H_2DoS: An Application-Layer DoS Attack Towards HTTP/2 Protocol

Xiang Ling[1](✉), Chunming Wu[1], Shouling Ji[1,3], and Meng Han[2]

[1] Zhejiang University, Hangzhou, China
{lingxiang,wuchunming,sji}@zju.edu.cn
[2] Kennesaw State University, Kennesaw, GA, USA
menghan@kennesaw.edu
[3] Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China

Abstract. HTTP/2, as the latest version of application layer protocol, is experiencing an exponentially increasing adoption by both servers and browsers. Due to the new features introduced by HTTP/2, many security threats emerge in the deployment of HTTP/2. In this paper, we focus on application-layer DoS attacks in HTTP/2 and present a novel H_2DoS attack that exploits multiplexing and flow-control mechanisms of HTTP/2. We first perform a large-scale measurement to investigate the deployment of HTTP/2. Then, based on measurement results, we test H_2DoS under a general experimental setting, where the server-side HTTP/2 implementation is *nginx*. Our comprehensive tests demonstrate both the feasibility and severity of H_2DoS attack. We find that H_2DoS attack results in completely denying requests from legitimate clients and has severe impacts on victim servers. Our work underscores the emerging security threats arise in HTTP/2, which has significant reference value to other researchers and the security development of HTTP/2.

Keywords: Web security · DoS attack · HTTP/2 protocol

1 Introduction

Hypertext Transfer Protocol (HTTP) is a dominant and fundamental application protocol, and it powers the data communication on the Internet. Recently, the latest version of HTTP protocol - HTTP/2 [1] has been standardized and received much attention as it can reduce the load latency of web pages by addressing some performance inhibitors inherent in HTTP/1.1 and HTTPS [8]. HTTP/2 protocol is primarily designed for improving performance by introducing new features, however, which can result in new and potential security threats. Those security threats introduced by HTTP/2 may have damaging effects on the Internet in terms of both end users and web servers, because the current HTTP/2 protocol has been adopted by most major browsers and many websites [8]. This brings up a significant challenge of how to explore new security threats against HTTP/2, and motivates us to begin the research of this paper.

The application-layer Denial-of-Service (DoS) attack is a form of DoS attacks where attackers target at the application-layer of web servers. By exploring characteristics and vulnerabilities of application layer protocols, application-layer DoS attacks aim to exhaust server resources that the application requires to function properly [23]. The application-layer DoS has become one of the most damaging attacks to threat the Internet ecosystem [5] in 2016 and was believed to increasingly escalate in the future. Since the HTTP/2 protocol is a new and significant part of web servers in terms of application layer protocols, HTTP/2 is also supposed to face application-layer DoS attacks. In this paper, we narrow our research scope and focus on application-layer DoS attacks towards the HTTP/2 protocol. We present a novel **H_2DoS** attack, which is the first real application-layer DoS attack targeting at HTTP/2-enabled web servers. By exploiting multiplexing and flow-control mechanisms in HTTP/2, the H_2DoS attack can completely deny legitimate users from accessing the victim server. Moreover, this attack also inflicts more severe impacts on server resources compared with other application-layer DoS attacks, which can result in severe damages to web servers.

Concretely, to investigate the potential extent of application-layer DoS attacks against HTTP/2 protocol in practice, we first perform a large scale measurement to understand the current deployment and implementation of HTTP/2. We find that 14% of Alexa's top million websites [11] have already begun to support HTTP/2 protocol. Moreover, most of these websites adopt nginx [17] as the server-side implementation, which can be strongly affected by our H_2DoS attack. Then, we analyze two new features introduced by HTTP/2: flow-control and multiplexing mechanism, and find that both of them are vulnerable to application-layer DoS attacks. Based on those analyses, we propose the novel H_2DoS attack which exploits both flow-control and multiplexing mechanisms. Our proposed H_2DoS attack can disrupt or even completely deny legitimate user from accessing the victim web servers. Next, we examine both the feasibility and severity of H_2DoS attack in our experiments. Our experimental tests show that victim web servers reply with HTTP 500 (Internal Server Error) code to legitimate users during H_2DoS attack. This result indicates that a real denial of service takes place on victim web servers. Even worse, we find that H_2DoS attack can massively consume server resources, and compared with other application-layer DoS attacks, it inflicts more severe impacts on the performance of victim servers. Overall, the main contributions of this paper can be summarized as follows:

- We provide a comprehensive security analysis of the HTTP/2 protocol specification, especially focusing on its multiplexing and flow-control mechanisms. According to our analysis, we find that both multiplexing and flow-control mechanisms are vulnerable to application-layer DoS attacks.
- We propose a novel application-layer DoS attack against HTTP/2, H_2DoS. H_2DoS exploits vulnerable multiplexing and flow-control mechanism of HTTP/2 protocol, and therefore can result in denial of service on victim servers.

– We systematically validate H_2DoS attack (feasibility), and evaluate its impact (severity) by performing extensive experiments, which to the best of our knowledge is first such an attempt.

The rest of this paper is organized as follows. We first review HTTP/2 and application-layer DoS attacks in Sect. 2. Next, in Sect. 3 we briefly describe current deployment and implementation information of HTTP/2 in practice. Section 4 presents the threat model of H_2DoS attack in detail. We examine both the feasibility and severity of H_2DoS attack through extensive experiments in Sect. 5. We also give further discussions on mitigation for H_2DoS attack and summarize the related works in Sects. 6 and 7. Finally, the work is concluded and the future work is addressed in Sect. 8.

2 Background

2.1 Application-Layer DoS Attack

Denial-of-service (DoS) attack is one of the most damaging attacks as it intends to deny legitimate users from accessing network resources and destroy the Internet ecosystem. Originally, DoS attacks basically mean network-layer DoS attacks, which mostly abuse TCP, UDP and ICMP protocols to exhaust network resources of the victim (*e.g.*, bandwidth, sockets, *etc.*) and further deny its services. However, this kind of DoS attacks has been fully studied for years and already been mitigated by many industry solutions. In order to evade such mitigation solutions, DoS attacks have been evolved to sophisticated application-layer DoS attack [22] as their stealthier appearance and lower attack cost than traditional network-layer DoS attacks.

Concretely, application-layer DoS attacks focus on disrupting or even completely denying legitimate users from accessing the victim web server by exhausting its resources, including not only network bandwidth and sockets, but also connections, CPU, memory, I/O bandwidth, *etc.* There are basically two types of application-layer DoS attacks - HTTP DoS and HTTPS DoS attacks, as both of them are based on two dominant protocols that used by the application layer.

