# Chapter 7

## Attacking Session Management:

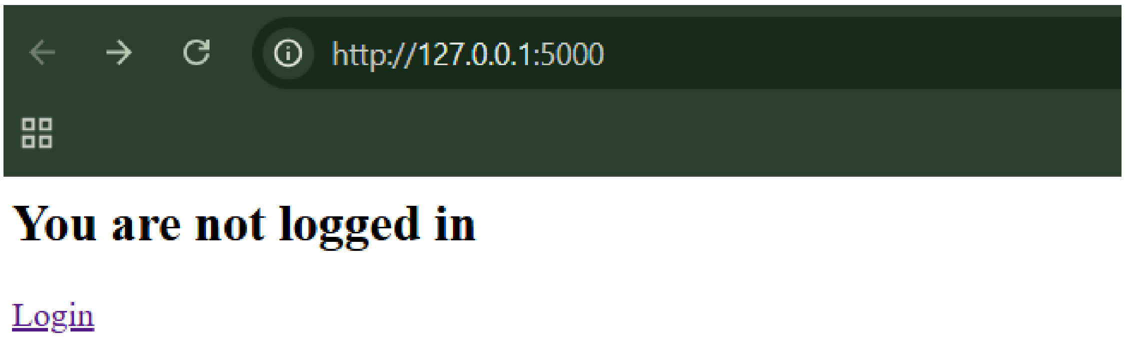
Session management is how a website keeps track of who a user is after they log in, allowing the site to recognize them across multiple requests. If this system is weak, attackers can hijack or fake sessions and act as other users without knowing their passwords. This can lead to full control of the application, especially if an admin account is compromised. Even strong authentication methods are useless if session management is insecure.

## The Need for State:

Web applications need sessions because HTTP does not remember users between requests. Sessions use unique tokens (usually stored in cookies) to link multiple requests to the same user, whether logged in or not. If session tokens are weak or poorly handled, attackers can hijack sessions and impersonate users. Identifying and protecting session tokens is therefore critical for application security.

**Practical Exploration of Session Management Flaws in a Deliberately Insecure Web Application:**

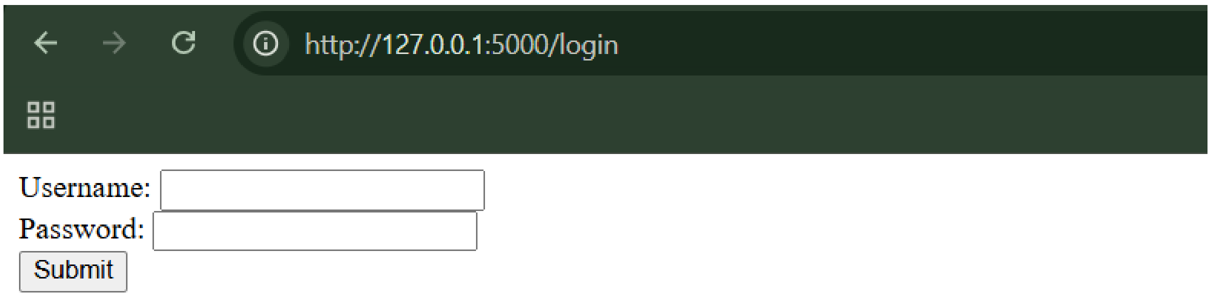
Step1: Visit the http://127.0.0.1:5000


http://127.0.0.1:5000

You are not logged in

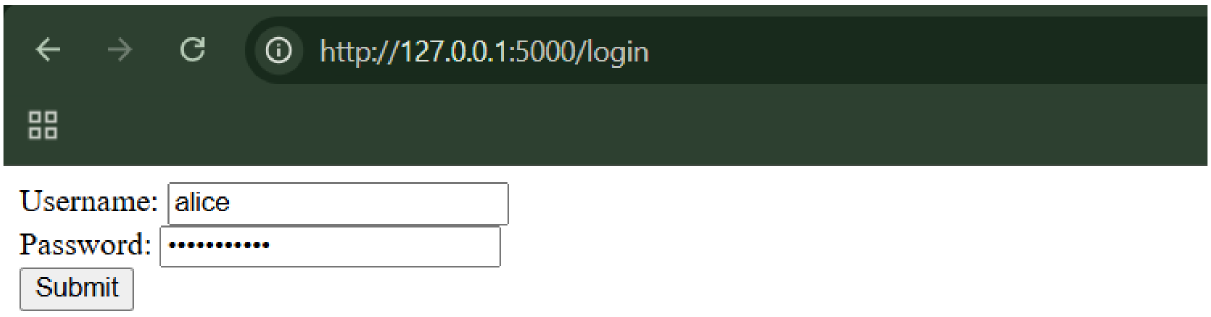
Login

Step2: Click on the "Login" link


http://127.0.0.1:5000/login

Username:
Password:
Submit

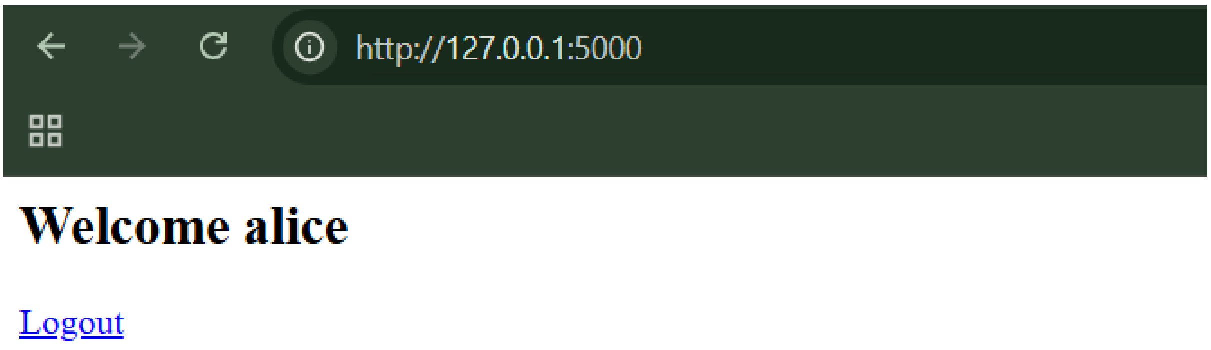
Enter the credentials: alice/password123



http://127.0.0.1:5000/login

Username: alice
Password: •••••••••••

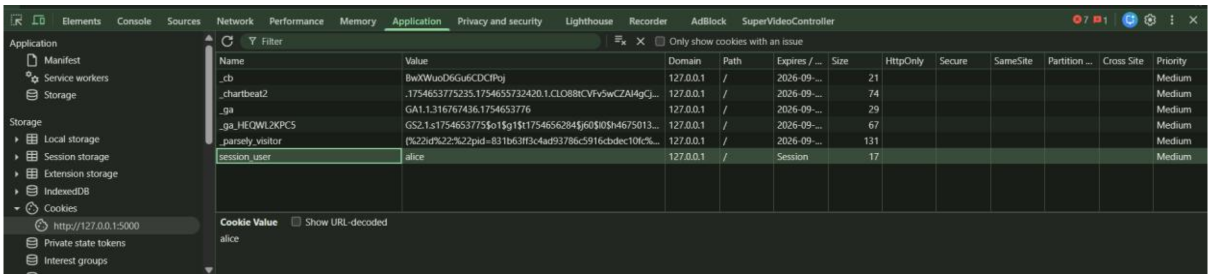
After clicking submit button:



http://127.0.0.1:5000

Welcome alice

Step3: Finding the real session token. Open browser dev tools → Application → Cookies.

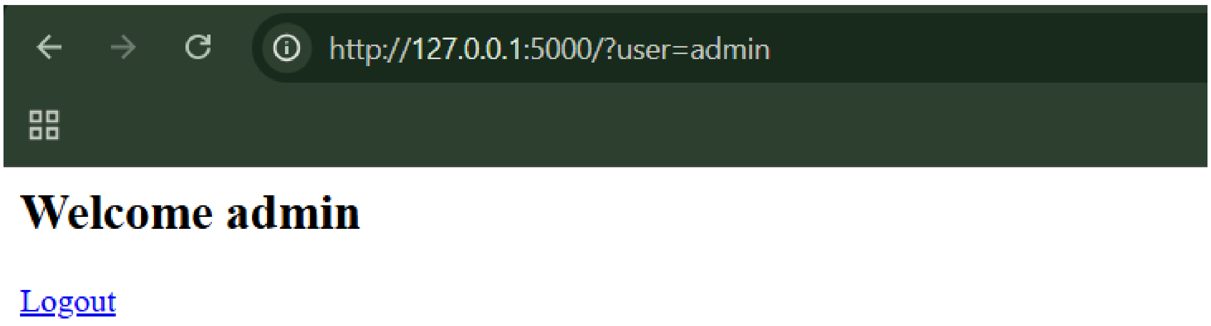


| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| _cb | BwXWuoD6Gu6CDCfPoj | 127.0.0.1 | / | 2026-09-... | 21 | | | | | | Medium |
| _chartbeat2 | .1754653775235.1754655732420.1.CLO88tCVFv5wCZAI4gCj... | 127.0.0.1 | / | 2026-09-... | 74 | | | | | | Medium |
| _ga | GA1.1.316767436.1754653776 | 127.0.0.1 | / | 2026-09-... | 29 | | | | | | Medium |
| _ga_HEQWL2KPC5 | GS2.1.s1754653775$o1$g1$t1754656284$j60$l0$h4675013... | 127.0.0.1 | / | 2026-09-... | 67 | | | | | | Medium |
| _parsely_visitor | {%22id%22:%22pid=831b63ff3c4ad93786c5916cbdec10fc%... | 127.0.0.1 | / | 2026-09-... | 131 | | | | | | Medium |
| session_user | alice | 127.0.0.1 | / | Session | 17 | | | | | | Medium |

Cookie Value ☐ Show URL-decoded
alice

Notice session_user=alice. That cookie is the session token

Step4: Cookie vs URL token confusion. Try this without logging in: http://127.0.0.1:5000/?user=admin

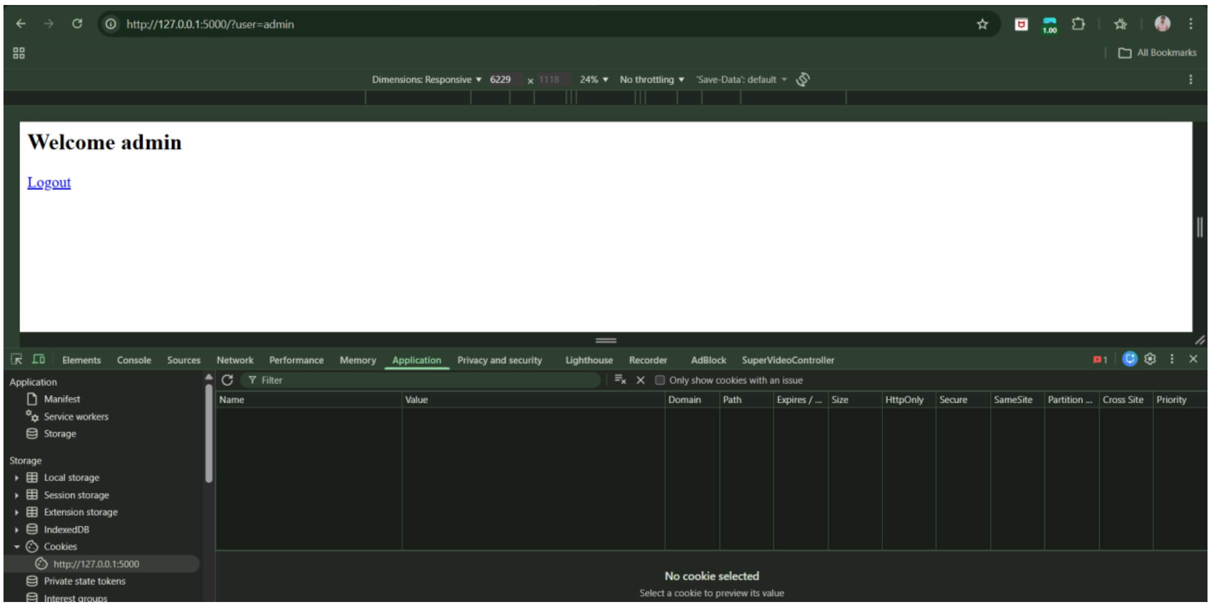


http://127.0.0.1:5000/?user=admin

Welcome admin

We are now “logged in” as admin. This shows multiple session sources (URL + cookie)

Step5: Removing tokens one by one. I just removed all the cookies, and refreshed the page, but it still allowed me to stay as admin.



http://127.0.0.1:5000/?user=admin

Welcome admin

Logout

No cookie selected

Select a cookie to preview its value

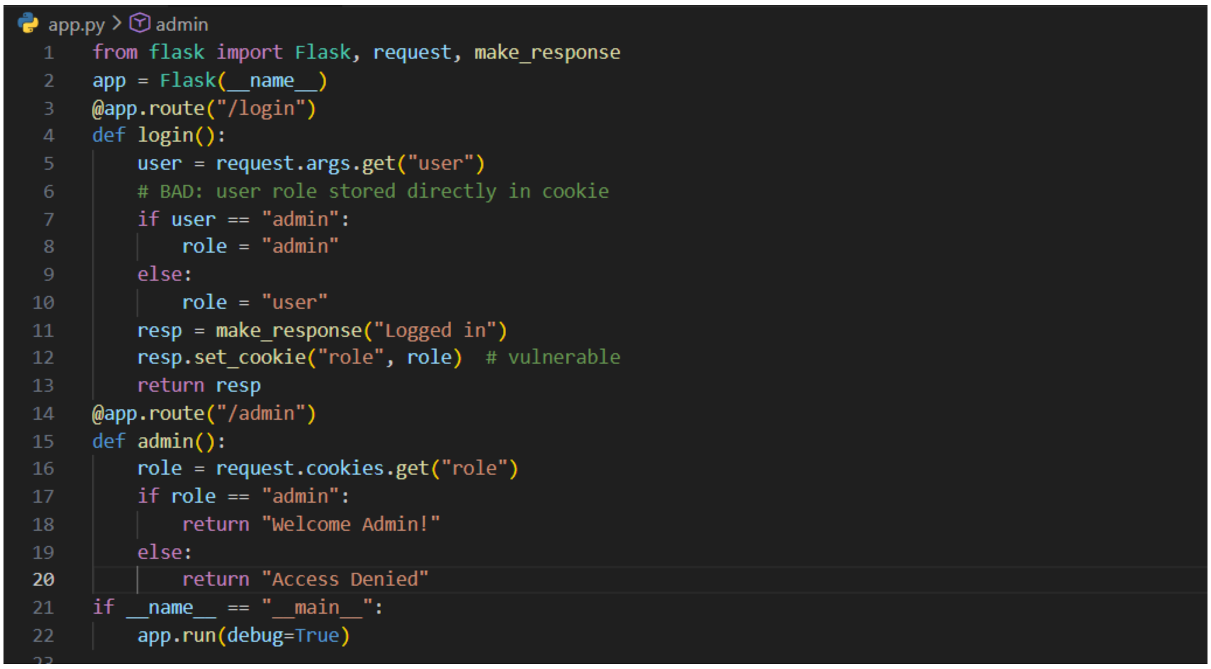
Dimensions: Responsive 6229 x 1118 24% No throttling 'Save-Data': default
Elements Console Sources Network Performance Memory Application Privacy and security Lighthouse Recorder AdBlock SuperVideoController
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
Extension storage
IndexedDB
Cookies
http://127.0.0.1:5000
Private state tokens
Interest groups
Filter
Only show cookies with an issue
Name Value Domain Path Expires / ... Size HttpOnly Secure SameSite Partition ... Cross Site Priority

## Alternatives to Sessions:

Some web applications don't use sessions to track users. Instead, they either rely on HTTP authentication (where the browser sends login details with every request) or store all user state on the client in encrypted or signed data like cookies or hidden fields. When these sessionless methods are used, traditional session-based attacks usually won't work, so testers should look for other vulnerabilities instead.

## Trying to manipulate the sessions on a local app:

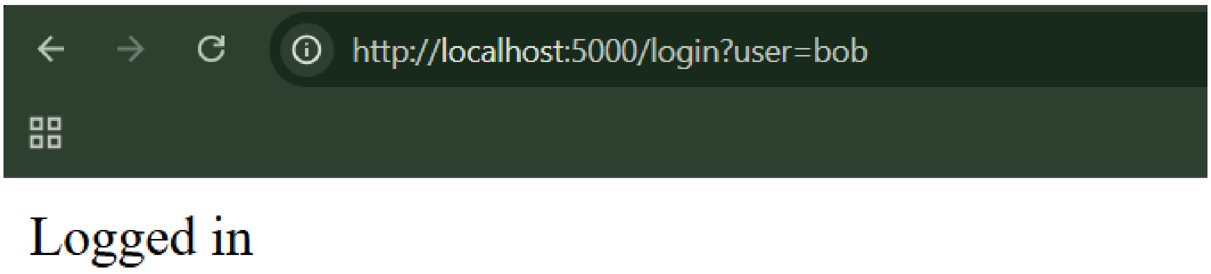
App.py:



app.py > admin

```python
from flask import Flask, request, make_response
app = Flask(__name__)
@app.route("/login")
def login():
    user = request.args.get("user")
    # BAD: user role stored directly in cookie
    if user == "admin":
        role = "admin"
    else:
        role = "user"
    resp = make_response("Logged in")
    resp.set_cookie("role", role)  # vulnerable
    return resp
@app.route("/admin")
def admin():
    role = request.cookies.get("role")
    if role == "admin":
        return "Welcome Admin!"
    else:
        return "Access Denied"
if __name__ == "__main__":
    app.run(debug=True)
```

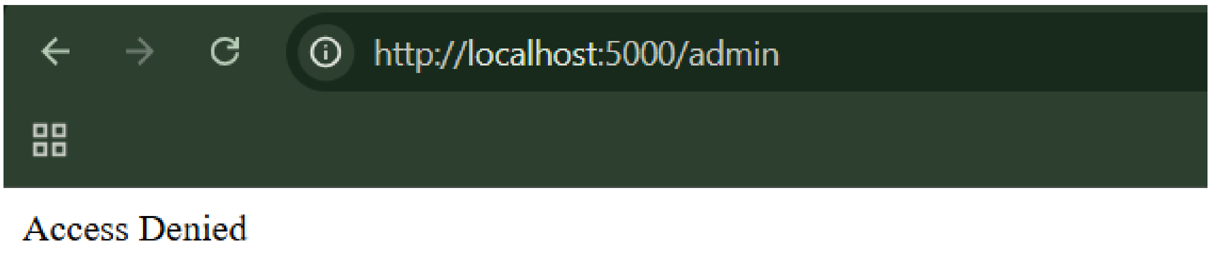
Step1: Log in as a normal user


http://localhost:5000/login?user=bob

Logged in

This will Cookie set: role=user

Step2: Try accessing admin page

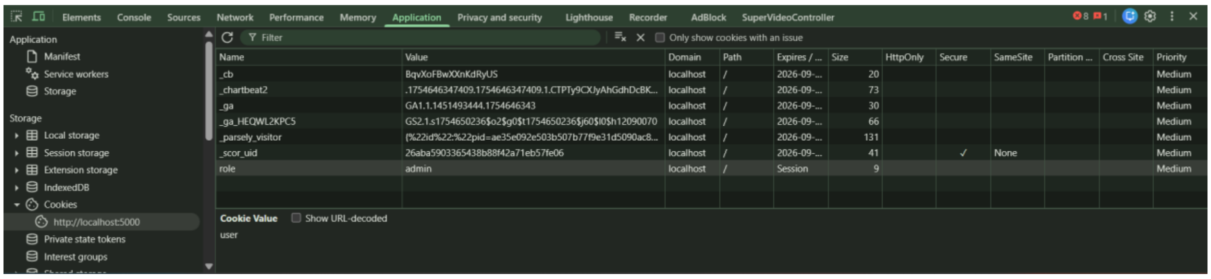

http://localhost:5000/admin

Access Denied

Correct behaviour so far.

Step3: Modify the cookie (the attack)

Using browser dev tools, change the role from role=user to role=admin.

