AKAMAI WHITE PAPER

Passive Fingerprinting of HTTP/2 Clients

Ory Segal, Sr. Director, Threat Research, Akamai
Aharon Fridman, Sr. Security Researcher, Akamai
Elad Shuster, Security Data Analyst, Akamai

Akamai

1.0 OVERVIEW: HTTP/2

HTTP/2 is the second major version of the HTTP protocol. It changes the way HTTP is transferred "on the wire" by introducing a full binary protocol that is made up of TCP connections, streams, and frames, rather than a plain-text protocol. Such a fundamental change from HTTP/1.x to HTTP/2 means that client-side and server-side implementations have to incorporate completely new code in order to support new HTTP/2 features. This introduces nuances in protocol implementations, which, in return, might be used to passively fingerprint web clients.

An example of HTTP/2 implementation nuances can be found in configuration parameters that are sent within the SETTINGS frames during connection initialization. These features describe the characteristics of the sending peer, and may be used for client fingerprinting.

The table below demonstrates some of the nuances and differences in features between selected HTTP/2 clients:

Examples for HTTP/2 Configuration Parameters

User-Agent	MAX CONCURRENT STREAMS	HEADER TABLE SIZE	MAX HEADER LIST SIZE	MAX FRAME SIZE	INITIAL WINDOW SIZE	ENABLE PUSH
Mozilla/5.0 (Android 6.0; Mobile; rv:52.0) Gecko/52.0 Firefox/52.0	[]	['4096']	[]	['16384']	['32768']	[]
Mozilla/5.0 (Android 6.0.1; Tablet; rv:47.0) Gecko/47.0 Firefox/47.0	[]	[]	[]	['16384']	['32768']	[]
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729; McAfee)	['1024']	[]	[]	[]	['10485760']	[]
Mozilla/5.0 (Linux; Android 7.1; Pixel XL...	['100']	['4096']	['131072']	['16384']	['163840']	['0']

The HTTP/2 RFC (the IETF Request for Comments document) mentions the possibility of passive fingerprinting in Section 10.8 — Privacy Considerations:

"Several characteristics of HTTP/2 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the manner in which flow-control windows are managed, the way priorities are allocated to streams, the timing of reactions to stimulus, and the handling of any features that are controlled by settings."

According to this HTTP/2 Adoption site, there are approximately 241,000 domains that announced support for HTTP/2 as of November 16, 2016. Supporters include Google, Amazon, Blogspot, Wikipedia, and WordPress.

Since this is a newly adopted technology, the number of known server and client implementations for HTTP/2 is rather low compared to the number of HTTP/1.x implementations and programming libraries. The full list of HTTP/2 implementations can be found in the HTTP Working Group dedicated HTTP/2 website. Akamai has been among the first to implement HTTP/2, allowing each client to communicate with the Akamai network over HTTP/2.

This white paper covers Akamai's Threat Research team's investigation of the possibility of passively fingerprinting HTTP/2 clients based on unique implementation features. The paper also proposes a format for passive HTTP/2 fingerprints, as well as a few examples of unique fingerprints belonging to common clients and implementations.

