



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Server Side Security and Safe Coding

Lecture 10

Trawsnewid Addysg; Trawsnewid Bywydau
Transforming Education; Transforming Lives



www.uwtsd.ac.uk

Learning Outcomes

- **Explain how CSRF attacks occur**, including how cookies, ambient authority, and automatic credential attachment enable unauthorized actions.
- **Compare CSRF mitigation techniques**, specifically SameSite cookies and CSRF tokens, and explain how each prevents cross-site attacks.
- **Identify framework-level security risks** by analyzing the GitHub OAuth CSRF bypass and the impact of HTTP method handling and abstractions.
- **Apply secure coding practices** such as explicit method checks, defensive programming, and minimizing leaky or overly complex abstractions.

Recall: Cross-Site Request Forgery (CSRF)

- Idea: Force users 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

Client
victim.com

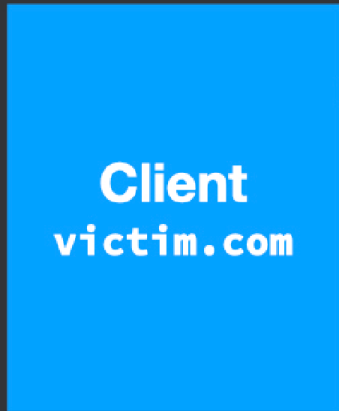
Server
victim.com

Client
victim.com

POST /login HTTP/1.1
username=alice&password=hunter2

Server
victim.com



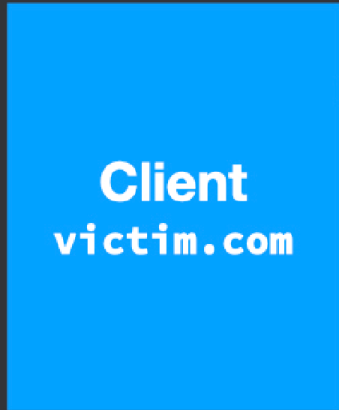


POST /login HTTP/1.1
username=alice&password=hunter2

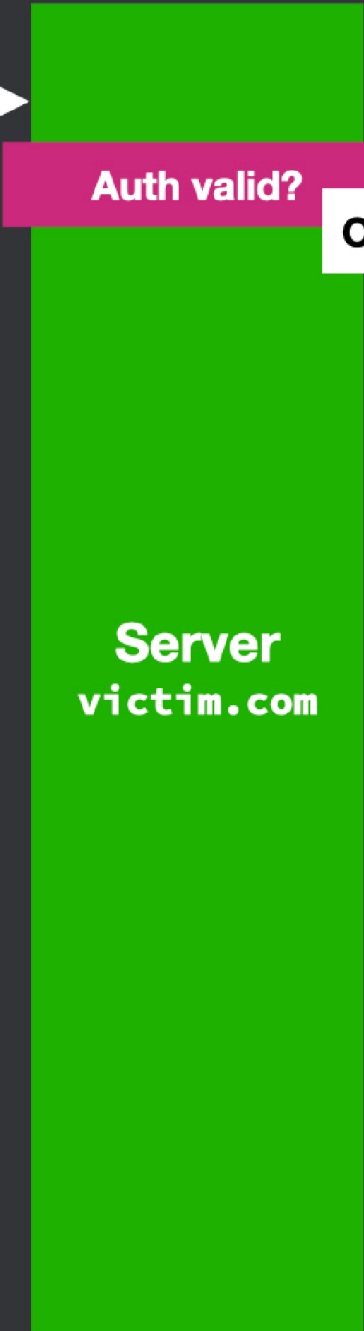


Auth valid?



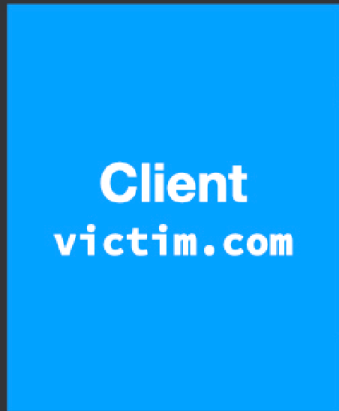


POST /login HTTP/1.1
username=alice&password=hunter2



Auth valid?

OK!



POST /login HTTP/1.1
username=alice&password=hunter2

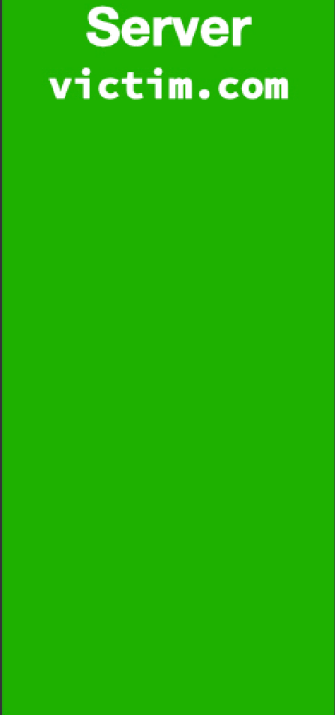


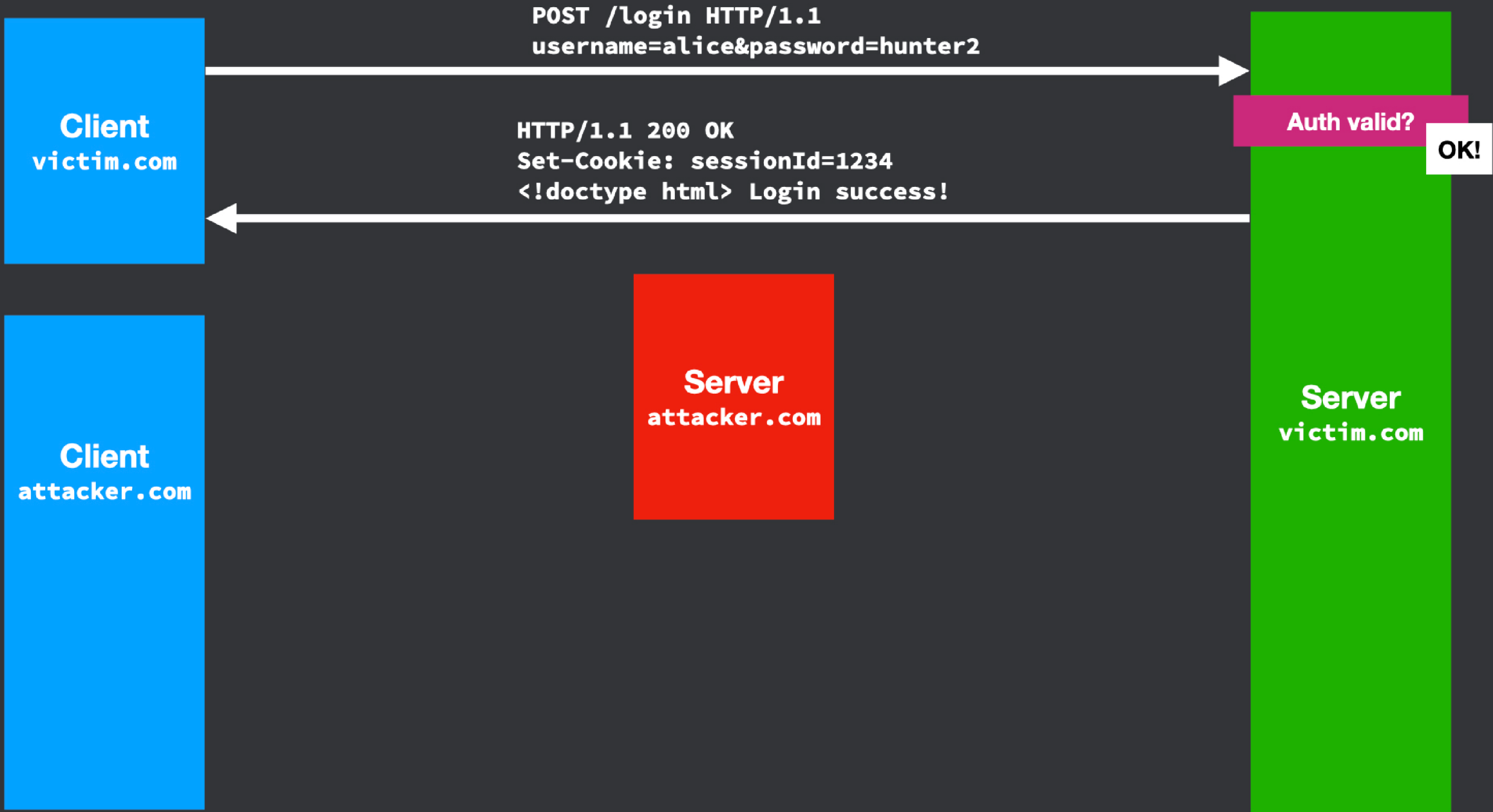
HTTP/1.1 200 OK
Set-Cookie: sessionId=1234
<!doctype html> Login success!



