IMPERVA®

HACKER INTELLIGENCE INITIATIVE

HTTP/2: In-depth analysis of the top four flaws of the next generation web protocol

Executive Summary

New versions of a protocol such as HTTP/2 are touted as game changers with the goal of addressing the major shortcomings of the existing versions. Unfortunately, everything new and shiny needs to be hardened and time-tested before a full-scale rollout. The web application security research team at the Imperva Defense Center took a hard look at the server side implementations of the HTTP/2 protocol and found four high-profile vulnerabilities.

In this study, we found an exploitable vulnerability in almost all of the new components of the HTTP/2 protocol. The four different attack vectors we discovered are Slow Read, HPACK (Compression), Dependency DoS and Stream abuse. The five popular servers under test from various vendors were found to be vulnerable to at least one attack vector, with Slow Read being the most prevalent. It would be fair to conclude that other implementations of the HTTP/2 protocol may suffer from these vulnerabilities too. The Imperva Defense Center research team worked with the vendors to make sure the vulnerabilities were fixed before publishing this report.

It takes a village to raise a child. And it pays to allow new technology to mature before planning for a significant change of infrastructure. Applying the same concept to new protocols, vendors alone cannot make a new protocol secure, it takes the full strength of the security industry to harden the extended attack surface. Traditionally, the research team approaches most problems with the mindset of a builder, but this is one case where the mindset of a hacker was required to mitigate future threats.

Introduction and Research Motivation

The HTTP protocol is one of the main building blocks of the World Wide Web. It is a simple request-response protocol, designed in 1965 for the client-server model and roughly still holds up today. Over the years, the Internet has been changed significantly while the communication protocol stayed the same except for some minor changes. Asynchronous mechanisms like Iframe tags and AJAX were introduced to accommodate the change of Internet content delivery from the transformation of simple hypertext pages into web pages that have dozens to hundreds of resources. Moreover, these mechanisms are used heavily to decrease the loading time of web pages and make client interaction with the application more fluid.

The HTTP/2 protocol was designed to be the next-generation protocol for web applications. While keeping some of the semantics of older HTTP/1.x, the new protocol displays a complete technical makeover and introduces new significant mechanisms. As in many other cases, this dramatic change and increased complexity are bound to come at a price–that of extending the attack surface and introducing new vulnerabilities into servers and clients.

The primary mandate for the web application security research group, a division of the Imperva Defense Center, is to understand the risks and threats to web technologies as they evolves. The web application security group, staying faithful to its security mandate, initiated a concerted effort to do the initial mapping of HTTP/2 server-side vulnerabilities. The best way to improve the security of standards and their corresponding implementations is through meticulous scrutiny–understanding the attack surface, identifying potential implementation pitfalls, looking for these flaws in implementations and putting a spotlight on the places where it gets translated into vulnerabilities. We set out to understand the risks our customers may be exposed to when adopting the new protocol. We also wanted to guarantee that these risks are responsibly communicated to vendors of HTTP/2 implementations and get adequately mitigated by both the vendors and the Imperva SecureSphere Web Application Firewall (WAF).

The vendors were notified of all the vulnerabilities described in this document before publishing. We coordinated a responsible disclosure process with Microsoft, Apache, Nginx, Jetty and nghttp communities to prevent these vulnerabilities from being exploited after the publication of this report. The mitigation of the vulnerabilities was through security fixes done in coordination with the vendors.

HTTP/2 Protocol

Highlights

The HTTP/2 protocol can be logically divided into three layers: The transmission layer, including streams, frames and flow control; HPACK, binary encoding and compression protocol; and the semantic layer, which is an enhanced version of HTTP/1.1 enriched with server Push capabilities.

The primary motivation for the transition into binary encoding and HPACK compression is to reduce bandwidth, while the other components are designed to reduce roundtrips and accelerate the loading time of complex web pages. Thus, HTTP/2 is expected to significantly improve the loading time and the overall browsing experience of web users while sometimes putting a heavier computational burden on servers.

The main new components in this protocol are:

- Multiplexing–multiple streams can be concurrently carried over a single TCP connection
- Compression–HTTP headers are compressed using a combination of compression schemes (static Huffman coding and context adaptive coding)
- Flow control and dependency–mechanisms that allow HTTP/2 clients and servers to signal how to transmit objects
- Resource push–mechanism that facilitates pushing resources from HTTP/2 servers to their clients

As in HTTP/1.1, the HTTP/2 is built on top of the TCP/TLS layer. The main building blocks of HTTP/2 transmission are as follows:

- **HTTP/2 Frames:** The basic HTTP/2 data unit, replacing HTTP/1.1 header and body format. As opposed to HTTP/1.1 ASCII encoding, HTTP/2 frames have a binary encoding. Frames are divided into Header frames, Data frames and Setting frames.
- **HTTP/2 Streams:** HTTP/2 replacement of HTTP/1.1 Request-Response Protocol. HTTP/2 Stream is a bidirectional channel on which frames are transmitted. A stream includes a single request and a single response.
- **TCP Connection**: HTTP/2, to reduce the number of TCP connections used by the server, allows a single TCP connection to carry multiple streams. See Figure 1: Frames in a connection

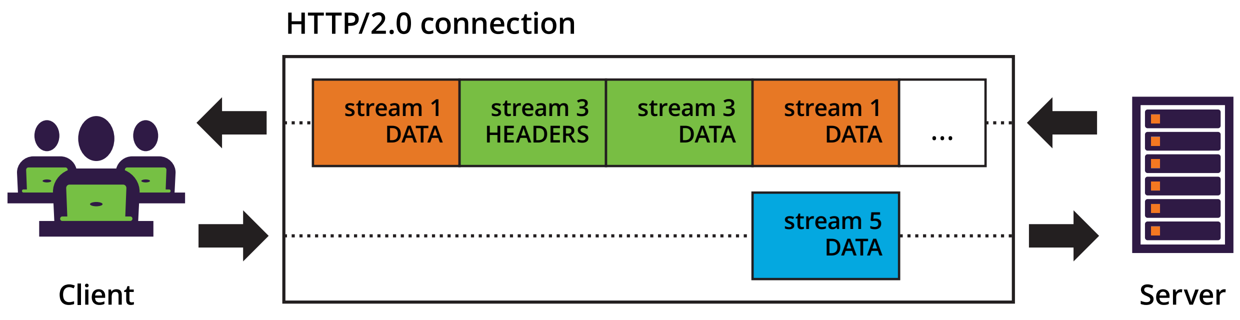
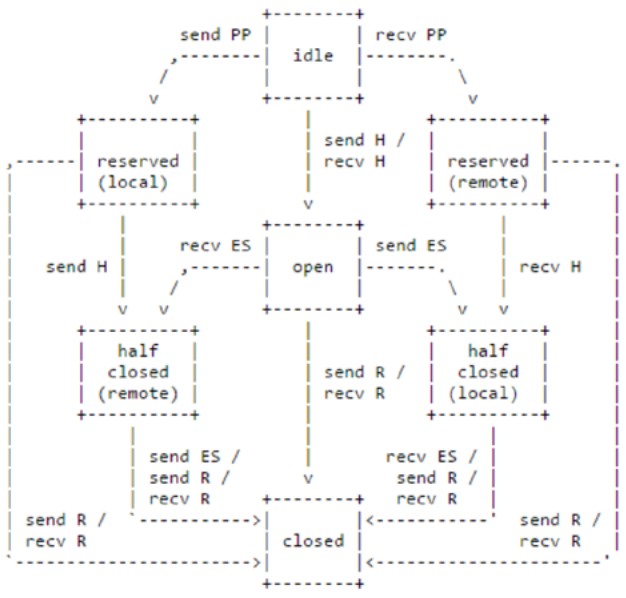


Figure 1: Frames in a connection

Streams and Flow Control

Several requests can use the same TCP connection in parallel with each request having a different stream. In a typical HTTP/2 scenario the client opens a new stream, sends a request over it in header frames and data frames, and waits for the server to respond. The server sends the response back on the same stream and closes it. When the server finishes, the stream (identified by a stream identifier) cannot be used again. Clients are only allowed to open streams with odd numbered identifiers, while servers open the even numbered identifiers. See Figure 2: Stream states.



```
                             +--------+
                     send PP |        | recv PP
                    ,--------|  idle  |--------.
                   /         |        |         \
                  v          +--------+          v
           +----------+          |           +----------+
           |          |          | send H /  |          |
    ,------| reserved |          | recv H    | reserved |------.
    |      | (local)  |          |           | (remote) |      |
    |      +----------+          v           +----------+      |
    |          |             +--------+             |          |
    |          |     recv ES |        | send ES     |          |
    |   send H |     ,-------|  open  |-------.     | recv H   |
    |          |    /        |        |        \    |          |
    |          v   v         +--------+         v   v          |
    |      +----------+          |           +----------+      |
    |      |   half   |          |           |   half   |      |
    |      |  closed  |          | send R /  |  closed  |      |
    |      | (remote) |          | recv R    | (local)  |      |
    |      +----------+          |           +----------+      |
    |           |                |                 |           |
    |           | send ES /      |       recv ES / |           |
    |           | send R /       v        send R / |           |
    |           | recv R     +--------+   recv R   |           |
    | send R /  `----------->|        |<-----------'  send R / |
    | recv R                 | closed |               recv R   |
    `----------------------->|        |<----------------------'
                             +--------+
```

Figure 2: Stream states
Source: rfc7540

HTTP/2 flow control, applicable to streams, is based on the following mechanisms:

- **Stream concurrency:** Carrying multiple streams over a single TCP connection
- **Flow Control:** receiver signals the sender for the maximal amount of data it is allowed to transmit (over a stream/TCP connection)
- **Stream priority:** Sender signals the receiver for the priority of this stream, compared to others
- **Stream dependency:** Sender signals the receiver on recommended order for processing the streams

The flow control mechanism has many parameters that can be used by the endpoints to configure the connection (using Setting frames) including SETTINGS_MAX_CONCURRENT_STREAMS to set the maximal number of concurrent streams over a TCP connection, WINDOW_UPDATE to indicate the maximal amount of data a receiver allows the sender to transmit, and much more. Other parameters like a stream priority (in Priority frame) and stream dependency in other streams (within header frame) can be sent by the sender when initiating a new stream.

The rich collection of novel flow control mechanisms gives significant power to the sender to affect the transmission protocol, which can be used by the typical sender to optimize the server side reception process. However, in the hand of a malicious sender this power to influence the processing at the receiver can be translated into intensive resource consumption and for some server implementations also to a Denial of Service (DoS) attack.

