CS 253: Web Security

Cookies, Sessions

Recall: Cookies

Server sends a cookie with a response

```
Set-Cookie: theme=dark;
```

Header Name

Cookie Name

Cookie Value

Client sends a cookie with a request

```
Cookie: theme=dark;
```

Header Name

Cookie Name

Cookie Value

Sessions

- **Cookies** are used by the server to implement **sessions**
- **Goal:** Server keeps a set of data related to a user's current "browsing session"
- Examples
  - Logins
  - Shopping carts
  - User tracking

Demos: Sessions

Demo: Insecure Session 1

```
<!doctype html>
<html lang='en'>
  <head>
    <meta charset='utf-8' />
    <title>My Cool Site</title>
  </head>
  <body>
    <h1>Bank login:</h1>
    <form method='POST' action='/login'>
      Username:
      <input name='username' />
      <br />
      Password:
      <input name='password' type='password' />
      <br />
      <input type='submit' value='Login' />
    </form>
  </body>
</html>
```

Demo: Insecure Session 1

```
const express = require('express')
const { createReadStream } = require('fs')
const cookieParser = require('cookie-parser')

const app = express()
app.use(cookieParser())
app.use(express.urlencoded({ extended: false }))

// Routes go here!

app.listen(8000)
```

Demo: Insecure Session 1

```
const USERS = { alice: 'password', bob: '50505' }
const BALANCES = { alice: 500, bob: 100 }

app.get('/', (req, res) => {
  const { username } = req.cookies
  if (username) {
    res.send(`
      <h1>Welcome, ${username}</h1>
      <p>Your balance is $${BALANCES[username]}</p>
    `)
  } else {
    createReadStream('index.html').pipe(res)
  }
})
```

Demo: Insecure Session 1

```
app.post('/login', (req, res) => {
  const { username } = req.body
  const { password } = req.body
  if (password === USERS[username]) {
    res.cookie('username', username)
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

app.get('/logout', (req, res) => {
  res.clearCookie('username')
  res.redirect('/')
})
```

First HTTP request:

```
POST /login HTTP/1.1
Host: example.com

username=alice&password=password
```

HTTP response:

```
HTTP/1.1 200 OK
Set-Cookie: username=alice
Date: Tue, 24 Sep 2019 20:30:00 GMT

<!DOCTYPE html ...
```

All future HTTP requests:

```
GET /page.html HTTP/1.1
Host: example.com
Cookie: username=alice;
```

Ambient authority

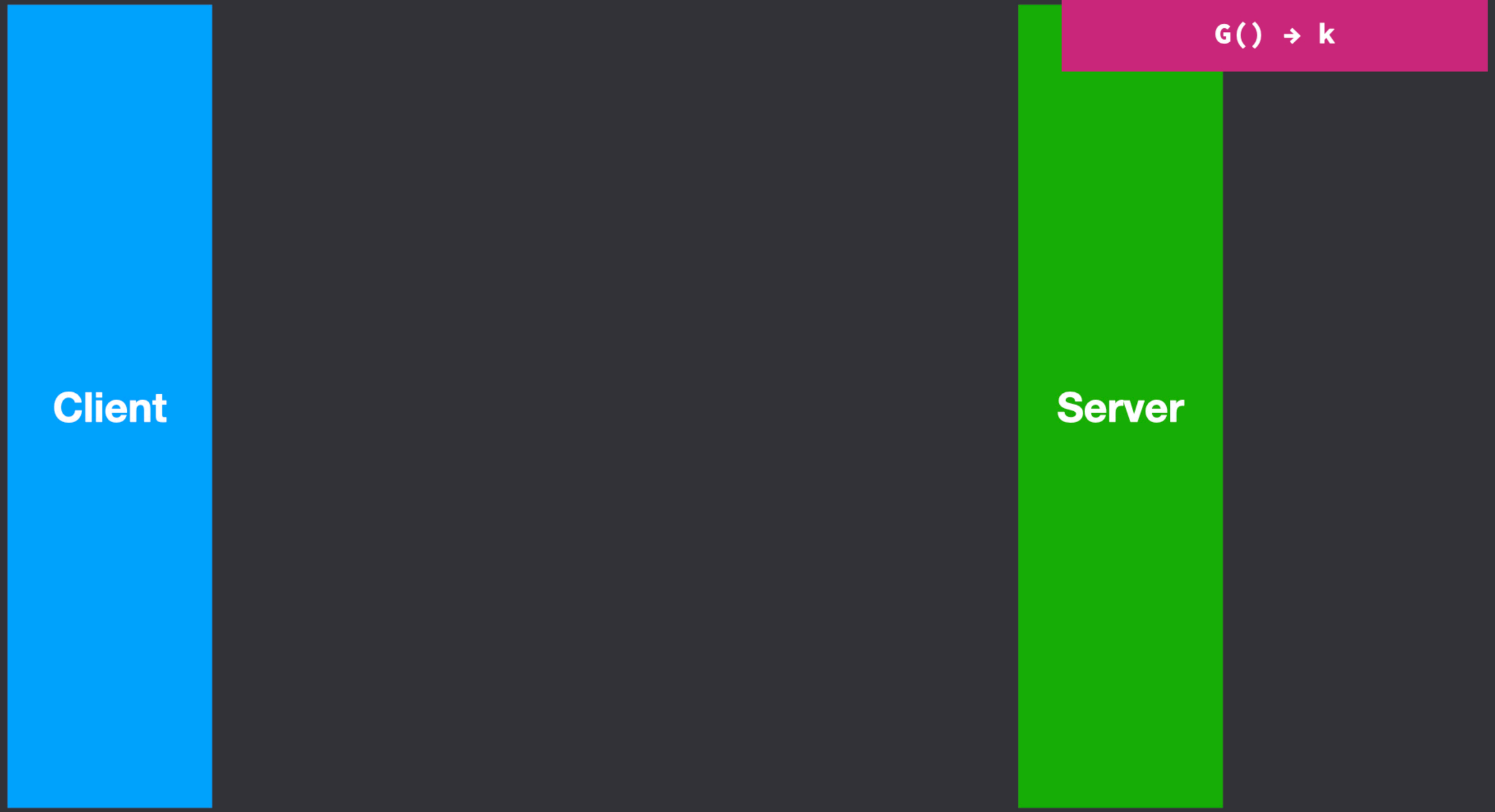
- **Access control** - Regulate who can view resources or take actions
- **Ambient authority** - Access control based on a **global and persistent property** of the requester
  - The alternative is explicit authorization **valid only for a specific action**
- There are four types of ambient authority on the web
  - **Cookies** - most common, most versatile method
  - **IP checking** - used at Stanford for library resources
  - **Built-in HTTP authentication** - rarely used
  - **Client certificates** - rarely used

