Web Security

Week 4 - Exceptions to the Same Origin Policy

Old Dominion University
Department of Computer Science
CS 495/595 Spring 2022
Michael L. Nelson <mln@cs.odu.edu>

2022-02-07

Same origin policy exceptions

- From last week: There are explicit opt-in mechanisms like document.domain, fragment identifier communication, and the postMessage API
- There are also automatic exceptions
  - Need to be aware of these!
  - Source of many security issues!

<img>, <script>, and CSS are same origin policy exceptions!

```
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8' />
    <link rel='stylesheet' href='https://other1.com/style.css' />
  </head>
  <body>
    <img src='https://other2.com/image.png' />
    <script src='https://other3.com/script.js'></script>
  </body>
</html>
```

Same origin policy and ambient authority

- Remember: Ambient authority is implemented by cookies
- One consequence: attacker.com can embed user's real avatar from target.com:

```
<h1>Welcome to your account!</h1>
<img src='https://target.com/avatar.png' />
```

In the above example, the URL `target.com/avatar.png`is used for all accounts, with the cookie used by the server to distinguish your avatar from my avatar. This is instead of having the server include different URLs (target.com/avatar-001.png, avatar-002.png, etc.) which would make the HTML page less cacheable.

Solution: SameSite cookies

- Use SameSite cookie attribute to prevent cookie from being sent with requests initiated by other sites

From target.com:

```
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
```

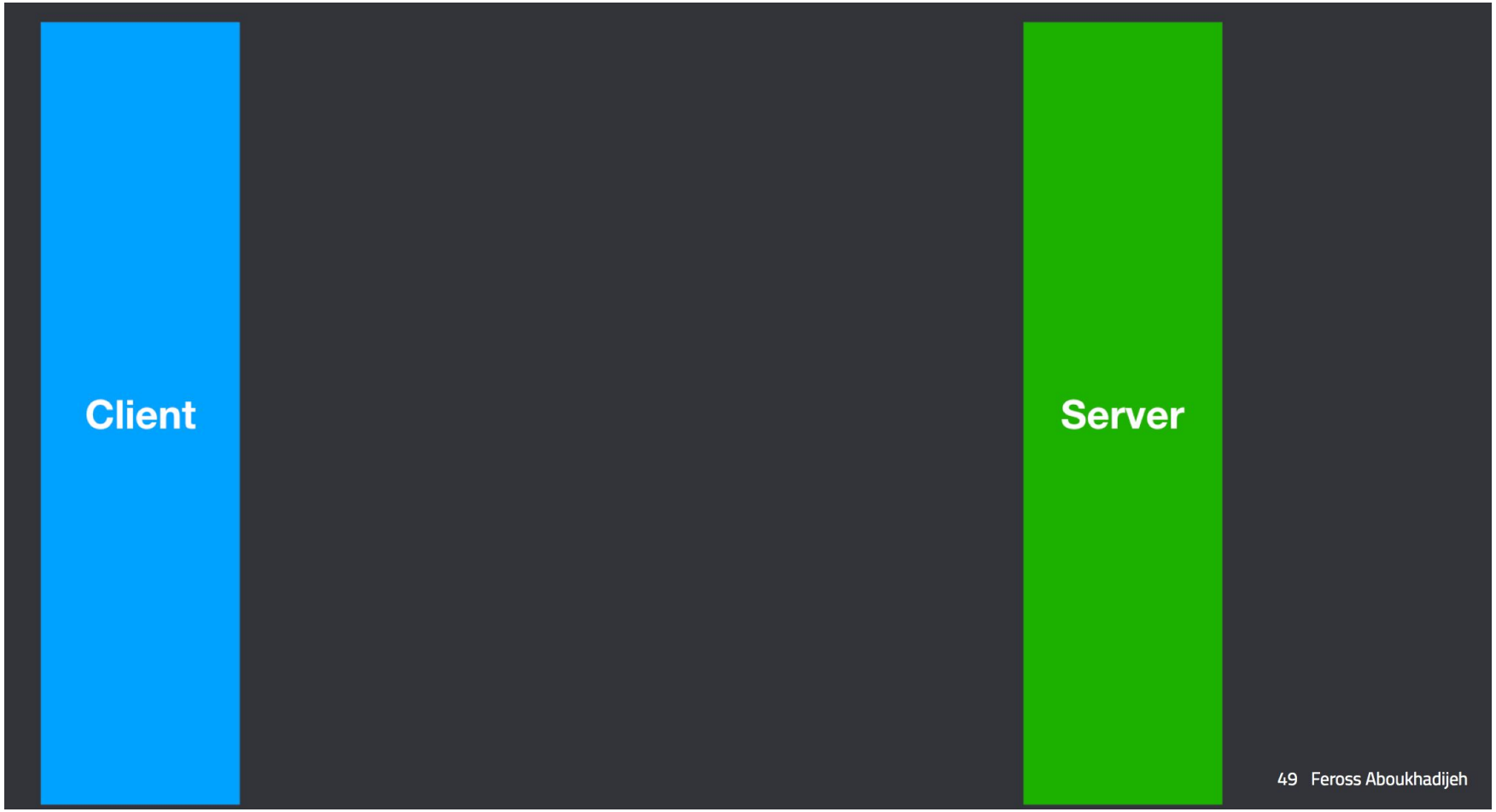
From attacker.com:

```
GET /avatar.png HTTP/1.1
Referer: https://attacker.com/
```

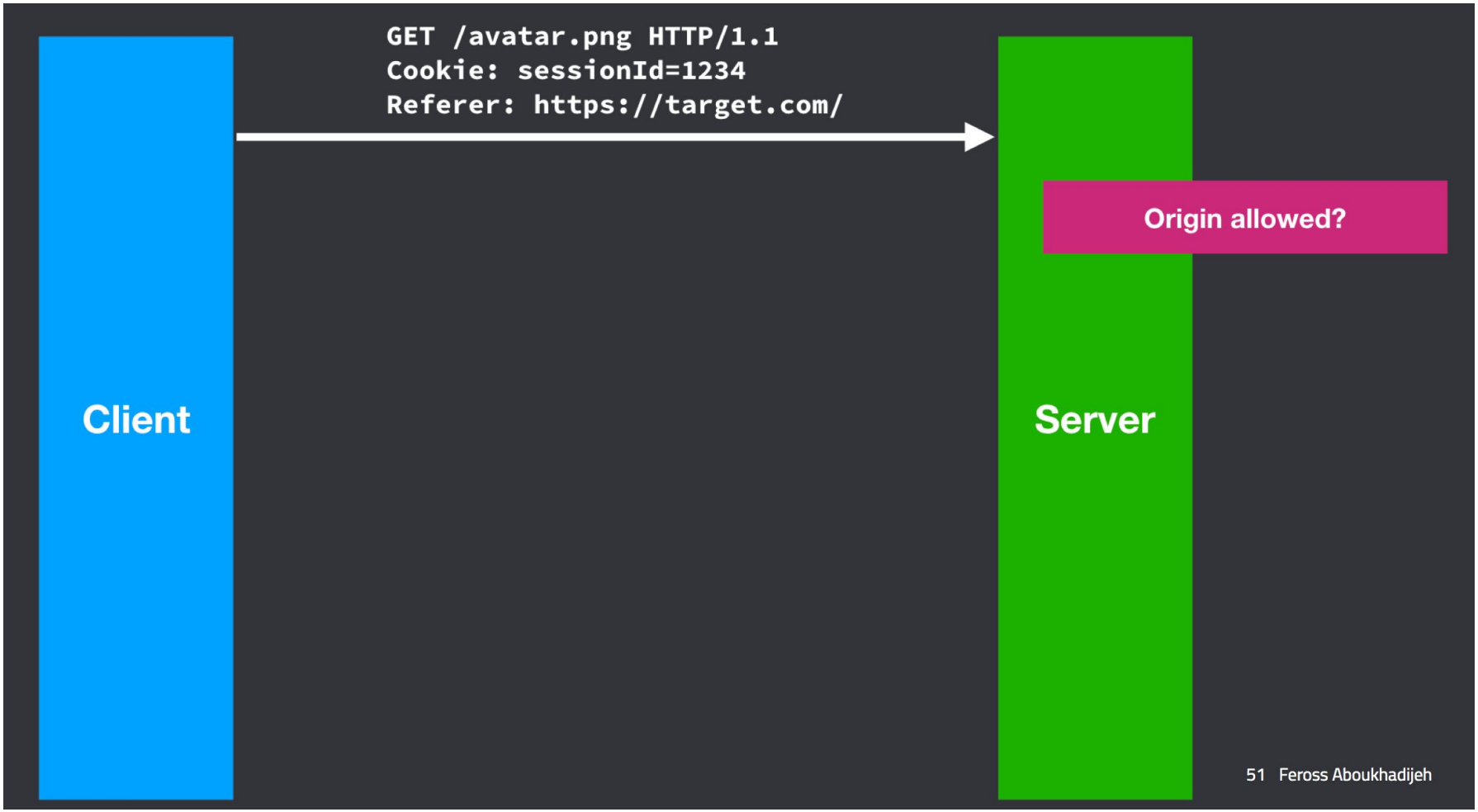
if `attacker.com` is a phishing site, it is now a lot harder to build a convincing look-alike.

Solution: Referer header

- Inspect the Referer HTTP header
- Reject any requests from origins not on an "allowlist"
- One gotcha: Watch out for HTTP caches!


Client
Server
49 Feross Aboukhadijeh

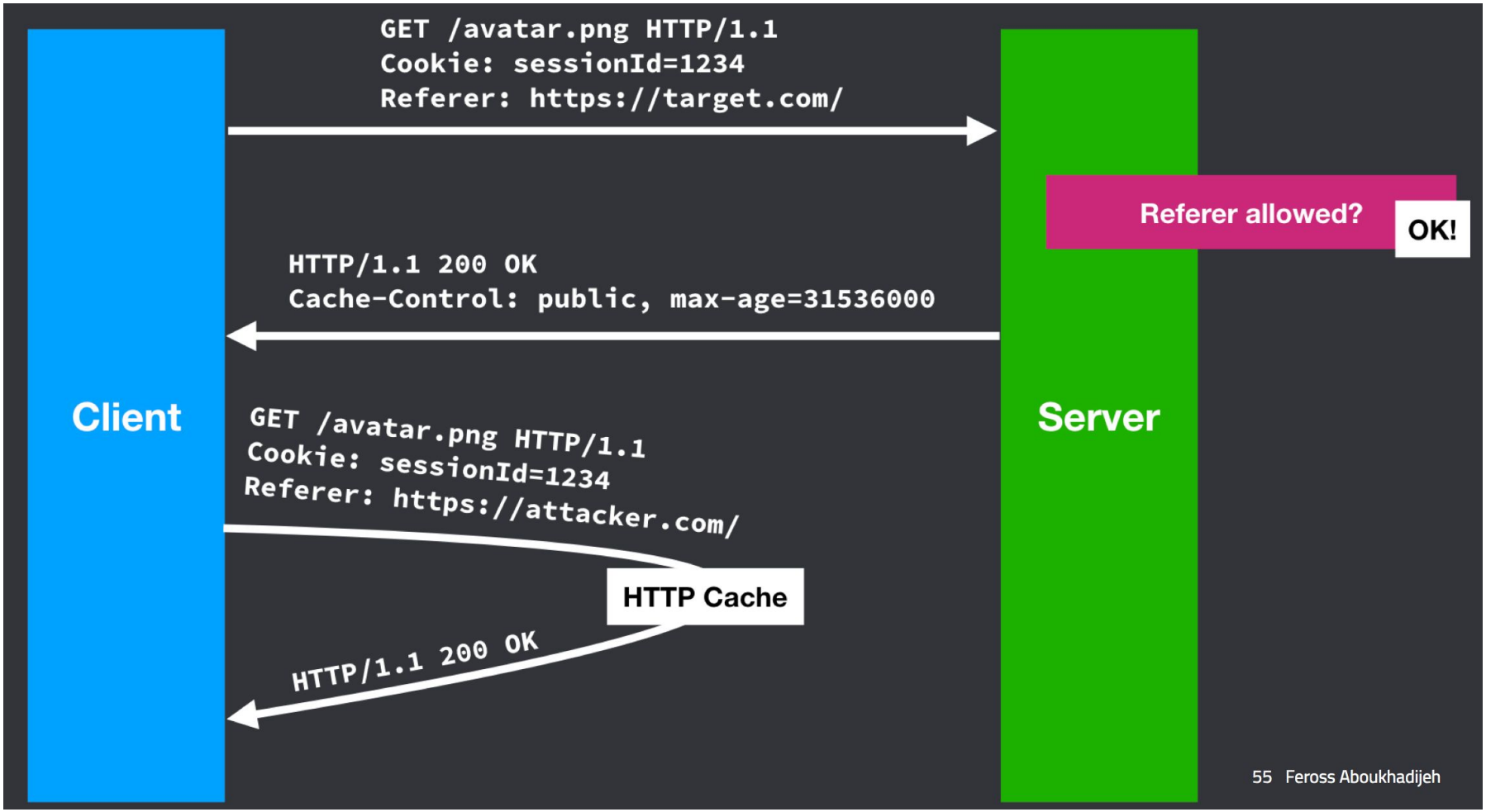

GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
Client
Server
50 Feross Aboukhadijeh


GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
Client
Server
Origin allowed?
51 Feross Aboukhadijeh


GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
Client
Server
Origin allowed?
OK!
52 Feross Aboukhadijeh


GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
Origin allowed?
OK!
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
Client
Server
53 Feross Aboukhadijeh


GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
Origin allowed?
OK!
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
Client
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://attacker.com/
Server
?
54 Feross Aboukhadijeh


GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://target.com/
Referer allowed?
OK!
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
Client
Server
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://attacker.com/
HTTP Cache
HTTP/1.1 200 OK
55 Feross Aboukhadijeh

Referer

- Inspect the Referer HTTP header
- Reject any requests from origins not on an "allowlist"
- One gotcha: Watch out for HTTP caches!
  - Add a `Vary: Referer` header
    - this could create a lot of otherwise duplicate cache entries
  - Or, add a `Cache-Control: no-store` header
    - this could create a lot unnecessary cache misses
- Another gotcha: Sites can opt out of sending the Referer header!
  - Defeats this whole mechanism. So, just use SameSite cookies!

Remember that forms can POST to another origin!

```
<form method='POST' action='http://localhost:4000/transfer'>
  <input name='amount' value='100' />
  <input name='to' value='alice' />
  <input type='submit' value='Send' />
</form>
<script>
  document.forms[0].submit()
</script>
```

57 Feross Aboukhadijeh

Cookies do not obey same origin policy

- Cookies were created before Same Origin Policy so have different security model
- Cookies are *more specific* than Same Origin Policy
  - Path is ineffective because same origin pages can access each other's DOMs (recall week 2's lecture)
- Cookies are *less specific* than Same Origin Policy
  - Different origins can mess with each others cookies (e.g., attacker.odu.edu can set cookies for odu.edu)
- This is why, among other reasons, we have leonline.odu.edu and not odu.edu/leoonline

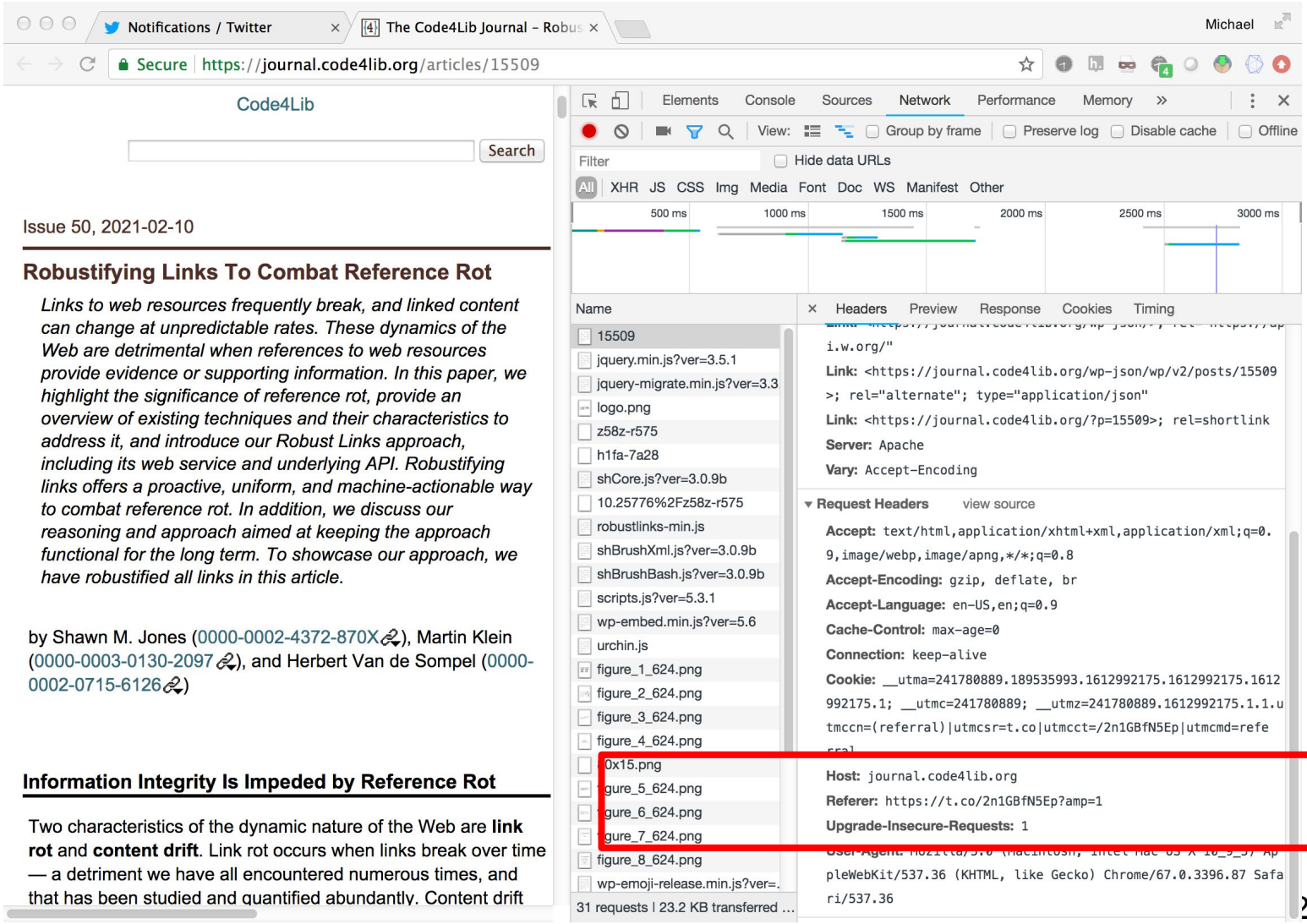
We can harden or relax the same origin policy

Revisiting what the same origin policy allows

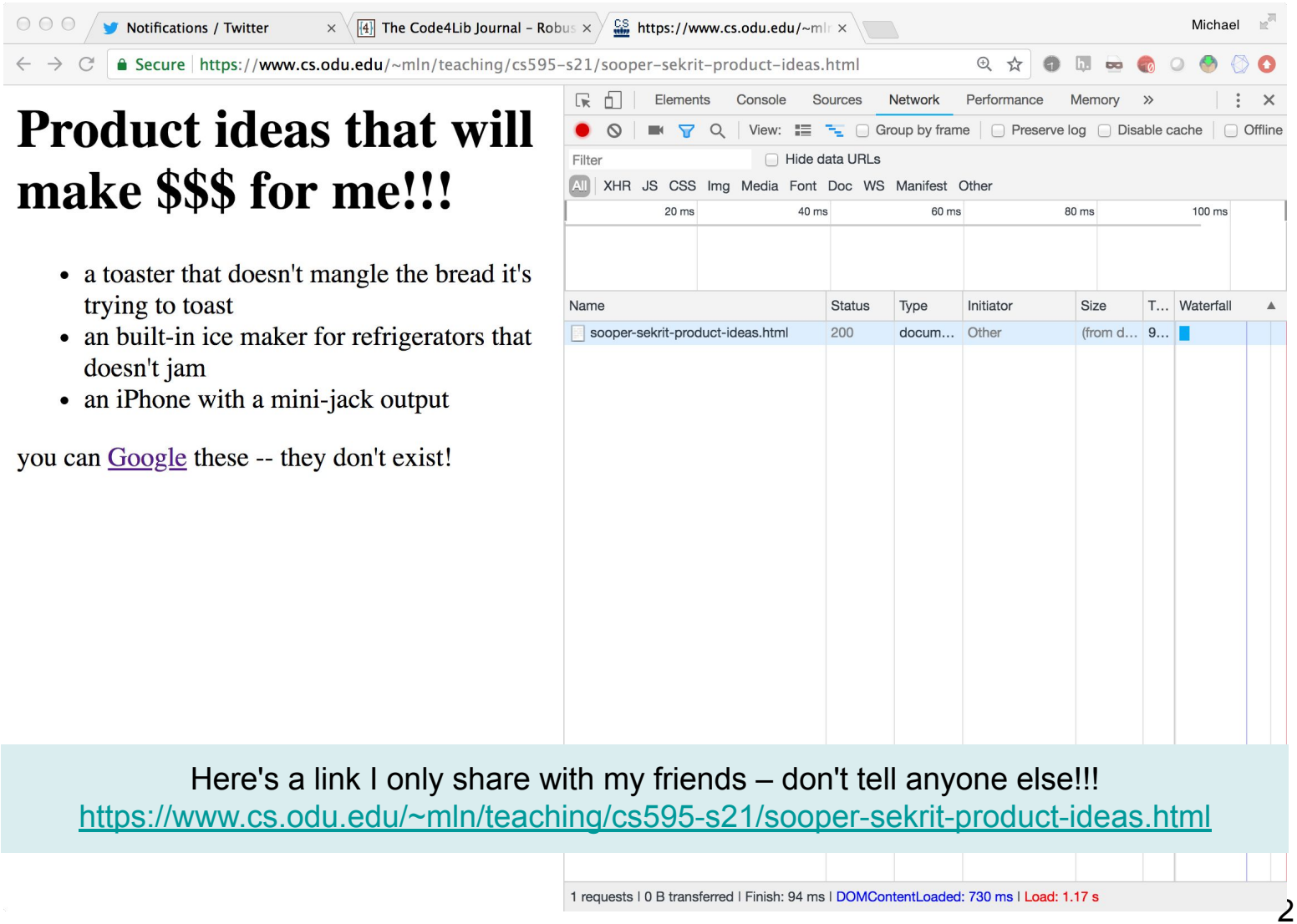
- Is site A allowed to link to site B? Yes!
- Is site A allowed to embed site B? Yes!
- Is site A allowed to embed site B and modify its contents? No!
- Is site A allowed to submit a form to site B? Yes!
- Is site A allowed to embed images from site B? Yes!
- Is site A allowed to embed scripts from site B? Yes!
- Is site A allowed to read data from site B? No!