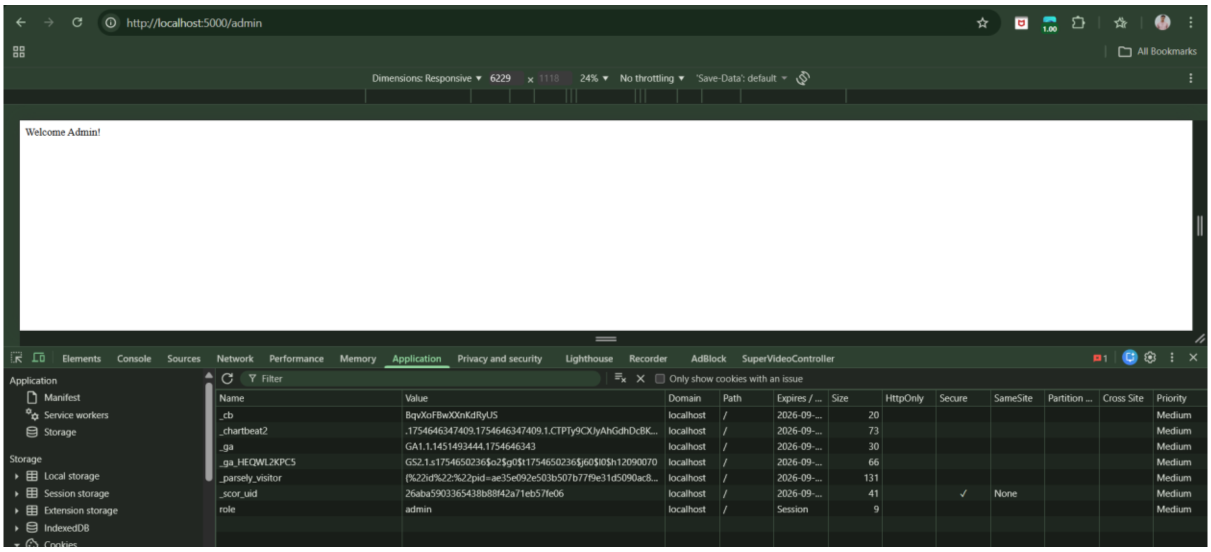


| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| _cb | BqvXoFBwXXnKdRyUS | localhost | / | 2026-09-... | 20 | | | | | | Medium |
| _chartbeat2 | .1754646347409.1754646347409.1.CTPTy9CXJyAhGdhDcBK... | localhost | / | 2026-09-... | 73 | | | | | | Medium |
| _ga | GA1.1.1451493444.1754646343 | localhost | / | 2026-09-... | 30 | | | | | | Medium |
| _ga_HEQWL2KPC5 | GS2.1.s1754650236$o2$g0$t1754650236$j60$l0$h12090070 | localhost | / | 2026-09-... | 66 | | | | | | Medium |
| _parsely_visitor | {%22id%22:%22pid=ae35e092e503b507b77f9e31d5090ac8... | localhost | / | 2026-09-... | 131 | | | | | | Medium |
| _scor_uid | 26aba5903365438b88f42a71eb57fe06 | localhost | / | 2026-09-... | 41 | | ✓ | None | | | Medium |
| role | admin | localhost | / | Session | 9 | | | | | | Medium |

**Cookie Value**

user

Refresh the page:



Welcome Admin!

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| _cb | BqvXoFBwXXnKdRyUS | localhost | / | 2026-09-... | 20 | | | | | | Medium |
| _chartbeat2 | .1754646347409.1754646347409.1.CTPTy9CXJyAhGdhDcBK... | localhost | / | 2026-09-... | 73 | | | | | | Medium |
| _ga | GA1.1.1451493444.1754646343 | localhost | / | 2026-09-... | 30 | | | | | | Medium |
| _ga_HEQWL2KPC5 | GS2.1.s1754650236$o2$g0$t1754650236$j60$l0$h12090070 | localhost | / | 2026-09-... | 66 | | | | | | Medium |
| _parsely_visitor | {%22id%22:%22pid=ae35e092e503b507b77f9e31d5090ac8... | localhost | / | 2026-09-... | 131 | | | | | | Medium |
| _scor_uid | 26aba5903365438b88f42a71eb57fe06 | localhost | / | 2026-09-... | 41 | | ✓ | None | | | Medium |
| role | admin | localhost | / | Session | 9 | | | | | | Medium |

What just happened?

- The app trusted client-side data
- No session ID was used
- No cryptographic protection
- You changed your privileges yourself

This is a classic broken sessionless state vulnerability.

## Weaknesses in Session Token Generation:

- Meaningful Tokens
- Predictable Tokens
  - Concealed Sequences
  - Time Dependency
  - Weak Random Number Generation

### Meaningful Tokens:

If session tokens are predictable or contain real user data, attackers can guess them, steal sessions, and log in as other users.

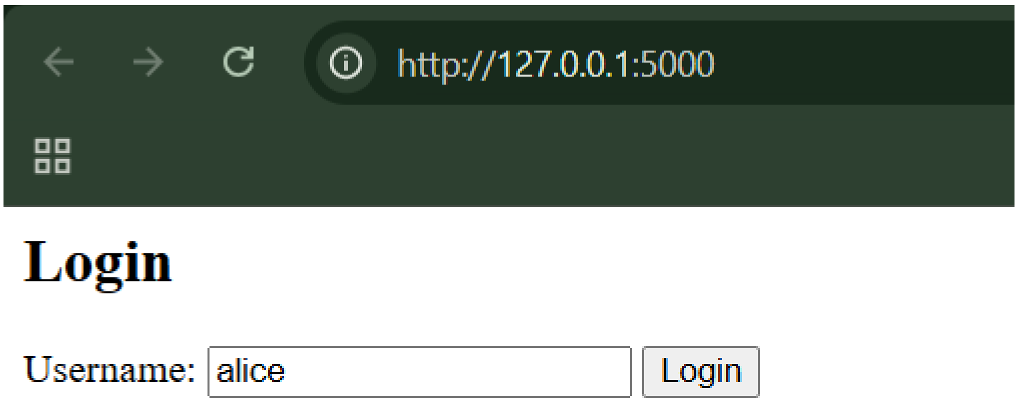
A secure session token should be:

- Completely random
- Impossible to predict
- Meaningless if decoded

If a token contains patterns, user data, or predictable structure, it's a serious security flaw.

**Testing on local website where tokens be not secured:**

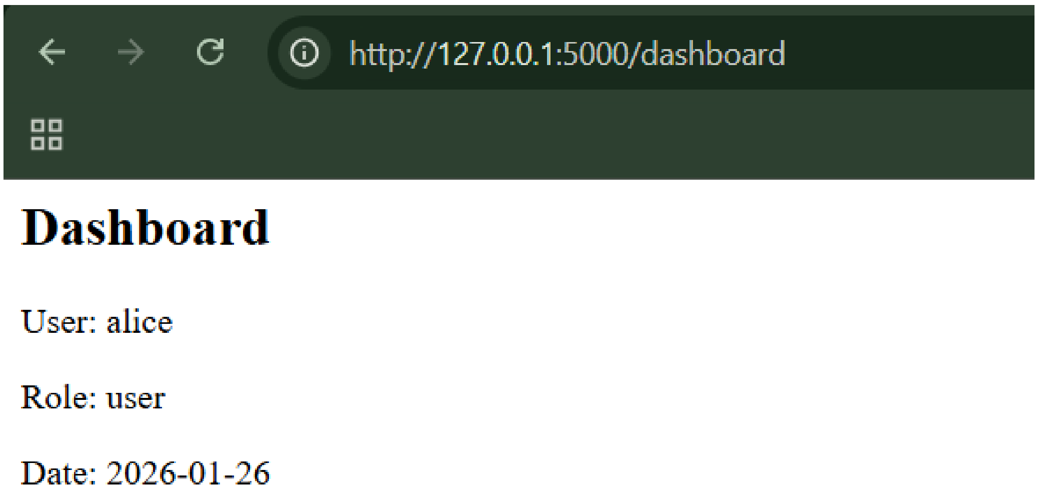
Step1: Log in as a normal user


http://127.0.0.1:5000

# Login

Username: alice Login

Output:


http://127.0.0.1:5000/dashboard

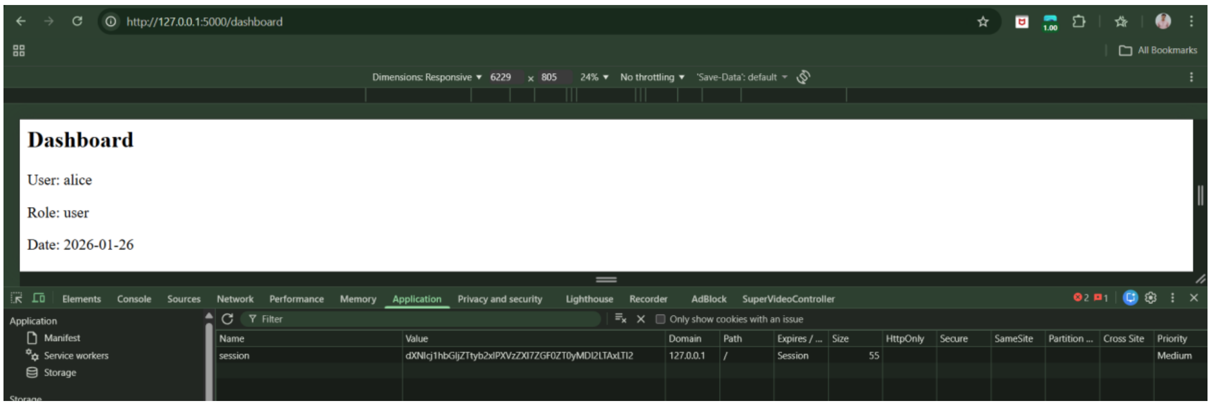
# Dashboard

User: alice

Role: user

Date: 2026-01-26

Step2: Look at the cookie. Open browser dev tools → Cookies → session


http://127.0.0.1:5000/dashboard

Dimensions: Responsive 6229 x 805 24% No throttling 'Save-Data': default

Dashboard

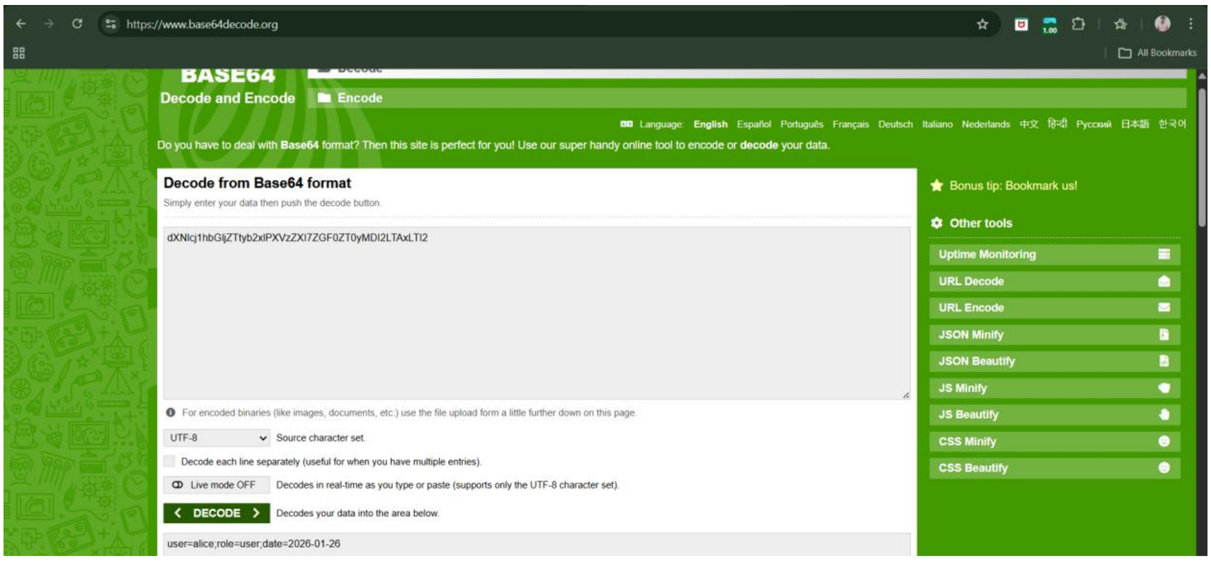
User: alice

Role: user

Date: 2026-01-26

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | dXNlcj1hbGljZTtyb2xlPXVzZXI7ZGF0ZT0yMDI2LTAxLTI2 | 127.0.0.1 | / | Session | 55 | | | | | | Medium |

Step3: Decode the value “dXNlcj1hbGljZTtyb2xlPXVzZXI7ZGF0ZT0yMDI2LTAxLTI2”.


https://www.base64decode.org

Decode from Base64 format

Simply enter your data then push the decode button.

```
dXNlcj1hbGljZTtyb2xlPXVzZXI7ZGF0ZT0yMDI2LTAxLTI2
```

UTF-8 Source character set.

Decode each line separately (useful for when you have multiple entries).

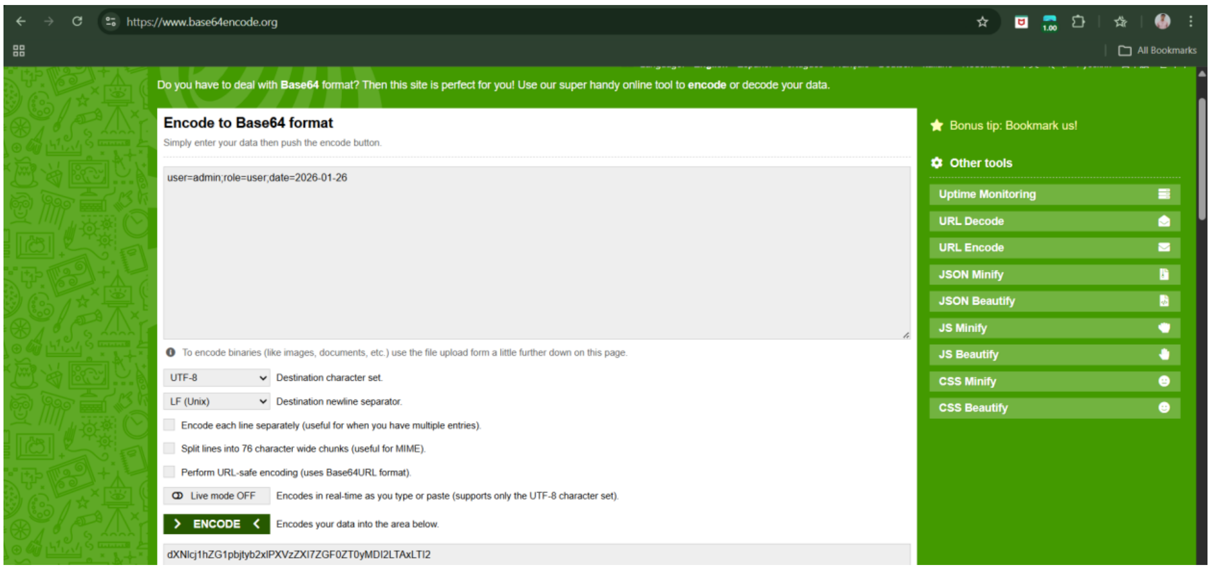
Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

DECODE Decodes your data into the area below.

```
user=alice;role=user;date=2026-01-26
```

Clearly, we can see user=alice;role=user;date=2026-01-26. That’s the problem — the token literally tells you everything.

Step4: Modify the token, change the user = alice to user = admin and then encode it, like this:


https://www.base64encode.org

Encode to Base64 format

Simply enter your data then push the encode button.

```
user=admin;role=user;date=2026-01-26
```

UTF-8 Destination character set.

LF (Unix) Destination newline separator.

Encode each line separately (useful for when you have multiple entries).

Split lines into 76 character wide chunks (useful for MIME).

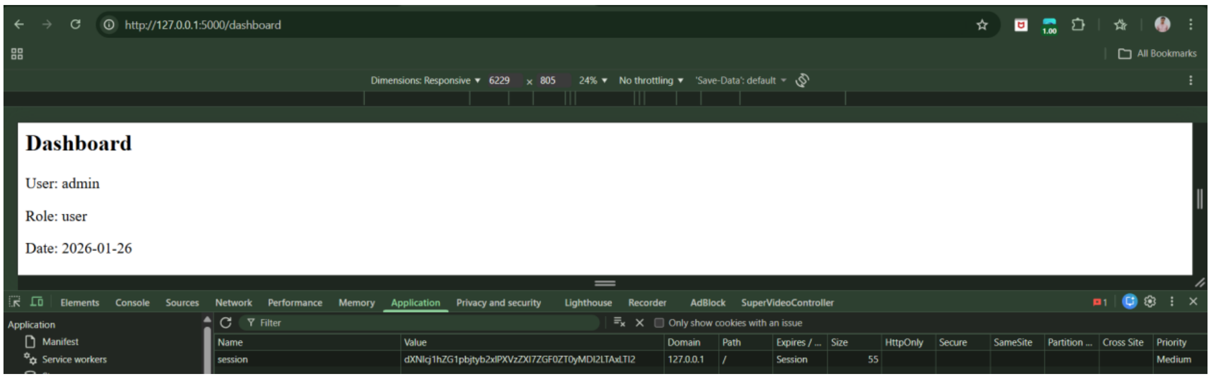
Perform URL-safe encoding (uses Base64URL format).

Live mode OFF Encodes in real-time as you type or paste (supports only the UTF-8 character set).

ENCODE Encodes your data into the area below.

```
dXNlcj1hZG1pbjtyb2xlPXVzZXI7ZGF0ZT0yMDI2LTAxLTI2
```

Re-encode to Base64, replace the cookie value, refresh the page.


http://127.0.0.1:5000/dashboard

Dashboard

User: admin

Role: user

Date: 2026-01-26

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | dXNlcj1hZG1pbjtyb2xlPXVzZXI7ZGF0ZT0yMDI2LTAxLTI2 | 127.0.0.1 | / | Session | 55 | | | | | | Medium |

We are now “admin”. No password. No crypto. No randomness. Why this is insecure? The server trusts user-controlled, predictable, meaningful session data instead of generating a random, unguessable token.

**<u>Predictable Tokens:</u>**

**Why this is dangerous?**

Even if a token doesn’t directly contain your username or data, it can still be unsafe if:

- Tokens are issued in a sequence (like counting up)
- Tokens are based on time
- Tokens use weak randomness

If an attacker gets a few real tokens, they can guess other valid ones and take over other users’ sessions. Even guessing 1 valid token out of 1,000 tries is enough when a computer is doing the guessing automatically.

**<u>Concealed Sequences:</u>**

- Some session tokens appear random but contain hidden sequences.
- The pattern is concealed using encoding (e.g., Base64).
- Encoding makes tokens look complex but does not add security.
- When tokens are decoded/unpacked, numeric patterns may appear.
- Tokens may be generated by:
    - Adding a fixed value to the previous token
    - Truncating to a fixed size (e.g., 32-bit)
    - Encoding the result for HTTP transport

    - Attackers can:
        - Decode tokens (Base64, hex, etc.)
        - Analyse numeric differences
        - Predict past and future tokens
    - This allows session hijacking without authentication.
    - Secure session tokens must use cryptographically strong randomness, not sequences.

**<u>Time Dependency:</u>**

Time dependency occurs when session tokens are generated using the time of creation as part of their value. If the application does not include enough randomness, the tokens become predictable even though they may appear random at first. Often, such tokens contain a sequential counter combined with a timestamp (for example, milliseconds since a fixed point in time).

By collecting multiple tokens and comparing them over time, an attacker can identify skipped sequence numbers issued to other users and estimate the time range in which those tokens were created. Because this time window is usually small, the attacker can brute-force possible values and successfully guess valid session tokens, leading to session hijacking. Therefore, secure session management must avoid relying on timestamps and instead use cryptographically strong random values.

**Weak Random Number Generation:**

Weak random number generation occurs when session tokens are created using predictable pseudo-random algorithms instead of true cryptographic randomness. Computers cannot generate true randomness, so they rely on algorithms that only *appear* random but actually produce numbers in a fixed sequence. If an attacker obtains a small sample of values from such a generator, they may be able to calculate past and future values with complete accuracy.

For example, Jetty's session management once used Java's java.util.Random, which is based on a linear congruential generator that derives each new value from the previous one using fixed constants. Because the algorithm is known, an attacker who captures a single session token can predict other users' tokens and hijack sessions. Simply combining multiple outputs from the same weak generator does not solve the problem and can even make attacks easier by revealing the generator's internal state.

## Weaknesses in Session Token Handling:

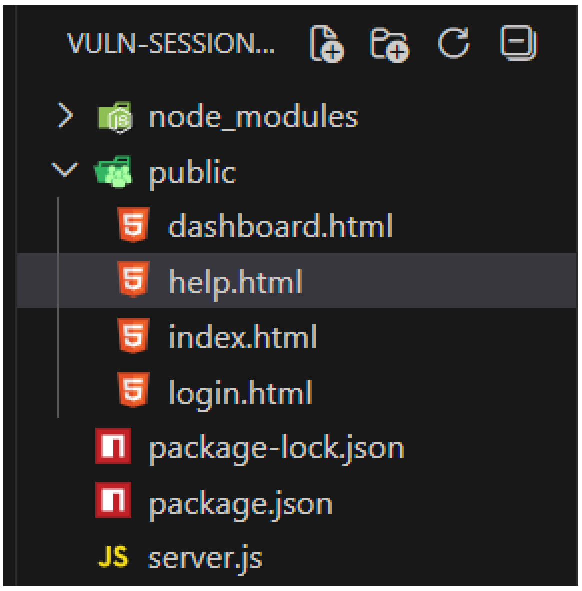
- Disclosure of Tokens on the Network
- Disclosure of Tokens in Logs
- Vulnerable Mapping of Tokens to Sessions
- Vulnerable Session Termination
- Client Exposure to Token Hijacking
- Liberal Cookie Scope
- Cookie Domain Restrictions
- Cookie Path Restrictions

**Disclosure of Tokens on the Network:**

If your session token is sent without encryption, anyone watching the network can steal it.

