Security Analysis of HTTP/2 Protocol

by

Naveen Tiwari

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2017 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Chair
Adam Doupé
Ziming Zhao

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

The Internet traffic, today, comprises majorly of Hyper Text Transfer Protocol (HTTP). The first version of HTTP protocol was standardized in 1991, followed by a major upgrade in May 2015. HTTP/2 is the next generation of HTTP protocol that promises to resolve short-comings of HTTP 1.1 and provide features to greatly improve upon its performance.

There has been a 1000% increase in the cyber crimes rate over the past two years. Since HTTP/2 is a relatively new protocol with a very high acceptance rate (around 68% of all HTTPS traffic), it gives rise to an urgent need of analyzing this protocol from a security vulnerability perspective.

In this thesis, I have systematically analyzed the security concerns in HTTP/2 protocol - starting from the specifications, testing all variation of frames (basic entity in HTTP/2 protocol) and every new introduced feature.

In this thesis, I also propose the Context Aware fuzz Testing for Binary communication protocols methodology. Using this testing methodology, I was able to discover a serious security susceptibility using which an attacker can carry out a denial-of-service attack on Apache web-server.

DEDICATION

To my parents for all their love and support, and their insistence on best possible education. I appreciate all their sacrifice without which this would not have been possible. Thanks Mom and Dad.

To my fiancée, Manisha for her support and understanding...

To Erik Trickel for his mentoring...

ACKNOWLEDGMENTS

The journey towards completion of my thesis has been circuitous and I am extremely grateful to the amazing people around who helped me in achieving this goal.

First of all, I would like to express the deepest gratitude to my Supervisor Dr. Gail-Joon Ahn , without whom this would not have been possible. I would like to thank him for his extensive support and encouragement during the tenure.

I am immensely thankful to Dr. Adam Doupé and Dr. Ziming Zhao for their guidance during every stage of this project.

The support and feedback from my friends - Sukhwa Kyung, Haehyun Cho, Erik Trickle, Faris Kokulu, Dr. Carlos Rubio and Vaibhav Dixit has been a motivating force for me and I can't thank them enough for this.

I would also like to acknowledge the everlasting support of my family and their words of wisdom at times when I felt low.

Last, but definitely not the least, Dr. Wonkyu Han. I would like to thank him for guiding me throughout the project, especially during the initial phases.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Internet, as we know today, is made up of massively distributed client and server based information systems. It comprises of diverse type of applications like email, file transfer, audio-video streaming, web browsing and so on. To support such wide range of applications there are enormous number of protocols like HTTP, FTP, SMTP, POP, Telnet, SSH, SMB to name a few [9]. Amongst all such protocols used over the Internet, HTTP is the most popular and widely used (based on the World-Wide-Web traffic [24]).

Figure 1.1 shows the traffic distribution among various network layer protocols, clearly identifying TCP as the most prevalent protocol on the Internet [8].

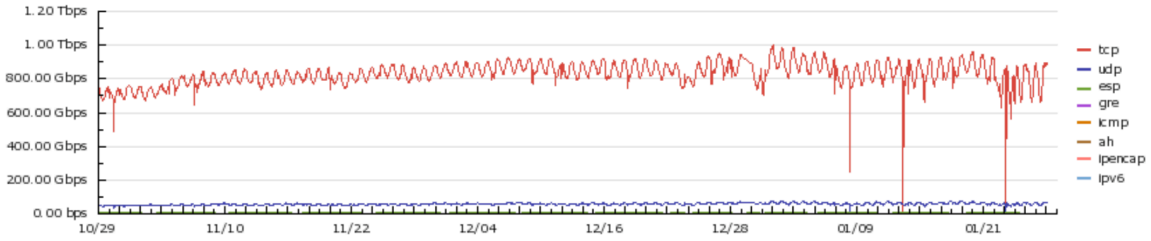

Figure 1.1: Network Layer Protocol Traffic Distribution (Source arbor.net [8]).

HTTP is an application layer protocol, built on top of TCP/IP. According to the data collected by 67 different ISPs (Source arbor.net [8]), amongst all the protocols built on top of TCP, HTTP is the most prominent (as depicted in figure 1.2).

This high prevalence and continued migration to HTTP makes it one of the most targeted protocol for the attackers. Figure 1.3 shows the percentage of attacks on various TCP based protocols [8].

Figure 1.2: Traffic on Various TCP Ports (Source arbor.net [8]).

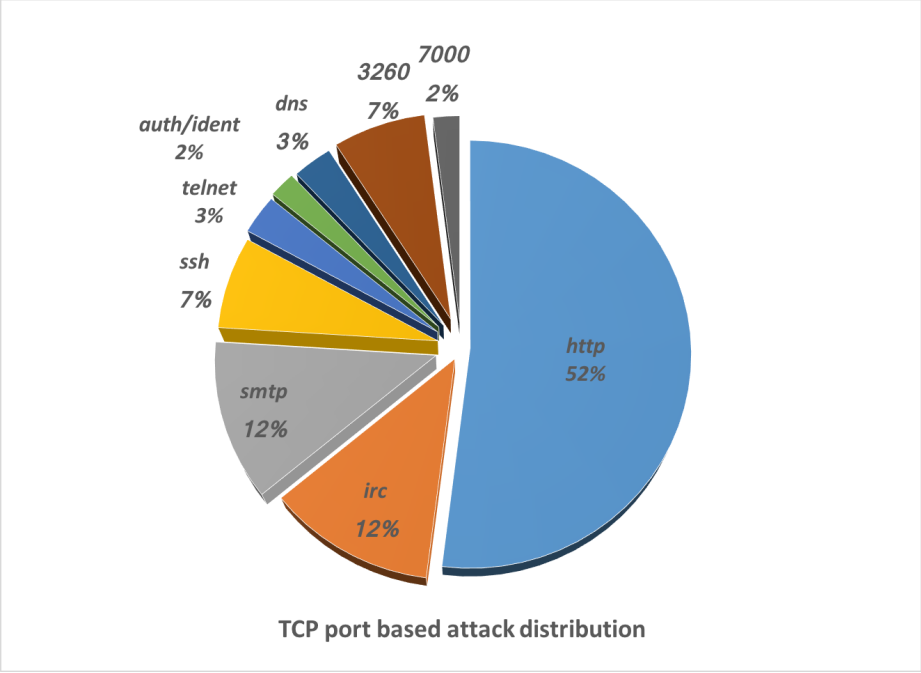

Figure 1.3: TCP Port Based Attack Distribution (Source arbor.net [8]).

Since its introduction in 1991 [31], there have been four different versions of HTTP. The first three variations included minor upgrades over their predecessor with addition of few new functionalities. HTTP/2, the latest version of HTTP, has its origin from the Google's SPDY protocol which attempts to re-look at HTTP from the perspective of new era web technologies requirements. HTTP/2 has been developed from scratch and was standardized in May 2015.

Since its standardization, there has been a tremendous rise in migration from HTTP 1.1 to HTTP/2. The biggest factor contributing to this shift is the significant improvement in page load time [15], thereby leading to high performance. Figure 1.4 shows the growth rate of HTTP/2 over time [34], taking into account both HTTP and HTTPS traffic.

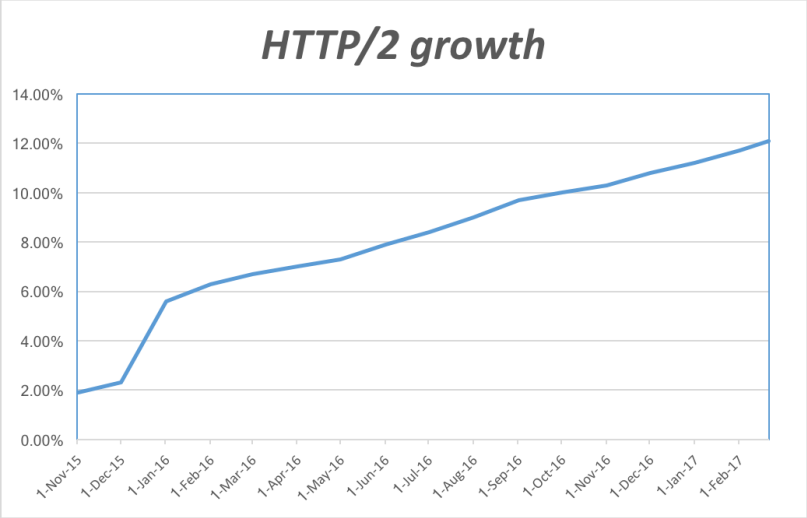

Figure 1.4: Growth of HTTP/2 (Source w3techs.com [34]).

Although HTTP/2 has a significant acceptance rate in totality, it would be important to check the statistics for HTTPS traffic only, since our primary focus is on security analysis. Figure 1.5 shows that among all communications over HTTPS, approximately 68% of the traffic utilizes HTTP/2 (as of April 2016) [20].

The motivation for the development of HTTP/2 lies in the shortcomings of HTTP 1.1 and also the evolved needs of the modern era of web technologies. The key factors are as follows:

- HTTP 1.1 is a good protocol but its performance has degraded over the past 15 years with the fast pacing advancements in the Internet world. At the time of its development, the major objective of the web had been to show static web pages

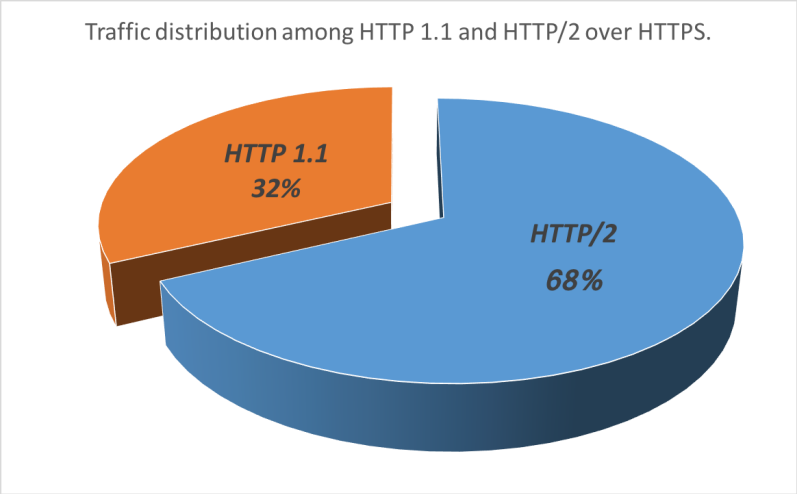

Figure 1.5: Percentage of Communication Over HTTPS (Source keycdn.com [20]).

with images. Over the course of time, loading a web page has become a resource intensive task, owing to the fact that most of the web pages are inherently dynamic with lots of images and scripts controlling the run time behavior of the page. Thereby arising the need to revisit the protocol's performance.