1. **HTTP DoS.** HTTP DoS attacks normally exploit seemingly legitimate HTTP GET/POST requests to occupy all available HTTP connections that permitted on the web server. **Slowloris** [6] is one of the most effective HTTP DoS attacks against many popular types of web server softwares like Apache and nginx. If an attacker initiates an HTTP request to open several connections to a server and periodically feeds the server with data before reaching timeout, the HTTP connection would remain to open until the attacker closes it. Ultimately, it easily fulfills the maximum concurrent connections of the web server and takes the server down.
2. **HTTPS DoS.** HTTPS layers HTTP on top of Transport Layer Security (TLS), which encrypts all communication data for end-to-end security and easily evades security managements [13]. Hence, HTTPS DoS can further

challenge many existing web application firewall detection solutions, as most of the solutions do not actually inspect encrypted traffics [10]. In addition to bypassing DoS prevention efforts, as encrypted HTTP attacks add burden of encryption and decryption, HTTPS DoS can exhaust all server resources by leveraging all possible approaches [7], such as encrypted SSL floods, SSL renegotiations and HTTPS floods.

Currently, the application-layer DoS attack increasingly escalates and has become a significantly severe threat for web servers. According to Radware Emergency Response Team's (ERT) annual report [5], 63% of its respondents have experienced application-layer based attacks in 2016, and 43% of experienced an HTTP flood, while 36% experienced an HTTPS flood.

2.2 HTTP/2 Protocol

Overview. HTTP/2 protocol is the latest version of HTTP protocol that dramatically reduces the load latency of web pages by addressing some performance inhibitors inherent in HTTP/1.1 or HTTPS. Shortly after being standardized as RFC 7540 [1] in 2015, HTTP/2 is experiencing an exponential growing industry adoption with both servers and browsers. Originally, HTTP/2 protocol mainly succeeds to SPDY [2], which is an experimental application-layer protocol designed by Google as a replacement for more efficient communication transmission [9]. Basically, HTTP/2 reserves majority of SPDY protocol, except with several changes, such as a new header compression for HTTP/2 - HPACK [3] instead of gzip or deflate used by SPDY.

The primary goal of HTTP/2 is to reduce the web page load latency by providing an optimized communication transmission. HTTP/2 enables fully request and response multiplexing, minimizes transmission overhead with support of flow-control and server push, and replaces with a less redundant header field compression method. Below, we detail three optimized features of HTTP/2 that related to our study and omit the other features.

1. **Frame Unit.** HTTP/2 protocol introduces the frame unit as the basic protocol unit to be exchanged between servers and browsers. There are ten different types of frames used to serve distinct purposes in the establishment and management of HTTP/2 connections or streams. For instance, *WINDOW_UPDATE* is a frame that used for HTTP/2 flow-control mechanisms. But in this paper, we will manipulate it and other frames to create a new application-layer DoS attack against HTTP/2.
2. **Multiplexing.** HTTP/2 initiates only one single TCP connection to one domain and multiples HTTP requests and responses. HTTP/2 can dramatically reduce the load latency of web pages, as the multiplexing feature not only reduces the number of TCP connections but also SSL encryption overhead at both browser and server sides. For the same reason, the multiplexing feature also becomes actually an important amplification factor to enhance the impact of our attack.

3. **Flow-Control.** Flow-control is one of the most distinguish features of HTTP/2, which can be used for both individual streams and the whole connection. The flow-control feature ensures that streams on the same TCP connection do not negatively interface with each other. The flow-control also allows customized algorithms to optimize data transmission between servers and browsers, especially when their resources are limited. This actually poses a severe security threat that an HTTP/2 connection can demand a greater resources to operate than an HTTP/1.1 connection.

The above three features of HTTP/2 protocol enable a significant reduction of page loading time and mitigate some existing security threats [4] to some extent. However, adopting a new protocol can bring new security threats since new features of HTTP/2 extend the new attack surface towards clients or servers. In fact, both multiplexing and flow-control features described above are vulnerable to the application-layer DoS attack, which motivates us to propose our H_2DoS attack. We will discuss those new features and their potential vulnerabilities in more detail in Sect. 4.

3 HTTP/2 Current Deployment

To investigate the potential extent of application-layer DoS attacks against HTTP/2 protocol in practice, a large-scale measurement is performed to investigate the current HTTP/2 deployment and its implementation. To this end, we build a measurement platform to conduct real crawling of the exact domain of all sites provided by Alexa top one million ranking list [11]. In this section, the Application-Layer Protocol Negotiation (ALPN) extension [12] is first employed during TLS handshake to measure how many websites adopt HTTP/2. Then, we extract the HTTP/2 implementation software information within the established HTTP/2 connection. Notice that our statistics results are all based on experiments and observations from January 10th to January 13th in 2017.

3.1 Measurement Setup

Generally, establishing an HTTP connection can be divided into three phrases: TCP handshake, TLS handshake, and application-layer communication. Figure 1 illustrates the process of HTTP/2 connection establishment. To answer the first question that how many websites adopt HTTP/2 protocol, we observe the ALPN extension [12] of TLS handshake within an HTTP/2 connection in step ❸ and ❹ of the Fig. 1. The ALPN extension is used for application-layer protocol negotiation in exchange of *Hello* message: the client provides a list of optional protocols which it supports and the server can respond with a selected protocol that want to use. Therefore in this measurement, if a website negotiates **"h2"** as the selected protocol within ALPN, we consider that the website adopts HTTP/2 protocol in terms of the application layer.