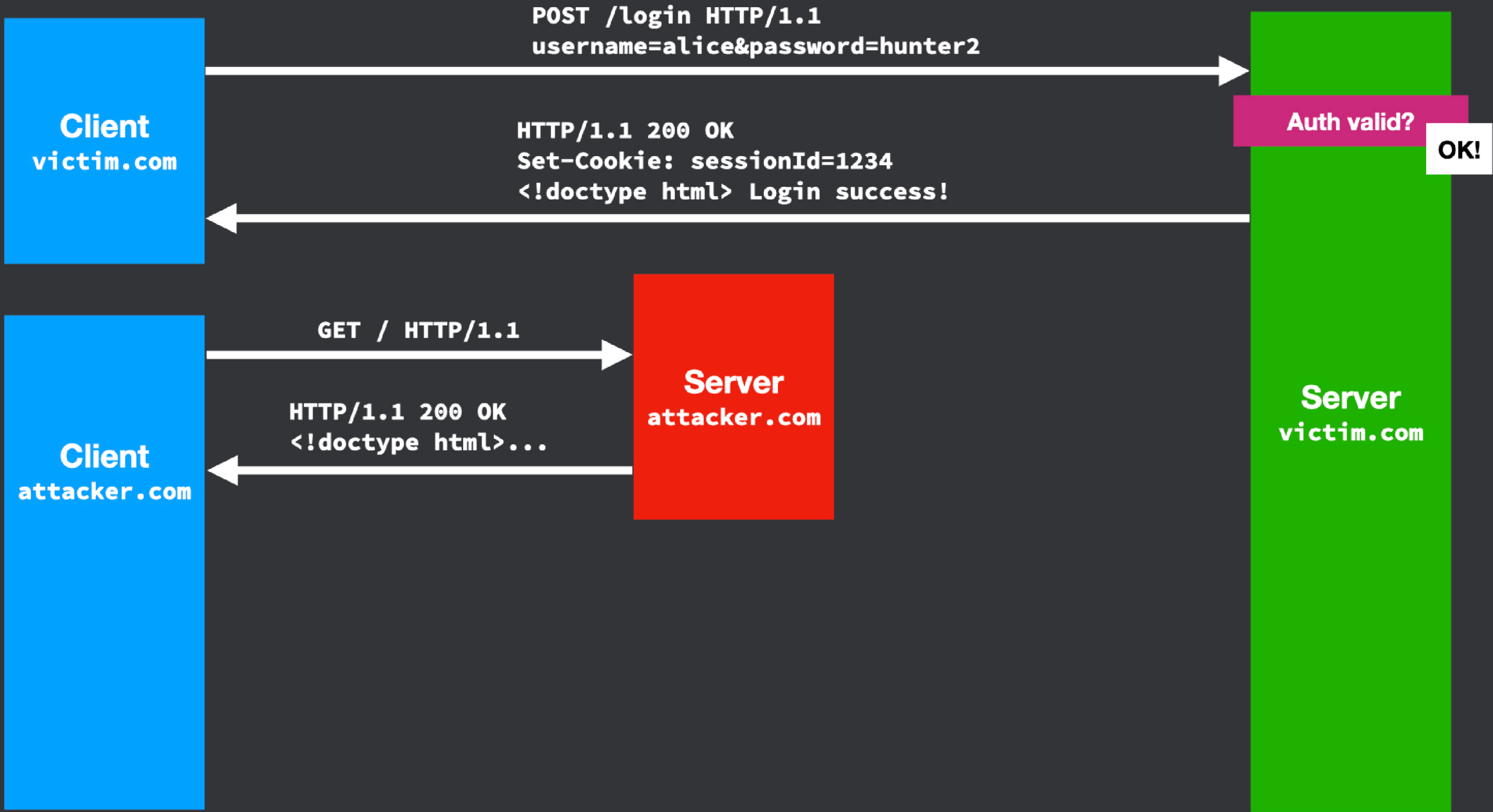
Auth valid?

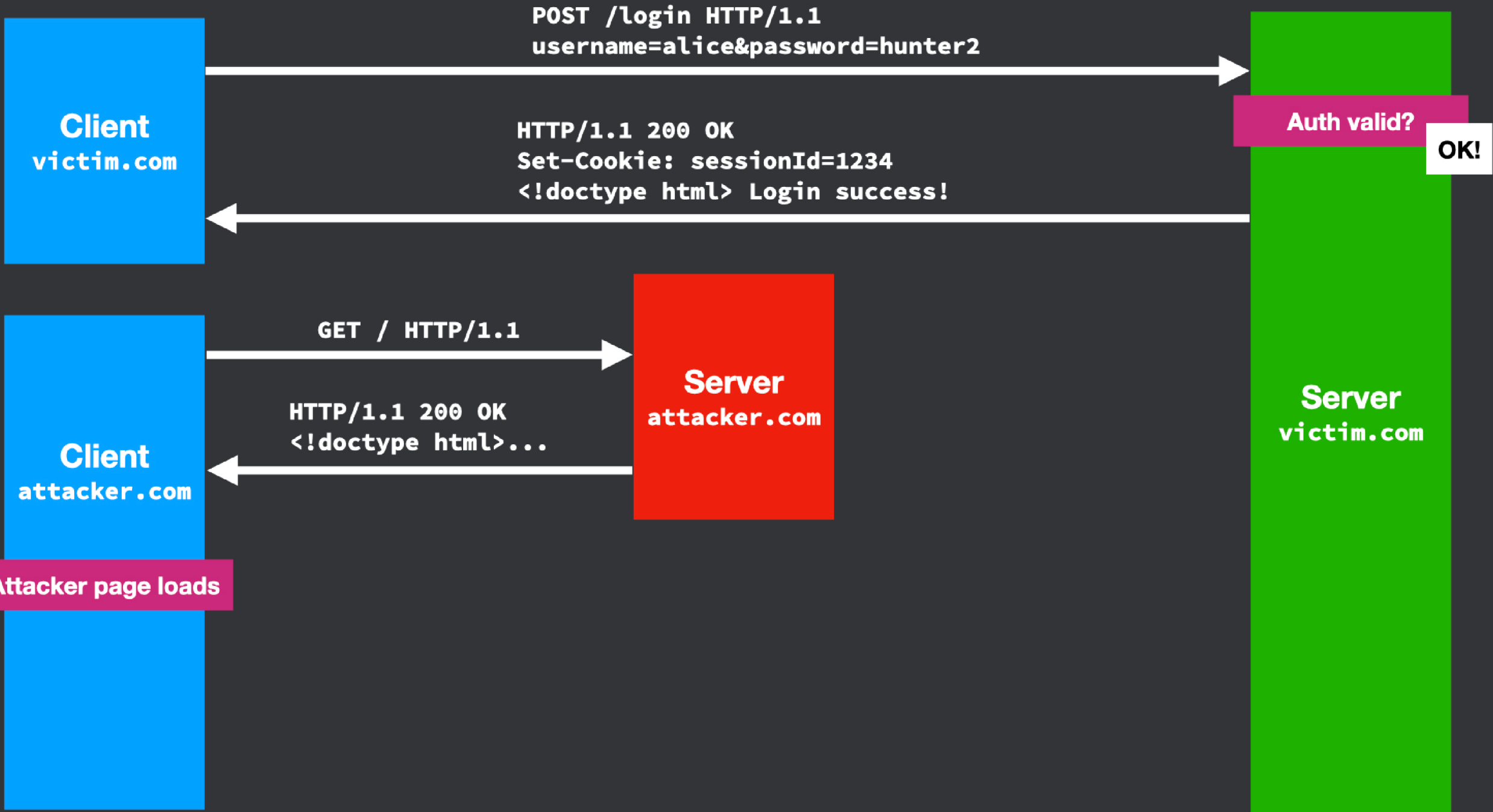
OK!















Recall: SameSite cookies

- Use the SameSite cookie attribute to prevent cookies 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='MzNjNGM5NmQtYzRjOS00NTEy' />`
- 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

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...

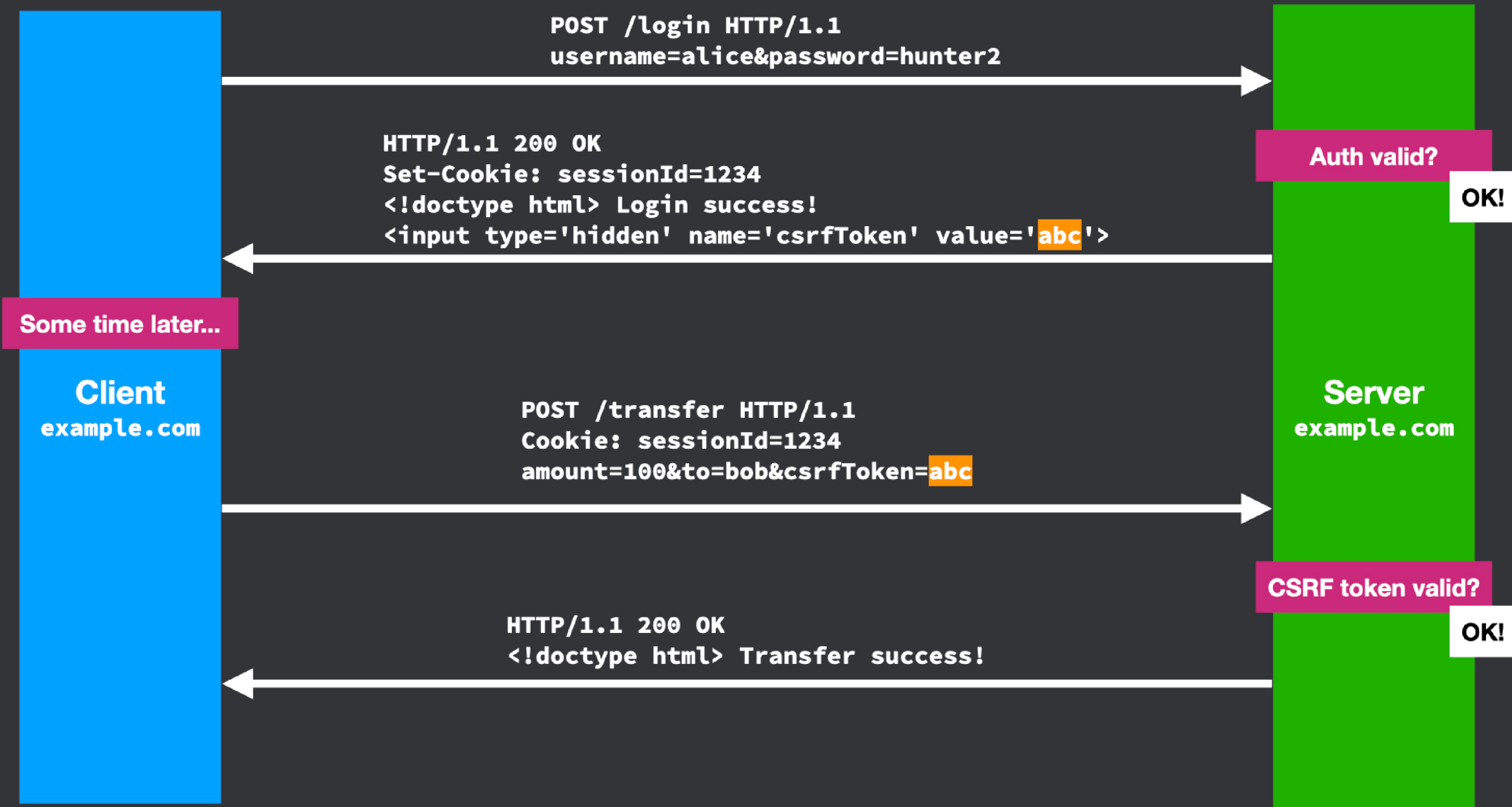
Client
example.com

POST /transfer HTTP/1.1
Cookie: sessionId=1234
amount=100&to=bob&csrfToken=abc

Server
example.com

CSRF token valid?

