Deciphering WhatsApp Web

CSEC380 - Computer Security Blogpost - Mehul Sen

Introduction

WhatsApp is a free messaging and voice-over-IP service available on multiple platforms. It is owned by Facebook, Inc. and enables users to send text and voice messages, make voice and video calls, as well as share images, documents, user locations, and other content[1].

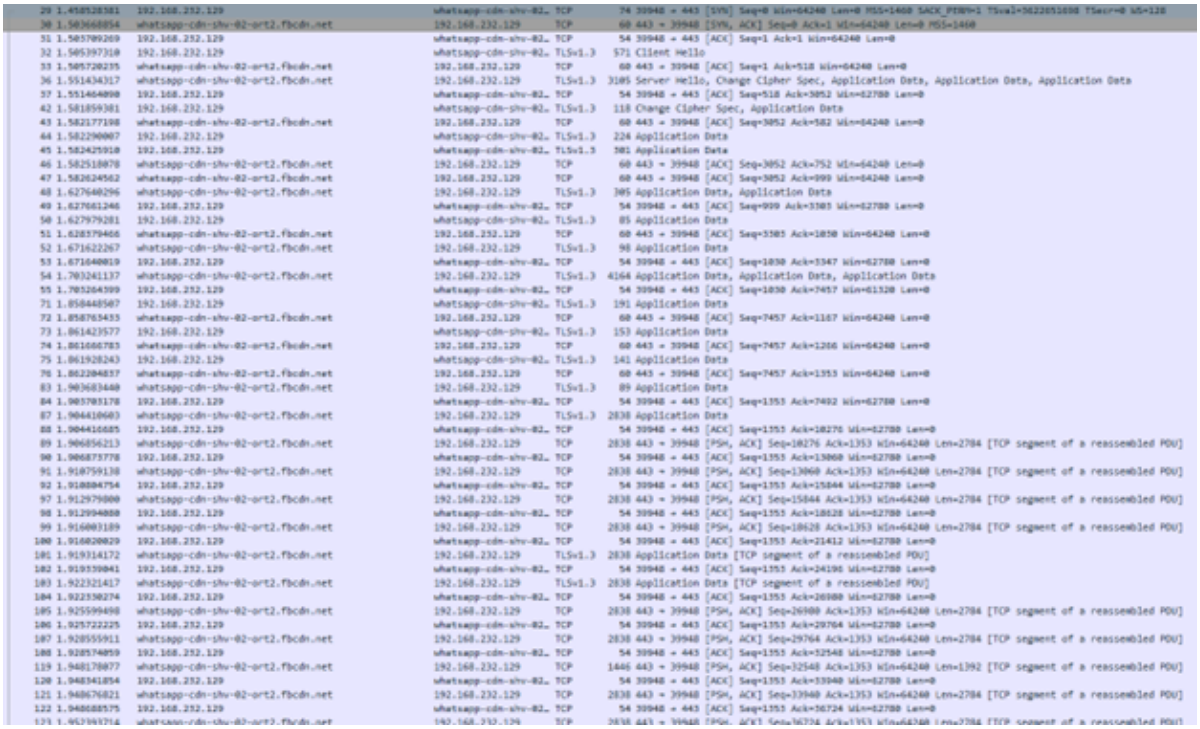
One of its notable features is known as WhatsApp Web[2], which allows users to access the messaging service on desktop computers as long as their mobile device remains connected to the internet. This capability was introduced in 2017 and it syncs the phone application with the desktop app, using the same WhatsApp account, thus retrieving all the messages and media. Unlike the mobile application, WhatsApp Web works on web browsers, which allows for further investigation into its infrastructure to better understand its behavior.

