Network Forensics and HTTP/2

GIAC (GCIA) Gold Certification

Author: Stefan Winkel, stefan@winkelsnet.com
Advisor: Chris Walker
Accepted: December 27, 2015

Abstract

Last May, a major new version of the HTTP protocol, HTTP/2, has been published and finalized in RFC 7540. HTTP/2, based on the SPDY protocol, which was primarily developed by Google, is a multiplexed, binary protocol where TLS has become the de-facto mandatory standard. Most of the modern web browsers (e.g. Chrome, Firefox, Edge) are now supporting HTTP/2 and some Fortune 500 companies like Google, Facebook and Twitter have enabled HTTP/2 traffic to and from their servers already. We also have seen a recent uptake in security breaches related to HTTP data compression (e.g. Crime, Beast) which is part of HTTP/2. From a network perspective there is currently limited support for analyzing HTTP/2 traffic. This paper will explore how best to analyze such traffic and discuss how the new version might change the future of network forensics.

Acknowledgements

The author would like to thank Chris Walker for his guidance and valuable inputs throughout the various stages of this paper, Christopher Pereda for many hours beta-testing the install scripts and environment described in this paper and Phil Hagen for timely feedback, support and validation of the research presented.
1. Introduction

The first publicly released version of Hypertext Transfer Protocol (HTTP), HTTP 1.0, was released in 1996. HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems (Berners-Lee, Fielding, & Frystyk, 1996). It is the basis of communication for the World Wide Web.

As a measure of its popularity, HTTP accounted for about 75% of Internet backbone traffic in a 1997 study (Krishnamurthy, Mogul, & Kristol, 1999, p. 2). The standards development of HTTP was coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C), culminating in the publication of a series of Requests for Comments (RFCs) ("User:Arefin/Internet Vs World wide web - Wikiversity," n.d.).

In HTTP/1.0, each resource request requires a separate network connection to the same server. HTTP/1.1 is a revision of the original HTTP protocol. “The first official HTTP/1.1 standard is defined in RFC 2068 which was officially released in January 1997, roughly six months after the publication of HTTP/1.0 (Berners-Lee, Fielding, & Frystyk, 1996). “Then, two and a half years later, in June of 1999, a number of improvements and updates were incorporated into the standard and were released as RFC 2616” (Fielding et al., 1999).

The diagram below describes the overall flow of a HTTP/1.1 connection.
The diagram above shows how a HTTP/1.1 connection reduces the latency as the client uses same connection to requests additional resources from the server. In the example above the same connection is used to transfer two additional resources, a CSS style sheet and JavaScript file. For simplicity purposes, only the connection setup (TCP handshake, step1-3) is shown and not the tear down. Version 1.1 of the HTTP protocol no longer requires the considerable expensive connection setup and tear down overhead for each resource.

1.1. Shortcomings of HTTP/1.1

Although the connection re-use for additional resources was a significant performance improvement over HTTP/1.0, there were still various other shortcomings impacting the overall latency.
1.1.1. Head-of-line blocking

HTTP/1.1 servers cannot make concurrent requests over the same connection. Hence, browsers often attempt to make many connections to speed up requests.

1.1.2. Many, expensive connections

Many modern browsers limit the number of open connections, as they are expensive to establish. At the same time, websites are continuously increasing their number of resources which creates significant performance issues.

1.1.3. Pipelining

The pipelining concept in HTTP/1.1 tried to address some of these performance limitations. “With pipelining multiple requests are sent on a single TCP connection without waiting on their responses” (Fielding, & Reschke, 2014). These requests still allow a single large of response to block other requests that follow. Unfortunately, many browsers do no support it because the intermediaries and servers fail to support it correctly.

1.2. Introduction of HTTP/2

There is emerging implementation experience and interest in a protocol that retains the semantics of HTTP without the legacy of HTTP/1.x message framing and syntax, which have been identified as hampering performance and encouraging misuse of the underlying transport.

The working group will produce a specification of a new expression of HTTP’s current semantics in ordered, bi-directional streams. As with HTTP/1.x, the primary target transport is TCP, but it should be possible to use other transports. (“HTTP/2 Charter,” 2012)
The IETF approved the proposed HTTP/2 standard in May 2015. The new standard is the first major update since releasing HTTP/1.1 almost ten years ago. To address the latency issues HTTP/2 adds support for request and response multiplexing, stream prioritization and compression of HTTP header fields, while maintaining the HTTP/1.1 syntax.

HTTP/2 does not modify the semantics of the protocol itself. Methods, status codes and header fields remain unmodified, minimizing impact on the application layer. It does modify how the data is formatted (framing layer) as well as how the data is transferred between the endpoints. As a result, existing applications can be delivered faster without modifications.

HTTP/2 enables a more efficient use of network resources and a reduced perception of latency by introducing header field compression and allowing multiple concurrent exchanges on the same connection as defined in RFC 7540. (Belsche, Peon, & Thomson, 2015). The RFC further notes that the protocol allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance (Belsche, Peon, & Thomson, 2015).

Explaining each of the HTTP/2 topics in detail is beyond the scope of this document. There are many excellent resources available like the online THE HTTP/2 BOOK (Stenberg, 2015) and William Chan’s blog post regarding HTTP/2 considerations and tradeoffs (Chan, 2014). However, we will describe the most important changes below.