Hardening the same origin policy

Recall the Referer request header

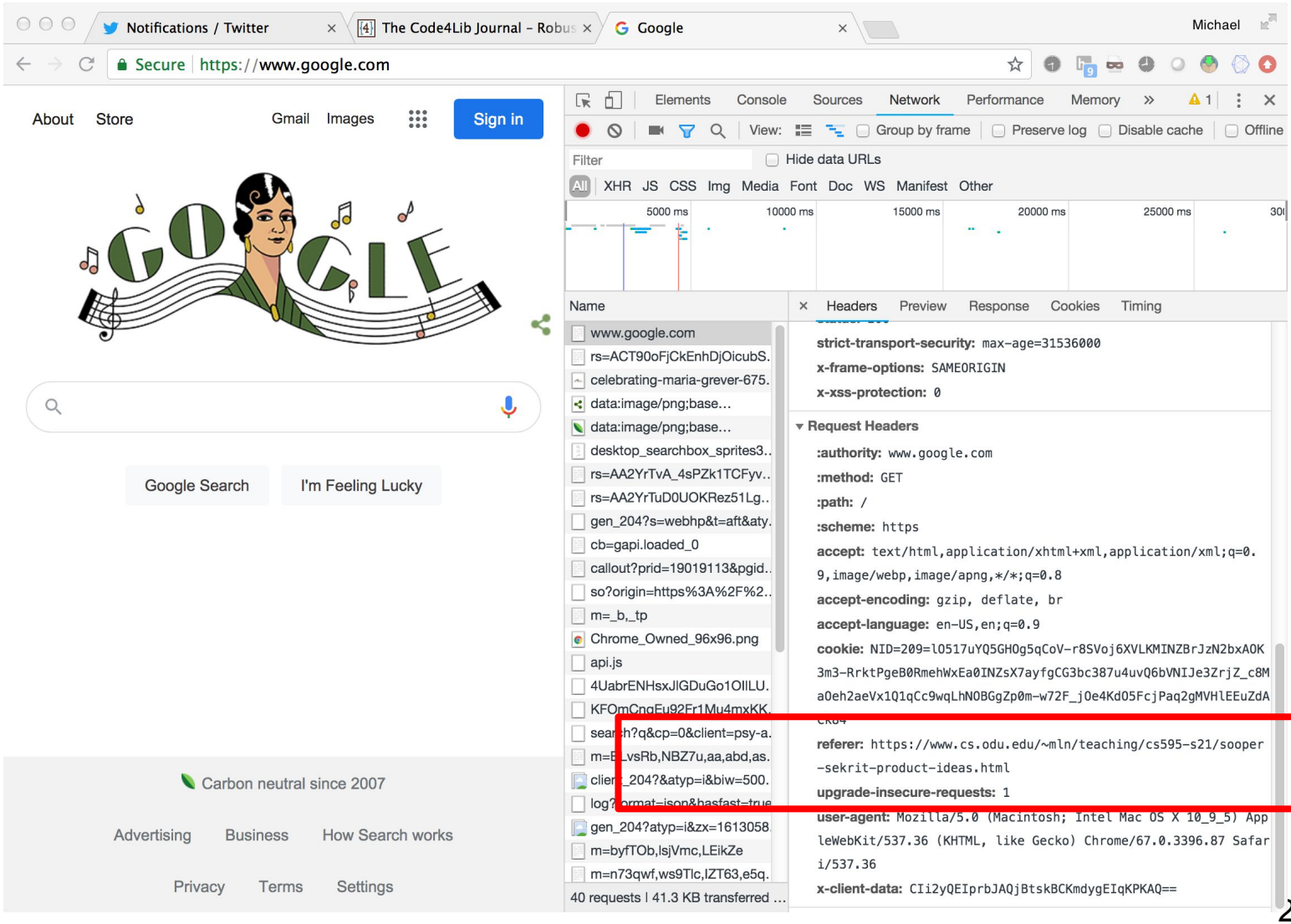


Referer can leak info to other sites



Here's a link I only share with my friends – don't tell anyone else!!!
https://www.cs.odu.edu/~mln/teaching/cs595-s21/sooper-sekrit-product-ideas.html

Now Google knows my secret URL



Referrer-Policy response header

Syntax

The original header name `Referer` is a misspelling of the word "referrer". The `Referrer-Policy` header does not share this misspelling.

```
Referrer-Policy: no-referrer
Referrer-Policy: no-referrer-when-downgrade
Referrer-Policy: origin
Referrer-Policy: origin-when-cross-origin
Referrer-Policy: same-origin
Referrer-Policy: strict-origin
Referrer-Policy: strict-origin-when-cross-origin
Referrer-Policy: unsafe-url
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy

docs.google.com prevents your browser from leaking `Referer` info to non-Google Docs sites

```
$ curl -I
https://docs.google.com/presentation/d/1NFyC1huil5uOic4ITtEUlogUqAfP8PQIpNGChtUJE7Y/edit#slide=id.gbc694
ac400_0_109
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
X-Robots-Tag: noindex, nofollow, nosnippet
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Mon, 01 Jan 1990 00:00:00 GMT
Date: Wed, 10 Feb 2021 23:09:33 GMT
Content-Length: 447525
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Strict-Transport-Security: max-age=31536000; includeSubDomains
Content-Security-Policy: base-uri 'self';object-src 'self' blob:;report-uri
https://docs.google.com/presentation/cspreport;script-src 'nonce-gLqphbwqOVspb3On6UUgRg' 'unsafe-inline'
'strict-dynamic' https: http: 'unsafe-eval'
Referrer-Policy: origin
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Server: GSE
Set-Cookie:
NID=209=I91MVBS_rjRS743-o56cJmCDW265Pc_heOTp1mkqZoWNSD607unmBwGbJtPAqW_yuWuLB1JcnelNlusFvFPv5vbceFJ2pQya
PkF6fZEUk-M6bJv8N_ccE6xDZAId6PHIU4dvTtcbZ9S0u_5waaDSs1ExL8DZ03ddkoy3Ytag5_w; expires=Thu, 12-Aug-2021
23:09:33 GMT; path=/; domain=.google.com; HttpOnly
Set-Cookie: S=apps-presentations=JGHgefN3DDdAbjDbv_fnwWTzOMb8ogaxLXJwVGrigXg; Domain=.docs.google.com;
Expires=Thu, 11-Feb-2021 00:09:33 GMT;
Path=/presentation/d/1NFyC1huil5uOic4ITtEUlogUqAfP8PQIpNGChtUJE7Y; Secure; HttpOnly; SameSite=none
Set-Cookie: GFE_RTT=462; Domain=.docs.google.com; Expires=Wed, 10-Feb-2021 23:14:33 GMT; Path=/; Secure;
Priority=LOW; SameSite=strict
Alt-Svc: h3-29=":443"; ma=2592000,h3-T051=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443";
ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"
```

Policy	Document	Navigation to	Referrer
`no-referrer`	https://example.com/page	*anywhere*	*(no referrer)*
`no-referrer-when-downgrade`	https://example.com/page	https://example.com /otherpage	https://example.com/page
		https://mozilla.org	https://example.com/page
		http://example.org	*(no referrer)*
`origin`	https://example.com/page	*anywhere*	https://example.com/
`origin-when-cross-origin`	https://example.com/page	https://example.com /otherpage	https://example.com/page
		https://mozilla.org	https://example.com/
		http://example.com/page	https://example.com/
`same-origin`	https://example.com/page	https://example.com /otherpage	https://example.com/page
		https://mozilla.org	*(no referrer)*
`strict-origin`	https://example.com/page	https://mozilla.org	https://example.com/
		http://example.org	*(no referrer)*
	http://example.com/page	*anywhere*	http://example.com/
`strict-origin-when-cross-origin`	https://example.com/page	https://example.com /otherpage	https://example.com/page
		https://mozilla.org	https://example.com/
		http://example.org	*(no referrer)*
`unsafe-url`	https://example.com /page?q=123	*anywhere*	https://example.com /page?q=123

Policies & Examples

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy

du

Change the response depending on how the user got to your site?

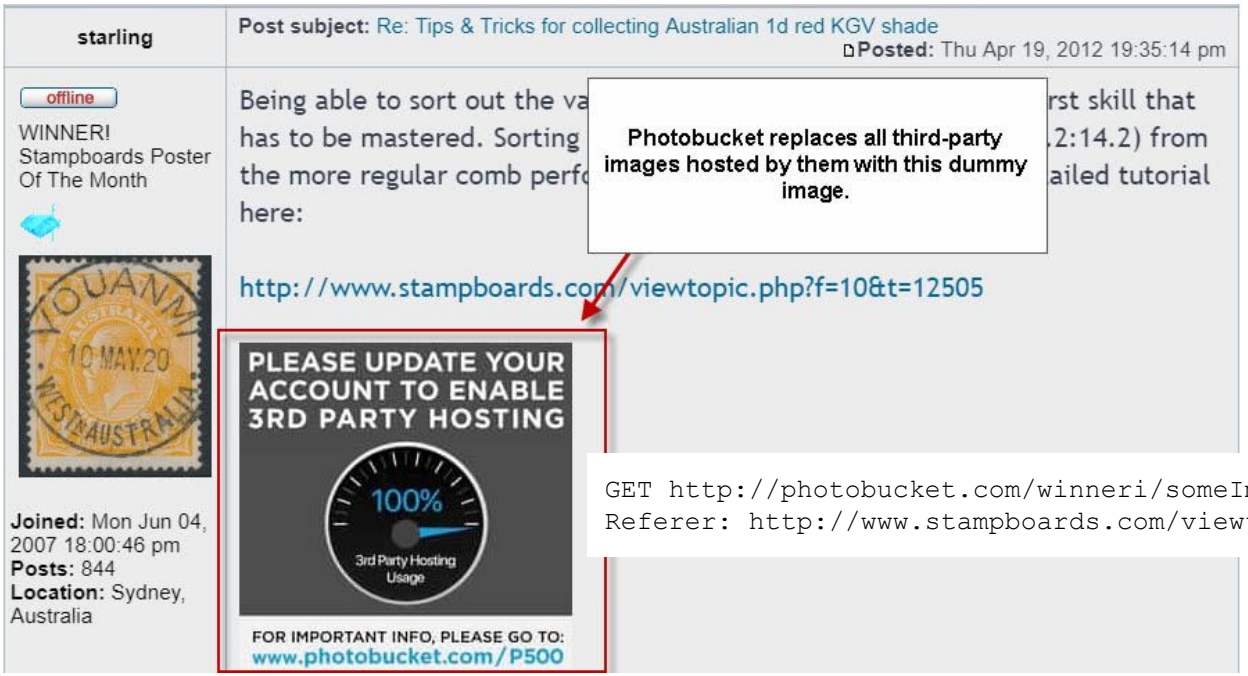
```
GET https://badguys.com/ HTTP/1.1
Referer: https://goodguys.com/somepage.html
…
```

Should badguys.com:

- block the request?
- modify the response?
- offer a discount or incentive?

Cf. Cloaking

In the bad old days...



- photo hosting sites would check the Referer: request header to ensure that it was coming from the same site (e.g., photobucket.com)
  - photo hosting site would rate limit based on page views per month, etc.
- average users did not know to do:

https://www.ghacks.net/2017/07/02/photobucket-alternatives-for-third-party/

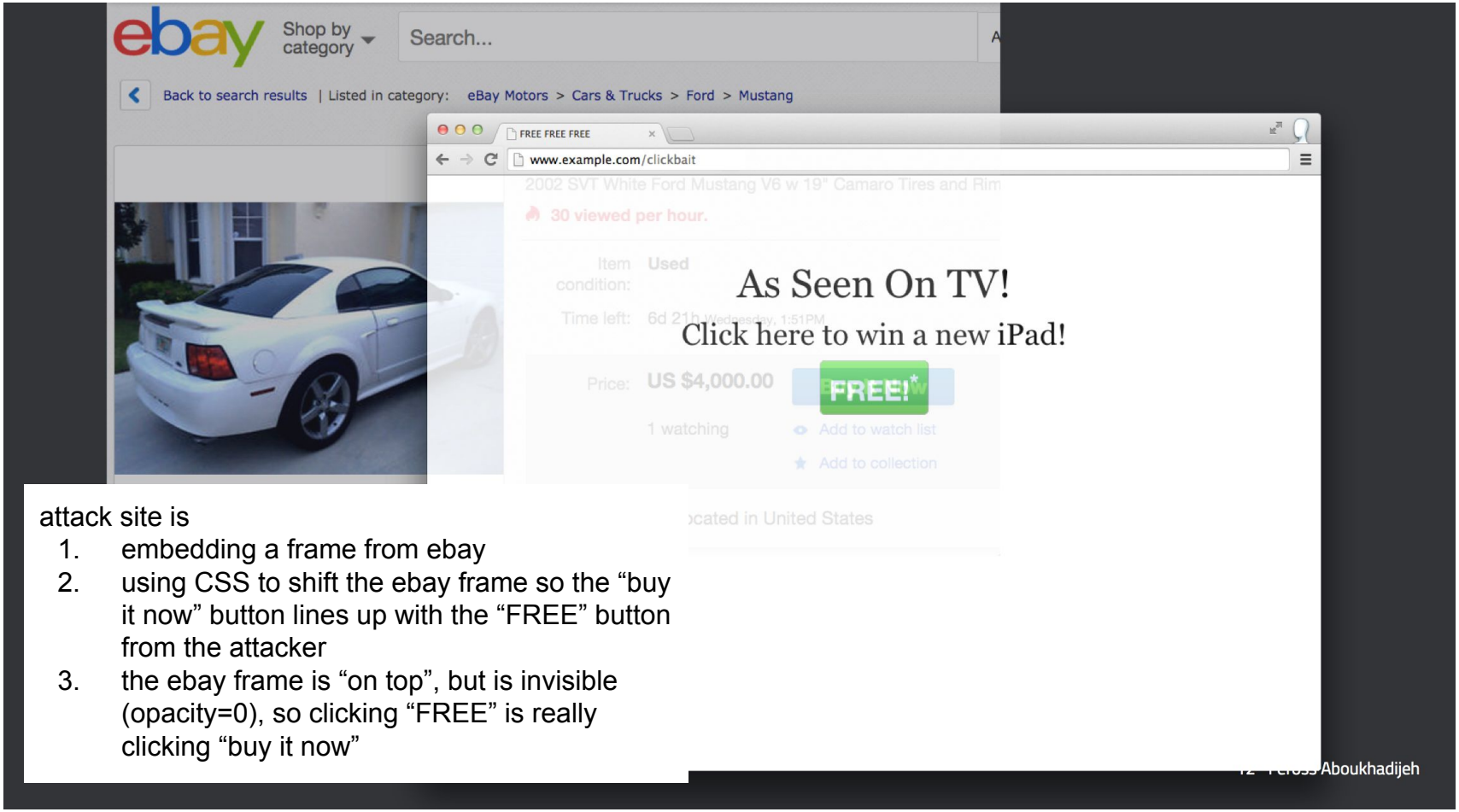
The server can't really prevent being linked to, but it can make it frustrating for normal users

a tutorial to implement limiting 3rd party image hosting, the demos of which, ironically, no longer work:
https://alistapart.com/article/hotlinking/

many browser extensions to defeat these kinds of restrictions, here's one:
https://github.com/Ryan-Myers/photobucket-embed-fix

Sites *can* control who embeds them

Clickjacking!



attack site is

1. embedding a frame from ebay
2. using CSS to shift the ebay frame so the “buy it now” button lines up with the “FREE” button from the attacker
3. the ebay frame is “on top”, but is invisible (opacity=0), so clicking “FREE” is really clicking “buy it now”

Frame busting!

```
<script>
if (window.top.location != window.location) {
  window.top.location = window.location
}
</script>
```

Busting Frame Busting:
a Study of Clickjacking Vulnerabilities on Popular Sites

Gustav Rydstedt, Elie Bursztein, Dan Boneh
Stanford University
{rydstedt,elie,dabo}@cs.stanford.edu

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

July 20, 2010

Abstract

Web framing attacks such as clickjacking use iframes to hijack a user's web session. The most common defense, called frame busting, prevents a site from functioning when loaded inside a frame. We study frame busting practices for the Alexa Top-500 sites and show that all can be circumvented in one way or another. Some circumventions are browser-specific while others work across browsers. We conclude with recommendations for proper frame busting.

1 Introduction

Frame busting refers to code or annotation provided by a web page intended to prevent the web page from being loaded in a sub-frame. Frame busting is the recommended defense against clickjacking [10] and is also required to secure image-based authentication such as the *Sign-in Seal* used by Yahoo. Sign-in Seal displays a user-selected image that authenticates the Yahoo! login page to the user. Without frame busting, the login page could be opened in a sub-frame so that the correct image is displayed to the user, even though the top page is not the real Yahoo login page. New advancements in clickjacking techniques [22] using drag-and-drop to extract and inject data into frames further demonstrate the importance of secure frame busting.

Figure 1: Visualization of a clickjacking attack on Twitter's account deletion page.

Figure 1 illustrates a clickjacking attack: the victim site is framed in a transparent iframe that is put on top of what appears to be a normal page. When users interact with the normal page, they are unwittingly interacting with the victim site. To defend against clickjacking attacks, the following simple frame busting code is a commonly used by web sites:

```
if (top.location != location)
  top.location = self.location;
```

Frame busting code typically consists of a *conditional statement* and a *counter-action* that navigates the top page to the correct place. As we will see, this basic code is fairly easy to bypass. We discuss far more sophisticated frame busting code (and circumvention techniques) later in the paper.

Turns out that doesn't work...

Takeaway: Don't rely on .js to bust out of frames because the framing (attacker) site can counter your .js. Instead, use HTTP headers to specify framing preferences and let the browser enforce those policies.

https://seclab.stanford.edu/websec/framebusting/

022 mln@cs.odu.edu
ss Aboukhadijeh

X-Frame-Options response header

```
$ curl -IL www.google.com
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more
info."
Date: Thu, 11 Feb 2021 00:03:43 GMT
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Transfer-Encoding: chunked
Expires: Thu, 11 Feb 2021 00:03:43 GMT
Cache-Control: private
Set-Cookie: 1P_JAR=2021-02-11-00; expires=Sat, 13-Mar-2021
00:03:43 GMT; path=/; domain=.google.com; Secure
Set-Cookie:
NID=209=LV-khsyfMtVPfMrFmpjNu373gMzk4jg4TsyvZ346nnZAu9zLyz1vVSFOc
HfTWOeVCUrEXI-XM_9XaeZ-1O8yWZKxCwxQQEBrdIgrV4ah1XPhJwYS9nQjn3B1ut
iTPCpGYuF3b8pFvQq57Y2LtHxVrzsFSz2IvGgO1e5w6Tt7Tdw; expires=Fri,
13-Aug-2021 00:03:43 GMT; path=/; domain=.google.com; HttpOnly
```

Deprecating the "X-" Prefix and Similar Constructs in Application Protocols

Unclear specs are bad...

There are two possible directives for `X-Frame-Options`:

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
```

not present = “anyone can frame me”

Directives

If you specify `DENY`, not only will attempts to load the page in a frame fail when loaded from other sites, attempts to do so will fail when loaded from the same site. On the other hand, if you specify `SAMEORIGIN`, you can still use the page in a frame as long as the site including it in a frame is the same as the one serving the page.