The priority and dependency mechanisms allow each peer to express how it can signal the other peer to allocate resources to streams by building a weighted dependency tree of streams. The HTTP/2 RFC gives much freedom in interpretation to the endpoints, in regards to dependency mechanisms, and they can altogether ignore these signals.

As opposed to that, the window-size mechanism, which allows the endpoints to signal how many bytes they are willing to get in a stream or connection, is mandatory. The window size can be updated as the peer progresses in data parsing.

HPACK

HTTP/2 header compression relies on the similarity of request headers coming from a single client to reduce the bandwidth consumption of header transmission, which has increased dramatically over the years with the HTTP textual encoding. To address security issues, mostly leakage of private data (as was done in CRIME, TIME and BREACH attacks), the HPACK protocol gives away some level of compression for the sake of preventing these attacks. The HPACK protocol relies on a static encoding table and dynamic encoding table. As opposed to the fixed static table, predetermined for the protocol, the dynamic table is used as a cache for each connection direction separately. The sender sets the size of the dynamic table, but shall not exceed a maximal size announced by the receiver. See Figure 4: HPACK example.

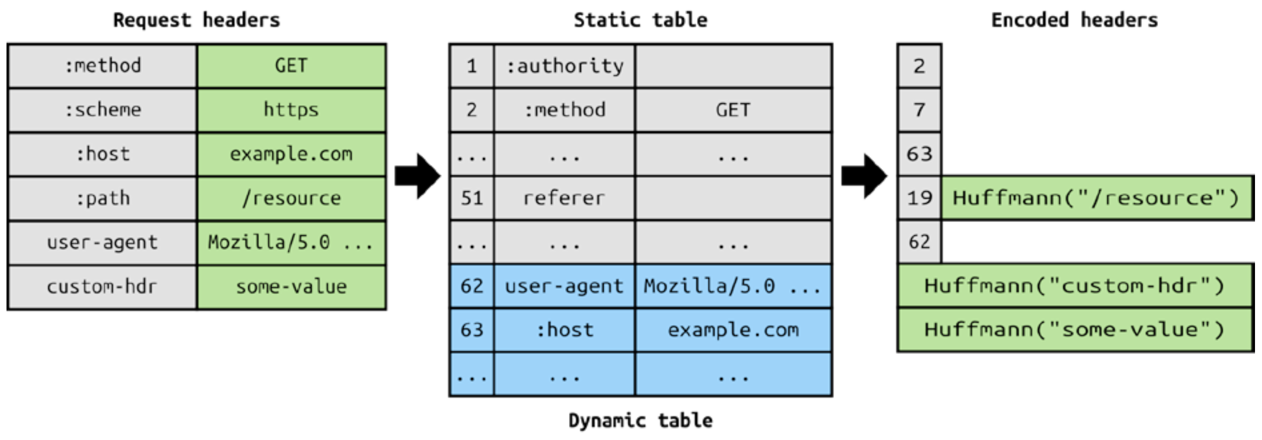


Figure 3: HPACK example
Source: Wikimedia.org

When transmitting a header name or value, the sender chooses whether to send it explicitly (ASCII encoding or the static Huffman encoding specified in the RFC) or to refer to the location of the value in the dynamic table, when this value appears in the table. The sender controls the population of the dynamic table, which signals to the receiver what values to insert to the table.

Server Push

The HTTP/2 Server Push mechanism allows the server to send resources proactively without waiting for a request, when it believes the client will need them, as shown in Figure 4: Server Push.. For example, in a typical web page delivery scenario, the server will send the requested page to the client and immediately push all the page resources to the client (each in a server-initiated stream), including CSS, javascript, images and HTML components. This saves the round-trip time and reduces the page-loading time.

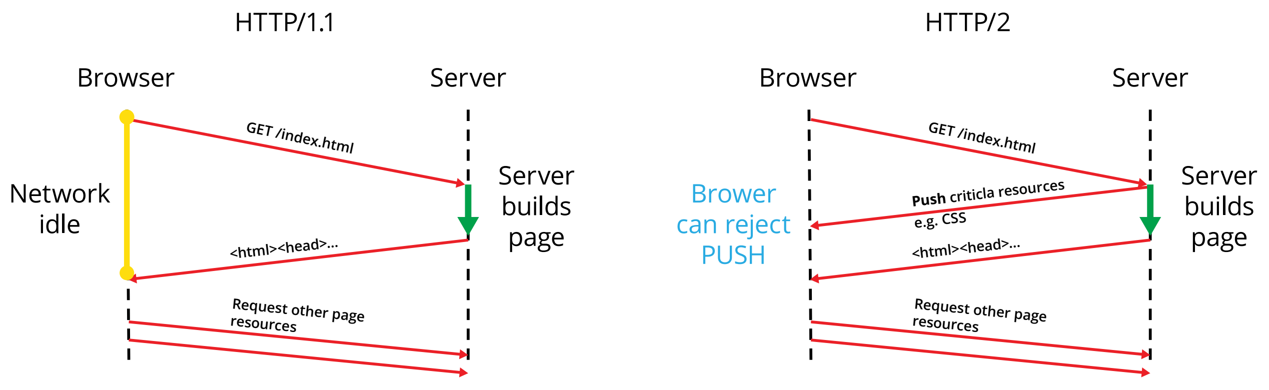
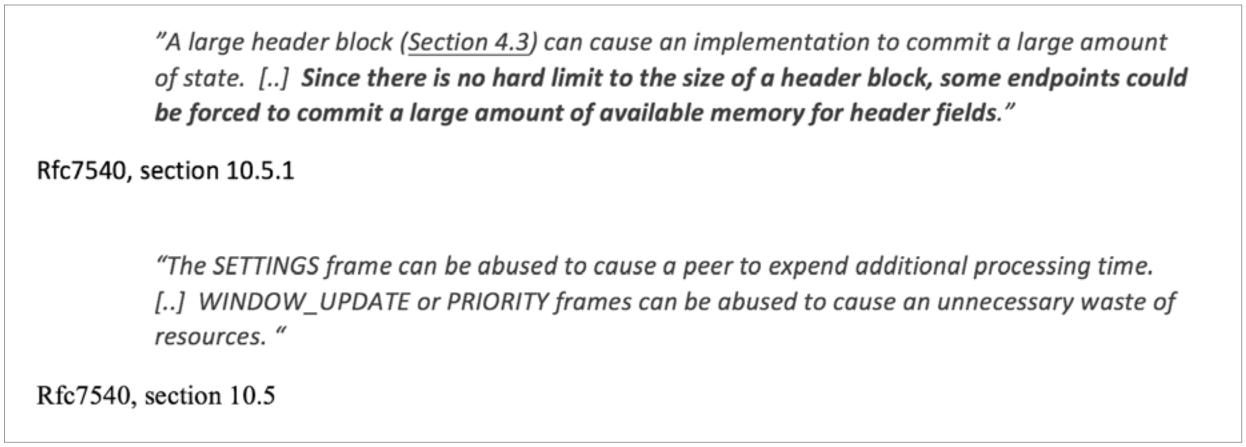


Figure 4: Server Push

HTTP/2 Security Considerations

The designers of HTTP/2 made significant effort to identify and address security risks involved in the new protocol through design choices, e.g., preferring HPACK on a better compression algorithm, or implementation guidelines, e.g., suggesting clients to avoid compressing sensitive data like session cookies.

However, as we have found in this research, implementations of HTTP/2 servers do not always follow these guidelines. In at least two cases we found HTTP/2 implementations that specifically failed to account for the typical traps designers warn about. See Flow Control DoS, Dependency Cycle. One example is an attack based on abuse of the flow control WINDOW_UPDATE for DoS attacks, which was specifically warned about as referenced in Figure 5: RFC Security Warnings. Another example is the risk of processing large header blocks in HPACK implementation, which can be used for causing intensive memory consumption on the server. See HPACK bomb.



*"A large header block (Section 4.3) can cause an implementation to commit a large amount of state. [..] **Since there is no hard limit to the size of a header block, some endpoints could be forced to commit a large amount of available memory for header fields."***

Rfc7540, section 10.5.1

"The SETTINGS frame can be abused to cause a peer to expend additional processing time. [..] WINDOW_UPDATE or PRIORITY frames can be abused to cause an unnecessary waste of resources. "

Rfc7540, section 10.5

Figure 5: RFC Security Warnings

HTTP/2 Deployment

The adoption of HTTP/2 protocol in the field is moving very fast. All of the major browsers support the HTTP/2 protocol as do all of the major application servers, including IIS, Apache, and NGINX. Also, Content Delivery Networks (CDNs) like Imperva Incapsula, CloudFlare, and Akamai; and Load Balancers like F5 Big-IP, all support HTTP/2 termination.

According to W3Techs, 6.5% of all the websites (~64 million sites) and 13.5% of the top 1000 sites, including Mega-websites like Twitter, Facebook, and Google, are using HTTP/2. It is interesting to note that the adoption rate for the top million sites is 18.5%, significantly higher. We attribute this phenomenon to the increasing reliance of websites on CDNs such as Incapsula, Cloud Flare, and Akamai, which show HTTP/2 support to clients even when the web server itself does not support it. While this can improve user experience, it could be dangerous if not done properly, as the intermediaries can expose the server to new vulnerabilities.

HTTP/2 Attacks

Migrating the Internet to HTTP/2 involves all the Internet players, clients, servers, and infrastructures to adjust or even rewrite a huge amount of code, in addition to the HTTP/2 mechanisms themselves. Releasing a large amount of new code into the wild in a short time creates an excellent opportunity for attackers.

In the research, we focused on server-side implementation as Apache, IIS, Ngnix, Jetty and nghttp. The first phase of the study was to understand the attack surface–the new mechanisms introduced and analyze their potential risks. The immediate suspect was the influence of HTTP/2 clients on the transmission protocol, and thus on the server's processing when serving HTTP/2 clients. More specifically, we asked ourselves whether specific HTTP/2 settings, when invoked by the client, can make the server crash.

Victim 1–HTTP/2 Stream Multiplexing (CVE-2016-0150)

The Stream Multiplexing mechanism tunnels multiple sessions through a single HTTP/2 connection. The risk in this mechanism stems from the fact that the partition of the connection is purely logical, and as such can be used to manipulate the server or to send frames out of context. See Figure 6: Stream Reuse on IIS.

