP based protocols often suffer from the vulnerability to a reflection attack. In reflection attack, an attacker spoofs the packet source IP and sends a request to the receiver. If the receiver replies with a response, the attacker can use this behavior to cause a denial-of-service, making the service slow or completely inaccessible to the clients. QUIC, and therefore HTTP/3, is strong against this when using 1-RTT, but vulnerable when using 0-RTT. The initial handshake process includes an address validation by using a source-address token. It is an authentication key containing the IP address of the user-agent and a server timestamp. This key is read by the server each time, and without the token, the server will not respond. The source-address changes only when there are changes in connectivity. The time-window between the change is minimal, so it is practically impossible to do an IP spoofing in-between. [12]

In a 0-RTT attack scenario, the adversary sends a 0-RTT reply to the server using the IP address originally used to generate the 0-RTT token. If the server replies, it can be instructed to forward traffic to the victim, resulting in a DDoS. This vulnerability is addressed from both HTTP/3 and QUIC: HTTP/3 mitigates this attack by its packet rate limits and validation tokens, whereas QUIC assesses this vulnerability in three ways. Firstly, it has constrained the data packet sizes the server can send at the beginning of a session. Secondly, it allows the server to validate the client address by using the built-in anti-amplification mechanism. The mechanism verifies the client's capability of receiving packets from the address it has validated before. If an unknown address sends in a packet, the client must then send a data packet with a size exactly three-times the size of the received packet. Thirdly, a server can validate a client by sending a cryptographic token inside a packet, which the client then needs to echo back for the connection to resume. [16] By only mitigating the attack though, it cannot be completely brushed off: In a session, where 0-RTT feature is enabled, an attacker can still cause havoc by replaying a request repeatedly [20]. While the 0-RTT feature gives the protocol its fast connection resumption ability, it suffers from different vulnerabilities, which is why the feature is not currently enabled by default in web browsers.

Both HTTP/3 and QUIC have implemented a replay attack protection. In a replay attack, an adversary eavesdrops a request made by a client, and resends it to the server, causing the server to send the respond back to both the adversary and the client. HTTP/3 uses the anti-replay mitigations when 0-RTT is enabled, as described in [21]. From the QUIC side, prevention happens by discarding requests performed repeatedly using the same key. This does not apply to the connection’s initial steps though, and it leaves the protocol momentarily vulnerable.

Due to the novelty of the protocol, it is important to consider the outlook as well. Since the protocol uses UDP port 443, it will most likely have complications with network middleboxes, like firewalls, for a while. Firewalls are usually designed with TCP in mind and are configured to limit UDP port traffic, since those ports often get amplification attacks. Opening UDP ports will most likely cause a high rise in UDP port scanning in the future. This is a configuration issue that is beyond HTTP/3 or QUIC to handle. QUIC has only prepared for the future case of version-unaware middleboxes, which means that when middleboxes do know the QUIC protocol, they cannot be exploited into downgrade attacks since the initial keys are version- and connection specific.

4. TESTING AND RESULTS

This testing aims to analyze in practice, how HTTP/3 on QUIC compares to HTTP/2 on TCP, considering the packet information security. The tests will provide an understanding of how both protocols stand up against a side-channel analysis. This is achieved by analyzing both encrypted and decrypted traffic using Wireshark, the network traffic analyzing tool.

Overall, the chapter will go through the testbed and setup, the results evaluation, and the discussion of the connection between theory and the testing.

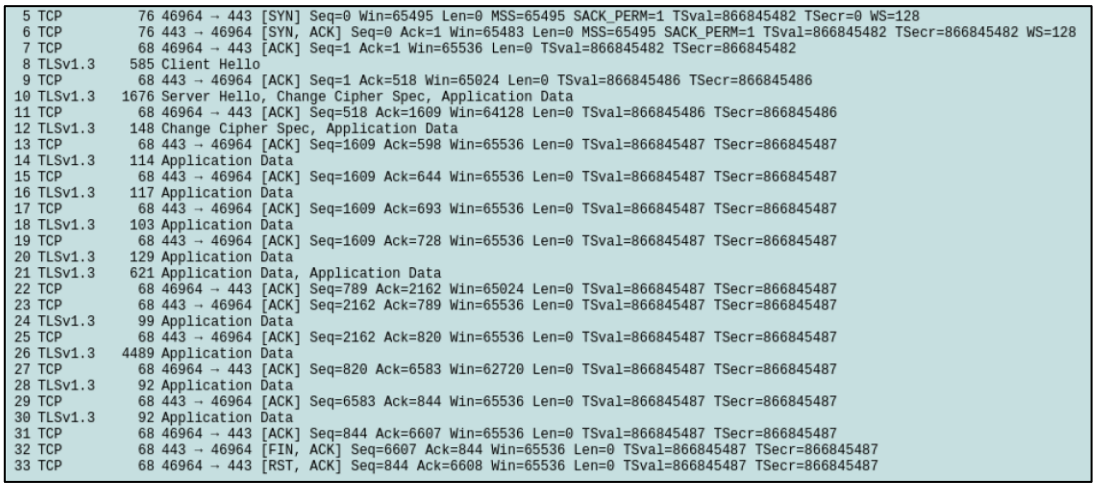
4.1 Testbed and setup

Protocol dissecting and packet analyzing was performed in a local test environment, where both client and the server were hosted on Linux (4.19.0-17-amd64 #1 SMP Debian 4.19.194-3 x86_64 Ubuntu 20.10) machines. The server was built by using Openlitespeed, with locally trusted certificates created by Mkcert [22]. The content of the server is limited, consisting of a title “HTTP/3 server” and a textbox of lorem-ipsum. When decrypting both protocols, these details will become visible in the progress.