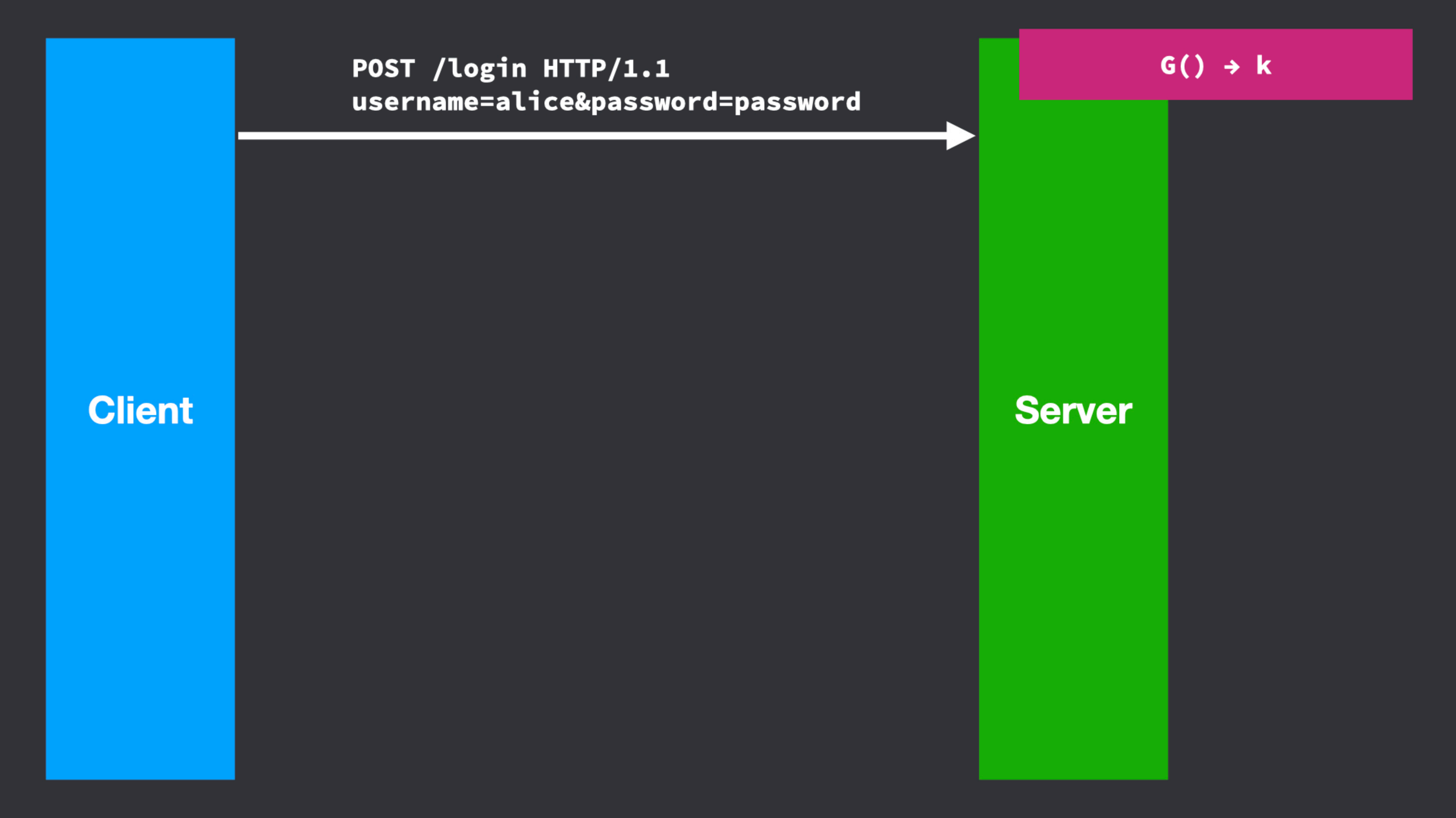
Quick primer: Signature schemes

- Triple of algorithms `(G, S, V)`
  - `G`() → `k` - generator returns key
  - `S`(`k`, `x`) → `t` - signing returns a tag `t` for input `x`
  - `V`(`k`, `x`, `t`) → `accept`|`reject` - checks validity of tag `t` for given input `x`
- Correctness property
  - `V`(`k`, `x`, `S`(`k`, `x`)) = `accept` should always be true
- Security property
  - `V`(`k`, `x`, `t`) = `accept` should almost never be true when `x` and `t` are chosen by the attacker