Attack Type	Vulnerable Server	Affected Versions
Stream Reuse	IIS	10

Figure 6: Stream Reuse on IIS

HTTP/2 stream represents one request-response cycle and once closed, the stream identifier is not supposed to be used over the same connection. See Figure 7: Stream reuse attack diagram. While the RFC document clearly indicates this requirement we decided to test its implementation in practice. We created a script that reuses a closed stream identifier in the same connection. Figure 8: IIS stack back trace, shows a schematic description of this attack. Soon enough we found our first vulnerable implementation. Due to a flaw in the HTTP.sys driver in IIS 10, when the server receives two requests on the same stream, the result is the blue screen of death as shown in Figure 9: HTTP.sys BSOD. We did not explore the crash further, but it is important to note that such behavior hints on memory error, possibly one process stepping on the memory of another process, which under certain circumstances can lead to arbitrary code execution.

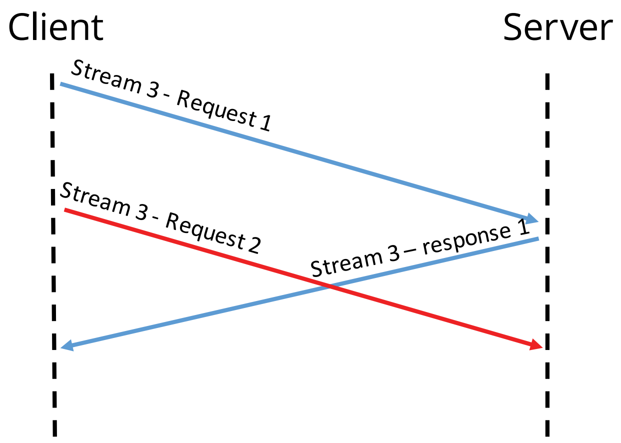
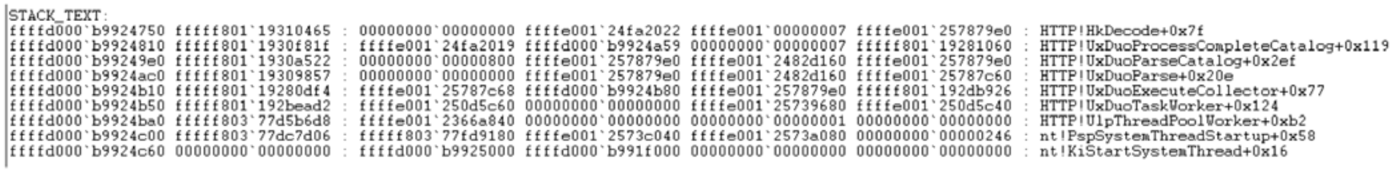
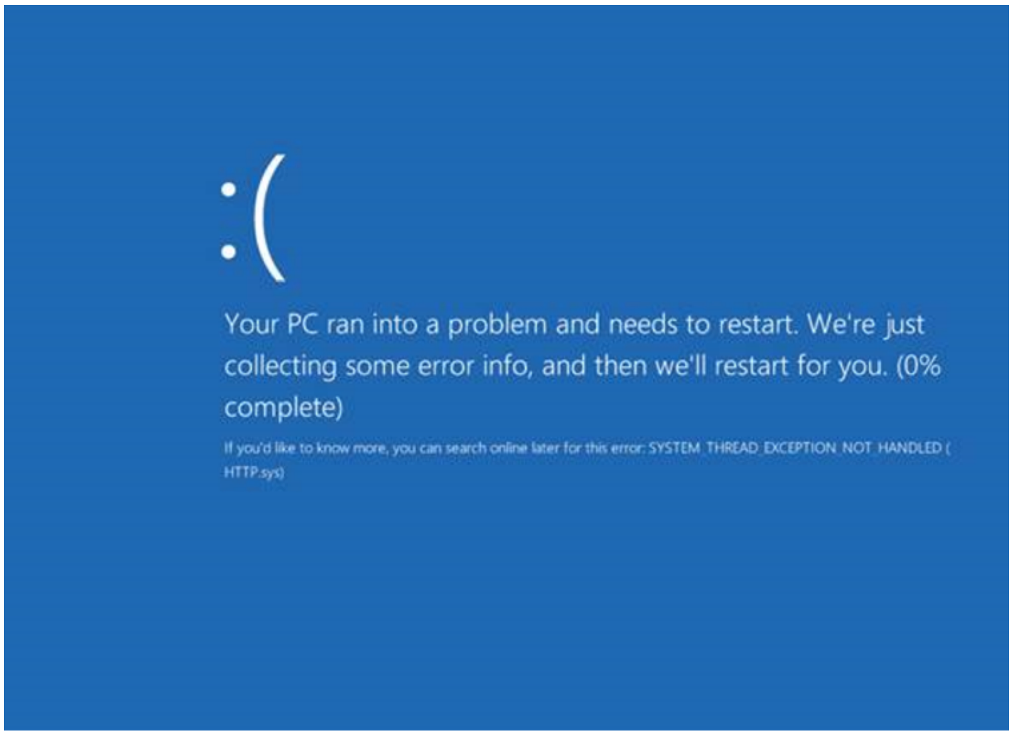


Figure 7: Stream reuse attack diagram



```
STACK_TEXT:
ffffd000`b9924750 fffff801`19310465 : 00000000`00000000 ffffe001`24fa2022 ffffe001`00000007 ffffe001`257879e0 : HTTP!HkDecode+0x7f
ffffd000`b9924810 fffff801`1930f81f : ffffe001`24fa2019 ffffd000`b9924a59 00000000`00000007 fffff801`19281060 : HTTP!UxDuoProcessCompleteCatalog+0x119
ffffd000`b99249e0 fffff801`1930a522 : 00000000`00000800 ffffe001`257879e0 ffffe001`2482d160 ffffe001`257879e0 : HTTP!UxDuoParseCatalog+0x2ef
ffffd000`b9924ac0 fffff801`19309857 : 00000000`00000000 ffffe001`257879e0 ffffe001`2482d160 ffffe001`25787c60 : HTTP!UxDuoParse+0x20e
ffffd000`b9924b10 fffff801`19280df4 : ffffe001`25787c68 ffffd000`b9924b80 ffffe001`257879e0 fffff801`192db926 : HTTP!UxDuoExecuteCollector+0x77
ffffd000`b9924b50 fffff801`192bead2 : ffffe001`250d5c60 00000000`00000000 ffffe001`25739680 ffffe001`250d5c40 : HTTP!UxDuoTaskWorker+0x124
ffffd000`b9924ba0 fffff803`77d5b6d8 : ffffe001`2366a840 00000000`00000000 00000000`00000001 00000000`00000000 : HTTP!UlpThreadPoolWorker+0xb2
ffffd000`b9924c00 fffff803`77dc7d06 : fffff803`77fd9180 ffffe001`2573c040 ffffe001`2573a080 00000000`00000246 : nt!PspSystemThreadStartup+0x58
ffffd000`b9924c60 00000000`00000000 : ffffd000`b9925000 ffffd000`b991f000 00000000`00000000 00000000`00000000 : nt!KiStartSystemThread+0x16
```

Figure 8: IIS stack back trace



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

If you'd like to know more, you can search online later for this error: SYSTEM_THREAD_EXCEPTION_NOT_HANDLED (HTTP.sys)

Figure 9: HTTP.sys BSOD

Victim 2 – HTTP/2 Flow Control

Attack Type	Vulnerable Server	Affected Versions
Slow POST	Jetty	9.3
Slow GET	Apache	2.4.18
Slow GET	IIS	10
Slow GET	Nginx	1.9.9

Figure 10: Slow Read Attacks on Various Servers

Since the RFC explicitly marked the flow control feature as a security hazard, it was one of the first candidates we chose to examine. The HTTP/2 window mechanism resembles the TCP window mechanisms that were the target to many attacks in the in the past, including zero-window and slow read attacks. When reducing or even resetting the inbound window size while asking for a large resource, the sender may keep the connection open for a long or even unlimited period, and consumes the server resources. Thus, it was reasonable to believe that the new implementation of this mechanism in HTTP/2 would suffer the same problems.

Slow Read Attack (CVE-2016-1546)

See Figure 10: Slow Read Attacks on Various Servers. When aimed at vulnerable server, Slow Read attacks can consume all of the worker threads in the server and result in DoS and crash the server. While the attacker, in the original setting had to open as many TCP connections as the victim server, in the HTTP/2 setting the attacks becomes simpler, since the attacker can use the stream multiplexing capabilities to multiplex a large number of streams over a single TCP connection. Although the server maintains a single TCP connection, it dedicates a thread per stream and thus the result of the attack is consumption of all the worker threads of the victim server. The fact that the asymmetry between the client and server resources) is amplified, makes the attack much easier to mount and target at powerful web servers, even with minimal computational resources.

Our malicious client, to implement Slow Read, had asked for a large resource from the server, and set the WINDOW_UPDATE parameter to a very small size, telling the server that the maximal amount of data it is permitted to transmit on the stream is very small. We updated the window size with small increments, each time allowing few more bytes to come in, and expected the server, trying to provide this large resource to the malicious client, will allocate resources for processing of this stream and will not free them until the entire data is transmitted–practically never. We repeated this process with multiple streams and expected that in places where our assumptions held, at some point the server would be unable to service other clients.

While the basic idea of Slow Read attacks is straightforward and has been studied for years, we found out that most servers we checked are vulnerable. The behavior of servers in Slow Read situations depends on the type and structure of the requests sent, but after examining different types and structures of requests, we were able to mount at least one practical attack scenario for most of the servers examined.

Slow Read on Jetty

When receiving a POST request to a resource larger than the remaining window size, Jetty servers send the thread that handles the request to sleep for 30 seconds. When repeating the process every 30 seconds, the attacker can keep a single thread sleeping for as much time as it wants. In parallel, the attacker repeats the process with other streams, causing the server to allocate all its resources to sleeping threads and to stop serving clients. See Figure 11: Jetty Slow POST Attack and Section Jetty Slow Read Attack (POST) in Appendix.