1.2.1. HTTP/2 Multiplexing

To overcome some of the performance limitations as listed above, HTTP/2 implements multiplexing. Multiplexing allows the endpoints to send multiple HTTP requests and responses asynchronously via a single TCP connection.

Stefan Winkel, stefan@winkelsnet.com
In Figure 2, the diagram on the right indicates how clients can requests multiple resources from the server (step 6 and 7) without having to wait for the results reducing the overall time to render the page on the client. The connection remains open at the end of the transfer, which reduces the connection overhead. This is in contrast to HTTP/1.1 which closes the connection after each transfer. In real world scenarios clients often require 80-100 resources from a web server at the costs of 6-8 HTTP/1.1 connections, which makes the overall efficiency of a HTTP/2 connection more obvious.

### 1.2.2. Binary Protocol

HTTP/2 is a binary protocol, which is another performance optimization. Binary protocols are smaller in size and more efficient to parse. At the same time it is less error-prone and reduces the implementation complexity. The binary format should also minimize attacks like HTTP response splitting, which exploits the implementation complexity of the textual HTTP/1 protocol (Klein, 2006).
1.2.3. Compression of HTTP Header Fields / HPACK

To decrease latency further and reduce bandwidth HTTP/2 implements binary compression to reduce redundant header fields. The SPDY protocol (Belshe & Peon, n.d.) used the DEFLATE format (Deutsch, 1996) but this turned out to be vulnerable to the Compression Ratio Info-leak Made Easy (CRIME) attack (Constantin, 2012). As a result, HPACK was defined in RFC 7541 (Peon, & Ruellan, 2015), which addresses compression (with a rate between 30 and 80 percent) and limits vulnerability to known attacks like CRIME.

1.2.4. Request Prioritization and Server Push

A browser renders resources with different priorities. Conceptually a client would want lower priority resources (like images) to be downloaded later rather than be in-lined in the middle of a high priority resource (like HTML). For example, images are less important than CSS style sheets when rendering. HTTP/2 implements this concept through the notion of stream dependencies and weights. The server push feature allows the server to suggest to the client which resources it needs to render a page. This feature fundamentally changes the overall page loading semantics of the web and eliminates the need for intransitive hypertext links. Techniques like these require work and maintenance for the web developer. Hence, we will see this feature initially only on large/performance intensive websites/servers where the development teams are willing to put up with the extra maintenance burden to deploy these features.

1.3. HTTP/2 and Network Forensics

As outlined above there are significant changes in the HTTP/2 protocol and while this new version does not break HTTP/1 backward compatibility it changes completely the connection management layer. As a result, we will see that many network forensics tools do currently not support HTTP/2. The characteristics of the HTTP/2 protocol in particular due to the nature of de-facto encryption and its binary format indicate that it might take a long time for current network forensics tools to catch up, if possible at all.

Stefan Winkel, stefan@winkelsnet.com
2. Test Environment

There are various HTTP/2 implementations available. The wiki maintained in Github.com ("Implementations · http2/http2-spec Wiki · GitHub," 2015) does an excellent job at categorizes known implementations by client, server and tools. At the time of this writing there is still a very limited selection of HTTP/2 servers and tools available.

To test HTTP/2 tools and to discuss the results we created a test environment in which we spend many hours inspecting HTTP/2 traffic.

2.1. Test Environment Operating System

We use a Kali Linux distribution for our testing purposes. We use the Kali 2.0 virtual image, downloaded from Offensive Security. We update our Kali image and install various additional packages as listed in Appendix A1. To test HTTP/2 in various scenarios we will use the following H2O configurations:

2.1.1. HTTP/2 Server: H20

H2O is a fast and secure HTTP/2 server written in C by Kazuho Oku. H2O can be used as an insecure HTTP server and as well as a secure server supporting HTTPS (TLS/SSL) supporting HTTP 1.0, 1.1 and as well HTTP/2. You can download the latest code from Github.com at h2o/h2o. More details are available at https://h2o.example.net/. To minimize the overhead of building and configuration H2O, we use a Linux Docker image to test our H2O server.

2.1.2. What is Docker?

Docker is an open platform for building, shipping and running distributed applications. It gives programmers, development teams and operations engineers the common toolbox they need to take advantage of the distributed and networked nature of modern applications ("What is Docker?", 2015).

Stefan Winkel, stefan@winkelsnet.com
There is a lot of information on Docker containers on the internet. Two good examples are the articles ‘What is Docker and why is it so darn popular?’ (Vaughen-Nichols, 2014) and “Get to Know Docker” (“Get to know Docker, container technology out of the box,” n.d.). Docker is installed on our Kali distribution as part of updating the Kali image as listed in Appendix A1.

2.1.3. H2O on Docker

We will use the Revollat/H20 image as available on http://hub.docker.com. On the Kali system type execute the following command to start the H2O Docker image:

docker run -P -d --name h2o revollat/h2o

This command will start the H2O web server in the Docker container. The web server listens on port 80 for HTTP as well as 443 for HTTPS connections. After starting the H2O server, the client (Kali system) can access the server by connecting to https://127.0.0.1.xip.io: <port>. Replace <port> with the port that shows when running the docker ps command.

![Figure 3 - Example H2O ports in Docker container]

In the example above, we used port 32769 for http connections on port 80 and port 32768 for https connections (port 443). Our Docker image contains H2O version 1.2.1-alpha1 at the time of testing.

