CS 253: Web Security

Exceptions to the Same Origin Policy, Cross-Site Script Inclusion

Why does Google put)]}' at beginning of all their API responses?

```
)]}'

[[["og.botresp",["<redacted>"]
,null,[null,null,null,"//www.google.com/js/bg/<redacted>.js"]
,"<redacted>"]
,["di",10]
,["af.httprm",20,"<redacted>",0]
,["e",4,null,null,3456]
]]
```

Can we configure Same Origin Policy?

- Yes!
- Can **harden** it, or **relax** it!

What does Same Origin Policy allow?

- 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

Can we prevent a site from linking to our site?

- Why do this?
  - Search Engine Optimization (SEO)
  - Make the linking site look bad?
- How might we accomplish this?
  - Can't prevent the link itself
  - Can we reject request?

Prevent linking

- Idea: Look for the **`Referer`** header and reject certain requests?

```
GET / HTTP/1.1
Referer: https://competitor.com/
```

`Referrer-Policy` HTTP header

- **`Referrer-Policy: unsafe-url`**
  - Send full URL.
- **`Referrer-Policy: no-referrer`**
  - Never send **`Referer`**
- **`Referrer-Policy: no-referrer-when-downgrade`** (default)
  - Send full URL. When HTTPS → HTTP downgrade, send nothing.
- **`Referrer-Policy: origin`**
  - All: Send origin instead of full URL.

`Referrer-Policy` HTTP header

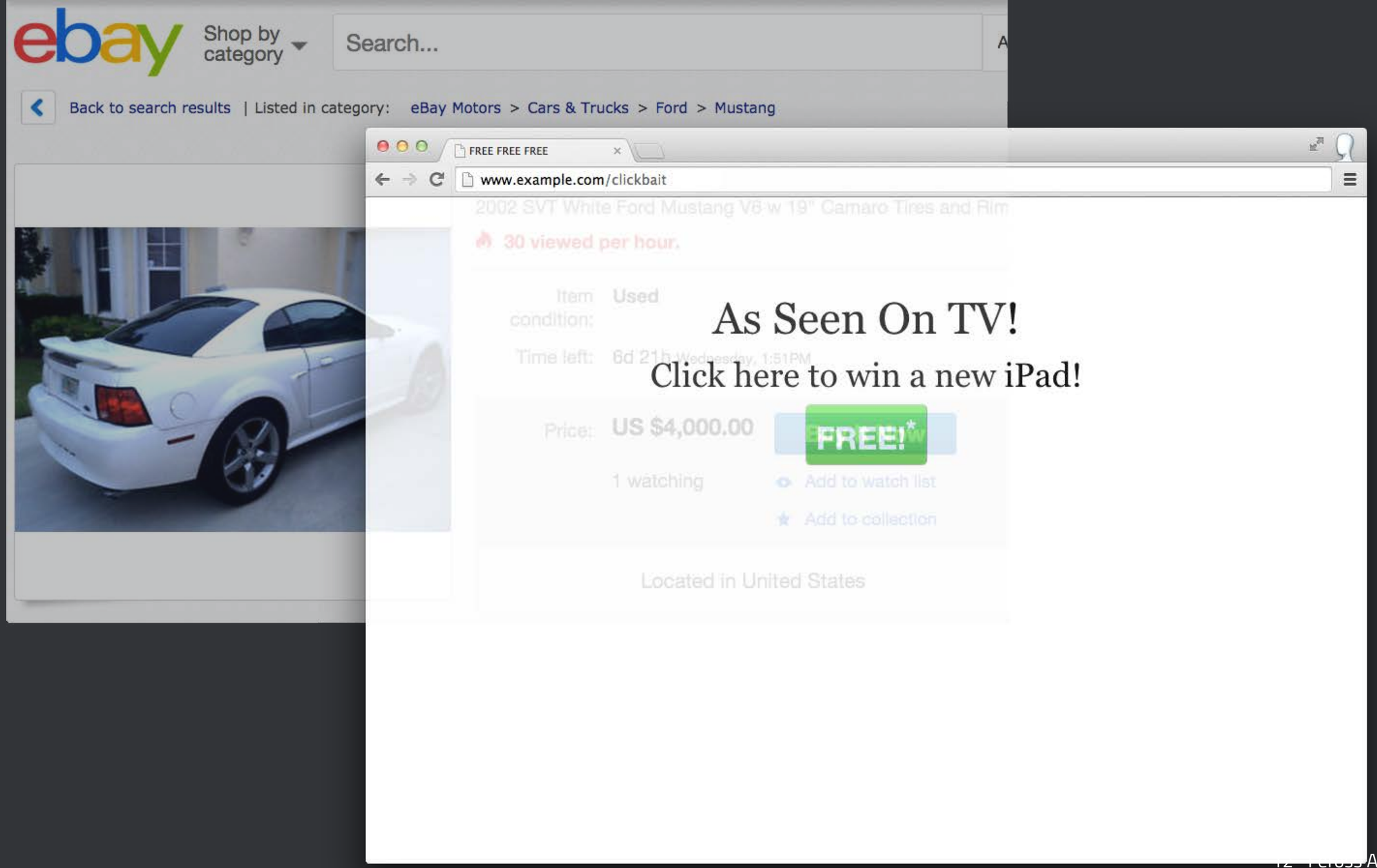
- **`Referrer-Policy: origin-when-cross-origin`**
  - Same origin: send full URL. Cross origin: send origin.
- **`Referrer-Policy: same-origin`**
  - Same origin: send full URL. Cross origin: send nothing.
- **`Referrer-Policy: strict-origin`**
  - Send origin. When HTTPS → HTTP downgrade, send nothing.
- **`Referrer-Policy: strict-origin-when-cross-origin`**
  - Same origin: send full URL. Cross origin: send origin. When HTTPS → HTTP downgrade, send nothing.

Prevent linking

- Seems hard to prevent linking!
- Let me know if you figure out a way to do it

Can we prevent a site from embedding our site?

- Why do this?
  - Prevent clickjacking attacks


ebay
Shop by category
Search...
Back to search results | Listed in category: eBay Motors > Cars & Trucks > Ford > Mustang
FREE FREE FREE
www.example.com/clickbait
2002 SVT White Ford Mustang V8 w 19" Camaro Tires and Rim
30 viewed per hour.
Item condition: Used
Time left: 6d 21h Wednesday, 1:51PM
Price: US $4,000.00
1 watching
Add to watch list
Add to collection
Located in United States
As Seen On TV!
Click here to win a new iPad!
FREE!*

Can we prevent a site from embedding our site?

- Why do this?
  - Prevent clickjacking attacks
- How might we accomplish this?
  - Check if we are framed via JavaScript (frame busting)
  - Need a new HTTP header!

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

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

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

***Keywords*-frames; frame busting; clickjacking**

***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.**

I. 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 click-jacking [9] 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

Our survey shows that an average of 3.5 lines of JavaScript was used while the largest implementation spanned over 25 lines. The majority of frame busting code was structured as a *conditional block* to test for framing followed by a *counter-action* if framing is detected. A majority of counter-actions try to navigate the top-frame to the correct page while a few erased the framed content, most often through a `document.write('')`. Some use exotic conditionals and counter actions. We describe the frame busting codes we found in the next sections.

sites	frame bust
Top 500	14%
Top 100	37%
Top 10	60%

Table I
FRAME BUSTING AMONG ALEXA-TOP 500 SITES

Frame busting

```
if (window.top.location != window.location) {
  window.top.location = window.location
}
```

- Don't do this!

`X-Frame-Options` HTTP Header

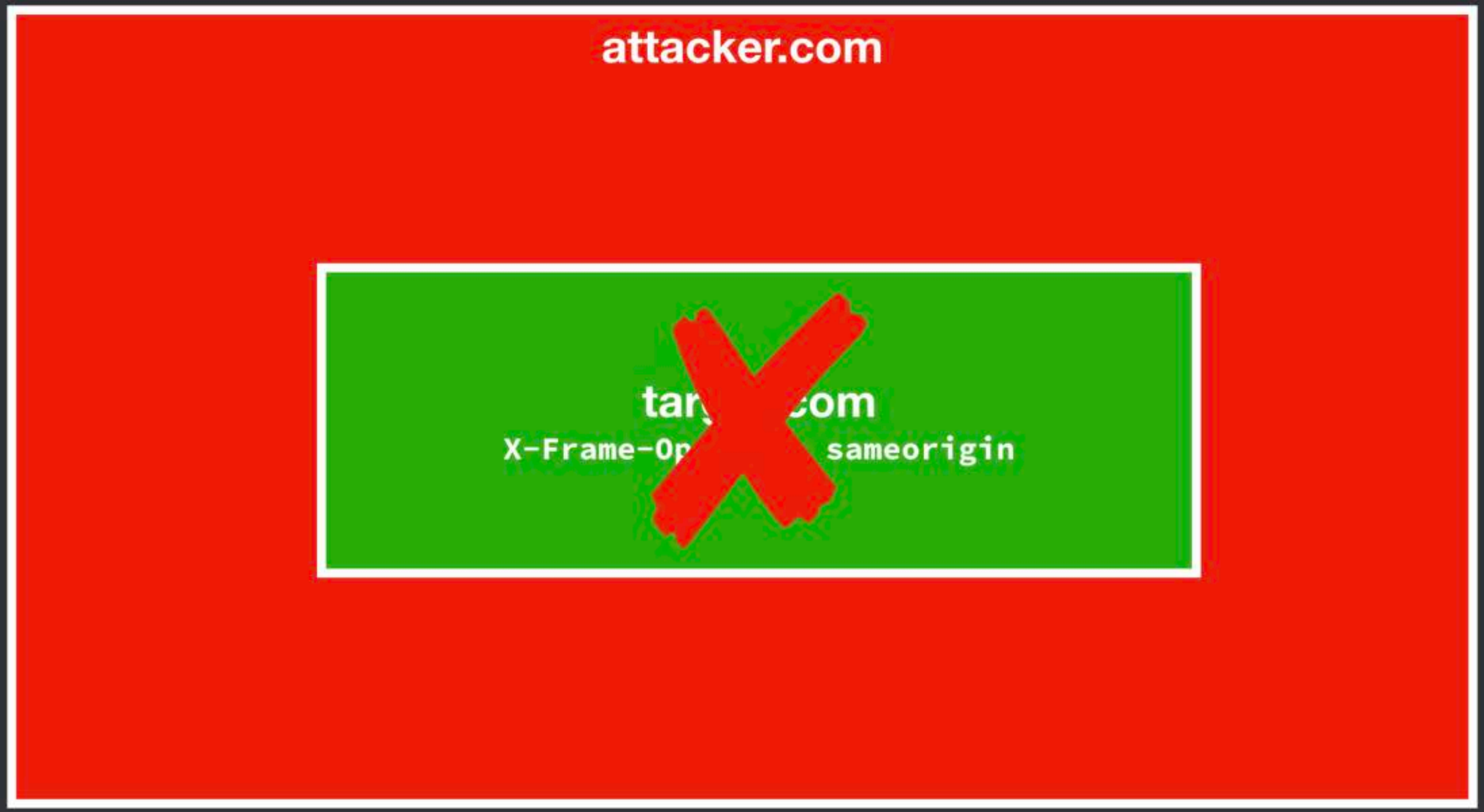
- **`X-Frame-Options`** not specified (Default)
  - Any page can display this page in an iframe
- **`X-Frame-Options: deny`**
  - Page can not be displayed in an iframe
- **`X-Frame-Options: sameorigin`**
  - Page can only be displayed in an iframe on the same origin as the page itself