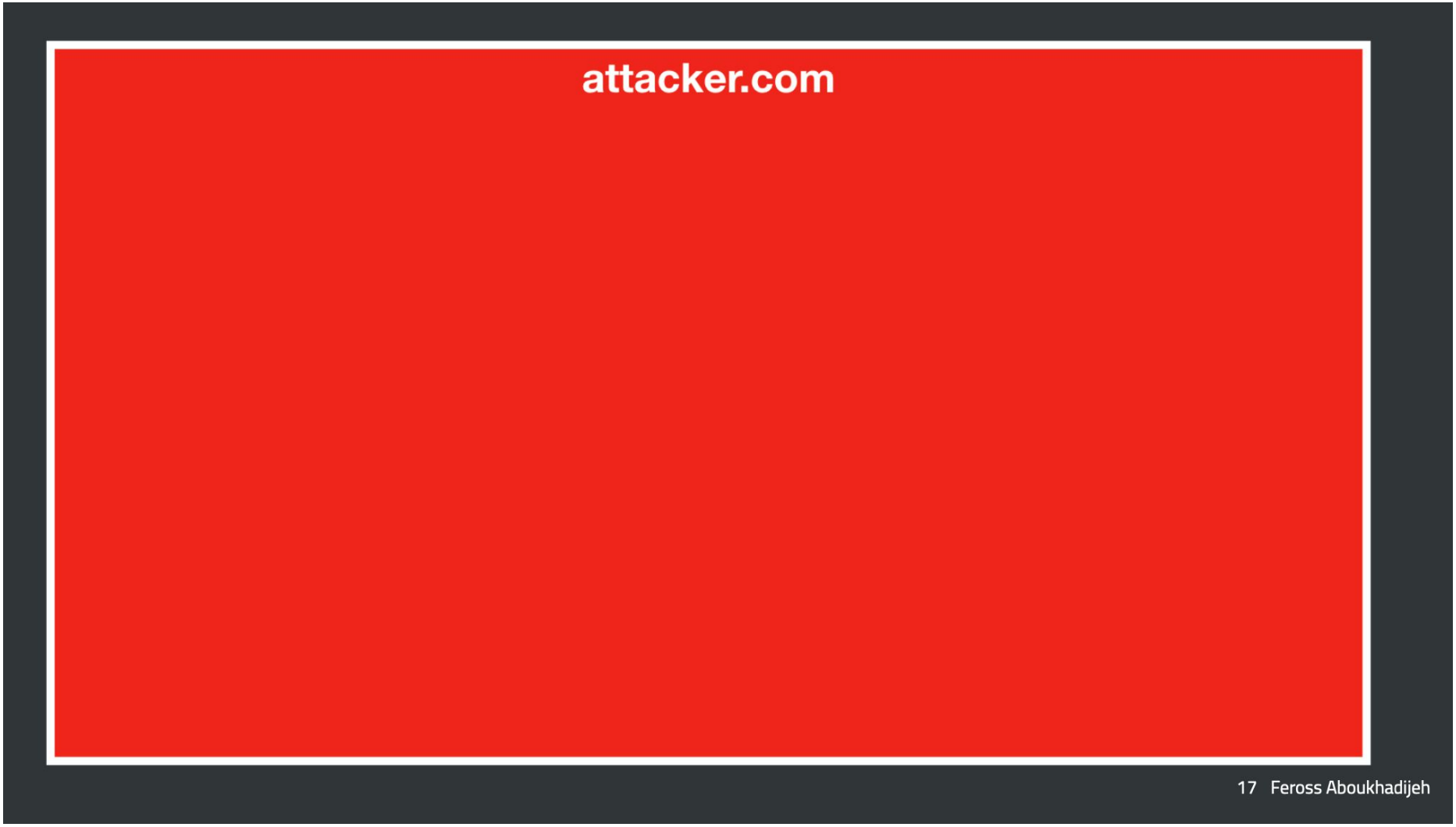
`DENY`

The page cannot be displayed in a frame, regardless of the site attempting to do so.

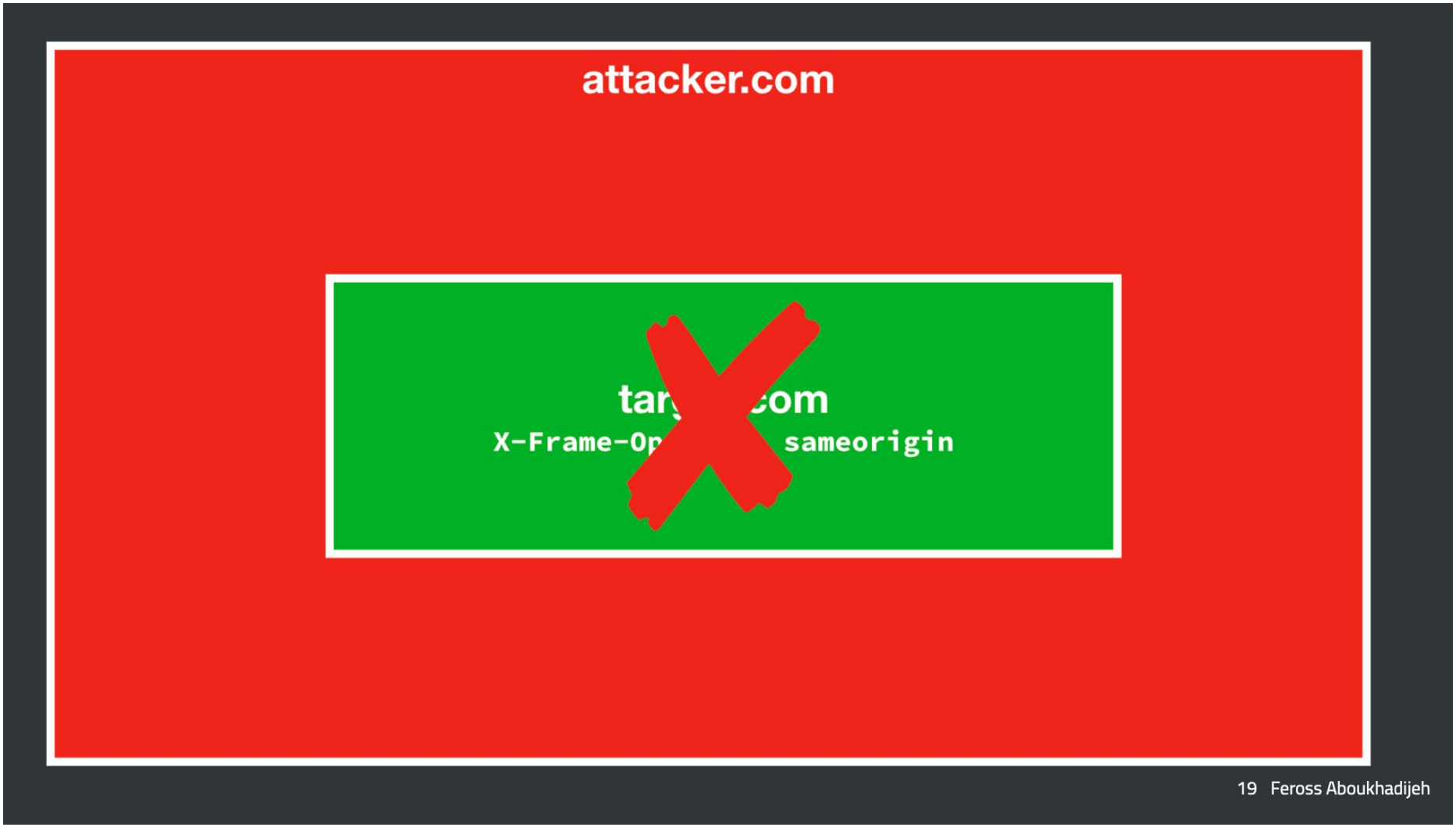
`SAMEORIGIN`

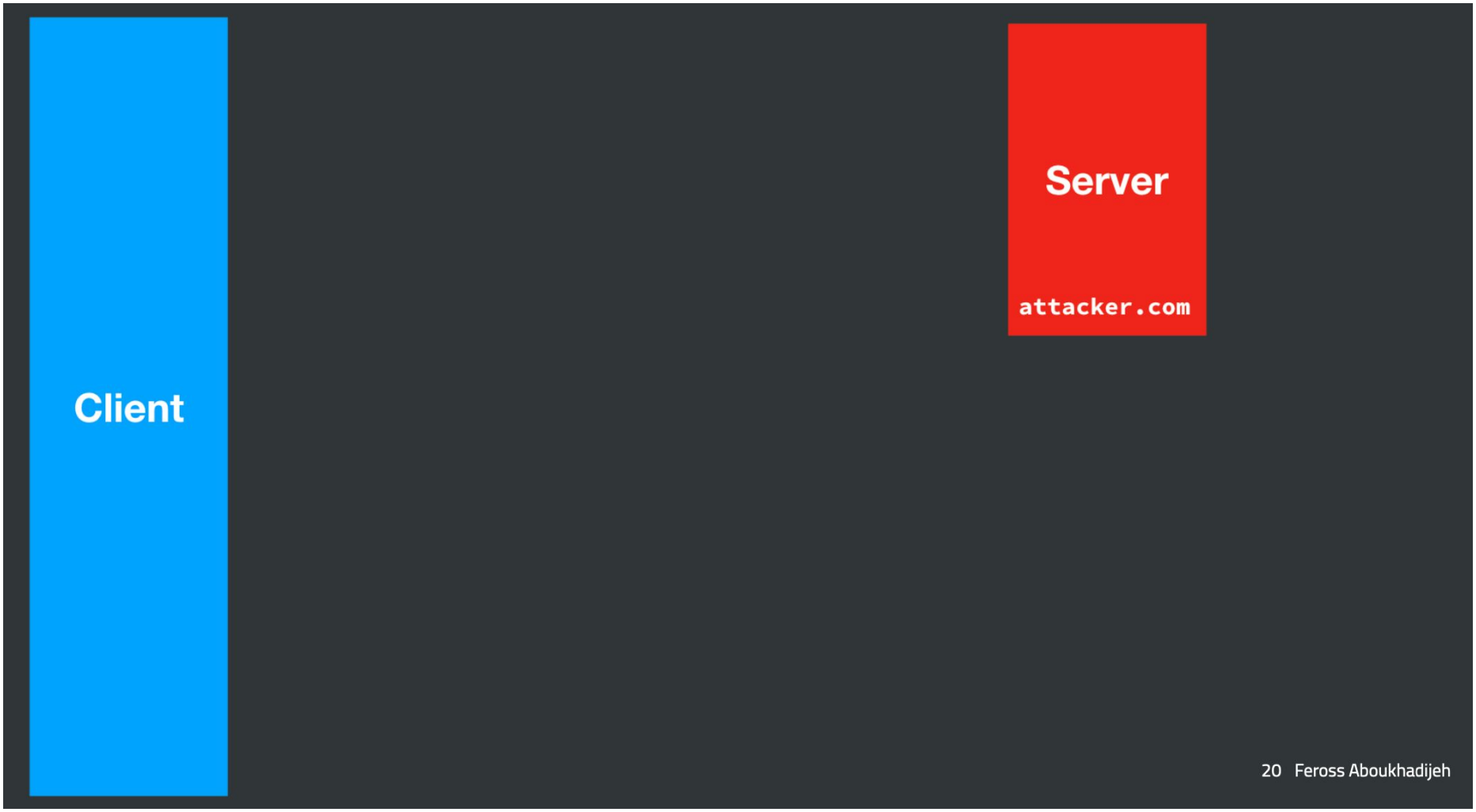
The page can only be displayed in a frame on the same origin as the page itself. The spec leaves it up to browser vendors to decide whether this option applies to the top level, the parent, or the whole chain, although it is argued that the option is not very useful unless all ancestors are also in the same origin (see bug 725490). Also see Browser compatibility for support details.

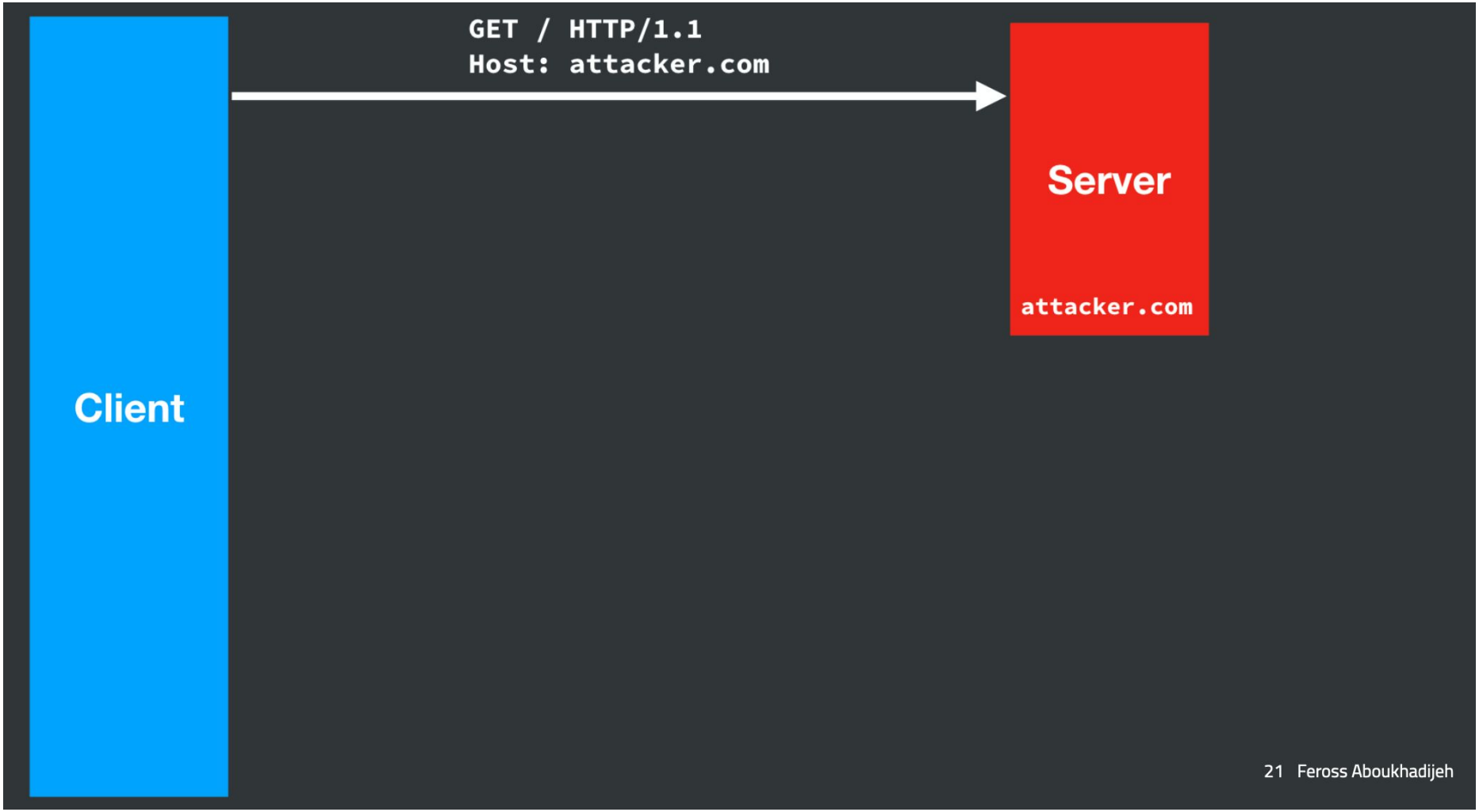
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options
https://datatracker.ietf.org/doc/html/rfc7034