2.0 PASSIVE CLIENT FINGERPRINTING

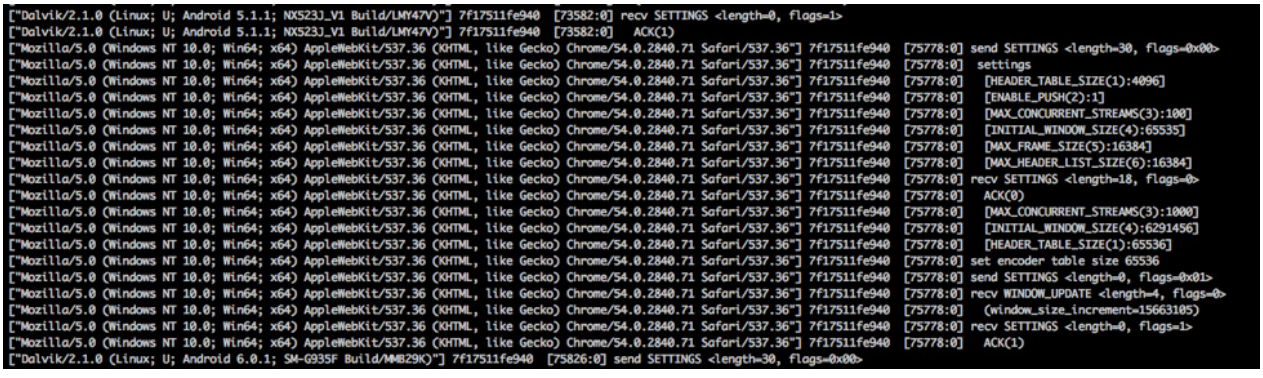
Passive client fingerprinting refers to the passive collection of attributes from a network-connecting client or server. Attributes may be collected from the transport, session, or application layer (e.g. TCP properties, TLS capabilities, or HTTP implementation characteristics). These attributes can be used to deduce information about the client, such as operating system (type and version), system up-time, or, in some cases, browser type. In addition, a client's passive fingerprint can be used to add uniqueness/entropy to the client's online identity, specifically when using a multi-layered device fingerprinting approach. Currently, there are three known and commonly used approaches to passively fingerprint web clients:

1. **TCP/IP Fingerprint —** described in detail in the p0f library documentation
2. **TLS fingerprint —** as described in the following paper
3. **HTTP Fingerprint —** described in detail in the p0f library documentation

3.0 RESEARCH DATA CORPUS

The data for this research was collected from Akamai's Edge servers, which regularly handle millions of HTTP/2 requests daily. For research purposes, granular logging levels were applied in order to log the full details of the HTTP/2 frames and streams.

This research is based on more than 10 million HTTP/2 connections collected from multiple Edge servers across the Akamai network. The data set for the research was anonymized and only contained information about the HTTP "User-Agent" header value and the HTTP/2 logged events and attributes.



HTTP/2 Fingerprint Features

The first step was to identify possible sources of fingerprint entropy, if any, in the HTTP/2 protocol. The data was searched for flows or messages in the protocol where different clients exposed a consistent unique behavior that could be used for fingerprinting purposes.

The data analysis yielded a consistent variation in the following protocol flows:

1. SETTINGS frame
2. WINDOW_UPDATE frame
3. PRIORITY frame

SETTINGS Frame

Before any data is exchanged, the HTTP/2 SETTINGS frame is sent from both client to server and server to client during the initial connection phase. The frame is defined by RFC 7540 as follows:

"The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. The SETTINGS frame is also used to acknowledge the receipt of those parameters. Individually, a SETTINGS parameter can also be referred to as a "setting."

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which are used by the receiving peer. Different values for the same parameter can be advertised by each peer. For example, a client might set a high initial flow-control window, whereas a server might set a lower value to conserve resources.

A SETTINGS frame MUST be sent by both endpoints at the start of a connection and MAY be sent at any other time by either endpoint over the lifetime of the connection. Implementations MUST support all of the parameters defined by this specification."

The following SETTINGS parameters are defined by RFC 7540:

Parameter Name	Scope
SETTINGS_HEADER_TABLE_SIZE (0x1)	Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets.
SETTINGS_ENABLE_PUSH (0x2)	This setting can be used to disable server push (Section 8.2).
SETTINGS_MAX_CONCURRENT_STREAMS (0x3)	Indicates the maximum number of concurrent streams that the sender will allow.
SETTINGS_INITIAL_WINDOW_SIZE (0x4)	Indicates the sender's initial window size (in octets) for stream-level flow control. The initial value is 216-1 (65,535) octets.
SETTINGS_MAX_FRAME_SIZE (0x5)	Indicates the size of the largest frame payload that the sender is willing to receive, in octets.
SETTINGS_MAX_HEADER_LIST_SIZE (0x6)	This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets.