2.1.4. HTTP/2 Server: Apache

At the time of this writing, the Apache Software Foundation just released Apache (“httpd”) 2.4.17 with support for HTTP/2. We use the script `build_apache2_with_http2_support.sh` from LazyProgrammer.io (See Appendix A2) to deploy this new release of the Apache server with HTTP/2 support on our Kali box.

Stefan Winkel, stefan@winkelsnet.com
2.1.5. HTTP/2 Client: Curl

The Curl tool has HTTP/2 support in version 7.43.0. To test client functionality, we will mainly use Curl v7.45, obtained from http://curl.haxx.se/download.html. We install this version of Curl also on our Kali Linux host (See Appendix A3 for details). Once installed Curl has a ‘--http2’ flag that will use HTTP/2 when it can. The verbose option (-v) will show information about HTTP/2 usage.

2.2. Decrypting SSL/TLS sessions

Sally Vandeven does an excellent job in her paper ‘SSL/TLS: WHAT’S UNDER THE HOOD’ in section ‘Dissecting the Application data’ describing how to decrypt HTTPS sessions so we can inspect the decrypted HTTP traffic (Vandeven, 2013, p. 24).

With key exchange methods like RSA, we would only need the server’s private key to decrypt traffic. With newer key generation algorithms like Diffie-Hellman (DH) one would need the so-called session keys, generated by the client (browser). Some browsers will export those keys if told to by setting the SSLKEYLOGFILE environment variable. Chrome and Firefox use this variable to write the session keys to disk. These keys, written in the NSS Key Log Format (Combs, n.d.), can then be used in Wireshark, a free and open-source packet analyzer that captures and dissects network traffic ("Wireshark Protocol Analyzer," n.d.) to decode the encrypted traffic. If keys exchanges and generations methods only produce one-time use keys, they are called ephemeral. Perfect Forward Secrecy is defined when ephemeral keys are combined with DH key generation method (Shirey, 2007).

Different key exchange mechanisms will use different methods of decrypting the data. We will use these different mechanisms throughout this paper to decrypt HTTP/2 traffic. For simplicity, to decrypt the SSL traffic with a private server key, we specify a weaker cipher during the TLS handshake, where possible.

We used the Sade Blok’s tutorial ‘SSL TROUBLESHOOTING WITH WIRESHARK AND TSHARK’ for troubleshooting some decryption issues (Blok, 2009).
2.3. Wireshark and HTTP/2

Wireshark added support for HTTP/2 in version 1.12.0 by decoding HTTP/2 frames. We install the latest Wireshark release (2.0.0rc3 or later) from http://www.wireshark.org on our Kali host. Since most HTTP/2 traffic is sent over TLS, Wireshark will not be able to decrypt the packets by default without decryption. See Appendix A2 for details.

2.3.1. Decrypting HTTP traffic

In figure 4, we request the index.html page from the Apache server by using the Curl command curl https://127.0.0.1/index.html -k. We don’t specify the ‘–http2’ option and hence are using HTTP/1. The window on the right show we were able to decrypt the traffic. By selecting the ‘follow SSL stream’ option in Wireshark, we see that the server returned the message ‘It works!’ in the right bottom window.

![Figure 4 - Example of decrypted SSL traffic](image)

2.3.2. Decrypting H2O traffic in Wireshark

We will use the H2O web server to view some HTTP/2 requests and responses in further details. Similar as with the Curl example above, we will need to setup Wireshark to view the decrypted network traffic. We will populate the private key of the H2O server

Stefan Winkel, stefan@winkelsnet.com
in Wireshark by copying it from our Docker container where the H2O server is running (/etc/h2o/server.key) and save it to the local disk on the Kali system. To copy the private key from the H2O server, we started the Docker image with the `/bin/bash` option. This will open up a shell to the Docker image so we can copy out the private key found in the /etc/h2o directory:

```
docker run -ti --name h2o_config revollat/h2o /bin/bash
```

In Wireshark we then select: Preferences > Protocols > SSL > RSA keys list > Edit. Next we enter the specifics for our H2O server key. In our instance, the H2O server is running on host 172.17.0.9 on port 32274 for the HTTPS. See Appendix C for an example.

### 2.3.3. Decrypting Browser (Chrome/Firefox) traffic in Wireshark

In order to save the session keys to disk, we need to set the SSLKEYLOGFILE environment variable before starting the Firefox/Chrome browser. We run the following command: `./capture_firefox_h2o_traffic.sh`. Details of this script are in Appendix A5. After running the script we can decrypt the traffic in Wireshark by loading the Pre-Master-Secret log file in Wireshark by selecting Edit > Preferences > Protocols > SSL. Next specify the /tmp/keylog in the (Pre)-Master-Secret log filename text box. After that click OK and the traffic will be decoded immediately.

### 2.4. Directory Traversal and HTTP/2

On September 16th, 2015, a security vulnerability in the H2O web server was disclosed under CVE-2015-5638 “Directory traversal” (Yusuke, 2015). This bug allowed remote attackers to read arbitrary files on the H2O web server via a crafted URL. While the crafted URL was not published as part of the CVE we are able to construct an exploit relatively easily and we will use this exploit to analyze the HTTP/2 traffic between client and server. We will refer to this exploit as the H2O exploit in the following sections.
2.4.1. Encrypted H2O exploit

To see the H2O exploit in full swing we execute the following command on our Kali system against H2O server that is running in the Docker container:

```
curl --http2 https://127.0.0.1:32774/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/\%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/\%2e%2e/%2e%2e/%2e%2e/%2e%2e/\%2e%2e/etc/passwd -k
```

This returns the contents of the password file on the H2O server as seen below.