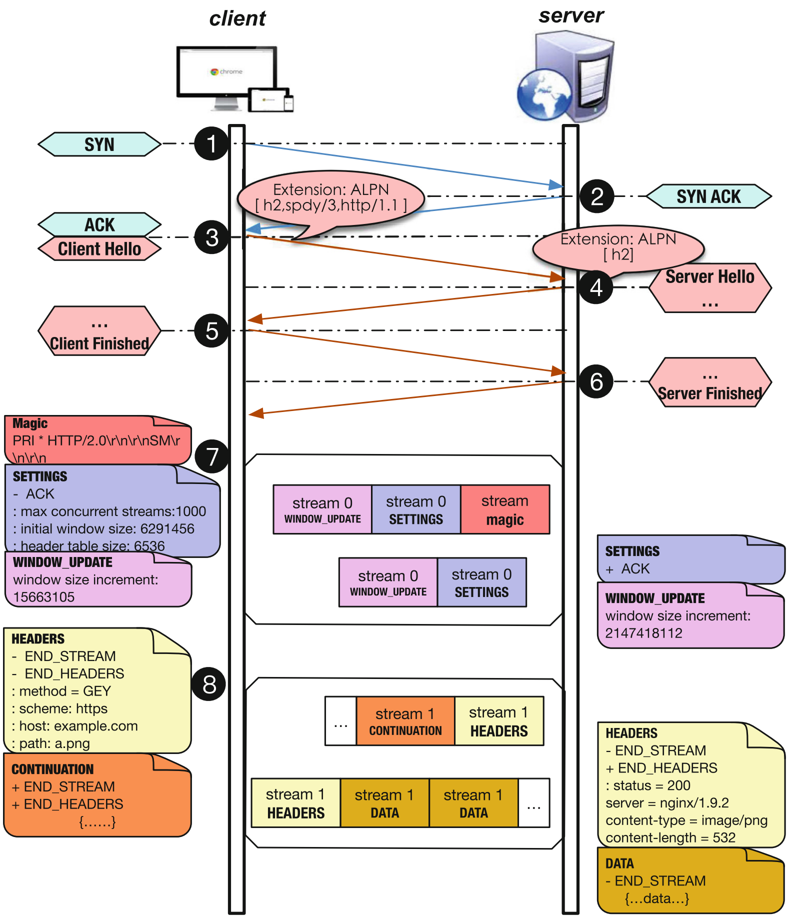

client
server
SYN
1
2
SYN ACK
Extension: ALPN [h2,spdy/3,http/1.1]
ACK
Client Hello
3
Extension: ALPN [h2]
4
Server Hello ...
... Client Finished
5
6
... Server Finished
Magic PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
7
SETTINGS - ACK : max concurrent streams:1000 : initial window size: 6291456 : header table size: 6536
WINDOW_UPDATE window size increment: 15663105
stream 0 WINDOW_UPDATE
stream 0 SETTINGS
stream magic
stream 0 WINDOW_UPDATE
stream 0 SETTINGS
SETTINGS + ACK
WINDOW_UPDATE window size increment: 2147418112
HEADERS - END_STREAM - END_HEADERS : method = GEY : scheme: https : host: example.com : path: a.png
8
CONTINUATION + END_STREAM + END_HEADERS {......}
...
stream 1 CONTINUATION
stream 1 HEADERS
stream 1 HEADERS
stream 1 DATA
stream 1 DATA
...
HEADERS - END_STREAM + END_HEADERS : status = 200 server = nginx/1.9.2 content-type = image/png content-length = 532
DATA - END_STREAM {...data...}

Fig. 1. Establishment process of an HTTP/2 connection: TCP handshake is first completed in step ❶–❸, followed by a TLS handshake in step ❹–❻. Then after step ❼ and ❽, the HTTP/2 connection is finally established.

Another question is how about implementations of websites enabled HTTP/2. Basically, an HTTP/2 connection starts with sending the connection preface, called *Magic* frame and followed by a *SETTING* frame and/or *WINDOW_UPDATE* frame in step ❼, which used for flow-control mechanism in HTTP/2. After that, by sending a *HEADERS* frame, we could receive a *HEADERS* response frame. Normally in *HEADERS* response frame, we find the exact implementation software information of the website through HTTP **"Server"** header field.

Table 1. Summary of protocols deployment of Alexa's top million websites

Protocol	Description	#	%
HTTP/1.x	Websites that not support TLS	465,693	46.57%
HTTPS(pure)	Websites that purely support HTTP/1.1 over TLS	355,025	35.50%
HTTP/2	Websites that support HTTP/2 over TLS	143,471	14.35%
SPDY	Websites that announce support SPDY	25	- - -
Others	Websites that cannot accessible or only support other protocols like QUIC *etc.*	35,786	3.58%

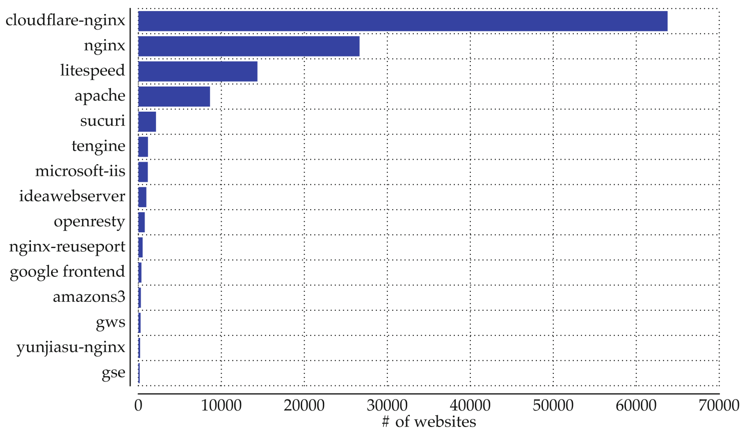


Fig. 2. Top 15 popular HTTP/2 server implementations

3.2 Measurement Result

HTTP/2 Deployment. Table 1 summarizes the current protocols deployment of Alexa top million websites, including HTTP/1.x, pure HTTPS, HTTP/2, SPDY and others. The table shows that there are around 50% websites supporting TLS connection in their web servers, in which 28.78% of websites have already supported HTTP/2 protocol via ALPN extension in TLS handshake. In addition, we also report that 14.35% of websites have supported HTTP/2 in top million websites. It implies that numerous website servers that support HTTP/2 are facing potential security threats including application-layer DoS attacks. We also believe that there will be more websites adopt HTTP/2 along with more security threats, which can leads to severe damage to the Internet.

HTTP/2 Implementation. We record HTTP/2 implementation information among all HTTP/2-enabled websites via HTTP/2 *HEADERS* frame in our measurement platform, in which we observe more than 414 different kinds of server implementations in total. For visibility, Fig. 2 plots top 15 popular server-side implementation softwares powering HTTP/2 websites. In spite that *cloudflare-nginx* and *nginx* are top two that used as implementation softwares of HTTP/2 websites, other variants like *tengine*, *nginx-reuseport* and *yunjiasu-nginx* also support thousands of HTTP/2 websites. As nginx community is the most prevalent HTTP/2 implementation that adopted by websites on the Internet, we choose the latest nginx stable implementation as the server-side HTTP/2 implementation in latter experiments.

4 Threat Model: H_2DoS Attack

In this session, we begin with a comprehensive security analysis of HTTP/2 flow-control mechanism and then present our novel H_2DoS attack which exploits multiplexing and flow-control mechanisms of HTTP/2 protocol in details.

4.1 HTTP/2 Flow-Control Mechanism Analysis

Flow-control mechanism is one of the most distinctive features enabled by HTTP/2 protocol that attempts to optimize the traffic transmission between browsers and servers. Generally, there are two types of frame that used for application-level flow-control in HTTP/2 protocol: *WINDOW_UPDATE* and *SETTINGS* frame. In an established HTTP/2 connection, the server and the client exchange configuration parameters including some flow-control parameters in *SETTINGS* frame, then in more fine-grained frame layer, the window size of flow-control can be updated by *WINDOW_UPDATE* frame that applied to a single frame or all frames in the HTTP/2 connection.

SETTINGS Frame. *SETTINGS* frame, which is used to inform the opposite (client or server) of configuration parameters, normally follows *Magic* frame at the start of an established HTTP/2 connection. If the stream identifier (Stream ID) of *SETTINGS* frame is set to be 0×0, It means that *SETTINGS* frame will apply to an entirely HTTP/2 connection instead of a single stream. In HTTP/2 protocol specification [1], there are totally 6 defined configuration parameters, in which 3 of them related to flow-control mechanism in an HTTP/2 connection. Table 2 shows the three flow-control related parameters in *SETTINGS* frame.