The tests were performed by having the client send a request to the server by using a HTTP/2 and HTTP/3 compliant version of the command-line tool curl. Packet capturing was done by using a source-built Wireshark version 3.7, which supports the decryption of the QUIC protocol. Decryption requires the private key of the server, which can be obtained by telling a browser to log the keys to a file called SSLKEYLOGFILE. This file is given to Wireshark, which uses it to decrypt packets, which have been encrypted by TLS. [23]

4.2 HTTP/2 and TCP

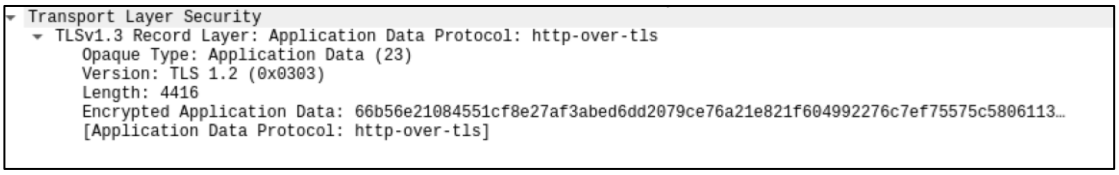
Capturing encrypted HTTP/2 data from the network will result what is displayed in figure 5. This is due to HTTP/2 encryption, as well as its compression. TCP handshake, as well as the TLS handshake are both encrypted but visible without further probing. The only thing that is not straight visible in the traffic is the encrypted HTTP/2 payload, which is being carried by TLS with the label "Application Data".



```
 5 TCP          76 46964 → 443 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=866845482 TSecr=0 WS=128
 6 TCP          76 443 → 46964 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=866845482 TSecr=866845482 WS=128
 7 TCP          68 46964 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=866845482 TSecr=866845482
 8 TLSv1.3     585 Client Hello
 9 TCP          68 443 → 46964 [ACK] Seq=1 Ack=518 Win=65024 Len=0 TSval=866845486 TSecr=866845486
10 TLSv1.3    1676 Server Hello, Change Cipher Spec, Application Data
11 TCP          68 46964 → 443 [ACK] Seq=518 Ack=1609 Win=64128 Len=0 TSval=866845486 TSecr=866845486
12 TLSv1.3     148 Change Cipher Spec, Application Data
13 TCP          68 443 → 46964 [ACK] Seq=1609 Ack=598 Win=65536 Len=0 TSval=866845487 TSecr=866845487
14 TLSv1.3     114 Application Data
15 TCP          68 443 → 46964 [ACK] Seq=1609 Ack=644 Win=65536 Len=0 TSval=866845487 TSecr=866845487
16 TLSv1.3     117 Application Data
17 TCP          68 443 → 46964 [ACK] Seq=1609 Ack=693 Win=65536 Len=0 TSval=866845487 TSecr=866845487
18 TLSv1.3     103 Application Data
19 TCP          68 443 → 46964 [ACK] Seq=1609 Ack=728 Win=65536 Len=0 TSval=866845487 TSecr=866845487
20 TLSv1.3     129 Application Data
21 TLSv1.3     621 Application Data, Application Data
22 TCP          68 46964 → 443 [ACK] Seq=789 Ack=2162 Win=65024 Len=0 TSval=866845487 TSecr=866845487
23 TCP          68 443 → 46964 [ACK] Seq=2162 Ack=789 Win=65536 Len=0 TSval=866845487 TSecr=866845487
24 TLSv1.3      99 Application Data
25 TCP          68 443 → 46964 [ACK] Seq=2162 Ack=820 Win=65536 Len=0 TSval=866845487 TSecr=866845487
26 TLSv1.3    4489 Application Data
27 TCP          68 46964 → 443 [ACK] Seq=820 Ack=6583 Win=62720 Len=0 TSval=866845487 TSecr=866845487
28 TLSv1.3      92 Application Data
29 TCP          68 443 → 46964 [ACK] Seq=6583 Ack=844 Win=65536 Len=0 TSval=866845487 TSecr=866845487
30 TLSv1.3      92 Application Data
31 TCP          68 46964 → 443 [ACK] Seq=844 Ack=6607 Win=65536 Len=0 TSval=866845487 TSecr=866845487
32 TCP          68 443 → 46964 [FIN, ACK] Seq=6607 Ack=844 Win=65536 Len=0 TSval=866845487 TSecr=866845487
33 TCP          68 46964 → 443 [RST, ACK] Seq=844 Ack=6608 Win=65536 Len=0 TSval=866845487 TSecr=866845487
```

Figure 5: HTTP/2 + TCP encrypted traffic

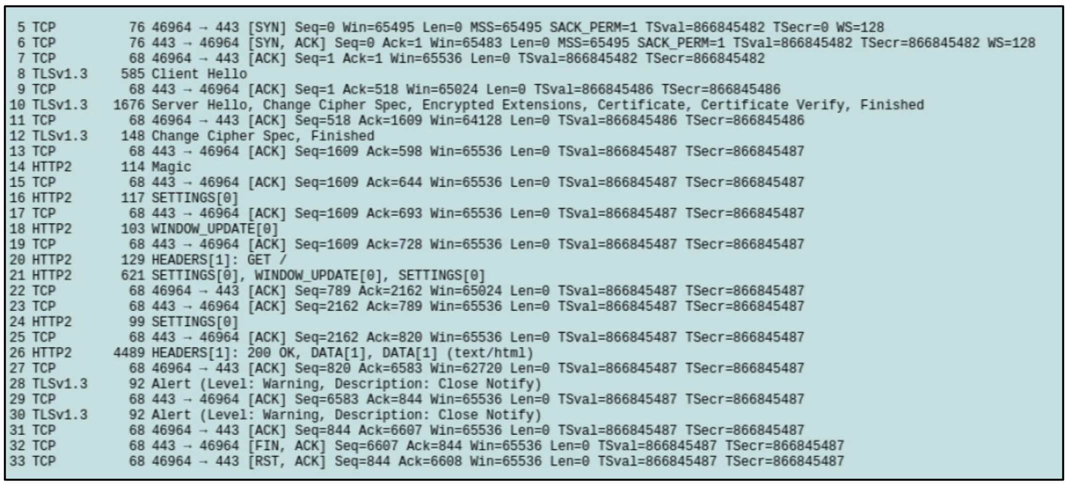
The displayed traffic content is human readable, in a sense. It shows when a connection is being initiated, ciphers negotiated, and the data stream opened. Furthermore, when looking into these data containing TLS packets, the application data can be seen in its encrypted format, but with a packet length visible. Figure 6 is going into packet number 26, which has the data size of 4416. This sizeable packet appears in the traffic amidst much smaller packets, which can tell a keen observer that it is the one containing the actual payload. The packet also tells that the protocol used is http-over-tls. These sorts of side-channel information leaks are also note-worthy when considering protocol security.



```
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 4416
      Encrypted Application Data: 66b56e21084551cf8e27af3abed6dd2079ce76a21e821f604992276c7ef75575c5806113…
      [Application Data Protocol: http-over-tls]
```