- The requirement of any new feature in HTTP 1.1 to cope up with the increasing needs of the web, made the protocol more convoluted, leading to implementations that are prone to bugs.

- HTTP 1.1 protocol is based in ASCII. Since the web pages are processed by computers and not humans (thereby eliminating the need for an ASCII based protocol), it increases the required bandwidth and also the processor load on the system.

- There is no way of simultaneously fetching resources due to inherent sequential nature of the protocol.

To achieve the targeted goals of HTTP/2, members of Internet Engineering Task Force (IETF) have looked at every aspect of the protocol and introduced many new functionalities. The key features are as follows:

- **Binary Protocol** - HTTP 1.x used ASCII format for request and response, that was highly resource intensive for client and server. HTTP/2 uses the same semantics but in binarized version, there by making it more efficient. This has many advantages like:
  - Reduced overhead in request and response.
  - Lower on-the-wire footprint thus reducing bandwidth requirements.
  - Reduced network latency and improvement in overall throughput
- **Stream Priority** - This feature of HTTP/2 protocol was introduced with the new capability requirements of the web. While loading a web page, the browser (or rendering engine) proceeds in a logical sequence and hence the client should be able to ask the server to prioritize a particular resource over others. An example for this would be prioritizing the HTML, CSS and JavaScript files over the image files.
- **Multiplexing** - This is one of the most important feature in HTTP/2 which was introduced to remove sequential request and response. Using this feature, client and server can use the same TCP connection to fetch and serve multiple resources.
- **Server Push** - This feature was introduced due to the fact that server can have prior knowledge about the resource requirement of client. The main advantage of this feature is utilized by servers that generate web pages based on the infor-

mation in request. The server can utilize this time for sending other resources like JavaScript, CSS and image files which are static in nature.

- **Header Compression** - This feature was introduced in order to further reduce the bandwidth requirements. The client and server use HPACK algorithm to compress the transmitted request and response header.

- **ALPN** - **A**pplication **L**ayer **P**rotocol **N**egotiation (or ALPN) is a **T**ransport **L**ayer **S**ecurity (TLS) extension for application layer protocol negotiation. This was introduced to decide upon the use of HTTP/2 as application layer protocol for communication by server and client. The important aspect of ALPN is that the protocol negotiation happens without introducing any additional round trips in TLS handshake.

HTTP/2 is a new protocol with high acceptance rate, which demands an urgent need for evaluating this protocol for any security concerns. This was my primary motivation for working on the security vulnerability analysis of HTTP/2 protocol.

In order to do the vulnerability analysis I chose NGINX (version 1.11.3) web server and Apache web server (version 2.4.17 - 2.4.23). The primary reason for selecting these as servers is that they are open source and serve around 70% of the entire web traffic. The table 1.1 shows the web server that developers share for top million web sites.

For client we selected Chrome (version 52) and Firefox (version 48) since they are used by more than 70% of all the users [2].

The main contributions of this thesis are as following:

- Developed a testing methodology that can be used for testing any binary network communication protocol.

Table 1.1: Web Server Developers: Market Share of Top Million Busiest Sites (Source netcraft.com [4]).

Developer	**January 2017**	**Percent**	**February 2017**	**Percent**	**Change**
Apache	416,257	41.63%	414,118	41.41%	-0.21
nginx	282,986	28.30%	283,409	28.34%	0.04
Microsoft	102,660	10.27%	101,909	10.19%	-0.08
Google	17,702	1.77%	17,648	1.76%	-0.01

- Discovered a serious security vulnerability which can allow the attacker to perform a denial-of-service attack on a server grade machine using a commodity hardware.
- Uncovered multiple ways in which one can fingerprint the server.

The rest of this thesis is organized in the following manner. Chapter 2 further describes the problems in HTTP 1.1 protocol, introduction to HTTP/2 and explanation of its various features. The chapter also explains white-box and fuzz testing, and provides information about the client and server that we have tested. Chapter 3 discusses other published research works related to HTTP/2 or the testing methodologies. Chapter 4 provides details on the utilized testing methodology, also outlining its applicability to other binary network communication protocols. Chapter 5 explains the implementation techniques. Chapter 6 explores the testing environment and the results obtained. Chapter 7 concludes this thesis.

Chapter 2

BACKGROUND

HTTP (Hyper Text Transfer Protocol) is an application layer protocol which was designed on the principles of simplicity. The original protocol, introduced in 1991 [31], had simple design goals which included:

- File transfer capabilities.
- Ability to search the indexed HTML archive.
- Ability to redirect a client to another server.

The initial design was developed under the assumption that all client requests are idempotent.

2.1 Why Do We Need HTTP/2

Internet has seen tremendous growth since the introduction of HTTP, but the basics have remained same [19]. The predecessor of HTTP/2 had started showing signs of aging - for every new demand of the growing web, it required patching which never completely resolved the problem as intended. For example the parallel loading of the resources required by the web pages.

The client can request only one resource at a time and since web pages nowadays contain huge number of scripts, style-sheet, images and other resources, the page load time increases. There were a couple of approaches devised in order to tackle this problem:

Multiple Client Connections In this approach, the client establishes multiple connections with servers (may be different servers based on resource location) and requests each resource on different connection. The main drawback with this approach is that it increases the load on the server due to higher consumption of resources. Hence, each web browser implements a practical limitation on the number of connections that can be established for each user request. Firefox limits the number of connections to 17 while Chrome limits it to 10 [13].

Request pipelining Figure 2.1 shows the concept of pipelining. This solves the sequential request response problem but faces Head of Line Blocking (a big response starves all the following responses) issue.

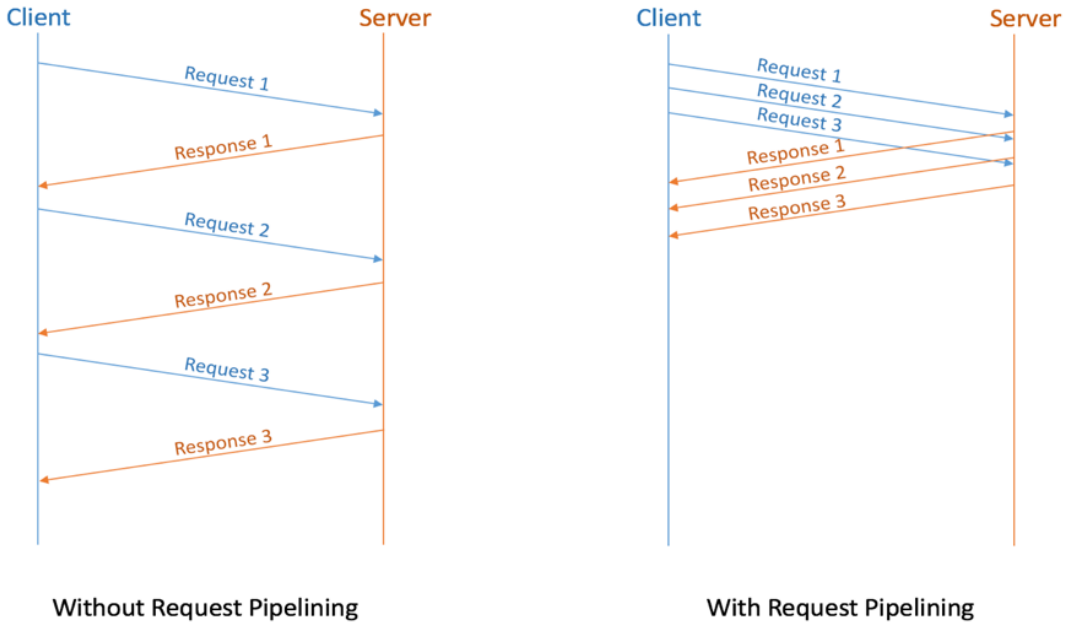

Figure 2.1: HTTP 1.1 Connection With And Without Pipelining.

2.2 Brief Introduction to HTTP/2

HTTP/2 is an application layer protocol that is build on top of TCP/IP layer and it shares all the resources that are used for HTTP 1.1 like the use of http://, https://, ports among other things. Since it shares everything with its predecessor, it requires a mechanism for the client and server to inform each other and agree on the use of HTTP/2.

HTTP/2 over clear text TCP If the client support HTTP/2, while sending the HTTP request it includes ***upgrade*** field with ***h2c*** as its value along with the HTTP/2 settings (refer section 2.2.1), indicating that the client supports HTTP/2. For example:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

The server ignores the ***upgrade*** field if it does not support HTTP/2 and proceed with normal HTTP 1.1 response. But if the server supports HTTP/2 it should send ***101 Switching protocols*** and then proceed with HTTP/2 communication. For example;

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

HTTP/2 over TLS During the handshake process of **T**ransport **L**ayer **S**ecurity (TLS) negotiation, the client and server negotiate the application layer protocol that they would be using. TLS defines an extension called **A**pplication **L**ayer **P**rotocol

Negotiation (ALPN) which allows the client and server to negotiate an application layer protocol in a secure manner without increasing the number of round trips. Using this protocol, the client sends a list of application layer protocols to server, which then selects one of them [16]. Earlier, the **N**ext **P**rotocol **N**egotiation (NPN) extension was used which has now been deprecated [3] [1].

After the client and server agree on HTTP/2 protocol, the client sends a string known as **Connection preface** such as

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

or

```
"PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n"
```

along with exchange of a setting frame (explained in section 2.2.1). The connection preface is a final conformation of the protocol in use that each end point is required to send.

The basic unit in HTTP/2 protocol is called a Frame that begins with a fixed 9-bytes header followed by a variable length of payload as shown in the figure 2.2. Fields in the header include:

- Length (24 Bits) - denotes total length of the frame.
- Type (8 Bits) - denotes type of the frame (refer section 2.2.1).
- Flag (8 Bits) - flags specific to frame type.
- Reservers (1 Bit) - Reserved for future use.
- Stream Id (31 Bits) - Used for parallelization of data transfer.