OK!



How a CSRF token works against an attacker

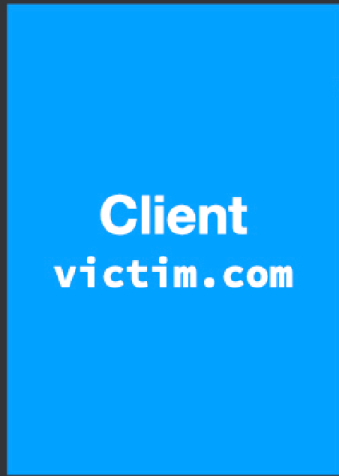


POST /login HTTP/1.1
username=alice&password=hunter2

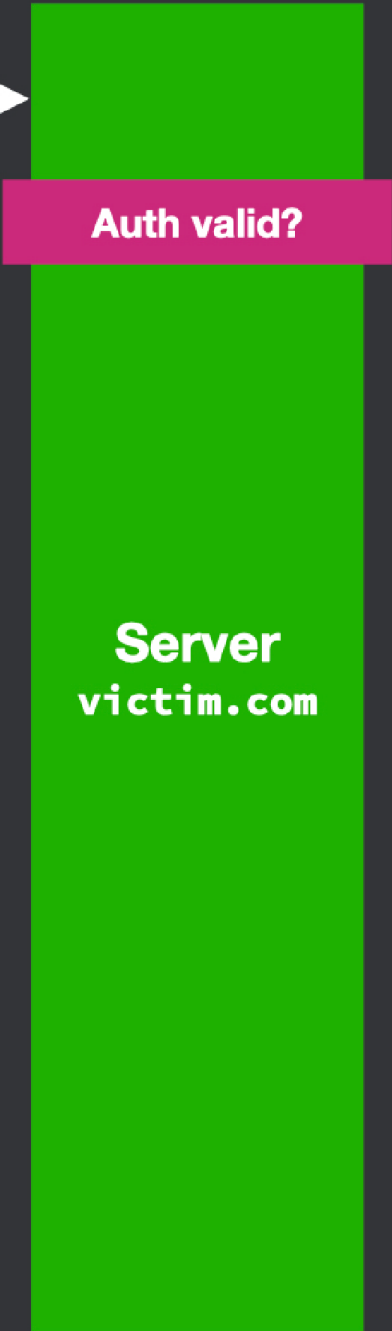
Client
victim.com

Server
victim.com





POST /login HTTP/1.1
username=alice&password=hunter2





POST /login HTTP/1.1
username=alice&password=hunter2



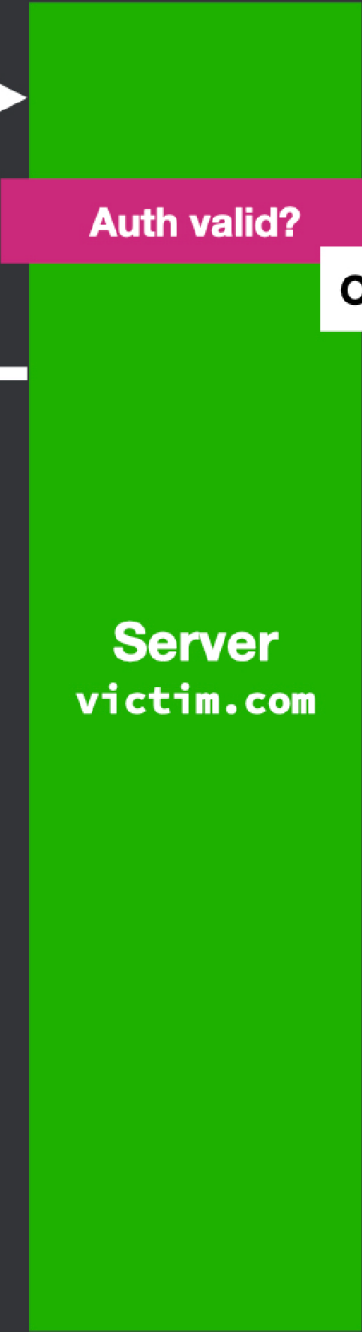
Auth valid?

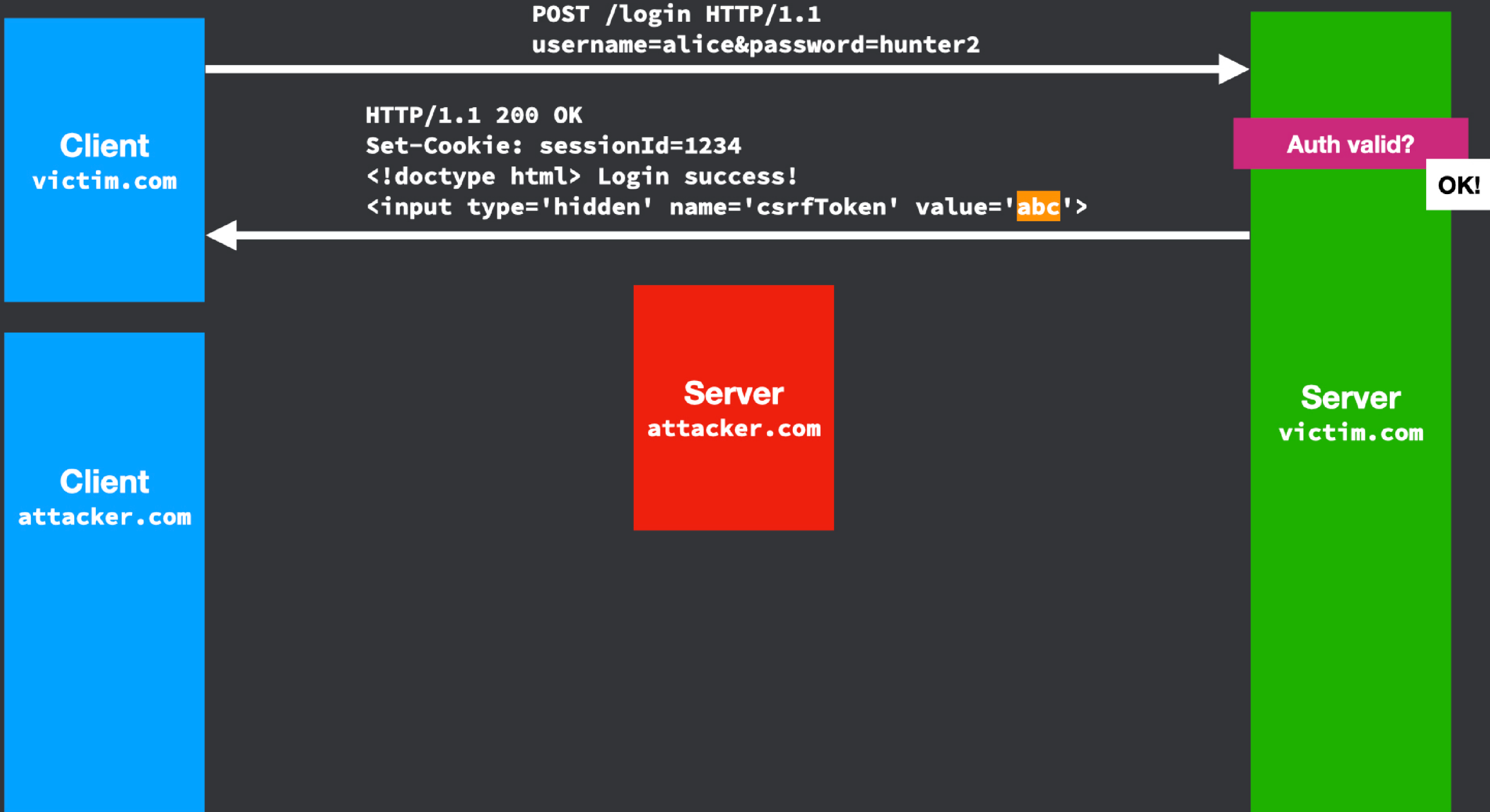
OK!



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'>





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?