Figure 6: Encrypted packet information

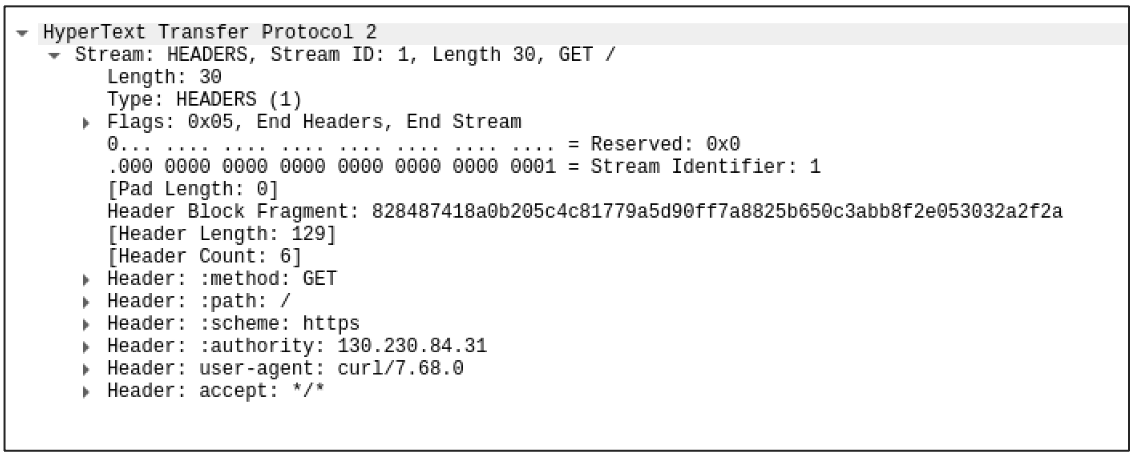
After decrypting and decompressing the traffic, the traffic feed becomes transparent. In figure 7 is displayed the exact same data stream from figure 5, but this time, the HTTP/2 protocol title has been exposed to the observer, and a lot of important headers are visible. The figure also shows a clear request-response architecture, with packet number 20 being GET, and packet number 26 the POST.



```
 5 TCP         76 46964 → 443 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=866845482 TSecr=0 WS=128
 6 TCP         76 443 → 46964 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=866845482 TSecr=866845482 WS=128
 7 TCP         68 46964 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=866845482 TSecr=866845482
 8 TLSv1.3    585 Client Hello
 9 TCP         68 443 → 46964 [ACK] Seq=1 Ack=518 Win=65024 Len=0 TSval=866845486 TSecr=866845486
10 TLSv1.3   1676 Server Hello, Change Cipher Spec, Encrypted Extensions, Certificate, Certificate Verify, Finished
11 TCP         68 46964 → 443 [ACK] Seq=518 Ack=1609 Win=64128 Len=0 TSval=866845486 TSecr=866845486
12 TLSv1.3    148 Change Cipher Spec, Finished
13 TCP         68 443 → 46964 [ACK] Seq=1609 Ack=598 Win=65536 Len=0 TSval=866845487 TSecr=866845487
14 HTTP2      114 Magic
15 TCP         68 443 → 46964 [ACK] Seq=1609 Ack=644 Win=65536 Len=0 TSval=866845487 TSecr=866845487
16 HTTP2      117 SETTINGS[0]
17 TCP         68 443 → 46964 [ACK] Seq=1609 Ack=693 Win=65536 Len=0 TSval=866845487 TSecr=866845487
18 HTTP2      103 WINDOW_UPDATE[0]
19 TCP         68 443 → 46964 [ACK] Seq=1609 Ack=728 Win=65536 Len=0 TSval=866845487 TSecr=866845487
20 HTTP2      129 HEADERS[1]: GET /
21 HTTP2      621 SETTINGS[0], WINDOW_UPDATE[0], SETTINGS[0]
22 TCP         68 46964 → 443 [ACK] Seq=789 Ack=2162 Win=65024 Len=0 TSval=866845487 TSecr=866845487
23 TCP         68 443 → 46964 [ACK] Seq=2162 Ack=789 Win=65536 Len=0 TSval=866845487 TSecr=866845487
24 HTTP2       99 SETTINGS[0]
25 TCP         68 443 → 46964 [ACK] Seq=2162 Ack=820 Win=65536 Len=0 TSval=866845487 TSecr=866845487
26 HTTP2     4489 HEADERS[1]: 200 OK, DATA[1], DATA[1] (text/html)
27 TCP         68 46964 → 443 [ACK] Seq=820 Ack=6583 Win=62720 Len=0 TSval=866845487 TSecr=866845487
28 TLSv1.3     92 Alert (Level: Warning, Description: Close Notify)
29 TCP         68 443 → 46964 [ACK] Seq=6583 Ack=844 Win=65536 Len=0 TSval=866845487 TSecr=866845487
30 TLSv1.3     92 Alert (Level: Warning, Description: Close Notify)
31 TCP         68 46964 → 443 [ACK] Seq=844 Ack=6607 Win=65536 Len=0 TSval=866845487 TSecr=866845487
32 TCP         68 443 → 46964 [FIN, ACK] Seq=6607 Ack=844 Win=65536 Len=0 TSval=866845487 TSecr=866845487
33 TCP         68 46964 → 443 [RST, ACK] Seq=844 Ack=6608 Win=65536 Len=0 TSval=866845487 TSecr=866845487
```