Client

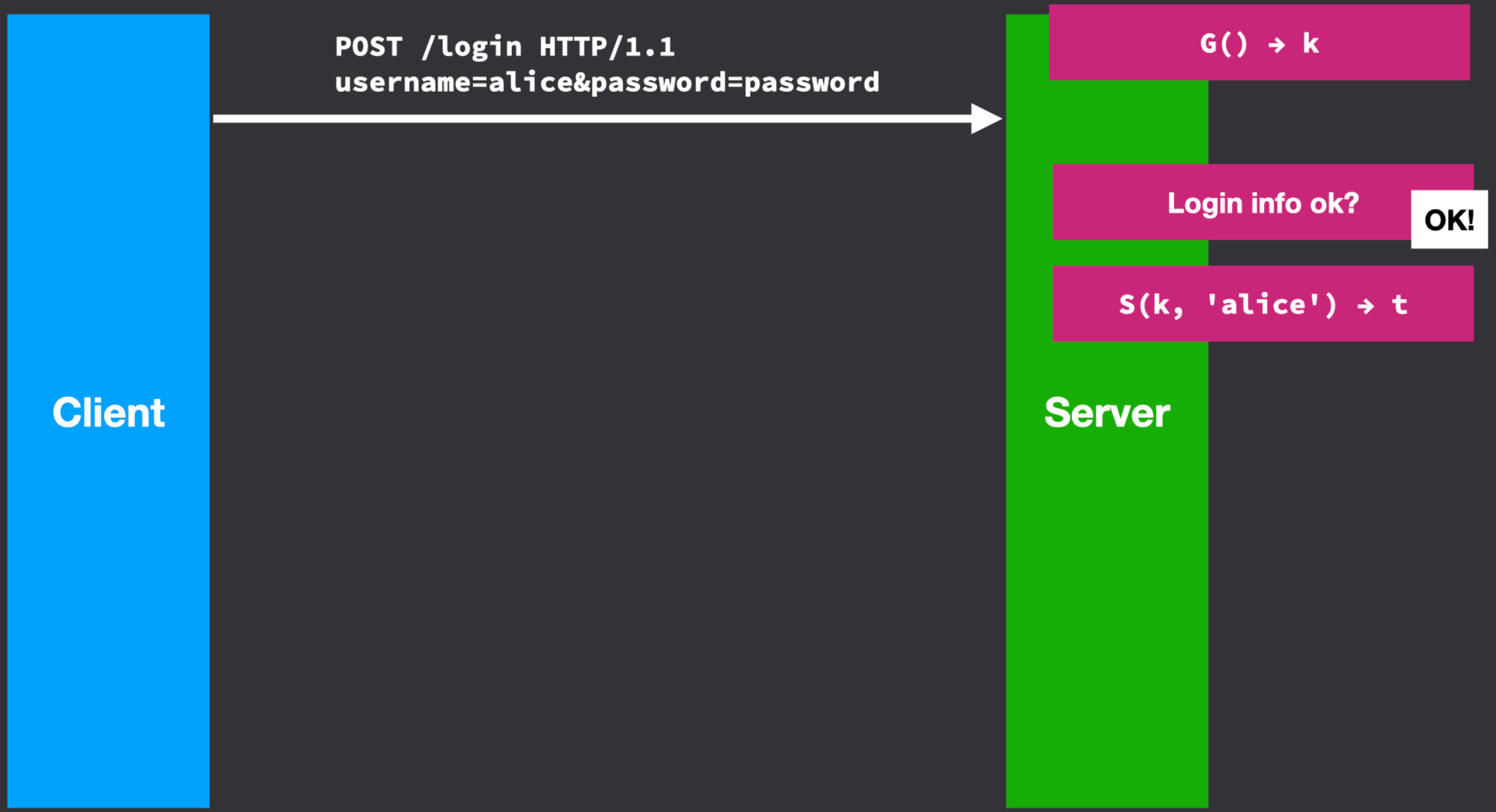
Server

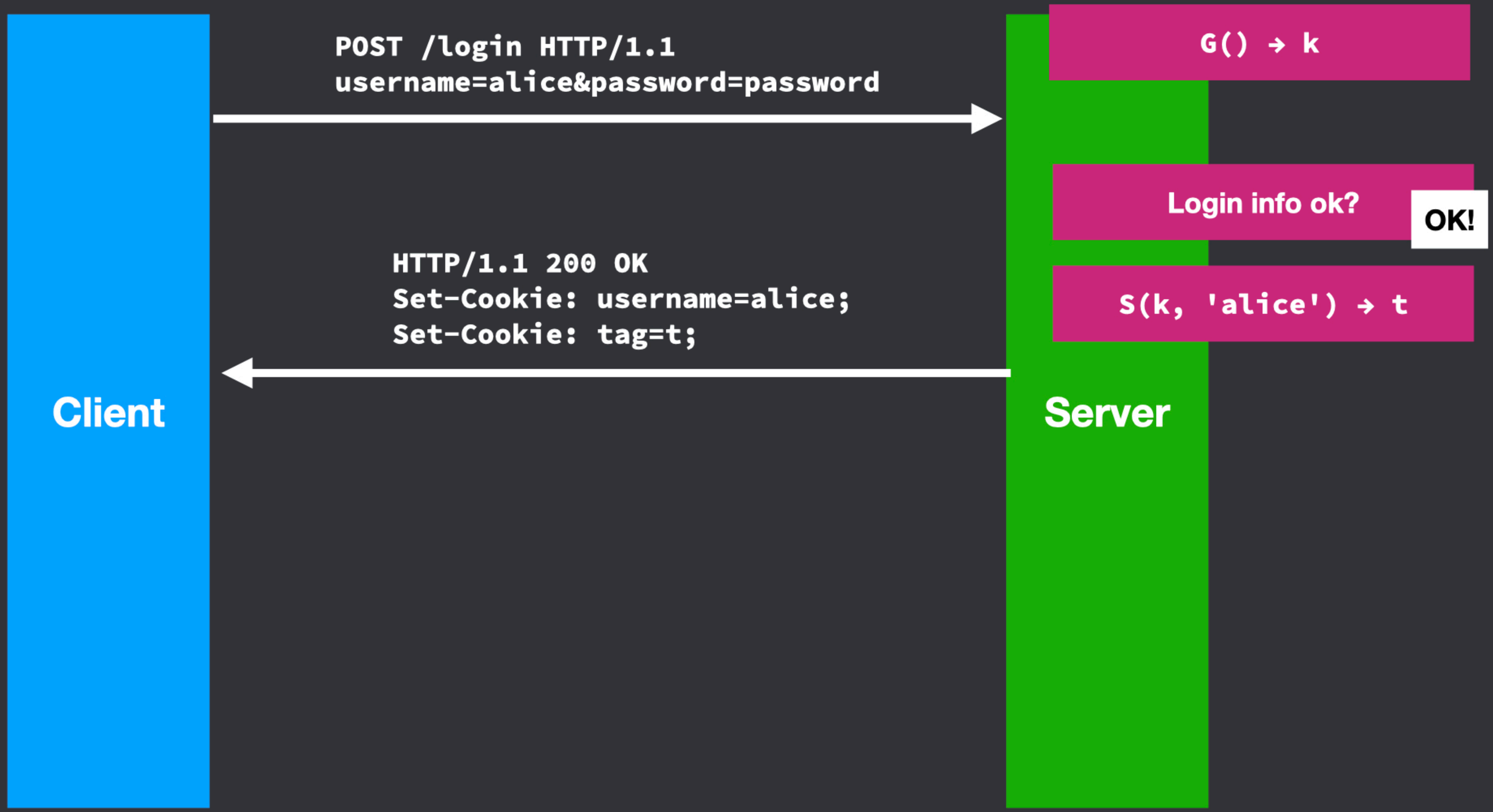

Client
Server
G() → k

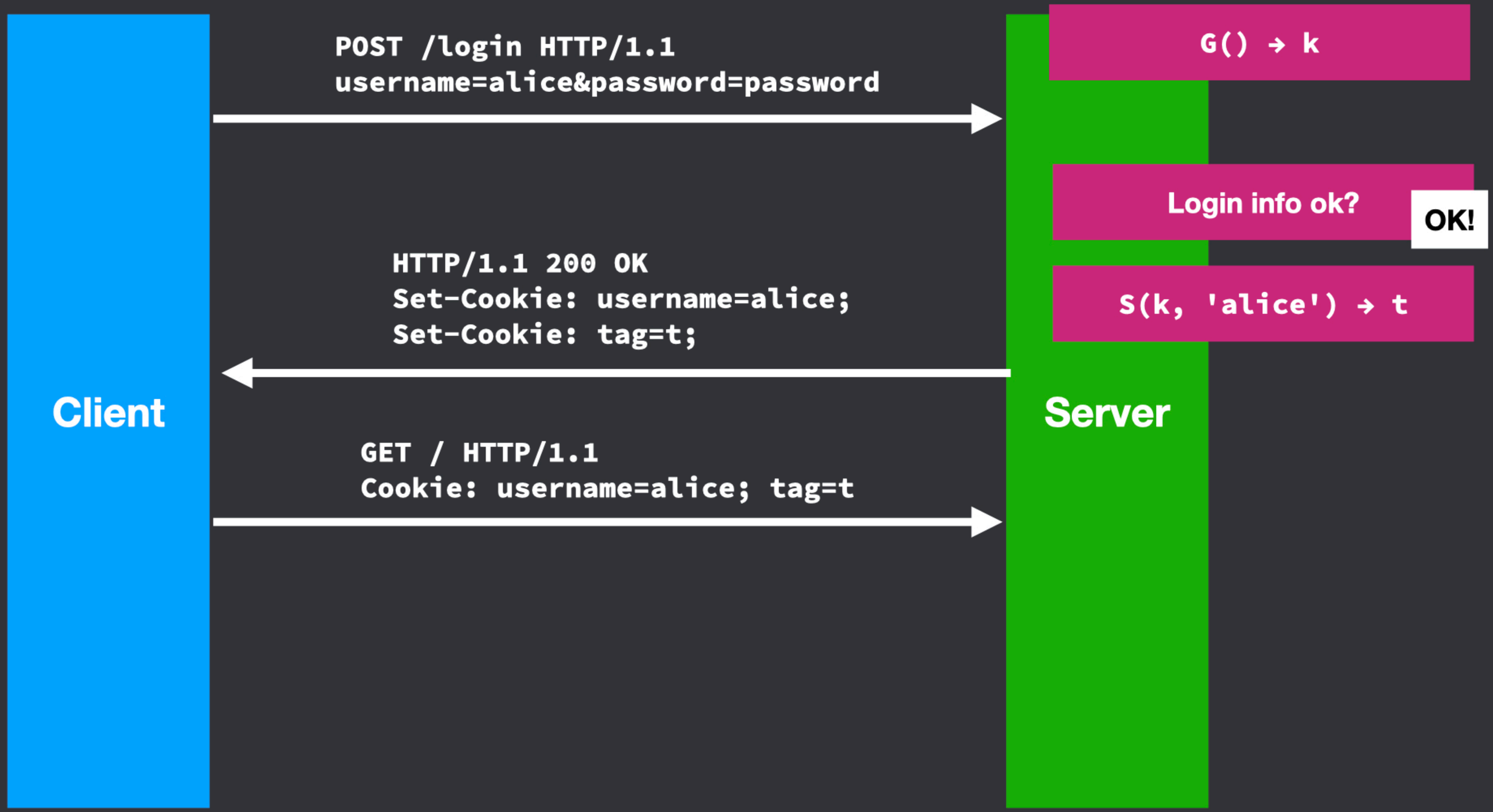

POST /login HTTP/1.1
username=alice&password=password
G() → k
Client
Server

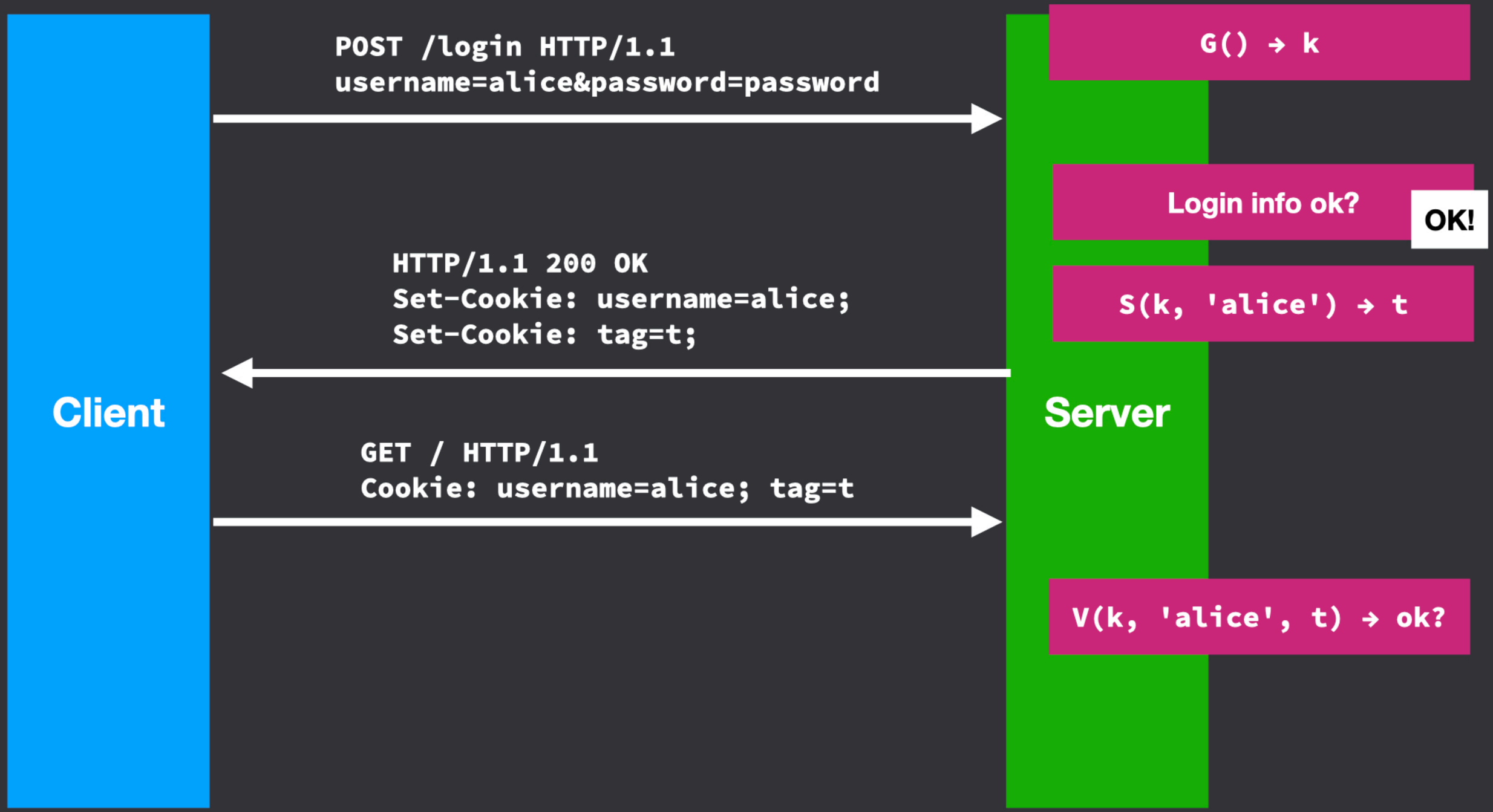

POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
Client
Server

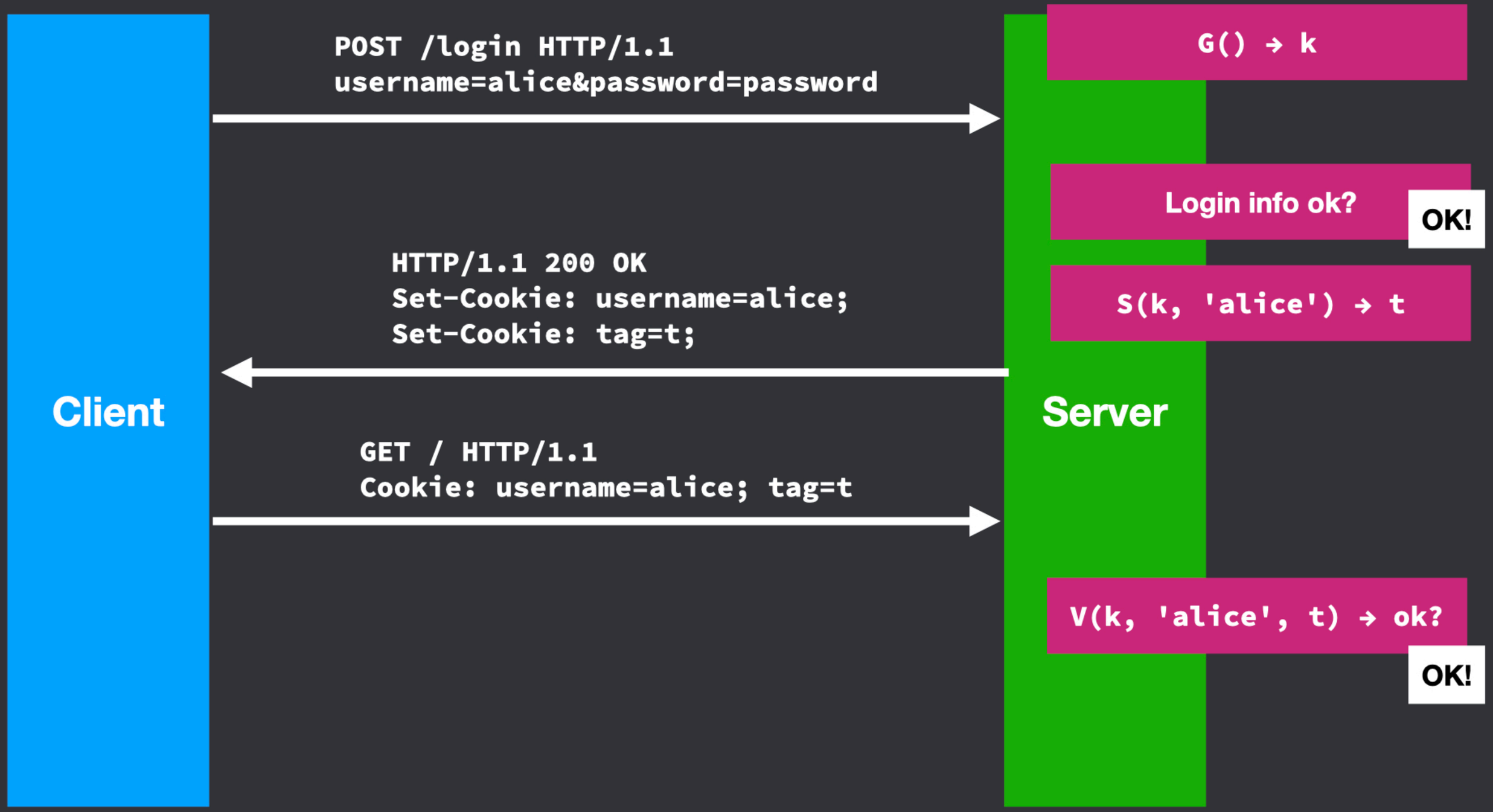

POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
Client
Server

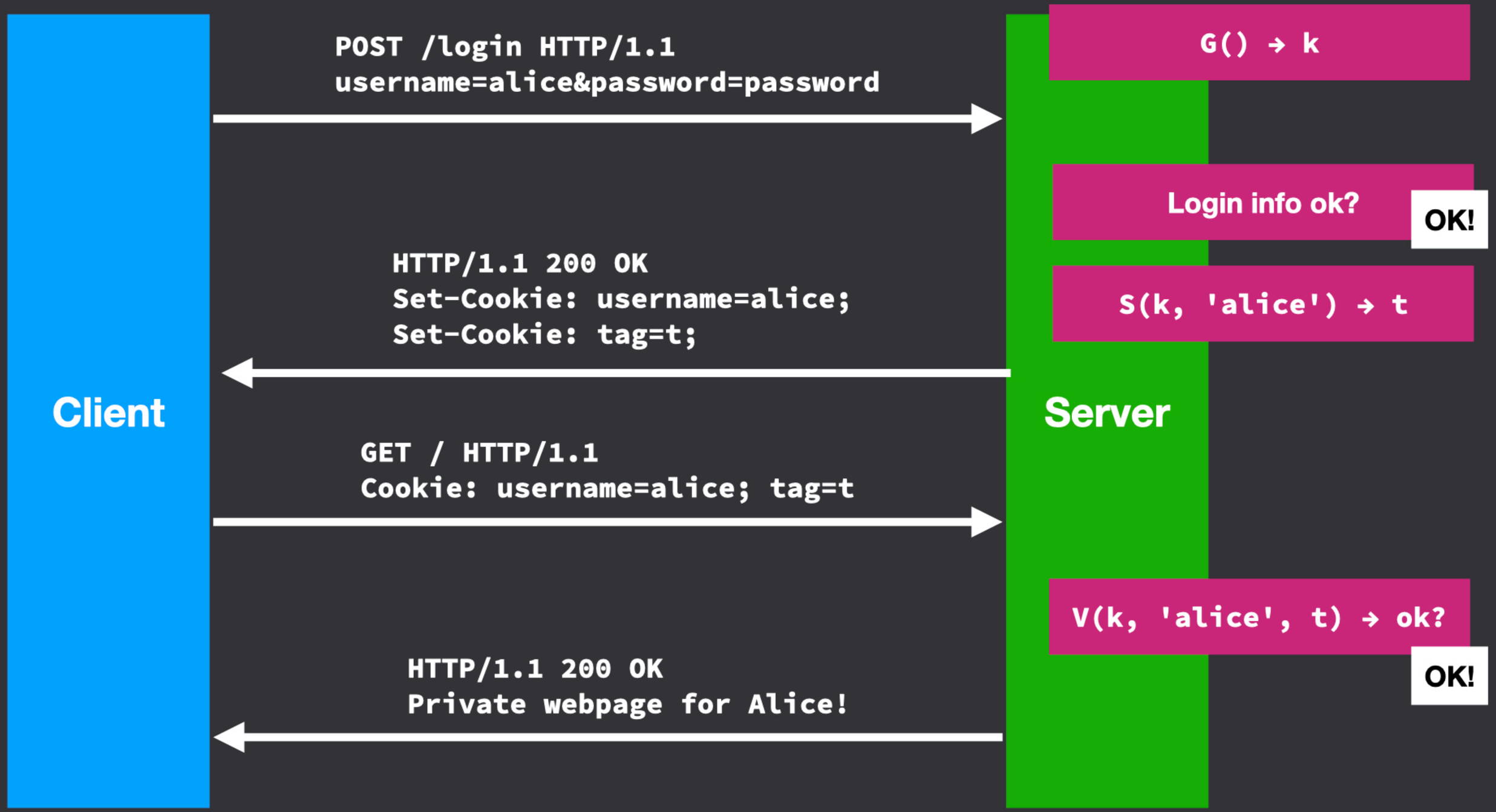

POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
S(k, 'alice') → t
Client
Server

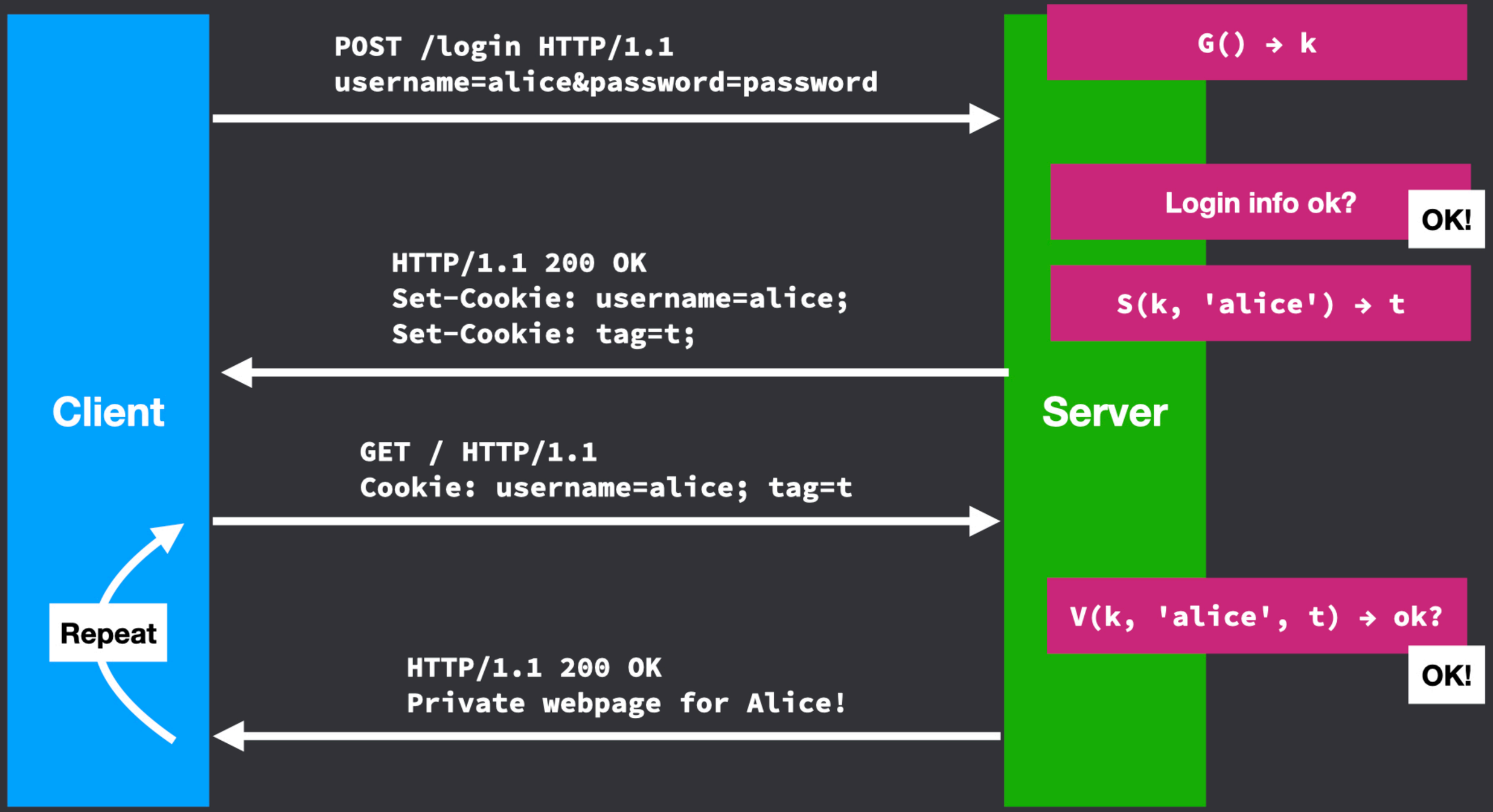

POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(k, 'alice') → t
Client
Server


POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(k, 'alice') → t
Client
Server
GET / HTTP/1.1
Cookie: username=alice; tag=t


Client
POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(k, 'alice') → t
Server
GET / HTTP/1.1
Cookie: username=alice; tag=t
V(k, 'alice', t) → ok?


POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(k, 'alice') → t
Client
Server
GET / HTTP/1.1
Cookie: username=alice; tag=t
V(k, 'alice', t) → ok?
OK!


Client
POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(k, 'alice') → t
Server
GET / HTTP/1.1
Cookie: username=alice; tag=t
V(k, 'alice', t) → ok?
OK!
HTTP/1.1 200 OK
Private webpage for Alice!


POST /login HTTP/1.1
username=alice&password=password
G() → k
Login info ok?
OK!
HTTP/1.1 200 OK
Set-Cookie: username=alice;
Set-Cookie: tag=t;
S(k, 'alice') → t
Client
Server
GET / HTTP/1.1
Cookie: username=alice; tag=t
Repeat
V(k, 'alice', t) → ok?
OK!
HTTP/1.1 200 OK
Private webpage for Alice!

