**Background**:

HTTP/2 implementations enable a range of unique attack vectors for request smuggling even on previously secure websites. HTTP/2 has made many previously secure sites more vulnerable to request smuggling. Request smuggling is all about exploiting discrepancies on how server interpret the length of a request. HTTP/2 introduces a robust mechanism for calculating length which was previously thought to be impossible to exploit.

Although you won't see this in Burp, 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. Therefore, the length of the request is the sum of its frame length. In theory, there is no way to directly exploit this, but we often see HTTP/2 downgrading which introduces vulnerabilities.

**HTTP/2 downgrading**

HTTP/2 downgrading is the process of rewriting HTTP/2 requests using HTTP/1 syntax to generate an equivalent HTTP/1 request. Web servers and reverse proxies often do this in order to offer HTTP/2 support to clients while communicating with back-end servers that only speak HTTP/1. This practice is a prerequisite for many of the attacks covered in this section.

HTTP/2 downgradeing is where the front end initially used HTTP/2 then the front end rewrites it in HTTP/1 syntax which is then ultimately passed on to the backend. These are both just different ways to display the same information. This is a very common thing to happen in production environments, many reverse proxies have this as the default behavior and some don’t even allow an option to disable it.

\*\*\* in the notes we will represent HTTP/2 messages in a single human readable format similar to how it is displayed in burp. However in reality, HTTP/2 is a binary protocol and each message is transferred in multiple frames. \*\*\*

**H2.CL Vulnerabilities:**

HTTP/2 requests don’t have to specify length in a header. When converted to HTTP/1 the content-length header is often added to the request. Interestingly, HTTP/2 requets can contain a content-length header and sometimes the front end will simply reuse this value in the resulting HTTP/1 request. HTTP/2 spec does say the content-length val must match the length calculated using the built in mechanism but this is often not validated before the downgrade. Thus, we can often smuggle a request using a misleading content-length header. In this case, the front-end will use the calculated length but the backend has to refer to the content-length header we created resulting in desync. Ex:

**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

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

\*\*\*\*\* With some attacks we want headers to be appended to our smuggled request. However, these can interfere with our attack resulting in duplicate header errors and the like. We can mitigate this by putting a content length header in our smuggled request with a small value, slightly longer then the body of our smuggled request. We still want the victims request to be appended but we do not want their headers sometimes. Using this small value for content-length we can achieve just that.

**H2.TE Vulnerabilities:**

HTTP/2 is incompatible with the transfer encoding header and the spec recommends that this header be stripped away or the request should be entirely blocked. If the server fails to do this and downgrades the request to a backend that supports chunked encoding this can enable request smuggling as shown below:

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

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

Essentially if the website is vulnerable to H2.CL or H2.TE we can potentially leverage the same attacks detailed in the exploiting file.

**Hidden HTTP/2 support**

**\*\*\* Essentially we should always be checking for hidden HTTP/2 support on all applications we test. If this works we have access to an entirely new attack surface which very well may introduce smuggling vulns even if they were not present when using HTTP/1.1 \*\*\***

Browsers and other clients, including Burp, typically only use HTTP/2 to communicate with servers that explicitly advertise support for it via ALPN as part of the TLS handshake.

Some servers support HTTP/2 but fail to declare this properly due to misconfiguration. In such cases, it can appear as though the server only supports HTTP/1.1 because clients default to this as a fallback option. As a result, testers may overlook viable HTTP/2 attack surface and miss protocol-level issues, such as the examples of HTTP/2 downgrade-based request smuggling that we covered above.

To force Burp Repeater to use HTTP/2 so that you can test for this misconfiguration manually:

1. From the **Settings** dialog, go to **Tools > Repeater**.
2. Under **Connections**, enable the **Allow HTTP/2 ALPN override** option.
3. In Repeater, go to the **Inspector** panel and expand the **Request attributes** section.
4. Use the switch to set the **Protocol** to **HTTP/2**. Burp will now send all requests on this tab using HTTP/2, regardless of whether the server advertises support for this.

**Response Queue Poisoning:**

Response queue poisoning is a powerful form of request smuggling attack that causes a front-end server to start mapping responses from the back-end to the wrong requests. In practice, this means that all users of the same front-end/back-end connection are persistently served responses that were intended for someone else.

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.

The impact of this is usually catastrophic. Once the queue is poisoned an attack can capture other users response simply by issuing arbitrary follow-up requests which could divulge sensitive information.