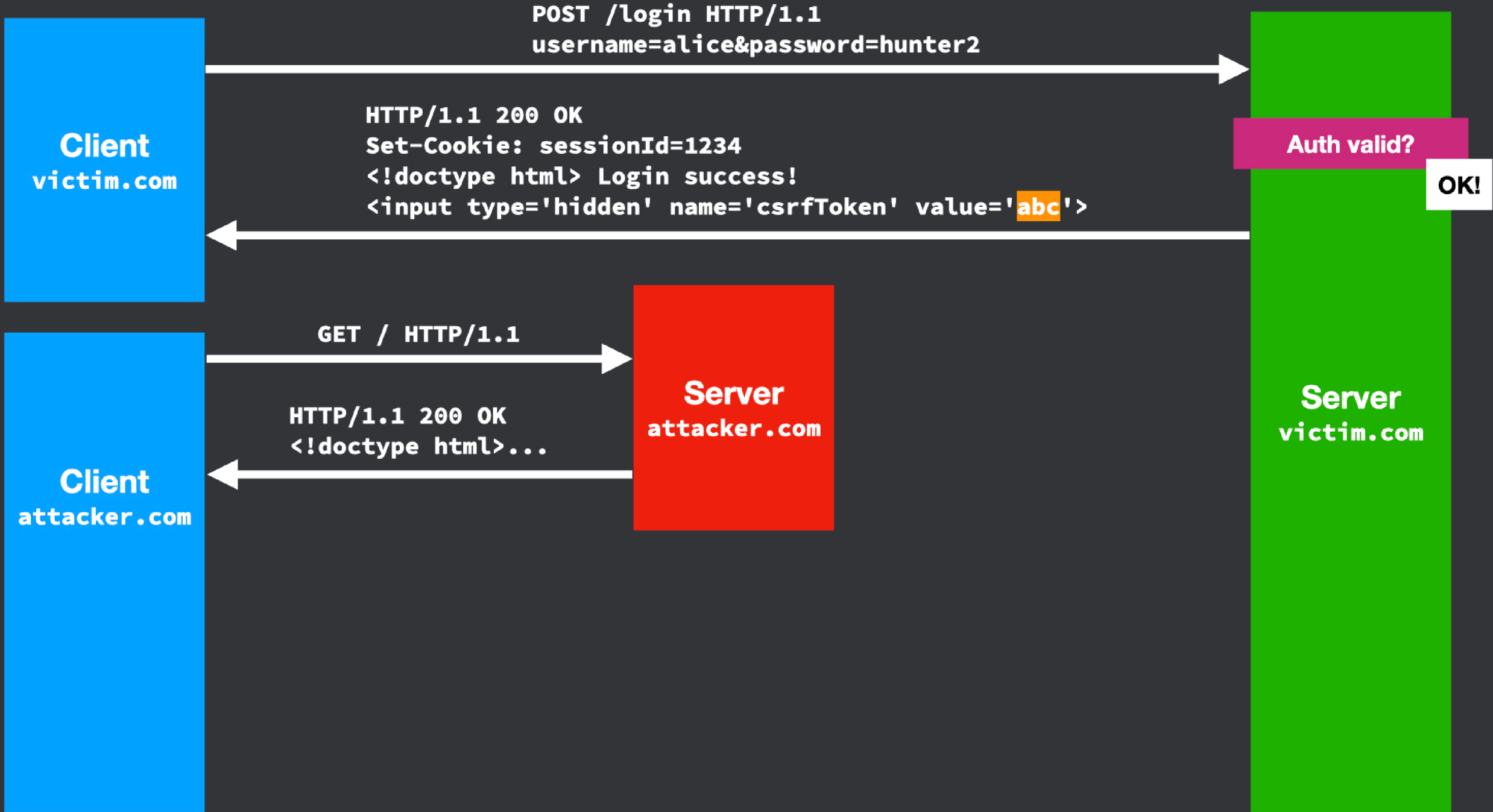
OK!

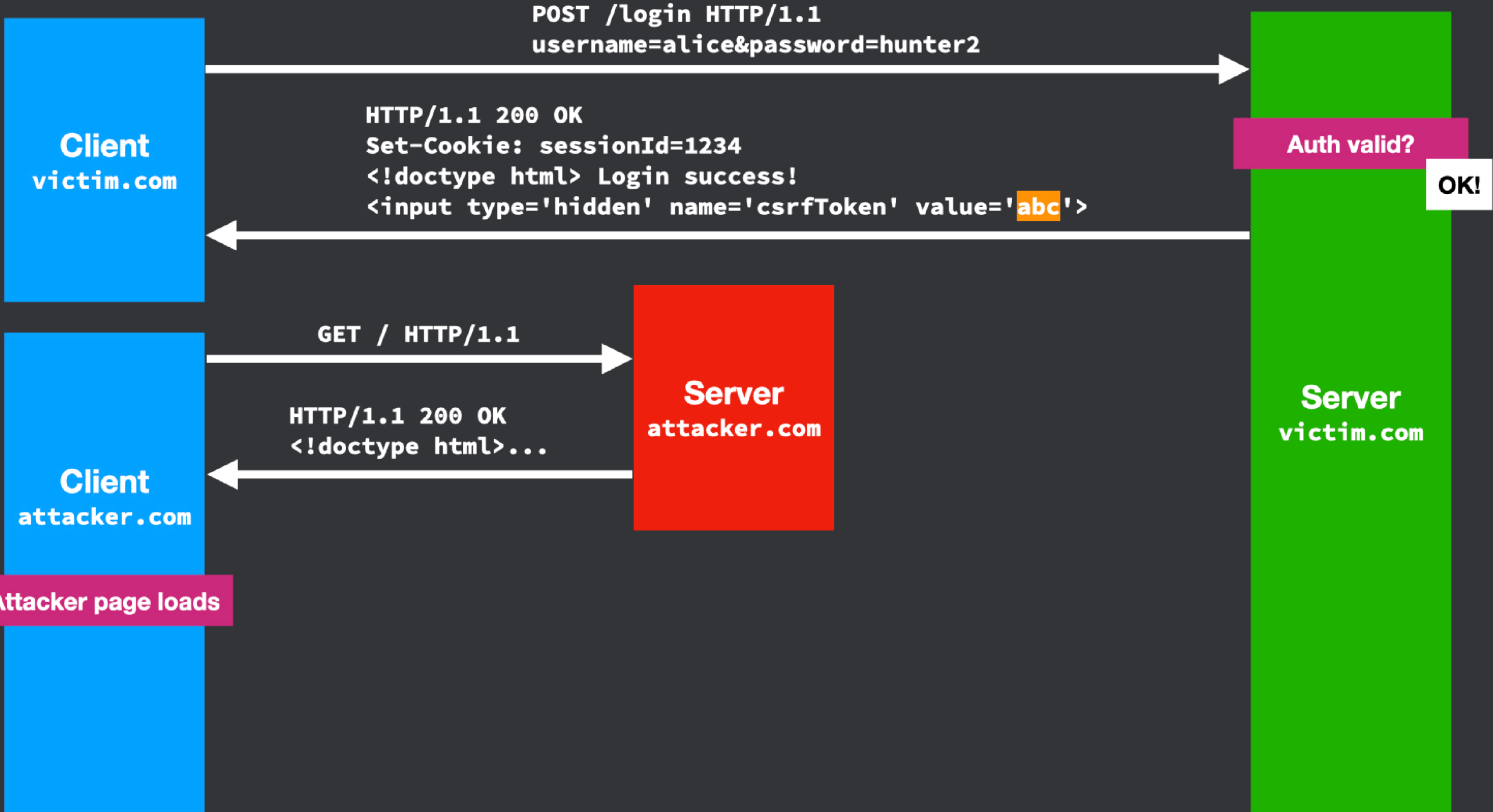
Client
attacker.com

Server
attacker.com

Server
victim.com















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 query string
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 query string, which can be used to access the requested data

How does the "Authorize" button works?

- 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

GET / HTTP/1.1

Server
example.com



Client
example.com

GET / HTTP/1.1

HTTP/1.1 200 OK
<!doctype html> Login with GitHub?