attacker.com

attacker.com

target.com

`X-Frame-Options: sameorigin`

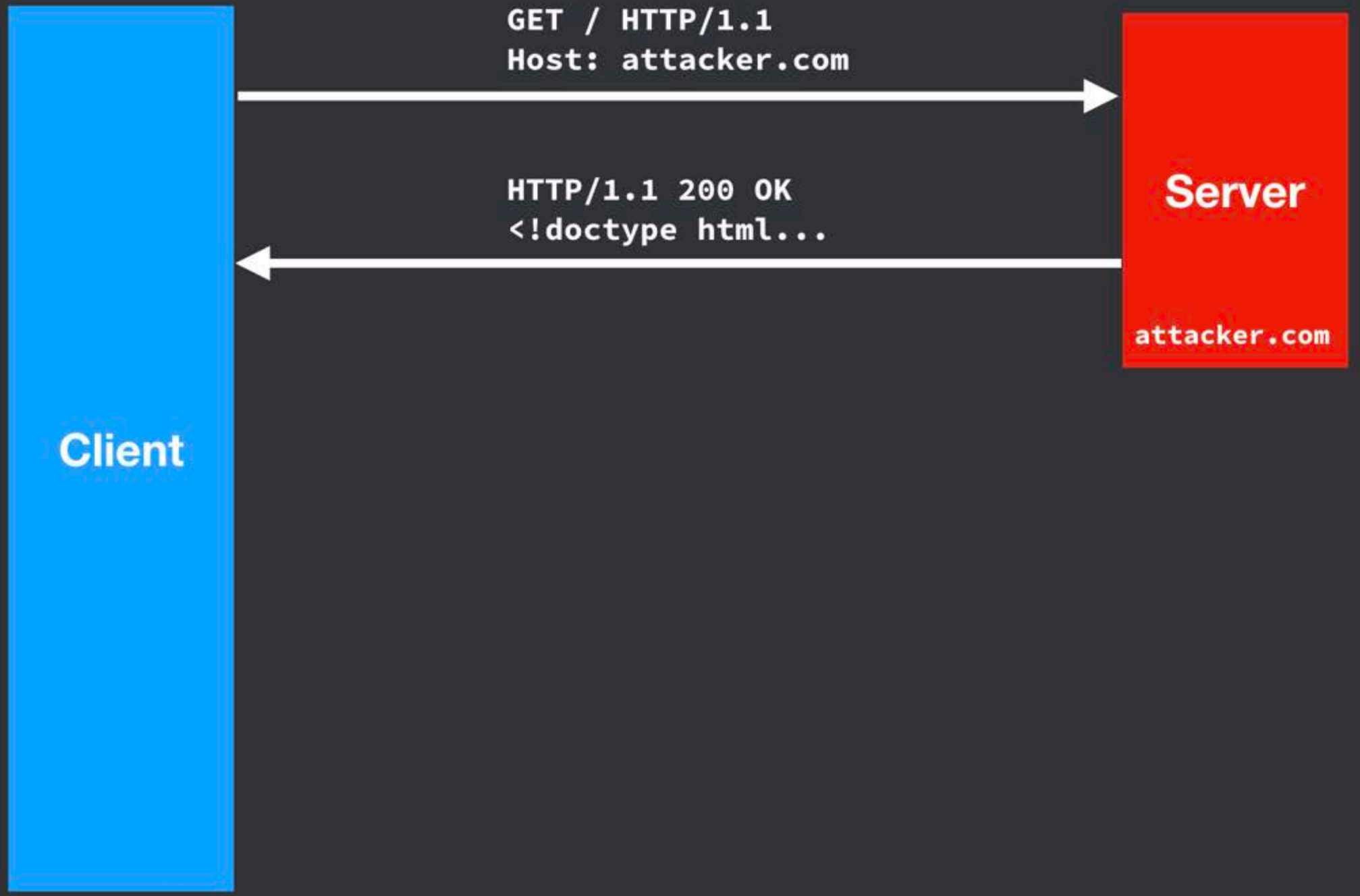

attacker.com
X-Frame-Op
sameorigin

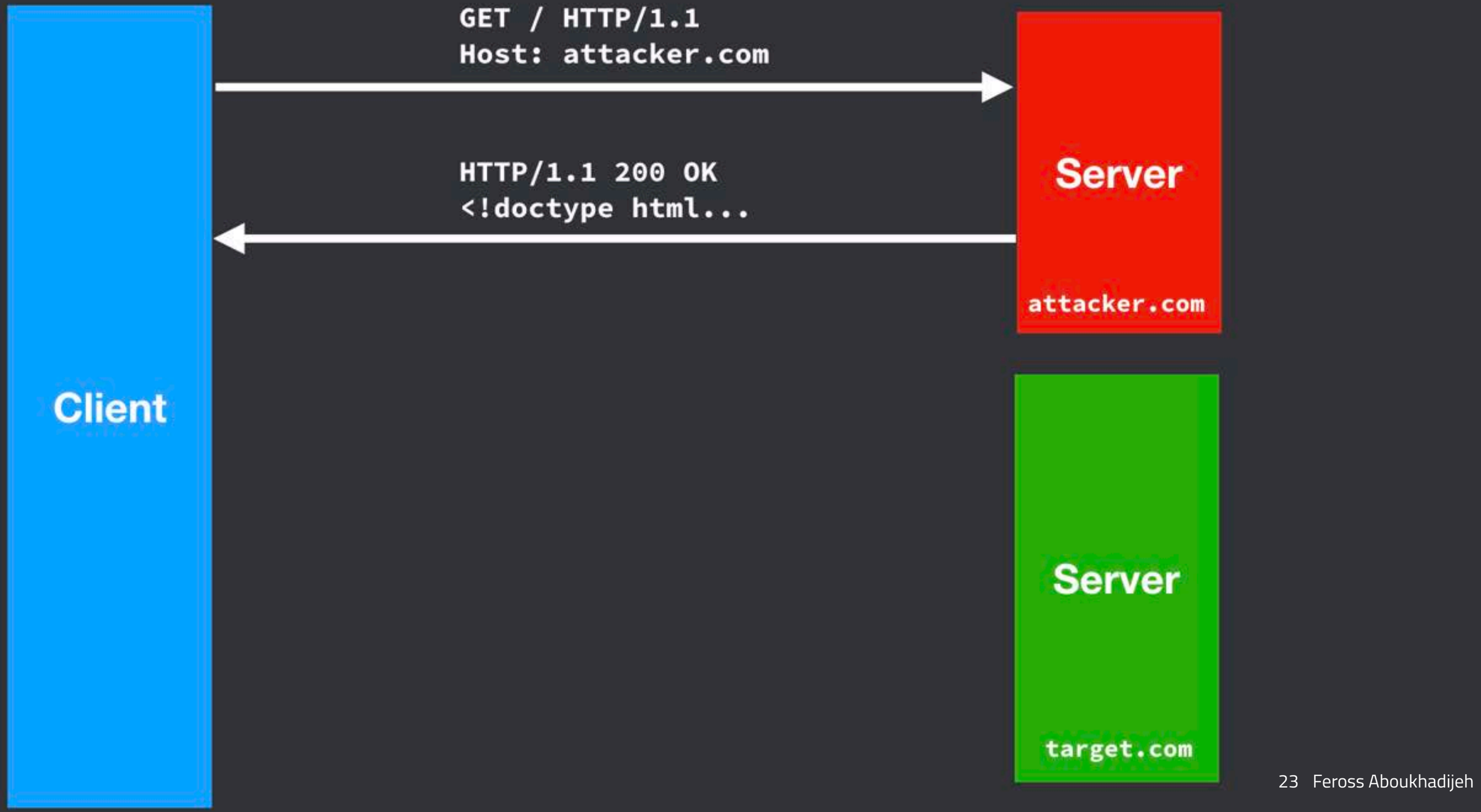
Client

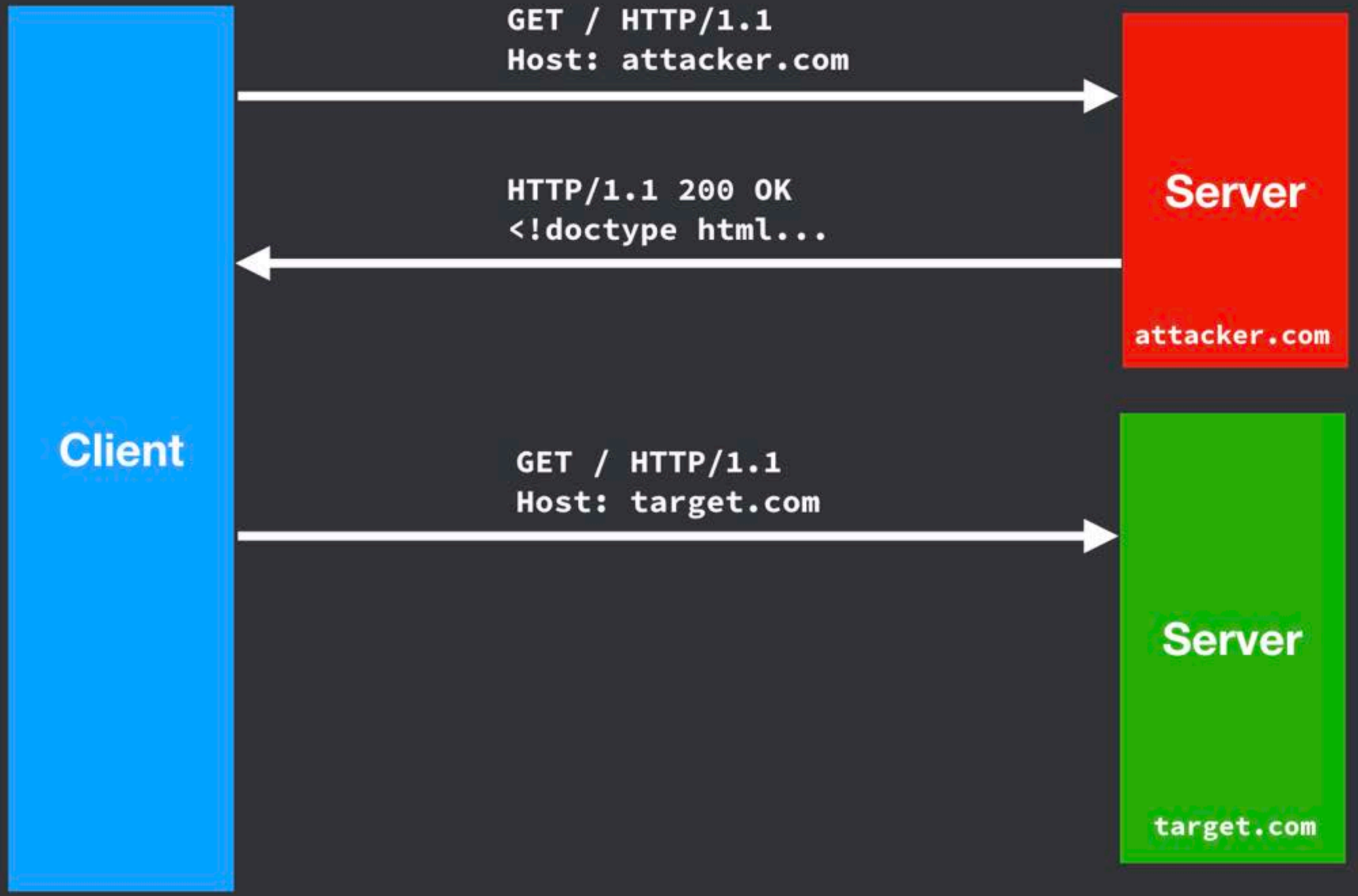
Server

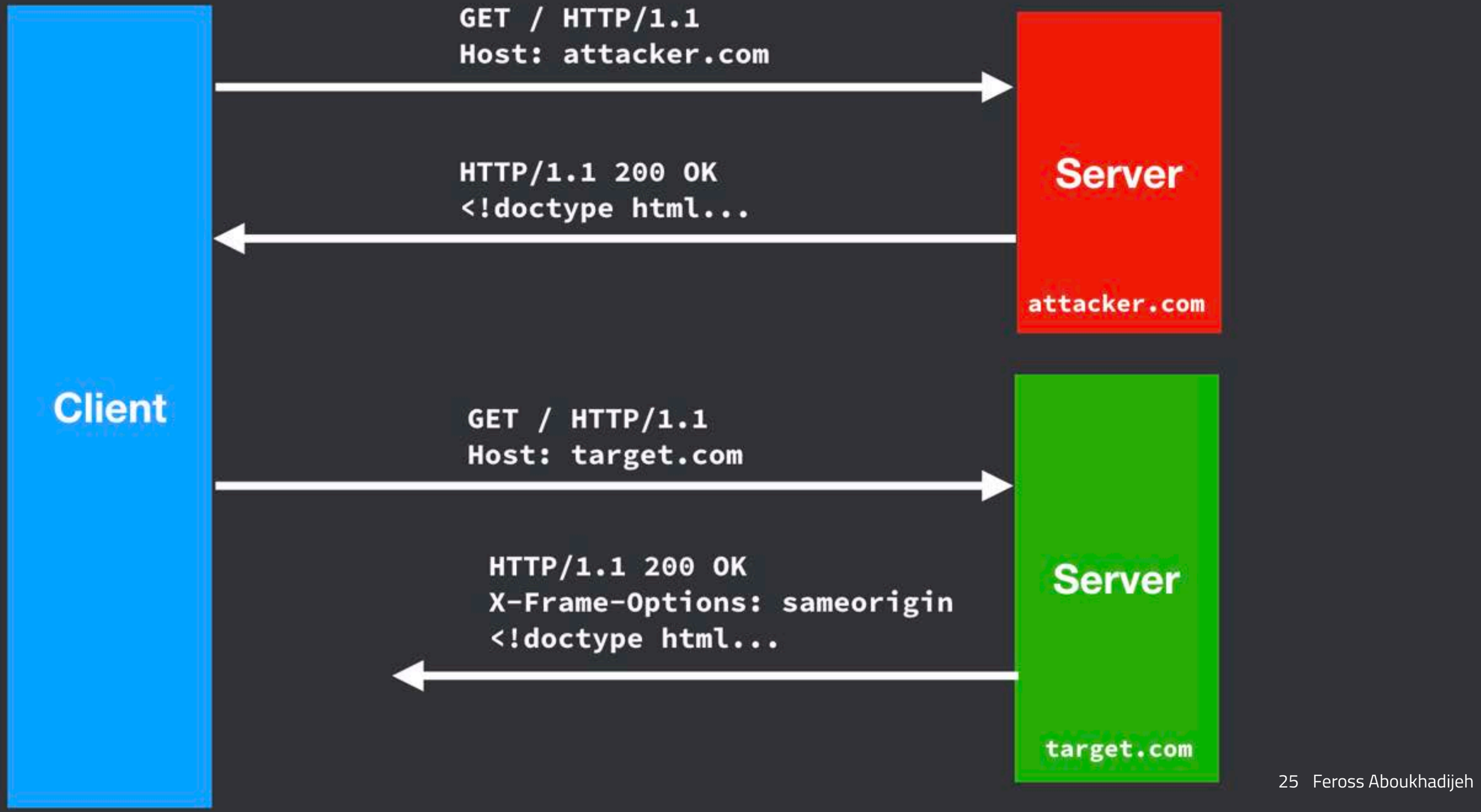
attacker.com

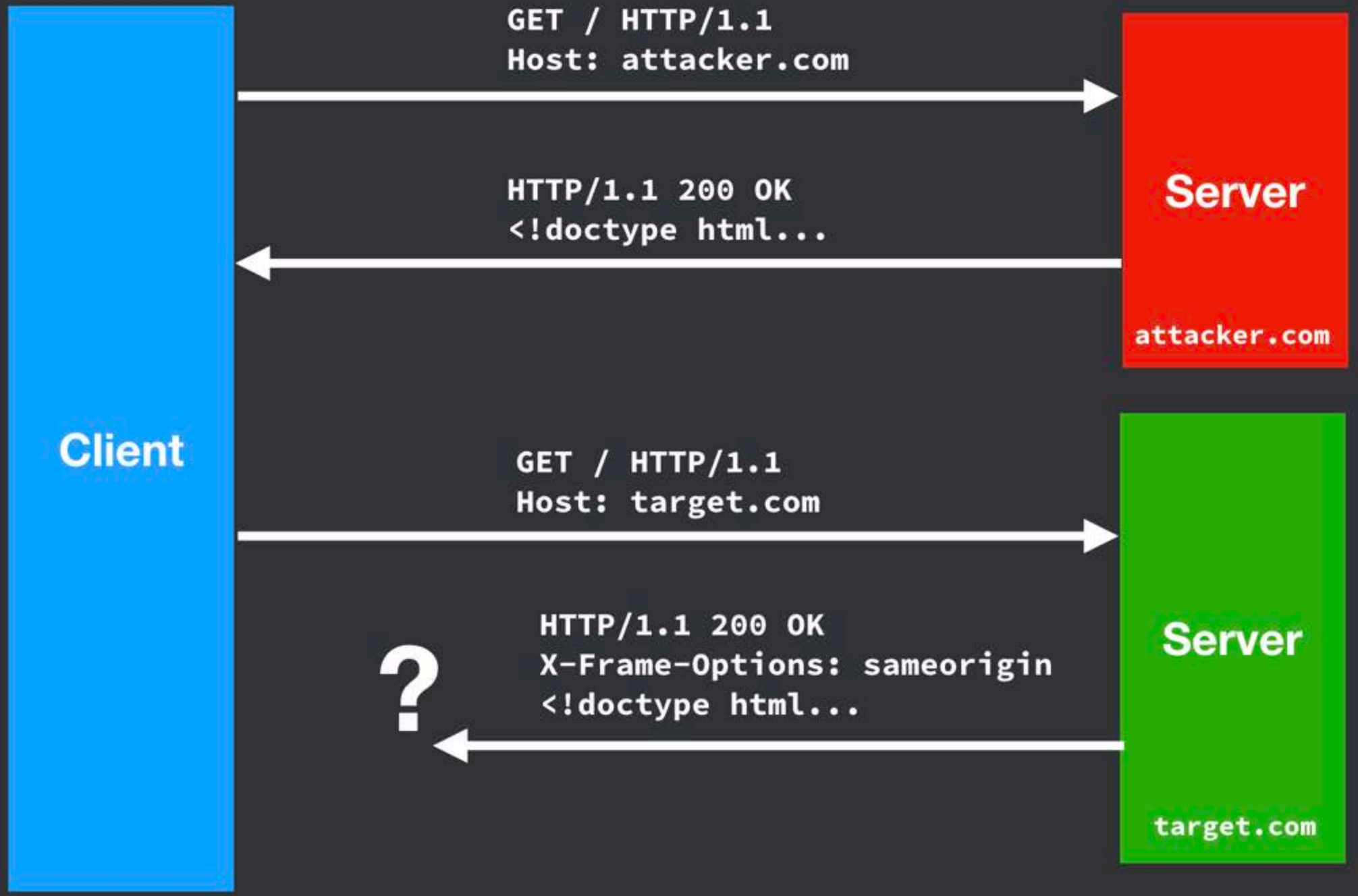

GET / HTTP/1.1
Host: attacker.com
Client
Server
attacker.com

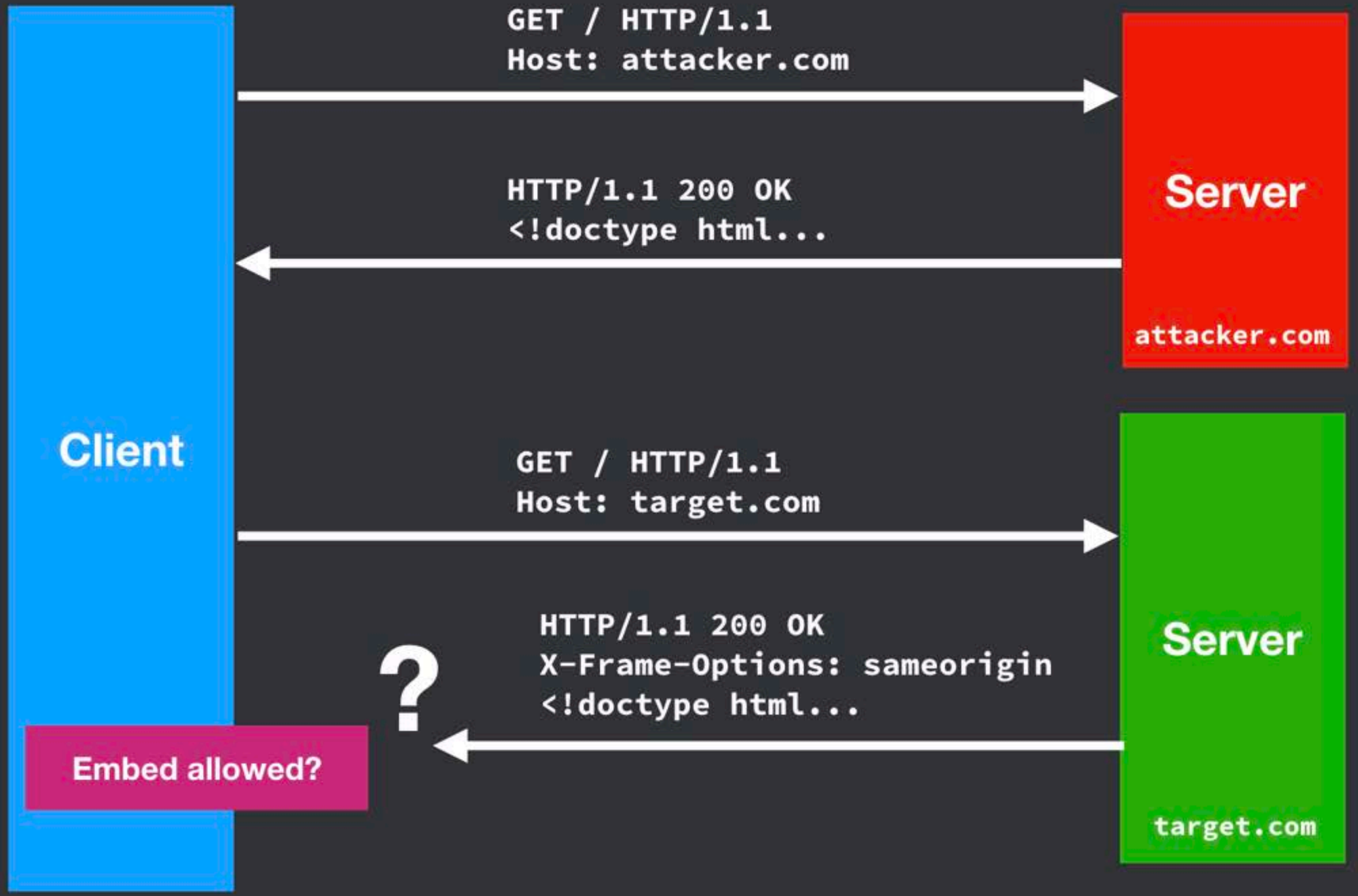

GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Server
attacker.com
Client

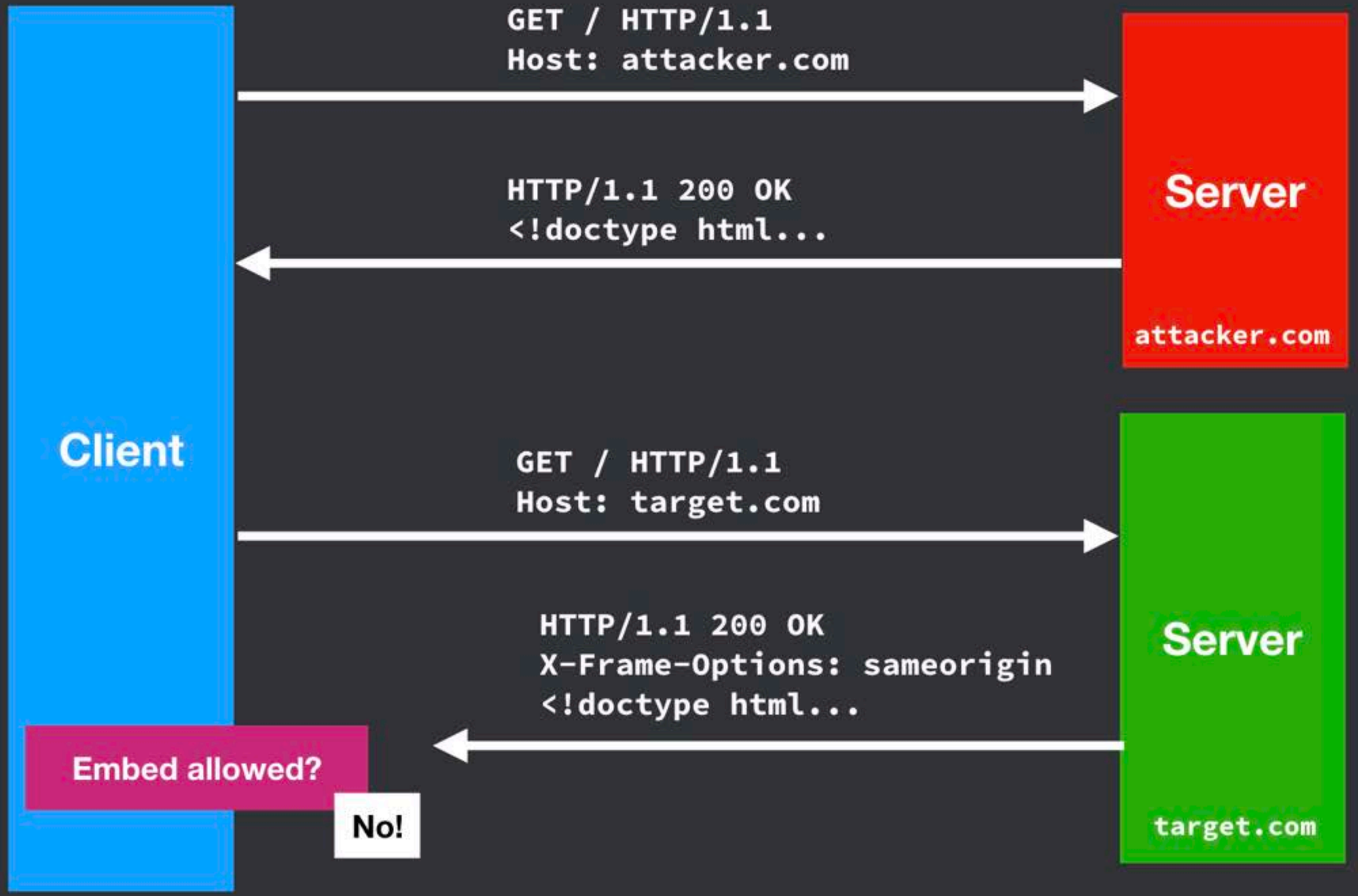

GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Client
Server
attacker.com
Server
target.com

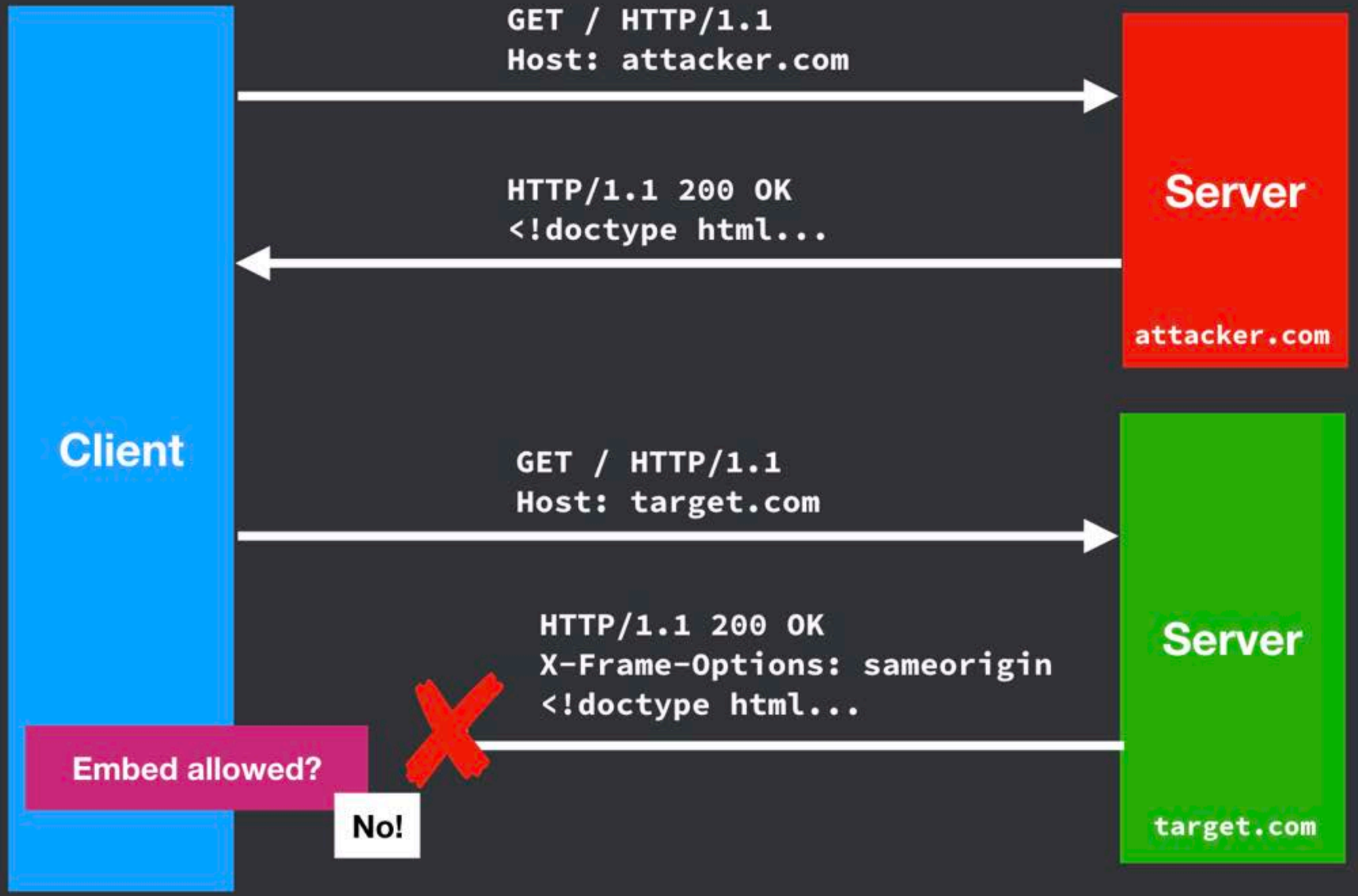

GET / HTTP/1.1
Host: attacker.com
HTTP/1.1 200 OK
<!doctype html...
Client
Server
attacker.com
GET / HTTP/1.1
Host: target.com
Server
target.com


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


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


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?


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!


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!