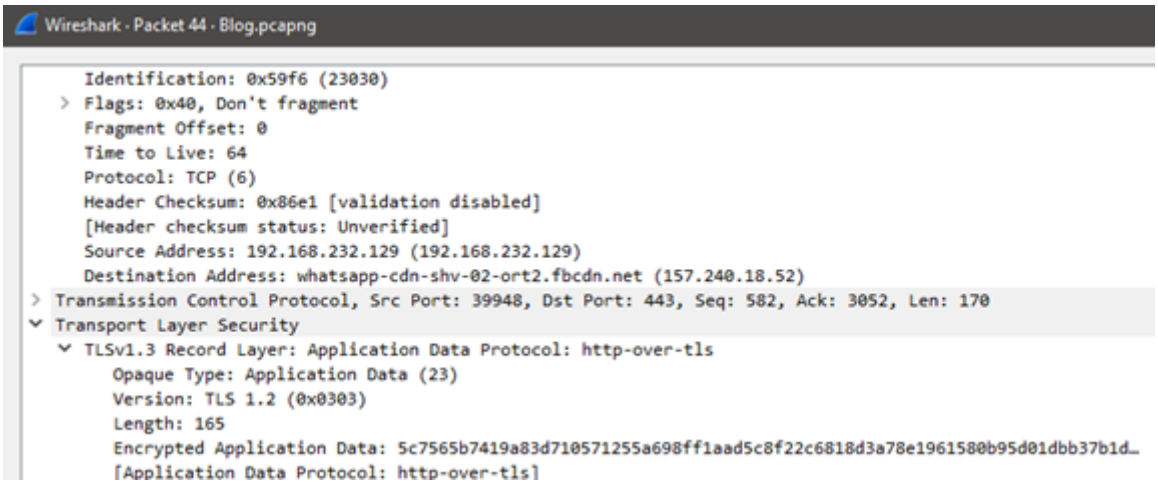
Setup

As part of my research, I used an Ubuntu machine to log my SSL handshakes. The machine had a Firefox web browser and Wireshark installed, along with a stable internet connection and a working WhatsApp account.

Since WhatsApp, like most messaging platforms, uses SSL to encrypt their traffic, I first needed to find a way to bypass the SSL encryption.

To confirm this, I took a small Wireshark capture while using WhatsApp web and observed that all the traffic going to the WhatsApp server from the web browser was indeed encrypted using TLS. I was also able to see the TLS Client Hello and the change Cipher protocols in the Wireshark capture.





A bit of digging around provided me with an article on the AskF5 forum[2]. To get a better explanation of this method, I would suggest going over the link itself however the basic steps to go decipher the TLS packets are as follows:

- Create a Log file to store all the TLS handshakes.

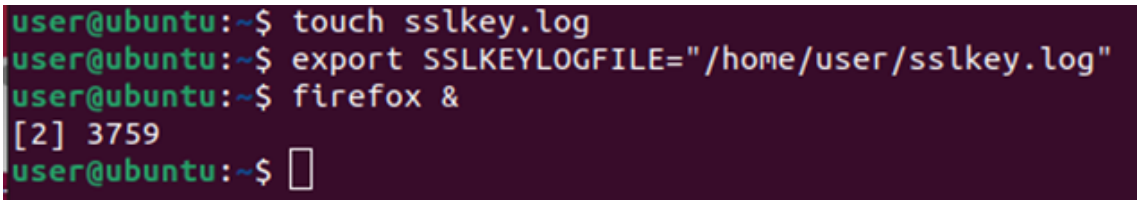
  `$ touch sslkey.log`

- Modify the SSLKEYLOGFILE environment variable to store the handshakes within that log file.