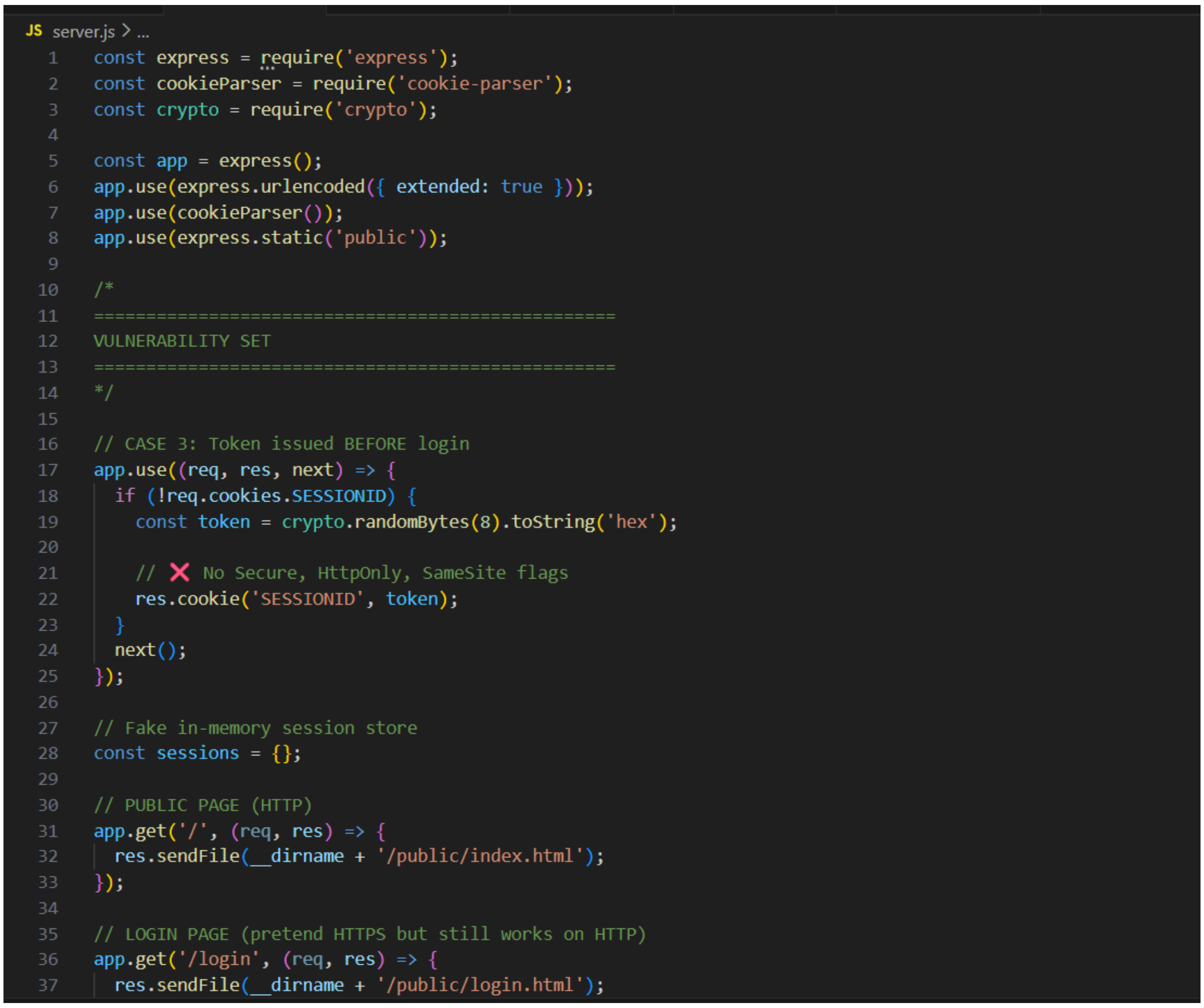
**Observing Disclosure of Tokens on the Network for a local website:**

Create this folder structure:



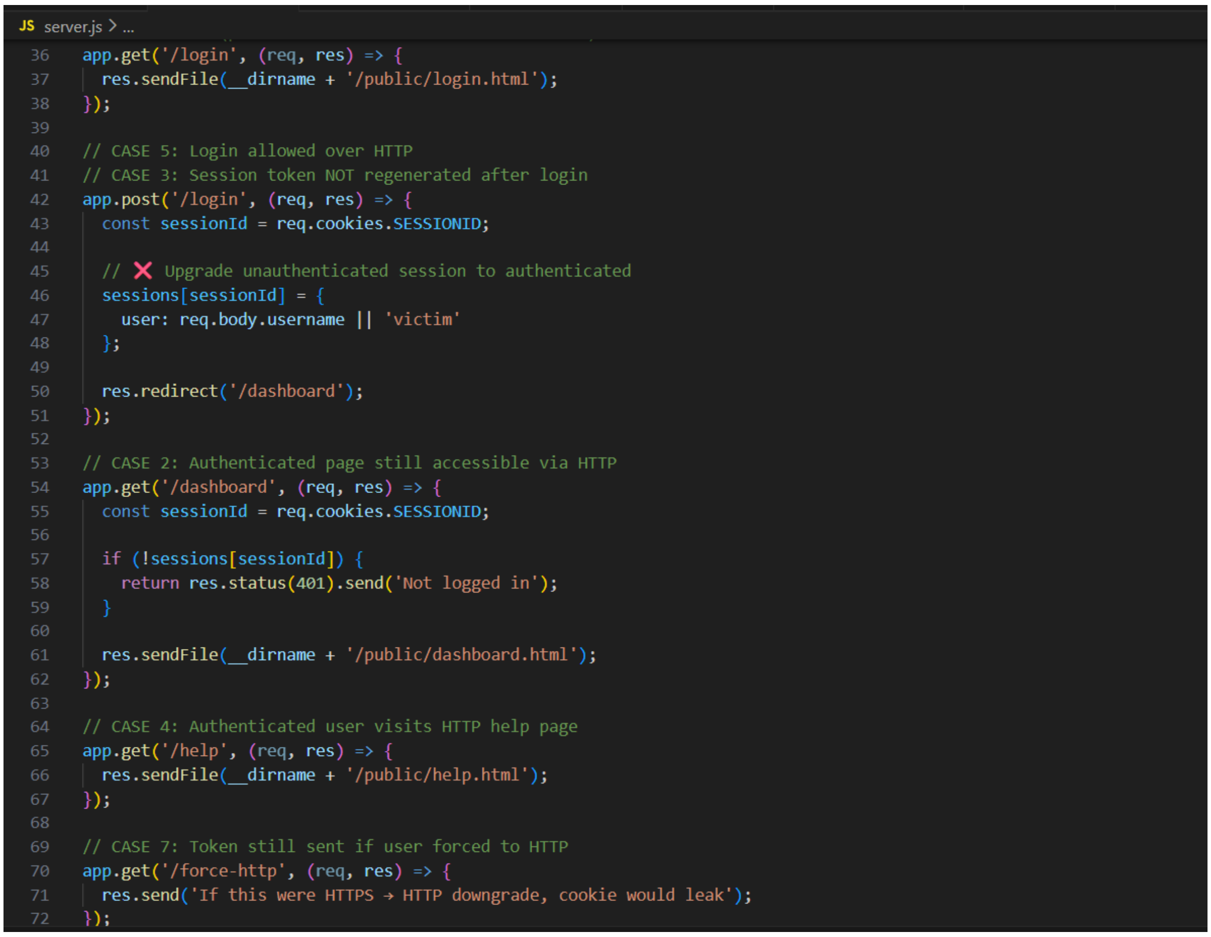
```
VULN-SESSION...
> node_modules
v public
    dashboard.html
    help.html
    index.html
    login.html
  package-lock.json
  package.json
  server.js
```

Server.js:

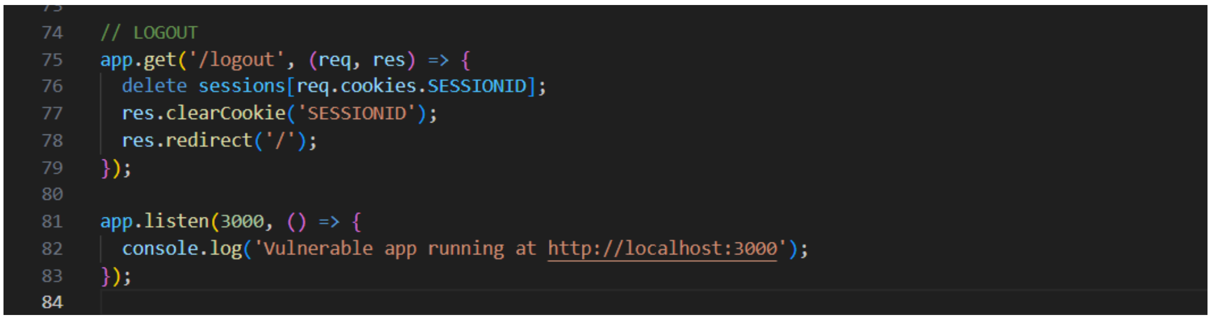


server.js > ...

```
const express = require('express');
const cookieParser = require('cookie-parser');
const crypto = require('crypto');

const app = express();
app.use(express.urlencoded({ extended: true }));
app.use(cookieParser());
app.use(express.static('public'));

/*
==================================================
VULNERABILITY SET
==================================================
*/

// CASE 3: Token issued BEFORE login
app.use((req, res, next) => {
  if (!req.cookies.SESSIONID) {
    const token = crypto.randomBytes(8).toString('hex');

    // ❌ No Secure, HttpOnly, SameSite flags
    res.cookie('SESSIONID', token);
  }
  next();
});

// Fake in-memory session store
const sessions = {};

// PUBLIC PAGE (HTTP)
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// LOGIN PAGE (pretend HTTPS but still works on HTTP)
app.get('/login', (req, res) => {
  res.sendFile(__dirname + '/public/login.html');
```

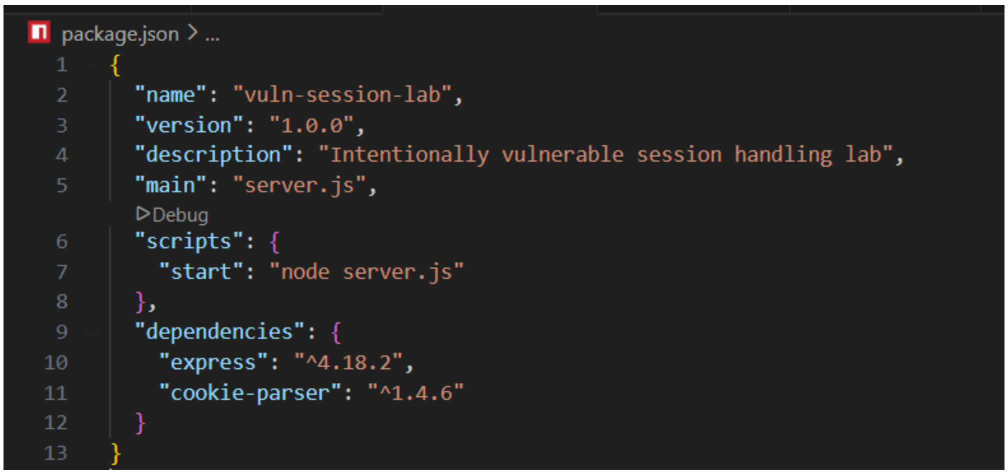


```javascript
app.get('/login', (req, res) => {
  res.sendFile(__dirname + '/public/login.html');
});

// CASE 5: Login allowed over HTTP
// CASE 3: Session token NOT regenerated after login
app.post('/login', (req, res) => {
  const sessionId = req.cookies.SESSIONID;

  // ❌ Upgrade unauthenticated session to authenticated
  sessions[sessionId] = {
    user: req.body.username || 'victim'
  };

  res.redirect('/dashboard');
});

// CASE 2: Authenticated page still accessible via HTTP
app.get('/dashboard', (req, res) => {
  const sessionId = req.cookies.SESSIONID;

  if (!sessions[sessionId]) {
    return res.status(401).send('Not logged in');
  }

  res.sendFile(__dirname + '/public/dashboard.html');
});

// CASE 4: Authenticated user visits HTTP help page
app.get('/help', (req, res) => {
  res.sendFile(__dirname + '/public/help.html');
});

// CASE 7: Token still sent if user forced to HTTP
app.get('/force-http', (req, res) => {
  res.send('If this were HTTPS → HTTP downgrade, cookie would leak');
});
```



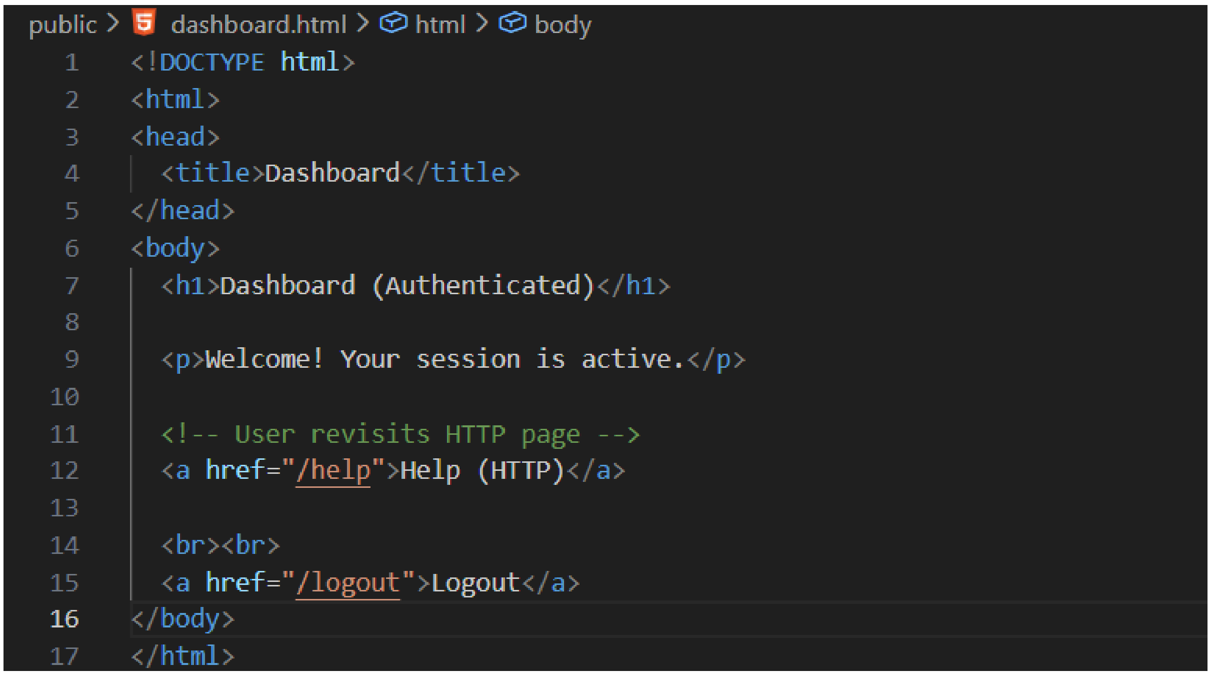
```javascript
// LOGOUT
app.get('/logout', (req, res) => {
  delete sessions[req.cookies.SESSIONID];
  res.clearCookie('SESSIONID');
  res.redirect('/');
});

app.listen(3000, () => {
  console.log('Vulnerable app running at http://localhost:3000');
});
```

Package.json:



```json
{
  "name": "vuln-session-lab",
  "version": "1.0.0",
  "description": "Intentionally vulnerable session handling lab",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cookie-parser": "^1.4.6"
  }
}
```

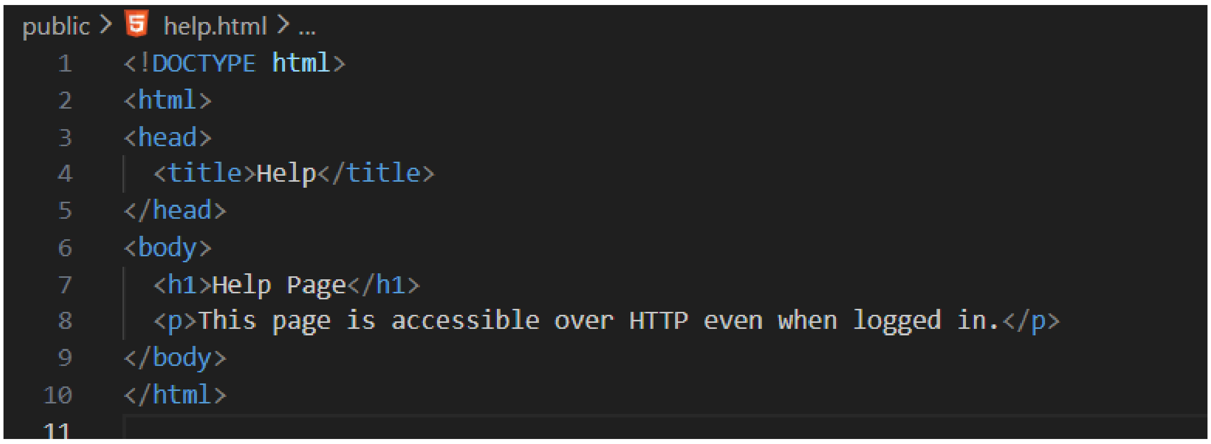
Dashboard.html:



public > dashboard.html > html > body

```html
<!DOCTYPE html>
<html>
<head>
  <title>Dashboard</title>
</head>
<body>
  <h1>Dashboard (Authenticated)</h1>

  <p>Welcome! Your session is active.</p>

  <!-- User revisits HTTP page -->
  <a href="/help">Help (HTTP)</a>

  <br><br>
  <a href="/logout">Logout</a>
</body>
</html>
```

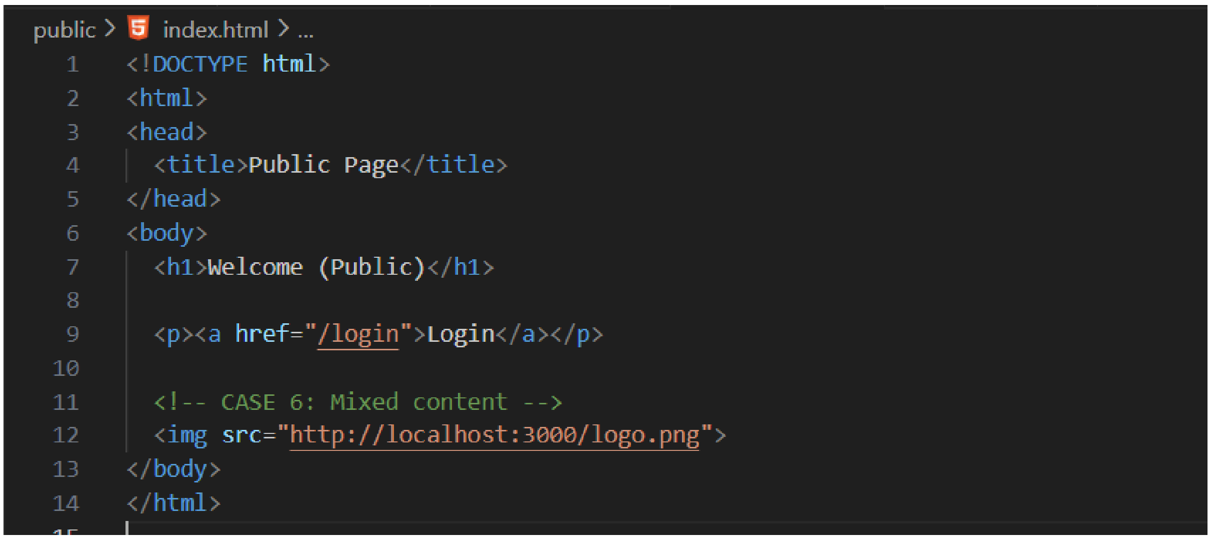
Help.html:



public > help.html > ...

```html
<!DOCTYPE html>
<html>
<head>
  <title>Help</title>
</head>
<body>
  <h1>Help Page</h1>
  <p>This page is accessible over HTTP even when logged in.</p>
</body>
</html>
```

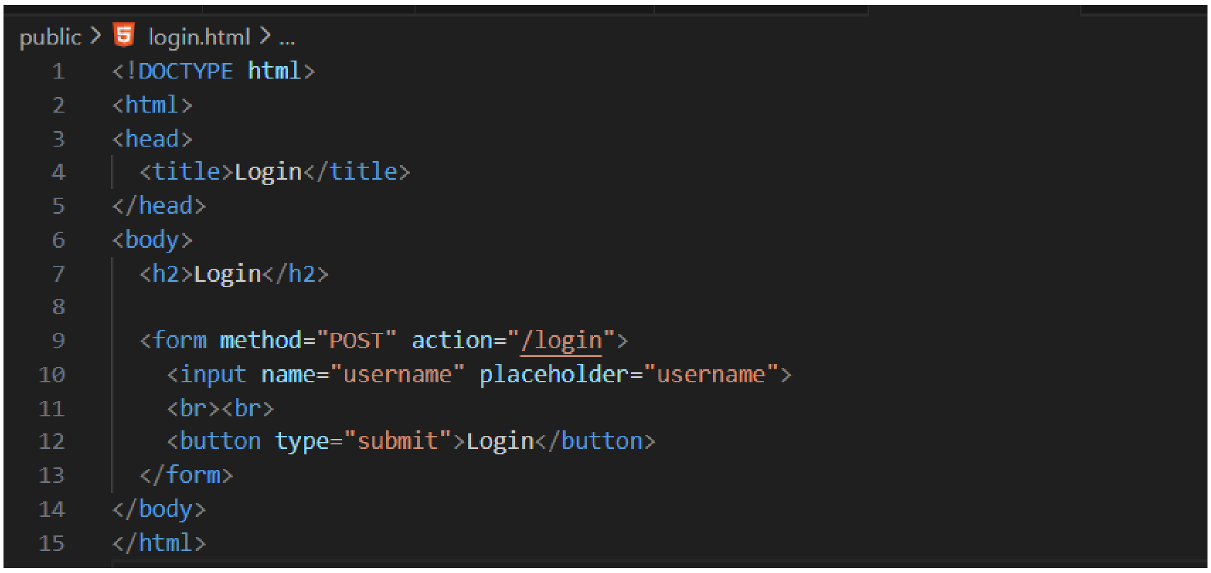
Index.html:



public > index.html > ...

```html
<!DOCTYPE html>
<html>
<head>
  <title>Public Page</title>
</head>
<body>
  <h1>Welcome (Public)</h1>

  <p><a href="/login">Login</a></p>

  <!-- CASE 6: Mixed content -->
  <img src="http://localhost:3000/logo.png">
</body>
</html>
```

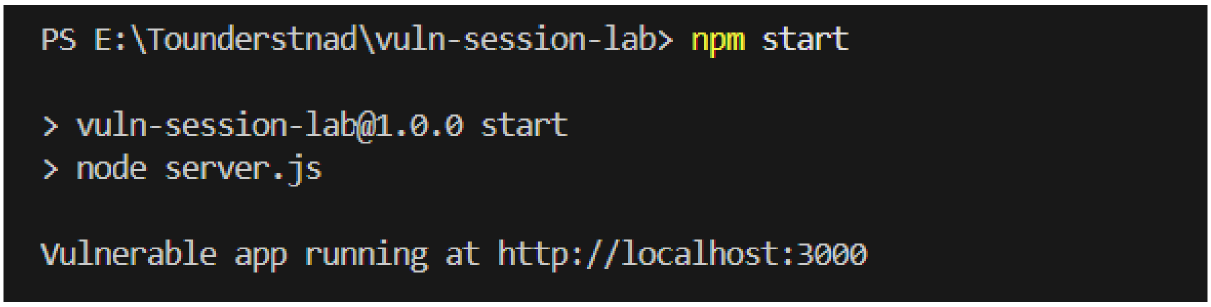
Login.html:



public > login.html > ...