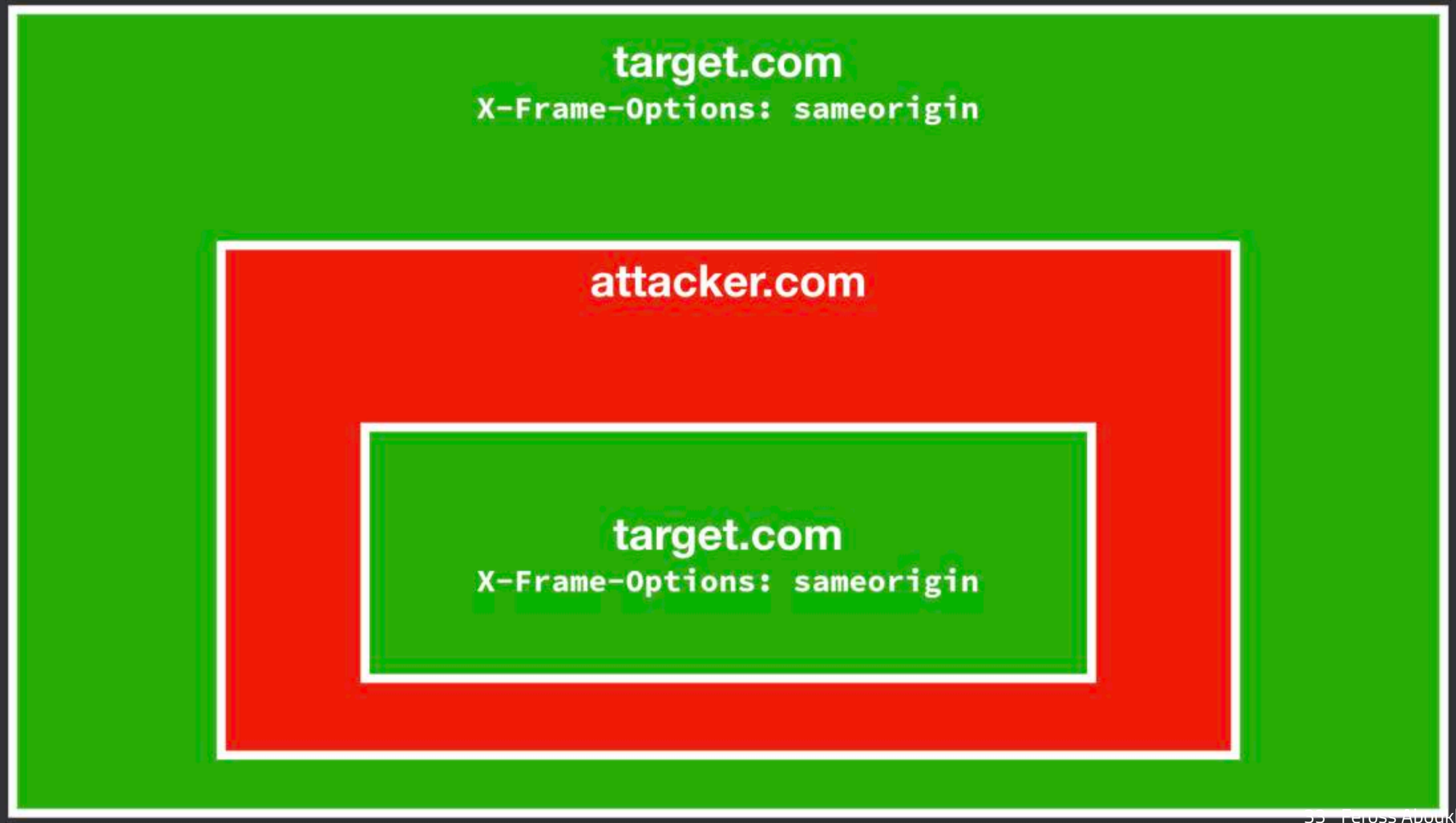
What does `X-Frame-Options: sameorigin` actually mean?

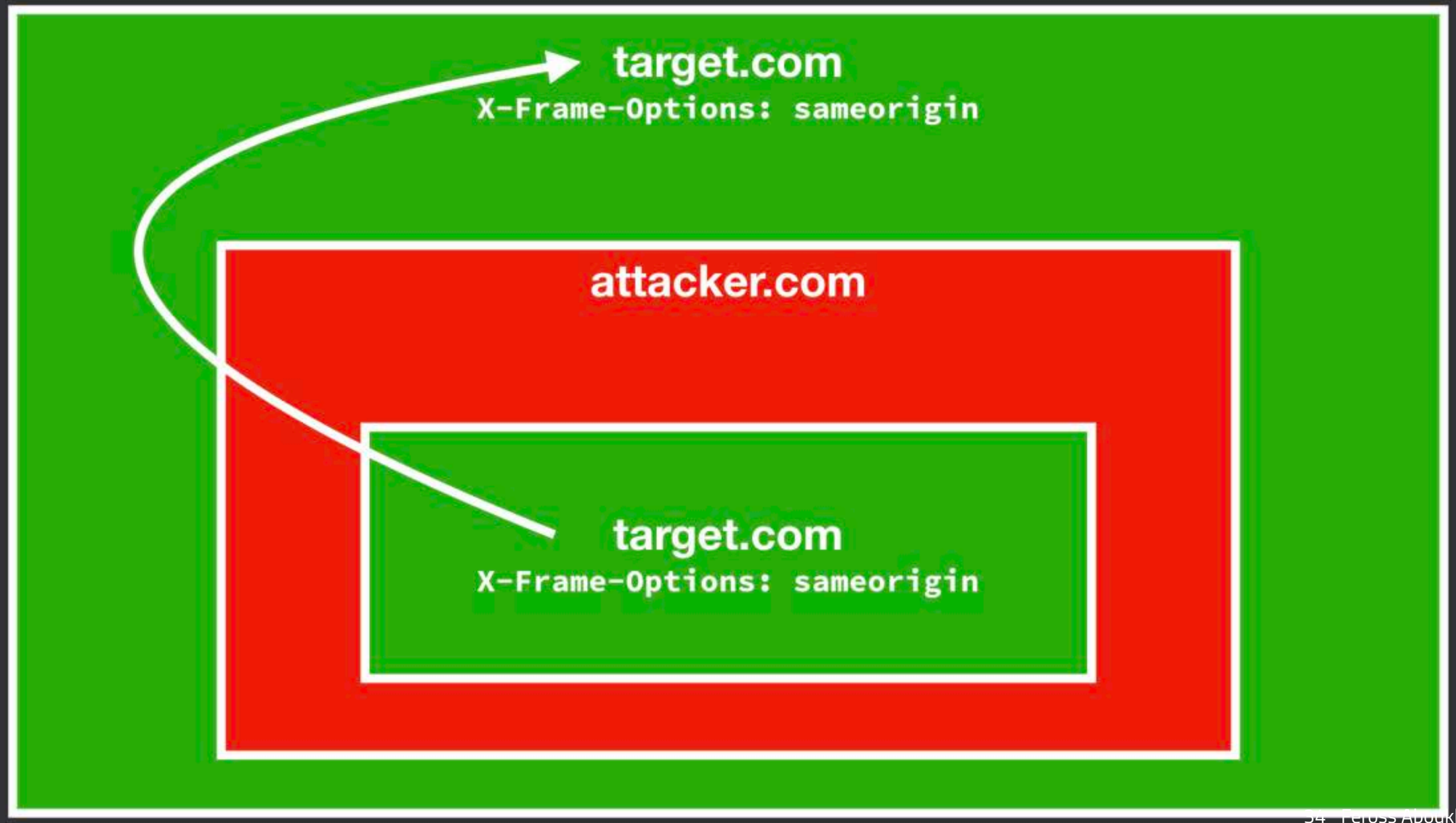
- What should happen if **target.com** embeds **othersite.com** which embeds **target.com**?
- Until recently, browsers performed a check only against **top-level window**
- Thus, attackers could set up a framing chain which would be **allowed**:
  - **target.com** embeds **attacker.com** embeds **target.com**