We looked into the SETTINGS frames sent from client to server, and we found that different clients differ in:

- The SETTINGS parameters they choose to send
- The order by which the SETTINGS parameters are sent
- The values they set for the SETTINGS parameters

WINDOW_UPDATE Frame

The WINDOW_UPDATE frame is sent in order to notify the other endpoint of an increment in the window size. It is defined by RFC 7540 as follows:

"The WINDOW_UPDATE frame (type=0x8) is used to implement flow control; see Section 5.2 for an overview... When an HTTP/2 connection is first established, new streams are created with an initial flow-control window size of 65,535 octets. The connection flow-control window is also 65,535 octets. Both endpoints can adjust the initial window size for new streams by including a value for SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame that forms part of the connection preface. The connection flow-control window can only be changed using WINDOW_UPDATE frames."

We observed that almost all the connecting clients send the WINDOW_UPDATE frame after the SETTINGS frame. We discovered that the increment value in the WINDOW_UPDATE frame consistently differs from client to client, as a result of different HTTP/2 client implementations.

PRIORITY for Reserved Streams Flow

The PRIORITY frame is sent in order to set a priority of any given stream. It is defined by the RFC as follows:

"The PRIORITY frame (type=0x2) specifies the sender-advised priority of a stream (Section 5.3). It can be sent in any stream state, including idle or closed streams....The PRIORITY frame can be sent on a stream in any state, though it cannot be sent between consecutive frames that comprise a single header block (Section 4.3). Note that this frame could arrive after processing or frame sending has completed, which would cause it to have no effect on the identified stream. For a stream that is in the "half-closed (remote)" or "closed" state, this frame can only affect processing of the identified stream and its dependent streams; it does not affect frame transmission on that stream."

We observed that certain clients, right after the connection phase, send the server several PRIORITY frames, all for streams that have not been opened yet. We can take the stream identifiers that were opened and use them as a part of the fingerprint. For example, Firefox browsers tend to demonstrate such a behavior. Looking at the Firefox HTTP/2 implementation code in Http2Session.cpp, we spotted the following relevant comment:

```
// The Hello is comprised of
// 1] 24 octets of magic, which are designed to
// flush out silent but broken intermediaries
// 2] a settings frame which sets a small flow control window for pushes
// 3] a window update frame which creates a large session flow control window
// 4] 5 priority frames for streams which will never be opened with headers
//    these streams (3, 5, 7, 9, b) build a dependency tree that all other
//    streams will be direct leaves of.
```

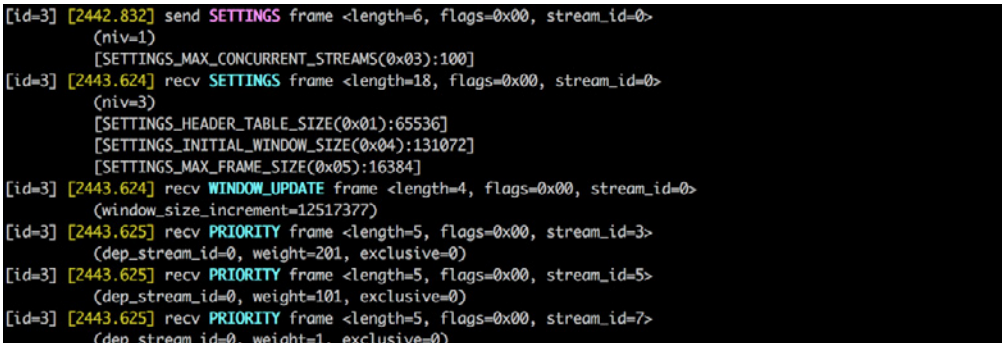


Image 2: Example Flow with PRIORITY for Reserved Streams

In addition to the stream identifier, the PRIORITY frame also includes the following elements:

- • A single-bit flag indicating that stream dependency is exclusive
  • A 31-bit stream identifier for the stream that this stream depends on
  • The weight assigned to that stream, which is defined by the RFC as an unsigned 8-bit integer representing a priority weight for the stream (values are between 1 and 256)

This information can also be used in the fingerprint.