  `$ export SSLKEYLOGFILE="/home/user/sslkey.log"`

- Open Firefox within the same shell

  `$ firefox &"`



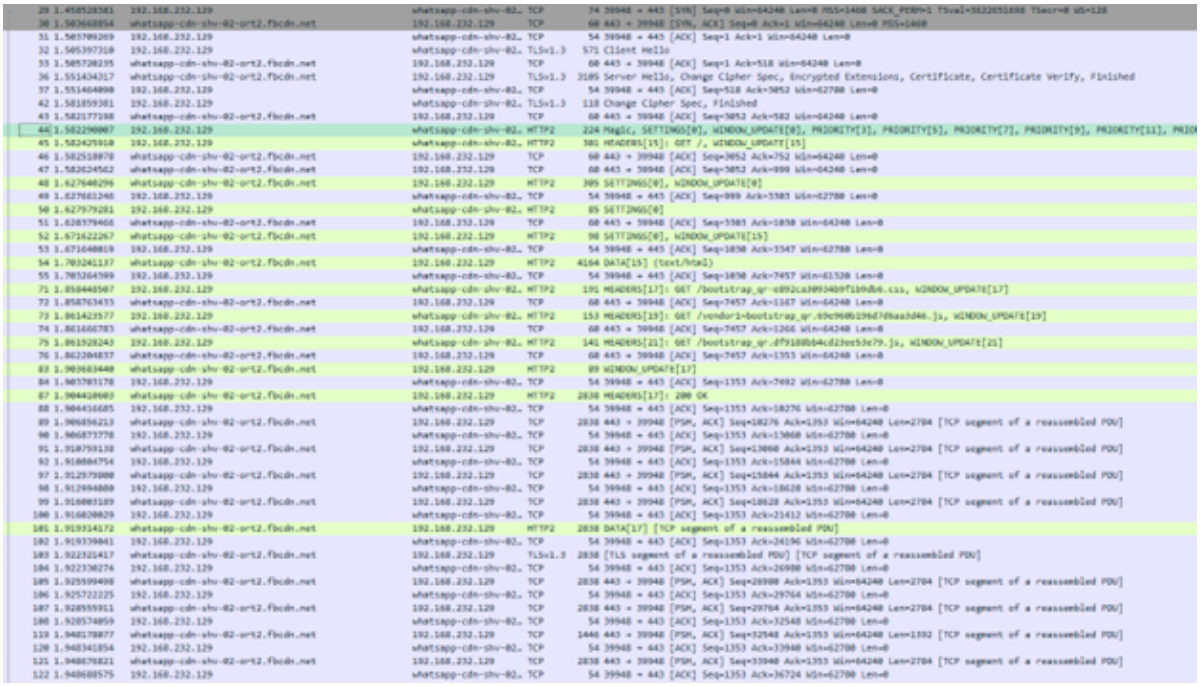
After using WhatsApp web using this web browser and capturing all this traffic through Wireshark we now have everything we need to see what is within the TLS packets.