WINDOW_UPDATE Frame. The primary goal of *WINDOW_UPDATE* frame is to implement the flow-control mechanism that prevents from exceeding capacity of the receiver in an HTTP/2 connection. *WINDOW_UPDATE* in HTTP/2 protocol specification [1] has two levels of application, one level operates in an individual stream with a specific stream ID, while another level operates in an entire HTTP/2 connection whose stream ID is zero. Basically, HTTP/2 specification requires that receiver of *WINDOW_UPDATE* frame must re-calculate the corresponding window size according to the 31-bit "Window Size Increment" field included in *WINDOW_UPDATE* frame. For instance, if the server-side advertises its initial window size in *SETTINGS* frame to be 16 KB

Table 2. Flow-control parameters in SETTINGS frame

Parameters	*SETTINGS_MAX_CONCURRENT_STREAMS*	*SETTINGS_INITIAL_WINDOW_SIZE*	*SETTINGS_MAX_FRAME_SIZE*
Functionality	Defining the maximum number of concurrent streams that the sender permits receiver to create in this HTTP/2 connection	Defining the initial window size of streams in this HTTP/2 connection	Defining the maximum frame payload size that the sender allows to receive in this HTTP/2 connection
Value	no limit(0–$2^{31}-1$), but recommended value is ≥ 10	no limit(0–$2^{31}-1$), initial value is $2^{16}-1$	range(2^{14}–$2^{24}-1$), initial value is 2^{14}

and sets "Window Size Increment" 5 KB in *WINDOW_UPDATE* frame, then the window size of server becomes 21 KB. In addition, the window size only applies to *DATA* frame, which means that the flow-control mechanism affected by the window size only constraints *DATA* frame instead of other frames like *HEADERS* frame.

To summarize, both *SETTINGS* and *WINDOW_UPDATE* frame play an important role in HTTP/2 flow-control mechanism by means of altering or updating window size kept by both sides in a stream or connection. Naturally, "Window Size Increment" field in *WINDOW_UPDATE* frame is used to increase window size that receiver can process, while sending new *SETTING* frame with smaller initial window size can cause window size reduces. Even window size can be negative because of receiving *DATA* frame will consume window size, which can make processes in streams stalled in the end.

4.2 H_2DoS Attack Presentation

Conceptually, multiplexing and flow-control are two novel essential mechanisms of HTTP/2 that introduced to improve web performance, however these excellent mechanisms come at an expense that introducing new security threats into servers and clients. In order to understand security threats evolved from HTTP/2, we carry on a comprehensive security analysis of HTTP/2 protocol specification [1] and find that both multiplexing and flow-control mechanisms are vulnerable to application-layer DoS attacks. If exploiting multiplexing and flow-control mechanisms, a DoS attack named **H_2DoS** can be easily launched by one malicious client to attack the victim web server.

The basic idea of H_2DoS attack is natural and straightforward: a massive number of HTTP/2 requests with limited receiving capacity are sent to consume as many resources as possible, or even result in denial-of-service. To this end, H_2DoS attack exploits two following important amplification factors that derived from both multiplexing and flow-control mechanisms.

– One amplification factor is to exploit HTTP/2 multiplexing mechanism since HTTP/2 enables multiplexing vast number of streams over a single TCP connection. Even though the attacker has to initial as many TCP connections

as the victim, in HTTP/2 connection each TCP connection can maintain large amount of streams to amplify malicious HTTP GET requests.
- Another amplification factor is to limit the receive processing window size to a small size, which results in stalling all send processes of victim until the entire response data is transmitted and thus occupying lots of server resources.

Figure 3 presents how a malicious client attack the victim web server by launching application-level H_2DoS attack and its attack proceeds as follows:

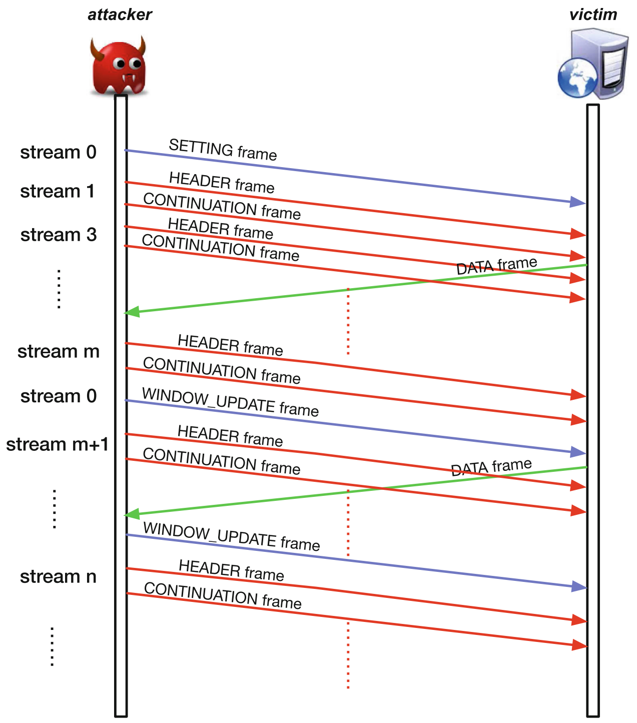


Fig. 3. H_2DoS: HTTP/2 application-level DoS attack presentation

1. Before H_2DoS attack, both TCP handshake and TLS handshake must be completed within a malicious client (called *attacker*) and a web server (called *victim*), followed by a *Magic* frame that initialized for establishing an HTTP/2 connection at first.
2. Then, the *attacker* sends a *SETTINGS* frame in stream 0. It means that the *SETTINGS* frame applies to the entire HTTP/2 connection. Two configuration parameters that mentioned in Table 2 are set up in this attack:
   (a) one parameter is *SETTINGS_MAX_CONCURRENT_STREAM*, which is supposed to set to a big number as it specifies the maximum number of streams created by the victim. And also, more streams in a connection means more threads allocated will be consumed. In fact, the maximum number of streams that the *attacker* can exploit is depend on

the *SETTINGS* frame of *victim*. What attack can do is to acknowledge the biggest *SETTINGS_MAX_CONCURRENT_STREAM* value among all *SETTINGS* frames and open as many streams as it allows.

(b) another parameter is *SETTINGS_INITIAL_WINDOW_SIZE*, which should be set as small as possible that allowed in a specific implementation software of HTTP/2 protocol in order to make the process of HTTP response slower or even make the *victim* stalled.

3. The *attacker* next constructs an HTTP/2 GET request in stream 1, which consists of a *HEADERS* frame and one or more subsequent *CONTINUATION* frames. For the necessity of the subsequent *CONTINUATION* frames, we enable the HTTP/2 GET request with a long header field, only small part of it is sent in *HEADERS* frame and the other is sent in one or more *CONTINUATION* frames.
4. Owing to multiplexing mechanism of HTTP/2, we repeat sending carefully constructed HTTP/2 request streams as above one after the other in odd-numbered stream ID (1, 3, 5, ...).
5. To prevent *victim* web server from rejecting the HTTP/2 request, *attacker* can send *WINDOW_UPDATE* frame in stream 0 periodically with a "Window Size Increment" field, but with a small size.
6. Since above processes are all in one single TCP connection, we can amplify the attack consequence by opening more than one single TCP connection.