![Curl output H2O exploit](image)

In the Wireshark interface, as highlighted below, we see that TLS handshake starts with packet 4. The session completes with packet 22.
2.4.2. Decrypted H2O exploit with HTTP/2

Next we will look at the actual HTTP/2 session. To decrypt the SSL session we specify the AES256-SHA cipher. We have configured Wireshark with the private key of the H2O as explained above.

We run curl --http2 https://127.0.0.1:32774/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/passwd -k --ciphers 'AES256-SHA' -vvvv -l

Our Wireshark output shows the decrypted traffic with the HTTP2 packets listed in green (packets 10-16).
Right-click on packet 33 and select ‘Follow SSL Stream’. This shows the transfer of the contents of password files to the client because of running the exploit.

Figure 7 - H2O Decrypted Directory Traversal

Figure 8 - Follow SSL Stream with HTTP/2
We can also inspect packet nr 15 listed above and look at the TCP stream 1 of type Data. By expanding the Data header field, we see the contents of the password file displayed.

![Figure 9 - HTTP/2 DATA stream showing password file content](image)

2.4.3. **Wireshark HTTP/2 header fields limitations**

While Wireshark does parse HTTP/2 traffic in the latest revisions (release 2.0 came out mid November 2015, the previous release 1.12.8 was just a month earlier), it
does not yet seem to be exposing the various header fields with the same granularity as we see with HTTP/1.x. We see still lots of ‘http.header.name’ and ‘http.header.value’ fields, which is a huge gap in visibility and capability when analyzing HTTP/2 traffic. For example the ‘http.user_agent’ field from HTTP/1.x does not have a corresponding field for HTTP/2. We experience this limitation as we try to search the user agent string ‘curl/7.45.0’ as seen in packet 14 (See Appendix E).

2.4.4. Network flow and Protocol statistics

The output of the Wireshark statistics in the screen capture below shows the protocol hierarchy statistics, the endpoint statistics, as well as the conversation details between the Curl and H2O endpoints. Statistics shows that traffic is encrypted (TLS/SSL) between the two endpoints, but we do not know anything about the underlying protocol (HTTP2). Since the endpoints reuse the SSL connection with HTTP/2 it is even more difficult to identify any abnormal behavior based upon packet size. Hence identifying that the client just exploited a directory traversal bug on the H2O web server becomes even more difficult.

On the other hand, if we were able to decrypt the traffic protocol Wireshark statistics show details of the HTTP/2 packets as seen in the right bottom pane below.

Stefan Winkel, stefan@winkelsnet.com
We leave as an exercise for our reader to write a Wireshark display filter that identifies the contents of the password file transferred to the client.

2.5. Tshark, TCPDump, SSLDump and HTTP/2

One of the benefits of using Wireshark’s command line interface, tshark, is that it can be used to script packet analysis. The tshark stats, specified with the ‘-z http2,tree‘ flag shows there are 2 HTTP/2 header packets and only 1 data packet being sent.
We inspect the headers by using the display filter ‘http2.type == 1’ and see that packet 14 contains the offending GET request in the decompressed headers.

```
/opt/wireshark-2.0.0rc3/tshark -r /opt/pcap/h2o_directory_traversal_http2_ssl_decrypted.pcapng -Y http2.type==1 -O http2.data -x
```

![Figure 13- tshark showing the decompressed HTTP/2 header with the offending request](image)

If we specify the flag ‘-Y http2.data.data’ in the tshark command we see that the H2O server is returning the contents of the password file in the data frame (frame 15)

Stefan Winkel, stefan@winkelsnet.com
SSLDump

There has not been any significant update to SSLDump for almost over a decade but it still displays useful information regarding the initial SSL negotiation phase. The tool will not be able to decrypt the application data if ephemeral cipher suites, like Diffie-Hellman (DHE) or RSA ephemeral are used during the key negotiation part of the SSL handshake. Steven Iveson has an interesting blog post regarding SSLDump at http://packetpushers.net/using-ssldump-decode-ssl-tls-packets.
2.5.2. TCPDump

We have used TCPDump as described in the capture_firefox_h2o_traffic.sh script (see Appendix A5), to capture the HTTP/2 traffic from the web browser before displaying in Wireshark. Reading HTTP/2 capture files (pcap/pcapng) back with TCPDump would require us to decrypt the packages with a tool like SSLDump before we could display them. We decrypted the capture in Wireshark and saved the output to a pcap file before we tried to list the contents in TCPDump as shown in Appendix G.

As we saw in Figure 7, the Wireshark output shows that this packet contains the password file that got transferred, but while trying to dump the packet data in ASCII or Hex format we notice that that we are unable to read the contents as expected. We expected to see the same output as shown in Figure 14, showing the password file contents.

![Figure 15 - Hexadecimal TCPDump HTTP/2 packet](image)

2.6. Snort, Bro, and HTTP/2

At the time of this writing Snort, the open-source IDS/IPS tool does not support HTTP/2 inspection. There have been efforts earlier this year to develop a new object oriented HTTP inspector that could support HTTP/2 as the new Snort 3.0 architecture, but there has not been any update in the last six months or so. Bro, an open source UNIX based network monitoring framework, neither supports HTTP/2 as of yet.