In HTTP/2 there are different types of information which has to be transfered between client and server. For each type of information the protocol defines different

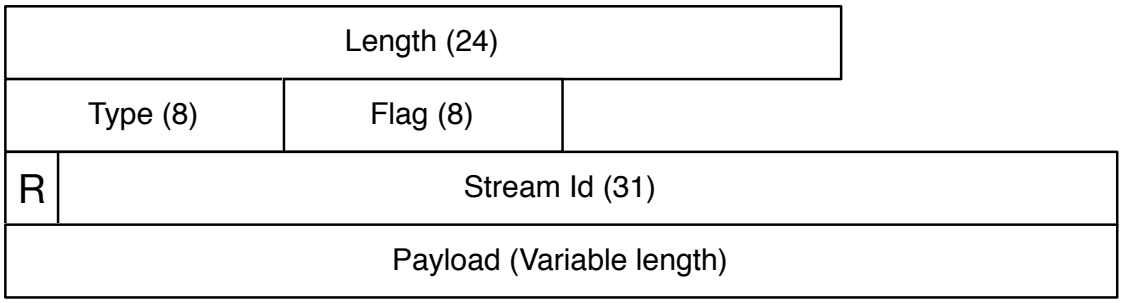

Figure 2.2: Structure of Frame.

types of frames. For example, the HTTP request is transfered using the **Header Frame**, data (HTML files, CSS, JavaScript or image files) is transfered using **Data Frame** and connection level settings like maximum size of frame are transfered using the **Setting Frame**.

2.2.1 Types of Frames

Data Frame (0x0) This frame is used to carry the data to and from the server.

Header Frame (0x1) This frame is used to send the HTTP request headers.

Priority Frame (0x2) This frame sets the priority of a stream. Priority is a 8 bit number, also called weight. An endpoint sends this frame to set priority of frame, but that is only a suggestion.

Reset Frame (0x3) This frame is used for immediate termination of the frame.

Setting Frame (0x4) This frame is used to transmit connection level setting information like maximum length of frame, maximum number of concurrent streams etc.

Push Promise Frame (0x5) This frame is used by the server when it knows the resource requirements of the user and wants to push them to avoid delays at the client end.

Ping Frame (0x6) This frame is utilized by the end points to check if the connection is active.

Go away Frame (0x7) This frame is used to close the connection.

Window Update (0x8) This frame type is used to implement flow control.

Continuation Frame (0x9) This frame is used along with header frame and push frame. The objective of continuation frame is to carry data of header and push frame when the payload exceeds frame size.

2.2.2 Communication Using HTTP/2 Protocol

A to and fro sequence of frames defines a **Stream**. Each stream can be used for only one request and response, Figure 2.3 shows the communication between the client and server over a stream. Each block in figure 2.3 represents a frame and all the frames together form a stream. The client sends HTTP header in Header frame to which the server responds using the Header frame along with the contents of index.html in Data frame.

Every stream in HTTP/2 comprises of multiple frames which can be assembled properly at the endpoint, provided they arrive in correct sequence. This capability allows the client and server to establish multiple streams of request/response for numerous resources, known as multiplexing.

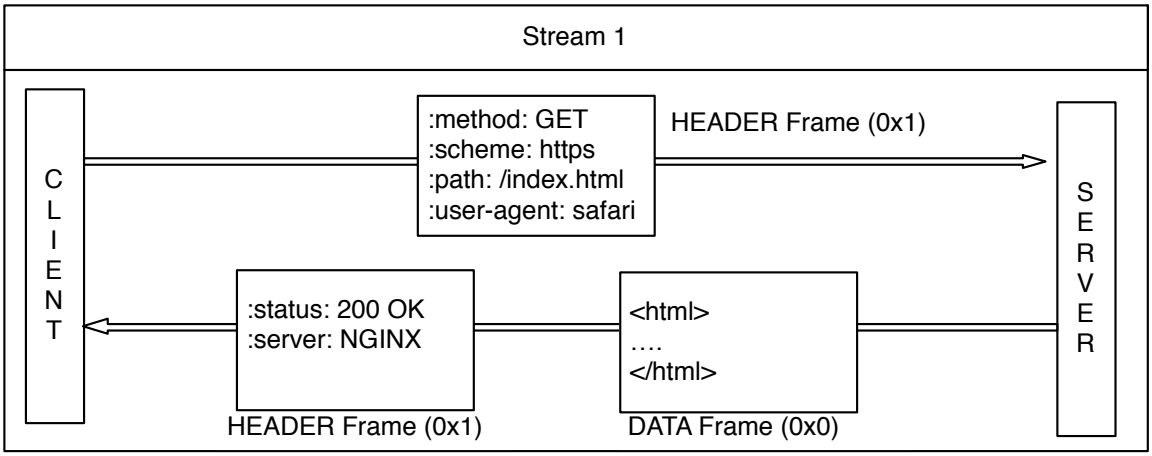

Figure 2.3: HTTP/2 Communication Over A Stream.

Figure 2.4 shows the communication between client and server. An important point to notice here is the multiplexing feature of HTTP/2 which the client utilizes to concurrently access resources.

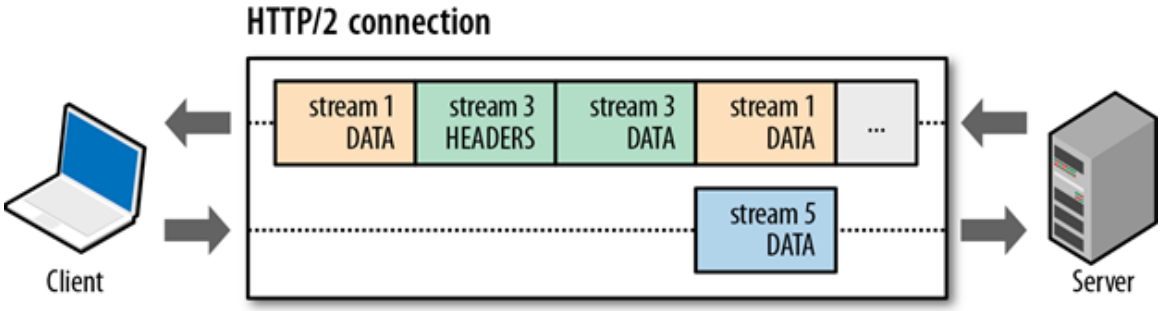

Figure 2.4: Overall HTTP/2 Communication Over A Connection.

2.2.3 HPACK: Header Compression For HTTP/2

The major objective of HTTP/2 was improved performance. Hence, it defines a simple and flexible mechanism of compressing the header fields to reduce bandwidth requirements.

HTTP/2 converts the HTTP request into a list of name value pairs. For example, consider the following HTTP header:

```
GET /index.html HTTP/1.1
```

This can be expressed as a list of header name and header value pairs as shown below:

Table 2.1: Example of Header Name and Header Value in HTTP/2.

Header Name	**Header Value**
:method	GET
:scheme	https
:path	/index.html

HPACK defines header compression algorithm for HTTP/2 where each endpoint maintains a mapping (known as HPACK table) of index to header name or header value or both, defined by the other endpoint. HPACK also defines a list of static entries (see table B.1 in Appendix B). Using the table B.1 we can convert the name value pairs in table 2.1. HPACK defines that the first bit of byte is reserved to mark the end of index. Since the index fits in 7 bits, they are represented as bytes in the request. In the following example, the client request would be 3 bytes in length as shown below:

10000010	10000111	10000101

2.3 Features of HTTP/2

2.3.1 Multiplexing

HTTP/2 defines **Stream** for communication between a client and server for each individual resource. The endpoints in this communication are allowed to asyn-

chronously request and respond for each Steam without stopping or blocking any other Stream. For example, consider the scenario where a web page is made up of 1 image, 1 JavaScript, 1 CSS and 1 HTML file. The client would create 4 different streams (assuming Server push is disabled) and the server is allowed to respond in any sequence, making sure that frames of a particular stream are in sequence. This would be beneficial, since the HTML page preparation might take some time while the other resources are read from disk and transmitted to the client.

2.3.2 Resource Prioritization

HTTP/2 defines a way of prioritizing a particular resource over another, since the web pages tend to contain lot of images and may take some time before the client receives them. It is important to note that for loading of a web page, the browsers (or rendering engines) require HTML, JavaScript and CSS before it need images.

2.3.3 Server Push

Accessing a web page over the Internet requires sending a request to remote server and receiving the response. A web page generally contains lot of resources like JavaScript, CSS and image files, most of which are generally static files.

In the traditional request response pattern, the client knows about the additional required resources only after receiving the web page. Most of the web pages are generated at run time, and the time elapsed between the server receiving the request and preparing the response is the idle time. This is illustrated in figure 2.5a.

Since server serving the request is (or can be made) aware of the additional resources that would be required by the client, it can push this information to client during the idle time. This feature increases the performance of the protocol by utilizing the time effectively. This is illustrated in figure 2.5b.

(a) Communication Without Server Push.

(b) Communication With Server Push.

Figure 2.5: Communication With And Without Server Push.

2.4 Whitebox Testing

White box testing is a software testing methodology that tests the internal structure of the software instead of the functionality like black box testing. White box testing requires internal knowledge of the system along with programming skills to develop the test cases. The test cases include multiple control flows of the software during execution along with data flow. The design of the test cases is based on following:

- Control Flow testing
- Data flow testing
- Branch testing
- Statement coverage
- Decision coverage
- Modified condition/decision coverage
- Prime path testing
- Path testing

2.5 Fuzz Testing

Fuzz testing (or simply Fuzzing), developed in 1989 [30], is an automated software testing technique used to find vulnerabilities or bugs in software by sending massive amount of random data or valid input data with random modifications as input.

Fuzz testing can be categorized in three different ways:

- Fuzzer can generate raw input for the program or it can randomly modify correct data to generate fuzzed input.
- Fuzzer can have knowledge about the structure of input data or can randomly generate the data.
- The fuzzed input data can be generated based on black box testing , grey box testing or white box testing.

Chapter 3

RELATED WORK

The web application security research team at **Imperva Defense Center** published a HTTP/2 analysis document outlining four different attack vectors for HTTP/2 implementations [6]:

- **Slow Read** attack sends correct application layer request but makes sure that it reads the response from server very slowly, thereby trying to exhaust all the server resources. By using the multiplexing feature of HTTP/2, client can simultaneously make multiple requests to the server with prior knowledge that most of the servers will allocate a different thread for each stream. This can be utilized by the client to attack the server.