In short, to create an effective application-layer DoS attack against HTTP/2, the whole H_2DoS attack exploits two amplification factors that derived from vulnerabilities of HTTP/2 in terms of both multiplexing and flow-control mechanisms. As H_2DoS attack repeats sending HTTP/2 GET streams to the *victim* infinitely, this attack can instantaneously occupy all available connections of the *victim*. In theory, the starvation of all available connections is the root cause of H_2DoS attack. In addition, the H_2DoS attack can also consume as much server resources as possible, which further strengthens the effect of DoS attack.

5 Experiments and Results

In this session, we seek to answer two key questions:

- Is H_2DoS attack a feasible DoS attack in real attack scenarios?
- Does H_2DoS attack have more severe impact on the targeted *victim* compared with other popular application-level DoS attacks? That is, can it become an underlying severe factor of DDoS attack?

The intent of answering above two questions is to demonstrate both the feasibility and severity of it, respectively. To this end, we first present our experiment setup, then observe experiment results from our experiments and analyze the feasibility and severity around them.

5.1 Experimental Setup

Our experiment setup consists of one *victim* web server and two clients: *attacker* and *benign user* as shown in Fig. 4, respectively. Both the *attacker* and *benign user* connect to *victim* server in HTTP/2. The *attacker* is a client that launches H_2DoS attack with malicious attack scripts, while the *benign user* is a normal client that used for testing whether the *victim* is in service. The *victim* is the web server that enabled with an HTTP/2 implementation and can be accessed in HTTP/2 connections. Table 3 summarizes detail configurations of the experiment environment. As mentioned in Sect. 3.2, nginx is the most widely adoption in HTTP/2 implementations, therefore we choose the latest nginx stable version to run on the *victim* server during the experiment execution.

Table 3. Detail configurations of the experiment environment

Configurations	*victim*	*attacker*	*benign user*
Operating system	Ubuntu 16.04.1 LTS	Ubuntu 16.04.1 LTS	Mac OSX 10.11.6
Processor	2 * Intel(R) Core(TM) i5-4590 CPU @3.30 GHz	2 * Intel(R) Core(TM) i5-4590 CPU @3.30 GHz	2.7 GHz Intel Core i5
Memory	4 GB	4 GB	8 GB
HTTP/2 implementation	nginx/1.10.0(stable)	Golang standard http/2 library [16]	Google Chrome 58.0
Others	Built with OpenSSL 1.0.2g TLS SNI support enabled	H_2DoS attack implementation in Go language	Google plug-in for connection checking

Figure 4 illustrates a straightforward process of H_2DoS attack: the *attacker* first launches the malicious attack script towards the *victim* in step ❶, where the malicious attack script implements H_2DoS attack described in Fig. 3 as well as other application-layer DoS attacks introduced in Sect. 2.1 for comparison. In *victim* server, there is a performance monitor program that used to monitor the server performances and record them down. Then during the attack, *benign user* periodically request to access resources of *victim* using normal browser in step ❷. Finally we check what contents that replied in the context of browser request before in step ❸. If we get errors instead of normal contents from received contents in *benign user*, we take it as the Denial-of-Service of *victim* that attacked by H_2DoS attack.

5.2 Experiment Result and Analysis

To answer the above two questions, we analyze in more detail for both the feasibility and severity of H_2DoS attack based on our observed experimental results.

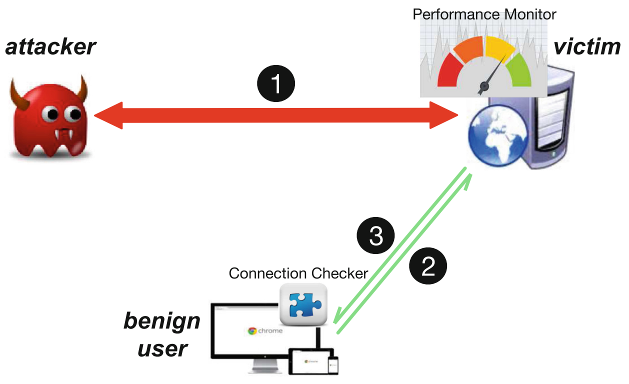
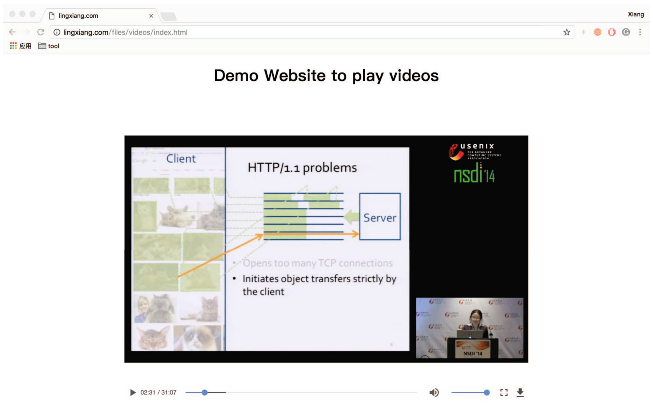


Fig. 4. H_2DoS attack experiment setup

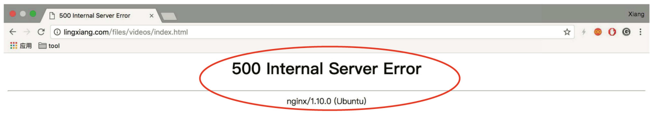
Feasibility. We offer evidence of the feasibility of H_2DoS attack by checking whether the *victim* web server is always available for a *benign user* during H_2DoS attack. The connection checker illustrated in Fig. 4 is a customized browser plug-in that used for application-layer connection checking and recording. Once the H_2DoS attack is launched against *victim* web server, we start to observe and record what we receive from the web server in the connection checker. More specifically, if we obtain an entire webpage with HTTP 200 (OK) status code in responses, we consider the *victim* web server is available in service. By contrast, if we obtain any error webpage with HTTP 500 (Internal Server Error) or other 5XX (Server Error) status codes [14], it means that *victim* itself has an error and crashes down, which is a kind of Denial-of-Service atta. Figure 5(a) and (b) visually show the content of *benign user* that received from *victim* web server before and after H_2DoS attack, respectively. We observe that the *benign user* receives an entire webpage from the *victim* before H_2DoS attack, while after H_2DoS attack the *benign user* receives an error webpage with HTTP 500 (Internal Server Error) status code. These observations indicate that H_2DoS attack indeed takes effect into the *victim* web server in terms of denial of service attacks.

The root cause of application-layer DoS attack is that H_2DoS attack can occupy all available connections of the *victim* server and all streams are possibly stuck on exhausted connection or stream windo. Figure 5(c) further depicts that in our 30-min experiment as long as H_2DoS attack starts from *attacker* client to *victim* web server, the HTTP response status code of *benign user* quickly changes from 200 to 500. The 500 status code is used for internal server error when the server suffers from starvation of connections, which prevents the server from replying any request. From what we have observed and analyzed above, the feasibility of H_2DoS attack is fully demonstrated in our experiments.