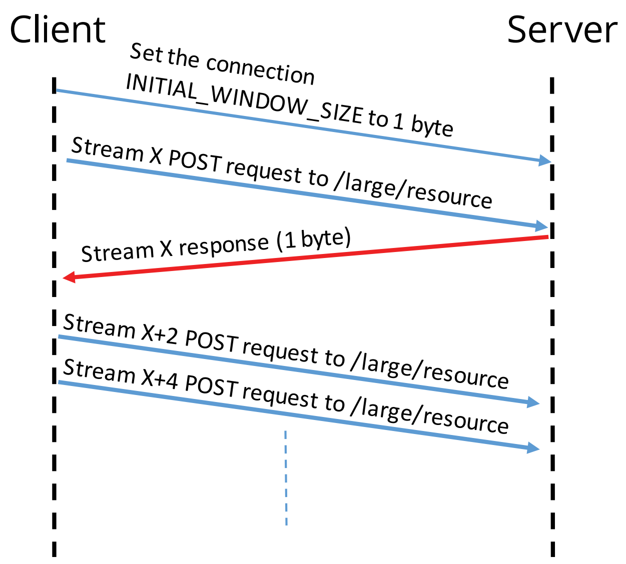

Client
Server
Set the connection INITIAL_WINDOW_SIZE to 1 byte
Stream X POST request to /large/resource
Stream X response (1 byte)
Stream X+2 POST request to /large/resource
Stream X+4 POST request to /large/resource

Figure 11: Jetty Slow POST Attack

Slow Read on Apache

For Apache servers, the effective request method was GET, again directed to a resource larger than the remaining window size. Also in this case, the server thread that handles the request goes to sleep when not having sufficient window size to send the rest of the response. To prevent the Apache server from killing the thread, the attacker can send further WINDOW_UPDATE frames as described in figure 14. As long as the attacker sends WINDOW_UPDATE frames, the thread is kept alive, and when replicating this scenario, we were able to occupy all the threads of the server easily, and the server stopped responding to other clients.
Figure 12: Apache/IIS Slow GET Attack and see Apache Slow Read Attack (GET) in the Appendix.

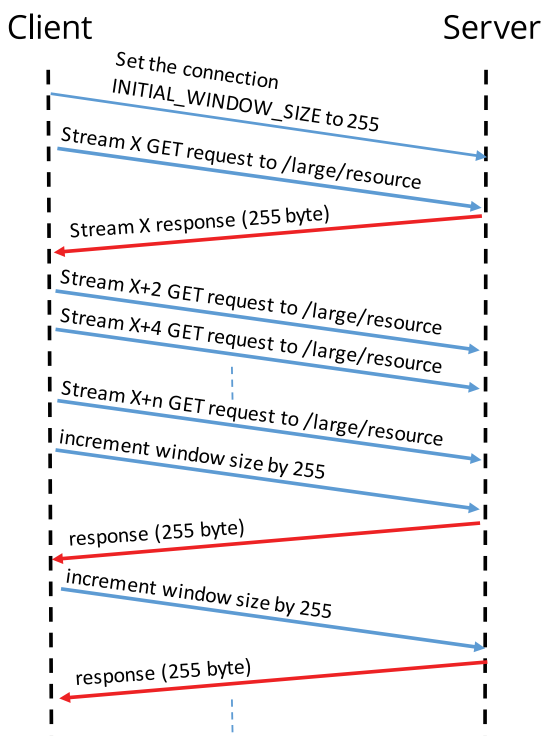

Client
Server
Set the connection INITIAL_WINDOW_SIZE to 255
Stream X GET request to /large/resource
Stream X response (255 byte)
Stream X+2 GET request to /large/resource
Stream X+4 GET request to /large/resource
Stream X+n GET request to /large/resource
increment window size by 255
response (255 byte)
increment window size by 255
response (255 byte)

Figure 12: Apache/IIS Slow GET Attack

Slow Read on IIS

We found the Apache variant of the Slow Read attack, as described in figure 12, highly effective on IIS servers, requiring less than 20 requests to make the server unavailable for other clients. This attack can be easily achieved using a single connection.

Slow Read on Nginx

In Nginx we used a slightly different attack scenario as the original one did not work, using 20 TCP connections and opening as many streams as MAX_CONCURRENT_STREAM allows with the server. See Figure 13 - Nginx Slow GET Attack. After a relatively small number of connections, the server stopped responding to the attacker requests and other clients accessing the server received the 500 response code (Internal Server Error).

The attack scenario is more intensive than the previous ones, requiring more resources on the attacker side, and having a clear footprint on the server side. However, with reasonable effort, the attack remains practical. Opening and controlling 20 TCP connections can be easily achieved using a single client or from a small number of compromised machines. IP-based rate limit mechanisms can be bypassed in various ways, e.g., routing the attack through the TOR network.

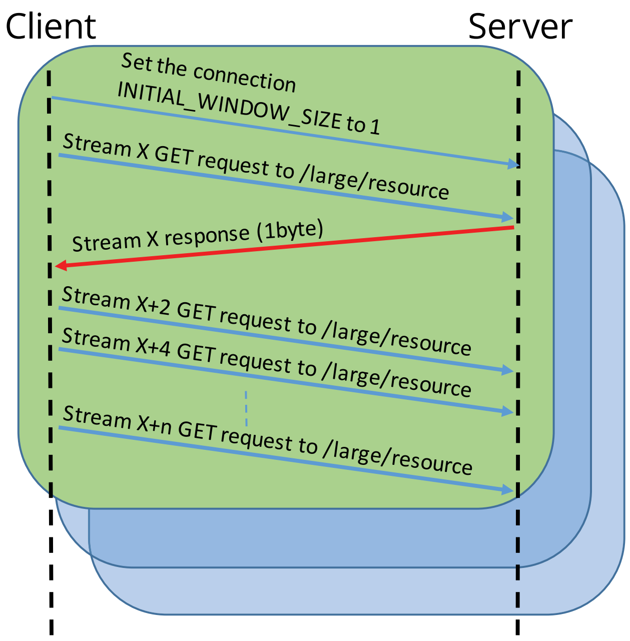

Client
Server
Set the connection INITIAL_WINDOW_SIZE to 1
Stream X GET request to /large/resource
Stream X response (1byte)
Stream X+2 GET request to /large/resource
Stream X+4 GET request to /large/resource
Stream X+n GET request to /large/resource

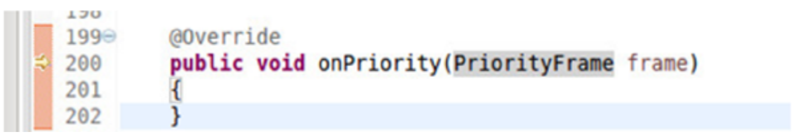
Figure 13: Nginx Slow GET Attack

We reported about the attack to [Nginx](), and it turned out that they are already aware of it.

Victim 3-Dependency and Priority

When looking on the new mechanisms in the transmission layer, also known as the binary layer, the new dependency and priority mechanisms stand out from a security perspective. The fact that the mechanisms are optional and that even when implemented, complying with the RFC recommendation is not mandatory, we suspected that some edge cases will not be handled properly by server implementations.

First, the size of the dependency tree is not limited. Thus, a server that naively trusts the client may be foiled to build a dependency tree that will consume its memory. Second, we suspected client-generated edge case in the dependency tree—such as dependency cycles, or rapid changes in the dependencies, sometimes conflicting with each other—will be ignored by developers and may result with unexpected consequences such as “hopefully” heavy CPU consumption or unreasonable memory consumption. With these potential risks in mind, we could understand why some implementations like Jetty, chose to skip this feature completely in the implementation as shown in Figure 14.



```
    @Override
    public void onPriority(PriorityFrame frame)
    {
    }
```

Figure 14: Jetty (lack of) priority implementation

Dependency Cycle DoS

Attack Type	Vulnerable Server	Affected Versions
Dependency Cycle DoS	nghttpd	Before 1.7.0
Dependency Cycle DoS	Apache	2.4.18

Figure 15: Dependency Cycle DoS

According to the RFC, the dependency graph should be a tree as a cycle in this graph may invoke and unwanted behavior such infinite loop or memory overrun. Additionally, the size of the graph is not limited by the RFC so each server can set its size limitation. Potentially invoking many graph operations may reduce server performance. See Figure 15: Dependency Cycle DoS.

Previous to our research, such vulnerabilities were found in nghttp2 and Apache (whose HTTP/2 implementation is derived from nghttp2). Nghttp2 restricts the dependency graph size to MAX_CONCURRENT_STREAMS. When a client tries to create a larger graph using priority frames, the server throws away old streams. However, due to some flaws in the memory cleanup, an attacker can still cause DoS or may even be able to execute arbitrary code. This vulnerability has been fixed in nghttp2 1.7.0 as part of a more general memory cleanup issue (CVE-2015-8659). However, further exploration showed that other implementations fall into the same trap.

DoS on nghttpd

When looking on the new mechanisms in the transmission layer, also known as the binary layer, the new dependency and priority mechanisms stand out from a security perspective. The fact that the mechanisms are optional and that even when implemented, complying with the RFC recommendation is not mandatory, we suspected that some edge cases will not be handled properly by server implementations.

First, the size of the dependency tree is not limited. Thus, a server that naively trusts the client may be foiled to build a dependency tree that will consume its memory. Second, we suspected client-generated edge case in the dependency tree—such as dependency cycles, or rapid changes in the dependencies, sometimes conflicting with each other—will be ignored by developers and may result with unexpected consequences such as "hopefully" heavy CPU consumption or unreasonable memory consumption. With these potential risks in mind, we could understand why some implementations like Jetty, chose to skip this feature completely in the implementation as shown in Figure 14.