Stefan Winkel, stefan@winkelsnet.com
2.7. Web browser support for HTTP/2

Reproducing the H2O exploit with different browsers helps us understand that HTTP/2 support varies by different implementations and/or vendors. Analysis of HTTP/2 browser support is beyond the scope of this paper but could provide an interesting follow-up research topic (Appendix D).

3. Continuing the HTTP/2 Journey

3.1. Many HTTP attack vectors

There are many HTTP attack vectors like HTTP Parameter tampering/pollution (SecureComm & Rajarajan, 2012, p. 415), request/response splitting, file download injections (Williams/Aspect Security, 2008, p. 3) and request/response smuggling (Klein, 2006). While some of these vectors are more protocol specific, like HTTP response splitting and smuggling, based on the textual aspect of the HTTP/1.1, they will continue to exist as HTTP/1.1 most likely will be around for a while.

As HTTP/2 deployment increases, besides the existing HTTP/1 attacks, we will see an increase in HTTP/2 protocol attacks, like the recent compression issues in the Breach and Crime exploits (Prado, Harris and Gluck, 2013).

Since there will be implementations that will support the different versions of the HTTP protocol, both HTTP/1.x and HTTP/2, consequently we will see more cross-protocol attacks. In a cross-protocol attack, an adversary causes a client to initiate a transaction in one protocol toward a server that understands a different protocol (Belsche, Peon, & Thomson, 2015). The adversary might be able to cause the transaction to appear as a valid transaction in the second protocol. In a web server context, an adversary could exploit this to interact with poorly protected servers in private networks.

We have mentioned new HTTP/2 protocol specific security issues like the CRIME exploit but even older attacks like the Directory Traversal attack demonstrated in...

Stefan Winkel, stefan@winkelsnet.com
this paper will not go away in particular as we see an uptake in the new web servers with HTTP/2 implementations like the H2O server discussed in this paper.

Ilya Griorik describes the new binary, length-prefixed framing layer format in his book *HIGH PERFORMANCE BROWSING NETWORKING* (Grigorik, 2013, Chapter 12). In the section below we take a closer look at the binary aspect of the protocol and the complexity that arises with network forensics of the new version of the protocol.

### 3.2. Binary Framing in detail

Whereas HTTP/1x uses variable length fields, HTTP/2 uses fixed-length (9 byte) fields only and offers a more compact representation than the newline-delimited plaintext HTTP/1 protocol and is both simpler and faster to encode/decode and more efficient to process. These frames, which have a relatively small overhead, are the basis for the communication between client and server. There are two different types of frames, control or header frames and data frames. As Griorik describes, frames from different streams might be interleaved and then reassembled via the embedded stream identifier in the header of each frame. He concludes that the communication between client and server is an exchange of binary encoded frames, which are mapped to messages that belong to a particular stream where streams can be multiplexed within a single TCP connection (Grigorik, 2013).

While the ASCII protocol in HTTP/1 is easier to inspect, it is more difficult to implement correctly. Issues like sequence termination and optional whitespace, while often used to improve readability, can make it harder to distinguish the protocol from the payload. This has lead to exploits like HTTP response splitting and smuggling. RFC 7230 tried to address some of these issues by disallowing whitespace between header field name and colon. A binary protocol, as introduced in HTTP/2, allows for more robustness, less implementation discrepancies while at the same time allowing for better performance because of the more compact format.

Stefan Winkel, stefan@winkelsnet.com
Because of the binary format, you would need tools to inspect and debug HTTP/2 traffic. According to Grigorik, you would need the same tools to inspect the encrypted TLS flows, which are also relying on binary framing. See section “TLS Record Protocol” of “HIGH PERFORMANCE BROWSER NETWORKING” (Grigorik, 2013, Chapter 12)

3.2.1. Complexity of HTTP/2 with forensic network analysis

Grigorik is correct that inspecting and debugging HTTP/2 traffic is not more complex than to inspect the encrypted TLS flows. When debugging the protocol one can safely assume that the user controls the endpoints and hence can decrypt the TLS session. While developers in general own the endpoint and hence are in a better position to decrypt the TLS sessions for debugging purposes, the forensic network investigator on the other hand, does often not have access to the decryption keys.

While HTTP/2 does not require encryption, most client implementations only support HTTP/2 over TLS, making encryption a de-facto requirement. Besides Firefox and Chrome which require HTTP/2 to be used over an encrypted connection, now also Apple as well as Microsoft’s HTTP/2 implementations will only support encrypted HTTP/2.

This de-facto standard will increase the complexity of network forensics, as more traffic will start to be encrypted. So the issue is not that HTTP/2 is a binary protocol but that its deployment is combined with a push for stronger security. It is this push for stronger security as a side effect of the adoption of HTTP/2 that increases the complexity of network forensics as we have learned in the sections above.

3.3. Trusted Proxies and Gateways

In a proposal submitted early February 2014 to the IETF, called “EXPLICITLY TRUSTED PROXY IN HTTP/2” (Loreto et al., 2014), the authors propose to use different ALPN extensions for https (“h2://”) vs. http (“htc://”) resources. The ALPN htc:// extension , c meaning clear, in which case it would use HTTP/2 with TLS, but intermediaries might be decrypt the traffic en route, which requires implicit user consent.