Server
example.com

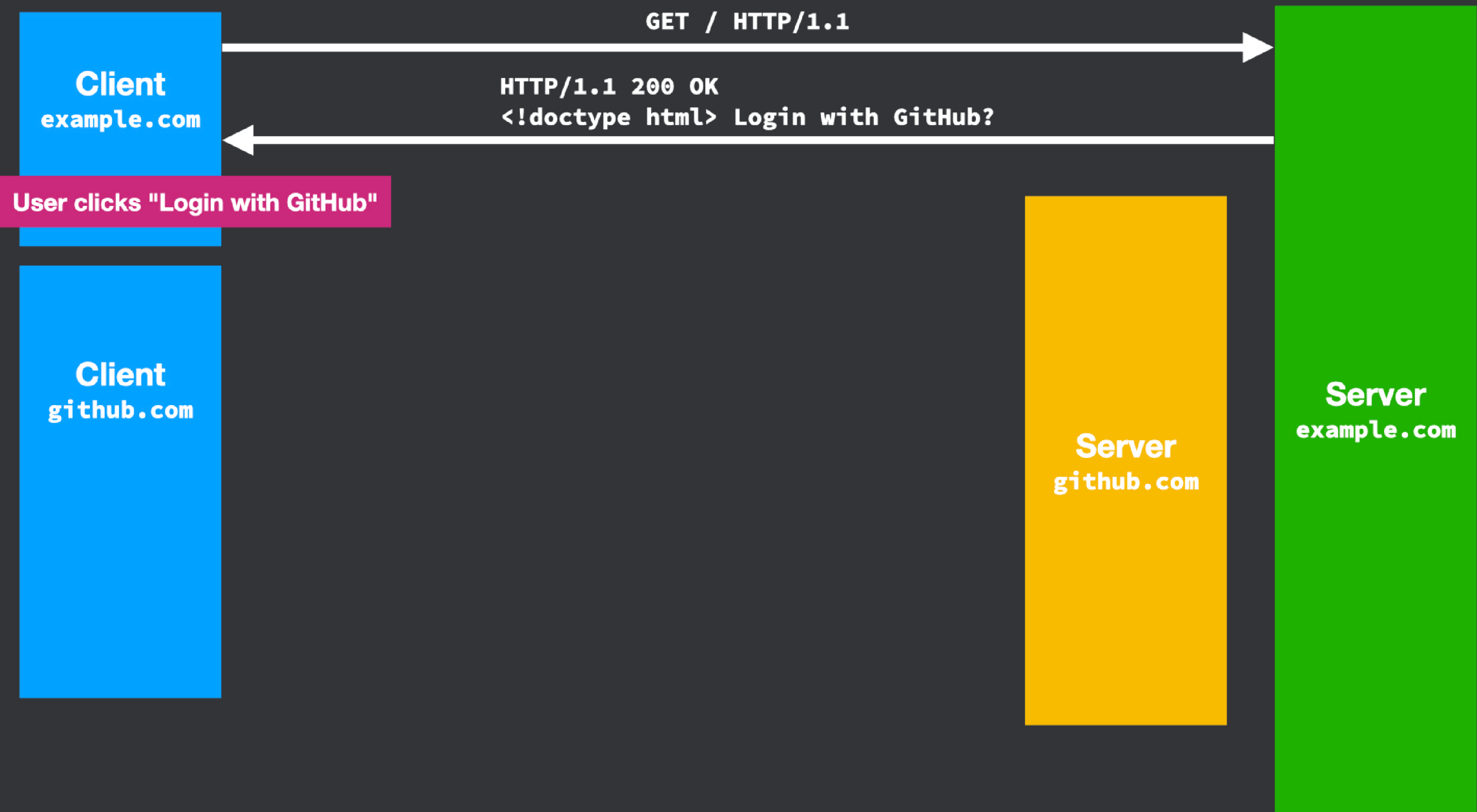
Client
example.com

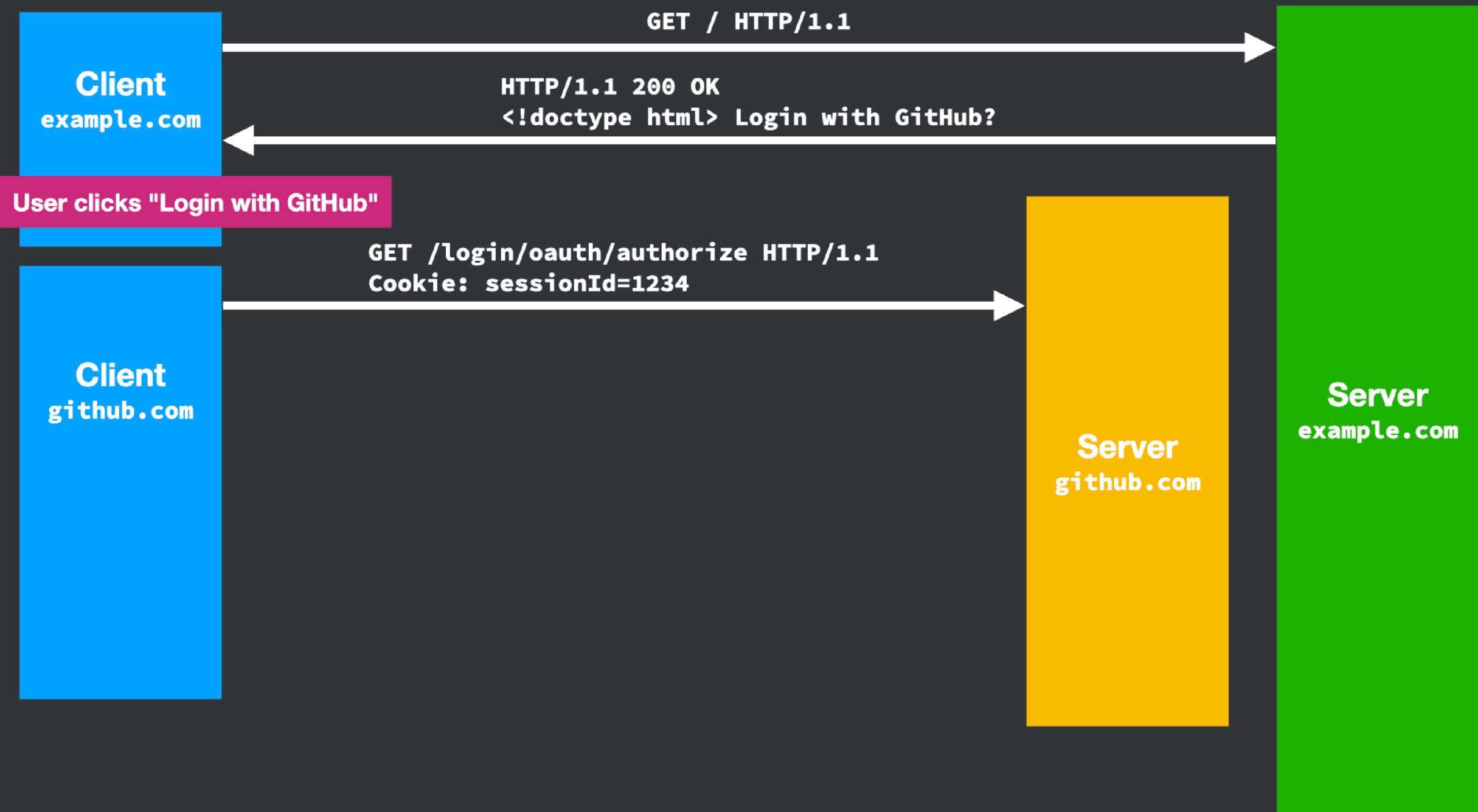
GET / HTTP/1.1

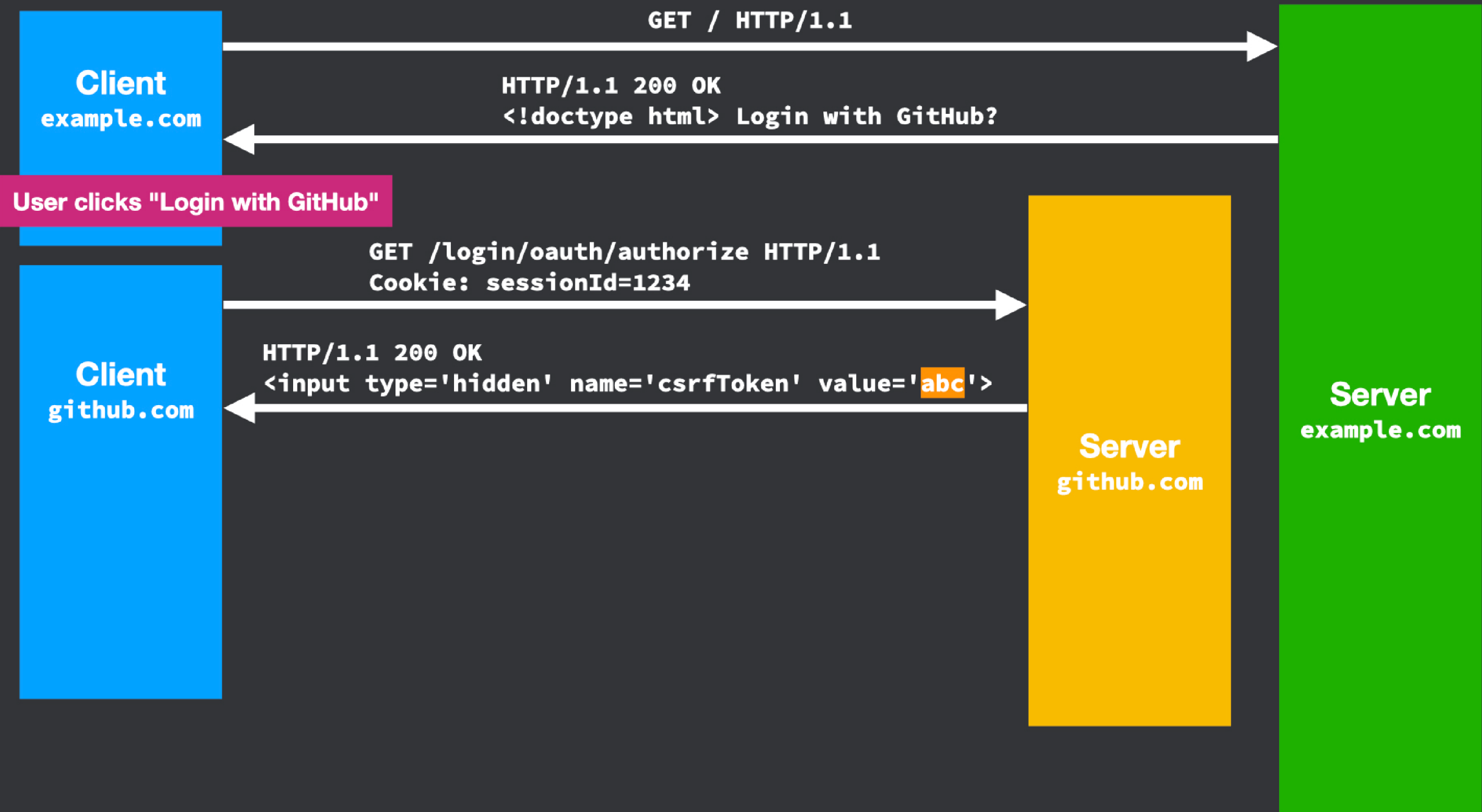
HTTP/1.1 200 OK
<!doctype html> Login with GitHub?