- **HPACK (Compression)** raises two main concerns. Firstly, there is a risk of data leak when compression precedes encryption operation in an application. Secondly, there is risk from specially crafted zipped message that can cause unexpected behavior in the decoder, which was exploited in the zip bomb attacks.

- **Dependency DoS** The nghttp2 implementation of HTTP/2 suffers from possiblity of DoS attack or even remote code execution attack. This happens because Nghttp2 restricts the dependency graph size to MAX_CONCURRENT_STREAMS and upon new priority request it drops old streams. Due to improper memory cleanup, the attack becomes a possibility.

- **Stream abuse** - HTTP/2 defines that one stream ID can be used only for a particular request and response, and prevents further reuse of this ID. When IIS receives two different requests on the same stream, it crashes [10].

The research done by web application security research team at the **Imperva Defense Center** has been very crucial to the evolution of HTTP/2 as a secure protocol. But, there is definitely need for formulating a systematic approach to test all the binary network communication protocols.

Taintscope [35] is a checksum aware fuzz testing approach which tries to solve a common drawback of fuzz testing in protocols containing checksum field. This is a fully automated approach - detection of checksum field, fuzzing and repairing the crash samples. Results from the various experiments are indicative of Taintscope's high accuracy in identifying and drastically improving the fuzz testing approach. Taintscope is one of the most interesting research works which directed my attention towards formulating a methodology for testing network communication protocol.

Taint-based Directed Whitebox Fuzzing [17] is an automated fuzzing technique (also tool). It uses dynamic taint tracing to locate previously fuzzed regions that influenced values used in key program attack points. This is followed by generation of a new test input file using the previous test input file as base and fuzzed region as seed for fuzzing.

TLS-Attacker [29] is an open source framework for evaluating TLS library by allowing the users to create custom TLS message flows and randomly modifying the contents of a message.

Chapter 4

APPROACH

4.1 Why Context Aware Fuzz Testing?

Fuzz testing is generally a black box testing methodology, but in our case of **Context Aware Fuzz Testing**, it is used as a white box testing paradigm. The reasons for selecting White box fuzzing can be attributed to its many advantages. The application of black box fuzz testing approach on binary communication protocol poses special kind of challenges like -

- Controlling the state of protocol during the fuzzing process. For example, different states of HTTP 2 protocol are - Idle, reserved (local), reserved (remote), open, half closed (local), half closed (remote) and close.

  Without controlling the state of protocol, the fuzzing process would generate unbound fuzzed output, thereby making the process of fuzz testing difficult. This will also lead to a reduction in test coverage.

- Controlling the fuzzing process itself to weed out unnecessary fuzzing of the input data. For example, any modification to the payload would not be of any use.

- Black box fuzz testing has very low code coverage and it is able to test only a fraction of test scenarios.

- Black box testing suffers from repetitive testing of same thing multiple times.

- No advantage can be obtained from the fact that we are testing most of the open source projects for which the code is readily available.

4.2 Open Source Implementation

Our objective is to analyze HTTP/2 protocol from server and client perspective, using both encrypted and unencrypted channels. The table 4.1 lists open source implementation of HTTP/2 protocol supporting both - server and client, over HTTP and HTTPS [7].

Table 4.1: List of Open Source HTTP/2 Implementation.

Name	**Language**
Deuterium	C
http-2	Ruby
http2	Go
hyper	Python
Jetty	Java
Netty	Java
nghttp2	C
Protocol::HTTP2	Perl
firefly	Java

I have analyzed the following open source implementations of HTTP/2:

- **NGINX** implements the server component of HTTP/2 protocol over the encrypted channel only. The implementation is lucid and easy but there is no support for client and communication over clear text TCP.

- **nghttp2** is an implementation of HTTP/2 along with the header compression algorithm. This is one of the most stable and matured libraries for HTTP/2 which is also consumed by Apache web server.

- **hyper-h2** is pure-python based implementation of the HTTP/2 protocol.

4.3 Why Not Use Available Open Source Implementation?

Although there are numerous open source implementations available on the Internet, none of them proved suitable due to the following reasons:

- Most of the libraries were not flexible enough i.e. the logic of the HTTP/2 communication was built into the library and not modifiable. This posed as a challenge during protocol testing, where the sequence of bytes sent, did not follow the guidelines of HTTP/2 protocol itself. For example, sending frames with incompatible flags enabled is not possible.

- The frame formation happened within the library and most of the fields were populated inside, which was difficult to handle for many of the test cases. For example, one of my test case was sending multiple requests to the server in decreasing order of the stream id.

- There was very little support provided by the libraries for creating a custom payload along with the header field values. Due to this, most of the work was required to be done even after using the library.

- For HTTPS, encryption and decryption was not supported by any library.

The above factors lead to our own implementation of HTTP/2 for both client and server, using clear text TCP and TLS. The design for our implementation is very modular, enabling creation of a HTTP/2 frame with any value.

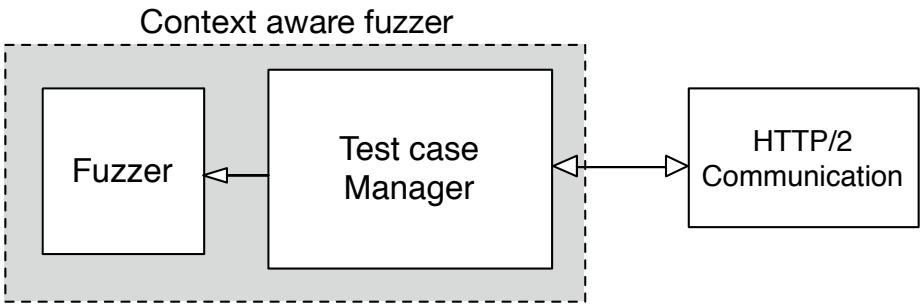

Figure 4.1: Design of Our Implementation.

Figure 4.1 show the major components in our design where each module has following functionality:

- **HTTP/2 Communication** module is responsible for creating packets as dictated by the test case manager. It supports the following functionalities:
  - Creating an empty frame.
  - Clearing values of any given frame.
  - Writing values in the frame. For example, writing stream id or frame type in the frame.
  - Handling header compression (when asked).
  - Automatically translating the header frame received from the other end point.
- **Test case manager** is responsible for maintaining the test cases that it has to run and the list of test cases it has already completed. All the values are disk persisted to avoid loss of data in case of failure. The test case manager is also responsible for providing the required values to the fuzzer. The flags are tested for all possible combinations and are not passed to the fuzzer.

- **Fuzzer** is responsible for fuzzing the HTTP/2 frame. Along with the frame, it also takes the bytes that it has to fuzz. This provides a more granular control to the **Test case manager** module.

- **Network module** is responsible for handling TLS negotiation for HTTP/2 and also for clear text TCP communication. During connection initiation, the **Test Case Manager** tells the **Network Module** if it has to establish a secure or an open connection.

4.4 Context Aware Fuzzing

Generally, all network communication protocols have same structure i.e. every packet is made up of header fields and payload. Hence, in order to test a protocol comprehensively, we should test them in isolation as well as with combination of packets which are used to provide a feature. For example, in TCP we have SYN, SYN-ACK and ACK packets to establish the connection, followed by the communication, and finally the FIN packet. Testing the protocol with each packet in isolation enables us to understand the behavior of system when it receives unexpected values. Fuzzing binary protocols may lead to unrestricted number of inputs being generated, making it unfeasible to use. In order to make this work, knowledge of protocol is required which helps in limiting the number of cases.

While testing communication protocols, it is essential to maintain states. Generally, communication protocols can be broken down into the following steps:

- Establishing Connection: This is the step where client and server exchange set of byte sequence to establish the connection. These set of bytes may or may not be first set of bytes in the entire communication. For example, in case of TCP these are actually the first set of bytes while in all the protocols that are based

on TCP like HTTP, these are not the first set of bytes. Most communication protocols utilize a simpler way of establishing the connection.However, new protocols like HTTP/2 have slightly complex way of doing that. In HTTP/2, client sends Magic sequence of bytes to server which informs the server that client is going to communicate in HTTP/2. To establish the connection, client sends another sequence of bytes (the header frame) to start communication over a stream. Here, unlike the traditional way of communication, server can also start a new communication stream.

This step of the communication protocol is extremely important from testing and vulnerability perspective - like the SYN-FIN attack in TCP, which does not have any logical sense in our regular use but should be handled properly. There are multiple such combinations that we have already seen in TCP protocol.

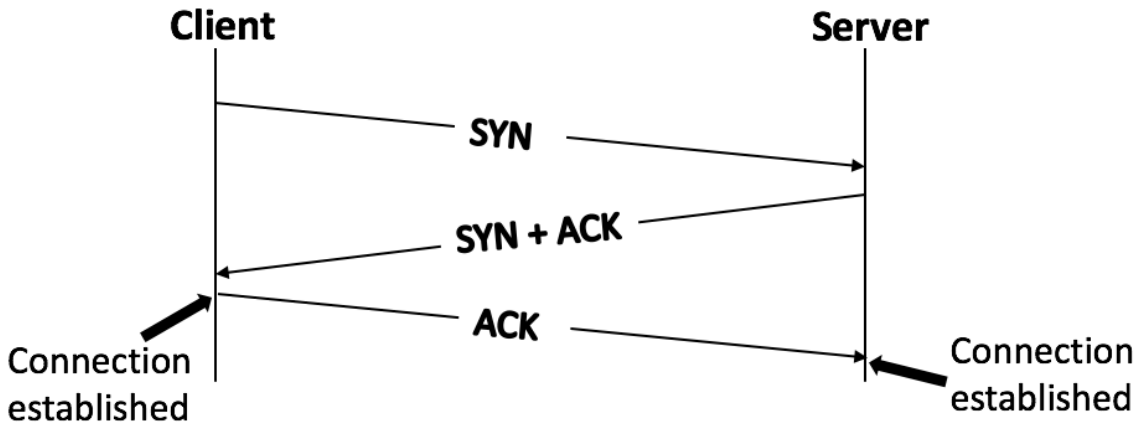

Figure 4.2: TCP Bytes Exchanges For Establishing Connection.

- Communication over the protocol: This is the part of the protocol where client and server exchange the data and is a rather mundane aspect from vulnerability and testing point of view.