This is what the generated sslkey.log file looks like once it had logged all the handshakes.

```
1 # SSL/TLS secrets log file, generated by NSS
2 CLIENT_HANDSHAKE_TRAFFIC_SECRET
  6ebfc2fde2468ad973a16fe122f9701dd9dd78c1a6fbbd06aa72548b9437d618
  a7665028243be1c447920ead01b79810d70833aae3a509be589086dea285feb0
3 SERVER_HANDSHAKE_TRAFFIC_SECRET
  6ebfc2fde2468ad973a16fe122f9701dd9dd78c1a6fbbd06aa72548b9437d618
  db07376dd3c5344eafc8bcbc227e1e72ccb18e174be30cf879901bafb95b8b5d
4 CLIENT_TRAFFIC_SECRET_0 6ebfc2fde2468ad973a16fe122f9701dd9dd78c1a6fbbd06aa72548b9437d618
  cab81afadaad8e4b5b8e02f3c163cfd34cc10a8540d419dc5b84a5fba6127560
5 SERVER_TRAFFIC_SECRET_0 6ebfc2fde2468ad973a16fe122f9701dd9dd78c1a6fbbd06aa72548b9437d618
  157a0901d6077893e295f10679ed1319fd16466ee49fca195da375b15d9b7934
6 EXPORTER_SECRET 6ebfc2fde2468ad973a16fe122f9701dd9dd78c1a6fbbd06aa72548b9437d618
  2aa66430a6ab3d7d8dd313ef805d2f76a2b3bfd764af5f4ecea88d0bb7efef7b
7 CLIENT_HANDSHAKE_TRAFFIC_SECRET
  c3d9e137d1a7d9788bf94e9476d6efbcc8b81d1c89184bfab88d4f615a8a1492
  b670e59020c8b8b2e25bd6a9d9be97382ba4c9927802a1264902b1f59aae977c
8 SERVER_HANDSHAKE_TRAFFIC_SECRET
  c3d9e137d1a7d9788bf94e9476d6efbcc8b81d1c89184bfab88d4f615a8a1492
  94f907c4cf117375a70bb5b9204c9563d970a537d6f561be9680b90d86de9abc
9 CLIENT_TRAFFIC_SECRET_0 c3d9e137d1a7d9788bf94e9476d6efbcc8b81d1c89184bfab88d4f615a8a1492
  a6cfc1b8204380f6e74fa901edecbc278d749a06ca6ee420e630169aa649efa7
10 SERVER_TRAFFIC_SECRET_0 c3d9e137d1a7d9788bf94e9476d6efbcc8b81d1c89184bfab88d4f615a8a1492
  ab26f8fbc4ac24975032a50587f01da433090471cf93d21420220d2af3cbfc4e
11 EXPORTER_SECRET c3d9e137d1a7d9788bf94e9476d6efbcc8b81d1c89184bfab88d4f615a8a1492
  9083568740f633bb607cccb5f1cffedcadddd580c088761ef86b8fcb10bf2792
12 CLIENT_HANDSHAKE_TRAFFIC_SECRET
  2a3a9550f79ebe4f3e5e42506738c74411175a48b2767d0398f58828eede4d7c
  1d0c04fb5f65646b472193874a9283f90c8966cd57d7bb78db204bf34c91b32e
13 SERVER_HANDSHAKE_TRAFFIC_SECRET
  2a3a9550f79ebe4f3e5e42506738c74411175a48b2767d0398f58828eede4d7c
  92b53d3d40e051970da22428c14d34c8b4224642e09c5f57bca2d7804aefd3e0
14 CLIENT_TRAFFIC_SECRET_0 2a3a9550f79ebe4f3e5e42506738c74411175a48b2767d0398f58828eede4d7c
  78911d8aaaae00094dfe0f1442d1382dba3b4195a893603cad16d516fe7eed5a
15 SERVER_TRAFFIC_SECRET_0 2a3a9550f79ebe4f3e5e42506738c74411175a48b2767d0398f58828eede4d7c
  8caff8d2a050dd7574e9bf9976cf78b272839c8c10105305f4999a589460141b
16 EXPORTER_SECRET 2a3a9550f79ebe4f3e5e42506738c74411175a48b2767d0398f58828eede4d7c
  b70812ad73c40043eab9995efbe481722078e8ff843029a369c9a4b1949a684c
```

Wireshark has a built-in capability to read these logs and decrypt the .pcap traffic. This option is available under Edit > Preferences > Protocols > TLS > (Pre)-Master-Secret Log, selecting the generated sslkey.log, we are now able to look within all the encrypted TLS traffic.

By doing this, we are immediately able to see all the TLS traffic between the WhatsApp server and our web browser.

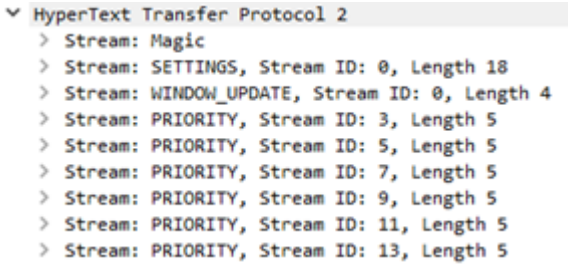


We can now go further and analyze these packets to get a better clue of how WhatsApp works.

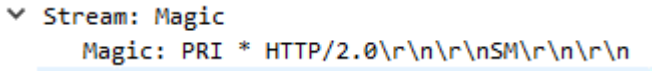
Analysis

After the TLS handshakes and the TCP connections, the first observation is that WhatsApp uses HTTP2 for most of its communication, the first packet sends out a Stream consisting of a couple of different headers.