Figure 7: HTTP/2 + TCP decrypted traffic

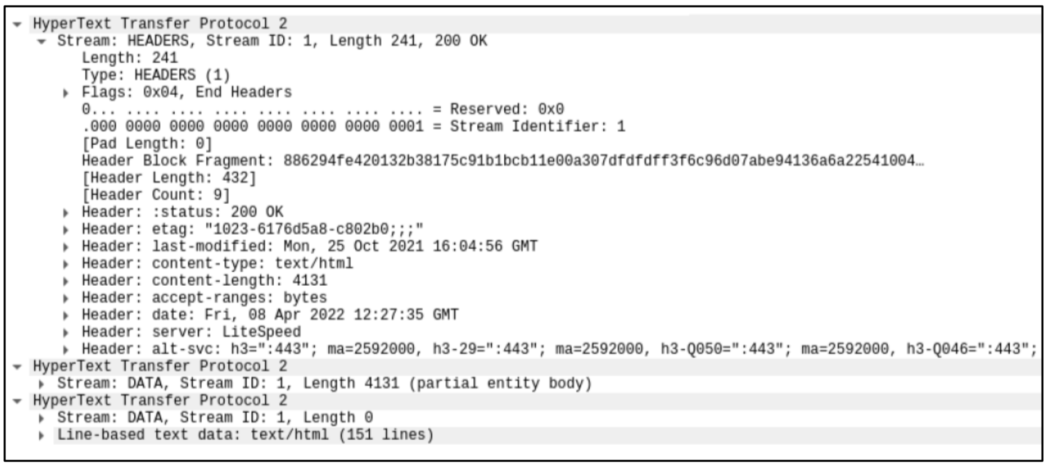
Figure 8 shows the decrypted packet number 20, the GET-request. Application data protocol has now been specified to HTTP/2 and shows its frame structure. The pseudo-headers of the packet include method, path, scheme, authority, and user-agent. Method being being obviously GET, the scheme part tells that it is using https. Authority part tells the target URI authority, which is, in this case, the test server IP address 130.230.84.31. Finally, the user-agent header



```
▾ HyperText Transfer Protocol 2
  ▾ Stream: HEADERS, Stream ID: 1, Length 30, GET /
        Length: 30
        Type: HEADERS (1)
      ▸ Flags: 0x05, End Headers, End Stream
        0... .... .... .... .... .... .... .... = Reserved: 0x0
        .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
        [Pad Length: 0]
        Header Block Fragment: 828487418a0b205c4c81779a5d90ff7a8825b650c3abb8f2e053032a2f2a
        [Header Length: 129]
        [Header Count: 6]
      ▸ Header: :method: GET
      ▸ Header: :path: /
      ▸ Header: :scheme: https
      ▸ Header: :authority: 130.230.84.31
      ▸ Header: user-agent: curl/7.68.0
      ▸ Header: accept: */*
```

Figure 8: GET-request sent to the server

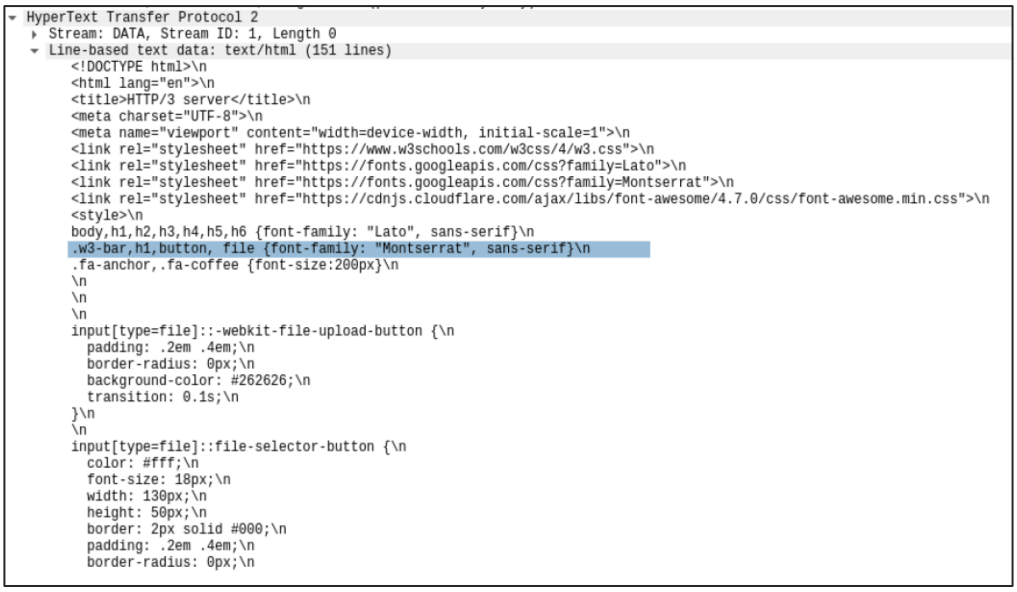
tells, that the client has used curl/7.68.0 to access this server. In figure 9, the same TLS packet that was in figure 6 has been decrypted and exposed as HTTP/2 packet now. This is the response from the server to the GET-request,



```
▾ HyperText Transfer Protocol 2
  ▾ Stream: HEADERS, Stream ID: 1, Length 241, 200 OK
      Length: 241
      Type: HEADERS (1)
    ▸ Flags: 0x04, End Headers
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
      [Pad Length: 0]
      Header Block Fragment: 886294fe420132b38175c91b1bcb11e00a307dfdfdff3f6c96d07abe94136a6a22541004…
      [Header Length: 432]
      [Header Count: 9]
    ▸ Header: :status: 200 OK
    ▸ Header: etag: "1023-6176d5a8-c802b0;;;"
    ▸ Header: last-modified: Mon, 25 Oct 2021 16:04:56 GMT
    ▸ Header: content-type: text/html
    ▸ Header: content-length: 4131
    ▸ Header: accept-ranges: bytes
    ▸ Header: date: Fri, 08 Apr 2022 12:27:35 GMT
    ▸ Header: server: LiteSpeed
    ▸ Header: alt-svc: h3=":443"; ma=2592000, h3-29=":443"; ma=2592000, h3-Q050=":443"; ma=2592000, h3-Q046=":443";
▾ HyperText Transfer Protocol 2
  ▸ Stream: DATA, Stream ID: 1, Length 4131 (partial entity body)
▾ HyperText Transfer Protocol 2
  ▸ Stream: DATA, Stream ID: 1, Length 0
  ▸ Line-based text data: text/html (151 lines)
```

Figure 9: The server response

with the status code 200 OK. The packet content tells the whole story: The Litespeed server has sent 4131 bytes of text/html content on 8th of April 2022. Line-based text data tab has the plain-text data included in the response, as