4.0 Passive HTTP/2 Fingerprint — Suggested Format

Combining the above fingerprinting factors, we suggest the following HTTP/2 Fingerprint format:

`S[;]|WU|P[,]#`

Where `S[...]` stands for a SETTINGS parameter and its value in the form of Key:Value. Multiple settings are concatenated using a semicolon (;) according to the order of their appearance.

`WU` stands for the `WINDOW_UPDATE` increment size — '00' if the frame is not present

`P[,]` = A tuple representing stream priority information in the following format:
StreamID:Exclusivity_Bit:Dependant_StreamID:Weight

Multiple priority frames are concatenated by a comma (,). If this feature does not exist, the value should be '0'.

Fingerprint Example:

Let's look at the flow from Image 2 above. We see the web server received a SETTINGS frame from the client, with the following parameters:

Parameter Name	Parameter Value
SETTINGS_HEADER_TABLE_SIZE (0x1)	65536
SETTINGS_INITIAL_WINDOW_SIZE (0x4)	131072
SETTINGS_MAX_FRAME_SIZE (0x5)	16384

Hence, the settings fingerprint will be denoted as: 1:65536;4:131072;5:16384.

Next, a window update was sent, with a value of 12517377 hence WU = 12517377.

As for P[,], the client sent the following five priority frames:

Stream ID	Exclusivity Bit	Dependent Stream ID	Weight
3	0	0	201
5	0	0	101
7	0	0	1
9	0	7	1
11	0	3	1

The resulting HTTP/2 fingerprint would be:

`1:65536;4:131072;5:16384|12517377|3:0:0:201,5:0:0:101,7:0:0:1,9:0:7:1,11:0:3:1`

Sample Fingerprints for Common HTTP/2 Implementations & Clients

Below are a few sample HTTP/2 fingerprints that demonstrate how unique HTTP/2 implementations can be, and how their fingerprints differ from one another:

Example 1: Chrome Browser on Mac OS X

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.96 Safari/537.36

HTTP/2 fingerprint:
`1:65536;3:1000;4:6291456|15663105|0`

Example 2: Chrome Browser on Windows 10 (Identical to Chrome in Example #1)

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.96 Safari/537.36

HTTP/2 fingerprint:
`1:65536;3:1000;4:6291456|15663105|0`

Example 3: Microsoft Edge Browser on Windows 10

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393

HTTP/2 fingerprint:

`3:1024;4:10485760|10420225|0`

Example 4: OkHttp (library) client (http://square.github.io/okhttp/)

User-Agent: okhttp/3.6.0

HTTP/2 fingerprint:
`4:16777216|16711681|0`

Example 5: Firefox 53.0 On Mac OS X 10.11

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:53.0) Gecko/20100101 Firefox/53.0

HTTP/2 fingerprint:
`1:65536;4:131072;5:16384|12517377|3:0:0:201,5:0:0:101,7:0:0:1,9:0:7:1,11:0:3:1`

Example 6: Firefox 53.0 Android Mobile

User-Agent: Mozilla/5.0 (Android 7.1.2; Mobile; rv:53.0) Gecko/53.0 Firefox/53.0

HTTP/2 fingerprint:
`1:4096;4:32768;5:16384|12517377|3:0:0:201,5:0:0:101,7:0:0:1,9:0:7:1,11:0:3:1`