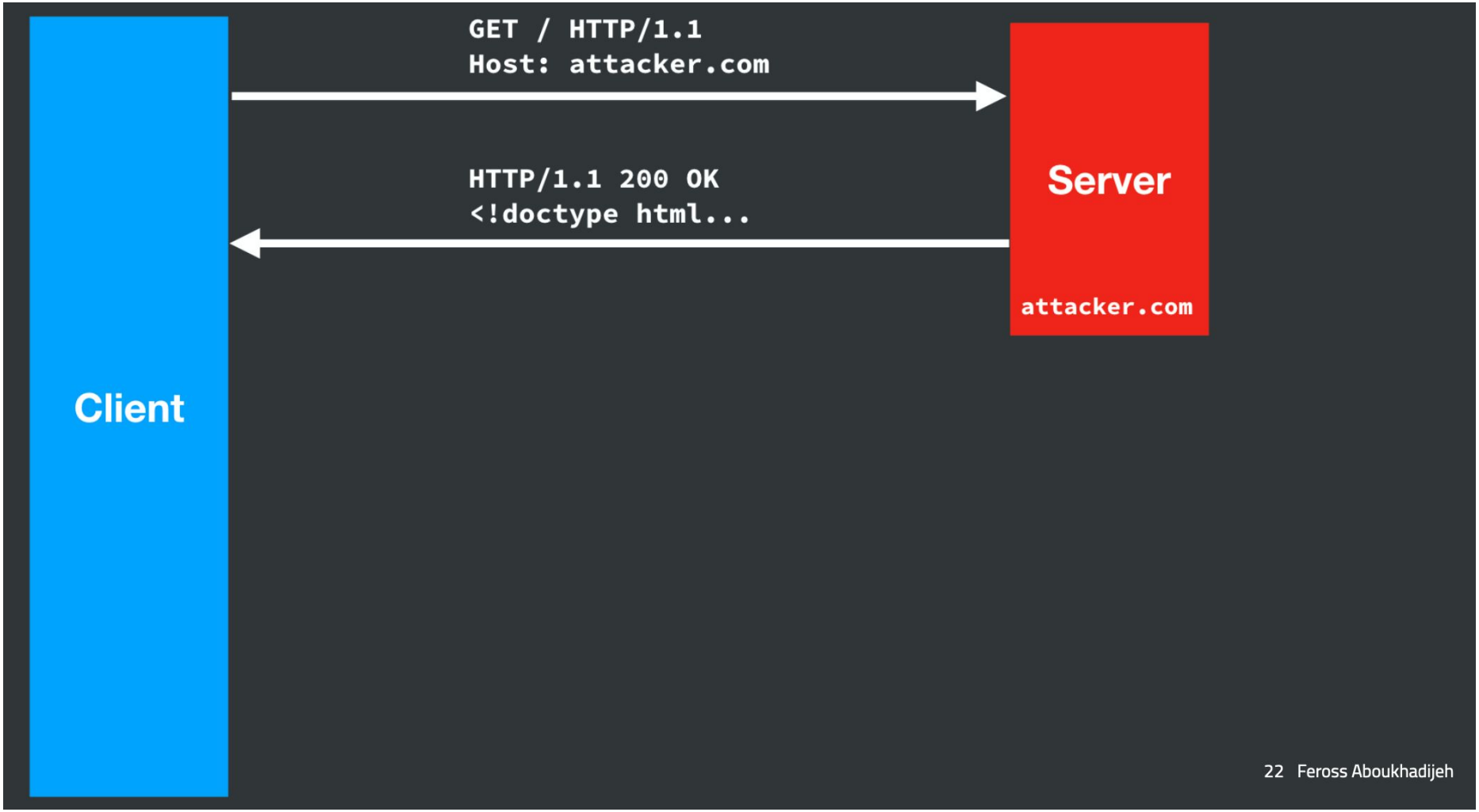

attacker.com
17 Feross Aboukhadijeh

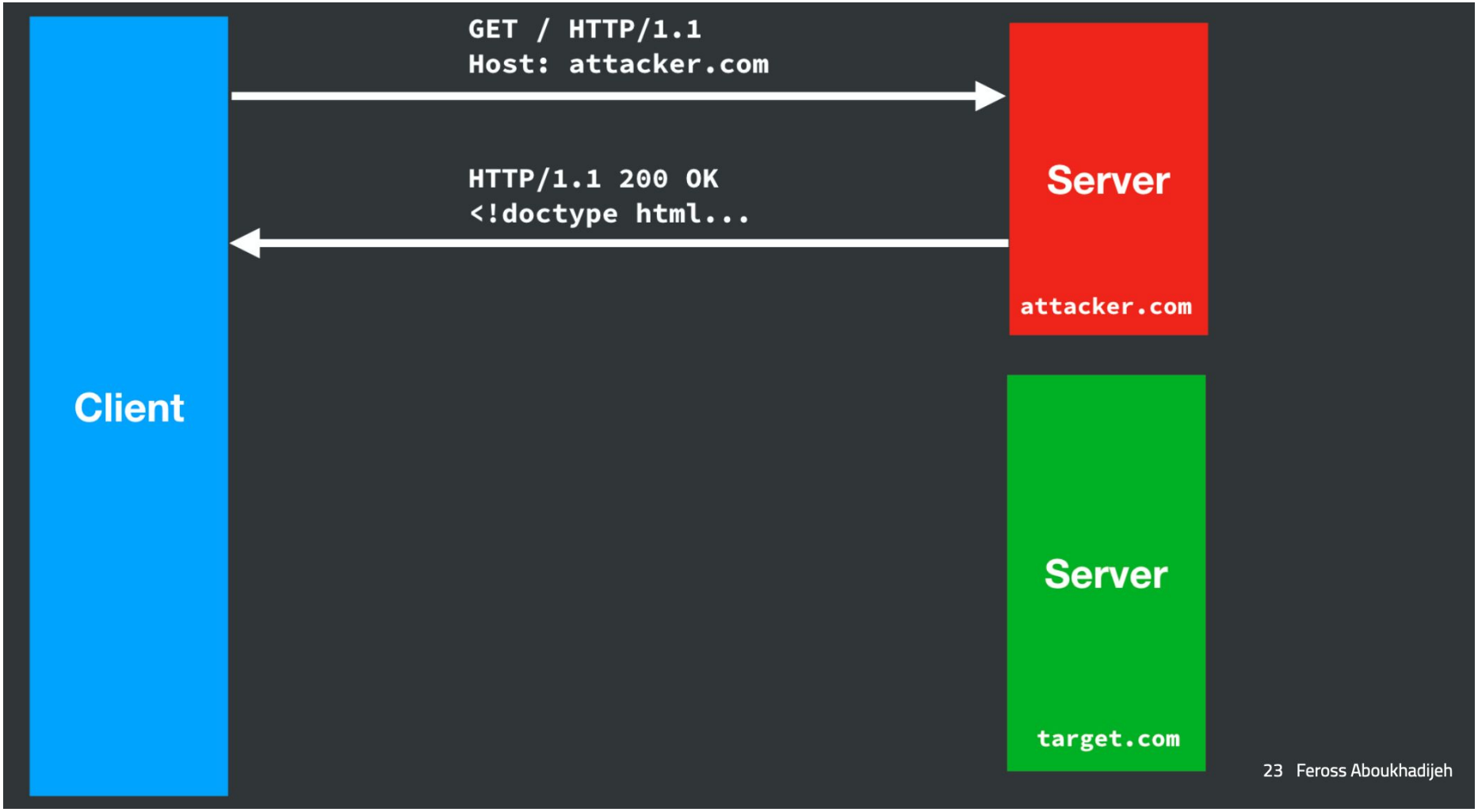

attacker.com
target.com
X-Frame-Options: sameorigin
18 Feross Aboukhadijeh

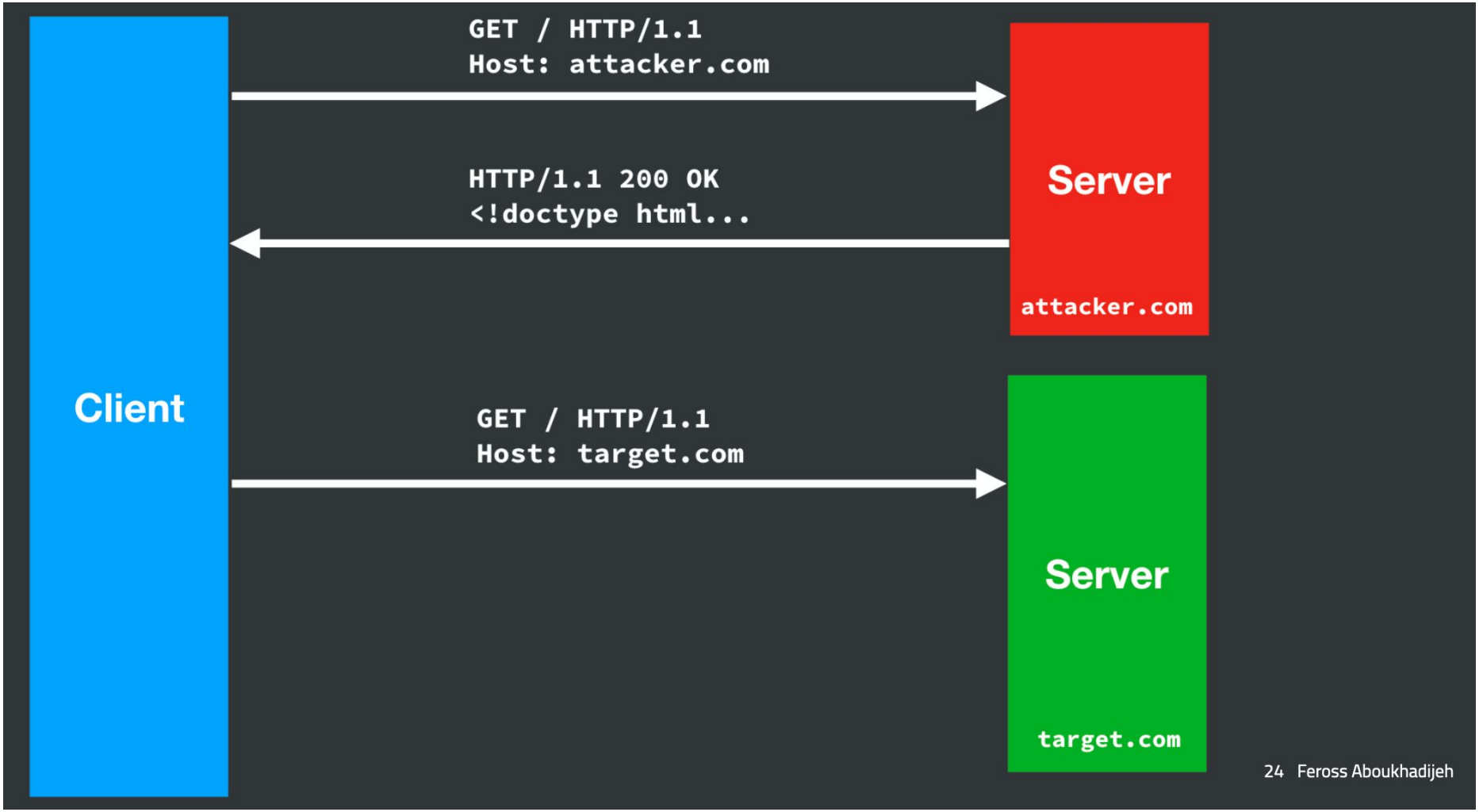

attacker.com
X-Frame-Op sameorigin
19 Feross Aboukhadijeh

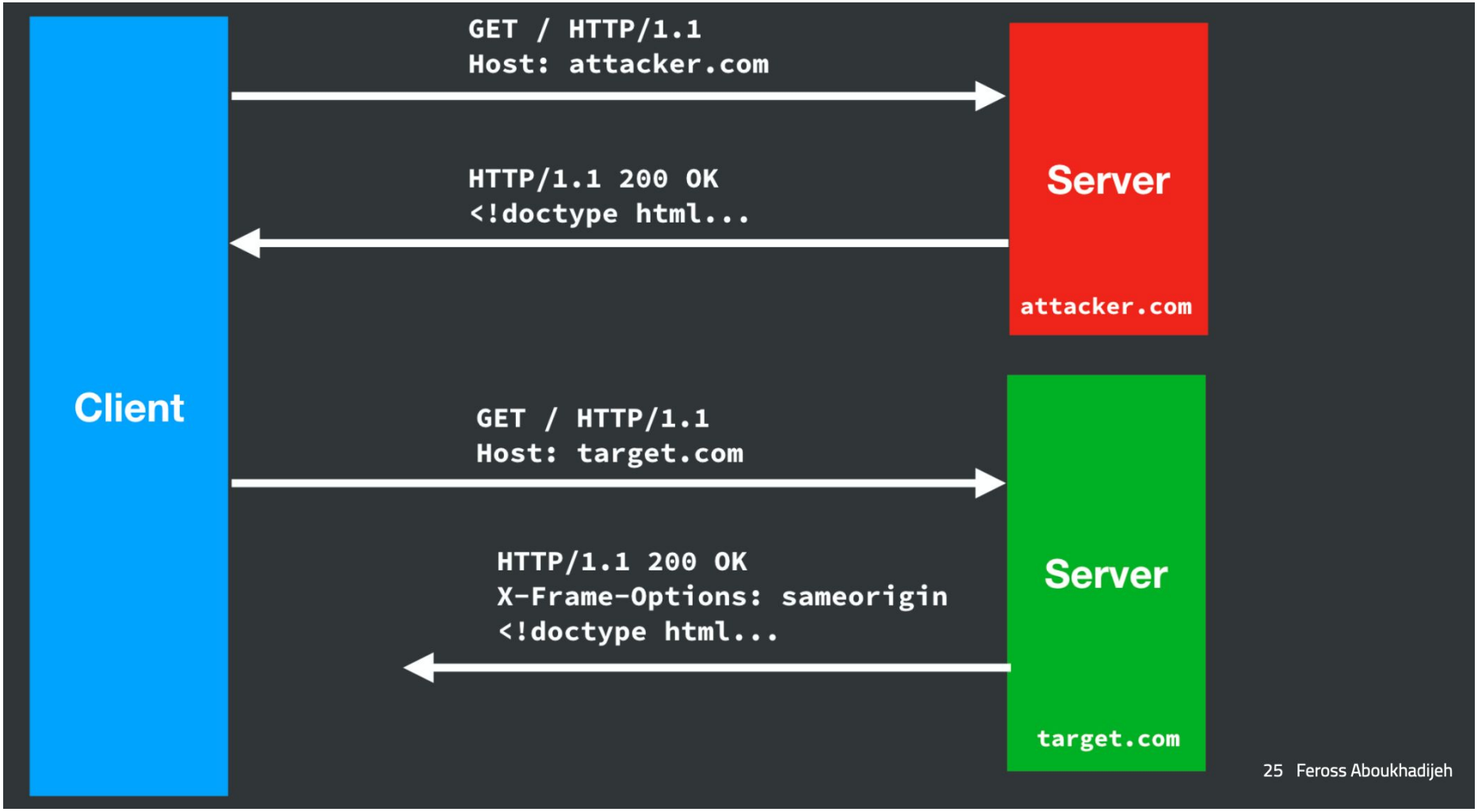

Client
Server
attacker.com
20 Feross Aboukhadijeh

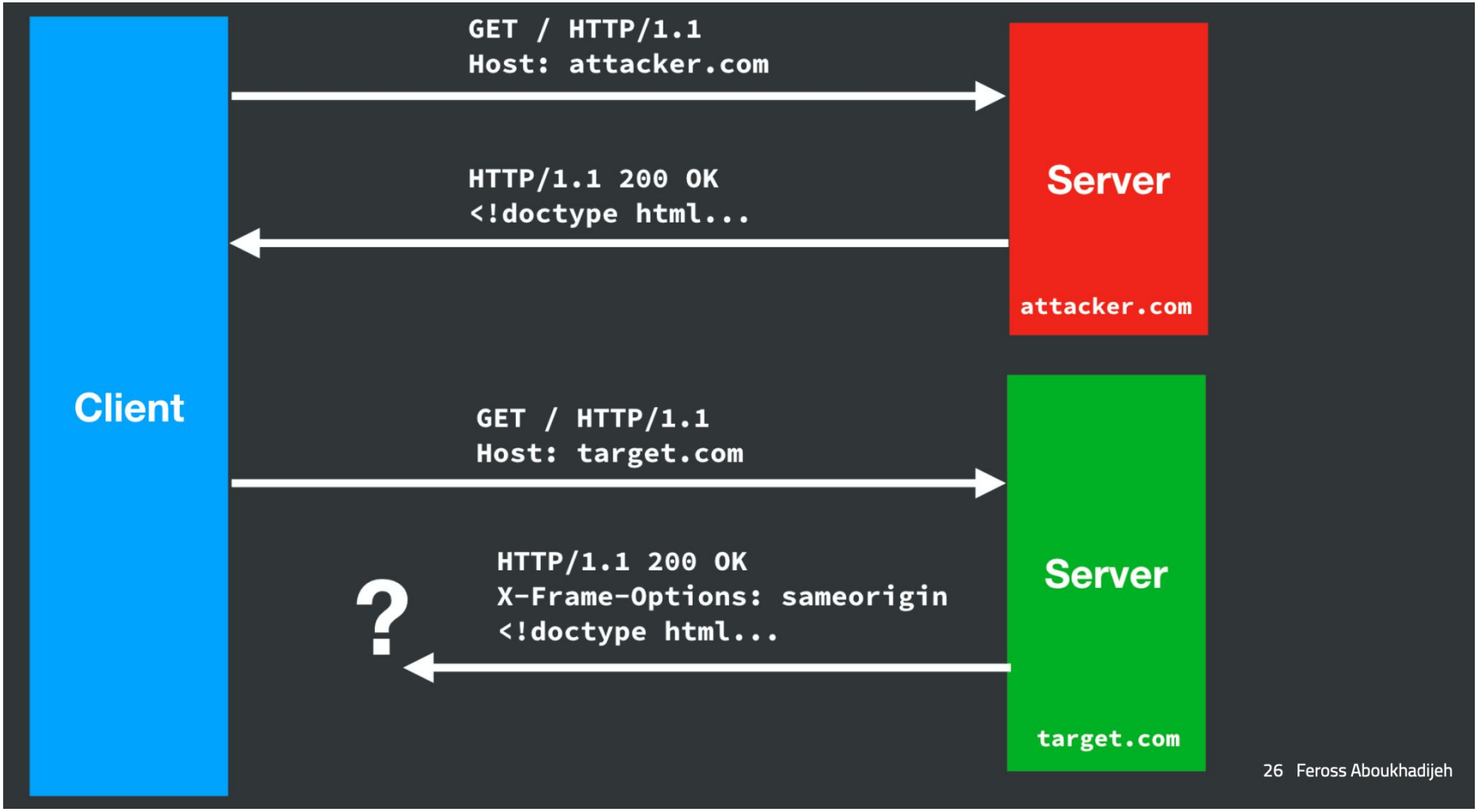

GET / HTTP/1.1
Host: attacker.com
Client
Server
attacker.com
21 Feross Aboukhadijeh

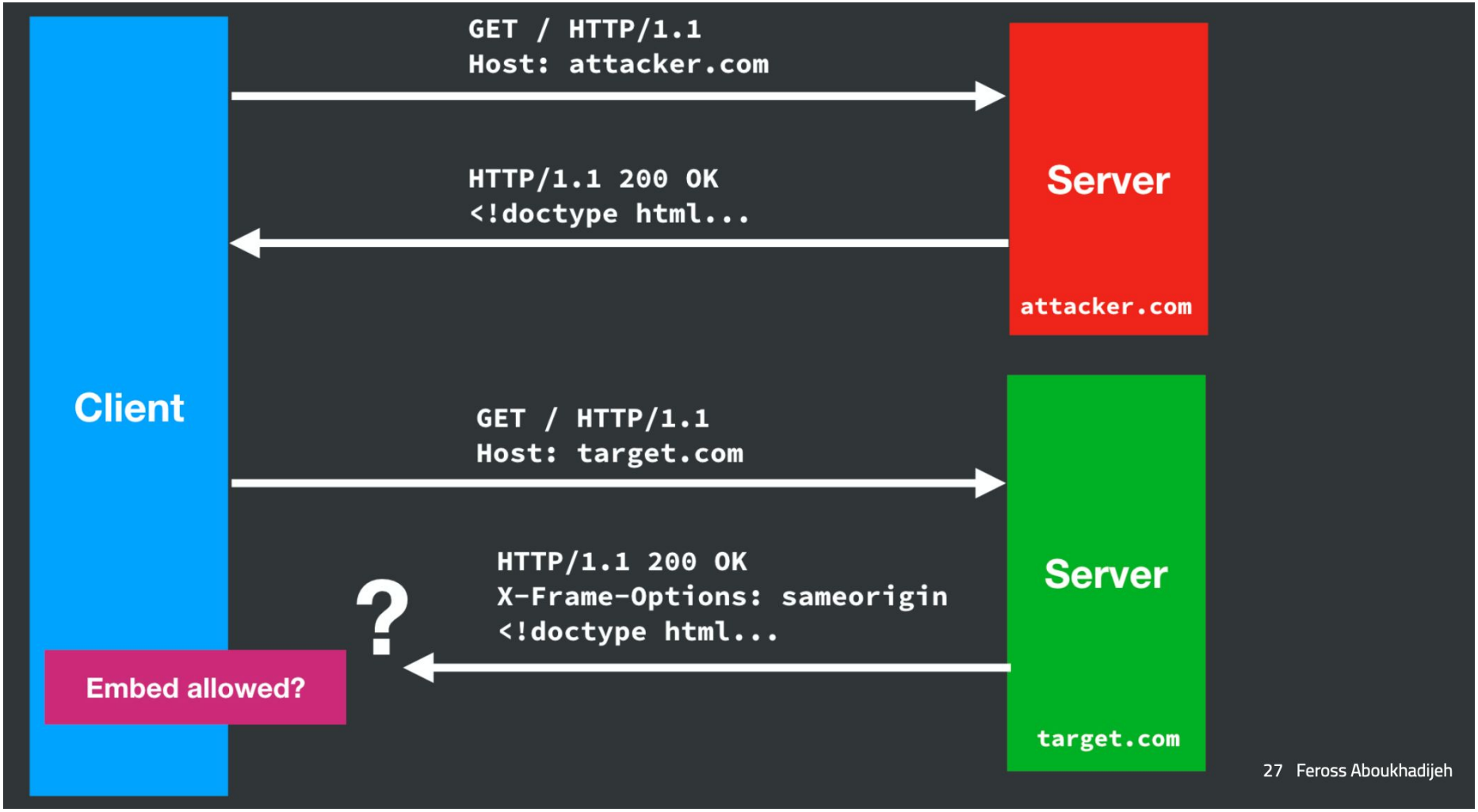

GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Client
Server
attacker.com
22 Feross Aboukhadijeh

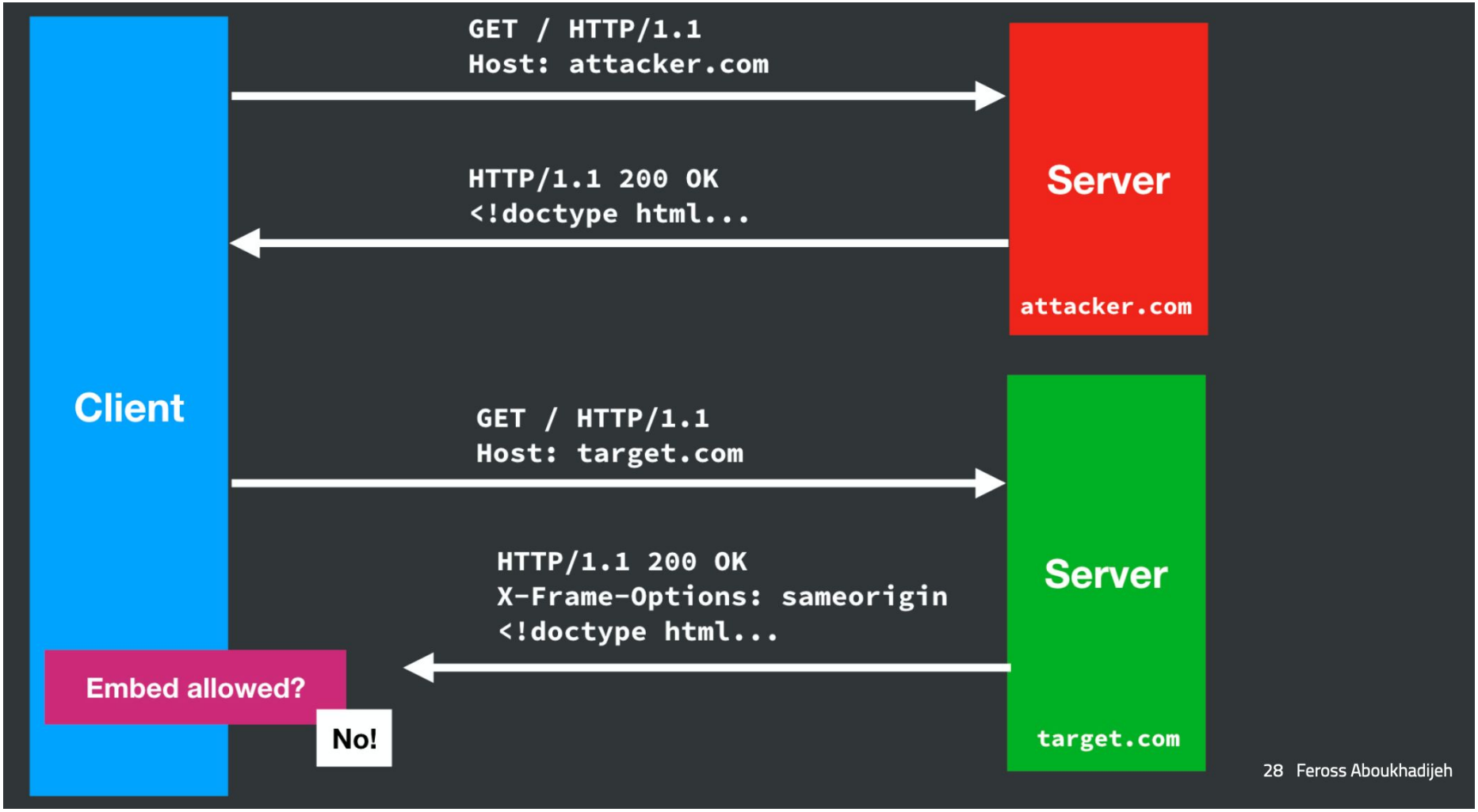

GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Client
Server
attacker.com
Server
target.com
23 Feross Aboukhadijeh

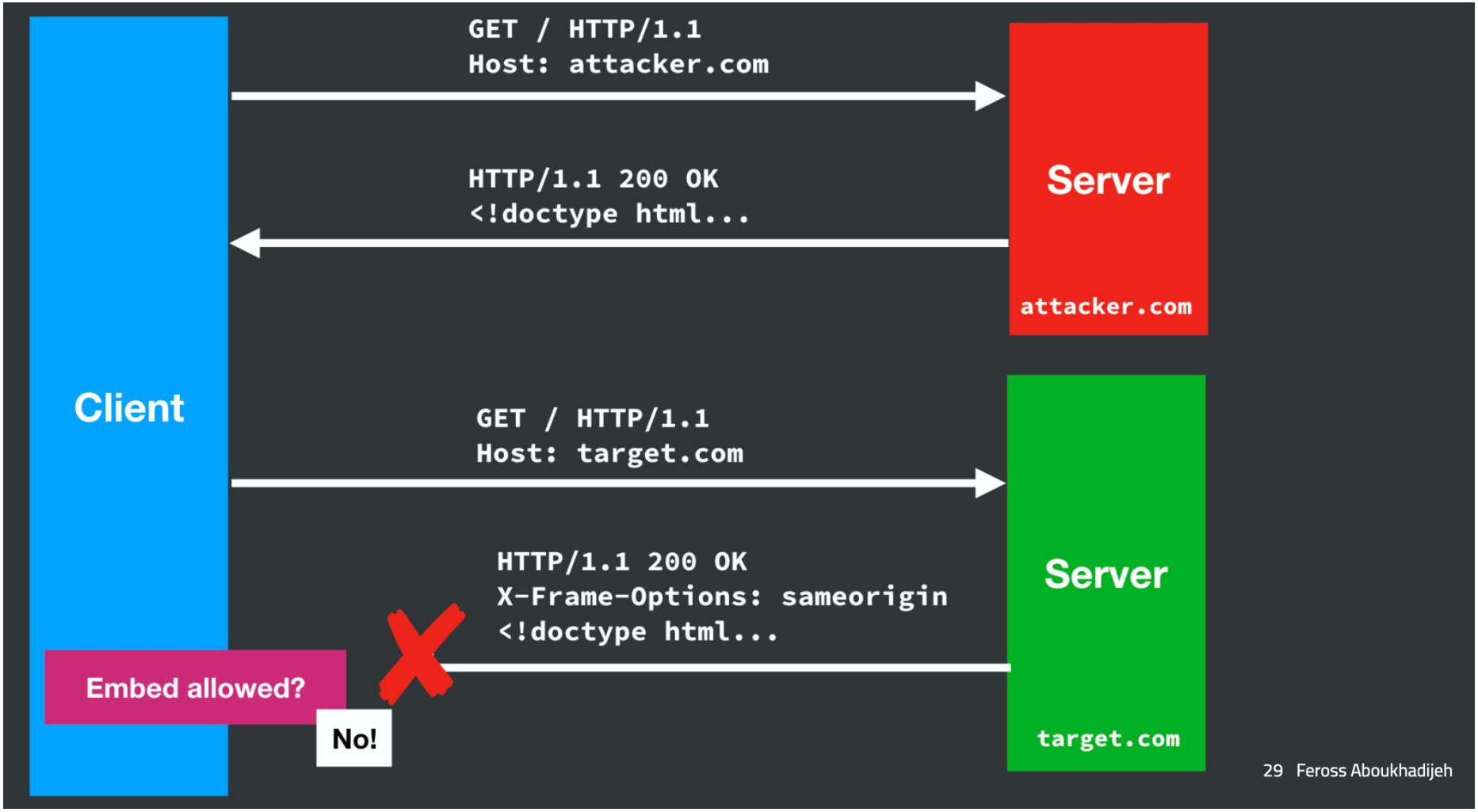

Client
GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
GET / HTTP/1.1
Host: target.com
Server
target.com
24 Feross Aboukhadijeh