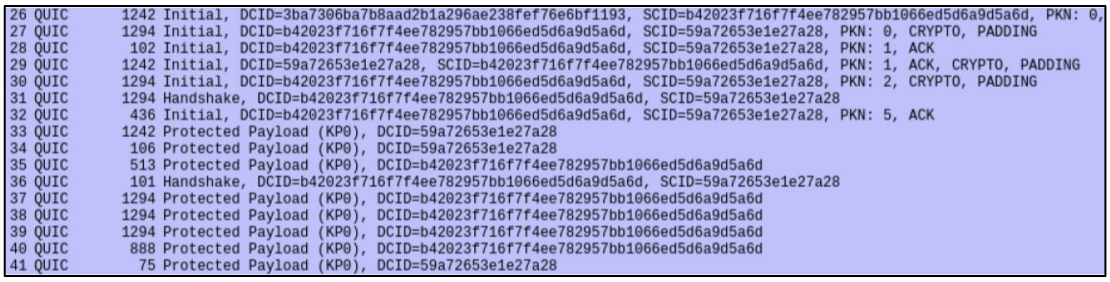
```
▾ HyperText Transfer Protocol 2
  ▸ Stream: DATA, Stream ID: 1, Length 0
  ▾ Line-based text data: text/html (151 lines)
      <!DOCTYPE html>\n
      <html lang="en">\n
      <title>HTTP/3 server</title>\n
      <meta charset="UTF-8">\n
      <meta name="viewport" content="width=device-width, initial-scale=1">\n
      <link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">\n
      <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Lato">\n
      <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Montserrat">\n
      <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">\n
      <style>\n
      body,h1,h2,h3,h4,h5,h6 {font-family: "Lato", sans-serif}\n
      .w3-bar,h1,button, file {font-family: "Montserrat", sans-serif}\n
      .fa-anchor,.fa-coffee {font-size:200px}\n
      \n
      \n
      \n
      input[type=file]::-webkit-file-upload-button {\n
        padding: .2em .4em;\n
        border-radius: 0px;\n
        background-color: #262626;\n
        transition: 0.1s;\n
      }\n
      \n
      input[type=file]::file-selector-button {\n
        color: #fff;\n
        font-size: 18px;\n
        width: 130px;\n
        height: 50px;\n
        border: 2px solid #000;\n
        padding: .2em .4em;\n
        border-radius: 0px;\n
```

Figure 10: Decrypted plain-text payload

seen in figure 10. The content has been cut short in the figure since it was not necessary to display it in whole. The text data here is the server’s HTML/CSS.

4.3 HTTP/3 and QUIC

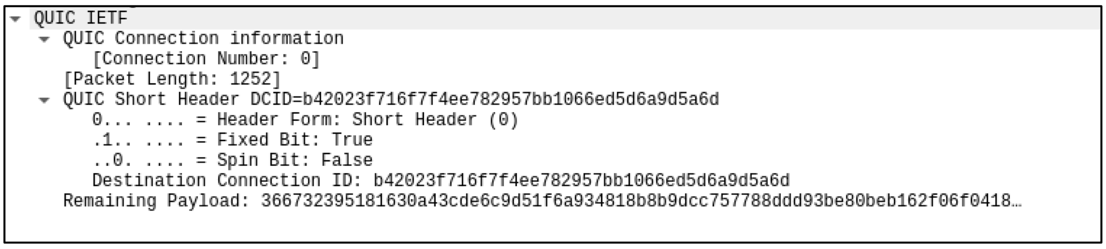
As expected in theory, capturing encrypted HTTP/3 and QUIC results in minimal traffic details. In figure 11, a new 1-RTT connection has been established with the same server, this time using HTTP/3 as a protocol. The whole stream is just labeled QUIC. Unlike in encrypted HTTP/2, It is not easy to spot from the feed, where the actual data stream is opened, since the stream content size remains seemingly the same. The different phases of QUIC are visible, though: the initial packets and the handshake. The initial packets have visible packet numbers: by the multiplexing feature of QUIC, the packet order differs. "Protected payload" label is used for everything after the connection establishment.



```
26 QUIC      1242 Initial, DCID=3ba7306ba7b8aad2b1a296ae238fef76e6bf1193, SCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 0,
27 QUIC      1294 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 0, CRYPTO, PADDING
28 QUIC       102 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 1, ACK
29 QUIC      1242 Initial, DCID=59a72653e1e27a28, SCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 1, ACK, CRYPTO, PADDING
30 QUIC      1294 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 2, CRYPTO, PADDING
31 QUIC      1294 Handshake, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28
32 QUIC       436 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 5, ACK
33 QUIC      1242 Protected Payload (KP0), DCID=59a72653e1e27a28
34 QUIC       106 Protected Payload (KP0), DCID=59a72653e1e27a28
35 QUIC       513 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
36 QUIC       101 Handshake, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28
37 QUIC      1294 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
38 QUIC      1294 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
39 QUIC      1294 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
40 QUIC       888 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
41 QUIC        75 Protected Payload (KP0), DCID=59a72653e1e27a28
```

Figure 11: HTTP/3 + QUIC encrypted traffic

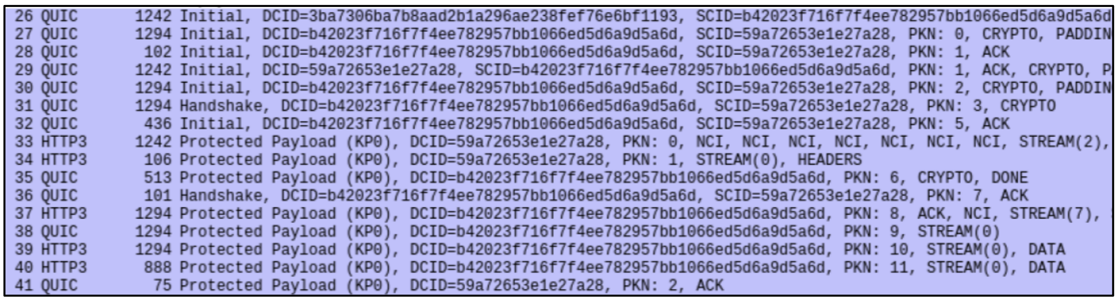
It is intriguing to see in practice, how the encryption and compression has been implemented. Examining packet number 39, which is later revealed to be a HTTP/3 data packet, has a similar emptiness to a plain TLS packet, as seen in figure 12. Everything in the QUIC tab is encrypted and it does not list all the



```
▾ QUIC IETF
  ▾ QUIC Connection information
      [Connection Number: 0]
    [Packet Length: 1252]
  ▾ QUIC Short Header DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
      0... .... = Header Form: Short Header (0)
      .1.. .... = Fixed Bit: True
      ..0. .... = Spin Bit: False
      Destination Connection ID: b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
    Remaining Payload: 366732395181630a43cde6c9d51f6a934818b8b9dcc757788ddd93be80beb162f06f0418…
```