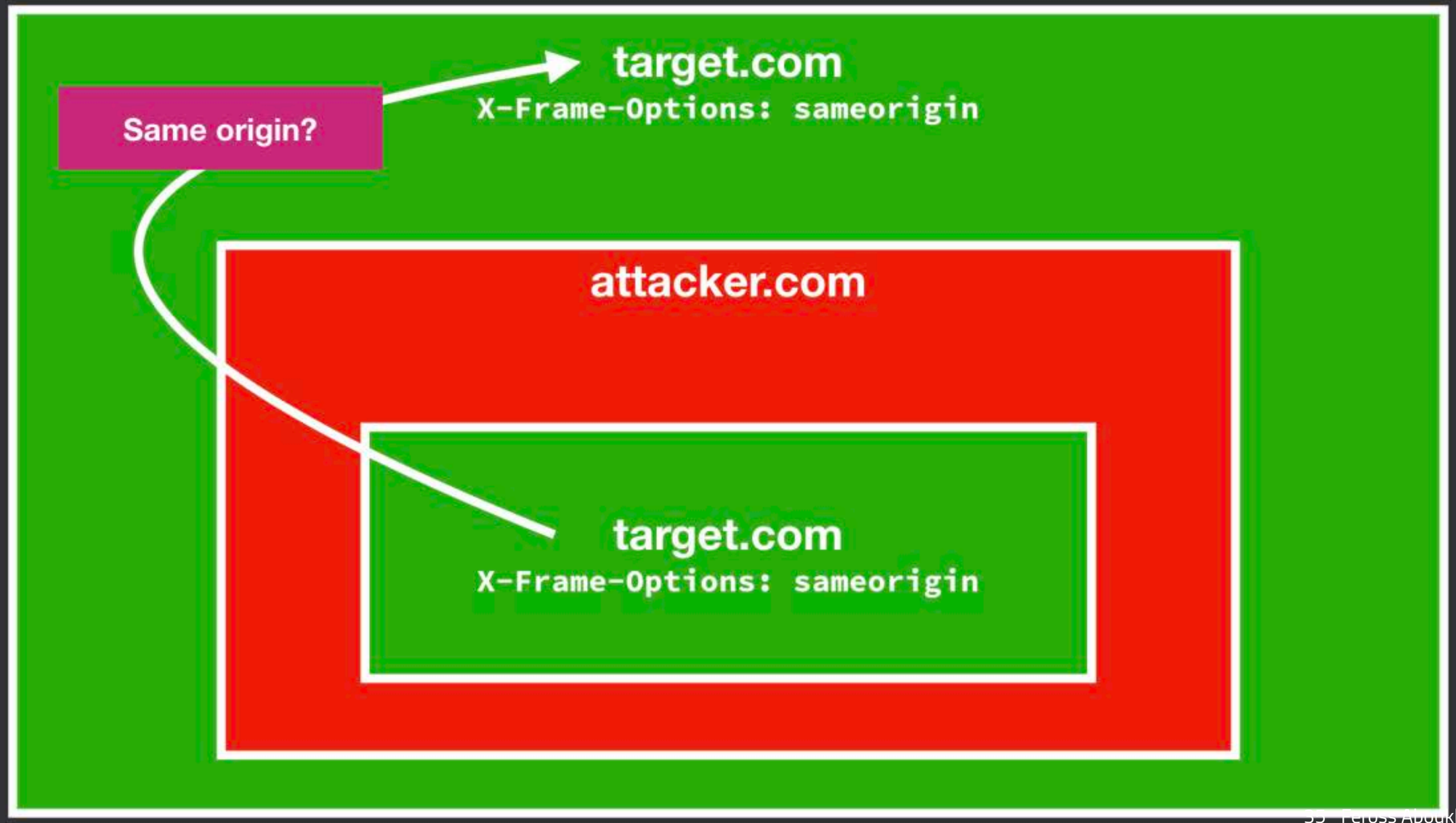
target.com

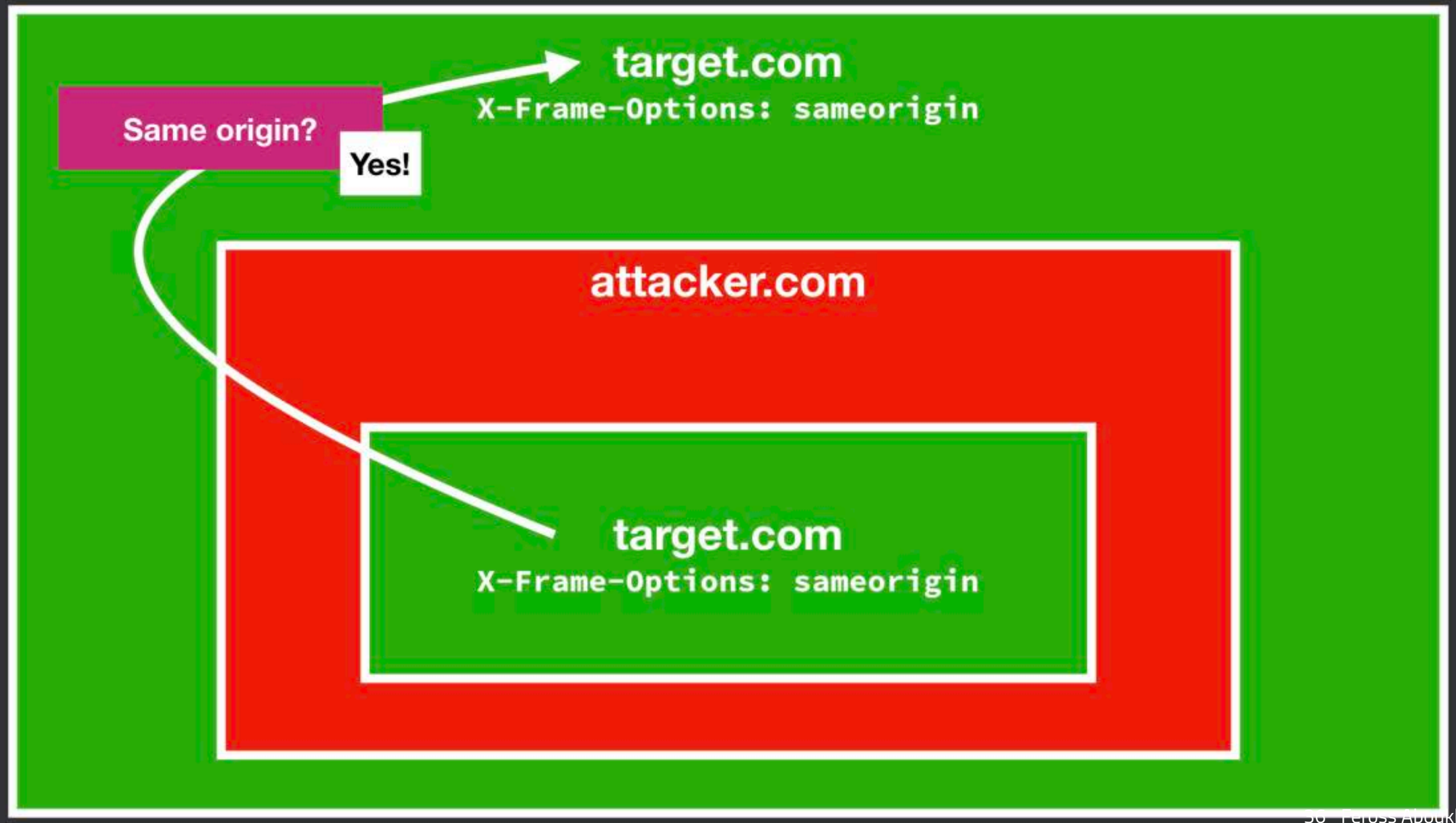
`X-Frame-Options: sameorigin`

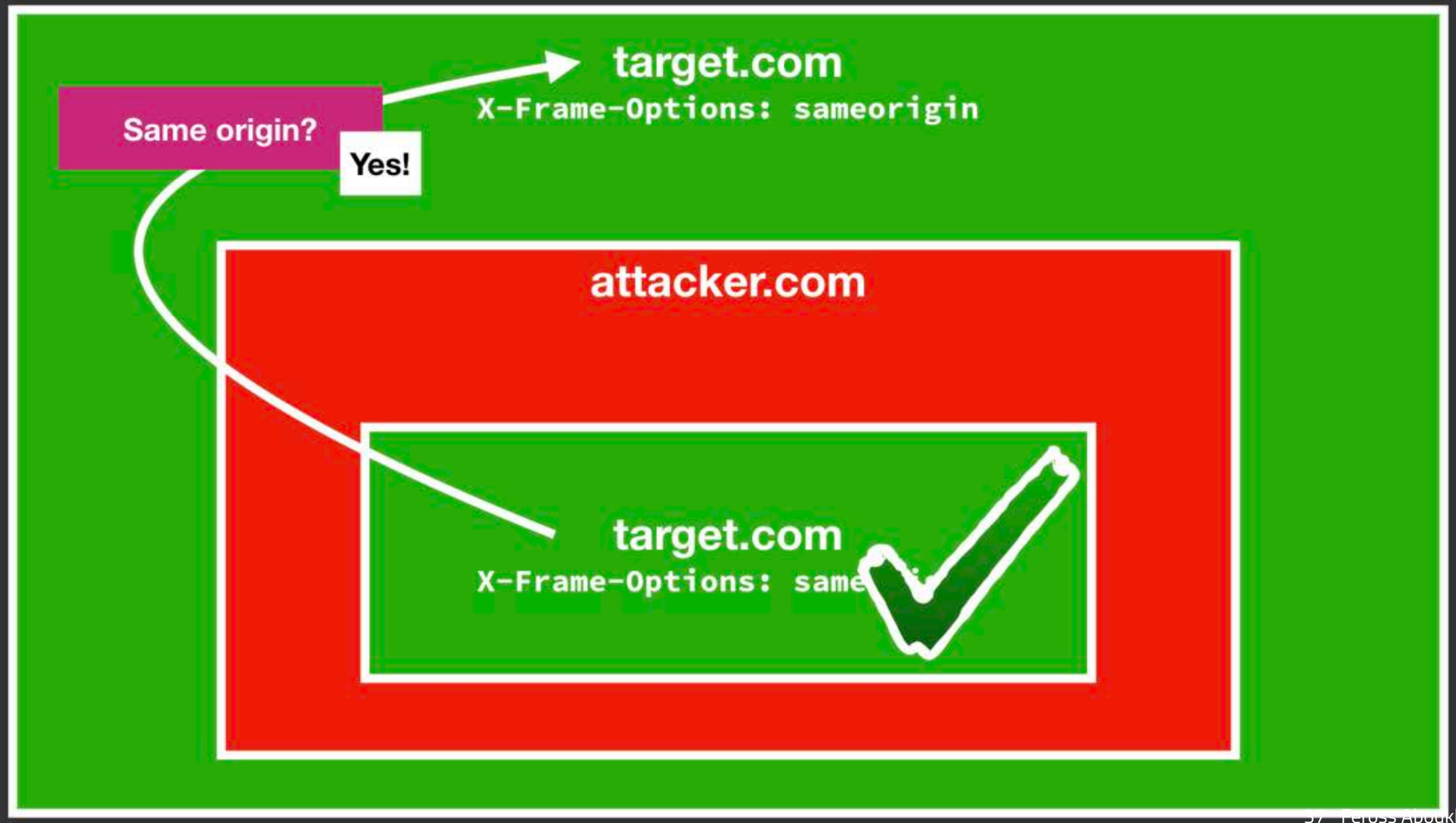

target.com
X-Frame-Options: sameorigin
attacker.com

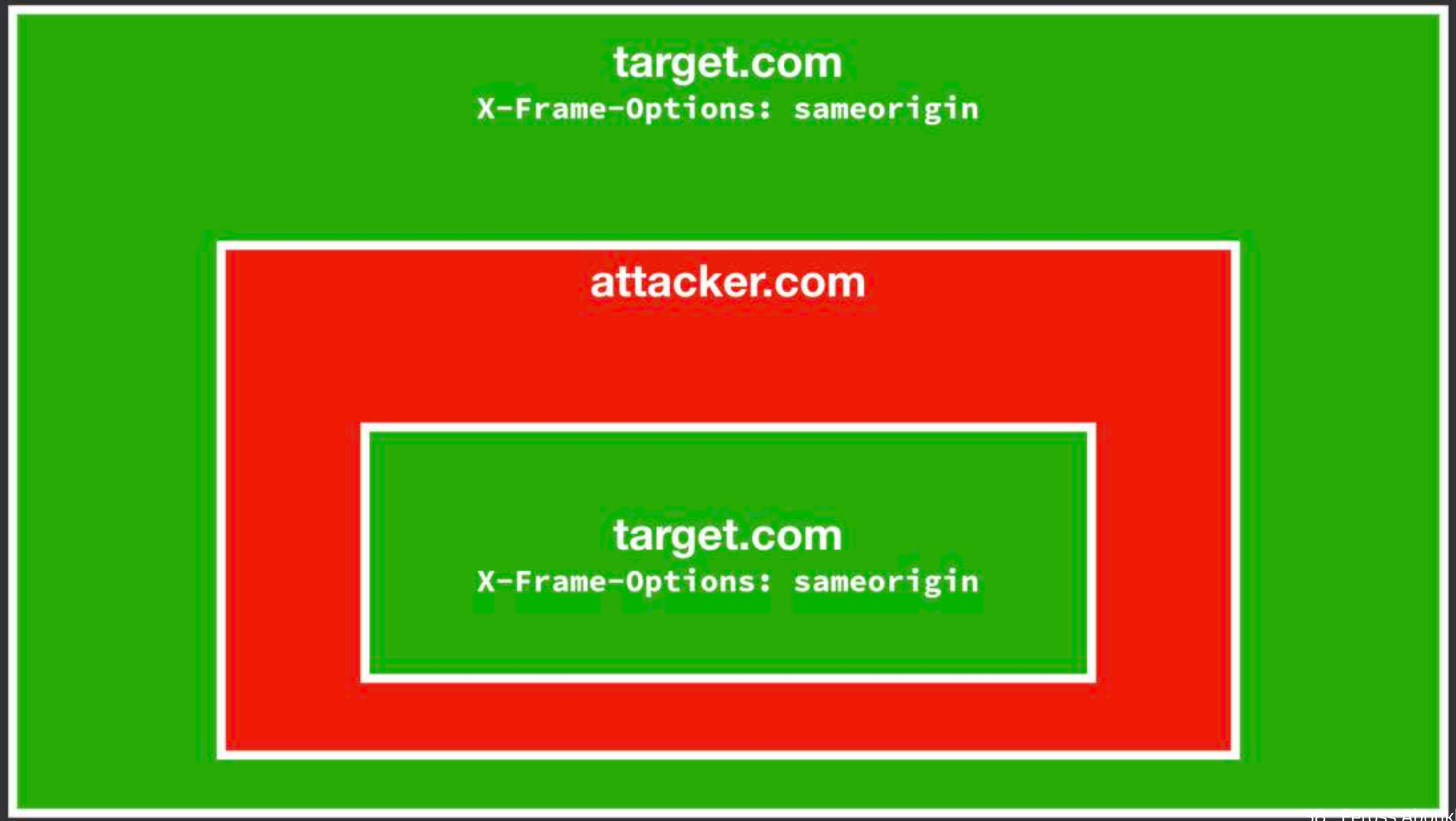

target.com
X-Frame-Options: sameorigin
attacker.com
target.com
X-Frame-Options: sameorigin

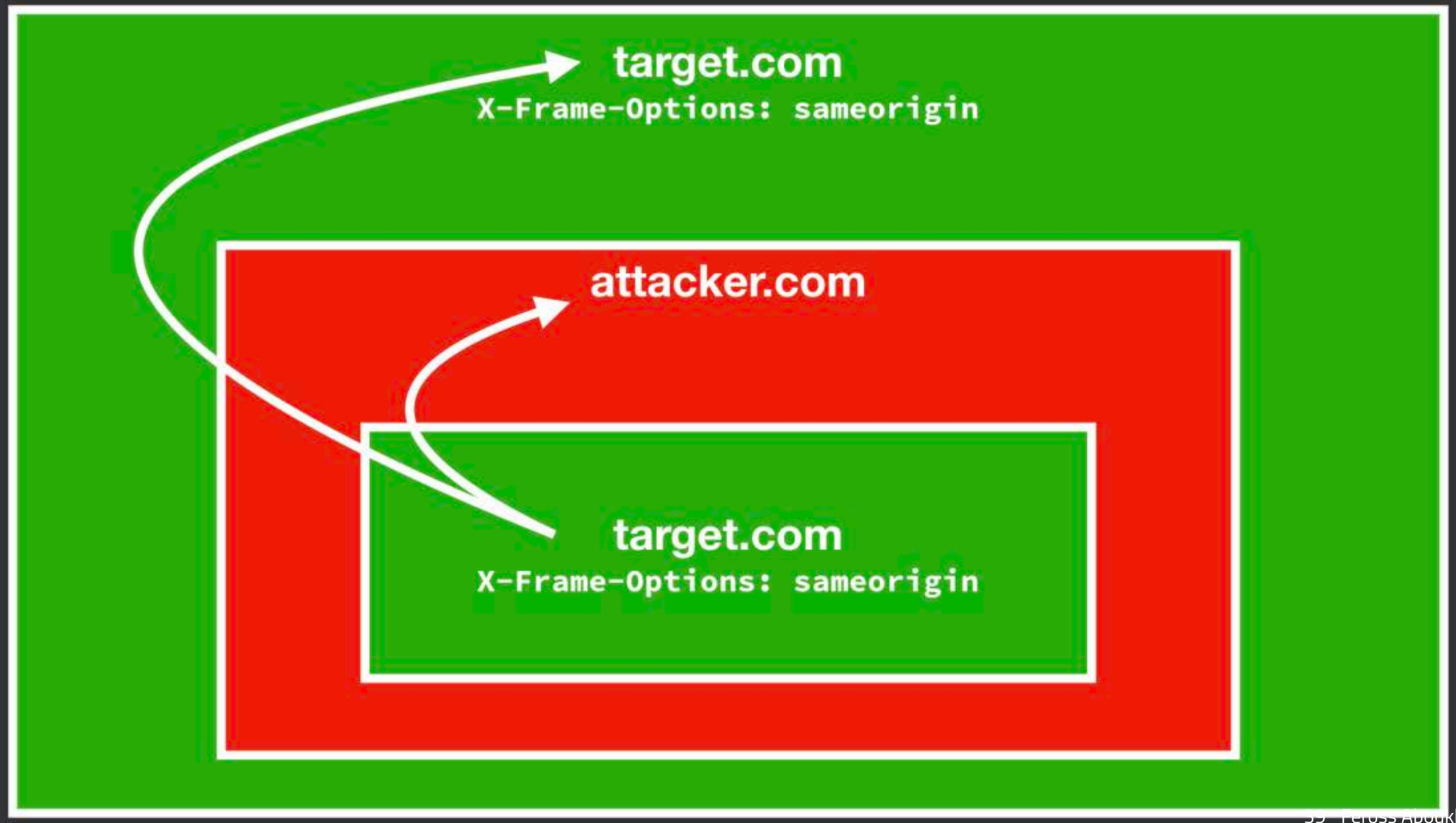

target.com
X-Frame-Options: sameorigin
attacker.com
target.com
X-Frame-Options: sameorigin

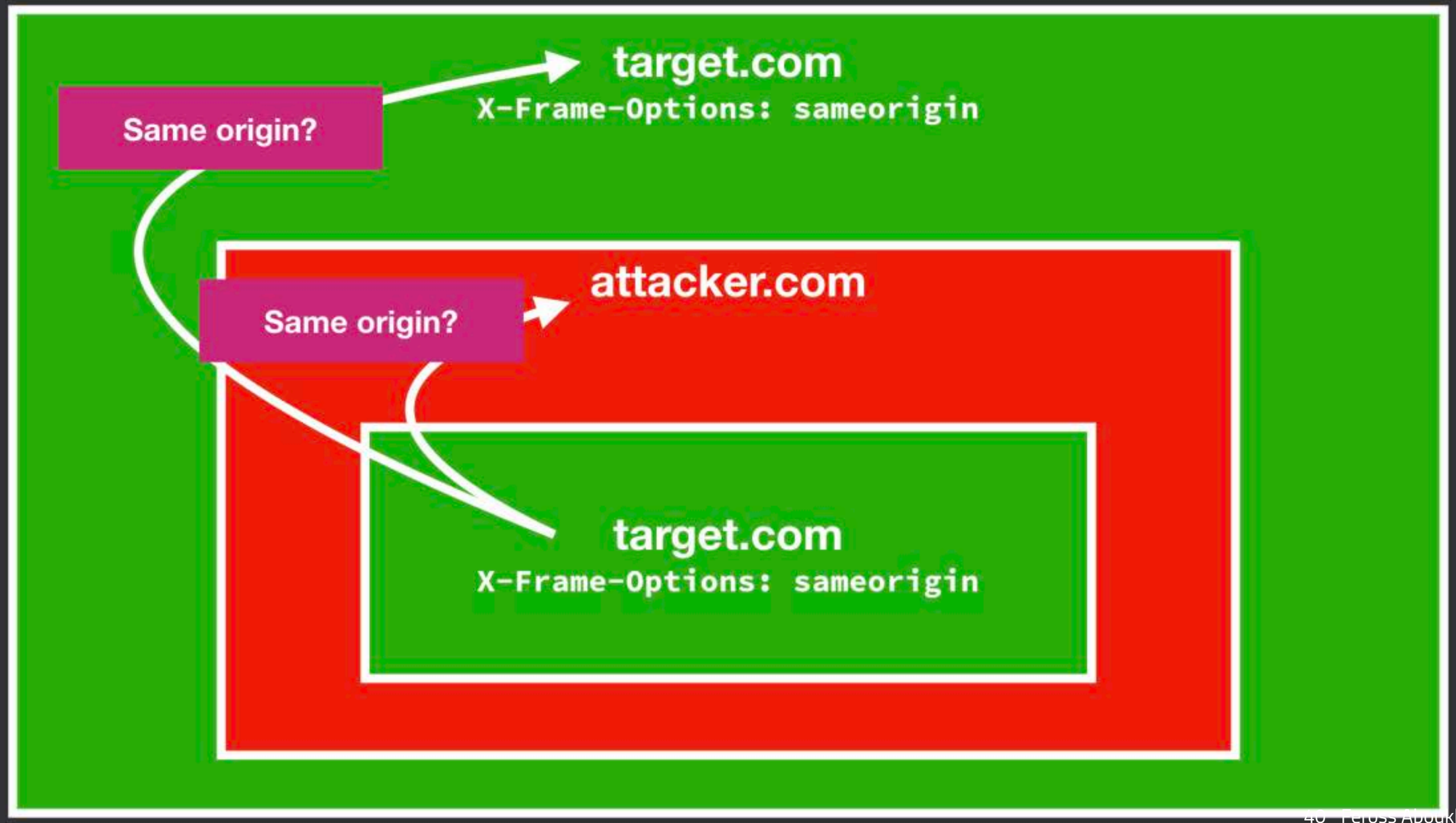

target.com
X-Frame-Options: sameorigin
Same origin?
attacker.com
target.com
X-Frame-Options: sameorigin