Example 7: Go-based client

User-Agent: Go-http-client/2.0

HTTP/2 fingerprint:
`2:0;4:4194304;6:10485760|1073741824|0`

Example 8: Curl/7.54.0

User-Agent: Curl/7.54.0

HTTP/2 fingerprint:
3:100;4:1073741824;2:0|1073676289|0

Example 9: nghttp2 CLI client

User-Agent: nghttp2/1.22.0

HTTP/2 fingerprint:
`3:100;4:65535|00|3:0:0:201,5:0:0:101,7:0:0:1,9:0:7:1,11:0:3:1`

Request Pseudo-Header Fields Order

Pseudo-header fields in HTTP/2 are defined by the RFC as follows:

"While HTTP/1.x used the message start-line (see [RFC7230], Section 3.1) to convey the target URI, the method of the request, and the status code for the response, HTTP/2 uses special pseudo-header fields beginning with ':' character (ASCII 0x3a) for this purpose.

Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this Document."

"The following pseudo-header fields are defined for HTTP/2 requests:

- The "`:method`" pseudo-header field includes the HTTP method ([RFC7231], Section 4).
- The "`:scheme`" pseudo-header field includes the scheme portion of the target URI ([RFC3986], Section 3.1). "`:scheme`" is not restricted to "http" and "https" schemed URIs. A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.
- The "`:authority`" pseudo-header field includes the authority portion of the target URI ([RFC3986], Section 3.2). The authority MUST NOT include the deprecated "userinfo" subcomponent for "http" or "https" schemed URIs...
- The "`:path`" pseudo-header field includes the path and query parts of the target URI (the "path-absolute" production and optionally a '?' character followed by the "query" production (see Sections 3.3 and 3.4 of [RFC3986]). A request in asterisk form includes the value '*' for the ":path" pseudo-header field."

We noticed that request pseudo-headers appeared in a different order which depends on client implementation. For example, Chrome browsers issued the pseudo-headers in the following order:

`:method:` GET

`:authority:` http2.some.site

`:scheme:` https

`:path:` /

While Firefox browsers sent them as follows:

:**`method:`** GET

`:path:` /

`:authority:` http2.some.site

`:scheme:` https

The following table demonstrates the difference in pseudo-headers order in several common HTTP/2 implementations:

Client / Implementation	Pseudo Headers Name Order
Google Chrome (58.0.3029.110 on Mac OS X)	`:method, :authority, :scheme, :path`
Firefox v53.0 (Mac OS X)	`:method, :path, :authority, :scheme`
Safari v10.1 (Mac OS X)	`:method, :scheme, :path, :authority`
Curl v7.54.0 (Mac OS X)	`:method, :path, :scheme, :authority`
Go-http-client v2.0	`:authority, :method, :path, :scheme`
Jetty HTTP2 Client v9.3.4.v20151007	`:scheme, :method, :authority, :path`

Looking at Chrome's source code, we can see where this pseudo-header order is defined:

```
void CreateSpdyHeadersFromHttpRequest(const HttpRequestInfo& info,
                                      const HttpRequestHeaders& request_headers,
                                      bool direct,
                                      SpdyHeaderBlock* headers) {
  (*headers)[":method"] = info.method;
  if (info.method == "CONNECT") {
    (*headers)[":authority"] = GetHostAndPort(info.url);
  } else {
    (*headers)[":authority"] = GetHostAndOptionalPort(info.url);
    (*headers)[":scheme"] = info.url.scheme();
    (*headers)[":path"] = info.url.PathForRequest();
  }
```

The source code for the Class SpdyHeaderBlock includes the following comment, which mentions that pseudo header order is maintained during insertion:

// This class provides a key-value map that can be used to store SPDY header.
// names and values. This data structure preserves insertion order.

The Class itself uses a C++ std::list container, which preserves insertion order.