  While testing this part of the protocol we need to ensure that during fuzzing of

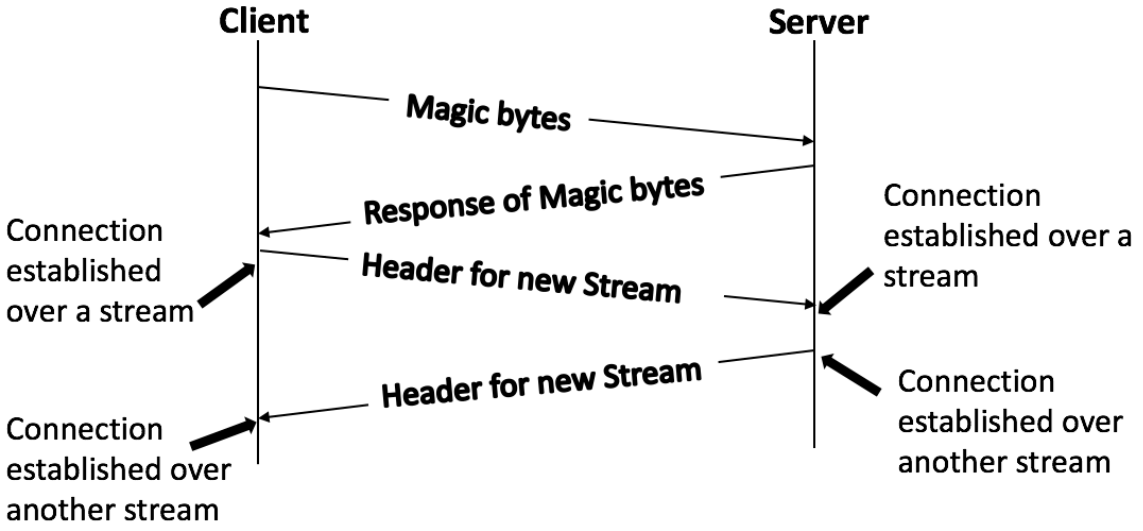

Figure 4.3: HTTP/2 Bytes Exchanges For Establishing Connection.

the input, no fields in the header frame are changed that could lead to changes in the state of the connection like closing or resetting it. Another important thing is to avoid fuzzing the payload part of the data as it does not concern the communication protocol itself.

- Connection closing from one end: This part of the protocol, generally called half open, is where the closing end only accepts the data. This step has some interesting aspects from vulnerability point of view - like if the closing end sends the data, would the other end still finish the remaining data or just abandon the connection. The fuzzing process should run uncontrolled for this step, since the connection has been closed only from our end and all the test cases generated by the fuzzing process are acceptable.

- Closing the connection from other end: This part of the protocol is generally called "half closed - remote". The chances of finding a security vulnerability here is slim because the other end would be releasing all the resources held for

this connection.

The fuzzing process can practically do anything because there is no next stage.

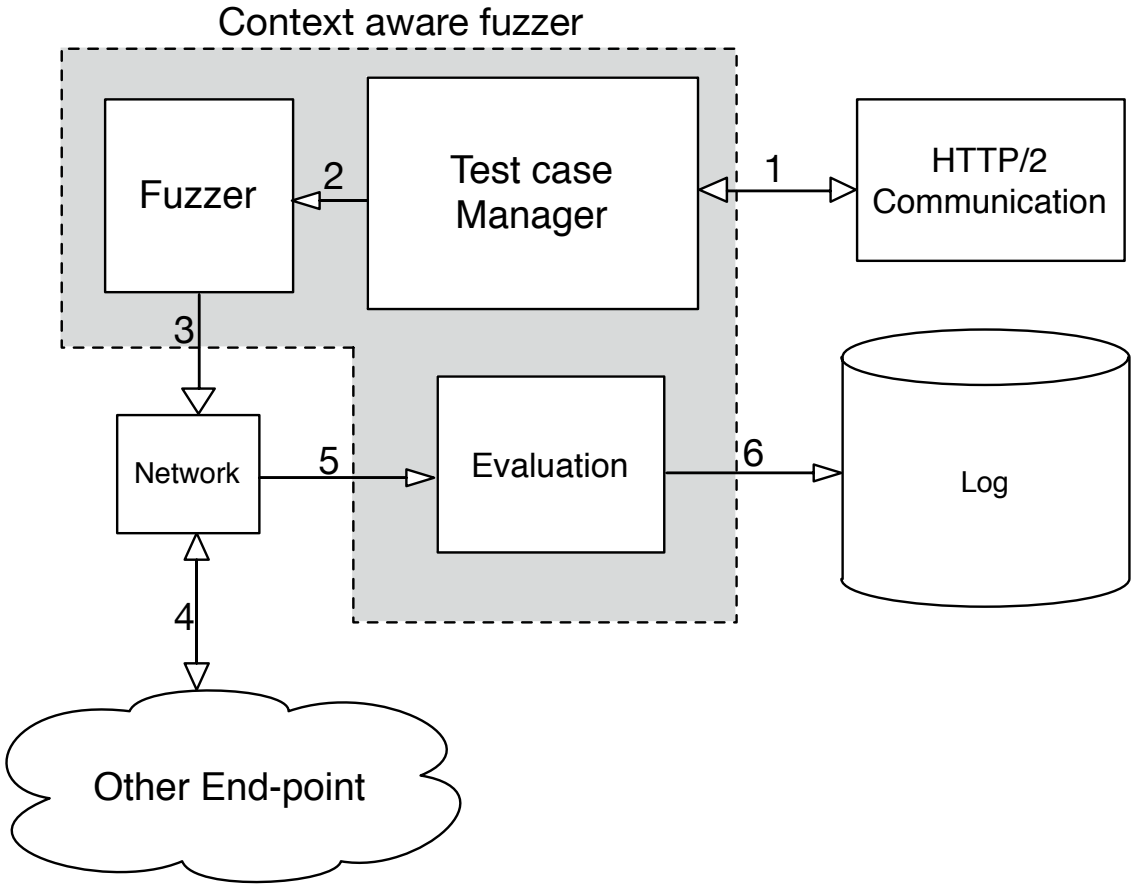

Figure 4.4: Architecture Of Our Testing Framework.

The figure 4.4 shows the architecture of our testing process. Most of the modules have been explained in section 4.3 and the explanation for the rest are as follows:

- **Evaluation** is the module that has pre-populated request-response pairs which are utilized to compare the response from the other endpoint. This enables automatic detection of success or failure of each test case.
- **Log** is module that keeps everything persisted, for restarting the testing process from the last position.

The figure 4.4 also shows the interaction between different components:

1. Generate a sequence of HTTP/2 frame (a full request/response).
2. Give a copy of the sequence of received HTTP/2 frame from step 1 to fuzzer along with sequence of bytes to fuzz.
3. Give the fuzzed input to the network module to transmit to other end-point.
4. Transmit the frame and collect the response (if any).
5. Give the received output to the Evaluation engine.
6. Log the response and state of the test case in files.

4.5 Challenges

One of the biggest challenge while working with the fuzzer based testing approach is finding out when the fuzzer is stalling and when it is running as expected. A lot of trial and error is involved in order to solve this issue. For example, when fuzzing a sequence of bytes representing a number, it requires heuristic and few optimizations - like not fuzzing higher order bytes in order to continuously generate large or small numbers.

4.6 Test Environment

The applicability of our test environment is suitable for testing any network based communication protocol like Transmission Control Protocol (TCP).

The Figure 4.5 shows the modification of our approach for the Transmission Control Protocol. Changes needed for testing TCP are:

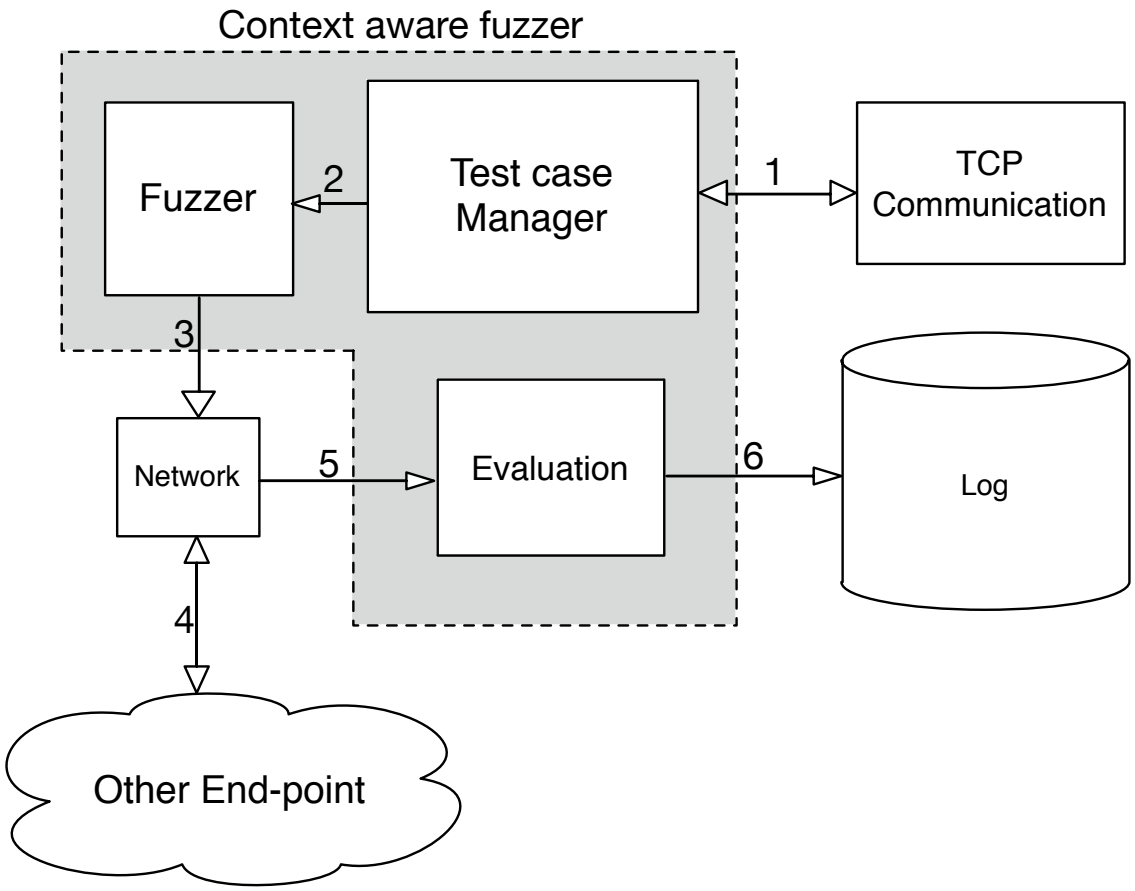

Figure 4.5: Architecture Of Testing Framework For TCP.