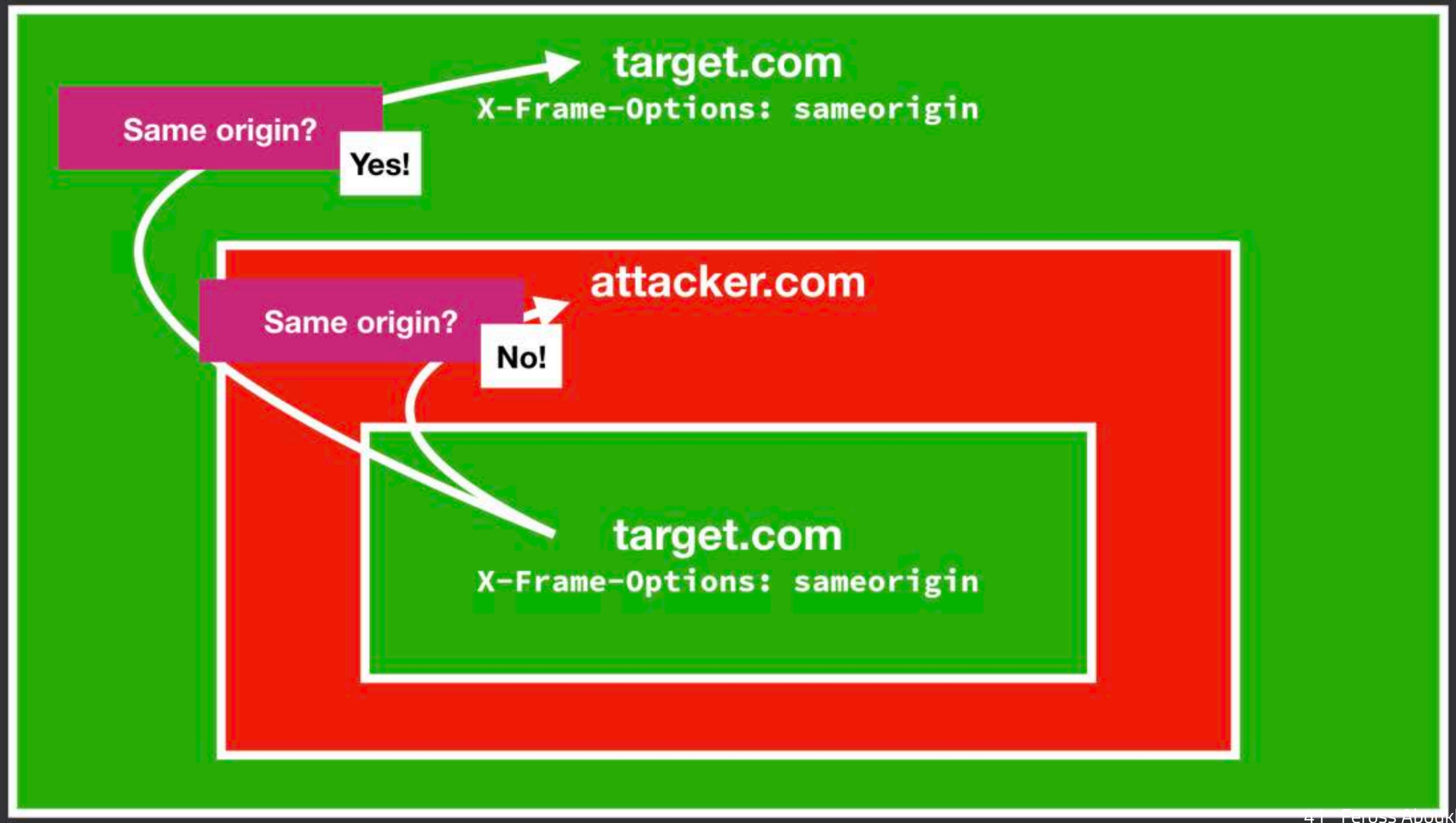

target.com
X-Frame-Options: sameorigin
Same origin?
Yes!
attacker.com
target.com
X-Frame-Options: sameorigin

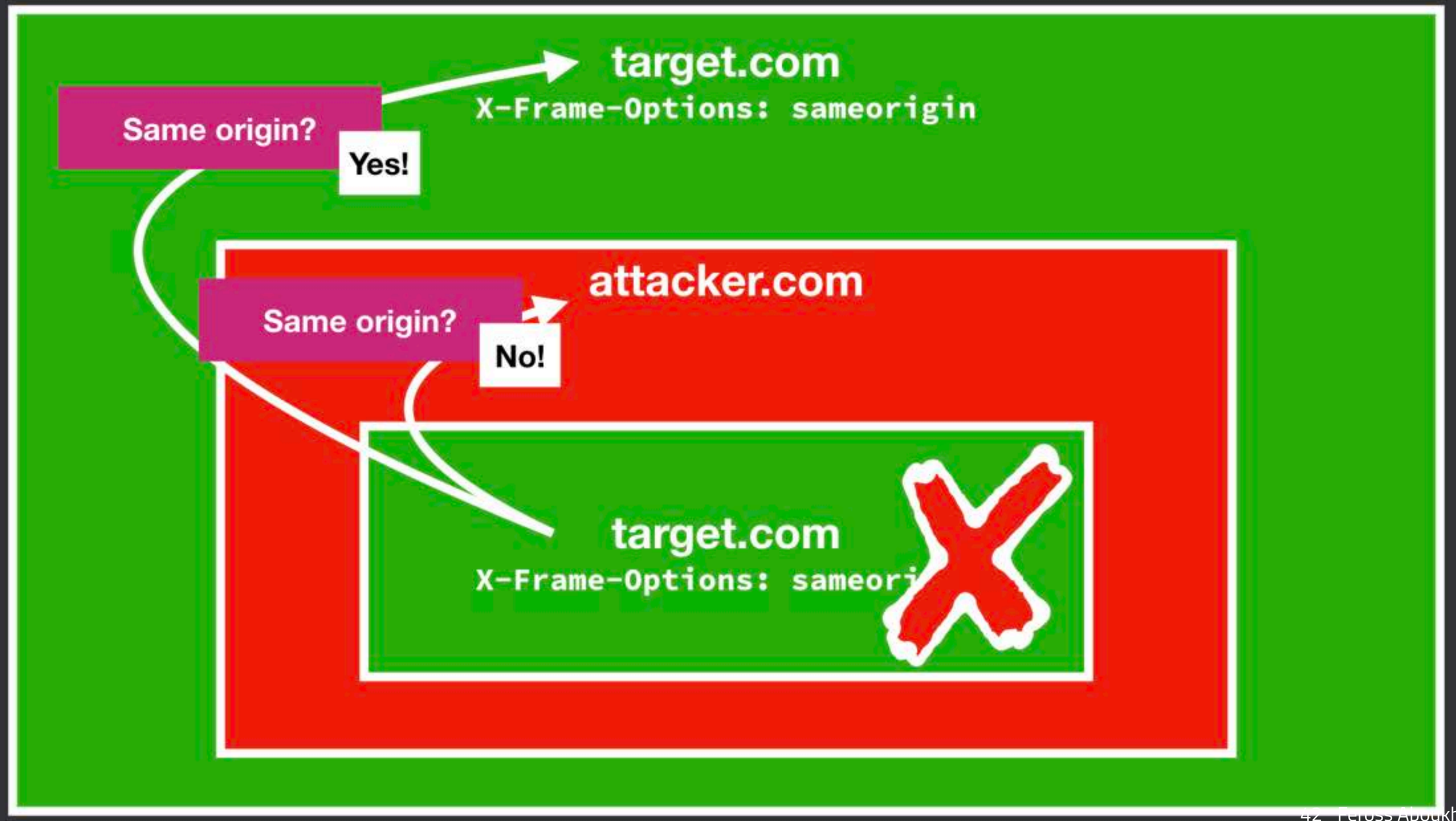

target.com
X-Frame-Options: sameorigin
Same origin?
Yes!
attacker.com
target.com
X-Frame-Options: same


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

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

- Why do this?
  - Prevent cross-site request forgery (CSRF)
- 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**

Typical cross-origin 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>
```

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?
  - Detect **`Referer`** header, use an allowlist (not foolproof)

What does Same Origin Policy allow?

- 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!***
  - Middle ground: **`SameSite`** prevents cross-origin cookie inclusion while still allowing form submission
- 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! (See following slides)***

Relaxing the Same Origin Policy

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://axess.stanford.edu/transcript.pdf')
const data = await res.body.arrayBuffer()
console.log(data)
```

Is site A allowed to read data from site B?

- 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 }
```

Problem: How to access data from 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?

Use `<script>` for cross-origin 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 }
```

- Questions:
  - Is this valid JavaScript?
  - Can the data be accessed by **site-a.com**?

JSON with Padding (JSONP)