Figure 12: Encrypted packet information

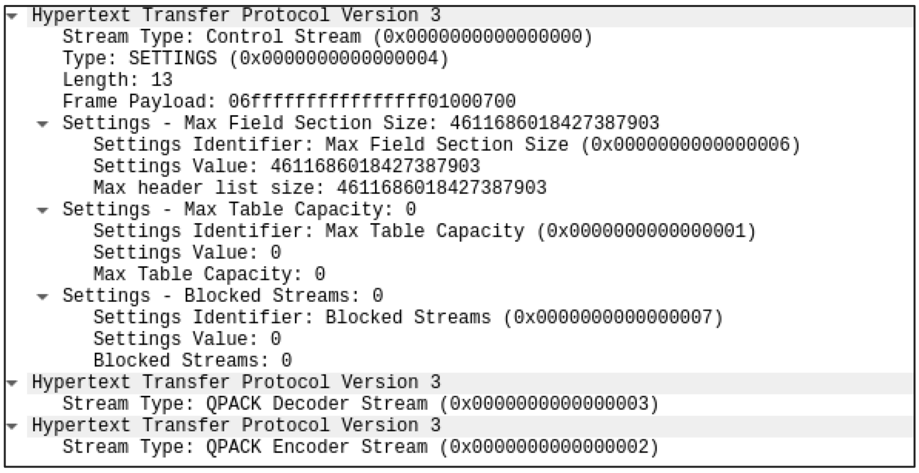
frames in QUIC structure. The protocol does not even tell the application-level protocol which it is delivering, unlike in plain TLS, which was seen in figure 6. Decrypting the captured traffic does not reveal much more. The QUIC packet contents are a bit more detailed, and the HTTP/3 protocol is now visible, as seen in figure 13. By analyzing the feed there is not a clear conversation going on, unlike in HTTP/2. Looking through the traffic to identify a possible GET-



```
26 QUIC      1242 Initial, DCID=3ba7306ba7b8aad2b1a296ae238fef76e6bf1193, SCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
27 QUIC      1294 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 0, CRYPTO, PADDIN
28 QUIC       102 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 1, ACK
29 QUIC      1242 Initial, DCID=59a72653e1e27a28, SCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 1, ACK, CRYPTO, P
30 QUIC      1294 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 2, CRYPTO, PADDIN
31 QUIC      1294 Handshake, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 3, CRYPTO
32 QUIC       436 Initial, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 5, ACK
33 HTTP3     1242 Protected Payload (KP0), DCID=59a72653e1e27a28, PKN: 0, NCI, NCI, NCI, NCI, NCI, NCI, NCI, STREAM(2),
34 HTTP3      106 Protected Payload (KP0), DCID=59a72653e1e27a28, PKN: 1, STREAM(0), HEADERS
35 QUIC       513 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 6, CRYPTO, DONE
36 QUIC       101 Handshake, DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, SCID=59a72653e1e27a28, PKN: 7, ACK
37 HTTP3     1294 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 8, ACK, NCI, STREAM(7),
38 QUIC      1294 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 9, STREAM(0)
39 HTTP3     1294 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 10, STREAM(0), DATA
40 HTTP3      888 Protected Payload (KP0), DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d, PKN: 11, STREAM(0), DATA
41 QUIC        75 Protected Payload (KP0), DCID=59a72653e1e27a28, PKN: 2, ACK
```

Figure 13: HTTP/3 + QUIC decrypted traffic

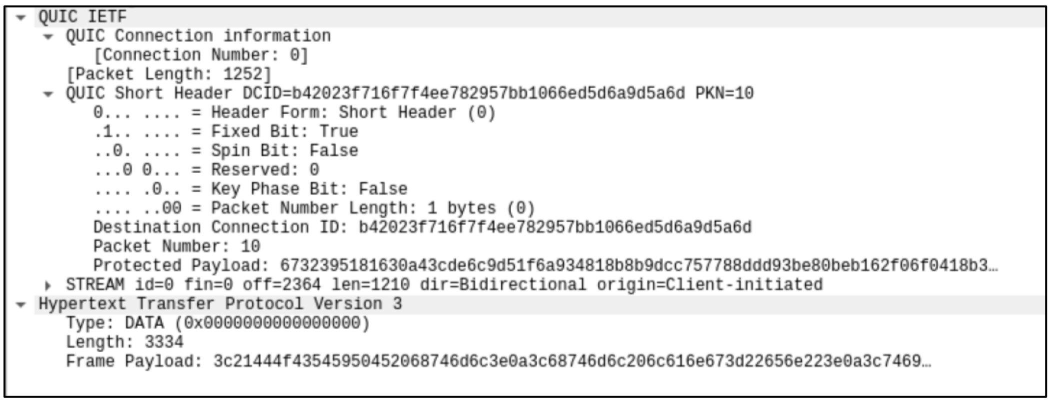
request, an observer can only find encrypted SETTINGS in packet 33. This frame, which is seen in figure 14 tells nothing to its reader since it is also still partly encrypted. The reason behind the remaining encryption was discussed in chapter 3.3 in the security in QUIC: the initial keys of the protocol are used to encrypt the payload before the actual handshake. The TLS decryption does not affect the QUIC encryption at that level, so it is able to keep most of the data secret.



```
▾ Hypertext Transfer Protocol Version 3
    Stream Type: Control Stream (0x0000000000000000)
    Type: SETTINGS (0x0000000000000004)
    Length: 13
    Frame Payload: 06ffffffffffffffff01000700
  ▾ Settings - Max Field Section Size: 4611686018427387903
      Settings Identifier: Max Field Section Size (0x0000000000000006)
      Settings Value: 4611686018427387903
      Max header list size: 4611686018427387903
  ▾ Settings - Max Table Capacity: 0
      Settings Identifier: Max Table Capacity (0x0000000000000001)
      Settings Value: 0
      Max Table Capacity: 0
  ▾ Settings - Blocked Streams: 0
      Settings Identifier: Blocked Streams (0x0000000000000007)
      Settings Value: 0
      Blocked Streams: 0
▾ Hypertext Transfer Protocol Version 3
    Stream Type: QPACK Decoder Stream (0x0000000000000003)
▾ Hypertext Transfer Protocol Version 3
    Stream Type: QPACK Encoder Stream (0x0000000000000002)
```