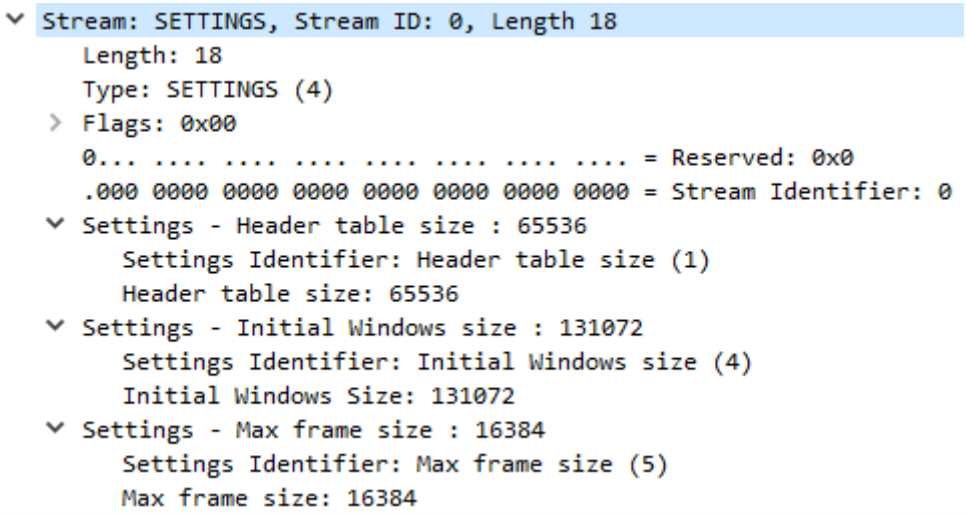
These are `MAGIC`, `SETTINGS`, `WINDOW_UPDATE`, and `PRIORITY`.



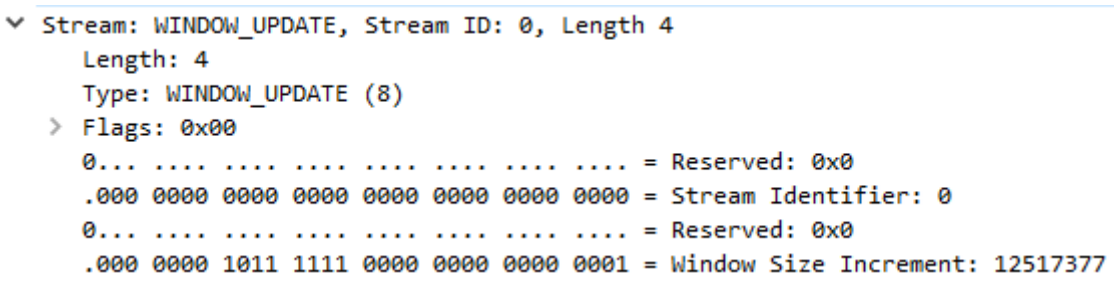
`MAGIC` – This contains the protocol and basic information about how data is being sent.



`SETTINGS` – Looking at the parameters which are Header Table Size, Initial Window Size, and Max Frame Size, This header would be used to set the initial browser window size, optimizing itself for the max size of frames it should receive based on the connection.



`WINDOW_UPDATE` – This header might provide any changes made to the window itself, updating the server using a parameter called “Window Size Increment”.



`PRIORITY` – This header contains five parameters, Reserved, Stream Identifier, Exclusive, and Stream Dependency and Weight. Looking at multiple PRIORITY headers, we learn that the Stream Identifier is the ID pertaining to the Data that gets sent at a later stage, The Stream Dependency is the number on which that PRIORITY header is dependent on and it is always a value lower than its own Stream ID. The Weight ranges from 0 to 240, and this might signify how important any particular Stream ID is.

```
Stream: PRIORITY, Stream ID: 3, Length 5
    Length: 5
    Type: PRIORITY (2)
  > Flags: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0000 0011 = Stream Identifier: 3
    0... .... .... .... .... .... .... .... = Exclusive: False
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Dependency: 0
    Weight: 200
    [Weight real: 201]
```

Although we can see `HEADERS` and `WINDOW_UPDATE` as well as `SETTINGS` throughout the packet capture, the `MAGIC` and `PRIORITY` headers are not sent again which might mean that they are used for only the initial setup for WhatsApp Web.