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>
```

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

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

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
  - Need to sanitize user-provided callback argument (see "reflected file download attack")
- From **site-b.com**'s perspective:
  - Only want to get data from **site-a.com**, but need to give **site-a.com** the ability to run arbitrary JavaScript – yikes!

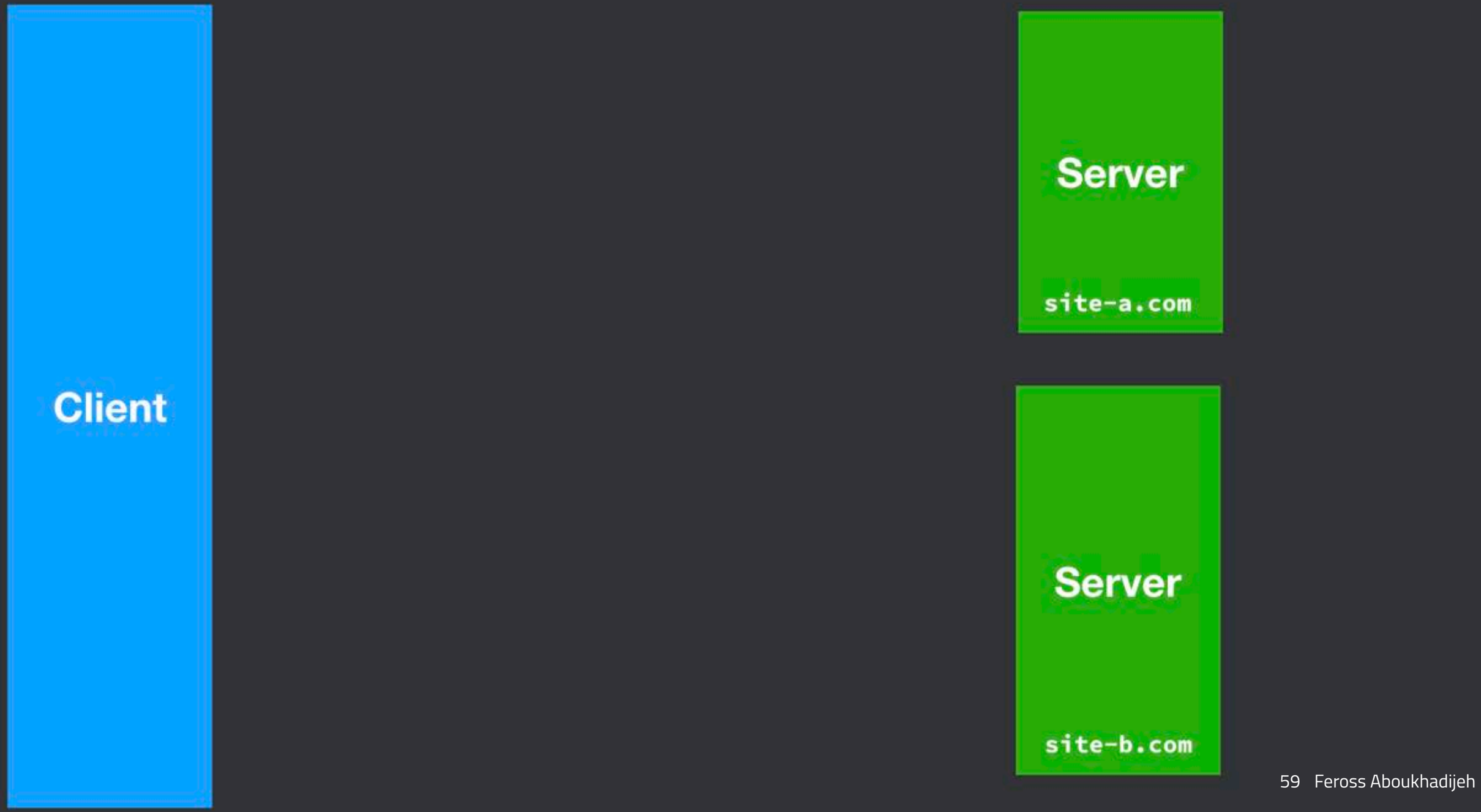
Cross-Origin Resource Sharing (CORS)

- Allow origin **https://site-a.com** to read data:

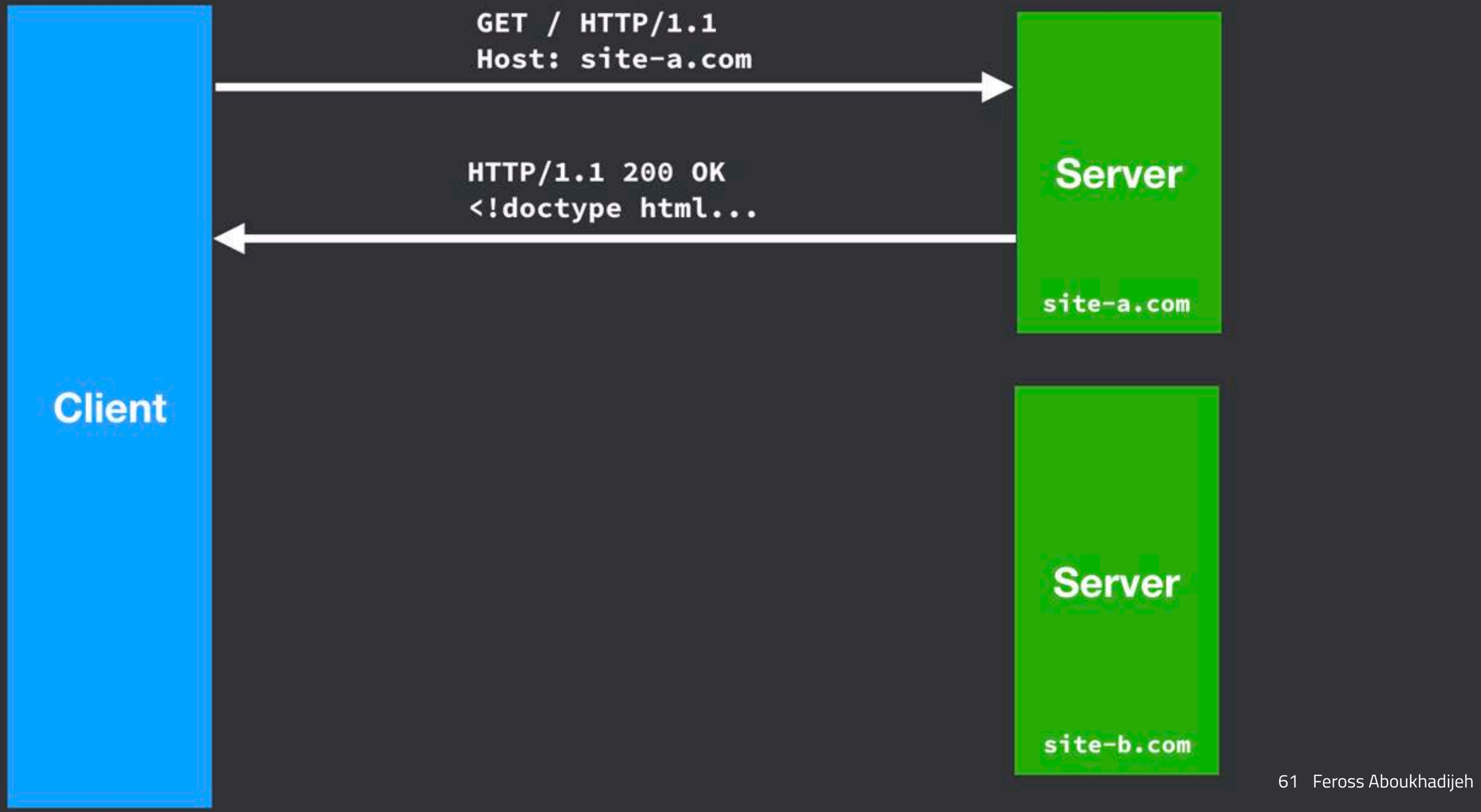
```
Access-Control-Allow-Origin: https://site-a.com
```

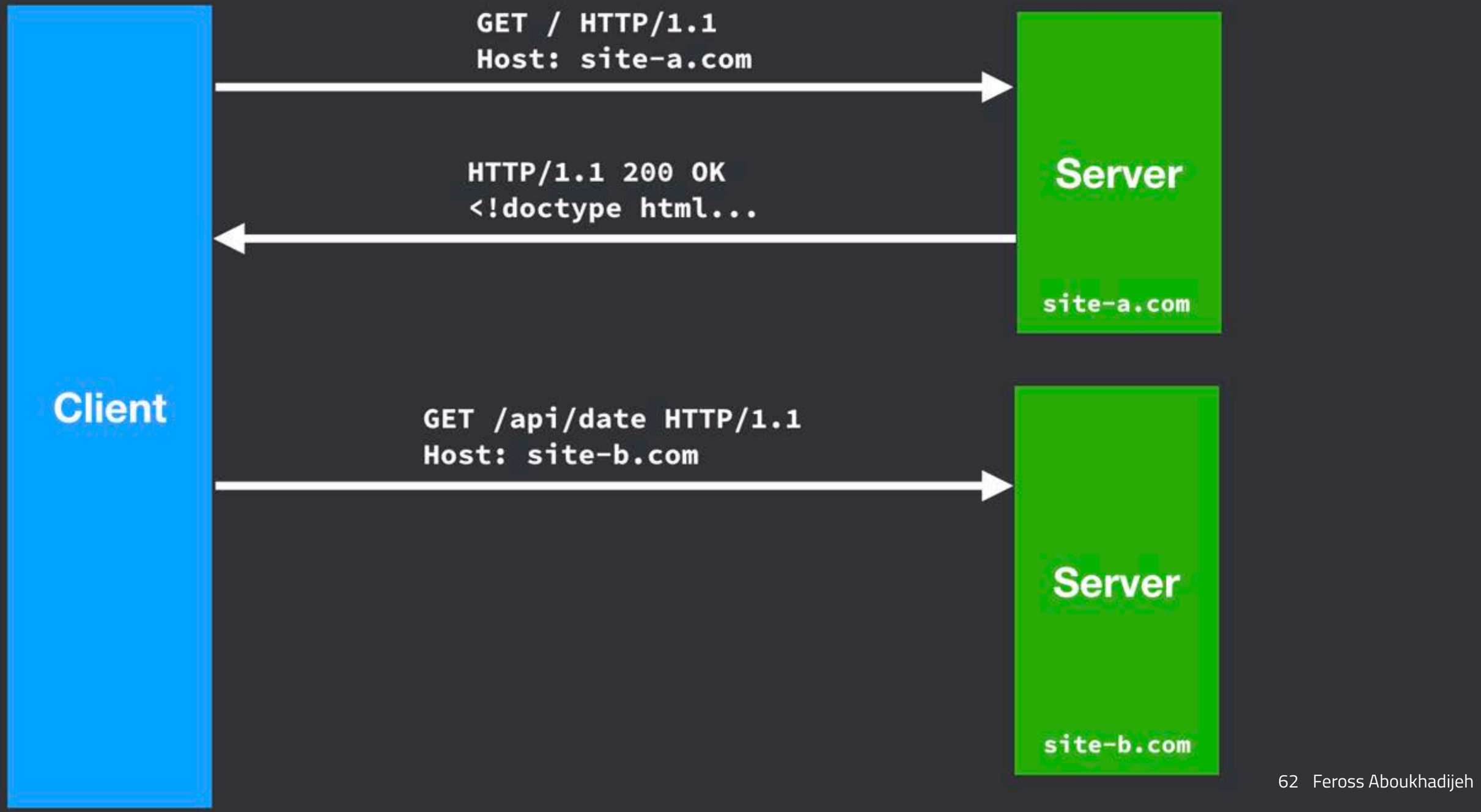
- Allow any origin to read data:

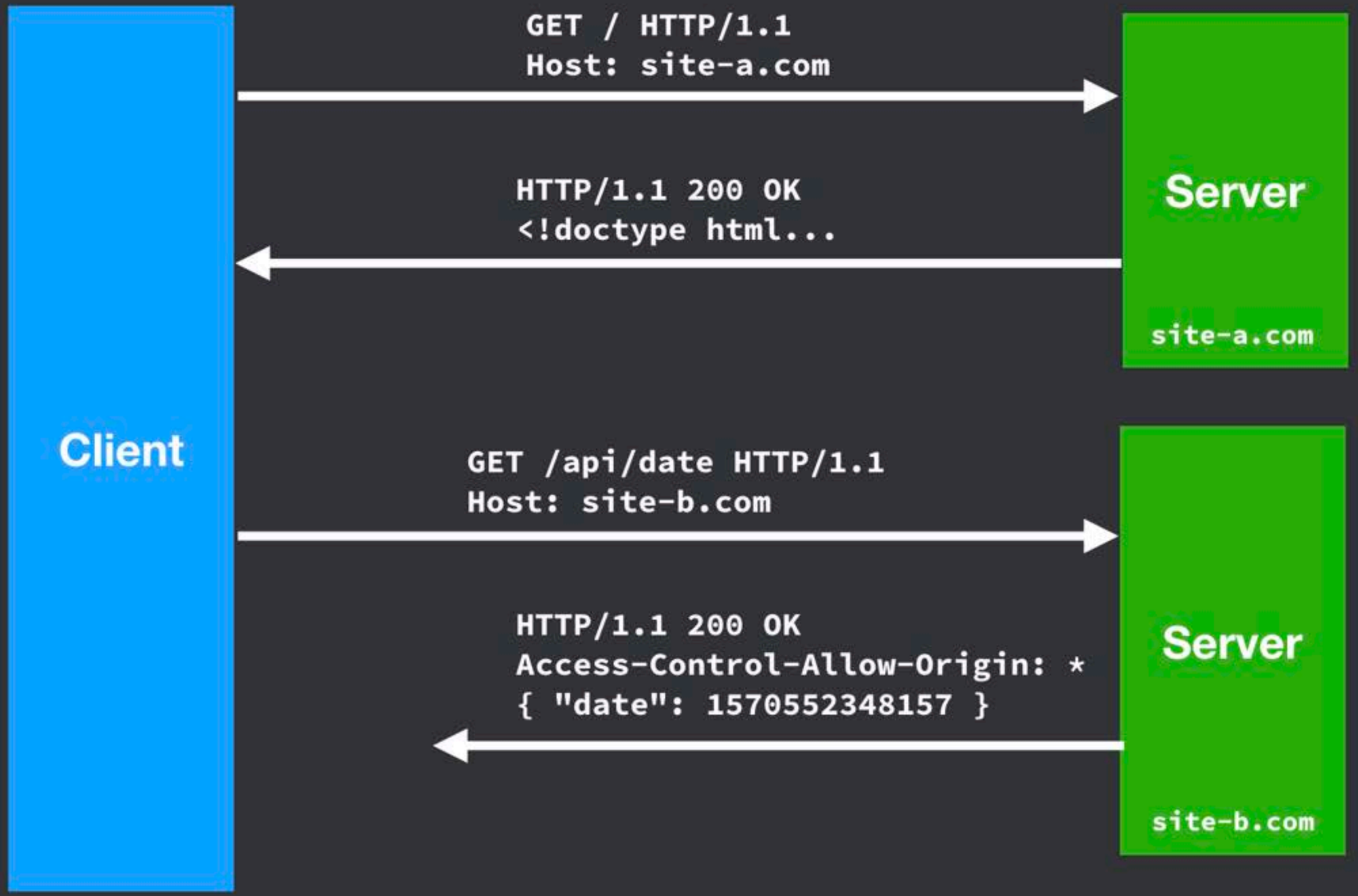
```
Access-Control-Allow-Origin: *
```

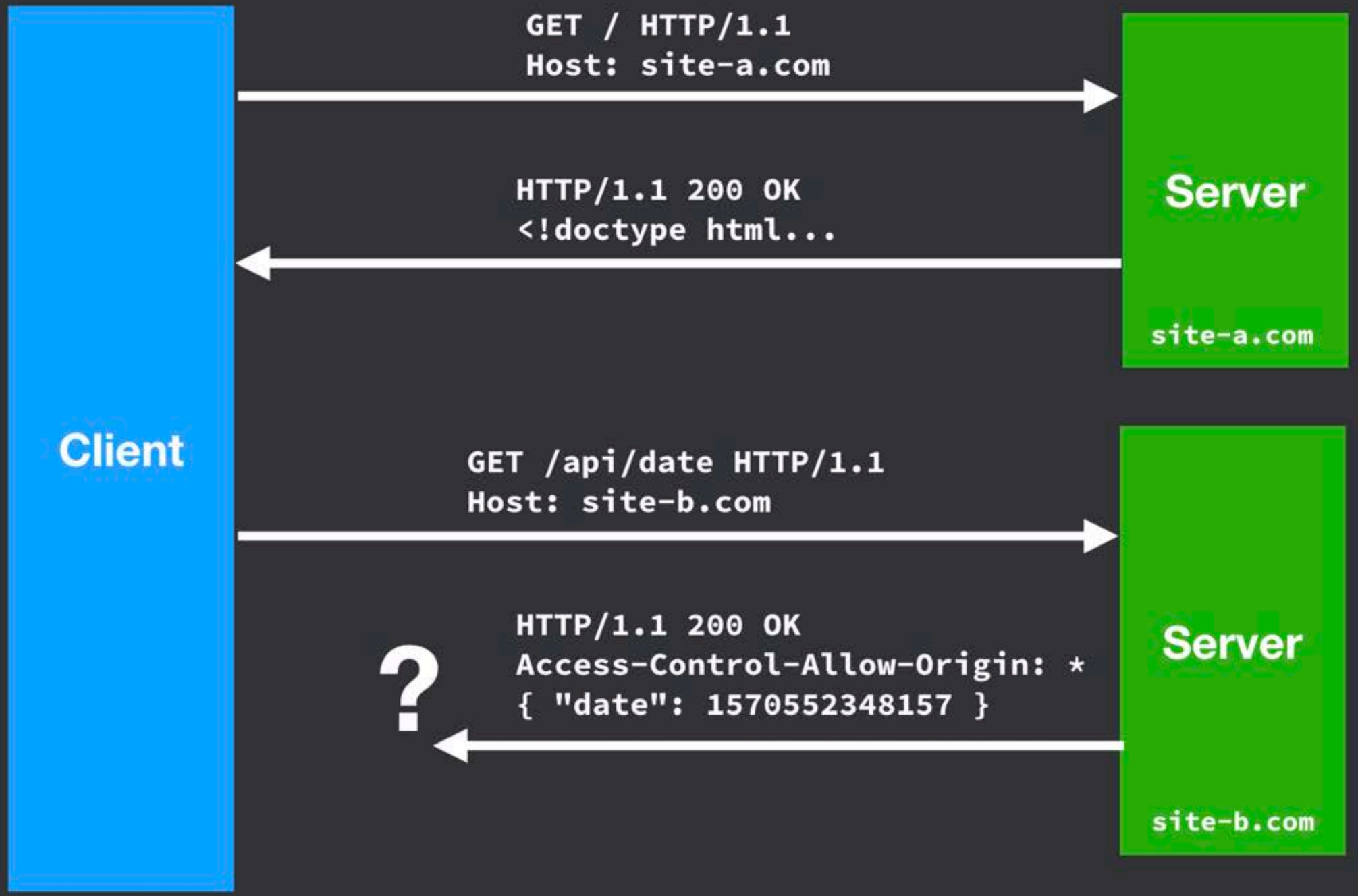

Client
Server
site-a.com
Server
site-b.com

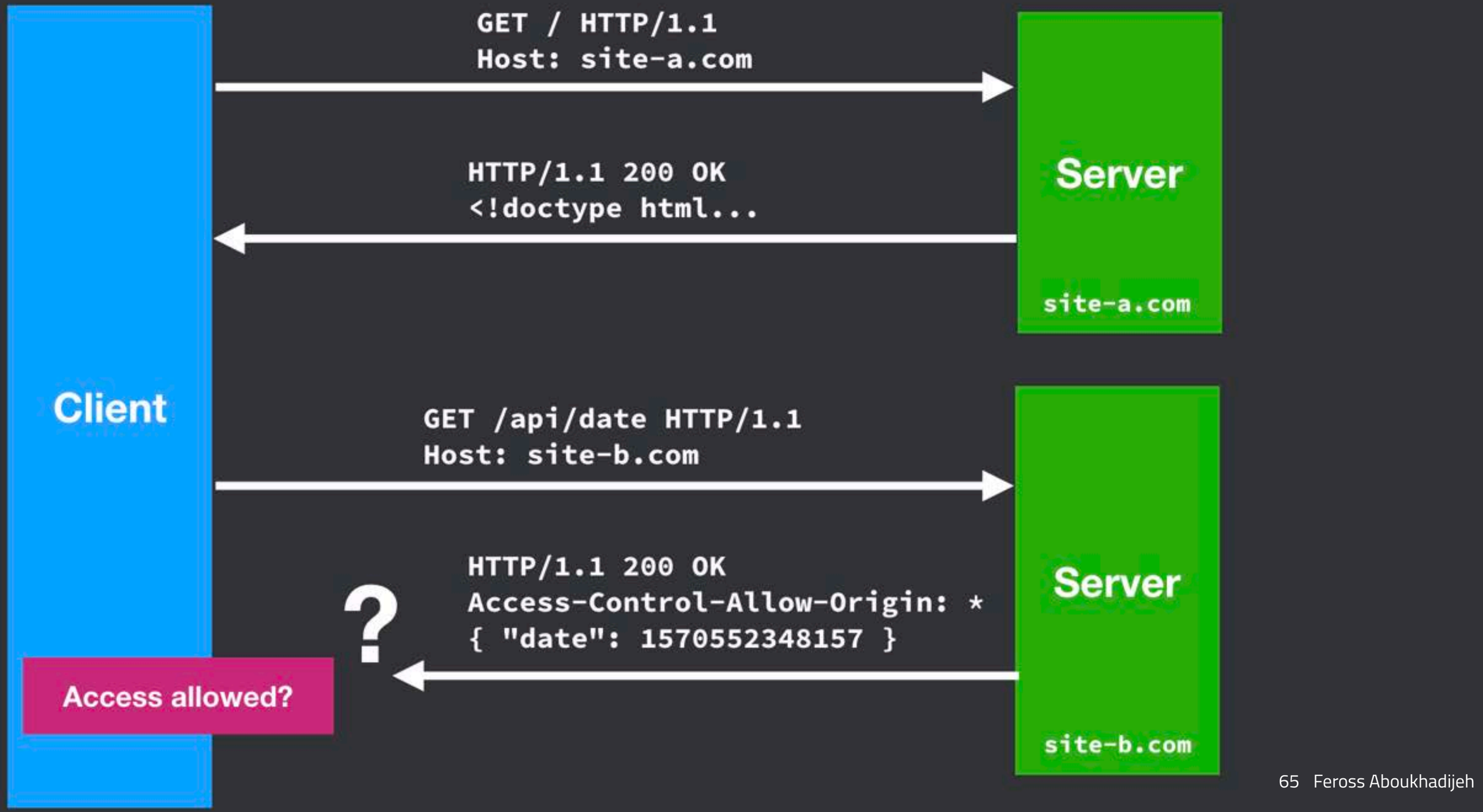

GET / HTTP/1.1
Host: site-a.com
Client