Client
GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
GET / HTTP/1.1
Host: target.com
HTTP/1.1 200 OK
X-Frame-Options: sameorigin
<!doctype html...
Server
target.com
25 Feross Aboukhadijeh


Client
GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
GET / HTTP/1.1
Host: target.com
HTTP/1.1 200 OK
X-Frame-Options: sameorigin
<!doctype html...
?
Server
target.com
26 Feross Aboukhadijeh


GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
Client
GET / HTTP/1.1
Host: target.com
HTTP/1.1 200 OK
X-Frame-Options: sameorigin
<!doctype html...
?
Server
target.com
Embed allowed?
27 Feross Aboukhadijeh

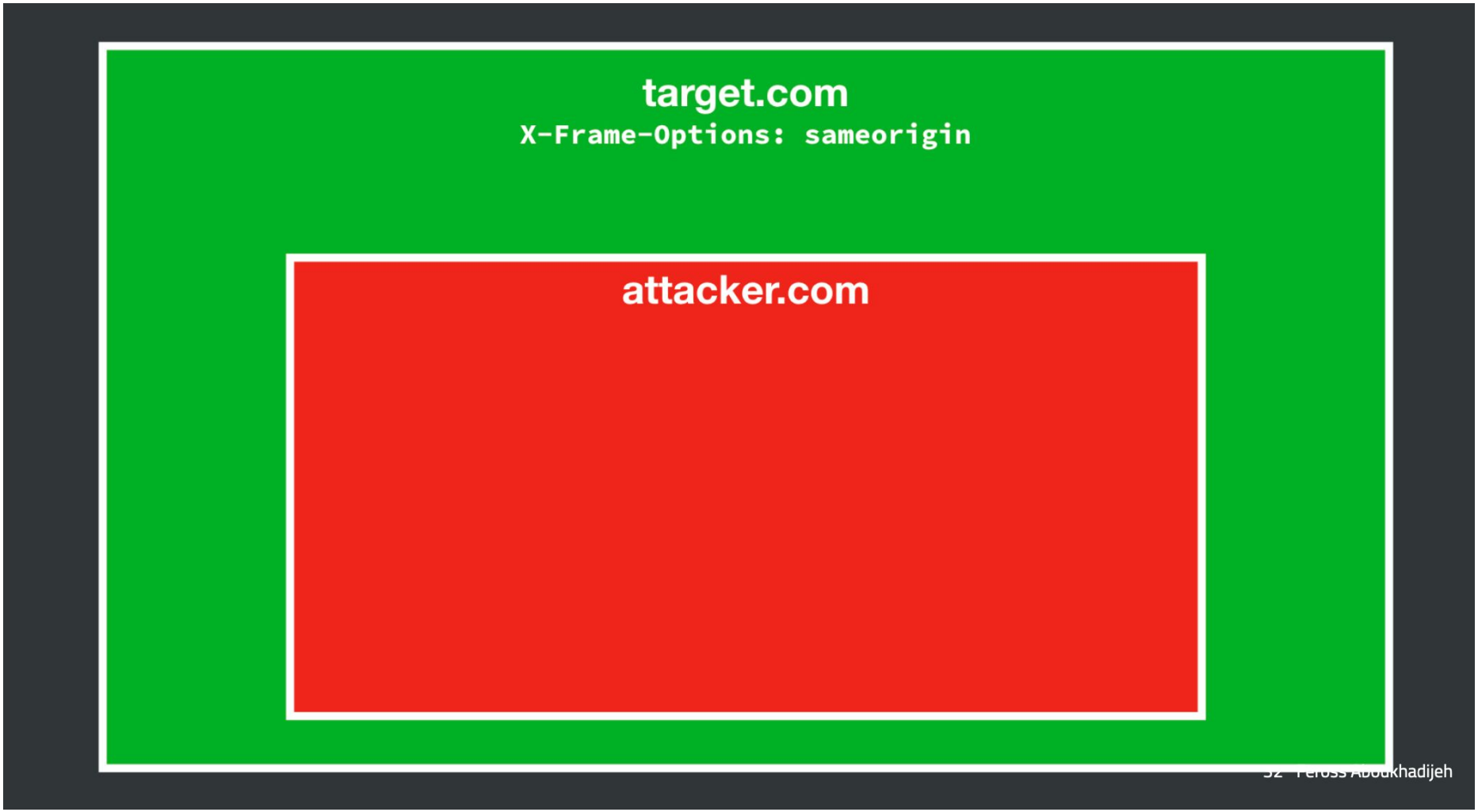

Client
GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
GET / HTTP/1.1
Host: target.com
HTTP/1.1 200 OK
X-Frame-Options: sameorigin
<!doctype html...
Server
target.com
Embed allowed?
No!
28 Feross Aboukhadijeh

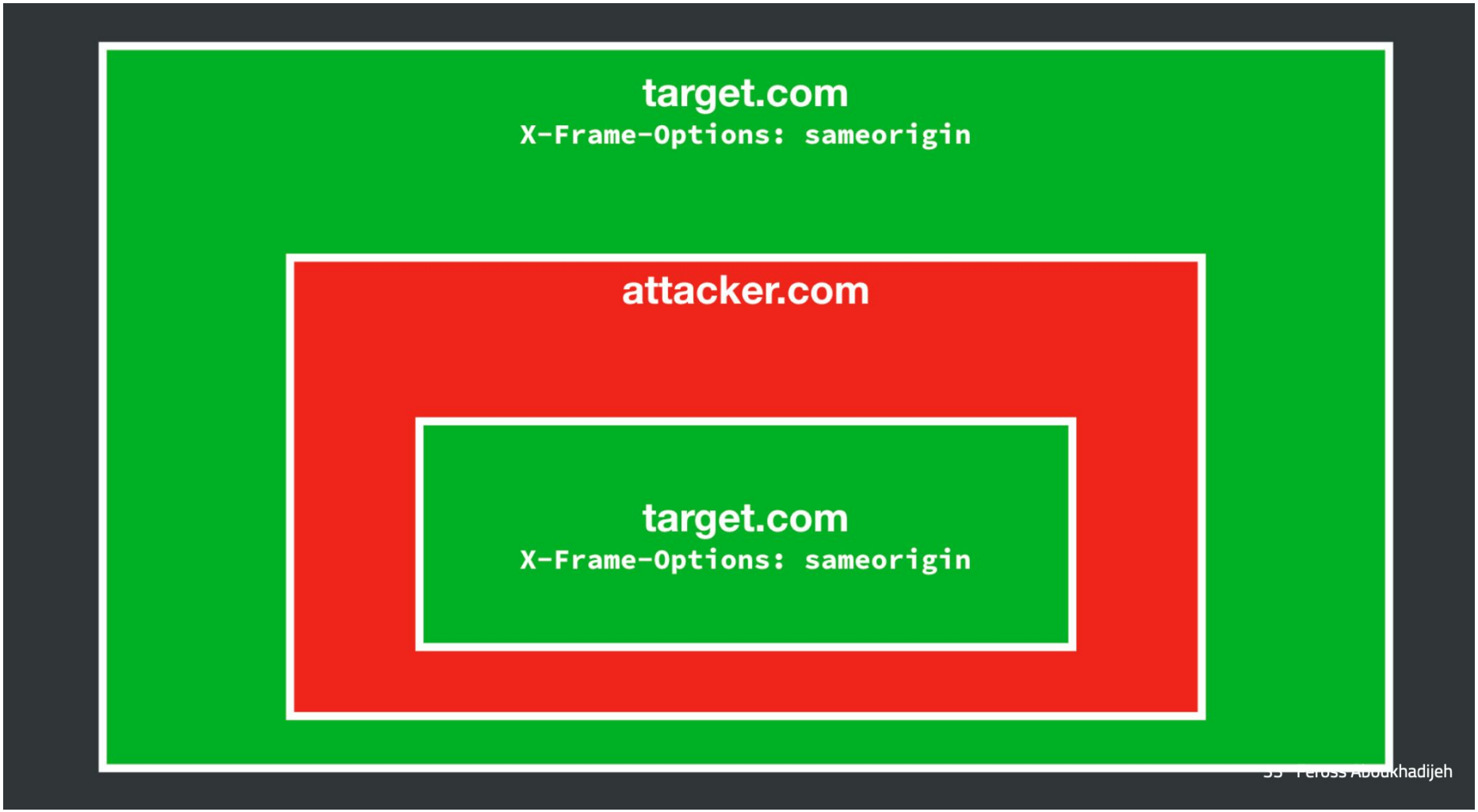

GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
Client
GET / HTTP/1.1
Host: target.com
HTTP/1.1 200 OK
X-Frame-Options: sameorigin
<!doctype html...
Server
target.com
Embed allowed?
No!
29 Feross Aboukhadijeh

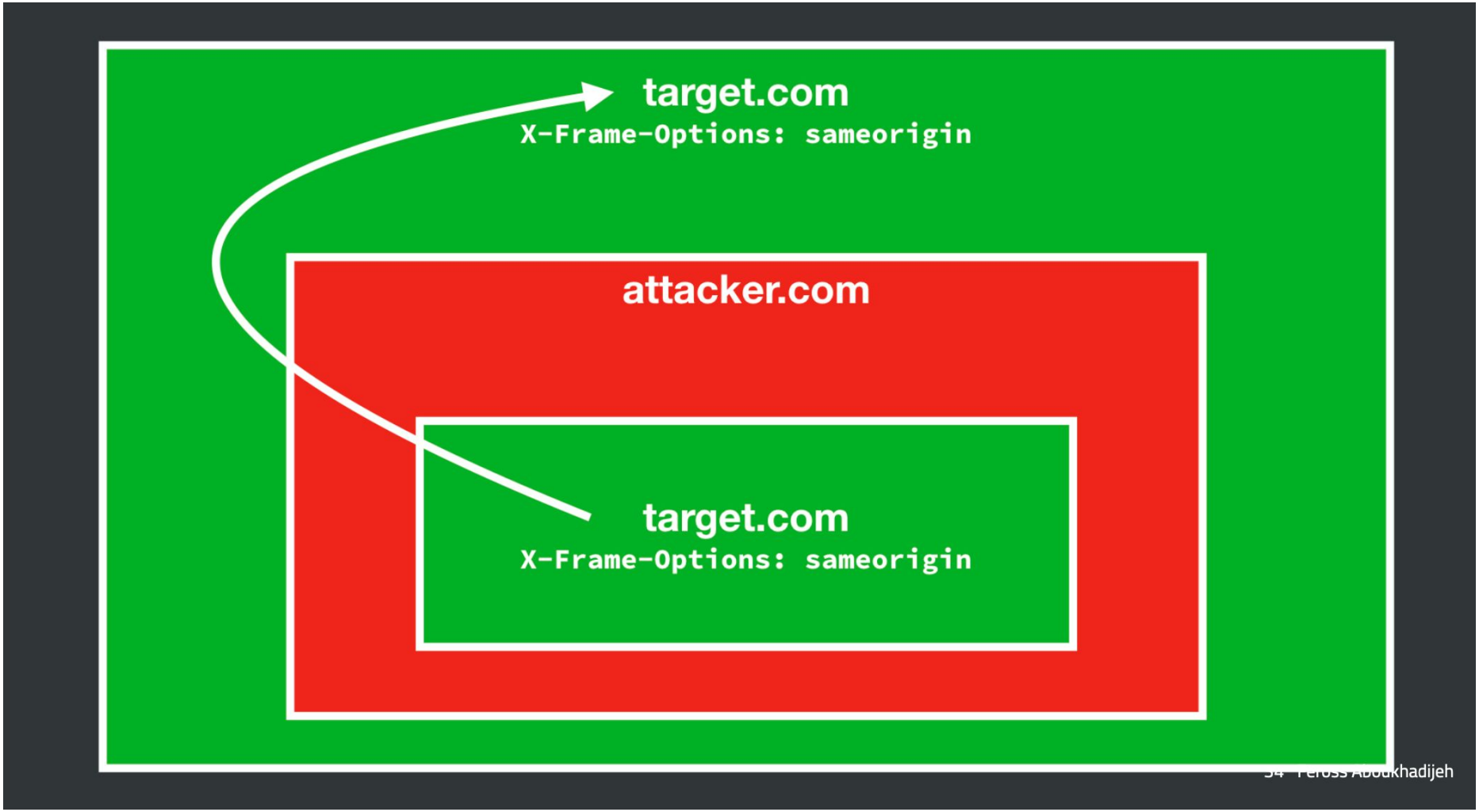
It's turtles all the way down...

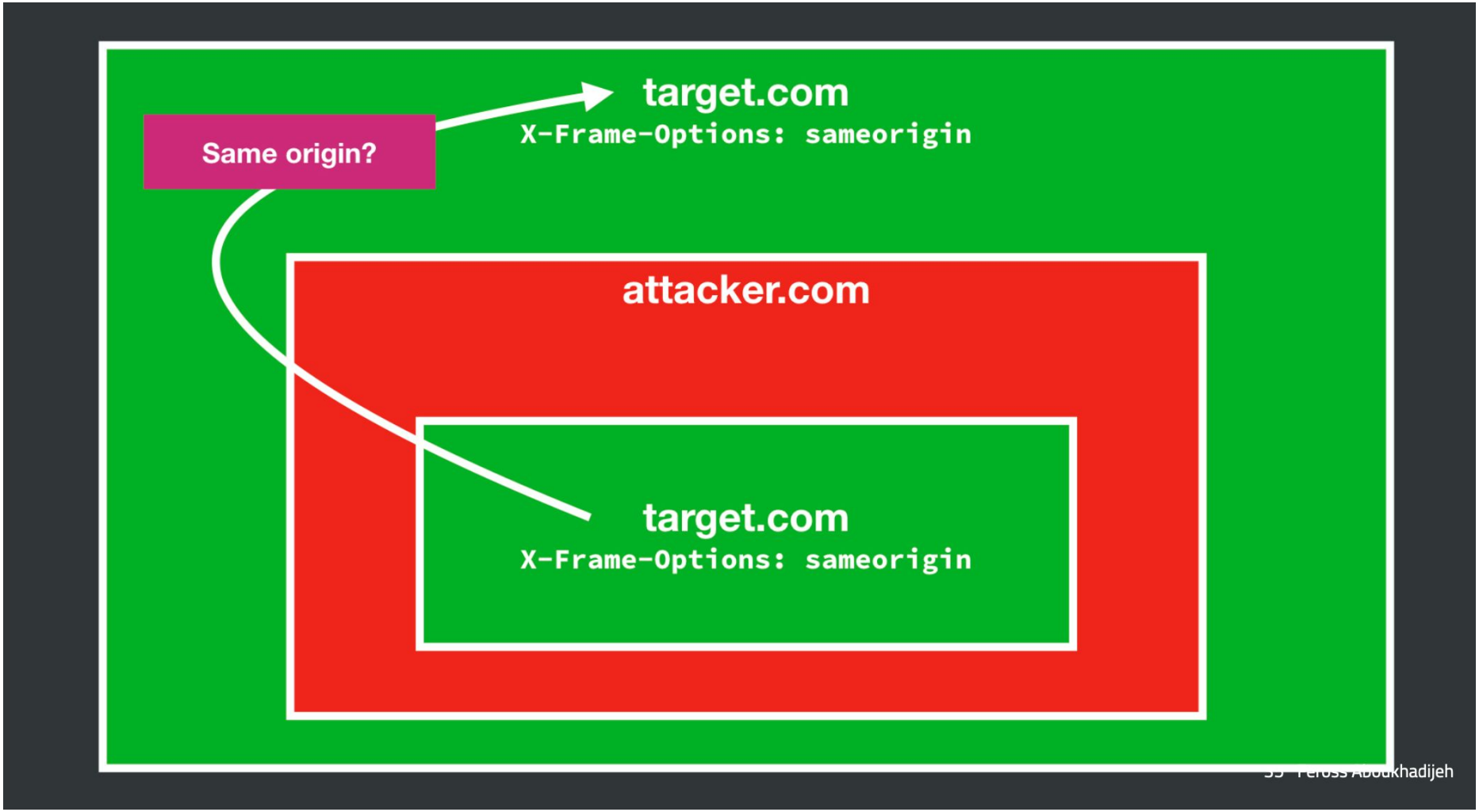
- Until recently, browsers performed a check only against top-level window
- Thus, attackers could set up a framing chain which would be allowed:
  - `target.com` embeds `attacker.com` embeds `target.com`
  - this could happen with evil or hacked ad sites!