- Instead of generating sequence of HTTP/2 frame, there would be a module that generates a sequence of TCP packets.
- The context awareness part of the Test case manager would require modification since it has to understand what fields to fuzz in order to generate a sequence of test cases.

4.7 Test Cases.

Basic Connect - Disconnect test cases:

- Connection Creation
- Stream Creation
- Communication Over Stream
- Stream Closing
- Connection Closing

Feature wise testing of HTTP/2 protocol:

- Odd-even Stream Id Test
- Exceeding Agreed Concurrent Streams
- Prioritize completed streams
- Prioritize unused stream ids
- Prioritize current streams over unused stream
- Circular Dependency
- Extreme values of dependency

Fuzz testing for different parts of the frames: Frame wise testing of the HTTP/2 protocol

Table 4.2: Frame-wise Testing Of The HTTP/2 Protocol (Part 1).

	Data	**Header**	**Priority**	**Reset Stream**	**Setting**
Flag	Done	Done	Done	The stream is	Done
Length	Done	Done	Done	closed hence not	Done
Stream Id	Done	Done	Done	much can be	Done
Payload	Its Data	Done	Done	achieved.	Done

Table 4.3: Frame-wise Testing Of The HTTP/2 Protocol (Part 2).

	Push Promise	**Ping**	**Goaway**	**Window Update**	**Continuation**
Flag	Done	Done	The Connect-	Done	Done
Length	Done	Done	ion is closed	Done	Done
Stream Id	Done	Done	by the other	Done	Done
Payload	Done	Done	end hence	Done	Done
			no apparent		
			change.		

Chapter 5

IMPLEMENTATION

One of the reasons for popularity of C programming language is that it allows programmers to implement functionalities at machine-level, without resorting to assembly code (or even machine language), in order to achieve the targeted goal. For example, bit level manipulation are very easy in C. These reasons make C an ideal language for development of any binary network communication protocol.

Moreover, most of the encryption libraries (like OpenSSL) are written in C and numerous languages provide a wrapper over these libraries. This poses a significant challenge, specially when using a feature that has been recently introduced in the base library. For example, HTTP/2 uses **A**pplication **L**ayer **P**rotocol **N**egotiation (ALPN) for negotiating HTTP/2 as the application layer protocol. But since ALPN was recently introduced (at the time of starting this project), it did not have properly documented wrappers.

Following **Libraries** are used for this project:

- **libcrypto** - provides all the required cryptographic functionalities.
- **libssl** - provides support for various protocols in TLS.

5.1 Modules

The major modules in my implementation are:

- **IO Layer** - This layer is responsible for reading and writing of data to and from the socket. It is also responsible for TLS handshake and encryption/decryption. APIs' exposed by this module are:

  - init_HTTP2_client_overTLS - returns a communication context. It takes care of ALPN negotiation and makes sure that HTTP/2 is the application layer protocol.
  - init_HTTP2_client_over_clear_text - returns same communication context as init_HTTP2_client_overTLS.
  - read / write - returns the read / write data based on the input communication context (takes care of decryption, if required).

- **Frame** - This module is responsible for handling the bit level frame nuances and read the buffered frame. APIs' exposed by this module are:
  - **get_empty_frame** - returns an empty frame.
  - Setter and getters for all the frame header field.
  - **get_next_frame** - returns the next received frame from the other end point, based on the input communication context.

- **Fuzzer** - This module takes the frame and an array of bytes that are to be fuzzed as input. After fuzzing it writes the frame to the other endpoint using IO Layer.

- **Test case Manager** - This module is responsible for generating the frame sequence using the **HTTP/2 Communication** module. This sequence is a complete and valid request/response. It passes a copy of this sequence along with bytes that have to be fuzz tested.

- **HTTP/2 Communication** module is responsible for creating packets as dictated by the test case manager. It supports the following functionalities:
  - Creating an empty frame.

– Clearing values of any given frame.

– Writing values in the frame. For example, writing stream id or frame type in the frame.

– Handling header compression (when asked).

– Automatically translating the header frame received from the other end point.

Chapter 6

EXPERIMENTAL RESULTS

6.1 Experiment Environment

For the purpose of deriving results that corresponds to real world scenarios, we setup a wide range of systems resembling such scenarios. Table 6.1 lists the configuration of all the systems used in test environment.

The first server in table 6.1 represents a real world high end server while the others

Table 6.1: System Configuration For Test Environment.

# Physical Processor	RAM (in GB)	Swap (in GB)	Disk (in GB)
24	128	128	2048
2	16	5	1048
2	4	5	1048
4	16	10	256

represent various types of commodity hardware. This spectrum enabled us to broaden and explore a wide range of testing scenarios.

6.2 Problems With The specification

In most cases the specifications covers all the important aspects of the protocol but generally miss out on the corner cases. Each vendor defines his/her own interpretation on how to handle such corner cases that eventually makes the system susceptible to

fingerprinting attacks. Although many security researchers have stated that operating system fingerprinting is not the key to a successful attack, but they also acknowledge that it is definitely a starting point [22] [28]. For example, let us look at the TCP/IP protocol:

Table 6.2: Passive OS Identification Using Only The Initial Values in TCP/IP [18].

Operating System	**TTL (IP Datagram)**	**Window Size (TCP Packet)**
Linux (kernel 2.4 and 2.6)	64	5840
Google's customized Linux	64	5720
FreeBSD	64	65535
Windows XP	128	65535
Windows 7, Vista and Server 2008	128	8192
Cisco Router (IOS 12.4)	255	4128

In HTTP/2, the specification defines a mechanism to send large sized HTTP header. As per this, the HTTP header is broken down to a size that fits into the first header frame (min size = 2^{14} (default) to $2^{24} - 1$). Rest of the header is broken and sent using the continuation frames. HTTP/2 specification states that all these frame should be contiguous and not be interleaved with frames of other stream. Thse same is depicted in figure 6.1.

However, this HTTP/2 specification does not specify the number of continuation frames that the other endpoint receives before declining the request.

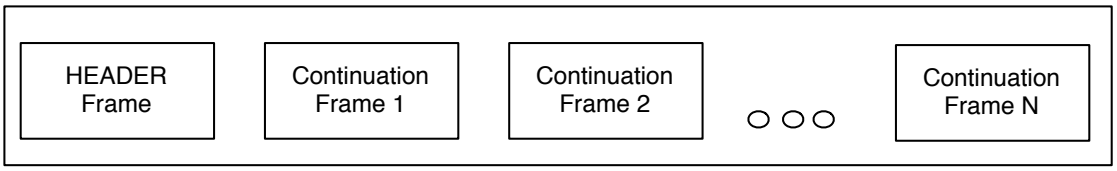

Figure 6.1: Sending Big HTTP Header Over A Stream In HTTP/2.

6.3 Finger Printing Of Web Server.

6.3.1 Difference in Half-Closed Stream Behavior.

Definition of half-closed (local/remote) as per specification.

```
half-closed (local):
A stream that is in the "half-closed (local)" state cannot be used
for sending frames other than WINDOW_UPDATE, PRIORITY, and
RST_STREAM.

A stream transitions from this state to "closed" when a frame that
contains an END_STREAM flag is received or when either peer sends
a RST_STREAM frame.

An endpoint can receive any type of frame in this state.
Providing flow-control credit using WINDOW_UPDATE frames is
necessary to continue receiving flow-controlled frames.  In this
state, a receiver can ignore WINDOW_UPDATE frames, which might
arrive for a short period after a frame bearing the END_STREAM
flag is sent.

PRIORITY frames received in this state are used to reprioritize
streams that depend on the identified stream.

half-closed (remote):
```

```
A stream that is "half-closed (remote)" is no longer being used by
the peer to send frames. In this state, an endpoint is no longer
obligated to maintain a receiver flow-control window.

If an endpoint receives additional frames, other than
WINDOW_UPDATE, PRIORITY, or RST_STREAM, for a stream that is in
this state, it MUST respond with a stream error of
type STREAM_CLOSED.

A stream that is "half-closed (remote)" can be used by the
endpoint to send frames of any type. In this state, the endpoint
continues to observe advertised stream-level flow-control limits.

A stream can transition from this state to "closed" by sending a
frame that contains an END_STREAM flag or when either peer sends a
RST_STREAM frame.
```

The end point which is half-closed(local) is not expected to transmit any frame to the other end point. But in case if the end point transmits frame the behavior of other end point is to close the stream with an error. Apache Web Server (versions 2.4.17 and above) does not close the steam and it chooses to ignore the received frames whereas the NGINX Web Server follows the specification and terminates the connection.

Using this difference in behavior of the both web servers we can utilize this to distinguish among them.

6.3.2 Frame Size in Apache Web Server.

The HTTP/2 specification states that every end-point should support minimum frame length of 2^{14}. NGINX Web Server supports the entire frame length range

i.e.from 2^{14} to 2^{24} where as the Apache Web Server supports only the bare minimum frame length 2^{14}.

This difference in frame length can be utilized in order to distinguish between Apache Web Server and NGINX Web Server.

6.3.3 Difference in Encoding String Using HPACK.

Header Compression.

```
The format defined in this specification treats a list of header
fields as an ordered collection of name-value pairs that can include
duplicate pairs.  Names and values are considered to be opaque
sequences of octets, and the order of header fields is preserved
after being compressed and decompressed.

Encoding is informed by header field tables that map header fields to
indexed values.  These header field tables can be incrementally
updated as new header fields are encoded or decoded.

In the encoded form, a header field is represented either literally
or as a reference to a header field in one of the header field
tables.  Therefore, a list of header fields can be encoded using a
mixture of references and literal values.

Literal values are either encoded directly or use a static Huffman
code.

The encoder is responsible for deciding which header fields to insert
as new entries in the header field tables.  The decoder executes the
modifications to the header field tables prescribed by the encoder,
reconstructing the list of header fields in the process.  This
enables decoders to remain simple and interoperate with a wide
```

```
variety of encoders.
```

Use of Huffman encoding is optional and both the web servers choose different length after which they encode the string using Huffman encoding.