\*\*\* This exploit causes significant collateral damage effective breaking the site for other users whos traffic is being routed across the same connection. While attempting to browse normally, users will be sent seemingly random response which will prevent the site from working correctly. \*\*\*

How To Construct an Attack:

For a successful response queue poisoning attack, the following criteria must be met:

* The TCP connection between the front-end server and back-end server is [reused for multiple request/response cycles](https://portswigger.net/web-security/request-smuggling#what-happens-in-an-http-request-smuggling-attack).
* The attacker is able to successfully smuggle a complete, standalone request that receives its own distinct response from the back-end server.
* The attack does not result in either server closing the TCP connection. Servers generally close incoming connections when they receive an invalid request because they can't determine where the request is supposed to end.

Background:

Normally we smuggle a partial request so the following request is just appended to it. If our smuggled request has a body, then the number of bytes specified by the content length header in the smuggled request will be appended to the body. This can often lead to 3 full requests going to the server where the third request is just the leftover bytes of the request that was partially appended to the body. (As we are familiar with). This last invalid, malformed request will typically cause the server to terminate the connection. This is why for this attack we must smuggle in two complete requests.

**Smuggling a complete request**

With a bit of care, you can smuggle a complete request instead of just a prefix. As long as you send exactly two requests in one, any subsequent requests on the connection will remain unchanged:

**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

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

Here no invalid requests would hit the backend so the connection should remain open.

**Desynchronizing the response queue:**

When you smuggle a complete request, the front-end server still thinks it only forwarded a single request but the backend sees 2 distinct requests and will send two responses. The front-end correctly maps the first response to the initial "wrapper" request and forwards this on to the client. As there are no further requests awaiting a response, the unexpected second response is held in a queue on the connection between the front-end and back-end. 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. This cycle is repeated every time a new request is forwarded down the same connection to the back-end.

**Stealing Other Users Responses:**

Once the queue is poisoned the attacker can just send arbitrary request and will then be served other users responses. There is no way to control what responses well get. It is often very useful to use a fuzzer to send a bunch of requests and then manually look through all of the responses. We will be able to get responses as long as the connection remains open. A common default is to close the connection after its handled 100 requests but this varies. It is extremely easy to just repeat this attack so we can essentially capture as many requests as we want.

\*\*\* To make it easier to differentiate stolen responses from responses to your own requests, try using a non-existent path in both of the requests that you send. That way, your own requests should consistently receive a 404 response, for example. \*\*\*

Any information can be useful for an exploit. If we can access other users session cookie or other sensitive information we may be able to even access their account.

**Request Smuggling via CRLF Injection:**

**\*\*\*\*\* to mess with headers go to the inspector pannel on the right side of repeater find the Request Headers section. Modify requests there. \r\n can be added by pressing ctrl+enter when entering name or value. \*\*\***

**\*\*\* Very important: CRLF can be present both in header NAMES and/or VALUES!!! \*\*\***

**When looking for CRLF try injecting malformed headers and look for errors such as an arbitrary host name, if it is accepted and leads to an unknown host header clearly we have CRLF. Any unique behavior we get from injecting malformed headers should be explored.**

Even if an app prevents basic H2 smuggling vectors such as by validating the content length or stripping transfer encoding headers, HTTP/2s binary format enables other methods to bypass these front end protections.

**\*\*\*\*\*\* In HTTP/1, you can sometimes exploit discrepancies between how servers handle standalone newline (\n) characters to smuggle prohibited headers. If the back-end treats this as a delimiter, but the front-end server does not, some front-end servers will fail to detect the second header at all. This is very useful to know when testing HTTP/1.**

**Foo: bar\nTransfer-Encoding: chunked**

This discrepancy doesn’t exist with the full CRLF (\r\n) sequence because HTTP/1 universally agrees this terminates the header.

However, with HTTP/2 messages are binary instead of text-based. The boundaries of the headers are explicit predetermined offsets rather than delimiter characters. Thus, CRLF sequences can be used within headers without causing them to terminate or split. However if the HTTP/2 request is rewritten as HTTP/1 this CRLF would then be used to determine headers allowing us to potentially use a Transfer encoding header.

**HTTP Request Tunnelling:**

**\*\*\*\*\* Refer to the bold notes in the CRLF section above for more information on CRLF detection and creating malformed headers\*\*\*\*\*\***

**\*\*\* also refer to this lab:** [**https://portswigger.net/web-security/request-smuggling/advanced/request-tunnelling/lab-request-smuggling-h2-bypass-access-controls-via-request-tunnelling**](https://portswigger.net/web-security/request-smuggling/advanced/request-tunnelling/lab-request-smuggling-h2-bypass-access-controls-via-request-tunnelling) **it contains a ton of great information for this vuln. It details how to find CRLF, how to leak internal headers, how to navigate some normal errors/ issues and how to formulate exploit payloads using the inspector tab, Reall great resource\*\*\***

Many vulnerabilities we went over are only possible where the same connection between the front and back end handle multiple quests. Some servbers will reuse the connection for any request while others have more strict policies. For example, some servers will only allow requests originating from the same IP or client to reuse the connection. Others wont reuse the connection at all. This limits exploitability as we cannot influence or see other users traffic.