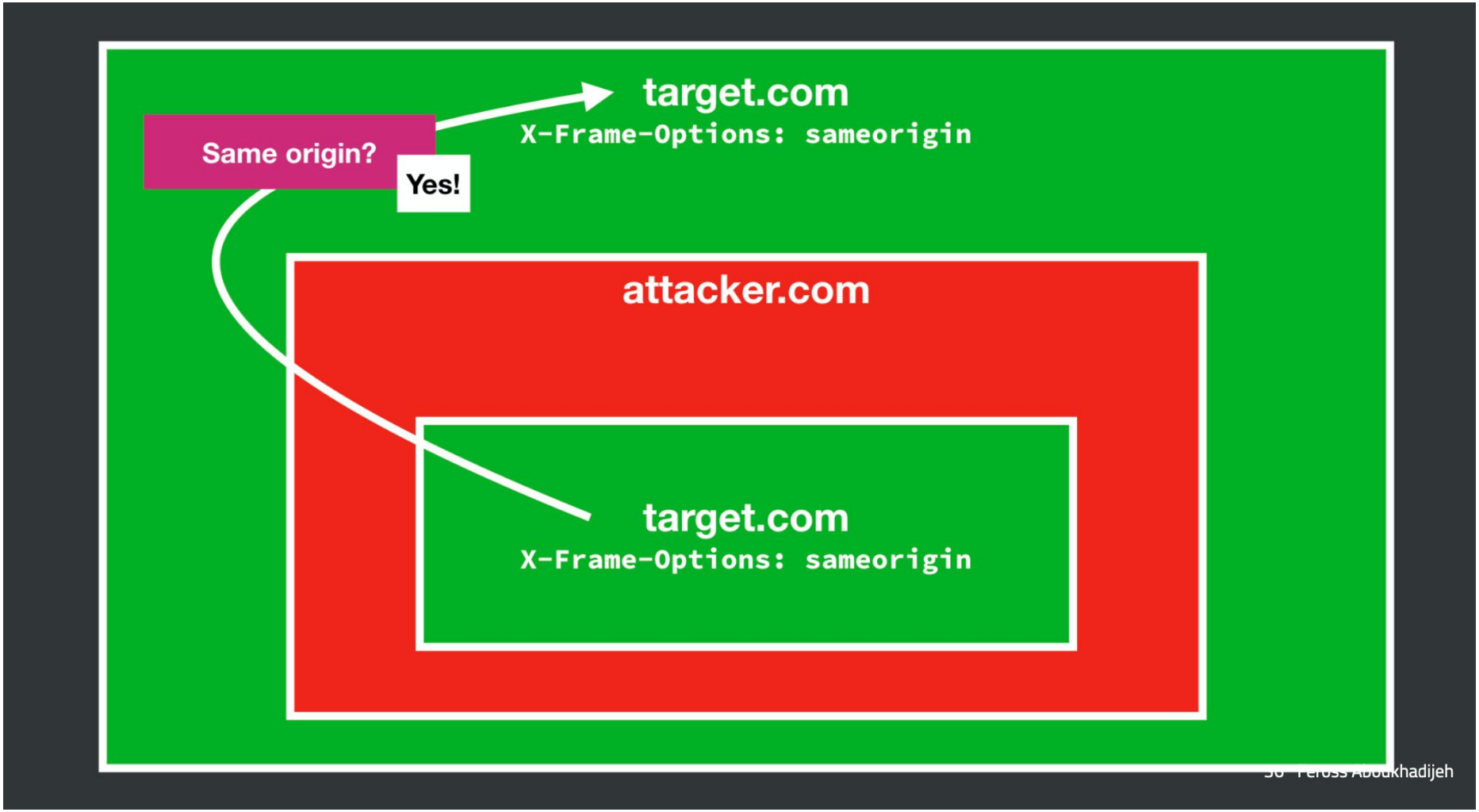

target.com
X-Frame-Options: sameorigin

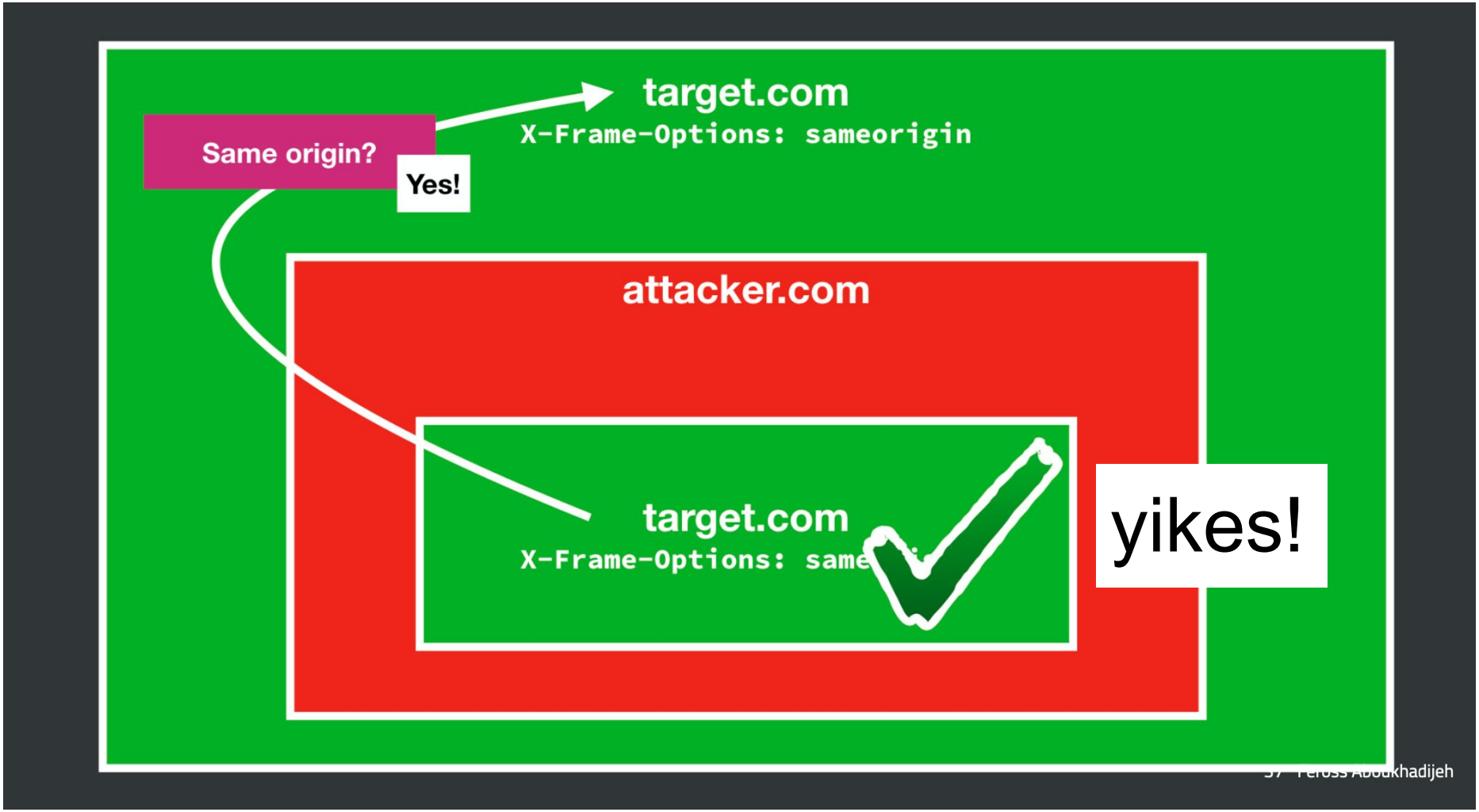

target.com
X-Frame-Options: sameorigin
attacker.com

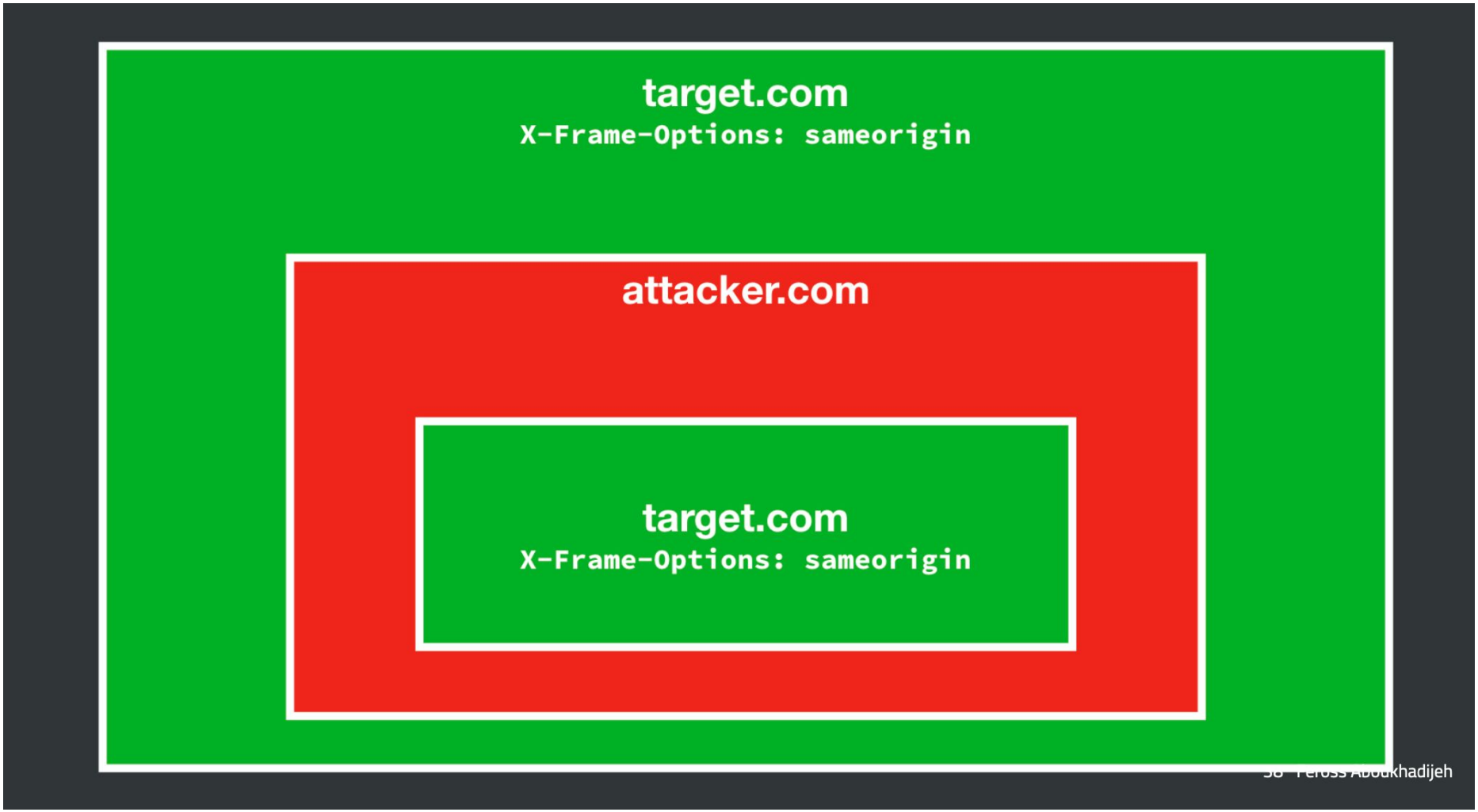

target.com
X-Frame-Options: sameorigin
attacker.com
target.com
X-Frame-Options: sameorigin

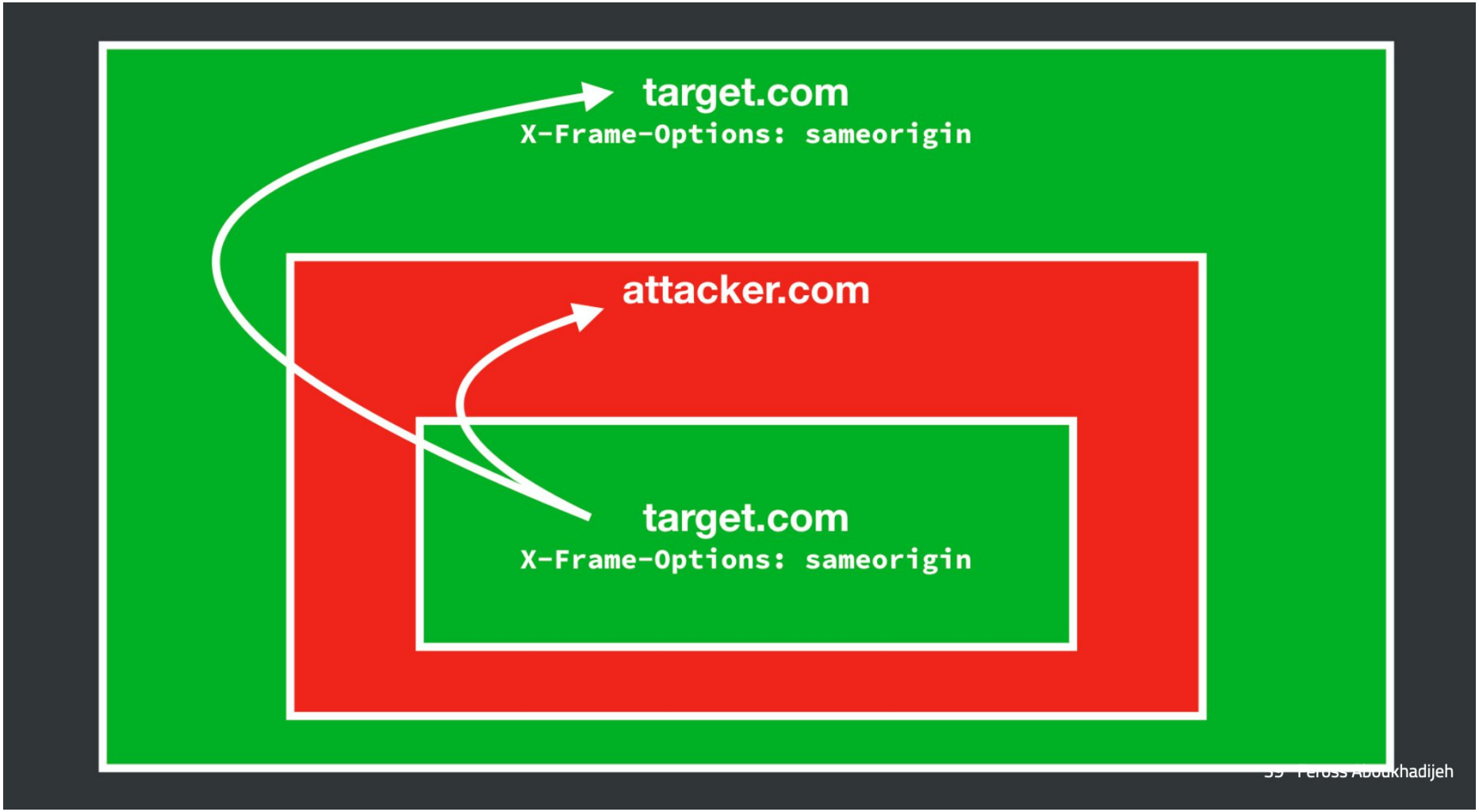

target.com
X-Frame-Options: sameorigin
attacker.com
target.com
X-Frame-Options: sameorigin

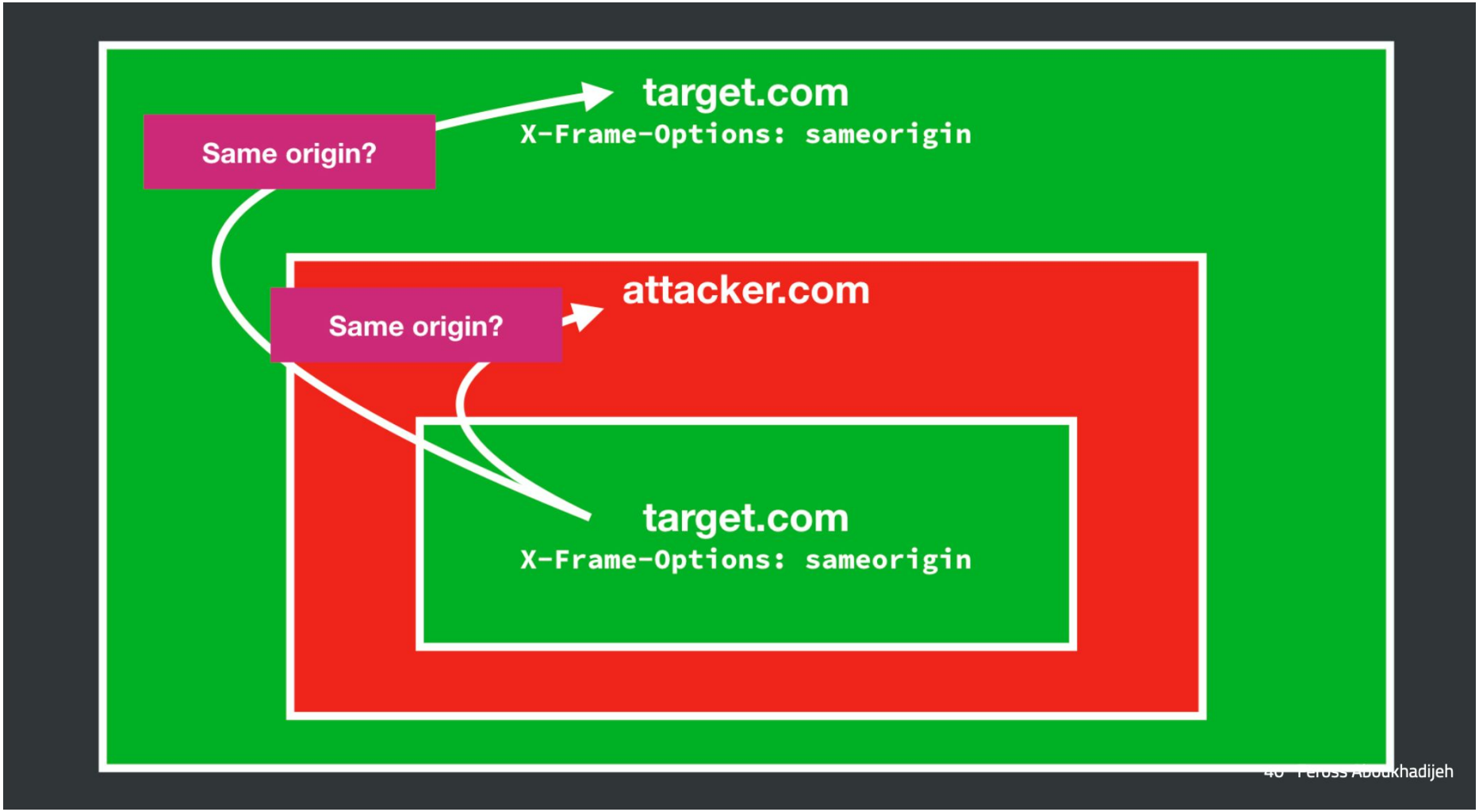

target.com
X-Frame-Options: sameorigin
Same origin?
attacker.com
target.com
X-Frame-Options: sameorigin

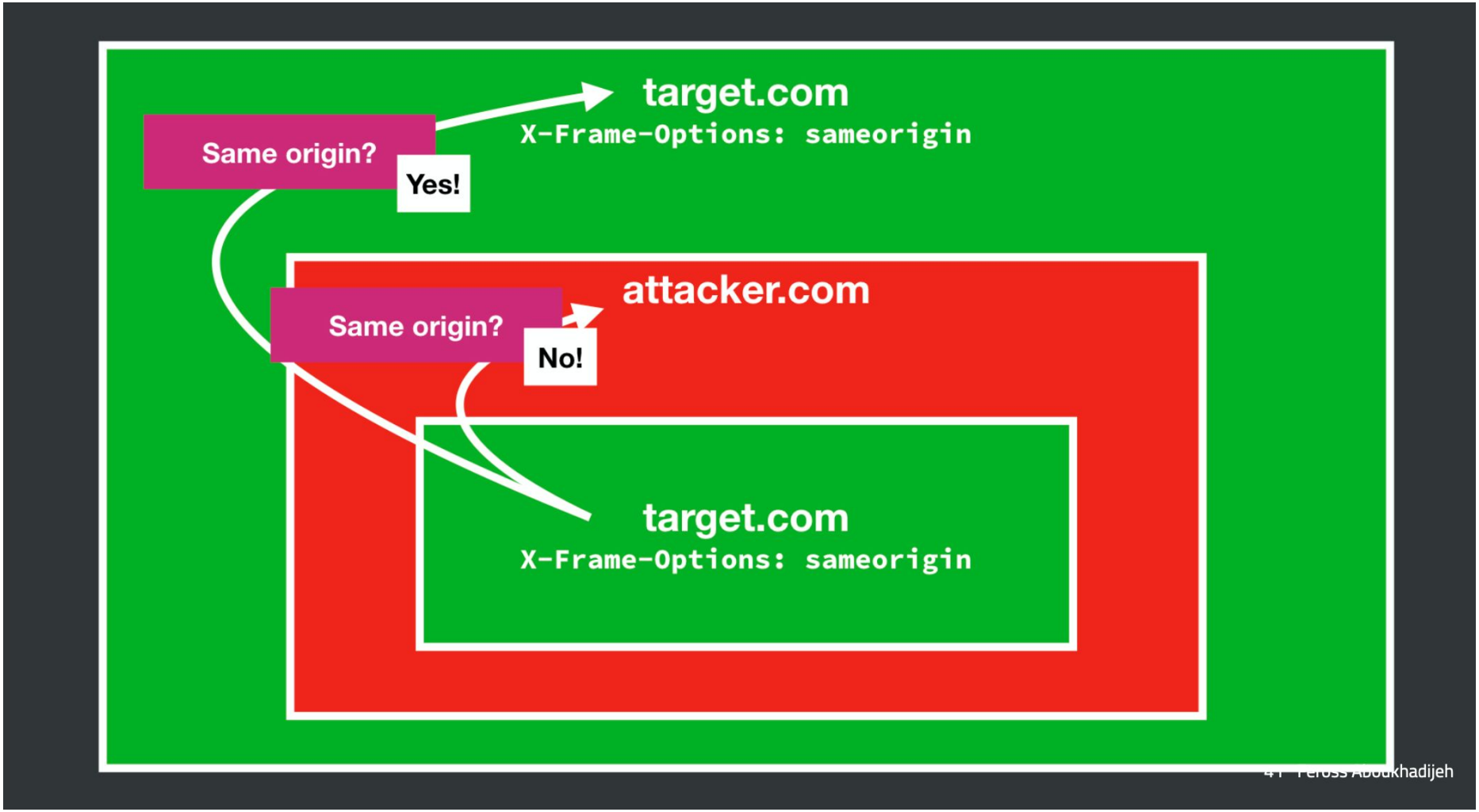

target.com
X-Frame-Options: sameorigin
Same origin?
Yes!
attacker.com
target.com
X-Frame-Options: sameorigin

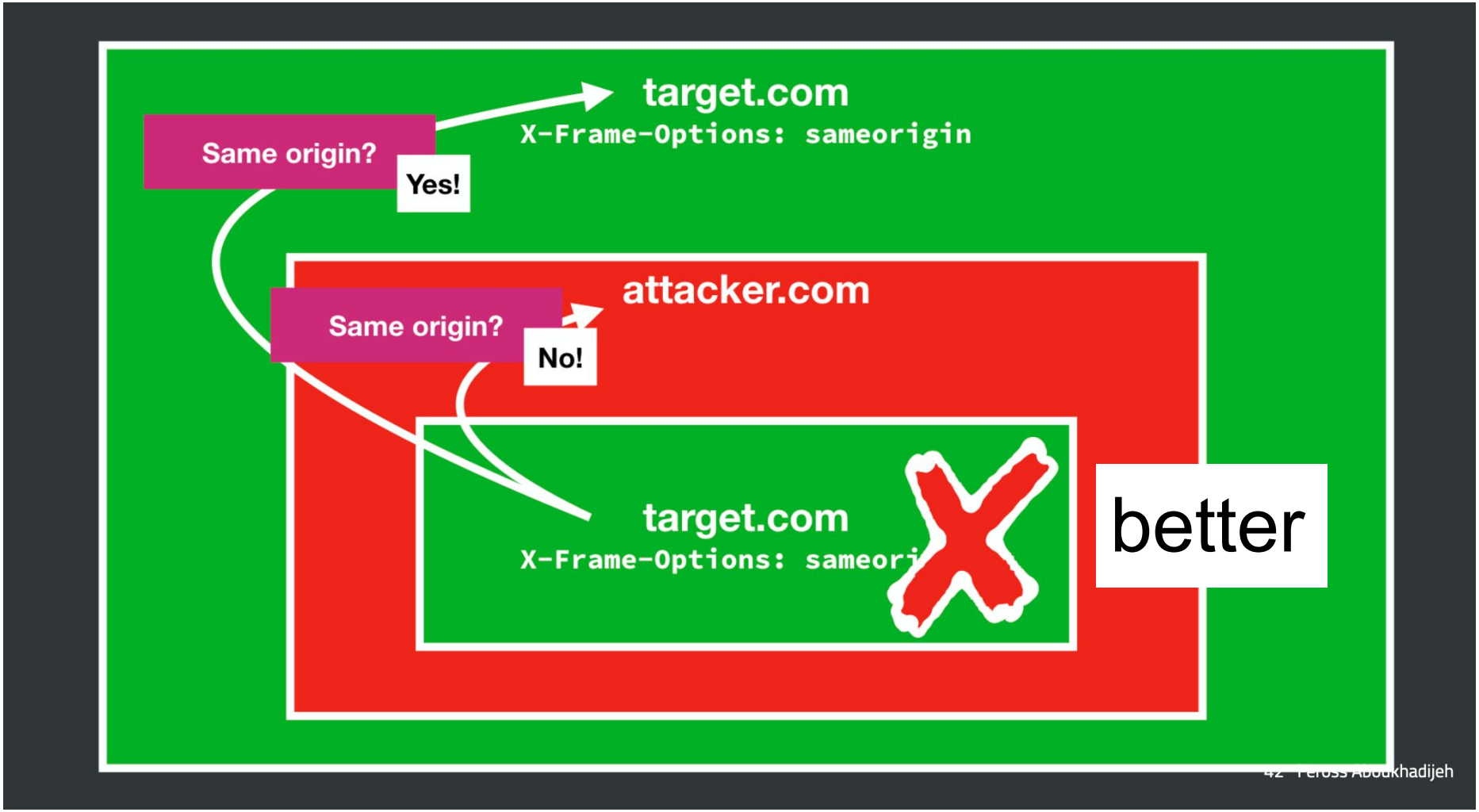

target.com
X-Frame-Options: sameorigin
Same origin?
Yes!
attacker.com
target.com
yikes!


target.com
X-Frame-Options: sameorigin
attacker.com
target.com
X-Frame-Options: sameorigin


target.com
X-Frame-Options: sameorigin
attacker.com
target.com
X-Frame-Options: sameorigin


target.com
X-Frame-Options: sameorigin
Same origin?
attacker.com
Same origin?
target.com
X-Frame-Options: sameorigin


target.com
X-Frame-Options: sameorigin
Same origin?
Yes!
attacker.com
Same origin?
No!
target.com
X-Frame-Options: sameorigin


target.com
X-Frame-Options: sameorigin
Same origin?
Yes!
attacker.com
Same origin?
No!
target.com
X-Frame-Options: sameori
better

Replacement for X-Frame-Options: Content-Security-Policy: frame-ancestors

Examples

```
Content-Security-Policy: frame-ancestors 'none';

Content-Security-Policy: frame-ancestors 'self' https://www.example.org;
```

same as X-Frame-Options: deny

allow frames from two origins:
1. where it's being served (self, aka sameorigin)
2. https://www.example.org/

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors

More on the Content-Security-Policy: response header in later weeks.