Demo: Insecure Session 2

```
const COOKIE_SECRET = 'G2T7SRHTX1T62DHR'
app.use(cookieParser(COOKIE_SECRET))

app.get('/', (req, res) => {
  const { username } = req.signedCookies
  if (username) {
    res.send(`
      <h1>Welcome, ${username}</h1>
      <p>Your balance is $${BALANCES[username]}</p>
    `)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const { username } = req.body
  const { password } = req.body
  if (password === USERS[username]) {
    res.cookie('username', username, { signed: true })
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

app.get('/logout', (req, res) => {
  res.clearCookie('username')
  res.redirect('/')
})
```

Demo: Insecure Session 3

```
let nextSessionId = 1
const SESSIONS = {} // sessionId -> username

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>
    `)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const { username } = req.body
  const { password } = req.body
  if (password === USERS[username]) {
    SESSIONS[nextSessionId] = username
    res.cookie('sessionId', nextSessionId)
    nextSessionId += 1
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

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

Demo: Secure Session

```
const { randomBytes } = require('crypto')

const SESSIONS = {} // sessionId -> username

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

  if (username) {
    res.send(`Hi ${username}. Your balance is $${BALANCES[username]}.`)
  } else {
    createReadStream('index.html').pipe(res)
  }
})

app.post('/login', (req, res) => {
  const username = req.body.username
  const password = USERS[username]
  if (password === req.body.password) {
    const sessionId = randomBytes(16).toString('hex')
    SESSIONS[sessionId] = username
    res.cookie('sessionId', sessionId)
    res.redirect('/')
  } else {
    res.send('fail!')
  }
})

app.get('/logout', (req, res) => {
  const sessionId = req.cookies.sessionId
  delete SESSIONS[sessionId]
  res.clearCookie('sessionId')
  res.redirect('/')
})
```

Sessions: Desired Properties

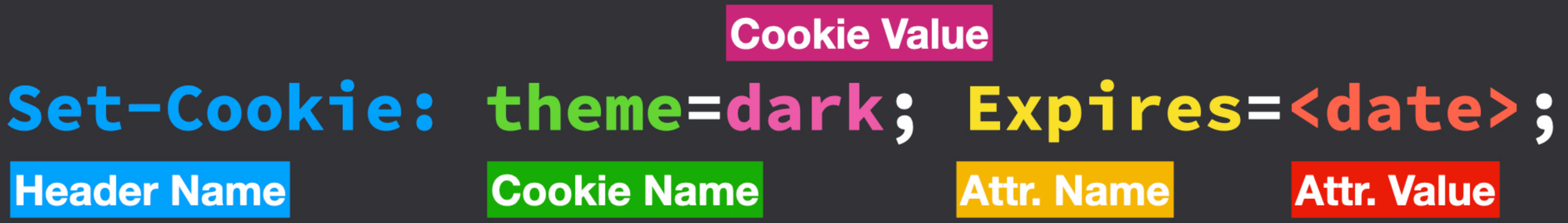
- Browser remembers user (so user doesn't need to repeatedly log in)
- User cannot modify session cookie to login as another user
- Session cookies are not valid forever
- Sessions can be deleted on the server-side
- Sessions should expire after some time, e.g. 30 days

History of cookies

- Implemented in 1994 in Netscape and described in 4-page draft
- No spec for 17 years
  - Attempt made in 1997, but made incompatible changes
  - Another attempt in 2000 ("Cookie2"), same problem
  - Around 2011, another effort succeeded (RFC 6265)
- Ad-hoc design has led to *interesting* issues

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. **`attacker.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`**, another origin can set cookies for another origin


Cookie Value
Set-Cookie: theme=dark; Expires=<date>;
Header Name
Cookie Name
Attr. Name
Attr. Value

How long can cookies last?

- Sites can set **`Expires`** to a very far-future date and the cookie will last until the user clears it.
  - 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
- When **`Expires`** not specified, lasts for current browser session
  - Caveat: Browsers do session restoring, so can last way longer

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
```