Server
site-a.com
Server
site-b.com

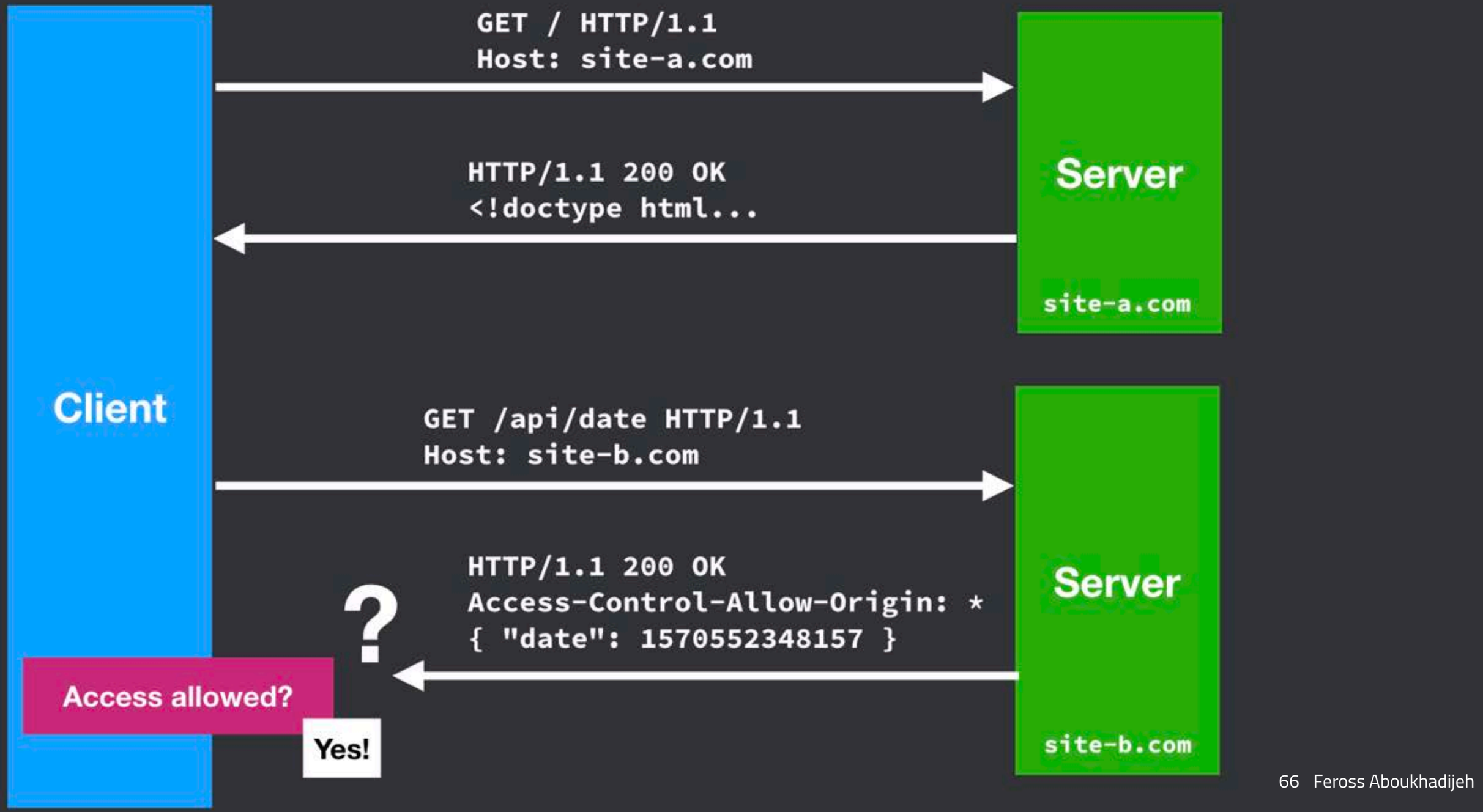

GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Client
Server
site-a.com
Server
site-b.com

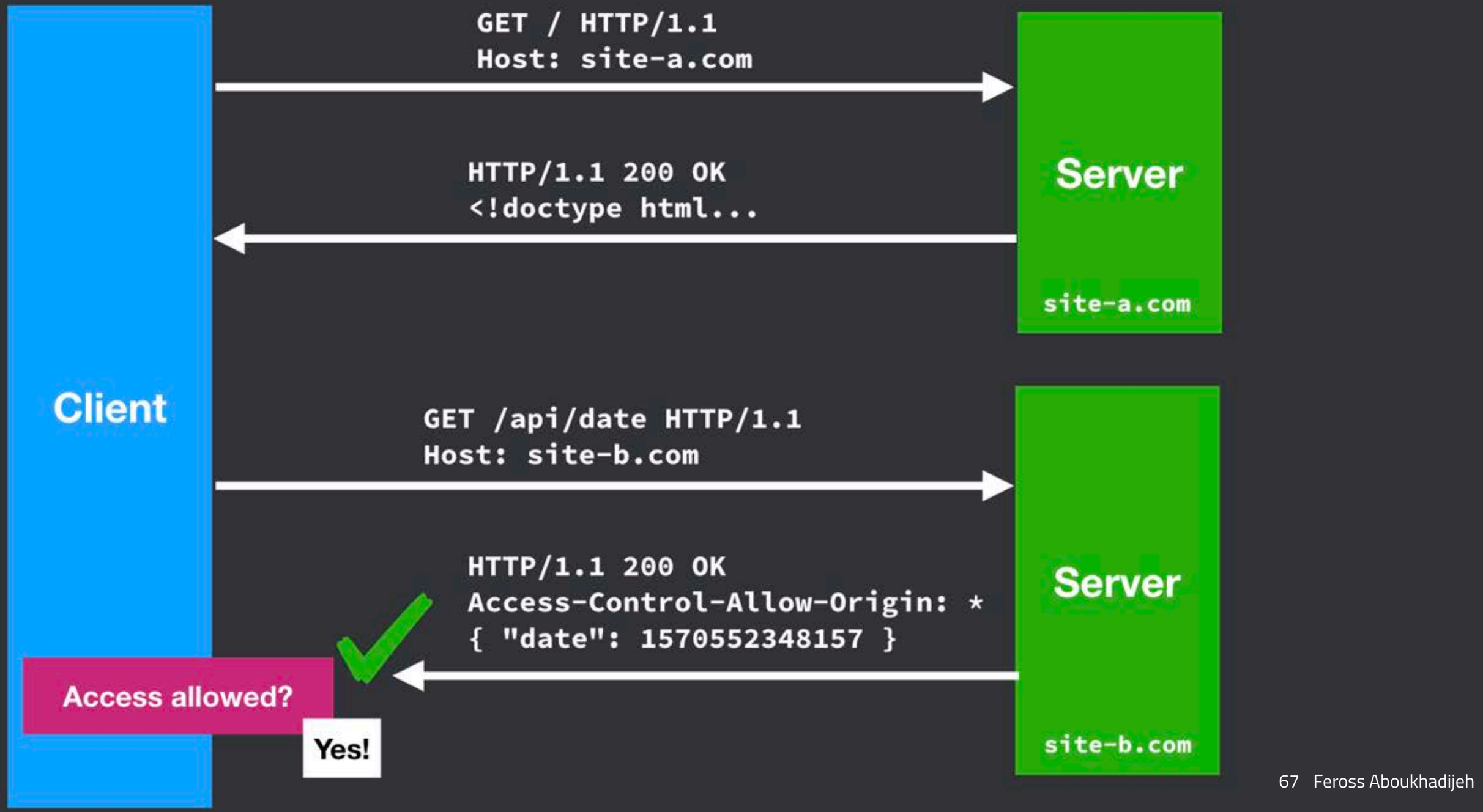

Client
GET / HTTP/1.1
Host: site-a.com
HTTP/1.1 200 OK
<!doctype html...
Server
site-a.com
GET /api/date HTTP/1.1
Host: site-b.com
Server
site-b.com


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


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


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?


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!


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!

Final version: Date API server

Server code:

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

Server response:

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

How does a site ensure private data returned by an authenticated API route isn't read by other sites?

Would this work?

- 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?*

Cross-Site Script Inclusion (XSSI)