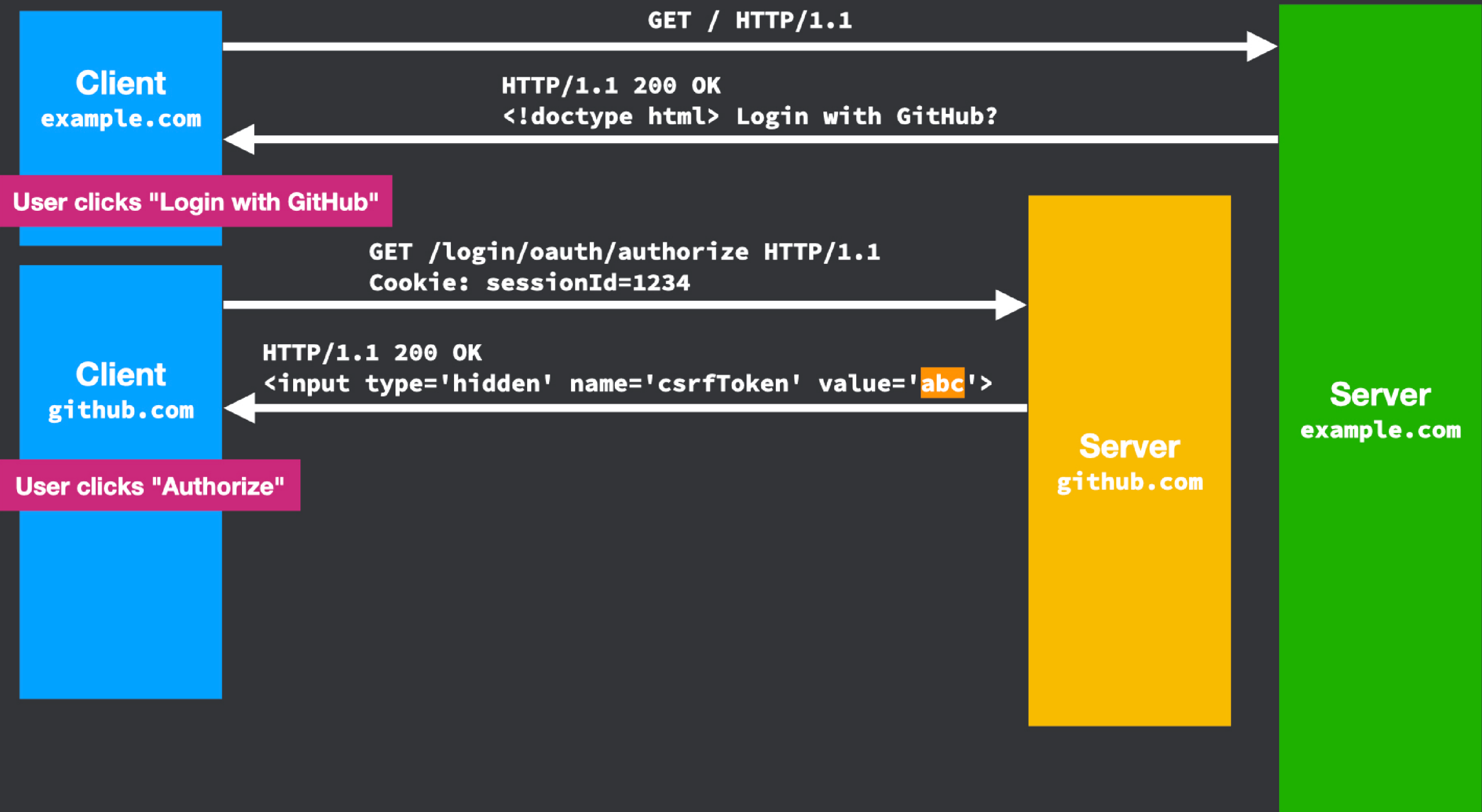
User clicks "Login with GitHub"