Severity. For severity, we measure the impact of H_2DoS attack towards *victim* web server and in what extent it enhances the severity if H_2DoS is converted to


lingxiang.com
Xiang
lingxiang.com/files/videos/index.html
应用 tool
Demo Website to play videos
Client
HTTP/1.1 problems
Server
Opens too many TCP connections
Initiates object transfers strictly by the client
usenix
nsdi'14
02:31 / 31:07

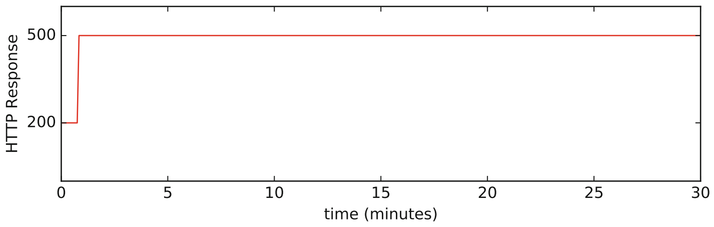
(a) Content of *benign user* received from the *victim* **before** H_2DoS attack.


500 Internal Server Error
Xiang
lingxiang.com/files/videos/index.html
应用 tool

500 Internal Server Error

nginx/1.10.0 (Ubuntu)

(b) Content of *benign user* received from the *victim* **after** H_2DoS attack.

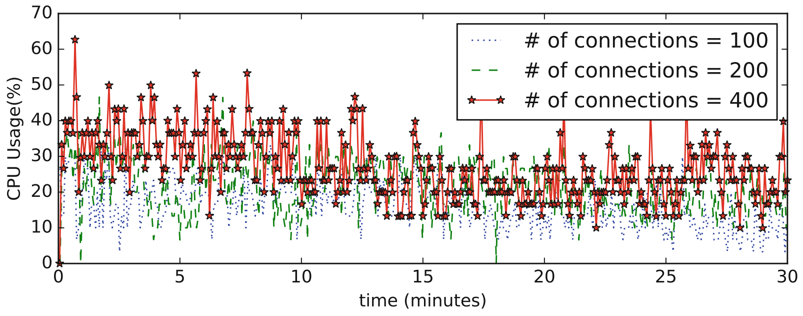


(c) HTTP response status code of *benign user* during H_2DoS attack

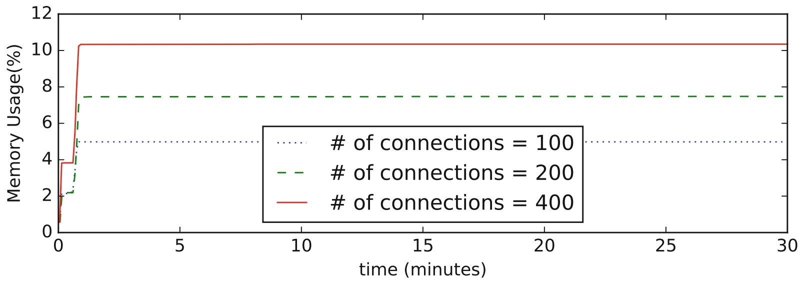
Fig. 5. Observations on *benign user* that received from *victim* during H_2DoS attack

Distributed Denial-of-Service (DDoS) attack against *victim*. Application-layer DDoS attack generally consumes less bandwidth and are stealthier in nature compared with other network-based DDoS attacks. Application-layer DDoS attack mainly focuses on disrupting legitimate user services by exhausting the server resources [22] like CPU and memory as much as possible.

Hence in the paper, we choose two key factors: *CPU* and *Memory* to measure the application-layer DoS attack impact. We obtain both *CPU* usage and *Memory* usage of *victim* server with the performance monitor program illustrated in Fig. 4. And the performance monitor is developed based on ***psutil*** [15], a process and system utilities library in Python. Intuitively, larger *CPU* or *Memory* usage consumption will result in larger probability of denying other benign users as



(a) CPU usage on the *victim* web server.



(b) Memory usage on the *victim* web server.

Fig. 6. Observations of resources consumption on *victim* when attacked by H_2DoS with different number of TCP connections.

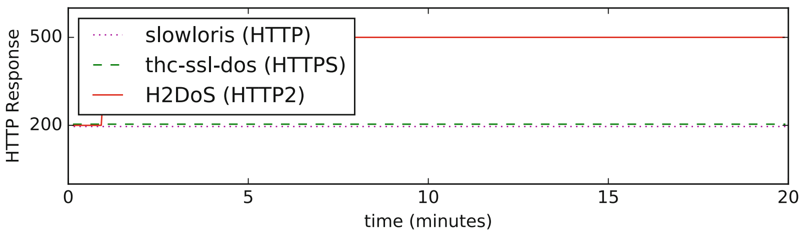
well as larger attack severity. In this part, we conduct two sets of experiments and analyze the attack impact of H_2DoS attack to confirm the severity intuition.

1. **Severe impact of H_2DoS attack.** In this experiment, we analyze how severe the impact of H_2DoS attack is in terms of *CPU* and *Memory* usage. Once the malicious client *attacker* begins to launch H_2DoS attack against *victim* server, the performance monitor is enabled to monitor both *CPU* and *Memory* usage of *victim* server. Besides, we increase the number of TCP connection within the same H_2DoS attack in our experiment, in order to further observe how the impact of H_2DoS attack behaves in regard to TCP connections.
   Figure 6(a) and (b) illustrate that H_2DoS attack can maliciously consume large volume of CPU and memory on the whole. Specifically, as depicted in Fig. 6(a), the *CPU* usage is very high in the first several minutes and gradually stabilizes later. That is because after several minutes the H_2DoS attack takes effect and the *victim* starts to reply error code instead of real contents, which result in less *CPU* usage later. However, as depicted in Fig. 6(b), the *Memory* usage is nearly unchanged except for the beginning of H_2DoS attack. Furthermore when comparing different number of connections in H_2DoS attack, we

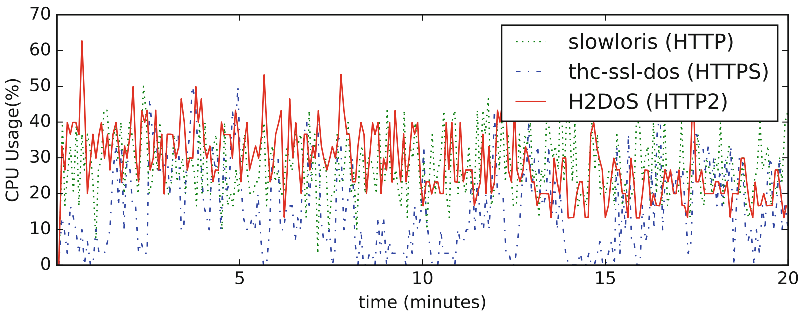
observe that the percentage of *CPU* consumption increases with the number of TCP connections in general, while the percentage of *Memory* consumption is amplified by connections all the time in our experiment.