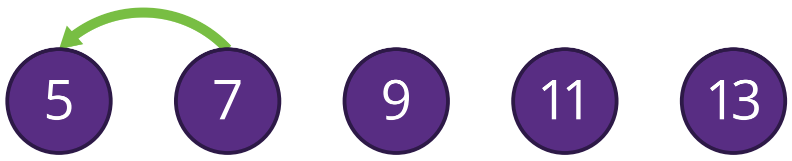


Figure 16: Dependency Cycle for 5 streams—initial state

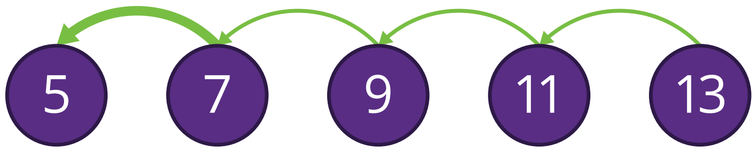


Figure 17: Dependency Cycle for 5 streams–after adding 5 dependencies

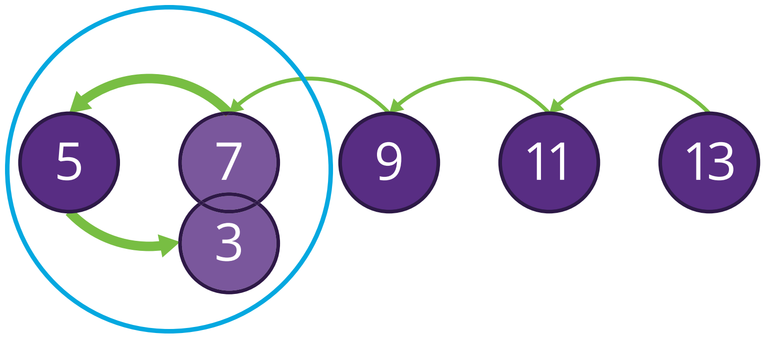
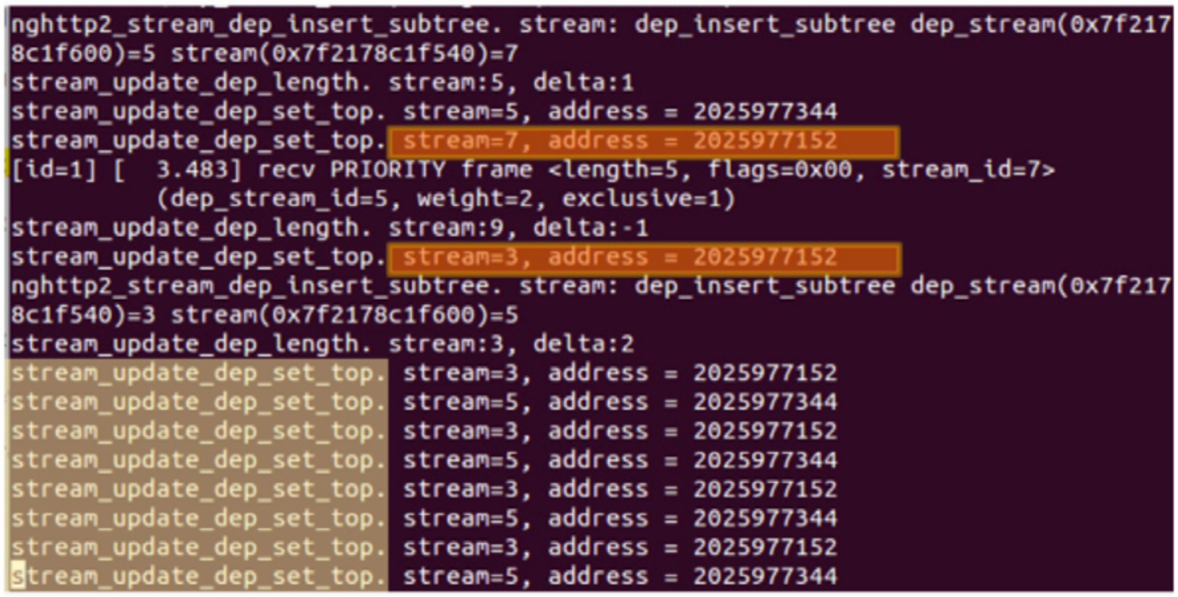


Figure 18: Dependency Cycle for 5 streams–after adding 6 dependencies

Since the number of streams exceeded MAX_CONCURRENT_STREAM, the oldest stream (7) is thrown away, and the newest stream (3) takes its place in the same address. However, due to a flaw in the cleanup process, the dependency pointer to stream 5 remains, resulting in a dependency cycle between streams 3 and 5.

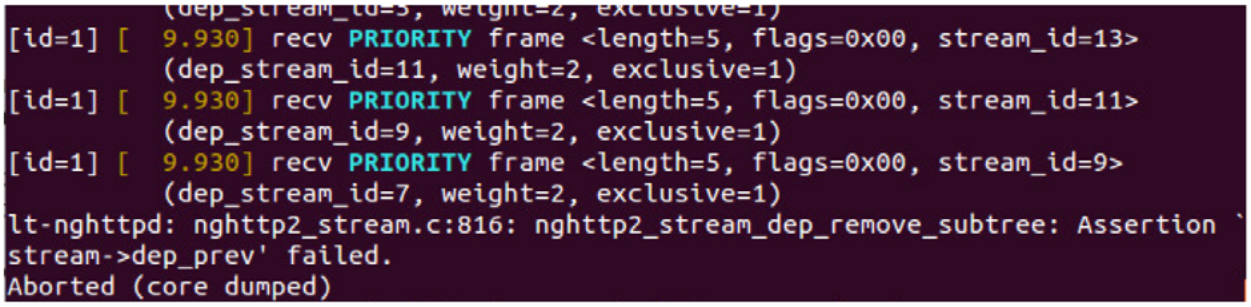


```
nghttp2_stream_dep_insert_subtree. stream: dep_insert_subtree dep_stream(0x7f217
8c1f600)=5 stream(0x7f2178c1f540)=7
stream_update_dep_length. stream:5, delta:1
stream_update_dep_set_top. stream=5, address = 2025977344
stream_update_dep_set_top. stream=7, address = 2025977152
[id=1] [  3.483] recv PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=5, weight=2, exclusive=1)
stream_update_dep_length. stream:9, delta:-1
stream_update_dep_set_top. stream=3, address = 2025977152
nghttp2_stream_dep_insert_subtree. stream: dep_insert_subtree dep_stream(0x7f217
8c1f540)=3 stream(0x7f2178c1f600)=5
stream_update_dep_length. stream:3, delta:2
stream_update_dep_set_top. stream=3, address = 2025977152
stream_update_dep_set_top. stream=5, address = 2025977344
stream_update_dep_set_top. stream=3, address = 2025977152
stream_update_dep_set_top. stream=5, address = 2025977344
stream_update_dep_set_top. stream=3, address = 2025977152
stream_update_dep_set_top. stream=5, address = 2025977344
stream_update_dep_set_top. stream=3, address = 2025977152
stream_update_dep_set_top. stream=5, address = 2025977344
```

Figure 19: nghttp2 dependency cycle–Stream 3 and Stream 7 are in the same memory address, which caused an infinite recursion

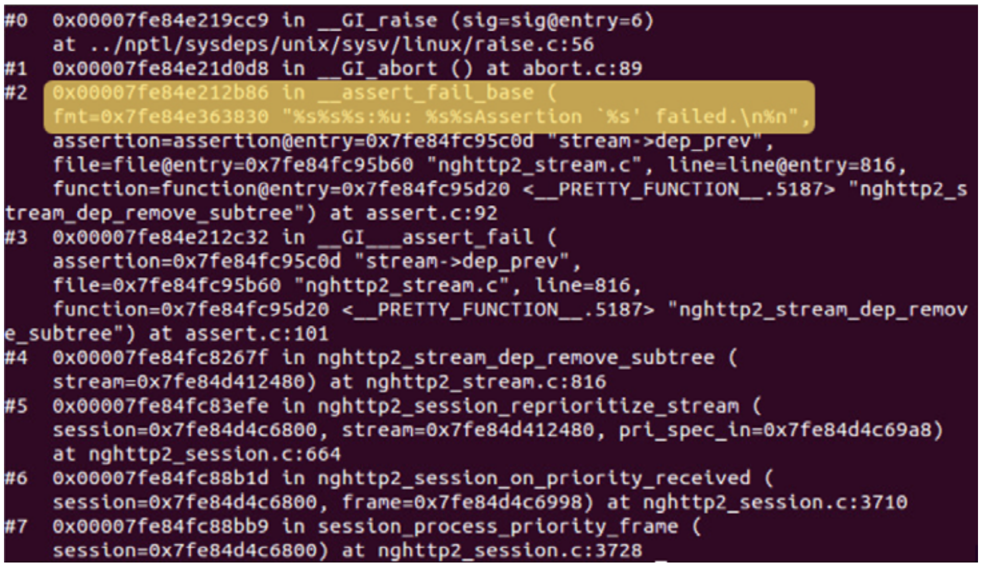
In nghttpd V.1.0.2-DEV, after the attacker creates the cycle in the graph, the server enters an infinite recursion and crashes with a segmentation fault as shown in Figure 19: nghttp2 dependency cycle–where Stream 3 and Stream 7 are in the same memory address, causing an infinite recursion..

In nghttpd V.1.4.0, the code was changed with an assertion statement verifying that there is no cycle in the graph and throw Assertion exception otherwise. However, the assertion exception is improperly handled and causes the server to crash and stop responding to other clients as shown in Figure 20: nghttp2 V1.4.0 assertion, showing the Dependency Cycle Attack on Apache 2.4.18.



```
        (dep_stream_id=5, weight=2, exclusive=1)
[id=1] [  9.930] recv PRIORITY frame <length=5, flags=0x00, stream_id=13>
        (dep_stream_id=11, weight=2, exclusive=1)
[id=1] [  9.930] recv PRIORITY frame <length=5, flags=0x00, stream_id=11>
        (dep_stream_id=9, weight=2, exclusive=1)
[id=1] [  9.930] recv PRIORITY frame <length=5, flags=0x00, stream_id=9>
        (dep_stream_id=7, weight=2, exclusive=1)
lt-nghttpd: nghttp2_stream.c:816: nghttp2_stream_dep_remove_subtree: Assertion `
stream->dep_prev' failed.
Aborted (core dumped)
```

Figure 20: nghttp2 V1.4.0 assertion



```
#0  0x00007fe84e219cc9 in __GI_raise (sig=sig@entry=6)
    at ../nptl/sysdeps/unix/sysv/linux/raise.c:56
#1  0x00007fe84e21d0d8 in __GI_abort () at abort.c:89
#2  0x00007fe84e212b86 in __assert_fail_base (
    fmt=0x7fe84e363830 "%s%s%s:%u: %s%sAssertion `%s' failed.\n%n",
    assertion=assertion@entry=0x7fe84fc95c0d "stream->dep_prev",
    file=file@entry=0x7fe84fc95b60 "nghttp2_stream.c", line=line@entry=816,
    function=function@entry=0x7fe84fc95d20 <__PRETTY_FUNCTION__.5187> "nghttp2_s
tream_dep_remove_subtree") at assert.c:92
#3  0x00007fe84e212c32 in __GI___assert_fail (
    assertion=0x7fe84fc95c0d "stream->dep_prev",
    file=0x7fe84fc95b60 "nghttp2_stream.c", line=816,
    function=0x7fe84fc95d20 <__PRETTY_FUNCTION__.5187> "nghttp2_stream_dep_remov
e_subtree") at assert.c:101
#4  0x00007fe84fc8267f in nghttp2_stream_dep_remove_subtree (
    stream=0x7fe84d412480) at nghttp2_stream.c:816
#5  0x00007fe84fc83efe in nghttp2_session_reprioritize_stream (
    session=0x7fe84d4c6800, stream=0x7fe84d412480, pri_spec_in=0x7fe84d4c69a8)
    at nghttp2_session.c:664
#6  0x00007fe84fc88b1d in nghttp2_session_on_priority_received (
    session=0x7fe84d4c6800, frame=0x7fe84d4c6998) at nghttp2_session.c:3710
#7  0x00007fe84fc88bb9 in session_process_priority_frame (
    session=0x7fe84d4c6800) at nghttp2_session.c:3728
```

Figure 21: nghttpd stack backtrace

Apache is using nghttp2 as its HTTP/2 engine. As a result, this vulnerability can also be exploited on Apache when using a vulnerable version of nghttp2. Unlike nghttp2, the mishandled assertion only affects a single thread of the server. To exploit this vulnerability, an attacker must repeat this attack until all allocated threads are exhausted. See Figure 21: nghttpd stack backtrace.