Stefan Winkel, stefan@winkelsnet.com
A discussion in the SANS ISC InfoSec forums (McRee, 2014) describes the pros and cons of this proposal and is summarized below.

Besides limitation with network forensics, TLS encryption hides knowledge from intermediaries and reduces efficiencies in both transport and caching, which makes things more difficult for internet service providers (ISPs). In this new mode clients and servers could use to upgrade to TLS in the absence of a digital certificate identifying the remote server. It would allow carriers (ISPs) to provide caching to give faster and more affordable access to users in locations with limited bandwidth. Since more traffic would be encrypted it would make it more expensive to analyze captured traffic on a giant scale (McRee, 2014).

This new mode of HTTP/2 operation is sometimes referred to as opportunistic encryption. It is not an official term and has many meanings in different contexts. For example, in RFC 4322 (Richardson, & Redelmeier, 2005) it is defined as encryption without a peer-specific arrangement while in RFC 5386 (Williams & Richardson, 2008) it is used to mean encryption without authentication.

Brad Hill states on his blog “"One thing this whole episode has finally convinced me of is that “opportunistic encryption” is a bad idea. I was always dubious that “increasing the cost” of dragnet surveillance was a meaningful goal (those adversaries have plenty of money and other means) and now I’m convinced that trying to do so will do more harm than good. I watched way too many extremely educated and sophisticated engineers and tech press get up-in-arms about this proxy proposal, as if the “encryption” it threatened provided any real value at all. “Opportunistic encryption” means well, but it is clearly, if unintentionally, crypto snake-oil, providing a very false sense of security to users, server operators and network engineers. For that reason, I think it should go, to make room for the stuff that actually works." (Hill, 2014).

The assumption is that the authors intend the proposal to be for ISPs, as enterprises already should deploy man-in-the-middle (MITM) proxies to inspect

Stefan Winkel, stefan@winkelsnet.com
outbound HTTPS to implement a robust data loss prevention system. It is worth noting that there are various legal (privacy) concerns with such MITM implementations as each jurisdiction has unique constraints on collecting network traffic.

One approach that some security vendors seem to be taken is to deliver a HTTP gateway that would enable a mix of HTTP 2.0, HTTP 1.x and SPDY on the outside while HTTP/1 on the inside (server side). This could mean that on the inside just plain HTTP is supported without encryption.

4. Conclusion

We will see a combination of HTTP/1.x and HTTP/2 traffic across the web for the foreseeable future. As a result we will see an increase in security vulnerabilities, either because of the new protocol and/or because of new implementations. As outlined above, many network forensics tools do currently not support HTTP/2. The characteristics of HTTP/2 (binary, compression, encryption), in particular due to the nature of de-facto encryption in browsers, catalyses the need for correlation of network forensics with end-point forensics (e.g. mobile/memory). Mobile applications are likely to benefit most from the performance enhancements provided by HTTP/2 as clients can be ‘forced’ to upgrade with minimal disruption. Since roundtrips are even more costly, and the uplink bandwidth is even more constrained on the mobile network this is most likely the area where we will see HTTP/2 deployed more broadly. As frequently identified in the forensic process, a comprehensive approach is necessary to conduct a thorough investigation. Heather Mahalik and Phil Hagen have put together an excellent presentation “SMARTPHONE AND NETWORK FORENSICS GOES TOGETHER LIKE PEAS AND CARROTS” (Hagen & Mahalik, 2015). Also, logging aggregation solutions like the ELK stack (Elasticsearch, Logstash, Kibana) as presented in SANS FOR572.4, ADVANCED NETWORK FORENSICS will become more important for forensic investigators as the deployment of HTTP/2 increases (Hagen & Oldham, 2015, p136).
References


Stefan Winkel, stefan@winkelsnet.com


Stefan Winkel, stefan@winkelsnet.com


Stefan Winkel, stefan@winkelsnet.com


Appendix A Scripts & Commands

Appendix A1
Update Kali Image

To test the different scenarios used in this paper we update our Kali image and install various additional packages by running the following command:

```bash
$ sudo apt-get update && apt-get install make binutils autoconf automake autotools-dev libtool libtool-bin pkg-config zlib1g-dev libcunit1-dev libssl-dev libxml2-dev libev-dev libevent-dev libjansson-dev libjemalloc-dev python3.4-dev bison libpcap-dev libgtk-3-dev docker.io docker nghttp2 libnghttp2-dev
```

Source: [http://pastebin.com/NkrsdBqs](http://pastebin.com/NkrsdBqs)
Appendix A2
build_apache2_with_http2_support.sh

#!/bin/bash
# Source: https://lazyprogrammer.io/entry/lazy-build-from-source-apache2-with-http-2-support
set -e

APACHE_INSTALL_DIR="/home/user/apache2"
APACHE_VERSION="httpd-2.4.17"
APACHE_SRC_FILE="http://www.us.apache.org/dist/httpd/${APACHE_VERSION}.tar.gz"
APACHE_DEPS="apache2-dev libapr1-dev libaprutil1-dev libpcre3 libpcre3-dev lynx"
NGHTTP2_DEPS="autoconf automake autotools-dev libtool pkg-config zlib1g-dev libcunit1-dev libbssl-dev libxml2-dev libevent-dev make binutils libjemalloc-dev libjemalloc-dev cython python3.4-dev python-setuptools"