After the initial packet, the next packet contains a new header called `HEADERS`. This contains information like the method used, the path, the authority, scheme, user-agent, data accepted, the language accepted, encoding accepted, upgrade-insecure-requests, and the trailers.

```
Stream: HEADERS, Stream ID: 15, Length 203, GET /
  Length: 203
  Type: HEADERS (1)
  Flags: 0x25, Priority, End Headers, End Stream
    00.0 ..0. = Unused: 0x00
    ..1. .... = Priority: True
    .... 0... = Padded: False
    .... .1.. = End Headers: True
    .... ...1 = End Stream: True
  0... .... .... .... .... .... .... .... = Reserved: 0x0
  .000 0000 0000 0000 0000 0000 0000 1111 = Stream Identifier: 15
  [Pad Length: 0]
  0... .... .... .... .... .... .... .... = Exclusive: False
  .000 0000 0000 0000 0000 0000 0000 0111 = Stream Dependency: 7
  Weight: 41
  [Weight real: 42]
  Header Block Fragment: 82048163418cf058d7f138d281d75ae43d3f877abad07f66a281b0dae053fafc087ed4e1…
  [Header Length: 397]
  [Header Count: 10]
> Header: :method: GET
> Header: :path: /
> Header: :authority: web.whatsapp.com
> Header: :scheme: https
> Header: user-agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0) Gecko/20100101 Firefox/88.0
> Header: accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
> Header: accept-language: en-US,en;q=0.5
> Header: accept-encoding: gzip, deflate, br
> Header: upgrade-insecure-requests: 1
> Header: te: trailers
```

The next couple of `HEADERS` send `GET` requests to "`/bootstrap_qr-e892ca30934b9f1b9db6.css`" for CSS, and "`/vendor1~bootstrap_qr.69e960b196d7d6aa3d46.js`", "`/bootstrap_qr.df9188bb4cd23ee53e79.js`" for JavaScript.

As a reply, the server sends back a packet containing a `DATA` header. The first couple of `DATA` headers contain HTML, CSS and JavaScript requested by the Web Browser.

```
▾ Stream: DATA, Stream ID: 15, Length 2793
    Length: 2793
    Type: DATA (0)
  ▾ Flags: 0x01, End Stream
      0000 .00. = Unused: 0x00
      .... 0... = Padded: False
      .... ...1 = End Stream: True
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0000 1111 = Stream Identifier: 15
    [Pad Length: 0]
  ▾ Content-encoded entity body (br): 2793 bytes -> 6803 bytes
      Data: 3c21646f63747970652068746d6c3e3c68746d6c20636c6173733d226e6f2d6a73222064…
  ▾ Line-based text data: text/html (1 lines)
      [truncated]<!doctype html><html class="no-js" dir="ltr" loc="en"><head><meta char
```

The `DATA` header is much simpler than its counterparts, it has a Stream Identifier parameter and a Data parameter that contain that data. It also has flags like Unused, Padded, and End-Stream. Padded is set to 1 when data is being sent, after the data has been completed, it gets set to 0 and the End Stream Flag is set to 1.

```
▾ Stream: DATA, Stream ID: 21, Length 4887
    Length: 4887
    Type: DATA (0)
  › Flags: 0x01, End Stream
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0001 0101 = Stream Identifier: 21
    [Pad Length: 0]
    [Reassembled body in frame: 392]
    Data: 84b019faa977967a07e0e735f53edb9bf6f6fa01d19deca7c582ecde7d22d49f849f3349…
```

Observing all this data, we can see that it is all encrypted as well, this would be because of the implemented end-to-end encryption by WhatsApp, where each chat is individually encrypted among the recipients. WhatsApp provides its white pages which have a good explanation of how its Encryption and various features work[3].

Although the chat itself is encrypted, we are still able to catch some data types that come from the server. One example of this is when a picture is shared, we can get the PNG signature, the image header, palette, image data chunk, and the image trailer.

```
▾ Stream: DATA, Stream ID: 37, Length 711
    Length: 711
    Type: DATA (0)
  › Flags: 0x01, End Stream
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0010 0101 = Stream Identifier: 37
    [Pad Length: 0]
  › [2 Body fragments (4807 bytes): #540(4096), #540(711)]
  › Content-encoded entity body (br): 4807 bytes -> 4803 bytes
  ▾ Portable Network Graphics
      PNG Signature: 89504e470d0a1a0a
    › Image Header (IHDR)
    › Palette (PLTE)
    › Image data chunk (IDAT)
    › Image Trailer (IEND)
```