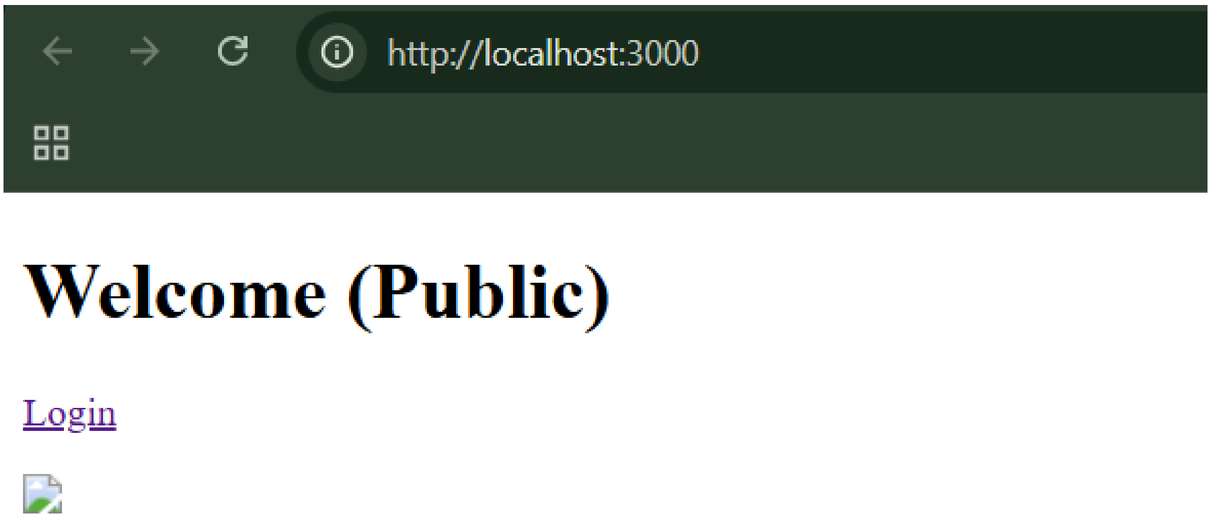
```html
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
</head>
<body>
  <h2>Login</h2>

  <form method="POST" action="/login">
    <input name="username" placeholder="username">
    <br><br>
    <button type="submit">Login</button>
  </form>
</body>
</html>
```

Start the server:



```
PS E:\Tounderstnad\vuln-session-lab> npm start

> vuln-session-lab@1.0.0 start
> node server.js

Vulnerable app running at http://localhost:3000
```

Browser:

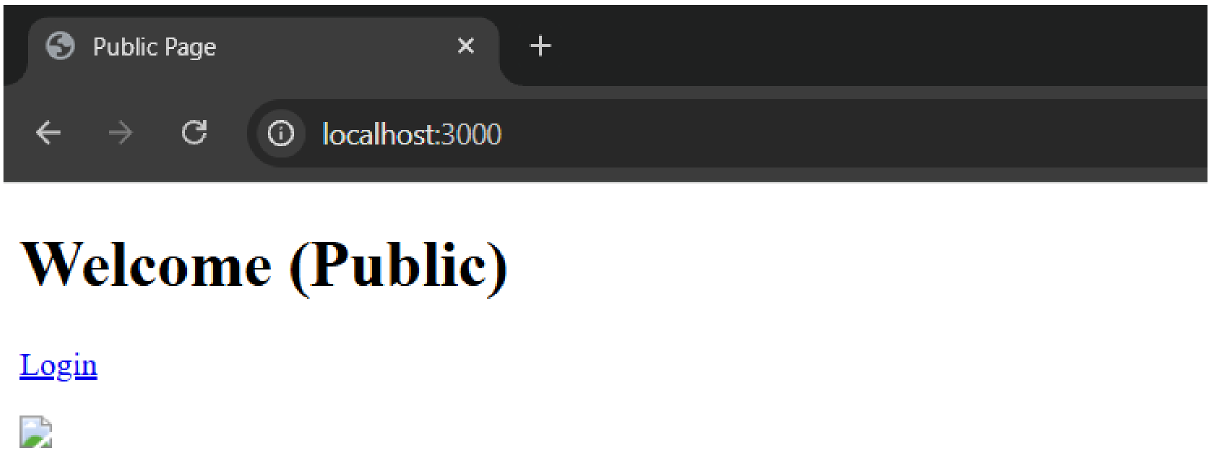


http://localhost:3000

Welcome (Public)

Login

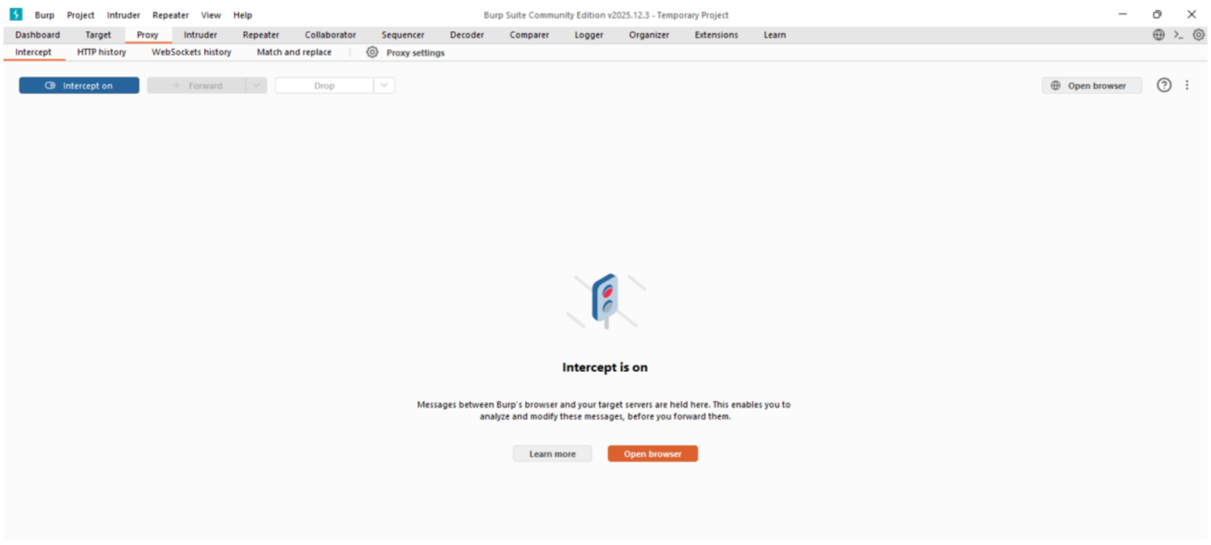
Step1:  Open the burp, and visit the page.



Welcome (Public)

Login

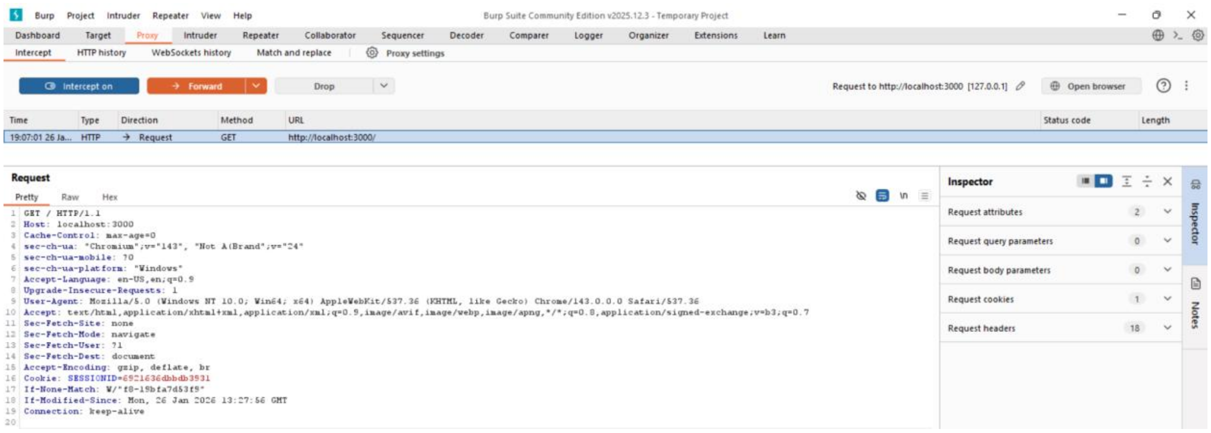
Make the intercept on:



Intercept is on

Messages between Burp's browser and your target servers are held here. This enables you to analyze and modify these messages, before you forward them.

See the request intercept: clearly, HTTP. Also, Cookie visible in cleartext.


Burp Suite Community Edition v2025.12.3 - Temporary Project
Intercept on
Forward
Drop
Request to http://localhost:3000 [127.0.0.1]
Open browser
Request
Pretty
Raw
Hex
Inspector
Request attributes 2
Request query parameters 0
Request body parameters 0
Request cookies 1
Request headers 18
Notes

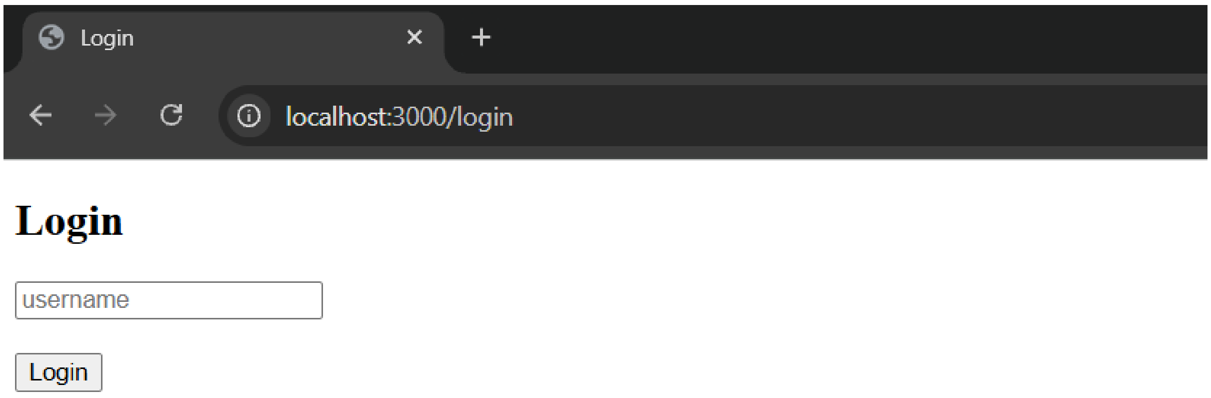
```
Time              Type   Direction    Method   URL                          Status code   Length
19:07:01 26 Ja... HTTP   → Request    GET      http://localhost:3000/
```

```
GET / HTTP/1.1
Host: localhost:3000
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Cookie: SESSIONID=6921636dbbdb3931
If-None-Match: W/"f8-19bfa7d53f9"
If-Modified-Since: Mon, 26 Jan 2026 13:27:56 GMT
Connection: keep-alive
```

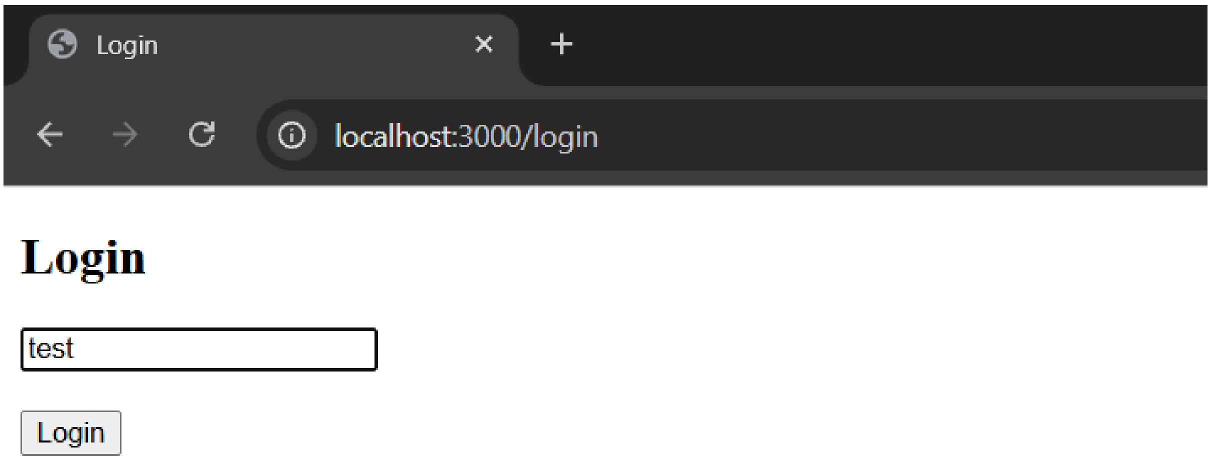
Session cookie transmitted over unencrypted HTTP, allowing network interception.

Step2: Now Authenticated session over HTTP. Show logged-in session still uses HTTP. Visit the login page.

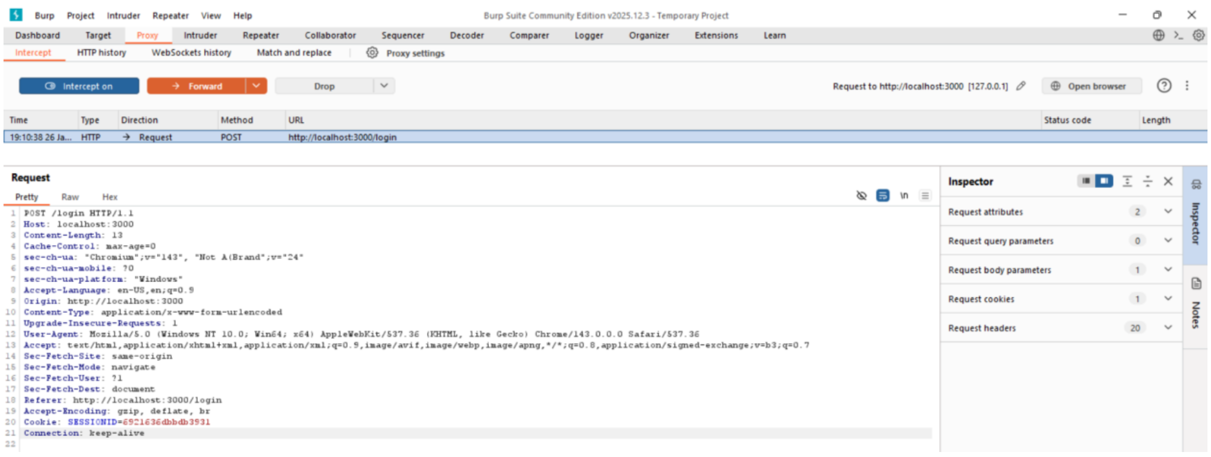
Browser: http://localhost:3000/login



Enter anything:



See the captured in burp:


Burp Suite Community Edition v2025.12.3 - Temporary Project
Proxy
Intercept
Intercept on
Forward
Drop
Request to http://localhost:3000 [127.0.0.1]
Open browser
Inspector
Request attributes 2
Request query parameters 0
Request body parameters 1
Request cookies 1
Request headers 20

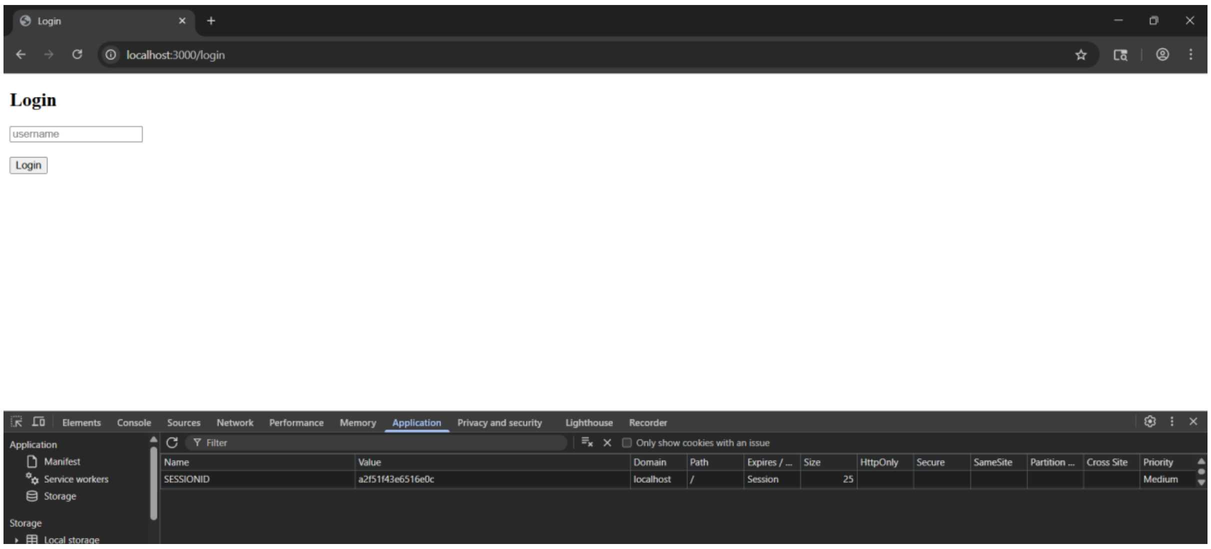
| Time | Type | Direction | Method | URL | Status code | Length |
|---|---|---|---|---|---|---|
| 19:10:38 26 Ja... | HTTP | → Request | POST | http://localhost:3000/login | | |

Request

```
POST /login HTTP/1.1
Host: localhost:3000
Content-Length: 13
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Origin: http://localhost:3000
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:3000/login
Accept-Encoding: gzip, deflate, br
Cookie: SESSIONID=6921636dbbdb3931
Connection: keep-alive

```

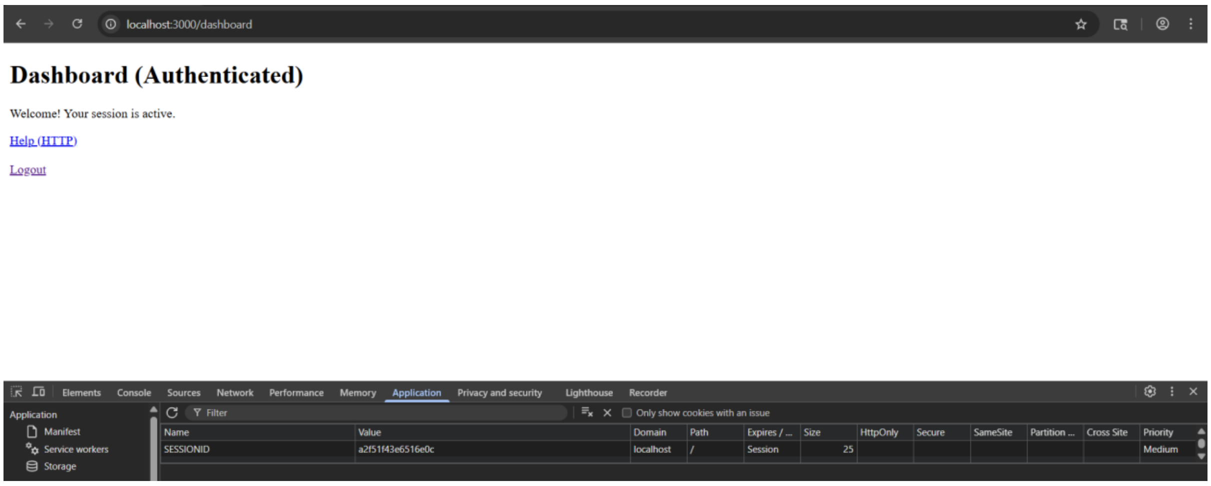
Session cookie sent over HTTP after login. Application continues using HTTP after authentication, exposing session tokens.