2. **Severe impact of H_2DoS attack Versus Others.** As observed above, H_2DoS attack has significantly severe impact on *victim* server. But how does H_2DoS attack compare with other application-layer DoS attacks is still a challenge question. To evaluate the impact of H_2DoS attack comparing with other application-layer DoS attacks, we first fix the number of TCP connections at 400 and examine the *CPU* and *Memory* usage of *victim* server during attack duration in our experiment. Next, we choose **slowloris** [6], **thc-ssl-dos** [7] and our H_2DoS attack in regard to application-layer DoS attack based on HTTP/1.1, HTTPS and HTTP/2, respec. Figure 7(b) and (c) show the *CPU* and the *Memory* usage of above three types of application-layer DoS attacks. Specifically, we show our results and evaluations in the following three aspects that related to the impact on *victim*:
   - **Connectivity.** Figure 7(a) shows that H_2DoS can quickly bring down the *victim* web server and replies with status code of HTTP 500 (Internal Server Error) to the *benign* user. However, at both **slowloris** and **thc-ssl-dos** attack duration, the *victim* server provides service with HTTP 200 status code to the *benign* user all the time. As depicted in Fig. 7(b) and (c), we observe that while both the CPU and Memory usage of *victim* caused by H_2DoS attack do not exceed 50% over time in most cases, but H_2DoS leads to a real denial-of-service attack. In fact, either CPU and Memory usage is not the exclusive reason for denial-of-service, the main reason is that H_2DoS occupies all available connections of the *victim* and denies access to legitimate clients.
   - **CPU usage.** As depicted in Fig. 7(b), the H_2DoS attack consumes more CPU than other two attacks on average, even though it decreases gradually after around 10 min and becomes less than **slowloris** attack at the end time of attack duration. One possible reason might be that at later time H_2DoS attack makes *victim* only reply with HTTP 500 error code and not in the service as H_2DoS results in starvation of *victim* connections, while **slowloris** is in the service all the time and maintains the CPU usage.
   - **Memory usage.** As depicted in Fig. 7(c), the H_2DoS attack depletes around ten times memory more than both **slowloris** and **thc-ssl-dos** attack, because H_2DoS can exploit multiplexing mechanism within HTTP/2 to amplify the power of occupying memory resources on the *victim* server.

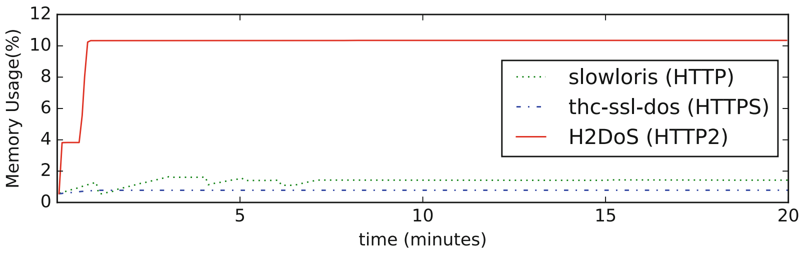
To summarize, the experiments described above analyze in detail that how the H_2DoS attack takes affect on the performance of *victim* web server and demonstrate the feasibility and severity of it in real attack scenarios. From the experiment results, we can conclude that the commercial HTTP/2 implementation nginx can be exploited and severely impacted by H_2DoS attack.



(a) HTTP response status code of *benign user*



(b) CPU usage on the *victim* web server



(c) Memory usage on the *victim* web server

Fig. 7. Observations of HTTP response on *benign user* and resource consumption on *victim* when attacked by different three kinds of application-layer DoS attacks.

6 Discussion

As we have presented, H_2DoS can occupy all available connections of the *victim* and completely deny the legitimate user from accessing the victim web servers. Moreover, H_2DoS attack can also consume more server resources than other application-layer attacks on average in terms of CPU and memory usage. Strictly speaking, this is an implementation and configuration problem of HTTP/2 specification in practice. We have measured that there are many top websites have supported HTTP/2 and therefore the potential impact of the H_2DoS attack is significant. We suggest that websites with such concerns could minimize the impact of H_2DoS attack by limiting the rate of requests and total number of connections from the same client. As we believe that the starvation of connections should not be present in any single benign request, we encourage developers of any deployed website that processes HTTP/2 requests should review their rate and total number with this threat in mind.

7 Related Work

Application-Layer DoS Against HTTP/2 Protocol. The understanding and mitigation of security risks of DoS attack have been an active area of research in recent years as DoS is a continuous critical threat on the current Internet ecosystem. Recently, the research community has gradually shifted their research interest from traditional network-layer based DoS attacks to escalating application-layer DoS attacks. There are lots of studies have been done on application-layer DoS attacks [18–23]. Yi and Yu [18] showed that new application-layer-based DDoS attacks can utilize legitimate HTTP requests to overwhelm victim resources and proposed an anomaly detector to detect such attacks on popular websites traffic. Jazi et al. [23] presented several unique features that characterize application-layer attacks and proposed a nonparametric CUSUM detection algorithm to detect them using found characterizes.

However, previous works of application-layer DoS attacks mostly bases on HTTP/1.1 or HTTPS protocol as well as their defense mechanisms for mitigation. To the best of our knowledge, very few studies focus on application-layer DoS attacks against HTTP/2 protocol and its various implementation softwares. We describe these studies as below.

A report of Imperva Defense Center [24] releases four high-profile vulnerabilities in total on new implementations of HTTP/2 from the major vendors. One of the attacks reported is the slow read attack, which exploits a malicious client to read responses very slowly from HTTP/2-enabled servers. Our work contributes further in this regard by broadly exploring the possibilities of a more general DoS against HTTP/2. We exploit both multiplexing and flow-control mechanisms to create such general application-layer DoS attack: H_2DoS attack, and also systematically validate its feasibility as well as evaluate the impact of it.

Adi et al. [25] firstly presented that it is possible to launch a DoS attack using apparently legitimate but malicious HTTP/2 flash crowd traffic. The malicious HTTP/2 packets was crafted by exploited the "Window Size Increment" value in *WINDOW_UPDATE* frame to model flooding-based attack against the HTTP/2 victim web server, as well as performed four investigations to observe the effect of resource consumption in the victim web server. Unfortunately, they limited their attacks to *WINDOW_UPDATE* frame and ignored other frames that can also be exploited to further amplify the impact of their attack. Instead, we take all frames into consideration and analyze the novel HTTP/2 flow-control and multiplexing mechanisms in details to construct our H_2DoS attack. Moreover, we conduct a systematically experiment instead of four investigation observations to present our attack model and demonstrate its feasibility and severity.

Other Security Threats Against HTTP/2 Protocol. Prior work also has shown others attacks that exploiting new features introduced in HTTP/2. Even before HTTP/2 protocol was standardized, Redelmeier et al. [26] systematically analyzed almost all possible security implications of HTTP/2 and explored a series of potential or known areas of vulnerabilities for HTTP/2, including cross-protocol attacks, intermediary encapsulation attacks and cacheability of pushed resources and so on. (Kate) Pearce and Vincent [29] discussed how we can launch multiplexing attacks over QUIC[1] and within HTTP/2, as well as how to make sense of and defend against H2/QUIC traffic on their network. It also indicated that security tools must keep up with technique updating and people should be aware of. Van Goethem and Vanhoef [28] introduced HEIST techniques and carried out side-channel attacks against SSL/TLS purely in the browser to directly infer the length of the plaintext message. By abusing new features of HTTP/2, they found that the attack remained possible and even further increased the impact of HEIST. Larsen and Villamil [27] introduced threats and vulnerabilities discovered during the course of their research on the HTTP/2 protocol and released first public HTTP/2 fuzzer - **http2fuzz**, which intended to find more security vulnerabilities before HTTP/2 implementations were widely deployed.