Accessing Cookies from JS

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

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

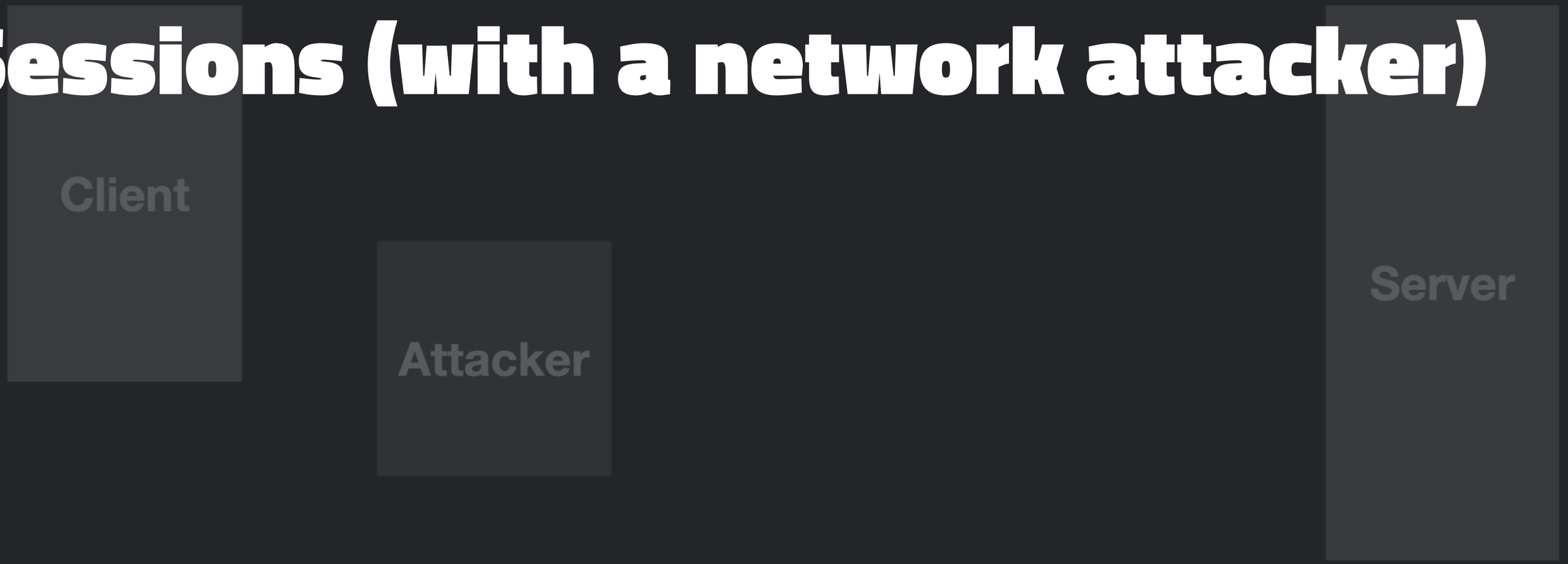
Server


Client
Server

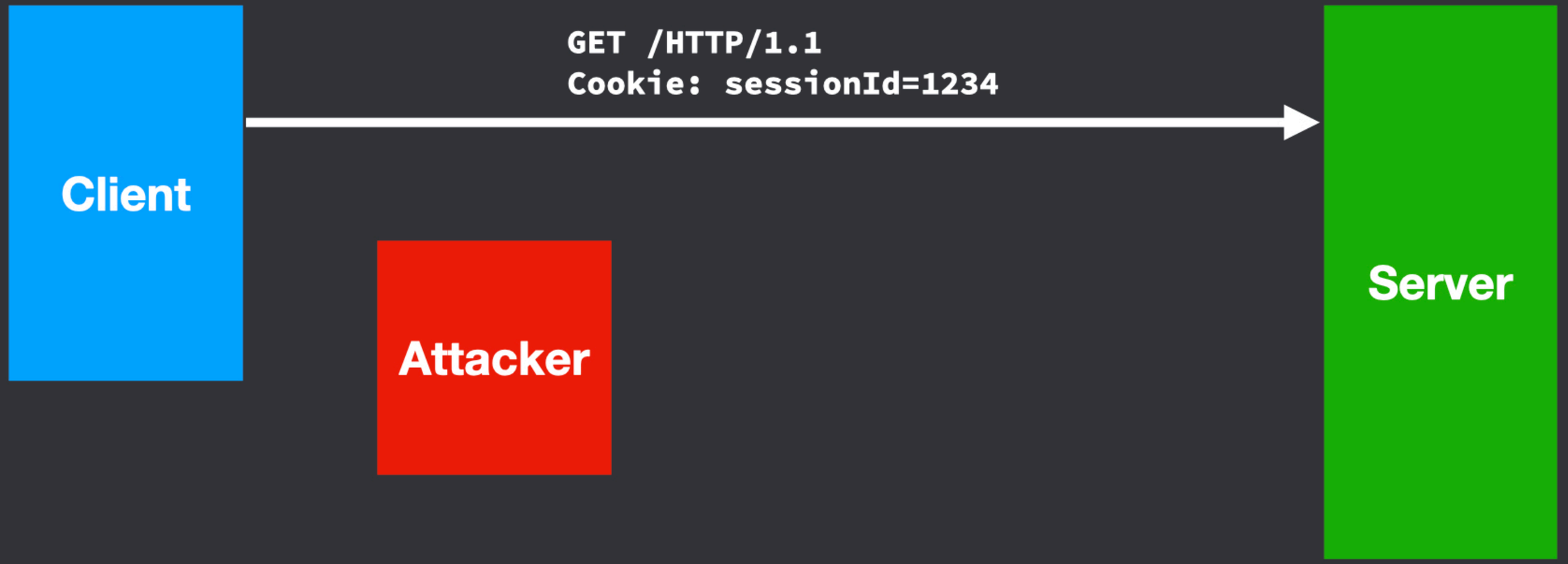

Client
GET /HTTP/1.1
Cookie: sessionId=1234
Server


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

Sessions (with a network attacker)




Client
Attacker
Server

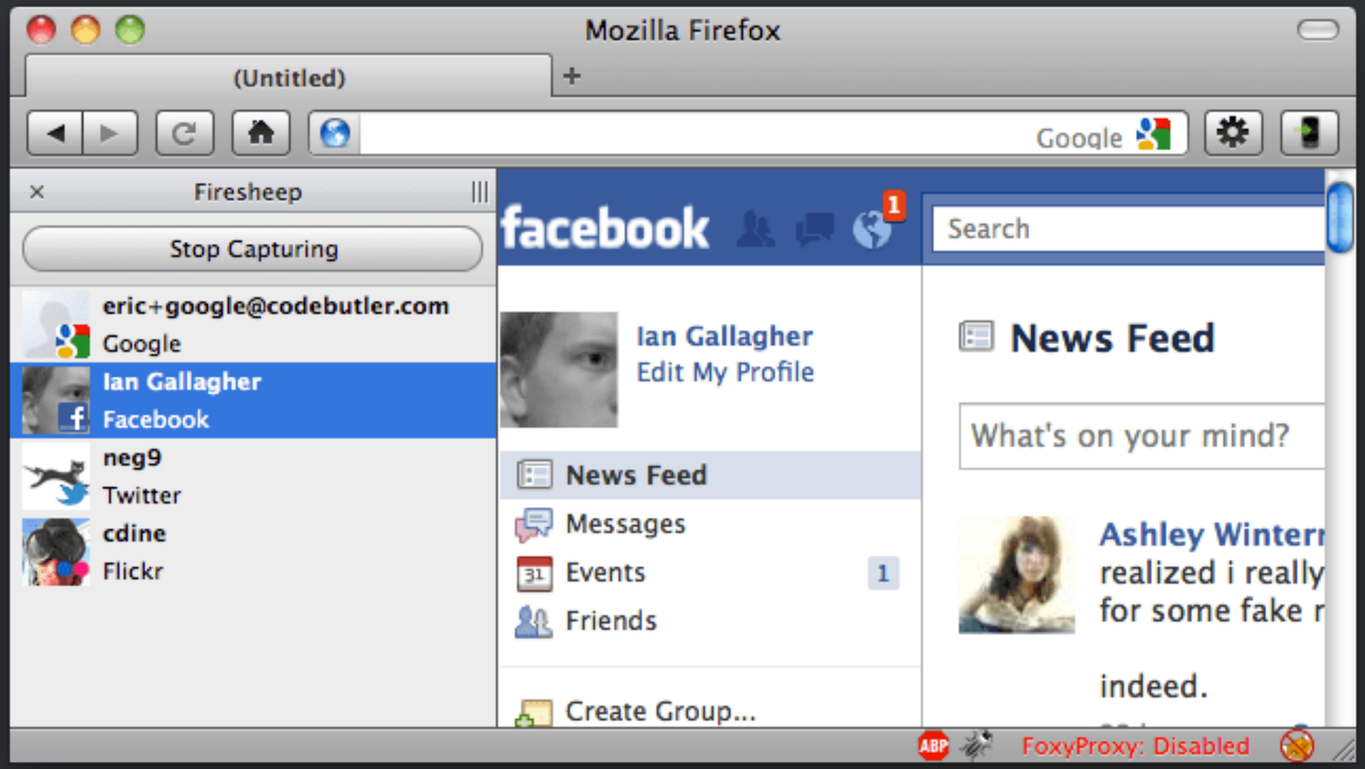

Client
GET /HTTP/1.1
Cookie: sessionId=1234
Attacker
Server


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

END