Step3: Prove session ID does not change after login. For this, visit the login page and then, see the session id in the dev tools:


Login
localhost:3000/login
Login
username
Login
Elements Console Sources Network Performance Memory Application Privacy and security Lighthouse Recorder
Application
Manifest
Service workers
Storage
Storage
Local storage
Filter
Only show cookies with an issue

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SESSIONID | a2f51f43e6516e0c | localhost | / | Session | 25 | | | | | | Medium |

Now, login and again check:


localhost:3000/dashboard
Elements Console Sources Network Performance Memory Application Privacy and security Lighthouse Recorder
Application
Manifest
Service workers
Storage
Filter
Only show cookies with an issue

Dashboard (Authenticated)

Welcome! Your session is active.

Help (HTTP)

Logout

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SESSIONID | a2f51f43e6516e0c | localhost | / | Session | 25 | | | | | | Medium |

Clearly, Same token reused. Session fixation vulnerability due to missing token regeneration after authentication.

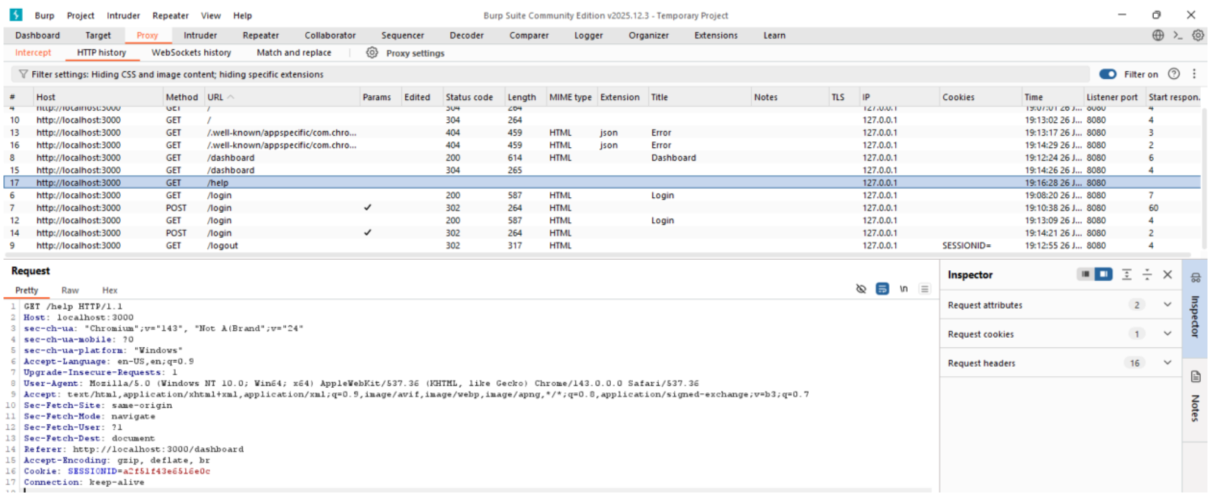
Step4: Show token leaks when visiting public HTTP page.

1. Login
2. On dashboard, click:

   *Help*

3. Burp intercepts:

   *GET /help*


Filter settings: Hiding CSS and image content; hiding specific extensions

| # | Host | Method | URL | Params | Edited | Status code | Length | MIME type | Extension | Title | Notes | TLS | IP | Cookies | Time | Listener port | Start respon. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | http://localhost:3000 | GET | / | | | 304 | 264 | | | | | | 127.0.0.1 | | 19:07:01 26 J... | 8080 | 4 |
| 10 | http://localhost:3000 | GET | / | | | 304 | 264 | | | | | | 127.0.0.1 | | 19:13:02 26 J... | 8080 | 4 |
| 13 | http://localhost:3000 | GET | /.well-known/appspecific/com.chro... | | | 404 | 459 | HTML | json | Error | | | 127.0.0.1 | | 19:13:17 26 J... | 8080 | 3 |
| 16 | http://localhost:3000 | GET | /.well-known/appspecific/com.chro... | | | 404 | 459 | HTML | json | Error | | | 127.0.0.1 | | 19:14:29 26 J... | 8080 | 2 |
| 8 | http://localhost:3000 | GET | /dashboard | | | 200 | 614 | HTML | | Dashboard | | | 127.0.0.1 | | 19:12:24 26 J... | 8080 | 6 |
| 15 | http://localhost:3000 | GET | /dashboard | | | 304 | 265 | | | | | | 127.0.0.1 | | 19:14:26 26 J... | 8080 | 4 |
| 17 | http://localhost:3000 | GET | /help | | | | | | | | | | 127.0.0.1 | | 19:16:28 26 J... | 8080 | |
| 6 | http://localhost:3000 | GET | /login | | | 200 | 587 | HTML | | Login | | | 127.0.0.1 | | 19:08:20 26 J... | 8080 | 7 |
| 7 | http://localhost:3000 | POST | /login | ✓ | | 302 | 264 | HTML | | | | | 127.0.0.1 | | 19:10:38 26 J... | 8080 | 60 |
| 12 | http://localhost:3000 | GET | /login | | | 200 | 587 | HTML | | Login | | | 127.0.0.1 | | 19:13:09 26 J... | 8080 | 4 |
| 14 | http://localhost:3000 | POST | /login | ✓ | | 302 | 264 | HTML | | | | | 127.0.0.1 | | 19:14:21 26 J... | 8080 | 2 |
| 9 | http://localhost:3000 | GET | /logout | | | 302 | 317 | HTML | | | | | 127.0.0.1 | SESSIONID= | 19:12:55 26 J... | 8080 | 4 |

Request

```
GET /help HTTP/1.1
Host: localhost:3000
sec-ch-ua: "Chromium";v="143", "Not A(Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:3000/dashboard
Accept-Encoding: gzip, deflate, br
Cookie: SESSIONID=a2f51f43e6516e0c
Connection: keep-alive
```

Inspector

| | |
|---|---|
| Request attributes | 2 |
| Request cookies | 1 |
| Request headers | 16 |

Authenticated session token disclosed when accessing non-secure page.

## <u>Disclosure of Tokens in Logs</u>

Most people think tokens only get stolen by hackers spying on the network — but a very common and more dangerous problem is that tokens end up being written into logs (records that systems keep).

Logs are often:

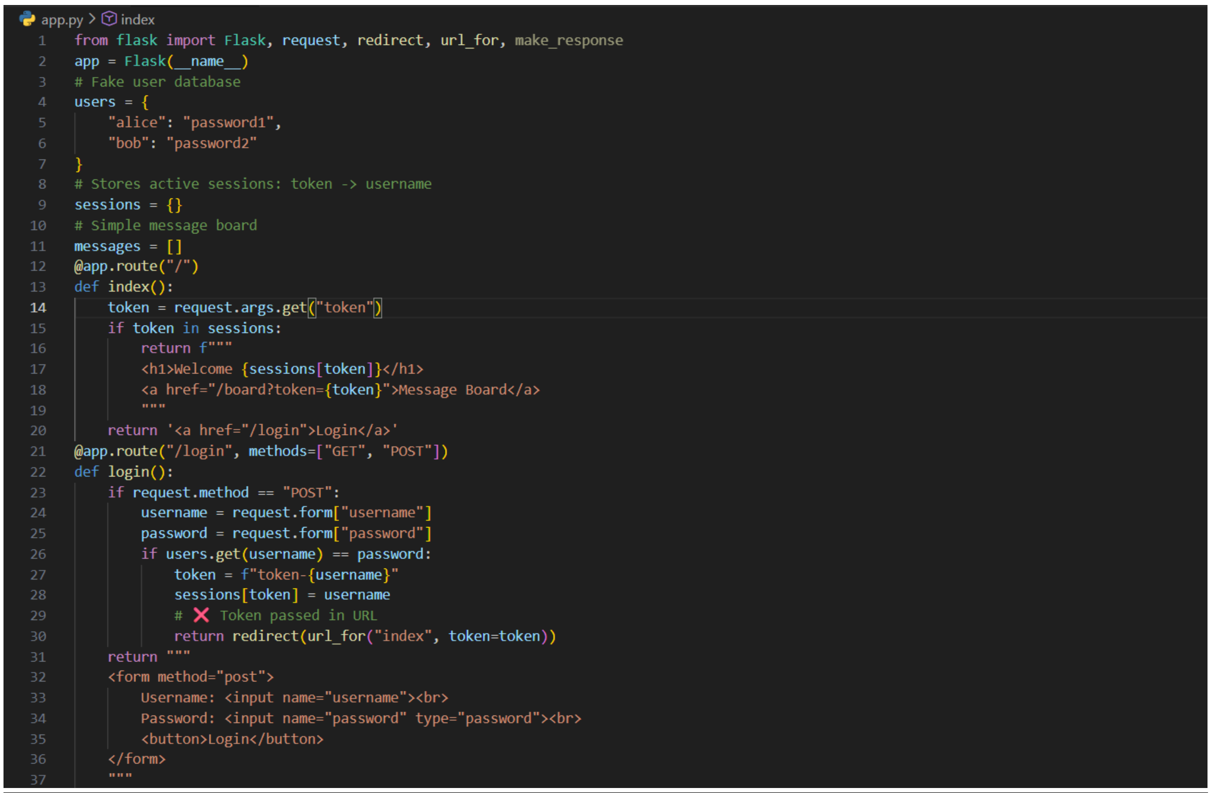
- Viewed by many people (admins, helpdesk staff, support teams)
- Stored for a long time
- Poorly protected

If a session token shows up in a log, anyone who can see that log can steal the session — even if they aren't hacking the network.

Session tokens are secret keys that keep users logged in, but they can be stolen if they appear in system logs. This often happens when admin tools expose tokens or when applications put tokens in URLs, which get recorded in browser, server, and proxy logs. Many people may access these logs, making the risk more serious than network eavesdropping. Even with HTTPS, leaked tokens can let attackers hijack user sessions and take over accounts.

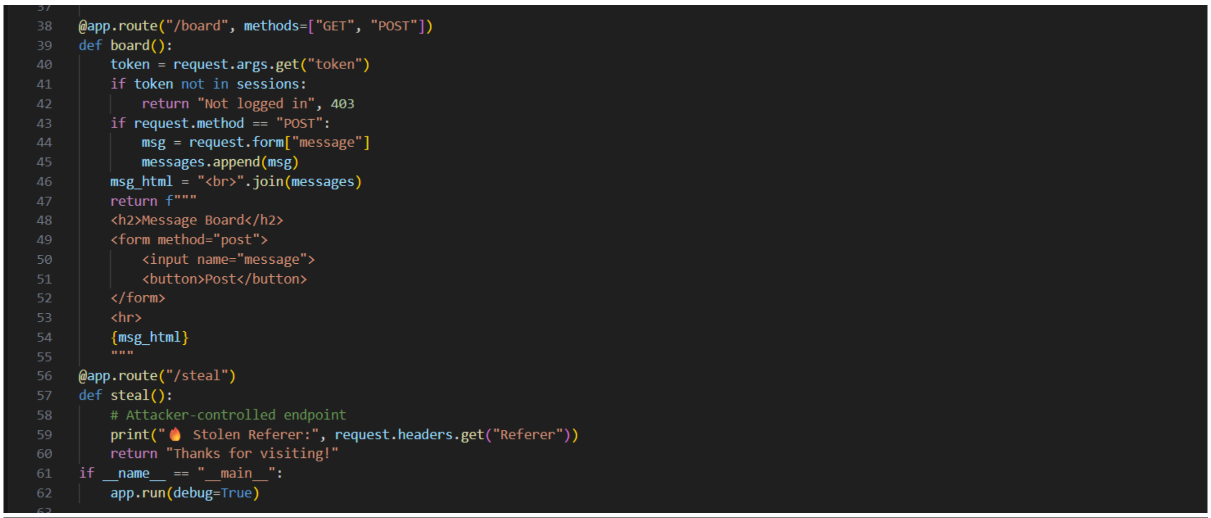
**Disclosure of Tokens in Logs for the local vulnerable website:**

App.py:



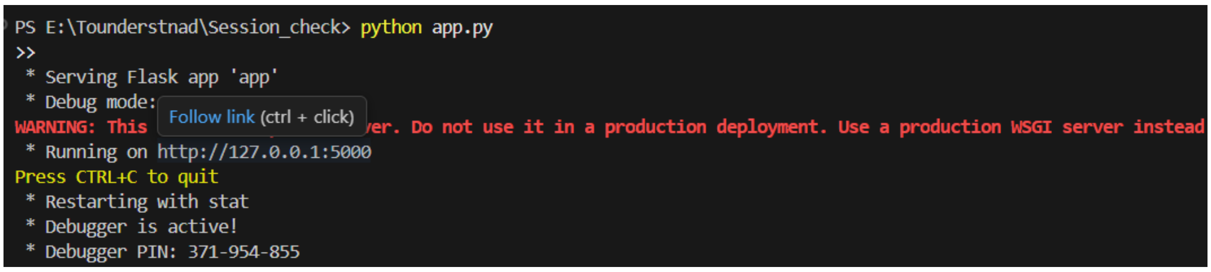
app.py > index

```python
from flask import Flask, request, redirect, url_for, make_response
app = Flask(__name__)
# Fake user database
users = {
    "alice": "password1",
    "bob": "password2"
}
# Stores active sessions: token -> username
sessions = {}
# Simple message board
messages = []
@app.route("/")
def index():
    token = request.args.get("token")
    if token in sessions:
        return f"""
        <h1>Welcome {sessions[token]}</h1>
        <a href="/board?token={token}">Message Board</a>
        """
    return '<a href="/login">Login</a>'
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        if users.get(username) == password:
            token = f"token-{username}"
            sessions[token] = username
            # ❌ Token passed in URL
            return redirect(url_for("index", token=token))
    return """
    <form method="post">
        Username: <input name="username"><br>
        Password: <input name="password" type="password"><br>
        <button>Login</button>
    </form>
    """
```



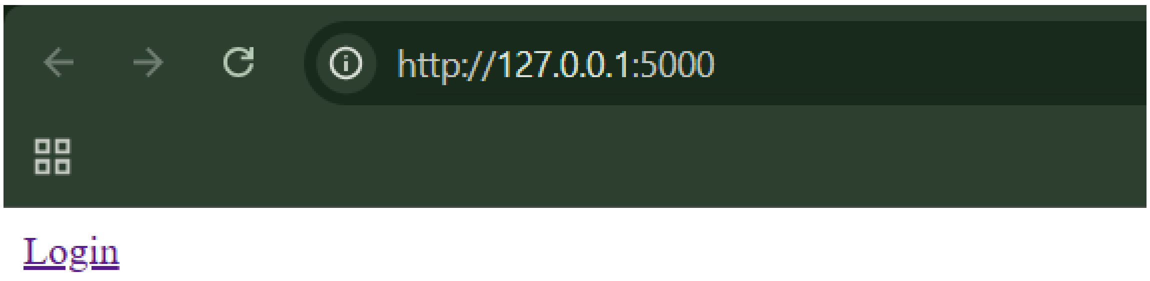
```python
@app.route("/board", methods=["GET", "POST"])
def board():
    token = request.args.get("token")
    if token not in sessions:
        return "Not logged in", 403
    if request.method == "POST":
        msg = request.form["message"]
        messages.append(msg)
    msg_html = "<br>".join(messages)
    return f"""
    <h2>Message Board</h2>
    <form method="post">
        <input name="message">
        <button>Post</button>
    </form>
    <hr>
    {msg_html}
    """
@app.route("/steal")
def steal():
    # Attacker-controlled endpoint
    print("🔥 Stolen Referer:", request.headers.get("Referer"))
    return "Thanks for visiting!"
if __name__ == "__main__":
    app.run(debug=True)
```

Terminal:



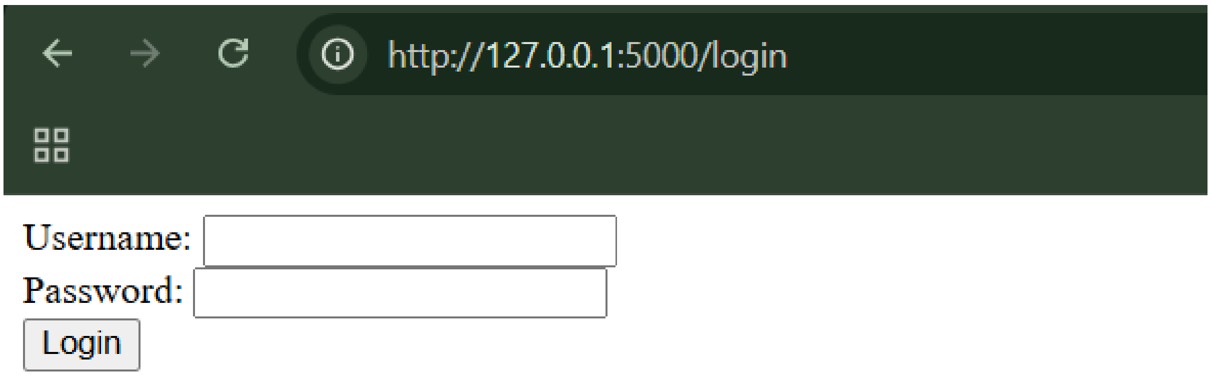
```
PS E:\Tounderstnad\Session_check> python app.py
>>
 * Serving Flask app 'app'
 * Debug mode: Follow link (ctrl + click)
WARNING: This                            er. Do not use it in a production deployment. Use a production WSGI server instead
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 371-954-855
```

Browser:


http://127.0.0.1:5000

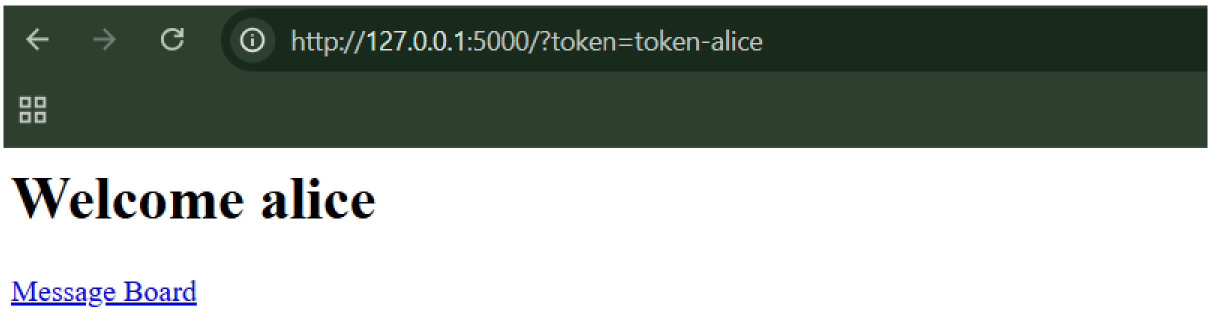
Login

When clicked on the login link, following page will appear:


http://127.0.0.1:5000/login

Username:
Password:
Login

Step1: we will be checking if the tokens are visible or not. For this we will enter the credentials ( alice / password1) and notice the URL.