8 Conclusion and Future Work

In this paper, we present a novel DoS attack against HTTP/2, H_2DoS, which can result in severe damages to web servers. First, we give the introduction of several new features of HTTP/2 protocol and present how the current HTTP/2 is deployed in practice by performing a large-scale measurement on Alexa top million websites. Second, we analyze the flow-control mechanism and propose the novel H_2DoS application-layer DoS attack, which can disrupt or even completely deny legitimate users from accessing the victim web server. Finally, we conduct a comprehensive study on the feasibility and severity of H_2DoS attack in real attack scenarios. We demonstrate that the malicious client can easily launch H_2DoS attack against web servers which support HTTP/2 protocol and make the service unavailable or massively consume server resources. We also compare our H_2DoS attack with other application-layer DoS attacks, which show H_2DoS attack has more severe impact on the same victim web server.

In future work, we plan to explore more other vulnerabilities and attacks against the HTTP/2 protocol of web security. As new features usually comes unintentionally at the expense of new or unknown security threats, we believe that HTTP/2 with new features also brings a lot of new attack vulnerabilities. Since the proposed H_2DoS attack poses serve threats to HTTP/2, we hope our work will provide insight into those security issues and motivate to study other potential security threats against HTTP/2. Finally, we also plan to open source our H_2DoS attack implementation to further promote the research on web security of HTTP/2 protocol.

[1] The QUIC Projects https://www.chromium.org/quic.

Acknowledgments. This work is supported by the National Key Research and Development Program of China under No. 2016YFB0800102 and 2016YFB0800201, the National High Technology Research and Development Program of China under No. 2015AA015602 and 2015AA016103, the Key Research and Development Program of Zhejiang Province under No. 2017C01064 and 2017C01055, the Fundamental Research Funds for the Central Universities, the NSFC under program No. 61772466, the Alibaba-Zhejiang University Joint Research Institute for Frontier Technologies (A.Z.F.T.) under Program No. XT622017000118, and the CCF-Tencent Open Research Fund under No. AGR20160109.

References

1. Mike, B., Roberto, P., Thomson, M: RFC 7540: hypertext transfer protocol version 2 (HTTP/2). Internet Engineering Task Force (IETF), Google Inc. (2015)
2. SPDY: An experimental protocol for a faster web. https://www.chromium.org/spdy/spdy-whitepaper
3. Roberto, P., Ruellan, H.: HPACK: Header Compression for HTTP/2. No. RFC 7541, Internet Engineering Task Force (2015)
4. Thai, D., Juliano, R.: The CRIME attack. In: Ekoparty Security Conference (2012)
5. Radware Emergency Response Team: Global Application & Network Security Report 2016–2017 (2016). https://www.radware.com/ert-report-2016/
6. RSnake, Kinsella, J.: Slowloris HTTP DoS. https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris
7. THC-SSL-DOS. http://kalilinuxtutorials.com/thc-ssl-dos/
8. Varvello, M., Schomp, K., Naylor, D., Blackburn, J., Finamore, A., Papagiannaki, K.: Is the web HTTP/2 yet? In: Karagiannis, T., Dimitropoulos, X. (eds.) PAM 2016. LNCS, vol. 9631, pp. 218–232. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30505-9_17
9. Wang, X.S., Balasubramanian, A., Krishnamurthy, A., Wetherall, D.: How speedy is SPDY? In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 387–399. Usenix Association (2014)
10. Meyer, C., Schwenk, J.: SoK: lessons learned from SSL/TLS attacks. In: Kim, Y., Lee, H., Perrig, A. (eds.) WISA 2013. LNCS, vol. 8267, pp. 189–209. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05149-9_12
11. Alexa Top Sites, September 2016. http://www.alexa.com/topsites
12. Friedl, S., Popov, A., Langley, A., Stephan, E.: Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension, No. RFC 7301, IETF (2014)
13. Dierks, T.: The Transport Layer Security (TLS) Protocol Version 1.2, No. RFC 5246, IETF (2008)
14. David, G., Totty, B.: HTTP: The Definitive Guide. O'Reilly Media, Sebastopol (2002)
15. Rodola, G.: A cross-platform process and system utilities module for Python. https://github.com/giampaolo/psutil
16. Fitzpatrick, B.: Http2 in GoDoc. https://godoc.org/golang.org/x/net/http2
17. NGINX Inc: nginx stable version 1.10.0, October 2016. https://nginx.org/en/linux_packages.html#stable
18. Yi, X., Yu, S.-Z.: Monitoring the application-layer DDoS attacks for popular websites. IEEE/ACM Trans. Netw. (TON) **17**(1), 15–25 (2009)

19. Ranjan, S., Swaminathan, R., Uysal, M., Nucci, A., Knightly, E.: DDoS-shield: DDoS-resilient scheduling to counter application layer attacks. IEEE/ACM Trans. Netw. (TON) **17**, 26–39 (2009)
20. Maci-Fernndez, G., Daz-Verdejo, J.E., Garca-Teodoro, P.: Mathematical model for low-rate DoS attacks against application servers. IEEE Trans. Inf. Forensics Secur. (TIFS) **4**, 519–529 (2009)
21. Durcekova, V., Schwartz, L.: Sophisticated denial of service attacks aimed at application layer. In: IELEKTRO, Nahid Shahmehri (2012)
22. Zargar, S.T., Joshi, J., Tipper, D.: A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. IEEE Commun. Surv. Tutor. **15**, 2046–2069 (2013)
23. Jazi, H.H., Gonzalez, H., Stakhanova, N., Ali, A.: Detecting HTTP-based application layer DoS attacks on Web servers in the presence of sampling. Comput. Netw. **121**, 25–36 (2017)
24. Imperva: 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
25. Adi, E., Baig, Z.A., Hingston, P., Lam, C.-P.: Distributed denial-of-service attacks against HTTP/2 services. Clust. Comput. **19**, 79–86 (2016)
26. Redelmeier, I.: The Security Implications of HTTP/2.0 (2013). http://www.cs.tufts.edu/comp/116/archive/fall2013/iredelmeier.pdf
27. Larsen, S., Villamil, J.: Attacking HTTP2 implementations. In: 13th PACific SECurity - Applied Security Conferences and Training in Pacific Asia (PacSec) (2015)
28. Van Goethem, T., Vanhoef, M.: HEIST: HTTP encrypted information can be Stolen through TCP-windows, Blackhat, USA (2016)
29. (Kate) Pearce, C., Vincent, C.: HTTP/2 & QUIC - teaching good protocols to do bad things, Blackhat, USA (2016)