Figure 14: The SETTINGS frame of packet number 33

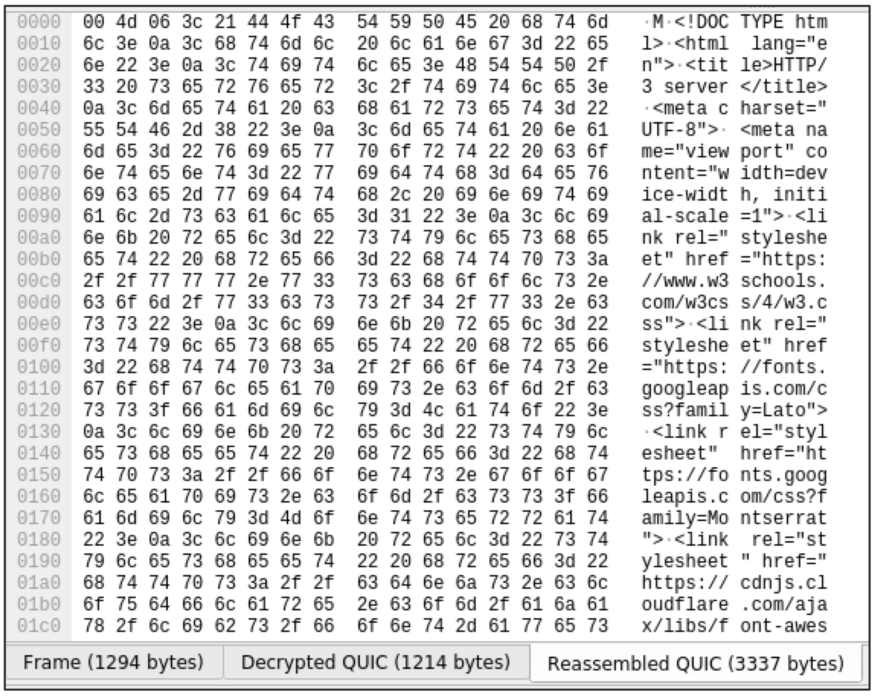
Selecting packet number 39 again, now revealed as HTTP/3 packet, results in an information window as seen in figure 15. The packet itself is labeled DATA-type, meaning it would logically be the server response. Nothing is shown in



```
▾ QUIC IETF
  ▾ QUIC Connection information
      [Connection Number: 0]
    [Packet Length: 1252]
  ▾ QUIC Short Header DCID=b42023f716f7f4ee782957bb1066ed5d6a9d5a6d PKN=10
      0... .... = Header Form: Short Header (0)
      .1.. .... = Fixed Bit: True
      ..0. .... = Spin Bit: False
      ...0 0... = Reserved: 0
      .... .0.. = Key Phase Bit: False
      .... ..00 = Packet Number Length: 1 bytes (0)
      Destination Connection ID: b42023f716f7f4ee782957bb1066ed5d6a9d5a6d
      Packet Number: 10
      Protected Payload: 6732395181630a43cde6c9d51f6a934818b8b9dcc757788ddd93be80beb162f06f0418b3…
  ▸ STREAM id=0 fin=0 off=2364 len=1210 dir=Bidirectional origin=Client-initiated
▾ Hypertext Transfer Protocol Version 3
    Type: DATA (0x0000000000000000)
    Length: 3334
    Frame Payload: 3c21444f43545950452068746d6c3e0a3c68746d6c206c616e673d22656e223e0a3c7469…
```

Figure 15: Decrypted packet number 39

the packet, other than the actual content length. It has a singular HTTP/3 tab after QUIC, and the frame payload of it appears to be still encrypted. QUIC frame structure instead has everything displayed but seeing its content does not help an eavesdropper. Looking through Wireshark decryption tabs, which summarizes the packet data in hexadecimal and plain-text, results in finding the plain-text payload. In figure 16, under “Reassembled QUIC” label, is the



```
0000  00 4d 06 3c 21 44 4f 43  54 59 50 45 20 68 74 6d   ·M·<!DOC TYPE htm
0010  6c 3e 0a 3c 68 74 6d 6c  20 6c 61 6e 67 3d 22 65   l>·<html  lang="e
0020  6e 22 3e 0a 3c 74 69 74  6c 65 3e 48 54 54 50 2f   n">·<tit le>HTTP/
0030  33 20 73 65 72 76 65 72  3c 2f 74 69 74 6c 65 3e   3 server </title>
0040  0a 3c 6d 65 74 61 20 63  68 61 72 73 65 74 3d 22   ·<meta c harset="
0050  55 54 46 2d 38 22 3e 0a  3c 6d 65 74 61 20 6e 61   UTF-8">· <meta na
0060  6d 65 3d 22 76 69 65 77  70 6f 72 74 22 20 63 6f   me="view port" co
0070  6e 74 65 6e 74 3d 22 77  69 64 74 68 3d 64 65 76   ntent="w idth=dev
0080  69 63 65 2d 77 69 64 74  68 2c 20 69 6e 69 74 69   ice-widt h, initi
0090  61 6c 2d 73 63 61 6c 65  3d 31 22 3e 0a 3c 6c 69   al-scale =1">·<li
00a0  6e 6b 20 72 65 6c 3d 22  73 74 79 6c 65 73 68 65   nk rel=" styleshe
00b0  65 74 22 20 68 72 65 66  3d 22 68 74 74 70 73 3a   et" href ="https:
00c0  2f 2f 77 77 77 2e 77 33  73 63 68 6f 6f 6c 73 2e   //www.w3 schools.
00d0  63 6f 6d 2f 77 33 63 73  73 2f 34 2f 77 33 2e 63   com/w3cs s/4/w3.c
00e0  73 73 22 3e 0a 3c 6c 69  6e 6b 20 72 65 6c 3d 22   ss">·<li nk rel="
00f0  73 74 79 6c 65 73 68 65  65 74 22 20 68 72 65 66   styleshe et" href
0100  3d 22 68 74 74 70 73 3a  2f 2f 66 6f 6e 74 73 2e   ="https: //fonts.
0110  67 6f 6f 67 6c 65 61 70  69 73 2e 63 6f 6d 2f 63   googleap is.com/c
0120  73 73 3f 66 61 6d 69 6c  79 3d 4c 61 74 6f 22 3e   ss?famil y=Lato">
0130  0a 3c 6c 69 6e 6b 20 72  65 6c 3d 22 73 74 79 6c   ·<link r el="styl
0140  65 73 68 65 65 74 22 20  68 72 65 66 3d 22 68 74   esheet"  href="ht
0150  74 70 73 3a 2f 2f 66 6f  6e 74 73 2e 67 6f 6f 67   tps://fo nts.goog
0160  6c 65 61 70 69 73 2e 63  6f 6d 2f 63 73 73 3f 66   leapis.c om/css?f
0170  61 6d 69 6c 79 3d 4d 6f  6e 74 73 65 72 72 61 74   amily=Mo ntserrat
0180  22 3e 0a 3c 6c 69 6e 6b  20 72 65 6c 3d 22 73 74   ">·<link  rel="st
0190  79 6c 65 73 68 65 65 74  22 20 68 72 65 66 3d 22   ylesheet " href="
01a0  68 74 74 70 73 3a 2f 2f  63 64 6e 6a 73 2e 63 6c   https:// cdnjs.cl
01b0  6f 75 64 66 6c 61 72 65  2e 63 6f 6d 2f 61 6a 61   oudflare .com/aja
01c0  78 2f 6c 69 62 73 2f 66  6f 6e 74 2d 61 77 65 73   x/libs/f ont-awes
```

Figure 16: Frame payload, reassembled from QUIC

server response which has the same server content as seen in figure 9 of HTTP/2 decryption. Interestingly, this content is not included in the HTTP/3 tabs, as it was in figure 10 under “line-based text data” of HTTP/2.