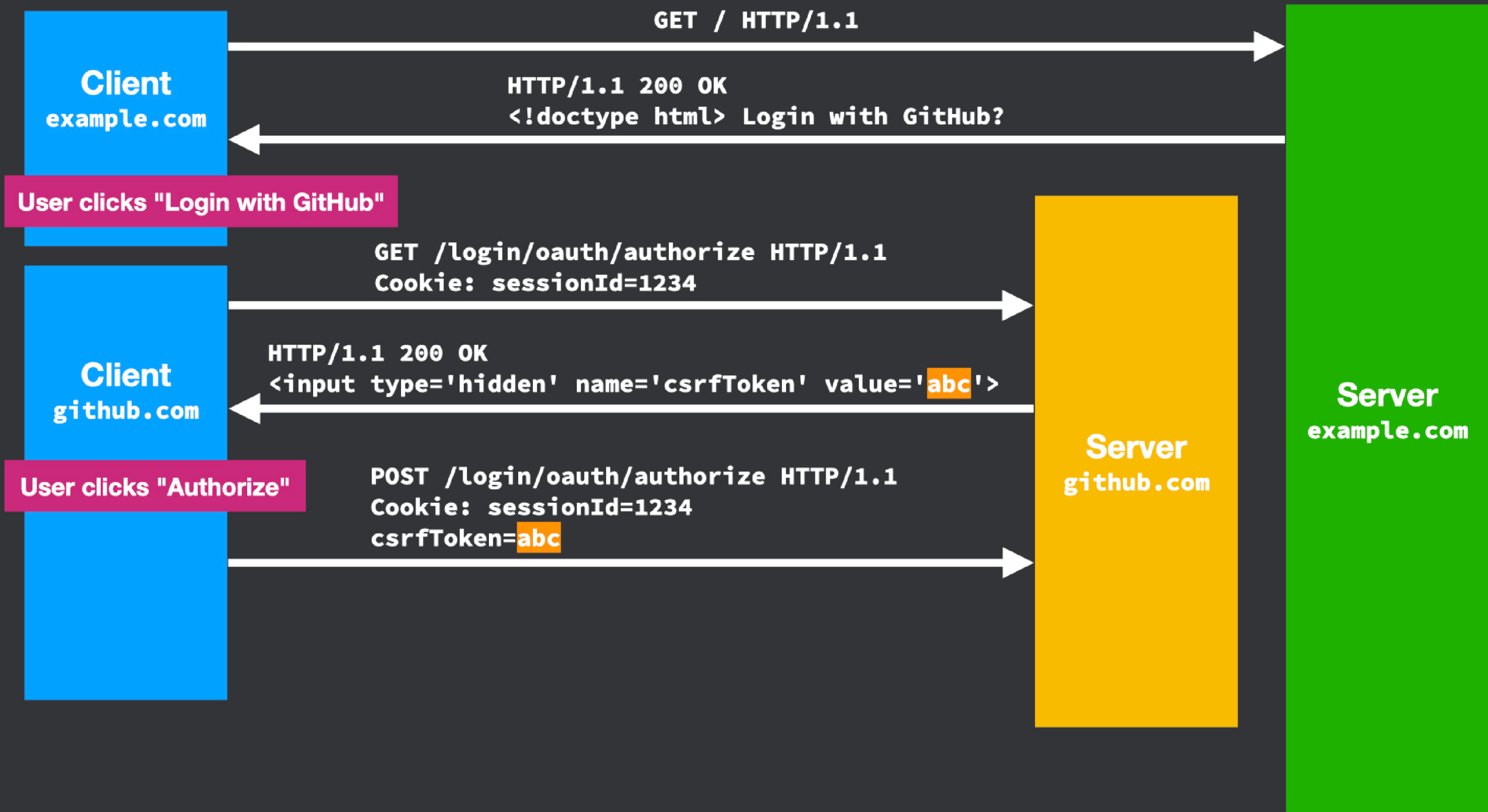
Server
example.com

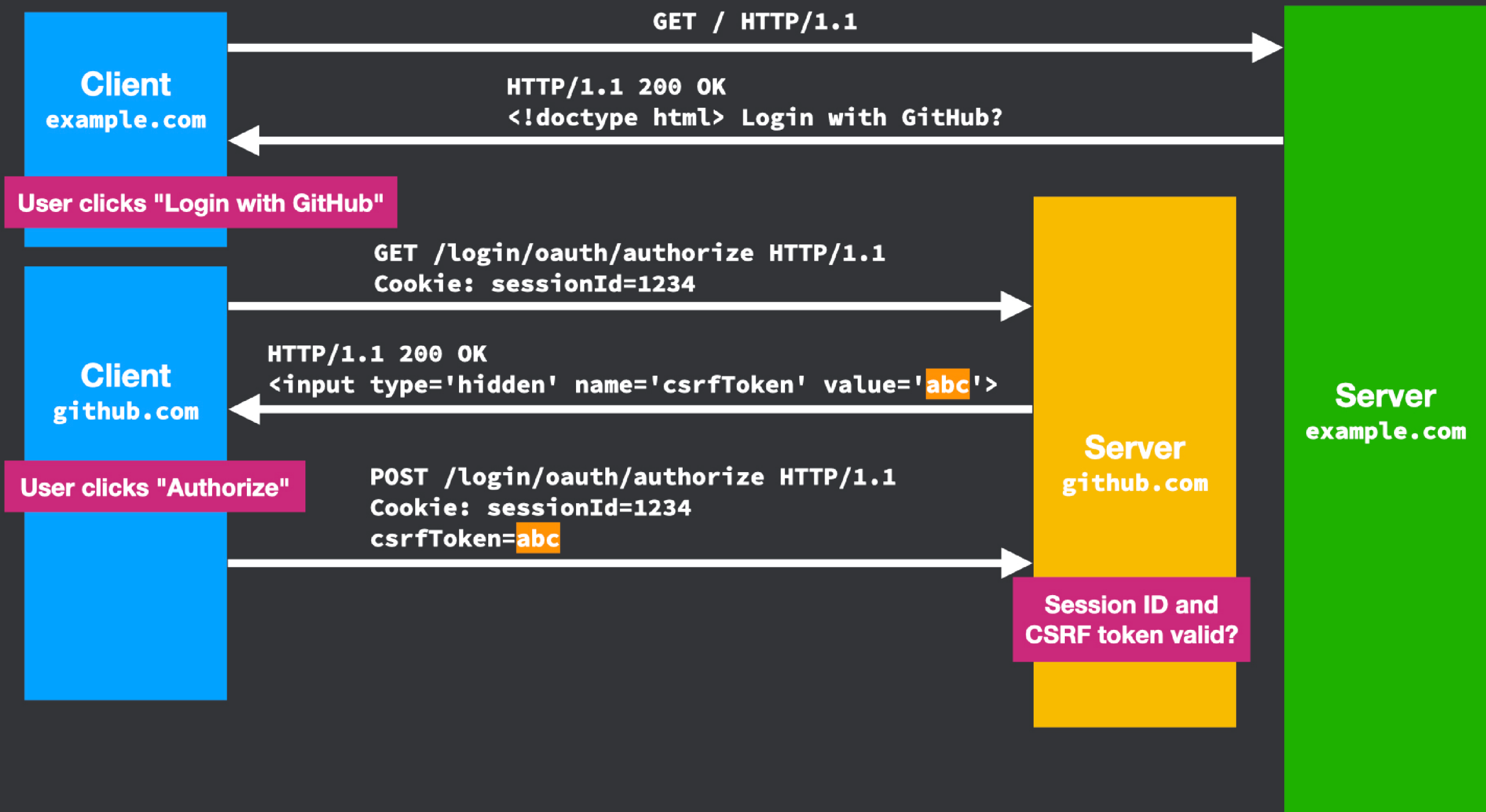


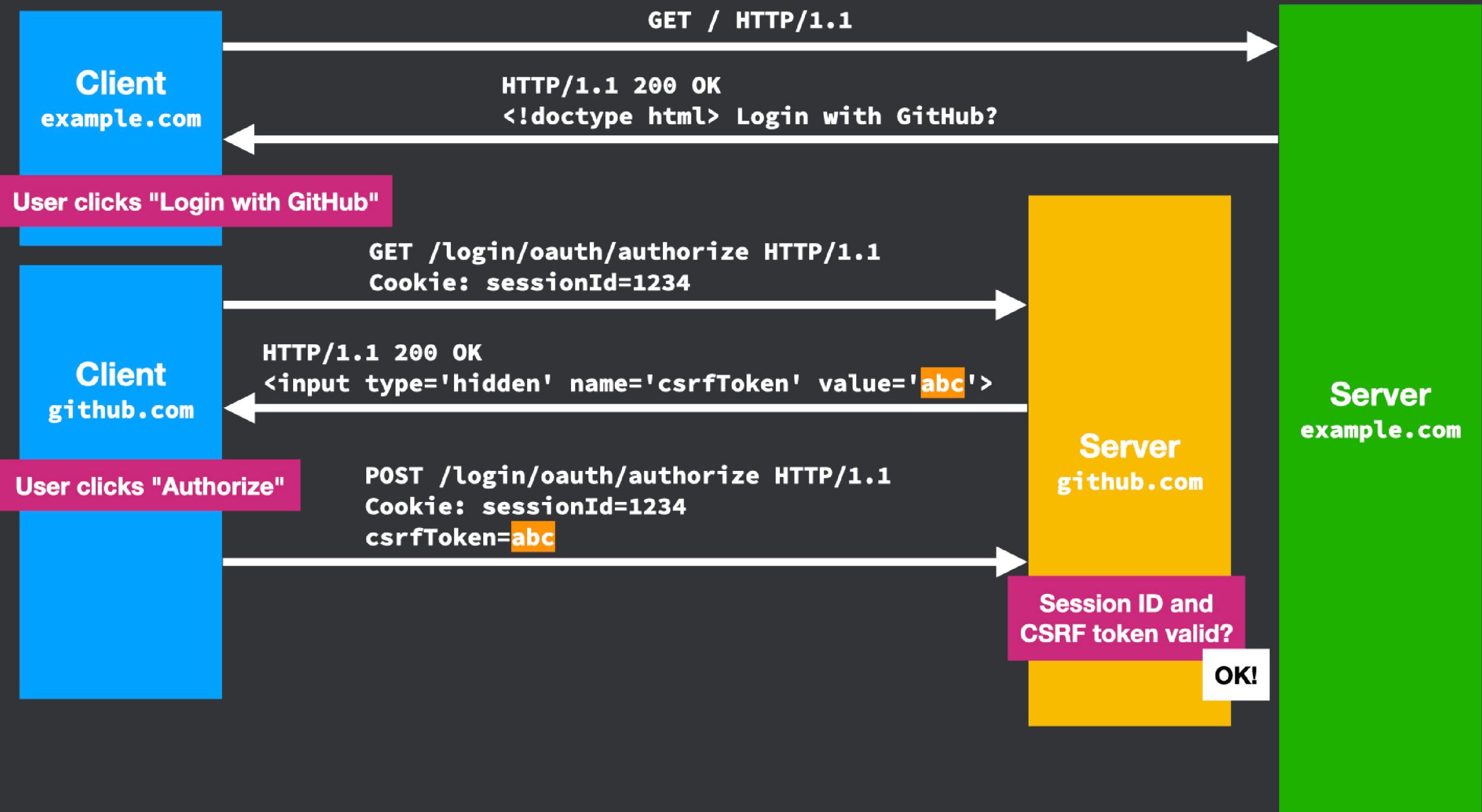


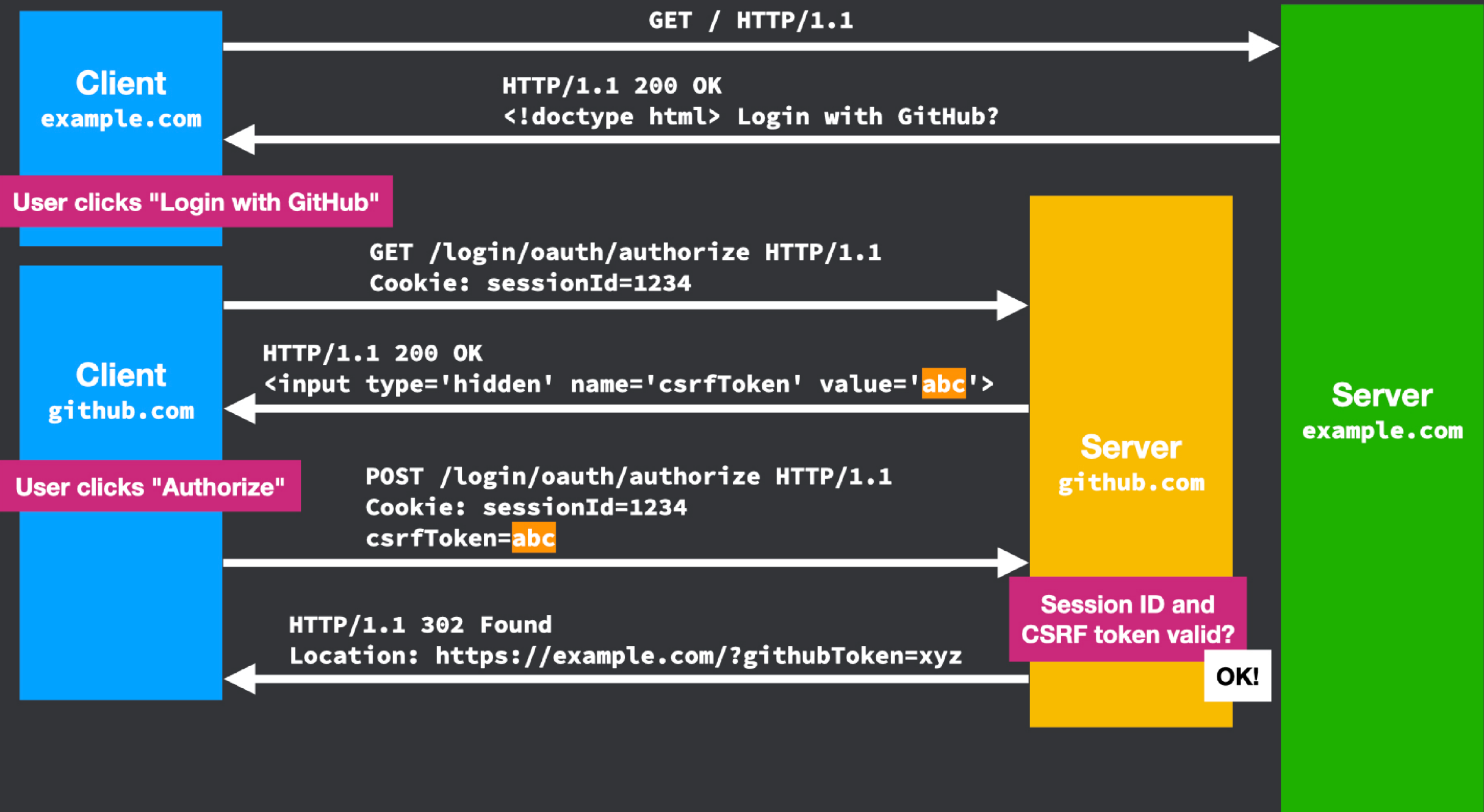


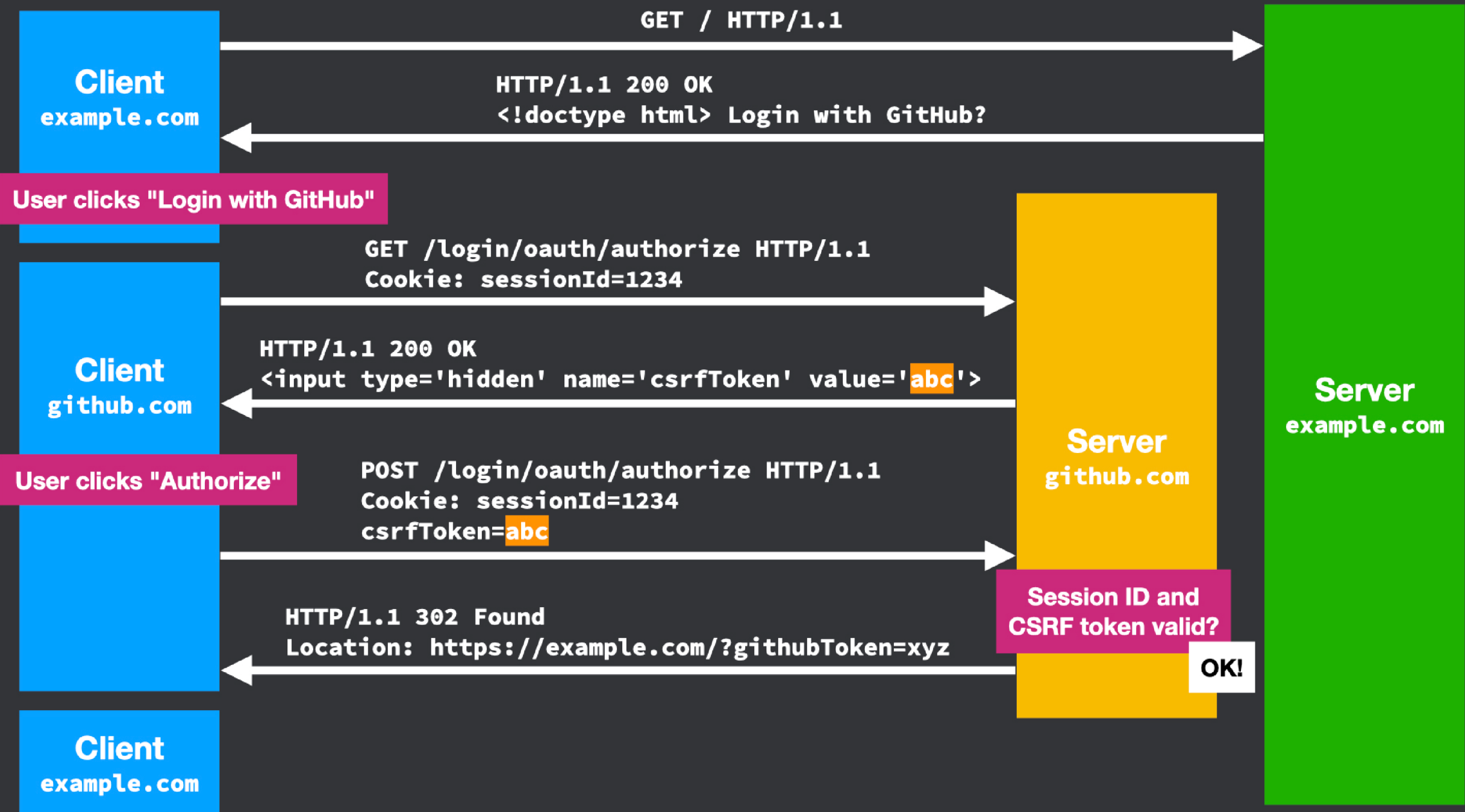


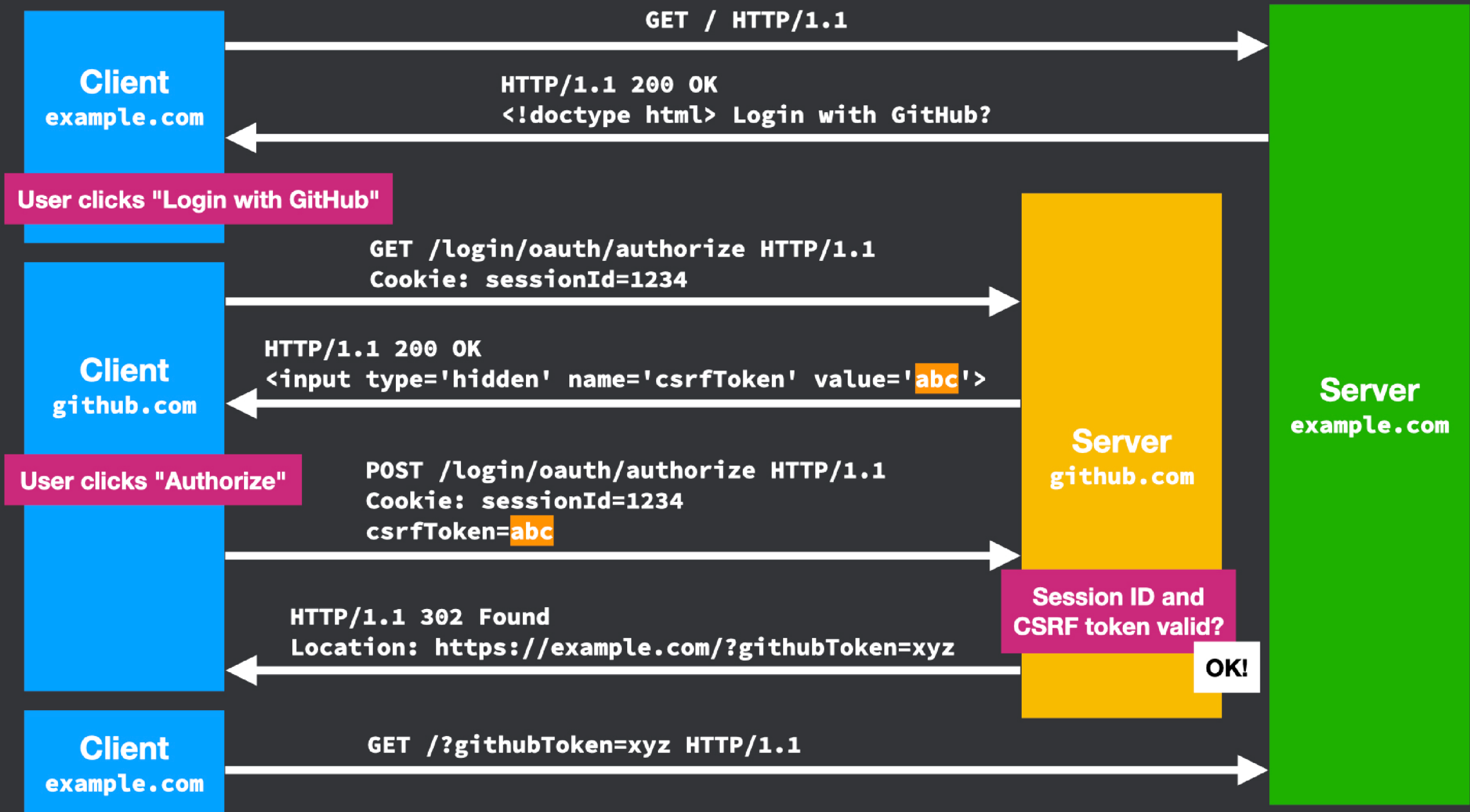












One URL, two HTTP methods

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 request

- 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)

GET / HTTP/1.1

Client
attacker.com

Server
attacker.com

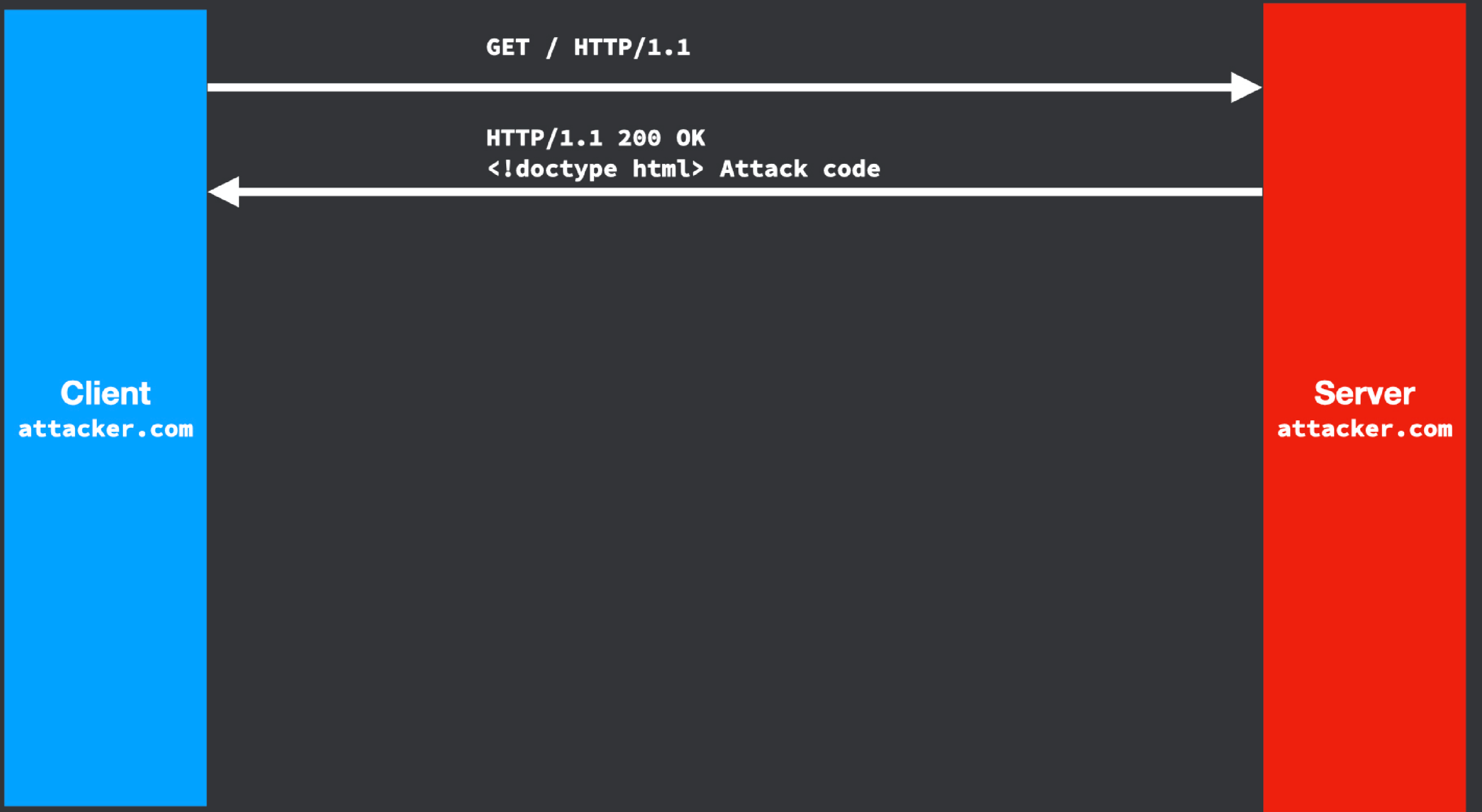


GET / HTTP/1.1

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









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 `elseif request.post?` to ensure HEAD or any other unexpected methods won't be treated as POST

Explicit check for POST

```
# 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 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



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Any Questions?

Trawsnewid Addysg; Trawsnewid Bywydau
Transforming Education; Transforming Lives



www.uwtsd.ac.uk