The emojis shared can also be seen when the web browser uses a `HEADERS` header to send a `GET` request for those resources, which goes as follows `GET /img/[emoji-name].webp`

```
Stream: HEADERS, Stream ID: 65, Length 37, GET /img/emoji-3-40_0b9fe45.webp
    Length: 37
    Type: HEADERS (1)
  > Flags: 0x25, Priority, End Headers, End Stream
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0100 0001 = Stream Identifier: 65
    [Pad Length: 0]
    0... .... .... .... .... .... .... .... = Exclusive: False
    .000 0000 0000 0000 0000 0000 0000 0111 = Stream Dependency: 7
    Weight: 21
    [Weight real: 22]
    Header Block Fragment: 82059560d4ccc1693f432ccacd022046fca569b5fc163affd587d4ccd2d1cdcf
    [Header Length: 375]
    [Header Count: 10]
  > Header: :method: GET
  > Header: :path: /img/emoji-3-40_0b9fe45.webp
  > Header: :authority: web.whatsapp.com
  > Header: :scheme: https
  > Header: user-agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0) Gecko/20100101 Firefox/88.0
  > Header: accept: */*
  > Header: accept-language: en-US,en;q=0.5
  > Header: accept-encoding: gzip, deflate, br
  > Header: referer: https://web.whatsapp.com/serviceworker.js
  > Header: te: trailers
```

Another interesting header that occasionally comes up is the `RST_STREAM` header which contains a Stream Identifier parameter and an Error parameter.

```
Stream: RST_STREAM, Stream ID: 59, Length 4
    Length: 4
    Type: RST_STREAM (3)
  > Flags: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0011 1011 = Stream Identifier: 59
    Error: CANCEL (8)
```

This is being used by the web browser to tell the server to stop processing certain streams or convey any other such errors.

The last header that gets sent to close the connections is the `GOAWAY` header, this is first sent by the web browser to the server and then the server echoes it back to the web browser, this contains a Stream Identifier, a reserved parameter, a Promised-Stream-ID parameter, and an Error parameter. A Successful closing takes place with the Error parameter set to 0 which is the `NO_ERROR` condition.

```
Stream: GOAWAY, Stream ID: 0, Length 8
    Length: 8
    Type: GOAWAY (7)
  > Flags: 0x00
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0000 0000 = Stream Identifier: 0
    0... .... .... .... .... .... .... .... = Reserved: 0x0
    .000 0000 0000 0000 0000 0000 0000 0000 = Promised-Stream-ID: 0
    Error: NO_ERROR (0)
```

Future Research Possibilities

Further research can be conducted on the protocol, traffic can be captured while performing various actions such as sending different media through chat, adding new users, deleting messages, creating group chats, etc. This will provide us with an even clearer understanding of the headers and protocols used by WhatsApp. We can even use the technical white pages[4] provided

by WhatsApp to further decrypt the end-to-end encrypted data, finding out exactly how each text message is sent.

Conclusion

WhatsApp Messenger is a very sophisticated environment with its own set of protocols and headers, WhatsApp Web allows us to monitor this traffic through HTTP and get a better understanding of how it behaves. We can bypass the SSL encryption and get a clearer view of the interactions taking place between WhatsApp servers and the Web browser.

Disclaimer

Since WhatsApp web is a private organization, it does not disclose its exact source code. We cannot know exactly how it behaves in the backend, the most we can do is observe the traffic and make educated guesses on how it might work. I analyzed the web traffic captured and tried making the best estimates as to how I believe these headers might be used. This information may not be completely accurate.

References

[1] https://en.wikipedia.org/wiki/WhatsApp

[2] https://web.whatsapp.com/

[3] https://support.f5.com/csp/article/K50557518

[4] https://scontent.whatsapp.net/v/t39.8562-34/122249142_469857720642275_2152527586907531259_n.pdf/WA_Security_WhitePaper.pdf?ccb=1-3&_nc_sid=2fbf2a&_nc_ohc=Uz1ThVZZLqUAX_10j-p&_nc_ht=scontent.whatsapp.net&oh=29e00608fc684abe813d602a491f5757&oe=60B54C19