Spoiler: Good news: CSP is rich and powerful! Bad news: CSP is rich and powerful!

Can we prevent a site from submitting a form to our site?

- Why do this?
  - Prevent cross-site request forgery (CSRF; previous lecture)
- How might we accomplish this?
  - Detect Origin header, use an allowlist
  - SameSite cookies
  - What's the difference?

Can we prevent a site from embedding images from our site?

- Why do this?
  - Prevent hotlinking
  - Prevent user's logged-in avatar from showing up on other sites
- How might we accomplish this?
  - For hotlinking: Detect Referer header, use an allowlist (not foolproof)
  - For avatar: Use SameSite cookies
  - For avatar: Use an unpredictable URL

Can we prevent a site from embedding scripts from our site?

- Why do this?
  - Prevent hotlinking
- Important notes
  - Scripts typically do not contain private user data
  - Scripts run in the context of the embedding site
- How might we accomplish this?
  - Similar to images: detect Referer header, use an allowlist (not foolproof)

Typical cross-site script embed

```
<script
src='https://ajax.googleapis.com/ajax/libs/d3js/5.12.0/d3.min.js'></script>
<script>
  d3.select('svg').selectAll('rect').data(data).enter()
</script>
```

Cross-site use of standard libraries (like D3) is common.
We want to *encourage* this kind of reuse.

Revisiting the same origin policy

- Is site A allowed to link to site B? Yes!
  - Or no! (No is not foolproof)
- Is site A allowed to embed site B? Yes!
  - Or no!
- Is site A allowed to embed site B and modify its contents? No!
- Is site A allowed to submit a form to site B? Yes!
  - Or no!
- Is site A allowed to embed images from site B? Yes!
  - Or no! (No is not foolproof)
- Is site A allowed to embed scripts from site B? Yes!
  - Or no! (No is not foolproof)
- Is site A allowed to read data from site B? No!
  - No!

Is site A allowed to read data from site B?

- No!
- Important: embedding an image, script, or iframe is not "reading data"
  - We could embed images, scripts, but not read the actual raw data in them
  - For iframes we couldn't access the DOM to read/write it
- This is precisely what we mean by "reading data":

```
const res = await fetch('https://leoonline.odu.edu/transcript.pdf')
const data = await res.body.arrayBuffer()
console.log(data)
```

What if site A and site B cooperate?

- If a page cooperates, then it can share data with another site
  - e.g., make an iframe and use postMessage to communicate
- What about for arbitrary (e.g. non-HTML) resources?
  - e.g., an API server that returns the current date as JSON:

```
{ "date": 1570552348157 }
```

Use case: Date API Server

- Server code:

```
app.get('/api/date', (req, res) => {
  res.send({ date: Date.now() })
})
```

- Server response:

```
{ "date": 1570552348157 }
```

How to read the response from the client?

- Ideally, site-a.com could write this code:

```
const res = await fetch('https://site-b.com/api/date')
const data = await res.body.json()
console.log(data)
```

- Need some way for site to specify that response is allowed to be read
  - Ideally, HTTP response could specify an HTTP header indicating that reading this data is allowed
  - Challenge: can we do it without an HTTP header?

Using <script> for cross-site communication

- Goal: site-a.com wants to read data from a cooperating site-b.com
- What if we requested data using a <script> tag?
  - <script> is not subject to the Same Origin Policy
- Remember: Cannot read data from a cross-origin script!
  - But, the contents will be treated as JavaScript and executed
  - Can we use this somehow?

Naive idea

- Add a script to site-a.com:

```
<script src='https://site-b.com/api/date'></script>
```

- Response from site-b.com/api/date:

```
{ "date": 1570552348157 }
```

- Problems:
  - Not quite valid JavaScript
  - Script can be executed and results observed (i.e., rendered), but contents can't be *read* by a script at site-a.com

JSONP: JSON with Padding

- Add a script to site-a.com:

```
<script>
  function handleTime (data)  {
        console.log('got the date', data.date)
  }
</script>
<script src='https://site-b.com/api/date?callback=handleTime'></script>
```

site-a.com: I would like to access your response in a function I'm choosing to call handleTime(). Please name it that.

- Response from site-b.com/api/date?callback=handleTime

```
handleTime({ "date": 1570552348157 })
```

site-b.com: I hear you loud and clear, and 've named my little Javascript response accordingly.

Downsides of JSONP

- From site-a.com's perspective:
  - Need to write additional code to support cross-origin requests
  - Need to be careful: Some valid JSON strings are not legal JavaScript
  - Only want to get data from site-b.com, but need to give site-a.com the ability to run arbitrary JavaScript from site-b.com – yikes!
- From site-b.com's perspective:
  - Need to sanitize user-provided callback argument (see upcoming "reflected file download attack")

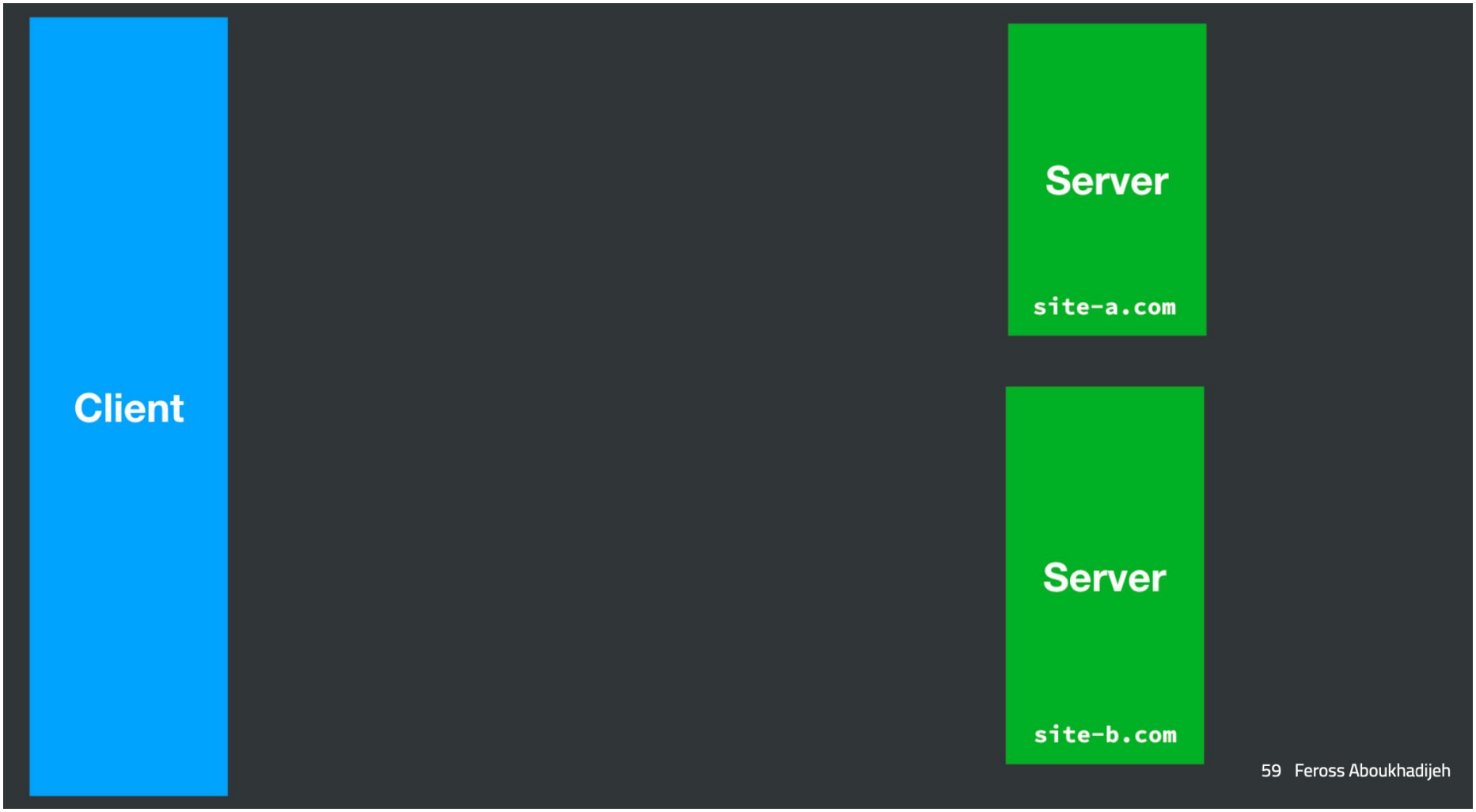
Cross-Origin Resource Sharing (CORS)

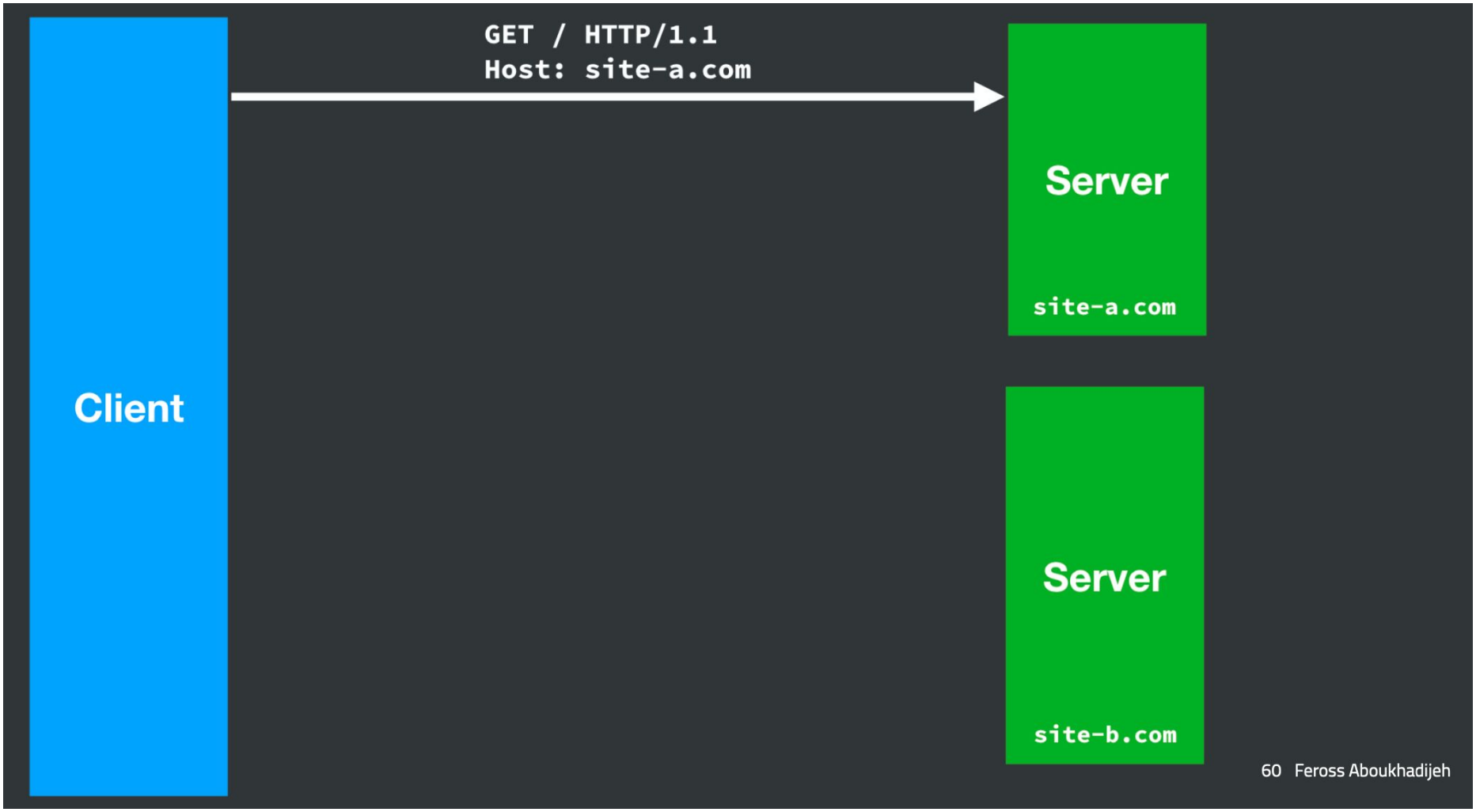
- site-b.com allows origin https://site-a.com to read data:

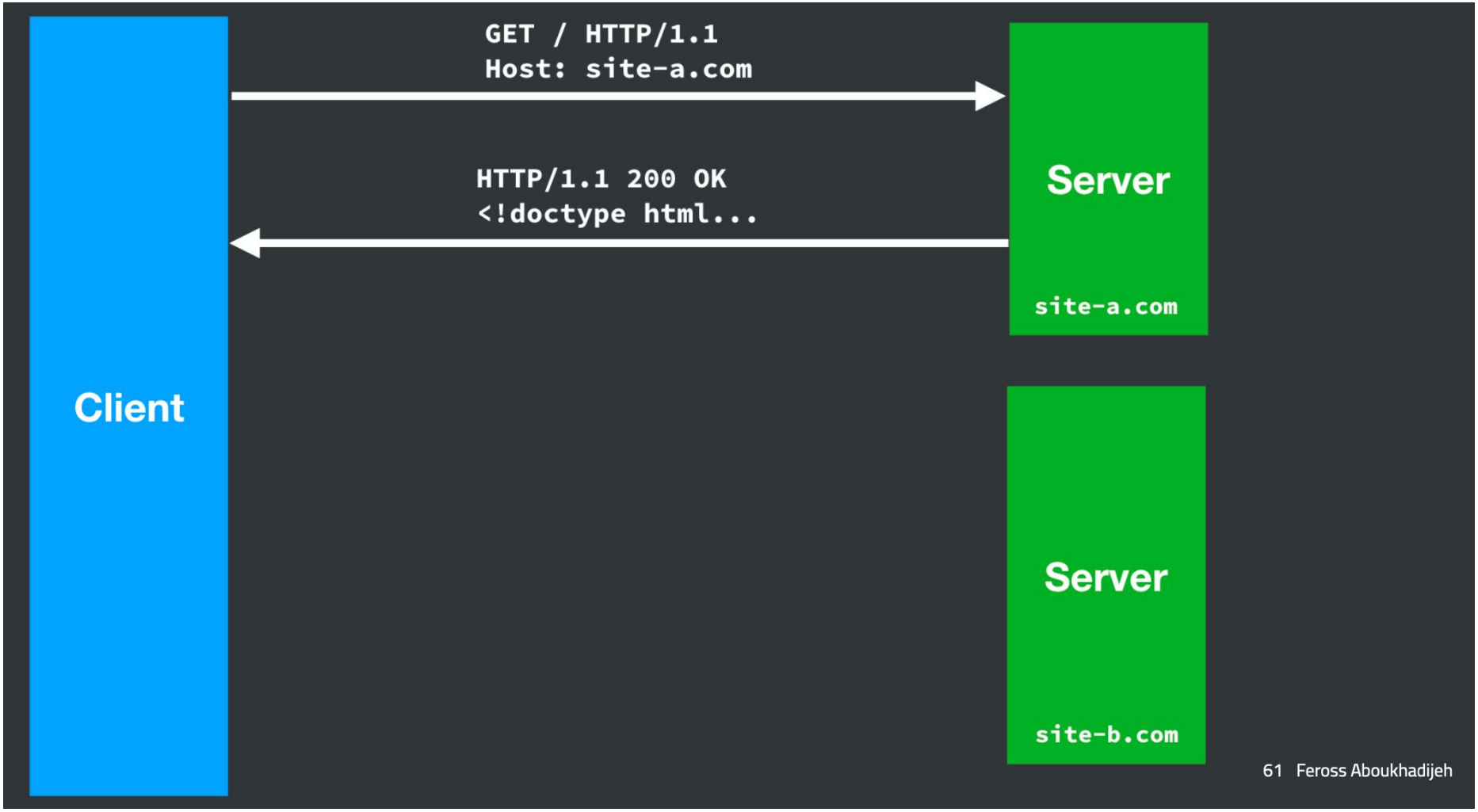
```
Access-Control-Allow-Origin: https://site-a.com
```

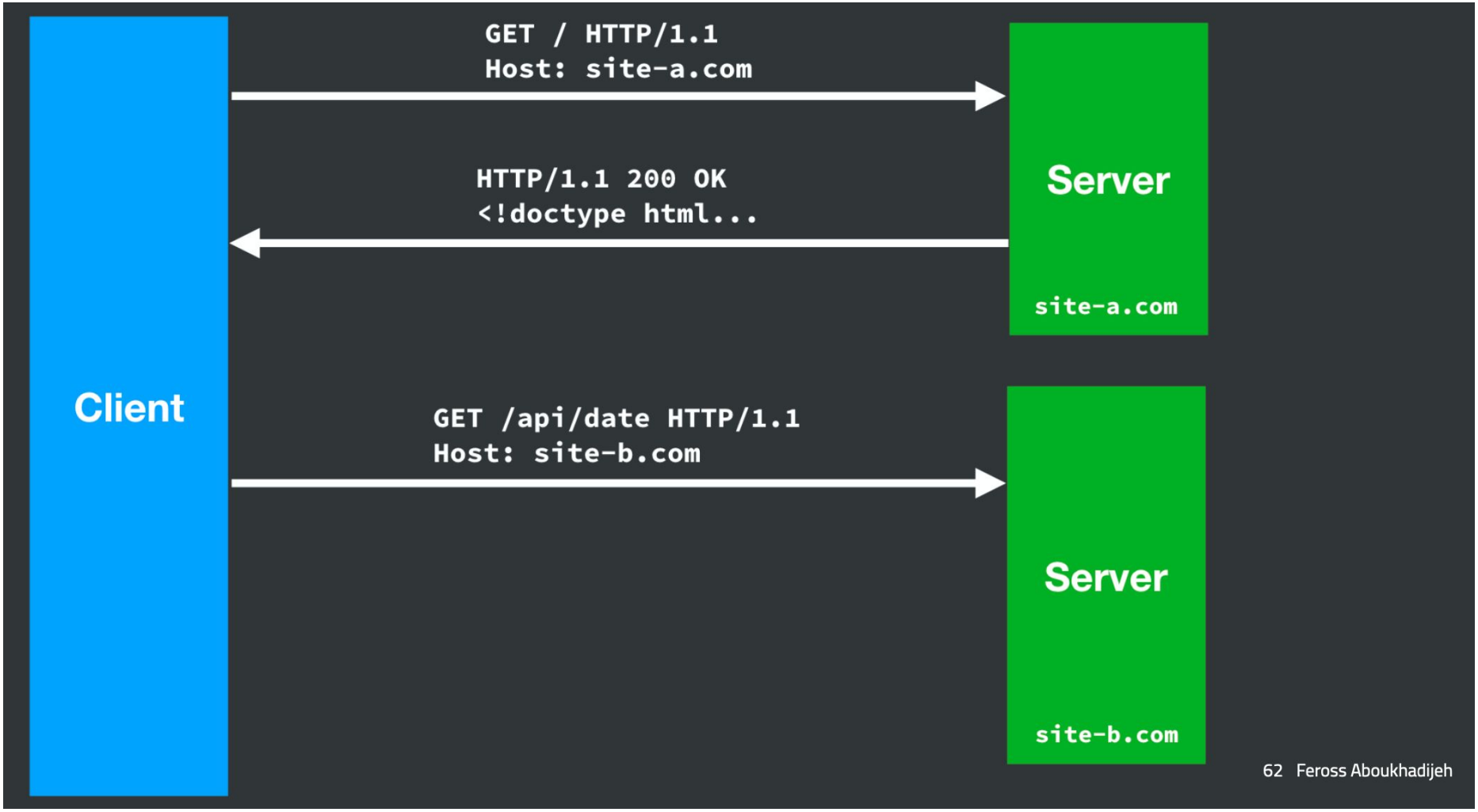
- site-b.com allows any origin to read data:

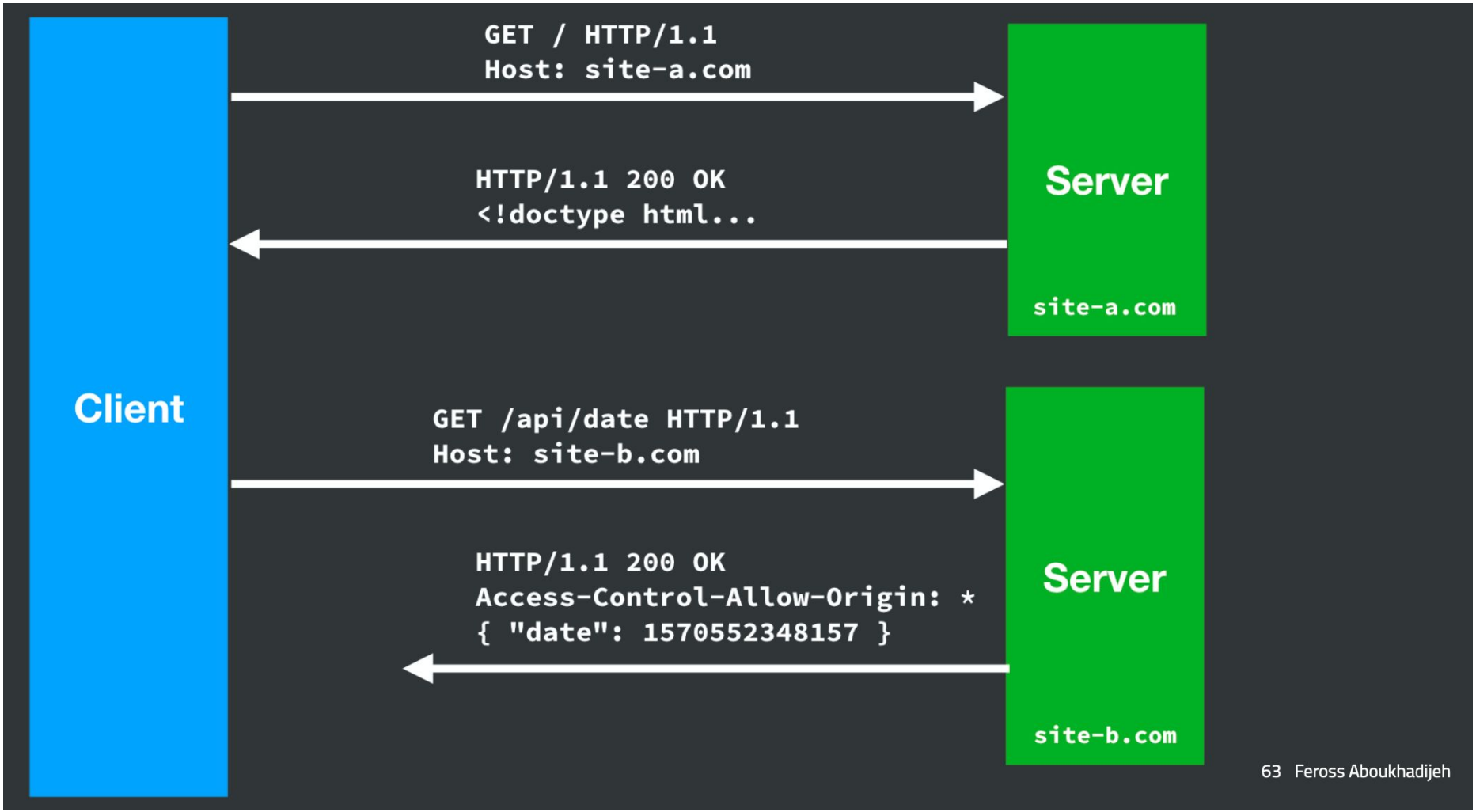
```
Access-Control-Allow-Origin: *
```

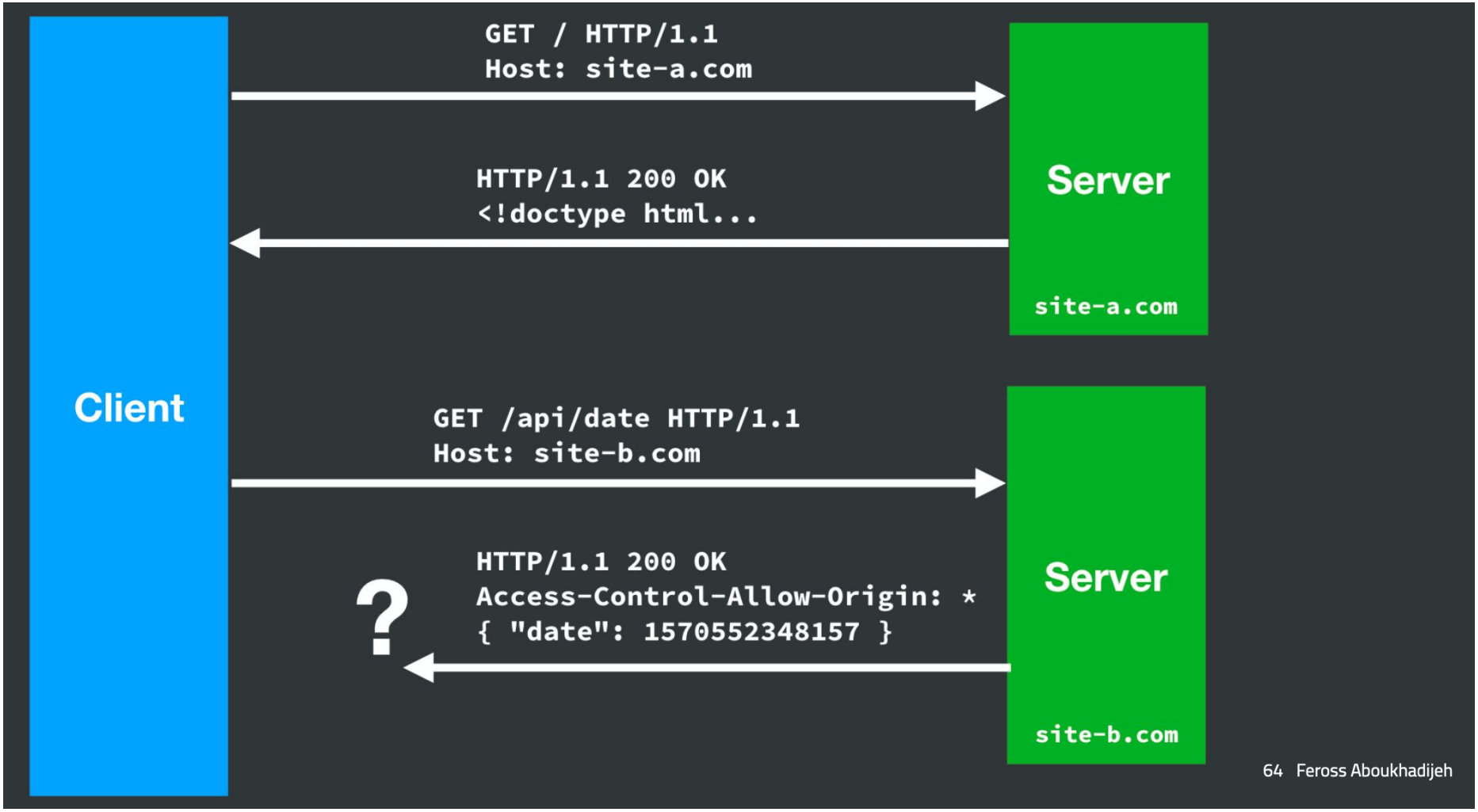

Client
Server
site-a.com
Server
site-b.com
59 Feross Aboukhadijeh

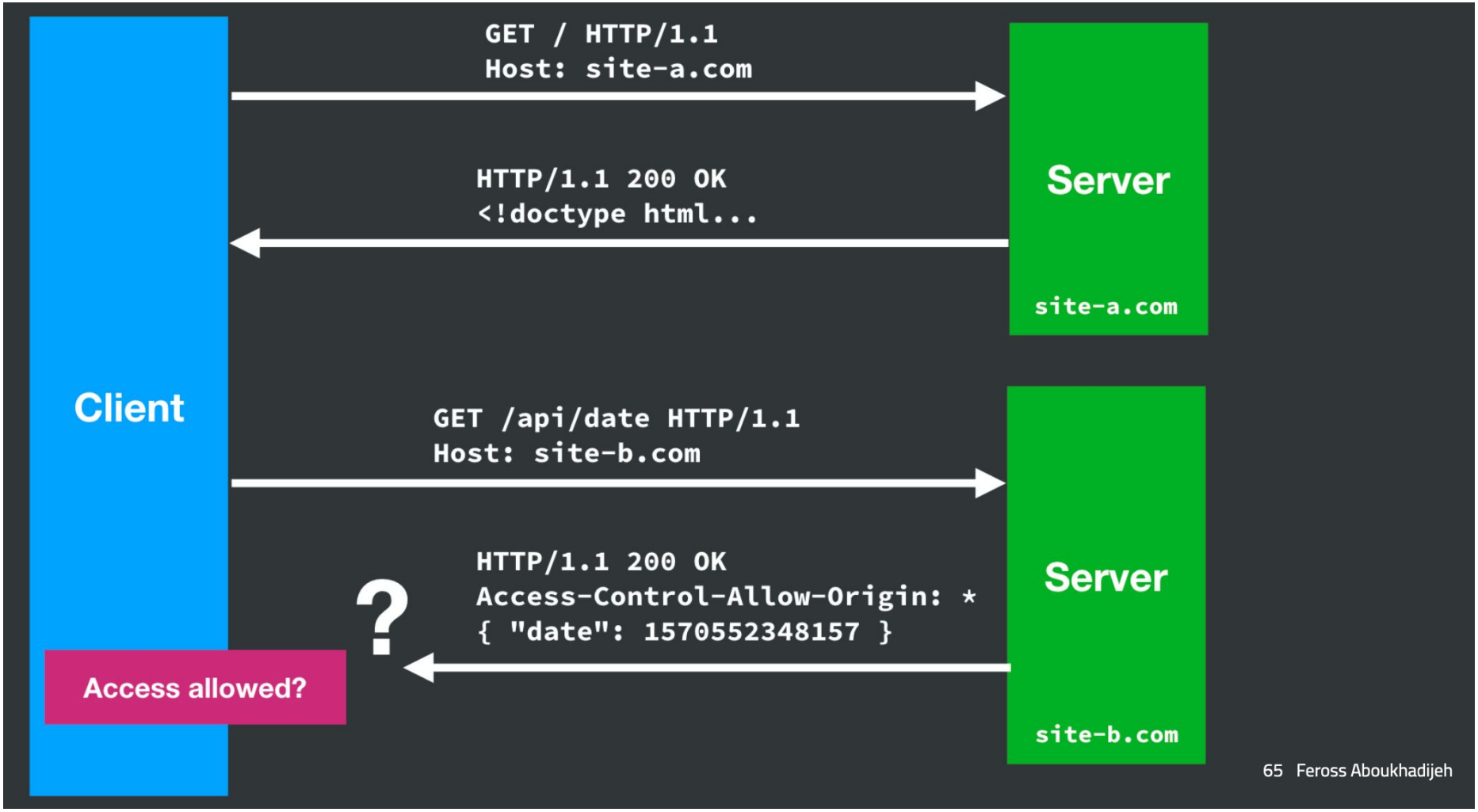

GET / HTTP/1.1
Host: site-a.com
Client
Server
site-a.com
Server
site-b.com
60 Feross Aboukhadijeh

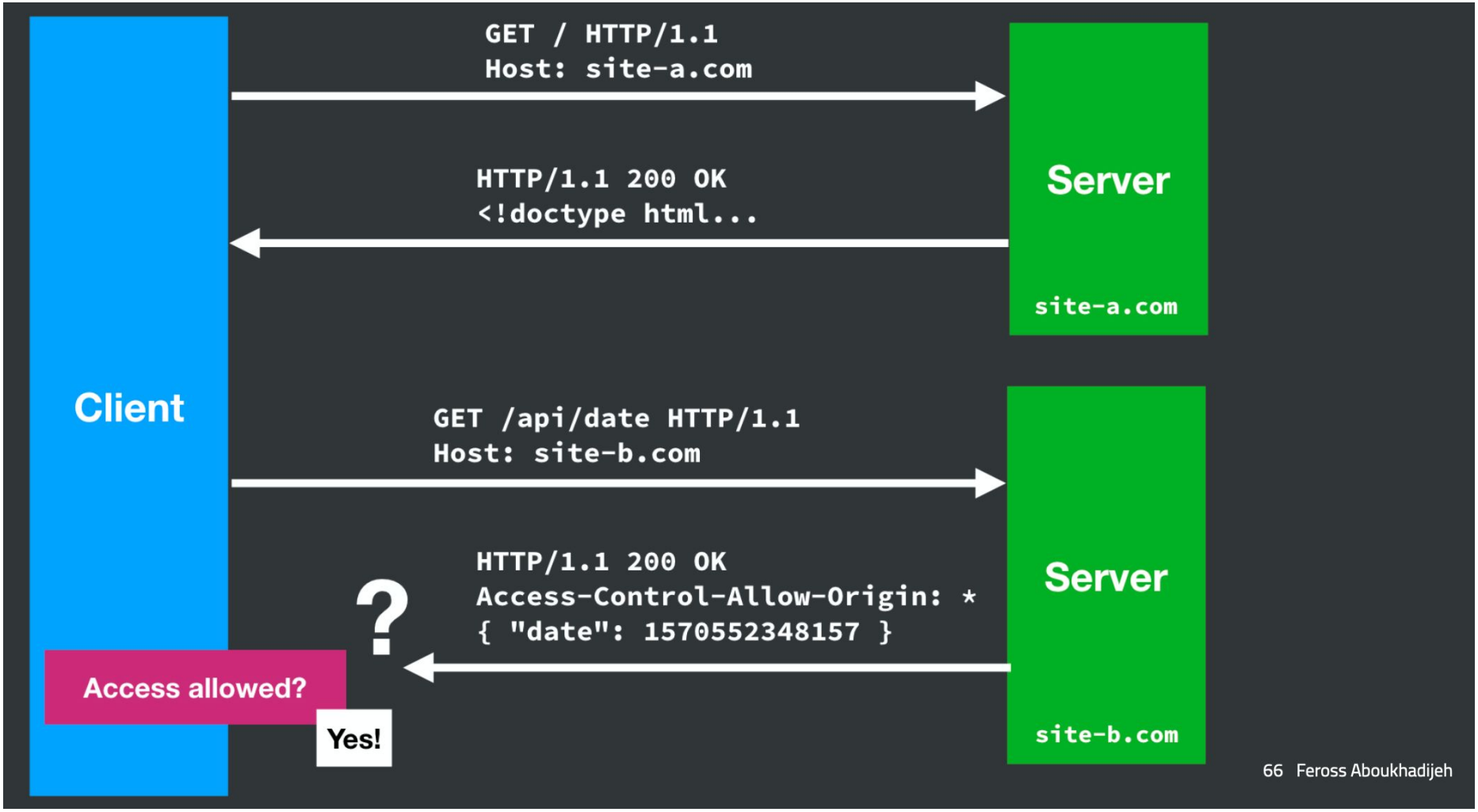

GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Client
Server
site-a.com
Server
site-b.com
61 Feross Aboukhadijeh

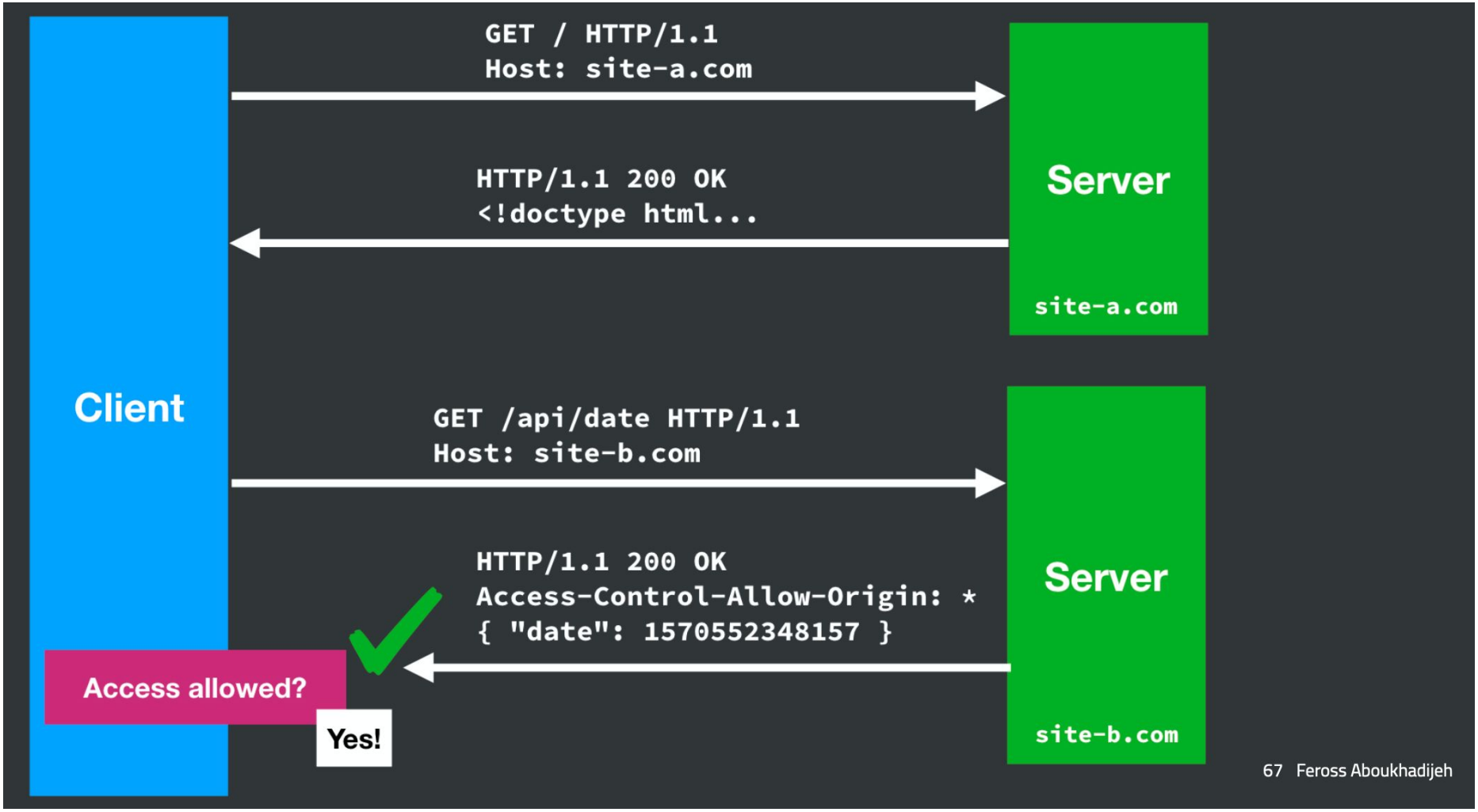

Client
GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
GET /api/date HTTP/1.1
Host: site-b.com
Server
site-b.com
62 Feross Aboukhadijeh


Client
GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
GET /api/date HTTP/1.1
Host: site-b.com
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
{ "date": 1570552348157 }
Server
site-b.com
63 Feross Aboukhadijeh


Client
GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
GET /api/date HTTP/1.1
Host: site-b.com
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
{ "date": 1570552348157 }
?
Server
site-b.com
64 Feross Aboukhadijeh


Client
GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
GET /api/date HTTP/1.1
Host: site-b.com
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
{ "date": 1570552348157 }
?
Server
site-b.com
Access allowed?
65 Feross Aboukhadijeh


GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
Client
GET /api/date HTTP/1.1
Host: site-b.com
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
{ "date": 1570552348157 }
?
Server
site-b.com
Access allowed?
Yes!
66 Feross Aboukhadijeh


GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
Client
GET /api/date HTTP/1.1
Host: site-b.com
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
{ "date": 1570552348157 }
Server
site-b.com
Access allowed?
Yes!
67 Feross Aboukhadijeh

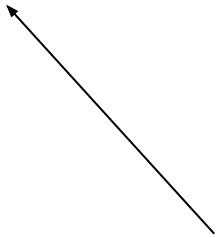
CORS version of the Date API

- Server code:

```
app.get('/api/date', (req, res) => {
  res.set('Access-Control-Allow-Origin', '*')
  res.send({ date: Date.now() })
})
```

- Server response:

```
{ "date": 1570552348157 }
```



this is the server (site-b.com) telling all browsers: “it’s cool if anyone reads this data -- there’s nothing private or protected in the response”

Ensuring private data returned by an authenticated API route isn't read by other sites?

- Don't set Access-Control-Allow-Origin header (no fetch read)
- Don't return data in JSONP format (no <script> read)
- Just return JSON
  - JSON like { "date": 1570552348157 } can't be read by <script>
  - JSON response will never be valid JavaScript *...right?*
  - Or even if it is, it's not assigned to a variable so it's inaccessible *...right?*

Real world CORS example

```
% curl -isL memgator.cs.odu.edu/timemap/json/http://www.odu.edu/ | head -20
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Link, Location, X-Memento-Count, Server
Content-Type: application/json
Date: Thu, 11 Feb 2021 17:42:30 GMT
Server: MemGator/1.0-rc8
X-Memento-Count: 5116
Transfer-Encoding: chunked

{
  "original_uri": "http://www.odu.edu/",
  "self": "https://memgator.cs.odu.edu/timemap/json/http://www.odu.edu/",
  "mementos": {
    "list": [
    {
    "datetime": "1996-12-21T05:13:52Z",
    "uri": "https://web.archive.org/web/19961221051352/http://odu.edu:80/"
    },
    {
    "datetime": "1997-12-11T00:24:13Z",
```

[much more to the response, this is only the first 20 lines]