# Install required dependencies
sudo apt-get install -y $APACHE_DEPS

# Download apache2 sources
wget $APACHE_SRC_FILE
# Unarchive the source files
tar -xzvf "${APACHE_VERSION}.tar.gz"
pushd $APACHE_VERSION
# Build from source nghttp2
git clone https://github.com/tatsuhiro-t/nghttp2.git
pushd nghttp2
sudo apt-get install -y $NGHTTP2_DEPS
autoreconf -i
automake
autoconf
sudo ./configure --prefix=/usr/local
sudo make
sudo make install
popd

# Build apache2
sudo ./configure --enable-http2 --prefix=$APACHE_INSTALL_DIR
sudo make
sudo make install

# now we should have $APACHE_INSTALL_DIR/bin/httpd and apachectl binaries
sudo chown -R www-data.www-data 
"${APACHE_INSTALL_DIR}/htdocs"
sudo "${APACHE_INSTALL_DIR}/bin/apachectl" -k start
# now if you navigate to http://localhost you should see "It works!"

# enabling http/2 protocol

Stefan Winkel, stefan@winkelsnet.com
sudo tee -a "${APACHE_INSTALL_DIR}/conf/httpd.conf" <<DELIM
LoadModule http2_module modules/mod_http2.so

<IfModule http2_module>
    LogLevel http2:info
</IfModule>

Protocols h2c http/1.1
DELIM

sudo "${APACHE_INSTALL_DIR}/bin/apachectl" -k restart

Source: [http://pastebin.com/kJe5XRf1](http://pastebin.com/kJe5XRf1)

After executing the command above to install and start the Apache server we need to run the following to decrypt the SSL/TLS traffic from the Apache webservers.

1. In Wireshark we select: Preferences > Protocols > SSL > RSA keys list > Edit. We add /home/user/apache2/conf/ssl/server.key as specified in Appendix 0
2. We update the /home/user/apache2/conf/extra/httpd-ssl.conf and added the following line at the end: ‘SSLCipherSuite AES256-SHA256’
3. We restart the apache server by running: 
   /home/user/apache2/bin/apachectl restart
Appendix A3

Building and installing Curl 7.46 with HTTP/2 support

To build curl from sources you will need OpenSSL, zlib, nghttp2 and libev. At the time of this writing we built Curl using the following commands:

```
$ curl -LO http://dist.schmorp.de/libev/libev-4.22.tar.gz
$ tar zvxf libev-4.22.tar.gz
$ cd libev-4.22
$ ./configure
$ make
$ sudo make install

$ curl -LO https://www.openssl.org/source/openssl-1.0.2e.tar.gz
$ tar zxvf openssl-1.0.2e.tar.gz
$ cd openssl-1.0.2e
$ ./config shared zlib-dynamic
$ make && make test
$ sudo make install

$ curl -LO http://zlib.net/zlib-1.2.8.tar.gz
$ tar zvxf zlib-1.2.8.tar.gz
$ cd zlib-1.2.8
$ ./configure
$ make && make test
$ sudo make install

$ curl -LO https://github.com/tatsuhiro-t/nghttp2/releases/download/v1.5.0/nghttp2-1.5.0.tar.gz
$ tar zvxf nghttp2-1.5.0.tar.gz
$ cd nghttp2-1.5.0
$ OPENSSL_CFLAGS="-I/usr/local/ssl/include" OPENSSL_LIBS="-L/usr/local/ssl/lib -lssl -lcrypto -ldl" ./configure
$ make
$ sudo make install
```

Stefan Winkel, stefan@winkelsnet.com
$ curl -LO http://curl.haxx.se/download/curl-7.46.0.tar.gz
$ tar zxvf curl-7.46.0.tar.gz
$ cd curl-7.46.0
$ ./configure
$ make && make test
$ sudo make install
$ sudo ldconfig

Source: [http://pastebin.com/BdFsE4E8](http://pastebin.com/BdFsE4E8)

To verify curl successfully built, execute the following command: `curl -V`. You should see HTTP2 listed under ‘Features:’ as shown below.

Alternatively you can run curl from a Docker container like ‘centminmod/docker-ubuntu-nghttp2’ as available on DockerHub.

# docker pull centminmod/docker-ubuntu-nghttp2
# docker run -ti --name nghttp centminmod/docker-ubuntu-nghttp2
/bin/bash
# curl -V

Stefan Winkel, stefan@winkelsnet.com
Appendix A4
Building and installing Wireshark-2.0.0

$ wget --no-check-certificate
https://www.wireshark.org/download/src/all-versions/wireshark-
2.0.0.tar.bz2
$ bzip2 -d wireshark-2.0.0.tar.bz2
$ tar xf Wireshark-2.0.0
$ cd wireshark-2.0.0
$ typeset -x PATH=/opt/curl-7.45.0:$PATH
$ ./autogen.sh
$ ./configure
$ make
$ sudo make install
$ sudo ldconfig

Source: http://pastebin.com/4P2wJHSG

Stefan Winkel, stefan@winkelsnet.com
Appendix A5

capture_firefox_h2o_traffic.sh

#!/bin/sh

LOGFILE=/tmp/keylog
PCAPFILE=/tmp/tcpdump.pcap

# Save the session keys to disk
export SSLKEYLOGFILE=$LOGFILE

# start the capture on the Docker interface
INTERFACE=docker0 && $(which tcpdump) -i ${INTERFACE} -s0 -XX -w $PCAPFILE port 443 &
TCPDUMP_PID=$! && echo "tcpdump running on $TCPDUMP_PID"