HPACK (Compression)

From a security perspective, compression mechanisms have two major concerns. The first is the data leak risk (CRIME, TIME, BREACH), applicable when compression precedes an encryption operation. These attacks use the length of crafted requests after gzip compression to leak sensitive encrypted information such as cookies, user-agent, and other headers. The other concern is the risk of a crafted zipped message that can cause unexpected behavior in the decoder as was done in zip bomb attacks. The idea behind this attack is simple–the attacker sends the relatively small amount of data, which will catch a large amount of memory after decompression.

From the RFC it is clear that immunity against data leakage attacks was one of the design criteria for HPACK. Thus, we chose to focus on the second type, DoS attacks, by crafting decoding mines. See Figure 22: HPACK Bomb Attacks

HPACK Bomb (CVE-2016-1544), (CVE-2016-2525)

Attack Type	Vulnerable Server	Affected Versions
HPACK Bomb	nghttpd	Before 1.7.1
HPACK Bomb	Wireshark	Before 2.0.2 and 1.12.10

Figure 22: HPACK Bomb Attacks

HPACK Bomb Attack on nghttpd (before 1.7.1)

We have found a vulnerability in the compression layer of HTTP/2. The RFC defines a mechanism that allows each peer to restrict the size of the dynamic table (SETTINGS_HEADER_TABLE_SIZE):

> *"Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block (see [COMPRESSION]). The initial value is 4,096 octets."*

However, the RFC does not provide any further restriction on the size of individual headers. Hence, the size of an individual header is only restricted by the scale of the dynamic table. Figure 23 illustrates an attack in which we chose to generate a first stream with one large header–as big as the entire table. Then we repeatedly open new streams on the same connection that reference this single large header as many times as possible. The server keeps allocating new memory to decompress the requests and eventually consumes all memory–denying further access by other clients.

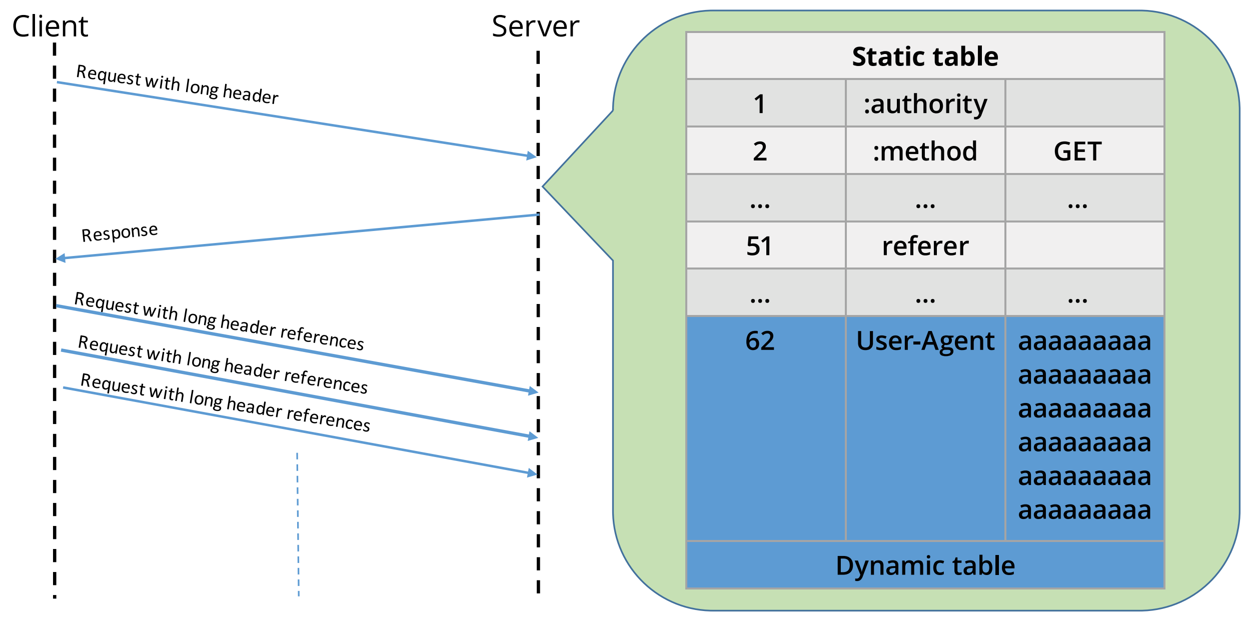
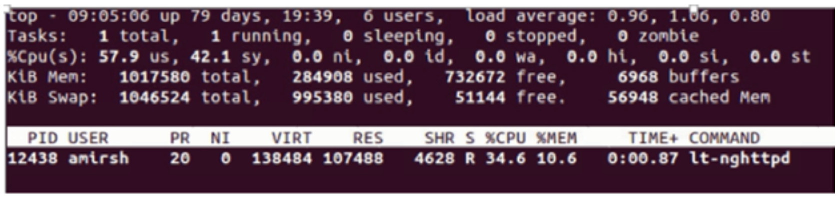
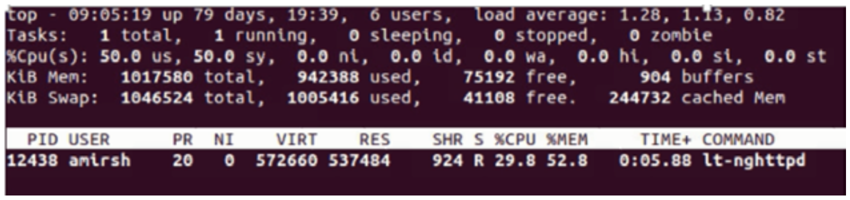

Client
Server
Request with long header
Response
Request with long header references
Request with long header references
Request with long header references
Static table
1 :authority
2 :method GET
... ...
51 referer
... ...
62 User-Agent aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa aaaaaaaaa
Dynamic table

Figure 23: HPACK Bomb



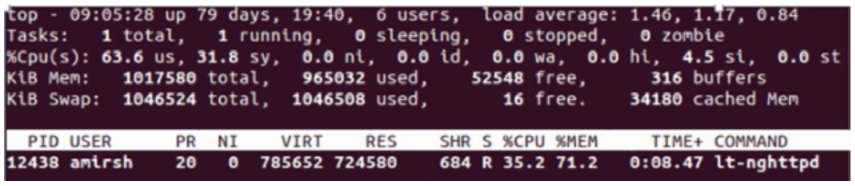
```
top - 09:05:06 up 79 days, 19:39,  6 users,  load average: 0.96, 1.06, 0.80
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s): 57.9 us, 42.1 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1017580 total,   284908 used,   732672 free,     6968 buffers
KiB Swap:  1046524 total,   995380 used,    51144 free.    56948 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
12438 amirsh    20   0  138484 107488   4628 R 34.6 10.6   0:00.87 lt-nghttpd
```

Figure 24: nghttpd memory consumption after decompressing 32K headers



```
top - 09:05:19 up 79 days, 19:39,  6 users,  load average: 1.28, 1.13, 0.82
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.0 us, 50.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1017580 total,   942388 used,    75192 free,      904 buffers
KiB Swap:  1046524 total,  1005416 used,    41108 free.   244732 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
12438 amirsh    20   0  572660 537484    924 R 29.8 52.8   0:05.88 lt-nghttpd
```

Figure 25: nghttpd memory consumption after decompressing 128K headers



```
top - 09:05:28 up 79 days, 19:40,  6 users,  load average: 1.46, 1.17, 0.84
Tasks:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s): 63.6 us, 31.8 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  4.5 si,  0.0 st
KiB Mem:   1017580 total,   965032 used,    52548 free,      316 buffers
KiB Swap:  1046524 total,  1046508 used,       16 free.    34180 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
12438 amirsh    20   0  785652 724580    684 R 35.2 71.2   0:08.47 lt-nghttpd
```

Figure 26: nghttpd memory consumption after decompressing 224K headers

The default size of the dynamic table is 4KB. The server allows one request to contain up to 16K of header references. By sending a single header of size 4KB and then sending a request with 16K references to this one header, the request is decompressed to 64MB on the server side. As we open more streams on the same connection, we quickly consume more and more memory as shown in Figures 24, 25, and 26. In our lab, 14 streams that consumed 896MB after decompression, were enough to crash the server.

Crashing Wireshark

The nghttpd server is a based on the nghttp2 library. The library itself counts on the application above it to manage its memory correctly. Since nghttp2 is widely used as a library implementation of HTTP/2, this vulnerability can affect other applications that are using it without paying attention to this issue. One example is Wireshark before versions 2.0.2 and 1.12.10: when trying to open a network capture file taken from this attack (pcap file), Wireshark uses all the available CPU and a large amount of memory and then hangs. Figure 32 was taken from a machine with a quad-core processor, therefore; 25 percent of CPU is a full resource consumption of one core. See Figures 27 and 28: Wireshark performance graph. (CVE-2016-2525)

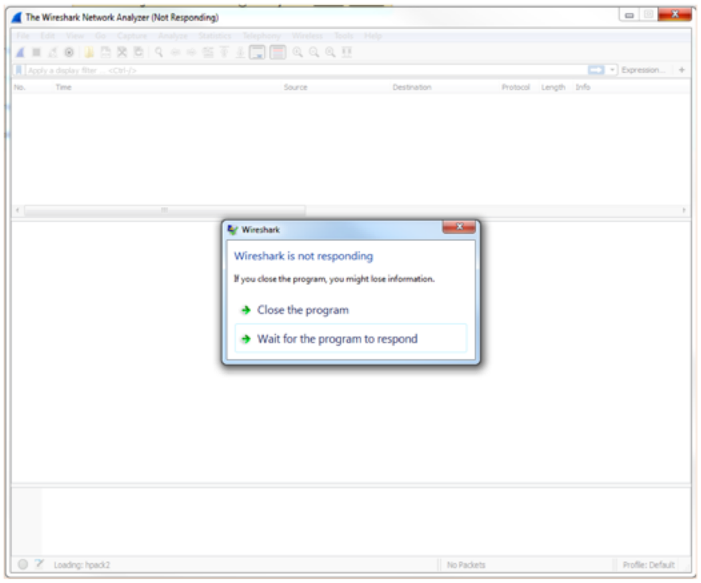

The Wireshark Network Analyzer (Not Responding)
Wireshark
Wireshark is not responding
If you close the program, you might lose information.
Close the program
Wait for the program to respond
Loading: hpack2
No Packets
Profile: Default

Figure 27: Wireshark is not responding when loading the pcap file

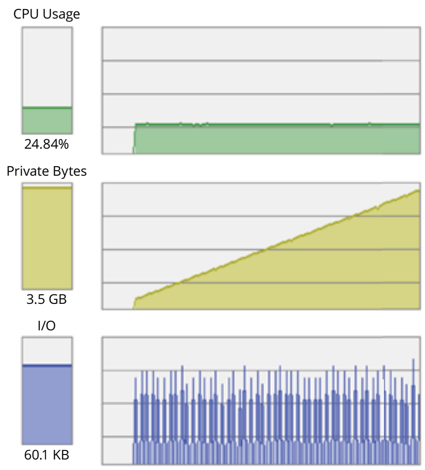


Figure 28: Wireshark performance graph, taken with Process Explorer

Summary and Conclusions

In this research, we have found four different attack vectors. We were able to find an exploitable vulnerability in almost all of the new components of the HTTP/2 protocol. We tested five popular servers and found all to be vulnerable to at least one attack. Thus, we believe other implementations of the protocol may suffer from these vulnerabilities especially those that rely on external HTTP/2 libraries such as nghttp2 (see collateral damage section).