Although we cannot posion the socket in these cases, we can still send a single request that illicits two responses from the back-end. This potentially enables us to hide a request and its response from the font end all together. This will bypass the security features implemented at the front end. Sometimes servers that are specially designed to prevent request smuggling will not prevent request tunneling. Tunneling offers a more limited form of smuggling but can still lead to high severity exploits.

Request tunnelling is possible with both HTTP/1 and HTTP/2 but is considerably more difficult to detect in HTTP/1-only environments. Due to the way persistent (keep-alive) connections work in HTTP/1, even if you do receive two responses, this doesn't necessarily confirm that the request was successfully smuggled.

In HTTP/2 on the other hand, each "stream" should only ever contain a single request and response. If you receive an HTTP/2 response with what appears to be an HTTP/1 response in the body, you can be confident that you've successfully tunneled a second request.

**With tunneling we can potentially leak internal headers**:

When only tunneling is possible we cannot leak headers as showed in prior labs. But if there is HTTP/2 downgrading we have an alternative solution.

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:

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

In this case, both the front-end and back-end agree that there is only one request. What's interesting is that they can be made to disagree on where the headers end.

The front-end sees everything we've injected as part of a header, so adds any new headers after the trailing comment= string. On the other hand, the back-end sees the \r\n\r\n sequence and thinks this is the end of the headers. The comment= string, along with the internal headers, are treated as part of the body. The result is a comment parameter with the internal headers as its value.

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

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

**Blind request tunnelling**

Some front-end servers read in all the data they receive from the back-end. This means that if you successfully tunnel a request, they will potentially forward both responses to the client, with the response to the tunnelled request nested inside the body of the main response.

Other front-end servers only read in the number of bytes specified in the Content-Length header of the response, so only the first response is forwarded to the client. This results in a blind request tunnelling vulnerability because you won't be able to see the response to your tunnelled request.

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

**Non-blind request tunnelling using HEAD**

Blind request tunnelling can be tricky to exploit, but you can occasionally make these vulnerabilities non-blind by using HEAD requests.

Responses to HEAD requests often contain a content-length header even though they don't have a body of their own. This normally refers to the length of the resource that would be returned by a GET request to the same endpoint. Some front-end servers fail to account for this and attempt to read in the number of bytes specified in the header regardless. If you successfully tunnel a request past a front-end server that does this, this behavior may cause it to over-read the response from the back-end. As a result, the response you receive may contain bytes from the start of the response to your tunnelled request.

**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 | |

As you're effectively mixing the content-length header from one response with the body of another, using this technique successfully is a bit of a balancing act.

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, as in the example above. On the other hand, if the returned content-length is longer than the response to your tunneled request, you will likely encounter a timeout as the front-end server is left waiting for additional bytes to arrive from the back-end.

Fortunately, with a bit of trial and error, you can often overcome these issues using one of the following solutions:

* 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.

**Web Cache Poisoning via HTTP/2 request tunneling:**

Although more limited than normal smuggling we can still use tunneling to create powerful exploits especially when chained with other exploits like web cache poisoning.

With non-blind request tunnelling, you can effectively mix and match the headers from one response with the body of another. If the response in the body reflects unencoded user input, you may be able to leverage this behavior for [reflected XSS](https://portswigger.net/web-security/cross-site-scripting/reflected) in contexts where the browser would not normally execute the code.

For example, the following response contains unencoded, attacker-controllable input:

HTTP/1.1 200 OK

Content-Type: application/json

{ "name" : "test<script>alert(1)</script>" }

[etc.]

By itself, this is relatively harmless. The Content-Type means that this payload will simply be interpreted as JSON by the browser. But consider what would happen if you tunnel the request to the back-end instead. This response would appear inside the body of a different response, effectively inheriting its headers, including the content-type.

|  |  |
| --- | --- |
| :status | 200 |
| content-type | text/html |
| content-length | 174 |
| HTTP/1.1 200 OK Content-Type: application/json { "name" : "test<script>alert(1)</script>" } [etc.] | |

As caching takes place on the front-end, caches can also be tricked into serving these mixed responses to other users.