4.4 Overview of the results

Comparing the two protocols gave an insight on how different HTTP/3 really is security-wise. In the case of HTTP/2, using the keylog method in Wireshark provides a way to strip the TLS encryption, as well as the decompression layer of the protocol. Doing so, the HTTP conversation becomes clearly visible. In the GET-request, headers exposed include the used method, path, scheme, authority, and the user-agent. In this test, this packet states that the client used curl/7.68.0 to form a GET request to a https-version of the server 130.230.84.31. The response from the server gave out the server type, Litespeed, a date on which the server has been last modified, and a plain-text version of the server content. In a similar test, the keylog method used on HTTP/3 traffic did not wield as informative results. Decryption gives some information on the details of QUIC, but nothing useful exploit-wise. The analysis of HTTP/3 packets resulted in getting the page content as a server response, with all the important details left out. The QUIC initial decryption does not wear down after a TLS decryption, so the traffic remains partly encrypted.

In conclusion, an attacker possessing a keylog file can easily find out everything they need about a HTTP/2 client-server conversation, by going through some extra steps. HTTP/3, on the other hand stands strong against an eavesdropper. Even though possessing a keylog file, an attacker can only get the page content, as one would with a curl request. While this would also be a poor scenario, it leaves out certain server details, which could in a long run cause a side-channel attack. This short comparison tells that HTTP/3 has achieved what it promised. In traffic analysis it is indeed more secure than HTTP/2, by having heightened encryption.

5. EVALUATION AND CONCLUSION

The research made for this thesis makes a promise of a fast but reliable protocol. The secure properties of the new protocol have been enforced significantly from the previous version, HTTP/2. Suffering from side-channel leaks and the downgrade vulnerabilities, HTTP/2 does not ensure a connection to achieve the security standards of today. HTTP/3 patches these flaws by having multiple layers of encryption by default, and an assurance that the protocol cannot be downgraded.
The security in HTTP/3 is mainly provided by QUIC, as supported by the practical test in the thesis. The initial keys which QUIC uses to encrypt a payload in the start eliminate the possibility of side-channel analysis. Although HTTP/3 is now built on UDP-like features, the protocol provides countermeasure like packet rate limiting, and QUIC uses an anti-amplification mechanism.

Regardless of the state of security now, the protocol is still in development. The changes it could still go through can affect it. After its release it should be treated with caution, since it still has vulnerabilities in features like 0-RTT. As time goes by, and more people start testing it, we will see if there are any underlying security flaws. Additionally, the future versions of QUIC may not be compatible with HTTP/3 and could enable cross-protocol vulnerabilities. The potential vulnerabilities may rise even after years of the initial release, in which case the security of it must be re-evaluated. In future, Wireshark will possibly support full QUIC decryption as well. It means QUIC and HTTP/3 could be analyzed in more detail, and if so, there could be similar results to traffic analysis as there is now to HTTP/2.

6. REFERENCES

[1] MDN contributors, "Evolution of HTTP," 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.

[2] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," June 1999. [Online]. Available: https://www.ietf.org/rfc/rfc2616.txt.

[3] "Introduction to HTTPS," [Online]. Available: https://https.cio.gov/faq/.

[4] Microsoft, "Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow," 2016. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/WebAppSideChannel-final.pdf .

[5] M. Belshe, R. Peon and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," 2015. [Online]. Available: https://tools.ietf.org/html/rfc7540.

[6] R. Peon and H. Ruellan, "HPACK: Header Compression for HTTP/2," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7541.

[7] J. Griffin, "HTTP/2 vs HTTP/1," February 2019. [Online]. Available: https://www.thewebmaster.com/hosting/articles/what-is-http2-and-how-does-it-compare-to-http1-1/.

[8] "Heartbleed bug," June 2020. [Online]. Available: https://heartbleed.com/.

[9] J. Kettle, "HTTP/2: The Sequel is Always Worse," August 2021. [Online]. Available: https://portswigger.net/research/http2.

[10] J. Iyengar, "QUIC is now RFC 9000," May 2021. [Online]. Available: https://www.fastly.com/blog/quic-is-now-rfc-9000.

[11] A. Kyratis and P. Cottis, "QUIC vs TCP: A Performance Evaluation over LTE with NS-3," *Communications and Network,* vol. 14, pp. 12-22, February 2022.

[12] R. -. P. Standard, "QUIC: A UDP-Based Multiplexed and Secure Transport," 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc9000/.

[13] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," August 2018. [Online]. Available: https://tools.ietf.org/html/rfc8446.

[14] J. Zhang, L. Yang, X. Gao, G. Tang, J. Zhang and Q. Wang, "Formal Analysis of QUIC Handshake Protocol Using Symbolic Model Checking," *IEEE Access,* vol. 9, pp. 14836-14848, 2021.

[15] M. Bishop, "Hypertext Transfer Protocol Version 3 (HTTP/3)," 2021. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-http-34.

[16] M. Thomson and S. Turner, May 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9001.html.

[17] E. Gagliardi and O. Levillain, "Analysis of QUIC session establishment and its implementations," 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02468596/document.

[18] R. Seal, "Looking Into QUIC Packets in your Network," 7 2021. [Online]. Available: https://blogs.keysight.com/blogs/tech/nwvs.entry.html/2021/07/17/looking_into_quicpa-pUtF.html.

[19] IEEE, "How Secure and Quick is QUIC? Provable Security and Performance Analyses," 2015. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7163028.

[20] A. Ghedini, "Even faster connection establishment with QUIC 0-RTT resumption," 2019. [Online]. Available: https://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/.

[21] M. Thomson, M. Nottingham and W. Tarreau, "Using Early Data in HTTP," September 2018. [Online].

[22] Koromicha, "Create Locally Trusted SSL Certificates with mkcert on Ubuntu 18.04," October 2018. [Online]. Available: https://kifarunix.com/how-to-create-self-signed-ssl-certificate-with-mkcert-on-ubuntu-18-04/.

[23] "SSLKEYLOGFILE - Everything curl," [Online]. Available: https://everything.curl.dev/usingcurl/tls/sslkeylogfile.