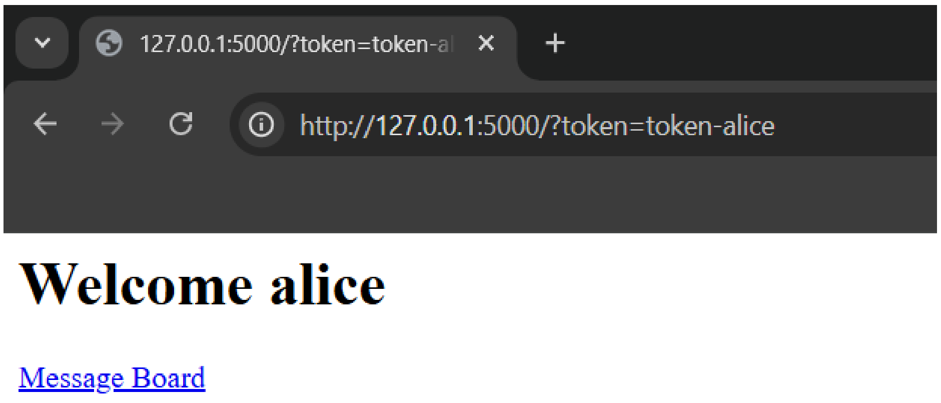

http://127.0.0.1:5000/?token=token-alice

Welcome alice

Message Board

Clearly, we can see the token as token=token-alice.

Step2: Copy the URL http://127.0.0.1:5000/?token=token-alice and paste it in new tab.


127.0.0.1:5000/?token=token-al

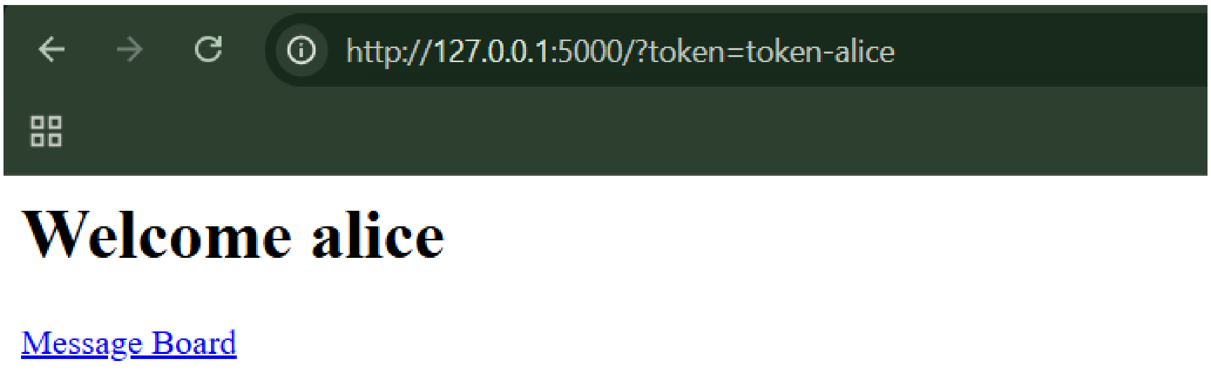
http://127.0.0.1:5000/?token=token-alice

Welcome alice

Message Board

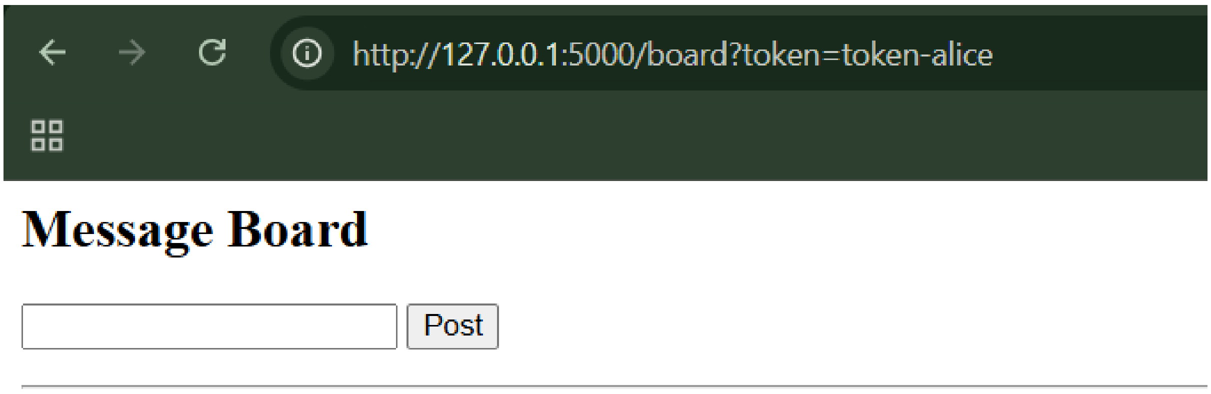
Clearly, we are logged in as Alice.

Step3: Open the message board


http://127.0.0.1:5000/?token=token-alice

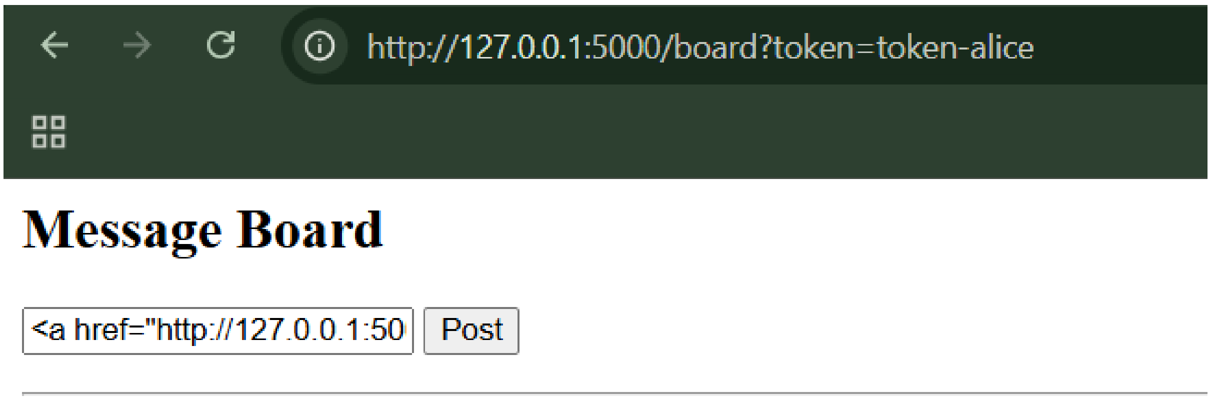
Welcome alice

Message Board


http://127.0.0.1:5000/board?token=token-alice

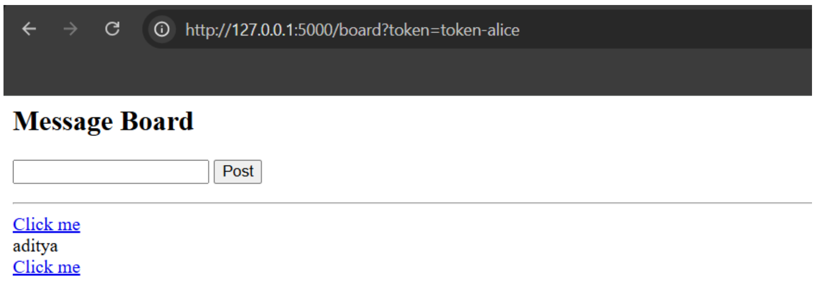
Message Board

In the appeared message box, paste the following <a href="http://127.0.0.1:5000/steal">Click me</a>


http://127.0.0.1:5000/board?token=token-alice

Message Board

<a href="http://127.0.0.1:50


http://127.0.0.1:5000/board?token=token-alice

Message Board

Click me
aditya
Click me

What just happened?

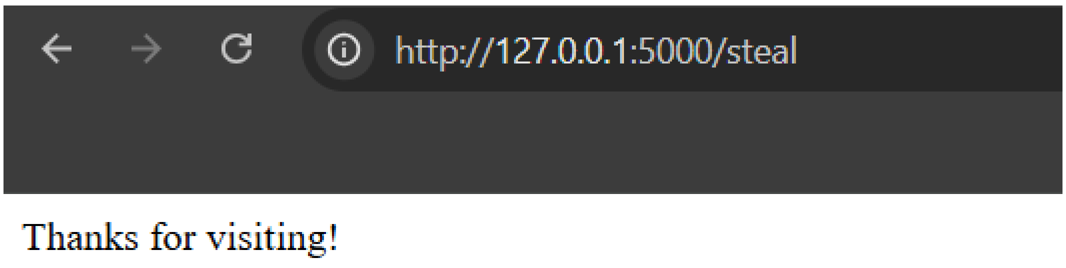
Because the app is insecure:

- It does not escape HTML
- It prints user input directly into the page

So instead of showing text, it renders a real clickable link.

Step4: Login via bob:password2 credentials, and click on the "Click me", which was created by the alice in step3.

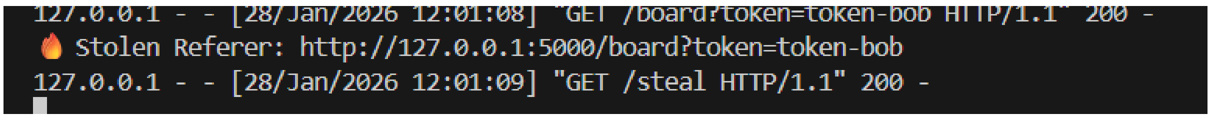
Browser:



```
http://127.0.0.1:5000/steal
```

Thanks for visiting!

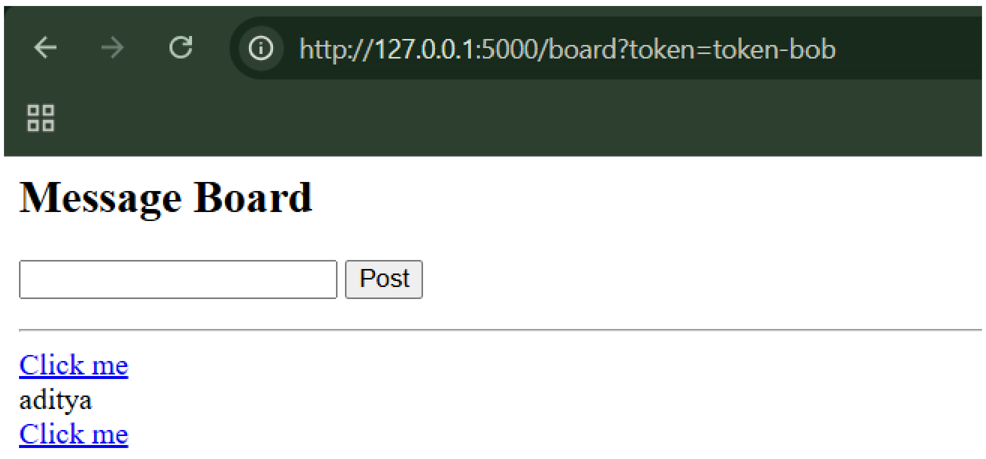
Terminal:



```
127.0.0.1 - - [28/Jan/2026 12:01:08] "GET /board?token=token-bob HTTP/1.1" 200 -
🔥 Stolen Referer: http://127.0.0.1:5000/board?token=token-bob
127.0.0.1 - - [28/Jan/2026 12:01:09] "GET /steal HTTP/1.1" 200 -
```

We as attacker captured Bob's session token successfully.

Step5: Performing session hijacking. Just copy paste the token we received earlier. We managed to use it.



```
http://127.0.0.1:5000/board?token=token-bob
```

Message Board

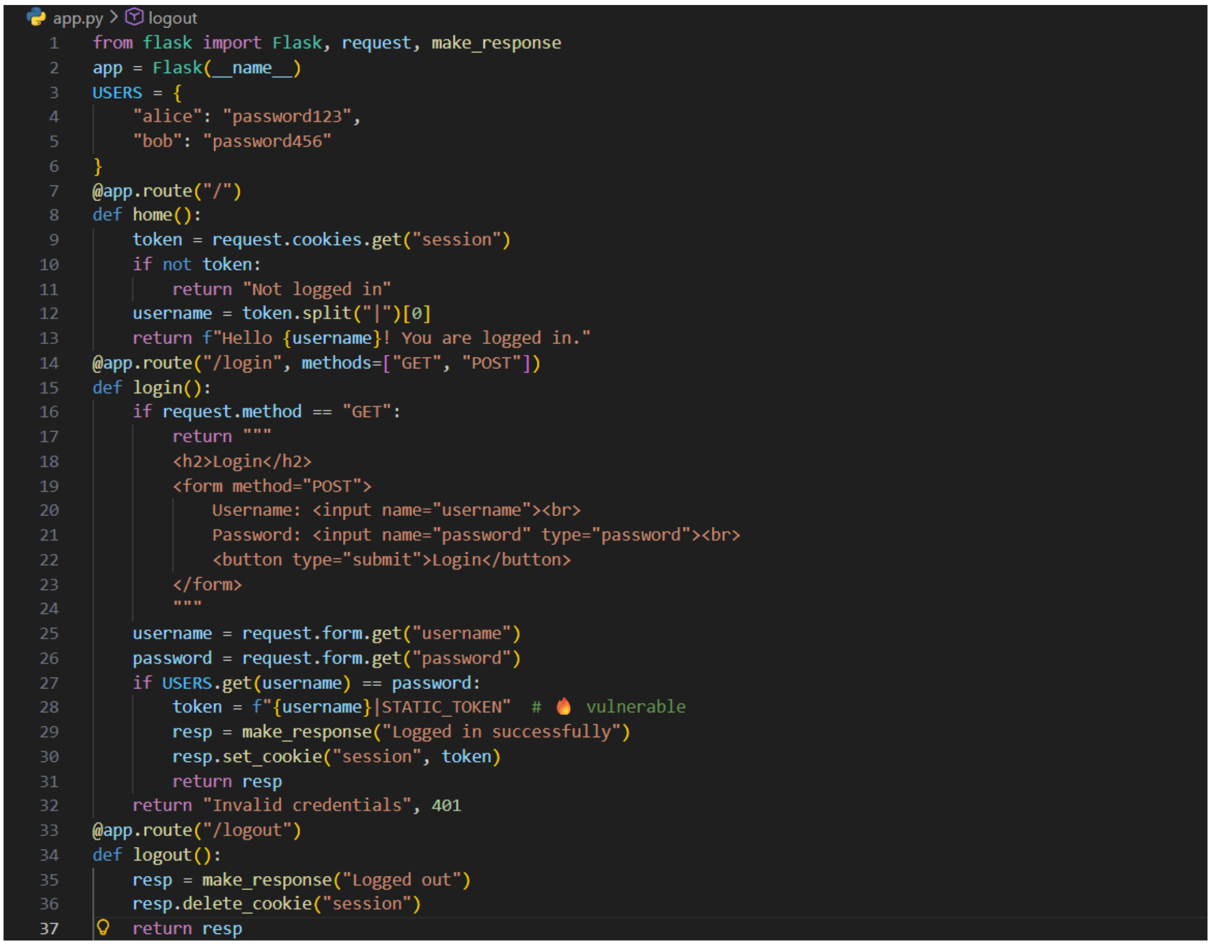
Click me
aditya
Click me

**<u>Vulnerable Mapping of Tokens to Sessions:</u>**

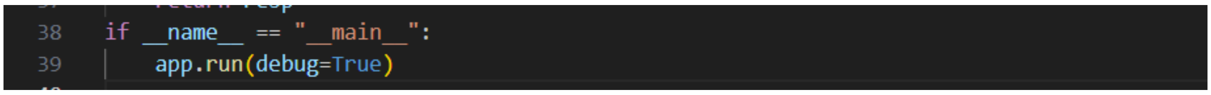
This section explains that session security problems happen when websites poorly link session tokens to users. Allowing multiple active sessions, reusing the same token every login, or trusting user-controlled data in tokens all make account hijacking easier. These flaws let attackers stay logged in unnoticed or impersonate other users. The core issue is failing to tightly bind a session token to one verified user and one active session.

**<u>Vulnerable Mapping of Tokens to Sessions on a local vulnerable webpage:</u>**

App.py:

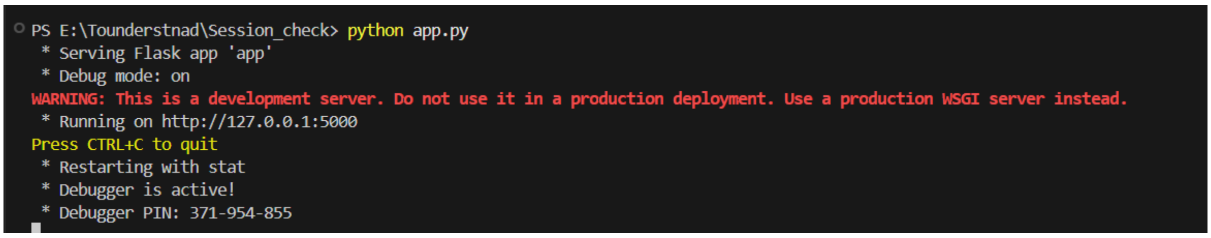


```python
from flask import Flask, request, make_response
app = Flask(__name__)
USERS = {
    "alice": "password123",
    "bob": "password456"
}
@app.route("/")
def home():
    token = request.cookies.get("session")
    if not token:
        return "Not logged in"
    username = token.split("|")[0]
    return f"Hello {username}! You are logged in."
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "GET":
        return """
        <h2>Login</h2>
        <form method="POST">
            Username: <input name="username"><br>
            Password: <input name="password" type="password"><br>
            <button type="submit">Login</button>
        </form>
        """
    username = request.form.get("username")
    password = request.form.get("password")
    if USERS.get(username) == password:
        token = f"{username}|STATIC_TOKEN"  # 🔥 vulnerable
        resp = make_response("Logged in successfully")
        resp.set_cookie("session", token)
        return resp
    return "Invalid credentials", 401
@app.route("/logout")
def logout():
    resp = make_response("Logged out")
    resp.delete_cookie("session")
    return resp
```



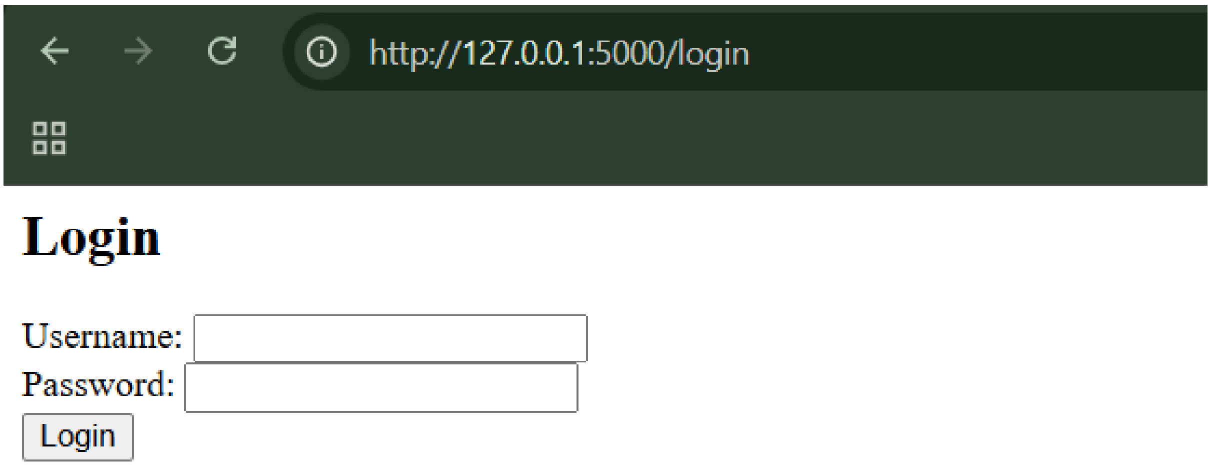
```python
if __name__ == "__main__":
    app.run(debug=True)
```

Terminal:



```
PS E:\Tounderstnad\Session_check> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 371-954-855
```

Browser:

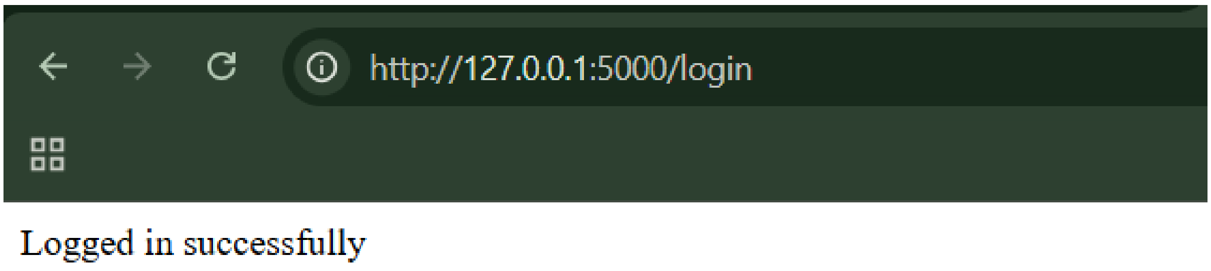


Login

Username:
Password:
Login

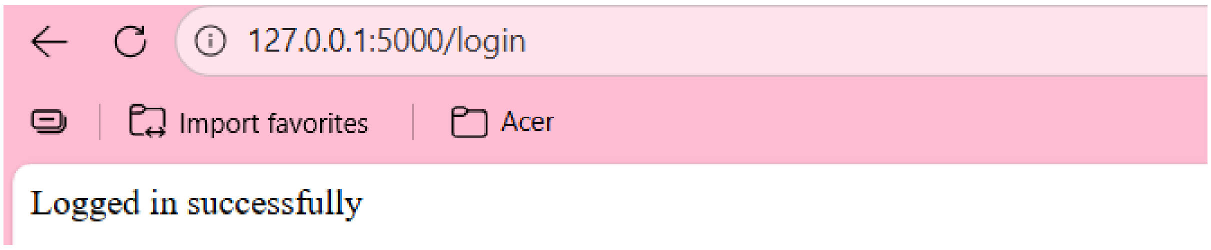
Step1: (Goal: to check if Concurrent sessions are allowed or not). Log in as alice in two different browsers (or incognito).