# start Firefox and point it to our H2O server in the Docker Container
/opt/firefox/firefox https://127.0.0.1:32774
BROWSER_PID=$! && echo "Firefox running on $BROWSER_PID"

# Exit the TCPDUMP process
kill -9 $TCPDUMP_PID

# Open up Wireshark and view the results
/opt/wireshark-2.0.0rc3/wireshark -r $PCAPFILE

Stefan Winkel, stefan@winkelsnet.com
Appendix B
Client Output H2O exploit

Figure 16 - Curl output with debug turned on
Appendix C

RSA Key Lists in Wireshark for H2O server

Figure 17 - SSL Keylist for debugging SSL sessions
Appendix D

Web browser support for HTTP/2

We experienced mixed results when testing the H2O exploit with various web browsers. A quick check of the default Iceweasel browser (v31.8) on the Kali system indicates that it does not support HTTP/2. Even with the latest, nightly Firefox (41.0.2) we do not get the expected results. Initially, we run into decoding issues with %s symbol part of our crafted exploit URL, but even worse is that Wireshark output of the decrypted session shows that HTTP/1.1 was used instead of HTTP/2, even with all the HTTP/2 configuration variables turned on.

![Figure 18 – Follow SSL Stream / Firefox](image)

We expected Firefox to send an Upgrade header (‘Upgrade: h2c’) as shown in Appendix E, indicating it is capable of handling HTTP/2 requests.

We have better results with Chrome browser via the Net Internals console. You can access this console by using the chrome://net-internals/ URL and in the drop-down menu, select HTTP/2. In the list of the HTTP/2 session is a link to view the current live sessions. By selecting a particular HTTP/2 session, you can see the raw output of the HTTP/2 streams and frames as seen below.

Stefan Winkel, stefan@winkelsnet.com
Reproducing the H2O exploit with different browsers as described here, helps us understand that HTTP/2 support varies by different implementations and/or vendors. Further analysis of HTTP/2 browser support is beyond the scope of this paper but could provide an interesting follow-up research topic.
Appendix E
HTTP/2 User-Agent String

Figure 20 - Packet 14 shows the user-agent string

Stefan Winkel, stefan@winkelsnet.com
Appendix F
Curl example with HTTP/2 upgrade request

<table>
<thead>
<tr>
<th>Stream Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET / HTTP/1.1</td>
</tr>
<tr>
<td>Host: 172.17.0.2</td>
</tr>
<tr>
<td>User-Agent: curl/7.46.0-DEV</td>
</tr>
<tr>
<td>Accept: <em>/</em></td>
</tr>
<tr>
<td>Connection: Upgrade, HTTP2-Settings</td>
</tr>
<tr>
<td>Upgrade: h2c</td>
</tr>
<tr>
<td>HTTP2-Settings: AAMAAABkAAQAAP</td>
</tr>
<tr>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>Date: Fri, 13 Nov 2015 19:24:25 GMT</td>
</tr>
<tr>
<td>Server: Apache/2.4.17 (Unix)</td>
</tr>
<tr>
<td>Last-Modified: Tue, 27 Oct 2015 02:20:19 GMT</td>
</tr>
<tr>
<td>ETag: &quot;2b60-5230cb81c5ac0&quot;</td>
</tr>
<tr>
<td>Accept-Ranges: bytes</td>
</tr>
<tr>
<td>Content-Length: 11104</td>
</tr>
<tr>
<td>Content-Type: text/html</td>
</tr>
</tbody>
</table>

Figure 21 - Curl example with HTTP/2 upgrade request
Appendix G
TCPDump of decrypted HTTP/2 packet

Figure 22 - TCPDump of decrypted HTTP/2 packets
<table>
<thead>
<tr>
<th>Course Name</th>
<th>Location</th>
<th>Dates</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS Munich July 2019</td>
<td>Munich, DE</td>
<td>Jul 01, 2019 - Jul 06, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SEC450 Security Ops-Analysis Beta 1</td>
<td>Crystal City, VAUS</td>
<td>Jul 08, 2019 - Jul 13, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Pittsburgh 2019</td>
<td>Pittsburgh, PAUS</td>
<td>Jul 08, 2019 - Jul 13, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>DFIR Summit &amp; Training 2019</td>
<td>Austin, TXUS</td>
<td>Jul 25, 2019 - Aug 01, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Riyadh July 2019</td>
<td>Riyadh, SA</td>
<td>Jul 28, 2019 - Aug 01, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Boston Summer 2019</td>
<td>Boston, MAUS</td>
<td>Jul 29, 2019 - Aug 03, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS San Jose 2019</td>
<td>San Jose, CAUS</td>
<td>Aug 12, 2019 - Aug 17, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Prague August 2019</td>
<td>Prague, CZ</td>
<td>Aug 12, 2019 - Aug 17, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Minneapolis 2019</td>
<td>Minneapolis, MNUS</td>
<td>Aug 12, 2019 - Aug 17, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Virginia Beach 2019</td>
<td>Virginia Beach, VAUS</td>
<td>Aug 19, 2019 - Aug 30, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Amsterdam August 2019</td>
<td>Amsterdam, NL</td>
<td>Aug 19, 2019 - Aug 24, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>