Pseudo-header order can be encoded into the suggested HTTP/2 client fingerprint in the following manner:

`S[;]|WU|P[,]#|PS[,]`

Where **PS** can have one of the following: values:

m (**:method**)

p (**:path**)

a (**:authority**)

s (**:scheme**)

For example:

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:53.0) Gecko/20100101 Firefox/53.0

HTTP/2 fingerprint:

`1:65536;4:131072;5:16384|12517377|3:0:0:201,5:0:0:101,7:0:0:1,9:0:7:1,11:0:3:1|m,p,a,s`

5.0 Use Cases for Passive HTTP/2 Client Fingerprinting

Spoofed User-Agent Detection

HTTP/2 fingerprint uniqueness is only influenced by the client's implementation of the protocol and is not affected by specific user environment factors. The HTTP/2 fingerprint, by itself, does not provide enough entropy to fingerprint or track specific users. However, it does expose information about the type of specific HTTP/2 implementation and, in many cases, reveals information about the client's vendor, operating system type, and version.

This information may be leveraged to detect clients that spoof their User-Agent string. For example, an automated web scraping tool, written using the "Go" programming language, may send a spoofed Chrome web browser User-Agent string in order to evade anti-automation protection mechanisms. In these cases, it would be quite simple to detect a spoofing attempt and deduce the real HTTP/2 client type.

However, applications could also use the passive HTTP/2 fingerprint to gain confidence and assurance about a client's stated User-Agent string.

Anonymous Proxy/VPN Detection

It is quite common to see web clients connecting through anonymizing proxies in order to mask their true identity or geolocation. In some cases, certain web applications and online services try to detect whether or not a request was routed through an anonymizing intermediary device such as Proxy or VPN. By correlating (discrepant) information from the passive TCP, TLS, and HTTP/2 fingerprints, an application can passively deduce that the client was routing traffic through a proxy.

Imagine a client running a Chrome browser on Mac OS X, routing HTTP/2 traffic through an intermediary anonymizing proxy that is running on a Windows 10 machine. Since the anonymizing proxy does not terminate TLS and does not terminate and rewrite HTTP/2 traffic, the TCP fingerprint will show that a Windows 10 machine was connecting to the web server, while the TLS and HTTP/2 fingerprints will expose the fact that the client is actually running a Chrome browser on Mac OS X.

While this information could have been deduced solely on the discrepancy between the TCP and TLS fingerprints, the HTTP/2 fingerprint contributes to the overall confidence of the detection. Additionally, while some web clients enable a user to launch them with customized TLS settings, our research shows that many HTTP/2 clients don't support modification of basic HTTP/2 implementation details such as the SETTINGS frame values, or the pseudo-headers name order.

It should be noted that while the HTTP/2 protocol does not mandate the use of TLS encryption, some implementations only support HTTP/2 over TLS, and currently no browser supports HTTP/2 over unencrypted connections. This means that passive TLS fingerprints can almost always be collected in conjunction with the HTTP/2 features mentioned in this document to form a more accurate fingerprint.

6.0 CONCLUSION

HTTP/2 is considered the future of the Internet. It is the second major version of the HTTP protocol, which was developed by the IETF's HTTP Working Group. HTTP/2 is a binary protocol that is fully multiplexed. It can therefore use one connection for parallelism. The protocol uses header compression to reduce overhead and also allows servers to "push" responses proactively into client caches.

The HTTP/2 standard was officially approved in February 2015, and is already supported by most web browsers and servers.

The dramatic and fundamental changes from HTTP/1.x to HTTP/2 mean that client-side and server-side implementations need to incorporate completely new code in order to support new HTTP/2 features.

This paper demonstrates how these new implementations create small nuances, which differentiate HTTP/2 clients from one another. In addition, we have shown how these unique implementation features can be leveraged to passively fingerprint web clients. Our research shows that passive HTTP/2 client fingerprinting can be used to deduce the true details about the client's implementation — for example, browser type, version, and sometimes even the operating system. This technique can be used to better detect clients that spoof or don't report their User-Agent string, and at the same time increase confidence in User-Agent strings reported by legitimate clients. Moreover, HTTP/2 fingerprints can be used to enhance anonymous proxies and VPNs, which are used by some users on the Internet.