Chrome:



Logged in successfully

MS Edge:

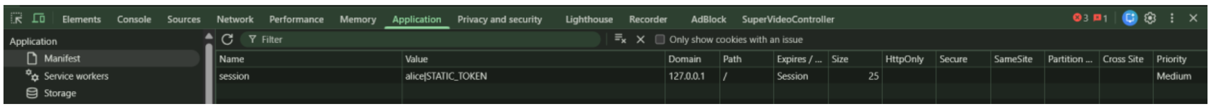


Logged in successfully

Both stay logged in → concurrent sessions allowed

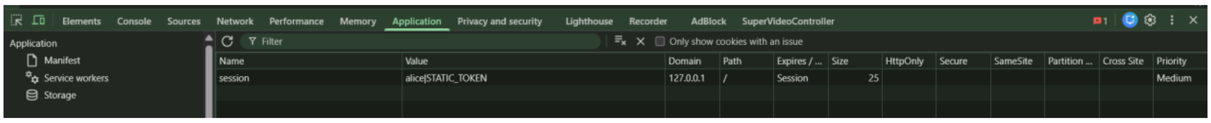
Step2: (Goal: Check for Static session token). Log in → log out → log in again. Open browser dev tools → Cookies.

Cookies at first login:



| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | alice\|STATIC_TOKEN | 127.0.0.1 | / | Session | 25 | | | | | | Medium |

Cookies at second login:



| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | alice\|STATIC_TOKEN | 127.0.0.1 | / | Session | 25 | | | | | | Medium |

We can see the same token every time. This indicates that tokens are constant.

Why this is vulnerable (in simple terms)

- The server trusts whatever username is inside the cookie
- Tokens never expire or change
- No server-side session tracking
- User identity is controlled by the attacker

Remedies:

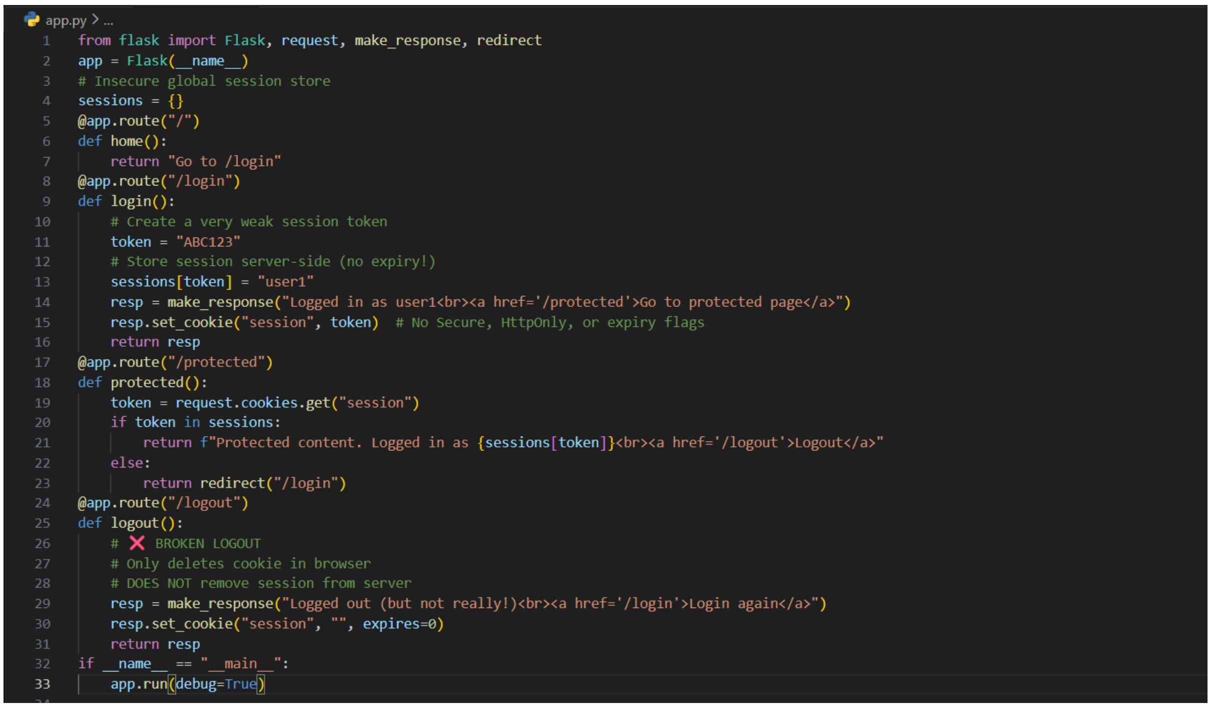
- Use server-side session management: Store session data on the server and use the token only as a random session ID, not as a container for user information.
- Generate a new session token on every login: Always issue a fresh, unpredictable token when a user logs in and invalidate any old ones.
- Prevent concurrent sessions (or monitor them): Either allow only one active session per user or actively detect and alert on multiple simultaneous logins.
- Never trust user-controlled data to identify users: Do not embed usernames, roles, or permissions inside tokens unless they are cryptographically protected and verified.
- Expire and invalidate sessions properly: Enforce session timeouts and ensure tokens are destroyed on logout or after inactivity.

**Vulnerable Session Termination:**

Proper session termination is critical for security because it limits how long a session token can be abused and allows users to safely end their sessions. Many applications fail to do this by allowing sessions to last too long or by implementing weak logout mechanisms that do not invalidate the session on the server. As a result, attackers who obtain session tokens can continue using them even after a user logs out, leading to session hijacking and unauthorized access.

**Observing Vulnerable Session Termination on a local website:**

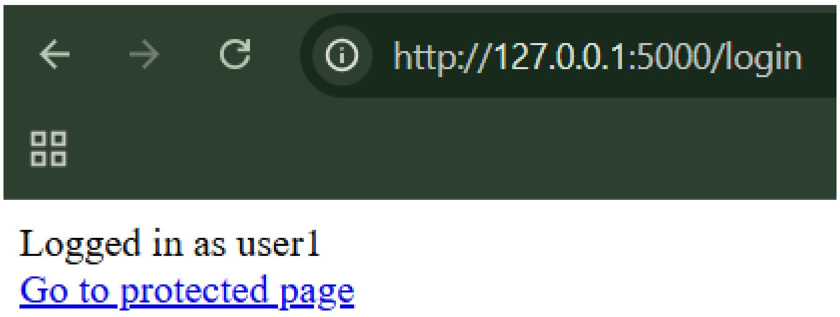
App.py:



app.py > ...

```python
from flask import Flask, request, make_response, redirect
app = Flask(__name__)
# Insecure global session store
sessions = {}
@app.route("/")
def home():
    return "Go to /login"
@app.route("/login")
def login():
    # Create a very weak session token
    token = "ABC123"
    # Store session server-side (no expiry!)
    sessions[token] = "user1"
    resp = make_response("Logged in as user1<br><a href='/protected'>Go to protected page</a>")
    resp.set_cookie("session", token)  # No Secure, HttpOnly, or expiry flags
    return resp
@app.route("/protected")
def protected():
    token = request.cookies.get("session")
    if token in sessions:
        return f"Protected content. Logged in as {sessions[token]}<br><a href='/logout'>Logout</a>"
    else:
        return redirect("/login")
@app.route("/logout")
def logout():
    # ❌ BROKEN LOGOUT
    # Only deletes cookie in browser
    # DOES NOT remove session from server
    resp = make_response("Logged out (but not really!)<br><a href='/login'>Login again</a>")
    resp.set_cookie("session", "", expires=0)
    return resp
if __name__ == "__main__":
    app.run(debug=True)
```

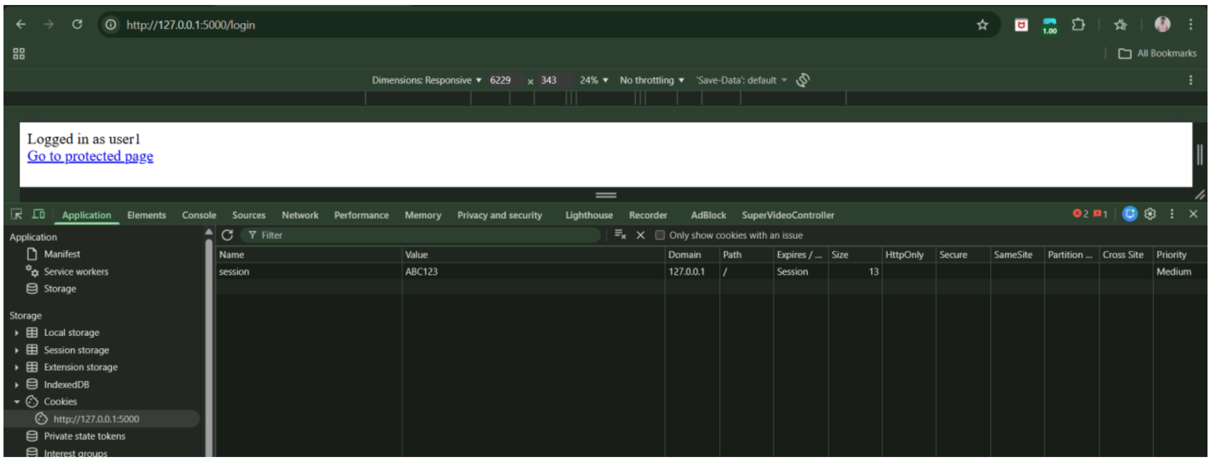
Browser:



http://127.0.0.1:5000/login

Logged in as user1
Go to protected page

Step1: (Goal: To prove Session never expires.) Just login, and see the session cookie value. Observe it again after hours, we can see that session value never expired.



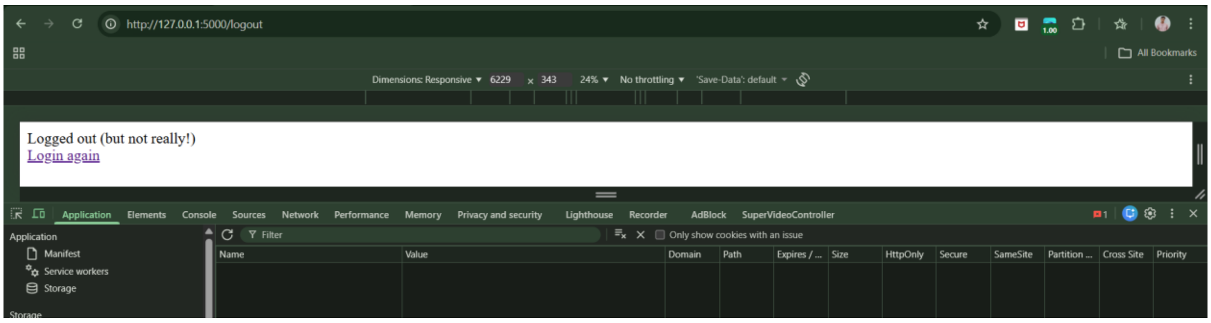
http://127.0.0.1:5000/login

Dimensions: Responsive 6229 x 343 24% No throttling 'Save-Data': default

Logged in as user1
Go to protected page

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | ABC123 | 127.0.0.1 | / | Session | 13 | | | | | | Medium |

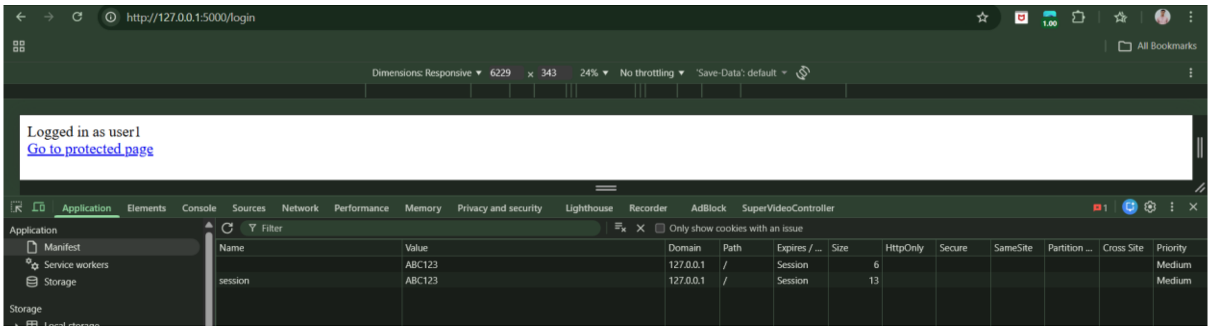
Step2: (Goal: To prove Logout doesn't invalidate session.) Logout (Visit /logout).



```
http://127.0.0.1:5000/logout

Logged out (but not really!)
Login again
```

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|

Manually set the old cookie again, and refresh it again.



```
http://127.0.0.1:5000/login

Logged in as user1
Go to protected page
```

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  | ABC123 | 127.0.0.1 | / | Session | 6 |  |  |  |  |  | Medium |
| session | ABC123 | 127.0.0.1 | / | Session | 13 |  |  |  |  |  | Medium |

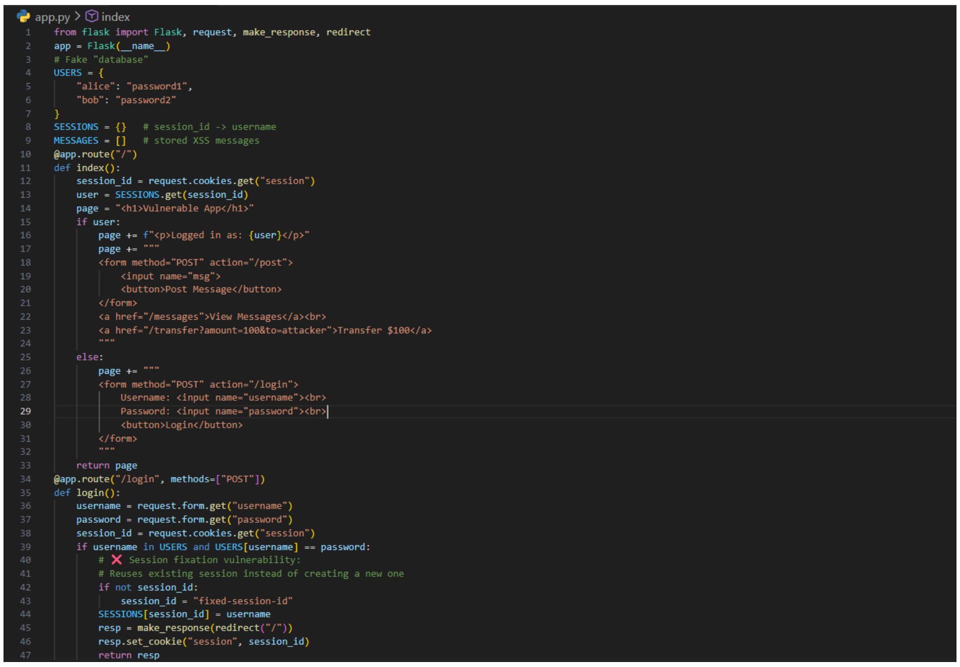
The server never destroys the session, so deleting cookies is meaningless.

## <u>Client Exposure to Token Hijacking:</u>

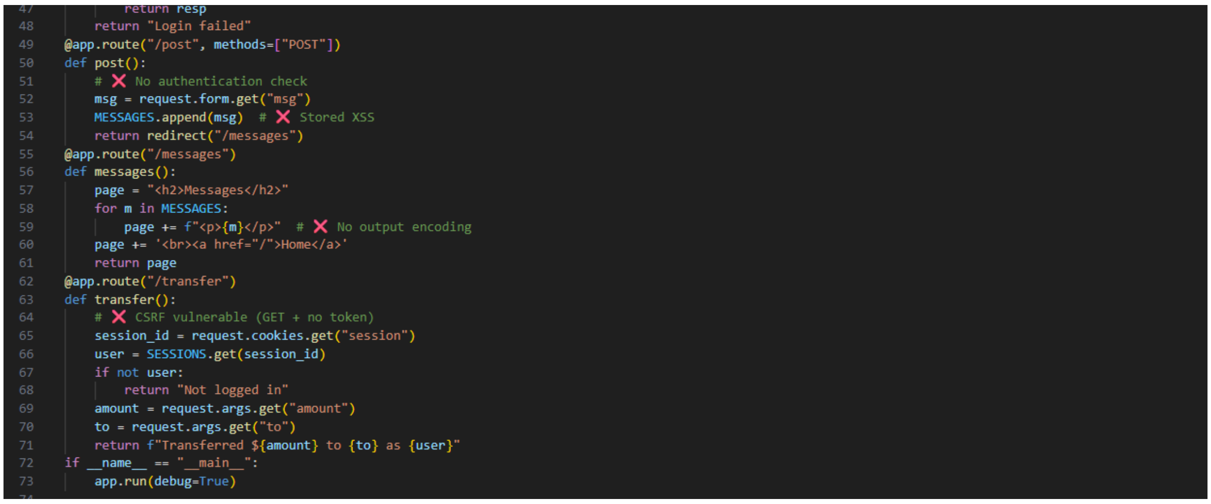
If an attacker can steal, predict, or force the use of a session token, they can take over a user's session — without knowing the password.

## <u>Testing concept of Client Exposure to Token Hijacking on a local website:</u>

App.py:

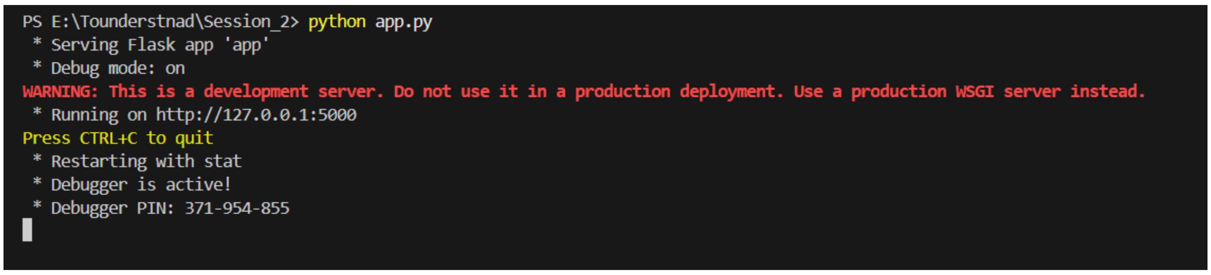


```python
from flask import Flask, request, make_response, redirect
app = Flask(__name__)
# Fake "database"
USERS = {
    "alice": "password1",
    "bob": "password2"
}
SESSIONS = {}   # session_id -> username
MESSAGES = []   # stored XSS messages
@app.route("/")
def index():
    session_id = request.cookies.get("session")
    user = SESSIONS.get(session_id)
    page = "<h1>Vulnerable App</h1>"
    if user:
        page += f"<p>Logged in as: {user}</p>"
        page += """
        <form method="POST" action="/post">
            <input name="msg">
            <button>Post Message</button>
        </form>
        <a href="/messages">View Messages</a><br>
        <a href="/transfer?amount=100&to=attacker">Transfer $100</a>
        """
    else:
        page += """
        <form method="POST" action="/login">
            Username: <input name="username"><br>
            Password: <input name="password"><br>
            <button>Login</button>
        </form>
        """
    return page
@app.route("/login", methods=["POST"])
def login():
    username = request.form.get("username")
    password = request.form.get("password")
    session_id = request.cookies.get("session")
    if username in USERS and USERS[username] == password:
        # ❌ Session fixation vulnerability:
        # Reuses existing session instead of creating a new one
        if not session_id:
            session_id = "fixed-session-id"
        SESSIONS[session_id] = username
        resp = make_response(redirect("/"))
        resp.set_cookie("session", session_id)
        return resp
```



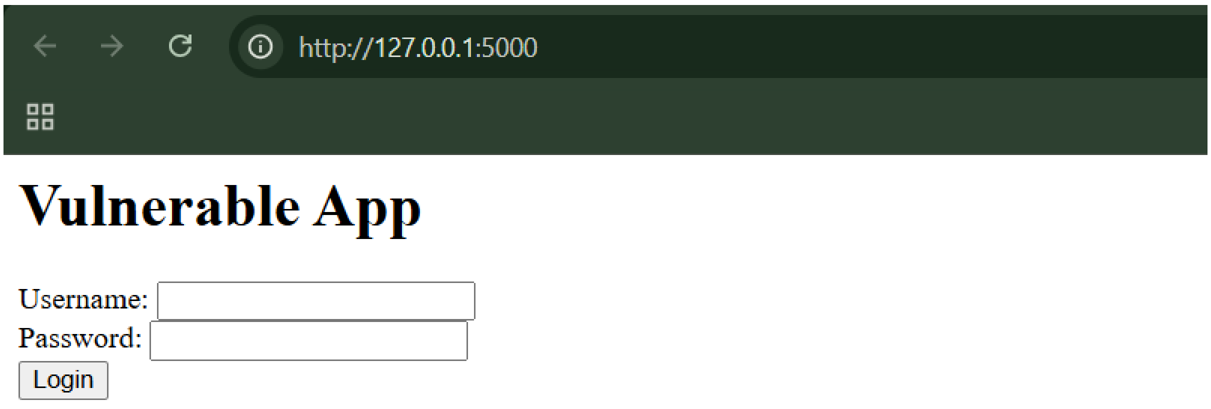
```python
            return resp
        return "Login failed"
@app.route("/post", methods=["POST"])
def post():
    # ❌ No authentication check
    msg = request.form.get("msg")
    MESSAGES.append(msg)  # ❌ Stored XSS
    return redirect("/messages")
@app.route("/messages")
def messages():
    page = "<h2>Messages</h2>"
    for m in MESSAGES:
        page += f"<p>{m}</p>"  # ❌ No output encoding
    page += '<br><a href="/">Home</a>'
    return page
@app.route("/transfer")
def transfer():
    # ❌ CSRF vulnerable (GET + no token)
    session_id = request.cookies.get("session")
    user = SESSIONS.get(session_id)
    if not user:
        return "Not logged in"
    amount = request.args.get("amount")
    to = request.args.get("to")
    return f"Transferred ${amount} to {to} as {user}"
if __name__ == "__main__":
    app.run(debug=True)
```

Terminal:



```
PS E:\Tounderstnad\Session_2> python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 371-954-855
```

Browser:



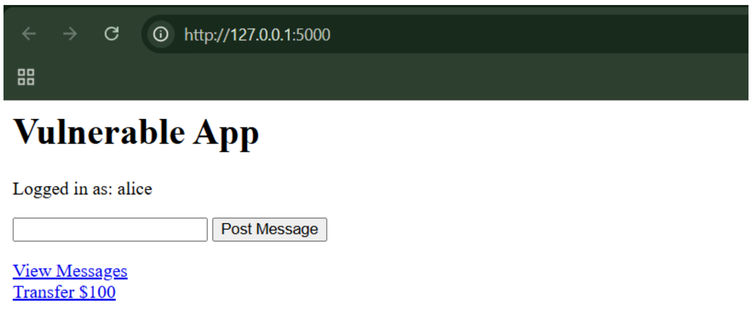
http://127.0.0.1:5000

# Vulnerable App

Username: 
Password: 
Login

Step1: (Goal: Session Fixation testing.) Visit the page, login using the credentials alice:password1.