```
function Array () {
  for (let i = 0; i < this.length; i++) {
    console.log(this[i])
  }
}

[1, 2, 3]
```

- Worked against Gmail
- Fixed in every browser

Cross-Site Script Inclusion (XSSI) variant

```
Object.prototype.__defineSetter__('username', function (obj) {
  for(let key in obj) {
    console.log(key + '=' + obj[i])
  }
})

[{ username: "Feross Aboukhadijeh" }]
```

- Worked against Twitter
- Fixed in every browser

Why does Google put)]}' at beginning of all their API responses?

```
)]}'

[[["og.botresp",["<redacted>"]
,null,[null,null,null,"//www.google.com/js/bg/<redacted>.js"]
,"<redacted>"]
,["di",10]
,["af.httprm",20,"<redacted>",0]
,["e",4,null,null,3456]
]]
```

XSSI "magic prefix"

- There are a few variants:
  - `)]}'`
  - `while(1);`
- Ensures that if the returned data would somehow have parsed as valid JavaScript, then "magic prefix" will guarantee it is is a syntax error
- How does the same site avoid the syntax error?
  - Can directly read the data with **`fetch`** API, so can strip out "magic prefix"

Tom in All Posts, Security, Web | March 4, 2019

Facebook exploit – Confirm website visitor identities

> ***Short version:***
>
> *I discovered a bug that would let any web page identify a logged in FB user by confirming their ID. Facebook fixed in 6-9 months and rewarded a $1000 bounty.*
>
> - *Hacker News Discussion*

In last years coverage of the Facebook / Cambridge Analytica privacy concerns, Mark Zuckerberg was asked to testify before Congress, and one of the questions they asked was around whether Facebook could track users even on other websites. There was a lot of news coverage around this aspect of Facebook, and a lot of people were up in arms. As one aspect of their response, Facebook launched a Data Abuse Bounty, with the aim of protecting user data from abuse.

So, having recently found a bug in Google's search engine, I set out to see whether ***I*** could track or identify Facebook users when they were on other sites. After a few false starts, I managed to find a bug which **allows me to identify whether a visitor is logged in to a *specific* Facebook** account, and can check hundreds of identities per second (in the range of 500 p/s).

I have created a proof of concept of the attack (now fixed), which checks both a small known list of IDs but also allows you to enter an ID and it will confirm whether you are logged in to that account or not.

END

Credits

https://www.owasp.org/images/6/6a/OWASPLondon20161124*JSON*Hijacking*Gareth*Heyes.pdf

https://www.tomanthony.co.uk/blog/facebook-bug-confirm-user-identities/