Server	Slow Read	HPACK	Dependency	Stream Abuse
Apache	V		V	
IIS	V			V
Jetty	V			
Nghttpd		V	V	
Nginx	V			

Figure 29: Summary of vulnerabilities, shows the vulnerabilities that we found per each server implementation.

Conclusions

This research is pointing out once again that new technology brings new risks. When releasing new code into the wild, it is only a matter of time until new vulnerabilities are found and exploited. As with any new technology, HTTP/2 suffers from creating new extended attack surfaces for attackers to target. Hence, server administrators need to understand they cannot simply turn on HTTP/2 and expect it to work without additional layers of security. Unfortunately, vendors usually cannot keep up with bad actors and allocate enough resources to mitigate all the vulnerabilities before damage is caused. Furthermore, some vendors (mostly open source) share the same code and therefore have the same vulnerabilities, meaning that vendors need to cooperate to mitigate a vulnerability which makes things even more complicated and lengthy.

The solution lies with an external component in the network that aims to reduce these risks. Such a component can respond faster to new vulnerabilities. When choosing a security vendor, consider one with a dedicated research group that monitors new technologies and offers proven solutions in advance.

Hacker Intelligence Initiative Overview

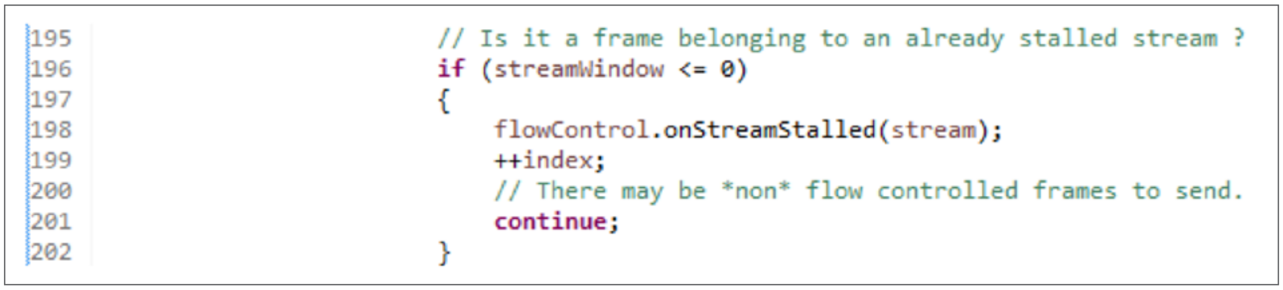
The Imperva Hacker Intelligence Initiative goes inside the cyber-underground and provides analysis of the trending hacking techniques and interesting attack campaigns from the past month. A part of the Imperva Defense Center research arm, the Hacker Intelligence Initiative (HII), is focused on tracking the latest trends in attacks, web application security and cyber-crime business models with the goal of improving security controls and risk management processes.

Appendix–Code Snippets of Vulnerable Components

Jetty Slow Read Attack (POST)

The relevant vulnerable code is located in the process function inside the HTTP2 Flusher class.

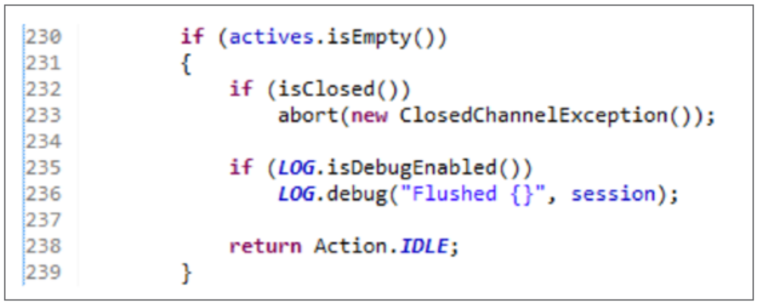
When the stream window is less than zero, no new entry is inserted into the "actives" collection.



```
// Is it a frame belonging to an already stalled stream ?
if (streamWindow <= 0)
{
    flowControl.onStreamStalled(stream);
    ++index;
    // There may be *non* flow controlled frames to send.
    continue;
}
```

Figure 30: process() function in jetty/http2/HTTP2Flusher.java

When the thread arrives at the end of this function with "actives" collection empty, it returns Action.IDLE, which makes it go to sleep.

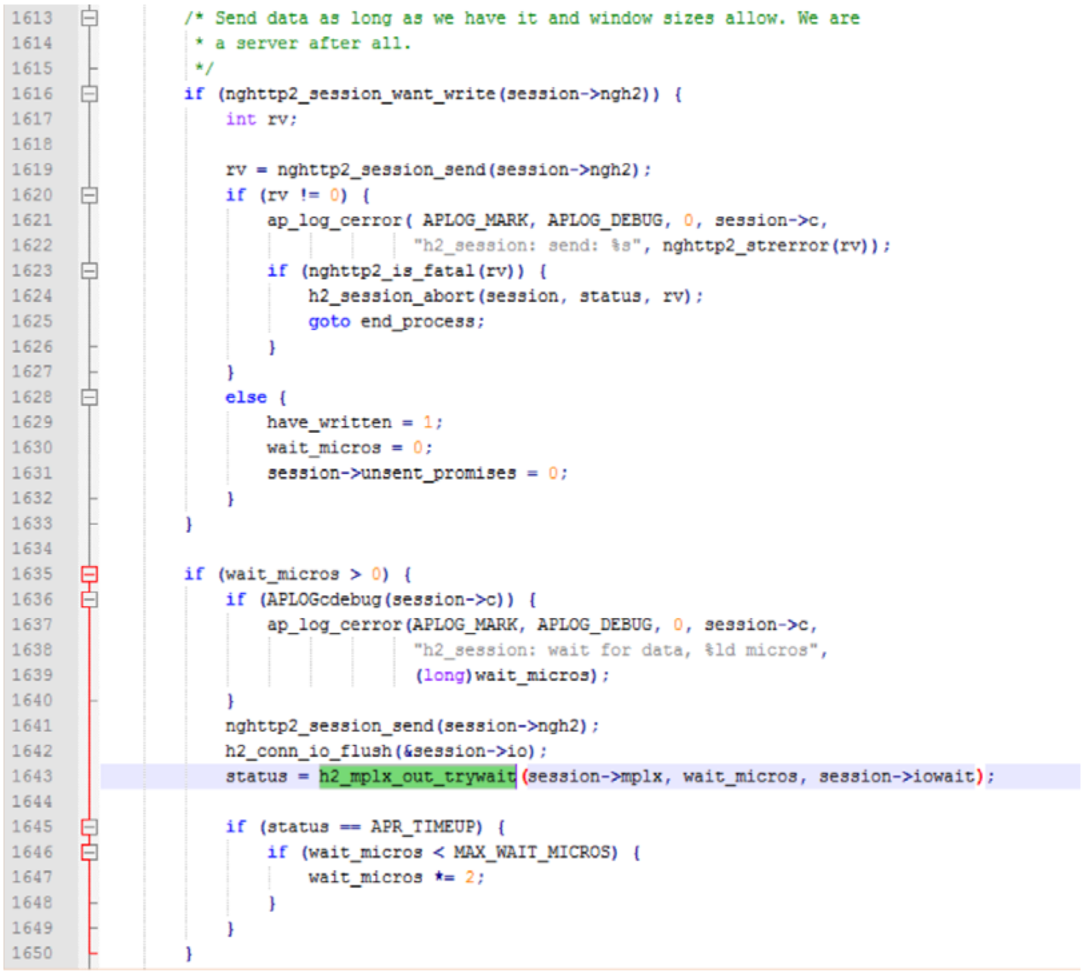


```
if (actives.isEmpty())
{
    if (isClosed())
        abort(new ClosedChannelException());

    if (LOG.isDebugEnabled())
        LOG.debug("Flushed {}", session);

    return Action.IDLE;
}
```

Figure 31: process() function in jetty/http2/HTTP2Flusher.java

Apache Slow Read Attack (GET)