6.4 Security Vulnerability: DoS

While testing this protocol using our approach, we discovered a serious security vulnerability in Apache HTTPD web server configured to use HTTP/2.
In HTTP/2 the client initiates a new stream by sending a header frame to server that contains the following -

- Unique unused odd number which acts as a stream id.
- Length of the entire frame.
- Appropriate flags
- List of HTTP request header like action (GET, PUT, POST, etc..), requested resource, user agent and other information.

As mentioned in the previous section, if the client cannot fit all the header content in one frame, the information is split into multiple frames where the last frame carries a flag indicating the end of header information. During the tests, we realized that the rate of packet transfer declined rapidly from client to server if the fuzzing process did not send the flag marking the end of HTTP header request. Upon further investigations, we discovered that the Apache web server did not enforce any restriction on the amount of memory that server can allocate for a particular client even after configuring the server with low values for '*LimitRequestFields*'. The figure 6.2 shows the amount of data received by the Apache web server and the memory it allocate for the client.

The server process gets killed by 'Linux out-of-memory killer'. The main point here is the amount of data that the client has to send in order to kill a server machine.

```
        Security Advisory - Apache Software Foundation
        Apache HTTPD WebServer  / httpd.apache.org

Server memory can be exhausted and service denied when HTTP/2 is used
CVE-2016-8740

The Apache HTTPD web server (from 2.4.17-2.4.23) did not apply
    limitations on request headers correctly when experimental module
    for the HTTP/2 protocol is used to access a resource.
The net result is that a the server allocates too much memory instead of
     denying the request. This can lead to memory exhaustion of the
    server by a properly crafted request.

Background:
___________

Apache has limits on the number and length of request header fields.
    which limits the amount of memory a client can allocate on the
    server for a request.

Version 2.4.17 of the Apache HTTP Server introduced an experimental
    feature:mod_http2 for the HTTP/2 protocol (RFC7540, previous
    versions were known as Google SPDY).
This module is NOT compiled in by default -and- is not enabled by
    default, although some distribution may have chosen to do so.

It is generally needs to be enabled in the 'Protocols' line in httpd by
    adding 'h2' and/or 'h2c' to the 'http/1.1' only default.
```

```
The default distributions of the Apache Software Foundation do not
    include this experimental feature.

Details:
________

- From version 2.4.17, upto and including version 2.4.23 the server
    failed to take the limitations on request memory use into account
    when providing access to a resource over HTTP/2. This issue has been
     fixed in version 2.4.23 (r1772576).

As a result - with a request using the HTTP/2 protocol a specially
    crafted request can allocate memory on the server until it reaches
    its limit. This can lead to denial of service for all requests
    against the server.

Impact:
_______

This can lead to denial of service for all server resources.

Versions affected:
__________________

All versions from  2.4.17 to  2.4.23.

Resolution:
___________

For a 2.4.23 version a patch is supplied. This will be included in the
    next release.

Mitigations and work arounds:
_____________________________
```

```
As a temporary workaround - HTTP/2 can be disabled by changing the
    configuration by removing h2 and h2c from the Protocols line(s) in
    the configuration file.

The resulting line should read:
Protocols http/1.1

Credits and timeline
--------------------

The flaw was found and reported by Naveen Tiwari <naveen.tiwari@asu.edu>
     and CDF/SEFCOM at Arizona State University on 2016-11-22. The issue
     was resolved by Stefan Eissing and incorporated in the Apache
    repository, ready for inclusion in the next release.

Apache would like to thank all involved for their help with this.
```

I further explored the possibility of bug or finger-printable attack for the same in NGINX and Apache web server and the results were shocking.

Figure 6.2 is a dual Y-axis graph which shows the memory consumption of the system versus time of the left Y-Axis and data received by the victim server versus time. The interesting thing in this graph is the scale on the two Y-Axis, memory consumption is measured in **GIGABYTES** while the data received is measured in **MEGABYTES**. The system configuration for this test was:

- Server i7 with 16 GB RAM
- Client Dual core Intel processor with 4GB RAM

Figure 6.3 shows the state of the system on right console, and the attack being carried out on the left console. This test was done on a server grade machine to

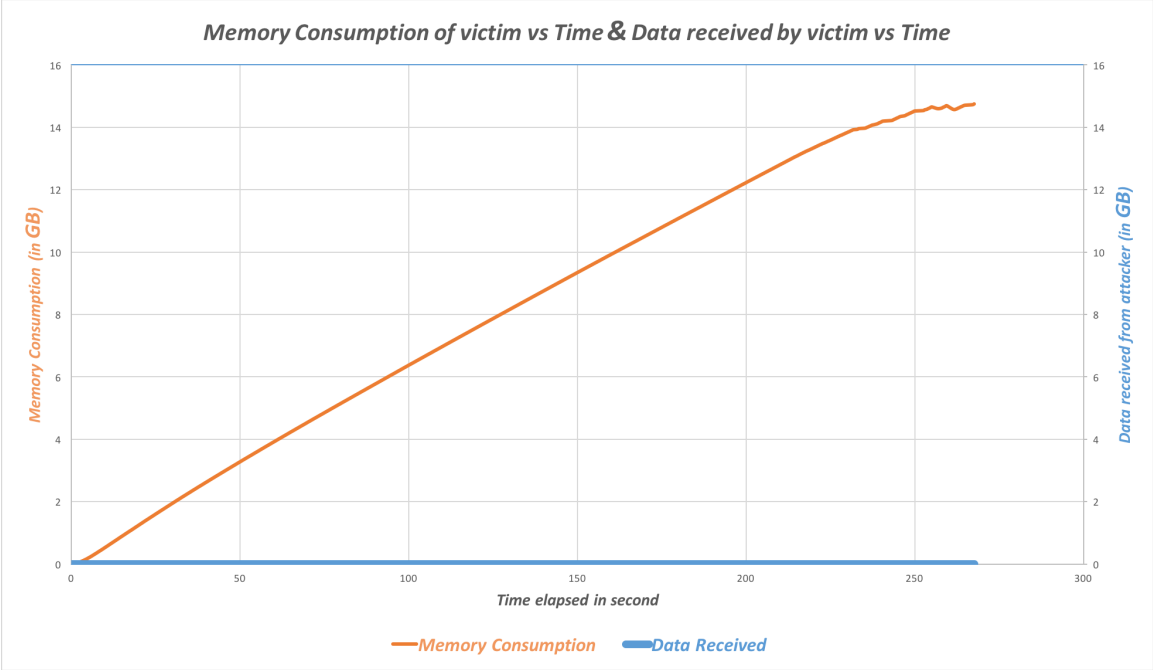

Figure 6.2: Memory Consumption of Victim vs Time & Data Received by Victim vs Time.

prove that even the big servers can crash with same client machines. The system configuration for the test was:

- Server Xeon dual processor each with 12 physical cores and 128 GB of RAM and 128 GB of swap.
- Client Dual core Intel processor with 4GB RAM

6.5 Server Push

The server push feature explained in the HTTP/2 specification lacks any practical use without extension of the specifications. The specifications talk about the packets exchange between client and server, and how the client can accept or deny one or all of the server push requests. The problem to be noted here is that the servers do not have

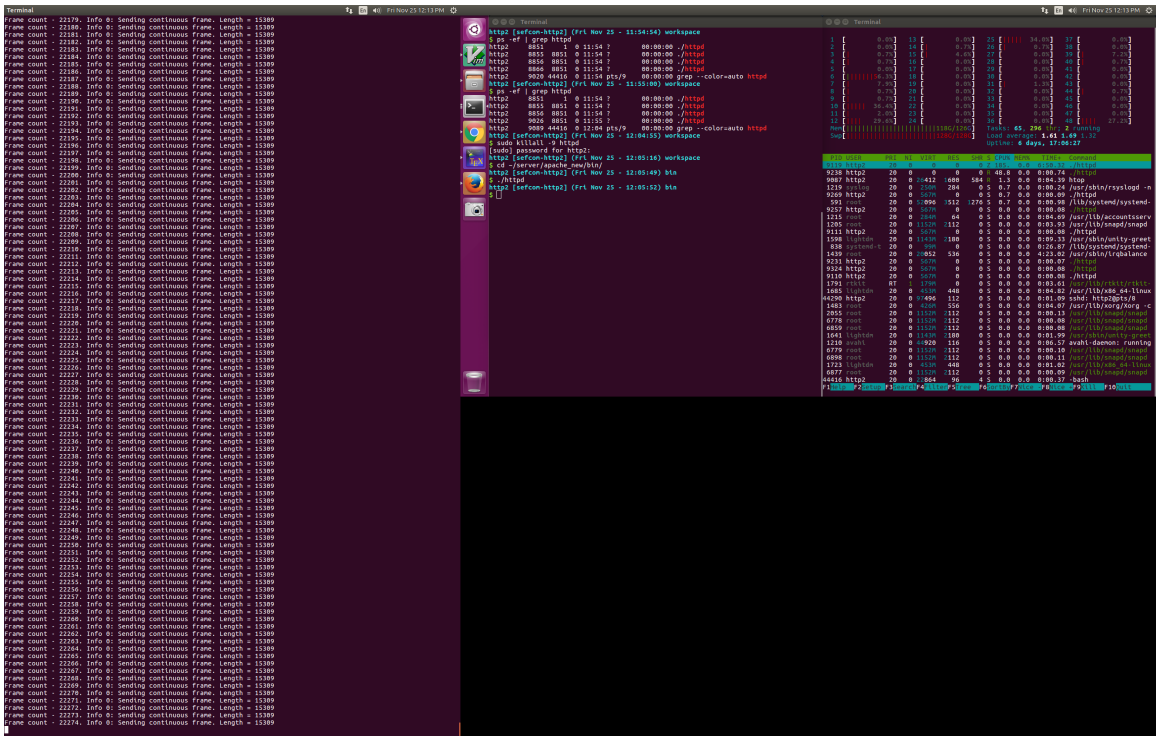

Figure 6.3: System Statistics While Being Attacked.

the ability to guess the resources required by the client only on the basis of information from the requested web page. Without knowledge of the required resources, the server cannot push to client. One company - CloudFlare, has attempted to extend the server push capability by utilizing the 'link' tag in HTTP request. Now, when the server tries to respond to a client request where the response is configured to include the 'link' tag, it first pushes the associated resources and then sends the header.

As far as the security of this feature goes, there can be only 2 possible vulnerable scenarios :

- Compromised server - Client is doomed anyways when it access the server, the entire safety of client comes from the browser security feature including the same-origin-policy.

- Legit Server - The basic idea behind the server push is to utilize the following times:

    1. Time required by the server to prepare the web page, server can start pushing the static resources like stylesheet, javascript, and may be images.
    2. The time taken by the client to parse the web page.

In both of the above cases the client has no idea about the resources required by the web page and hence can only cache the server push response and use them only if required by the requested page.

Chapter 7

CONCLUSION

With each passing day, the number of security attacks and threats to any system connected to the Internet is rapidly accelerating. These threats exploit minute vulnerabilities at each network and software layer, making it of paramount importance that we identify and prevent these vulnerabilities. Due to tremendous surge in acceptance rate of HTTP/2 protocol , it is a need of the hour to evaluate this protocol for any such security susceptibilities.

Being a relatively new protocol, in comparison to its predecessor - HTTP 1.1, containing many new functionalities, an exhaustive and effective testing methodology needed to be developed. **Context aware fuzzing for binary network communication protocol**, as outlined and explained in this thesis, has proven to achieve significant results in covering major aspects of security testing for HTTP/2 protocol. Moreover, this approach has been generalized and can be modified to test any binary network communication layer protocol, and not just HTTP/2.

Security vulnerability identified in the Apache Web Server, as outlined in section 6.4, has been very notable in preventing major DOS attack on systems using these servers.

REFERENCES

[1] "Deprecate and remove npn.", URL https://bugs.chromium.org/p/chromium/issues/detail?id=526713 (2017).

[2] "Desktop browser market share.", URL https://www.netmarketshare.com/browser-market-share.aspx?qprid=0&qpcustomd=0 (2017).

[3] "Do not negotiate http/2 using npn when using a blacklisted ciphersuite.", URL https://bugs.chromium.org/p/chromium/issues/detail?id=484709 (2017).

[4] "February 2017 web server survey.", URL https://news.netcraft.com/archives/2017/02/27/february-2017-web-server-survey.html (2017).

[5] "Global application trends.", URL http://www.menog.org/presentations/menog-6-7-8-9/MENOG-Trends%20in%20Internet%20Traffic%20Patterns_0.pdf (2017).

[6] "Http/2: In-depth analysis of the top four flaws of the next generation web protocol.", URL https://www.imperva.com/docs/Imperva_HII_HTTP2.pdf (2017).

[7] "Implementations", URL https://github.com/http2/http2-spec/wiki/Implementations (2017).

[8] "Internet traffic trends.", (2017), URL https://www.nanog.org/meetings/nanog43/presentations/Labovitz_internetstats_N43.pdf.

[9] "Lists of network protocols.", URL https://en.wikipedia.org/wiki/Lists_of_network_protocols (2017).

[10] "Microsoft security bulletin ms16-049.", URL https://technet.microsoft.com/en-us/library/security/ms16-049.aspx (2017).

[11] "Relation between http/2 and spdy.", URL https://http2.github.io/faq/#whats-the-relationship-with-spdy (2017).

[12] Belshe, M., R. Peon and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", The effects of brief mindfulness intervention on acute pain experience: An examination of individual difference **1**, 1–96 (2015).

[13] BrowserScope, "How does your browser compare", URL http://www.browserscope.org/?category=network&v=top (2017).

[14] Cadar, C., D. Dunbar and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", Proceedings of the 8th USENIX conference on Operating systems design and implementation pp. 209–224, URL http://portal.acm.org/citation.cfm?id=1855756 (2008).

[15] de Saxcé, H., I. Oprescu and Y. Chen, "Is http/2 really faster than http/1.1?", in "Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on", pp. 293–299 (IEEE, 2015).

[16] Friedl, S., A. Popov, A. Langley and E. Stephan, "Transport layer security (tls) application-layer protocol negotiation extension", URL https://tools.ietf.org/rfc/rfc7301.txt (2017).

[17] Ganesh, V., T. Leek and M. Rinard, "Taint-based directed whitebox fuzzing", in "Proceedings of the 31st International Conference on Software Engineering", pp. 474–484 (IEEE Computer Society, 2009).

[18] Hjelmvik, E., "Passive os fingerprinting.", URL http://www.netresec.com/?page=Blog&month=2011-11&post=Passive-OS-Fingerprinting (2017).

[19] InternetWorldStats, "Internet growth statistics", URL http://www.internetworldstats.com/emarketing.htm (2017).

[20] JACKSON, B., "Http/2 statistics keycdn report on http/2 distribution.", URL https://www.keycdn.com/blog/http2-statistics (2017).

[21] Kerner, S. M., "The month of the browser bugs begins.", URL http://www.internetnews.com/security/article.php/3618126 (2017).

[22] Lee, R. E., "Block os detection.", URL http://seclists.org/pen-test/2007/Sep/30 (2017).

[23] Mogull, R., "the month of kernel bugs (mokb) archive.", URL https://jon.oberheide.org/mokb/ (2017).

[24] Odlyzko, A. M., "Internet traffic growth: Sources and implications", in "Proceedings of SPIE", vol. 5247, pp. 1–15 (2003).

[25] Peon, R. and H. Ruellan, "RFC 7541 HPACK: Header Compression for HTTP/2", The effects of brief mindfulness intervention on acute pain experience: An examination of individual difference **1**, 1–55, URL https://tools.ietf.org/html/rfc7541 (2015).

[26] Peon, R. and H. Ruellan, "Huffman code", URL https://tools.ietf.org/html/rfc7541#appendix-B (2017).

[27] REUTERS, "Chinese see almost 1,000 percent increase in cyber attacks.", URL http://www.nbcnews.com/tech/tech-news/chinese-see-almost-1-000-percent-increase-cyber-attacks-n689466 (2017).

[28] Smart, M., G. R. Malan and F. Jahanian, "Defeating tcp/ip stack fingerprinting.", (2000).

[29] Somorovsky, J., "Systematic fuzzing and testing of tls libraries", in "Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security", pp. 1492–1504 (ACM, 2016).

[30] Takanen, A., J. Demott, C. Miller and I. Books24x7, "Fuzzing for software security testing and quality assurance", (2008).

[31] TimBL, "The original http as defined in 1991.", URL https://www.w3.org/Protocols/HTTP/AsImplemented.html (2017).

[32] Tiwari, N., "Apache httpd web server 2.4.23 memory exhaustion.", URL https://packetstormsecurity.com/files/140023/Apache-HTTPD-Web-Server-2.4.23-Memory-Exhaustion.html (2017).

[33] Tiwari, N., "Common vulnerabilities and exposures.", URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8740 (2017).

[34] w3techs, "Historical trends in the usage of site elements, December 2016", URL https://w3techs.com/technologies/history_overview/site_element/all (2017).

[35] Wang, T., T. Wei, G. Gu and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection", in "Proceedings - IEEE Symposium on Security and Privacy", pp. 497–512 (2010).

APPENDIX A

PYTHON EXPLOIT CODE

```
#!/usr/bin/python

"""
The mod_http2 module in the Apache HTTP Server 2.4.17 through 2.4.23,
    when the Protocols configuration includes h2 or h2c, does not
    restrict request-header length, which allows remote attackers to
    cause a denial of service (memory consumption) via crafted
    CONTINUATION frames in an HTTP/2 request.(https://access.redhat.com/
    security/cve/cve-2016-8740)

Usage : exploit.py [HOST] [PORT]
"""

import sys
import struct
import socket

HOST = sys.argv[1]
PORT = int(sys.argv[2])

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

# https://http2.github.io/http2-spec/#ConnectionHeader
s.sendall('PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n')

# https://http2.github.io/http2-spec/#SETTINGS
SETTINGS = struct.pack('3B', 0x00, 0x00, 0x00) # Length
SETTINGS += struct.pack('B', 0x04) # Type
SETTINGS += struct.pack('B', 0x00)
SETTINGS += struct.pack('>I', 0x00000000)

s.sendall(SETTINGS)

# https://http2.github.io/http2-spec/#HEADERS
HEADER_BLOCK_FRAME = '\x82\x84\x86\x41\x86\xa0\xe4\x1d\x13\x9d\x09\x7a\
    x88\x25\xb6\x50\xc3\xab\xb6\x15\xc1\x53\x03\x2a\x2f\x2a\x40\x83\x18\
    xc6\x3f\x04\x76\x76\x76\x76'

HEADERS = struct.pack('>I', len(HEADER_BLOCK_FRAME))[1:] # Length
HEADERS += struct.pack('B', 0x01) # Type
HEADERS += struct.pack('B', 0x00) # Flags
HEADERS += struct.pack('>I', 0x00000001) # Stream ID

s.sendall(HEADERS + HEADER_BLOCK_FRAME)

# Sending CONTINUATION frames for leaking memory
# https://http2.github.io/http2-spec/#CONTINUATION
while True:
    HEADER_BLOCK_FRAME = '\x40\x83\x18\xc6\x3f\x04\x76\x76\x76\x76'
    HEADERS = struct.pack('>I', len(HEADER_BLOCK_FRAME))[1:] # Length
    HEADERS += struct.pack('B', 0x09) # Type
    HEADERS += struct.pack('B', 0x01) # Flags
    HEADERS += struct.pack('>I', 0x00000001) # Stream ID
    s.sendall(HEADERS + HEADER_BLOCK_FRAME)
```

APPENDIX B

STATIC TABLE ENTRIES IN HPACK

Table B.1: Static Table Entries.

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate
17	accept-language	
18	accept-ranges	
19	accept	
20	access-control-allow-origin	
21	age	
22	allow	
23	authorization	
24	cache-control	
25	content-disposition	
26	content-encoding	
27	content-language	
28	content-length	
29	content-location	
30	content-range	
31	content-type	
32	cookie	
33	date	
34	etag	
35	expect	
36	expires	
37	from	
38	host	
39	if-match	
40	if-modified-since	
41	if-none-match	
42	if-range	
43	if-unmodified-since	
44	last-modified	

Index	Header Name	Header Value
45	link	
46	location	
47	max-forwards	
48	proxy-authenticate	
49	proxy-authorization	
50	range	
51	referer	
52	refresh	
53	retry-after	
54	server	
55	set-cookie	
56	strict-transport-security	
57	transfer-encoding	
58	user-agent	
59	vary	
60	via	
61	www-authenticate	