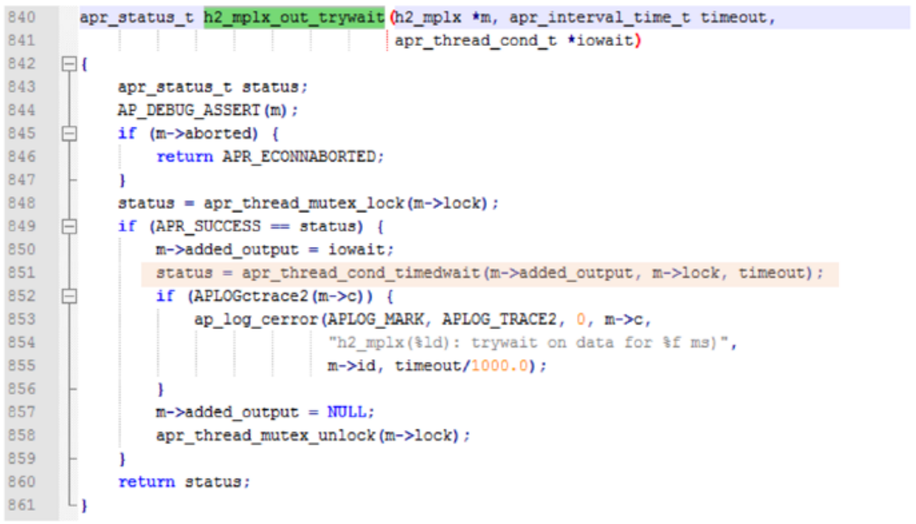
According to the server logs and code, we have found the server tries to send all the response data to the client in h2_session_process function (h2_session.c):



```
/* Send data as long as we have it and window sizes allow. We are
 * a server after all.
 */
if (nghttp2_session_want_write(session->ngh2)) {
    int rv;

    rv = nghttp2_session_send(session->ngh2);
    if (rv != 0) {
        ap_log_cerror( APLOG_MARK, APLOG_DEBUG, 0, session->c,
                      "h2_session: send: %s", nghttp2_strerror(rv));
        if (nghttp2_is_fatal(rv)) {
            h2_session_abort(session, status, rv);
            goto end_process;
        }
    }
    else {
        have_written = 1;
        wait_micros = 0;
        session->unsent_promises = 0;
    }
}

if (wait_micros > 0) {
    if (APLOGcdebug(session->c)) {
        ap_log_cerror(APLOG_MARK, APLOG_DEBUG, 0, session->c,
                      "h2_session: wait for data, %ld micros",
                      (long)wait_micros);
    }
    nghttp2_session_send(session->ngh2);
    h2_conn_io_flush(&session->io);
    status = h2_mplx_out_trywait(session->mplx, wait_micros, session->iowait);

    if (status == APR_TIMEUP) {
        if (wait_micros < MAX_WAIT_MICROS) {
            wait_micros *= 2;
        }
    }
}
```

Figure 32: h2_session_want_write() function in h2_session.c

If the window is low, it calls to the h2_mlpx_out_trywait function (h2_mplx.c):



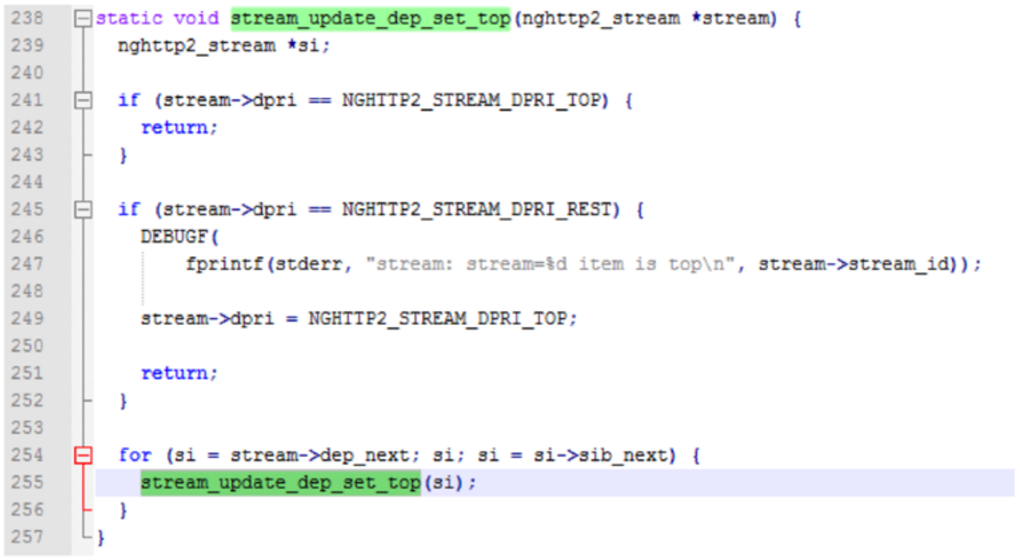
```
apr_status_t h2_mplx_out_trywait(h2_mplx *m, apr_interval_time_t timeout,
                                 apr_thread_cond_t *iowait)
{
    apr_status_t status;
    AP_DEBUG_ASSERT(m);
    if (m->aborted) {
        return APR_ECONNABORTED;
    }
    status = apr_thread_mutex_lock(m->lock);
    if (APR_SUCCESS == status) {
        m->added_output = iowait;
        status = apr_thread_cond_timedwait(m->added_output, m->lock, timeout);
        if (APLOGctrace2(m->c)) {
            ap_log_cerror(APLOG_MARK, APLOG_TRACE2, 0, m->c,
                          "h2_mplx(%ld): trywait on data for %f ms)",
                          m->id, timeout/1000.0);
        }
        m->added_output = NULL;
        apr_thread_mutex_unlock(m->lock);
    }
    return status;
}
```

Figure 33: h2_mplx_out_trywait() function in h2_mlpx.c

This function sends the thread to sleep which ultimately results in the server's DoS.

Dependency Cycle Attack on Nghttpd V.1.0.2-DEV

The relevant vulnerable code is located in the stream_update_dep_set_top function (nghttp2_stream.c). Under the assumption the graph is a tree, the last line of the function contains a recursive call to itself.

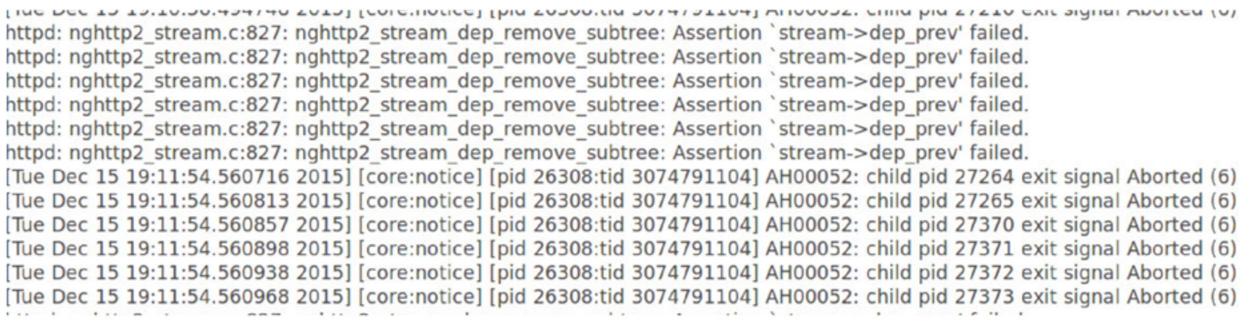


```
static void stream_update_dep_set_top(nghttp2_stream *stream) {
  nghttp2_stream *si;

  if (stream->dpri == NGHTTP2_STREAM_DPRI_TOP) {
    return;
  }

  if (stream->dpri == NGHTTP2_STREAM_DPRI_REST) {
    DEBUGF(
        fprintf(stderr, "stream: stream=%d item is top\n", stream->stream_id));

    stream->dpri = NGHTTP2_STREAM_DPRI_TOP;

    return;
  }

  for (si = stream->dep_next; si; si = si->sib_next) {
    stream_update_dep_set_top(si);
  }
}
```

Figure 34: stream_update_dep_set_top() function in nghttp2_streams.c

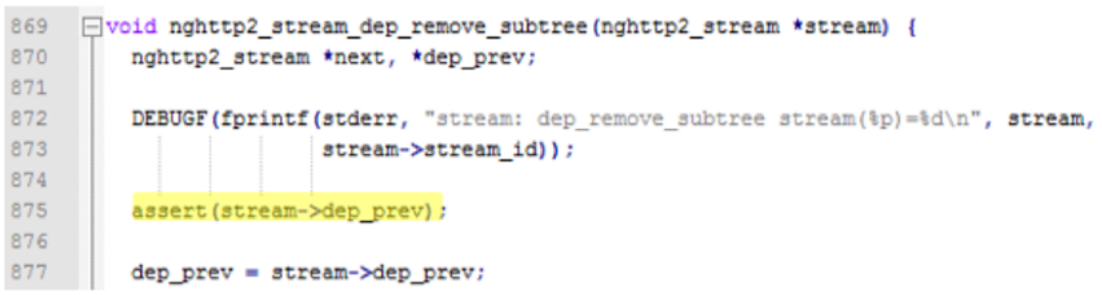
Dependency Cycle Attack on Apache 2.4.18

Following the server logs and code, we have found that the server asserts in nghttp2_stream_dep_remove_subtree function (nghttp2_streams.c) before a cycle is created:



```
httpd: nghttp2_stream.c:827: nghttp2_stream_dep_remove_subtree: Assertion `stream->dep_prev' failed.
httpd: nghttp2_stream.c:827: nghttp2_stream_dep_remove_subtree: Assertion `stream->dep_prev' failed.
httpd: nghttp2_stream.c:827: nghttp2_stream_dep_remove_subtree: Assertion `stream->dep_prev' failed.
httpd: nghttp2_stream.c:827: nghttp2_stream_dep_remove_subtree: Assertion `stream->dep_prev' failed.
httpd: nghttp2_stream.c:827: nghttp2_stream_dep_remove_subtree: Assertion `stream->dep_prev' failed.
httpd: nghttp2_stream.c:827: nghttp2_stream_dep_remove_subtree: Assertion `stream->dep_prev' failed.
[Tue Dec 15 19:11:54.560716 2015] [core:notice] [pid 26308:tid 3074791104] AH00052: child pid 27264 exit signal Aborted (6)
[Tue Dec 15 19:11:54.560813 2015] [core:notice] [pid 26308:tid 3074791104] AH00052: child pid 27265 exit signal Aborted (6)
[Tue Dec 15 19:11:54.560857 2015] [core:notice] [pid 26308:tid 3074791104] AH00052: child pid 27370 exit signal Aborted (6)
[Tue Dec 15 19:11:54.560898 2015] [core:notice] [pid 26308:tid 3074791104] AH00052: child pid 27371 exit signal Aborted (6)
[Tue Dec 15 19:11:54.560938 2015] [core:notice] [pid 26308:tid 3074791104] AH00052: child pid 27372 exit signal Aborted (6)
[Tue Dec 15 19:11:54.560968 2015] [core:notice] [pid 26308:tid 3074791104] AH00052: child pid 27373 exit signal Aborted (6)
```

Figure 35: Dependency Cycle, Apache log file



```c
void nghttp2_stream_dep_remove_subtree(nghttp2_stream *stream) {
  nghttp2_stream *next, *dep_prev;

  DEBUGF(fprintf(stderr, "stream: dep_remove_subtree stream(%p)=%d\n", stream,
                 stream->stream_id));

  assert(stream->dep_prev);

  dep_prev = stream->dep_prev;
```

Figure 36: nghttp2_stream_dep_remove_subtree() function in nghttp2_stream.c

Acknowledgment

The authors would like to thank Alex Maidanik and Avihai Cohen from Technion – Israeli Institute of Technology for their contributions to this research initiative.