Following page will occur:



http://127.0.0.1:5000

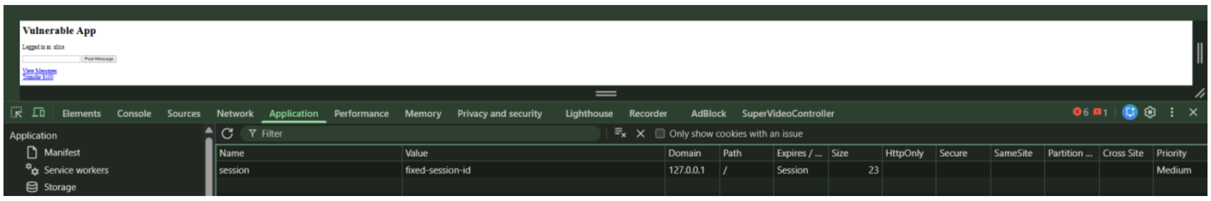
Vulnerable App

Logged in as: alice

Post Message

View Messages
Transfer $100

Look for the session id:


Vulnerable App

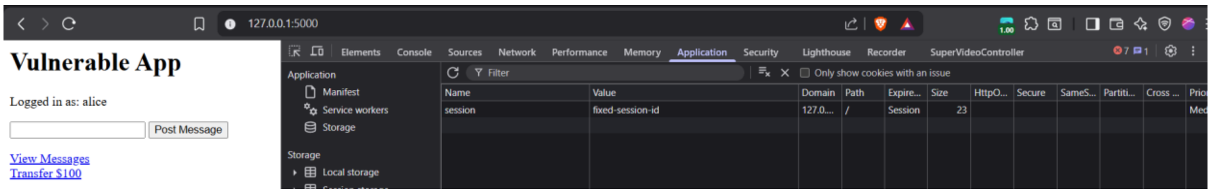
Logged in as: alice

Post Message

View Messages
Transfer $100

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | fixed-session-id | 127.0.0.1 | / | Session | 23 | | | | | | Medium |

Open the same login page in any other browser, and login for the same user, then observe the credentials. We can see that it doesn't change:


127.0.0.1:5000

Vulnerable App

Logged in as: alice

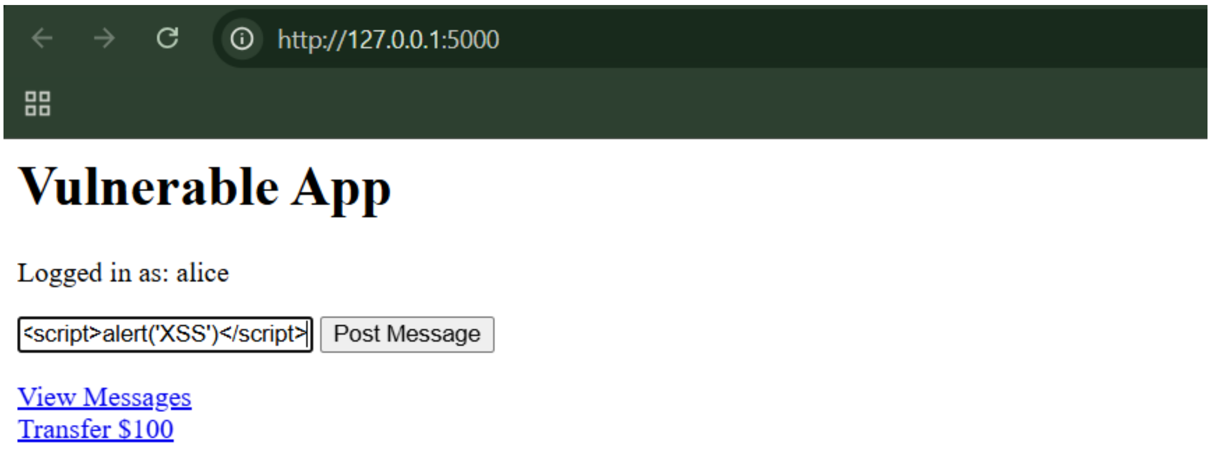
Post Message

View Messages
Transfer $100

| Name | Value | Domain | Path | Expire... | Size | HttpO... | Secure | SameS... | Partiti... | Cross ... | Prio |
|---|---|---|---|---|---|---|---|---|---|---|---|
| session | fixed-session-id | 127.0.... | / | Session | 23 | | | | | | Med |

This is a vulnerability.

Step2: (Goal: Checking for the Stored XSS.) Post this as a message: <script>alert('XSS')</script>


http://127.0.0.1:5000

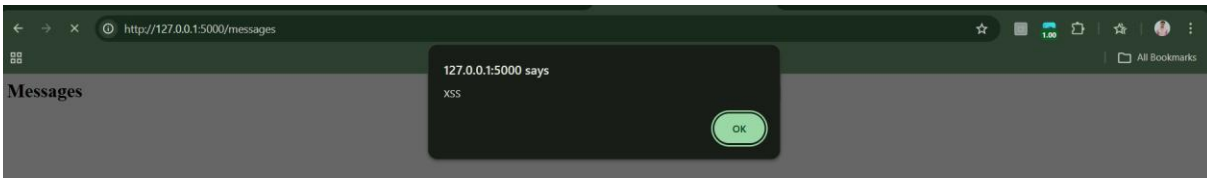
Vulnerable App

Logged in as: alice

<script>alert('XSS')</script> Post Message

View Messages
Transfer $100

Following page will occur:


http://127.0.0.1:5000/messages

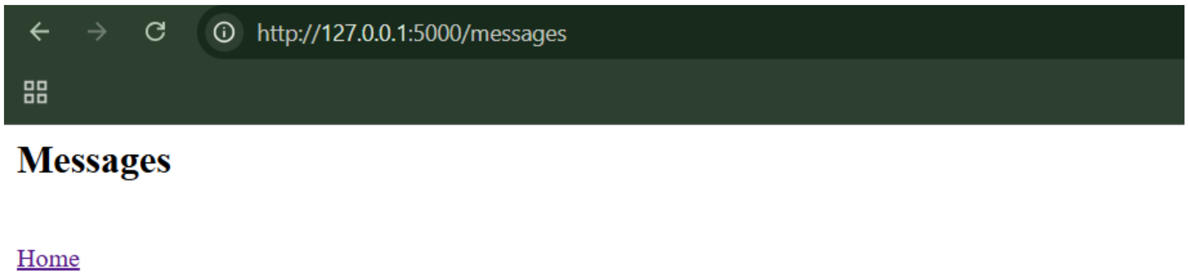
Messages

127.0.0.1:5000 says

XSS

OK

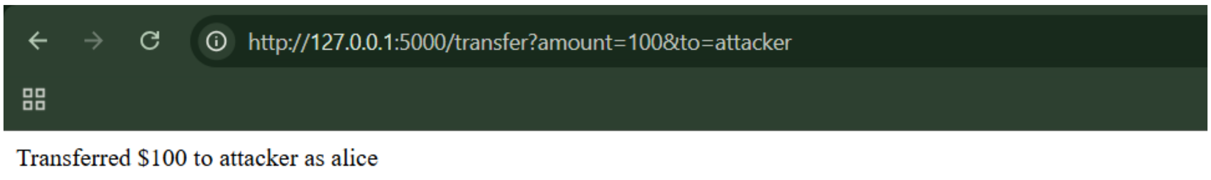
Now open /messages.


http://127.0.0.1:5000/messages

Messages

Home

Step3: (Goal: Checking for CSRF.) While logged in, just opening: /transfer?amount=100&to=attacker



```
http://127.0.0.1:5000/transfer?amount=100&to=attacker
```

```
Transferred $100 to attacker as alice
```

- That means the browser automatically sent your session cookie
- The server trusted it without checking where the request came from
- This is exactly the CSRF vulnerability

**<u>Liberal Cookie Scope:</u>**

Cookies aren't just "sent back to the same server." They're only sent back when the domain and path rules match.

**Cookie Domain Restrictions:**

Cookies should be shared with as few domains as possible — widening their scope makes it much easier for attackers to steal session data.

- By default, a cookie is sent only to the domain that created it and its subdomains.
- A server can widen this by setting a domain attribute, causing the cookie to be sent to multiple subdomains.
- Browsers restrict this to prevent abuse (cookies can't be set for unrelated or top-level domains).
- Making cookie domains too broad is dangerous because cookies (like session IDs) may be sent to less trusted or user-controlled subdomains.
- This can allow attackers to steal session cookies using vulnerabilities like cross-site scripting (XSS).
- The safest approach is to scope cookies as narrowly as possible, especially for sensitive applications, often using a fully qualified domain (e.g., www.example.com).

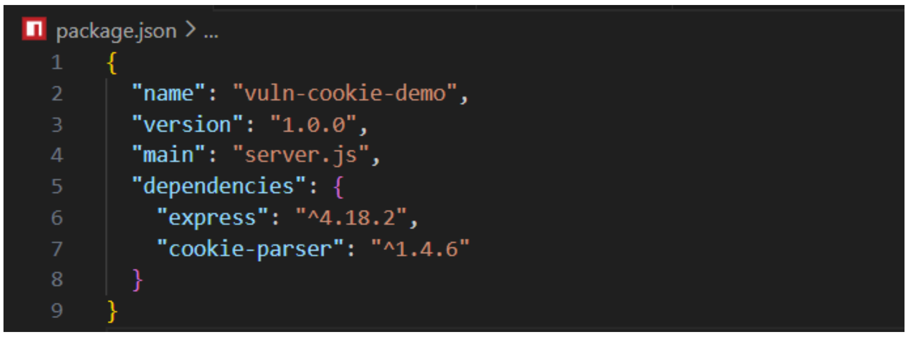
**Cookie Path Restrictions:**

If a cookie is shared with too many URL paths, a weak app on the same site can steal or abuse it — even if the main app is secure.

A cookie's path decides which parts of a website receive it, usually limiting it to one folder and its subfolders for safety. If the path is set too broadly (like /apps/ or /), the cookie gets shared with many apps on the same site. This is risky because a weak or test app can access sensitive cookies from a secure app. Path limits help, but they're not perfect and shouldn't be the only security protection.

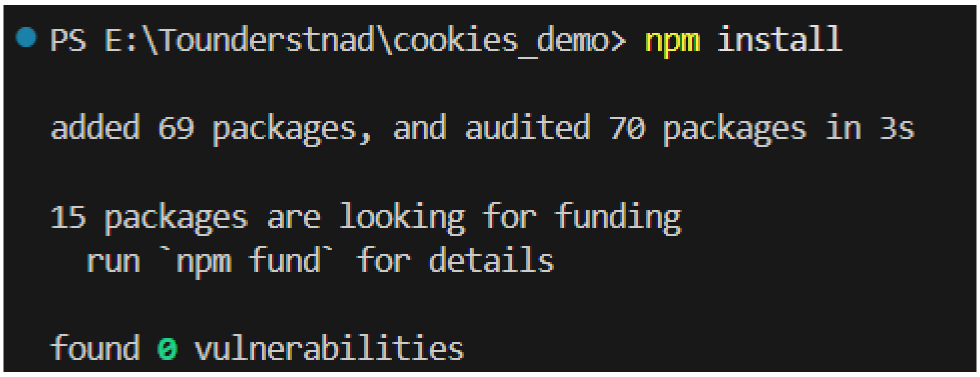
**Observing Cookie Path Restrictions Vulnerability:**

Package.json:



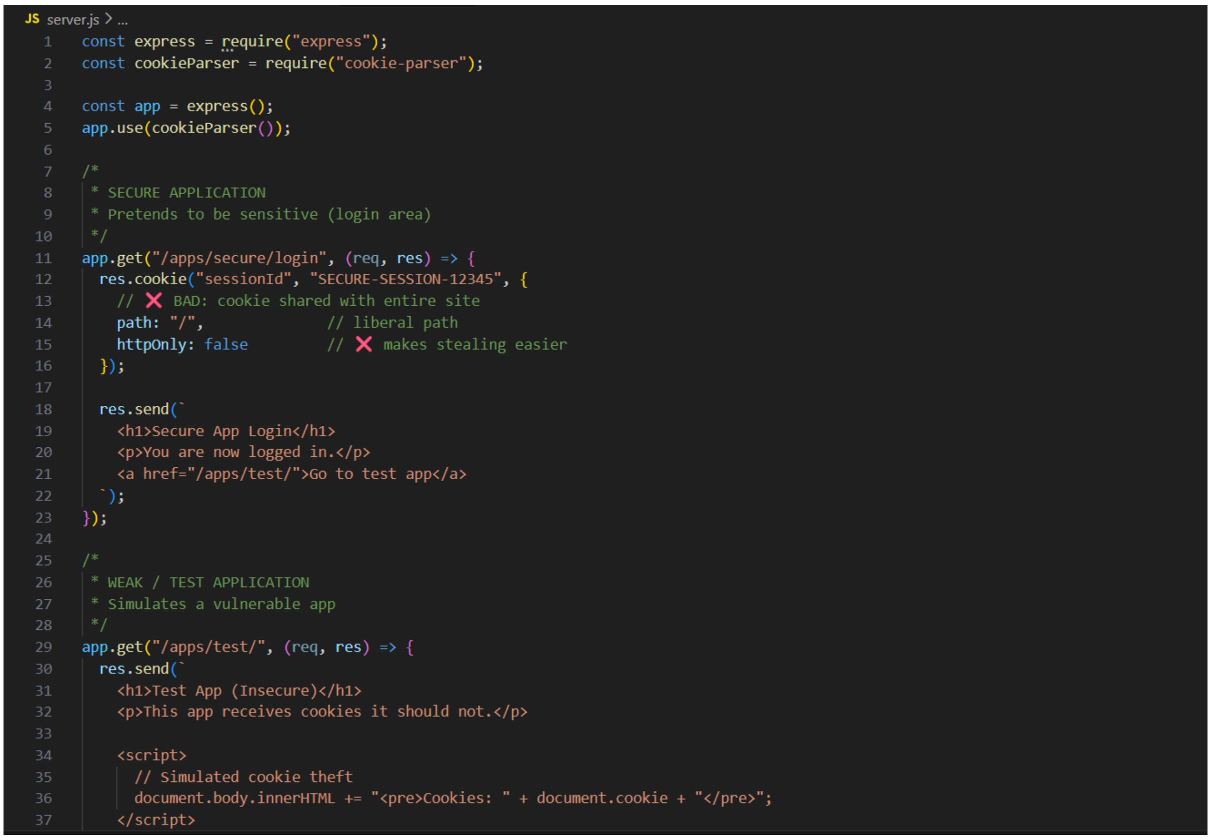
```json
{
  "name": "vuln-cookie-demo",
  "version": "1.0.0",
  "main": "server.js",
  "dependencies": {
    "express": "^4.18.2",
    "cookie-parser": "^1.4.6"
  }
}
```

Terminal:

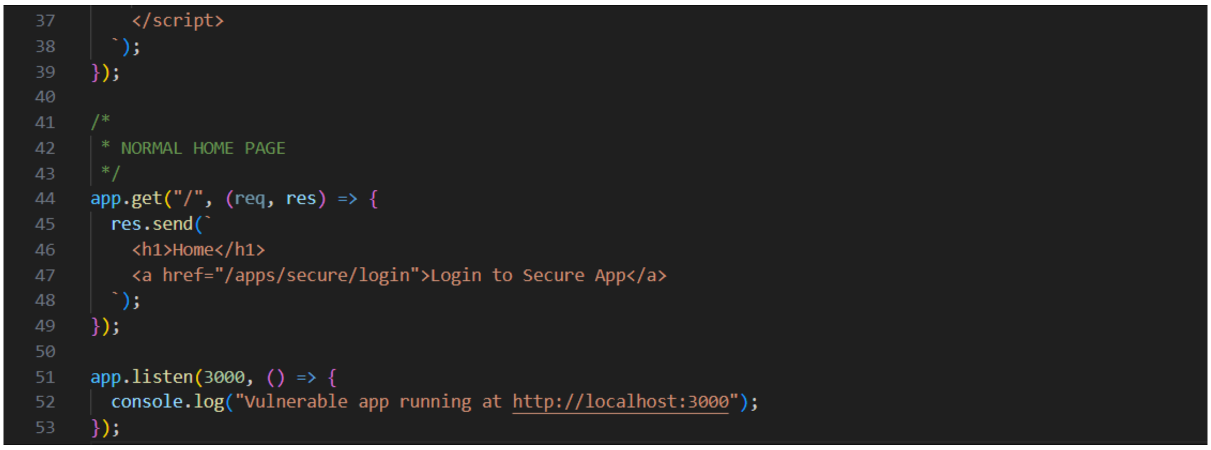


```
PS E:\Tounderstnad\cookies_demo> npm install

added 69 packages, and audited 70 packages in 3s

15 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Server.js:

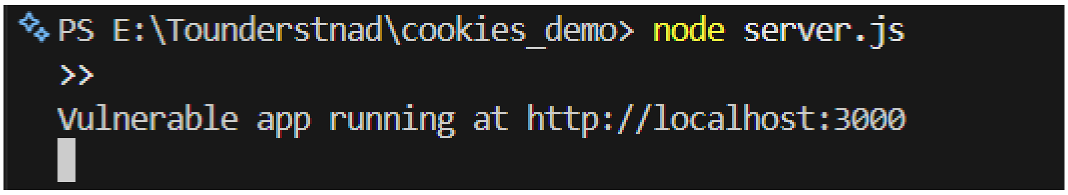


```javascript
const express = require("express");
const cookieParser = require("cookie-parser");

const app = express();
app.use(cookieParser());

/*
 * SECURE APPLICATION
 * Pretends to be sensitive (login area)
 */
app.get("/apps/secure/login", (req, res) => {
  res.cookie("sessionId", "SECURE-SESSION-12345", {
    // ❌ BAD: cookie shared with entire site
    path: "/",            // liberal path
    httpOnly: false       // ❌ makes stealing easier
  });

  res.send(`
    <h1>Secure App Login</h1>
    <p>You are now logged in.</p>
    <a href="/apps/test/">Go to test app</a>
  `);
});

/*
 * WEAK / TEST APPLICATION
 * Simulates a vulnerable app
 */
app.get("/apps/test/", (req, res) => {
  res.send(`
    <h1>Test App (Insecure)</h1>
    <p>This app receives cookies it should not.</p>

    <script>
      // Simulated cookie theft
      document.body.innerHTML += "<pre>Cookies: " + document.cookie + "</pre>";
    </script>
```



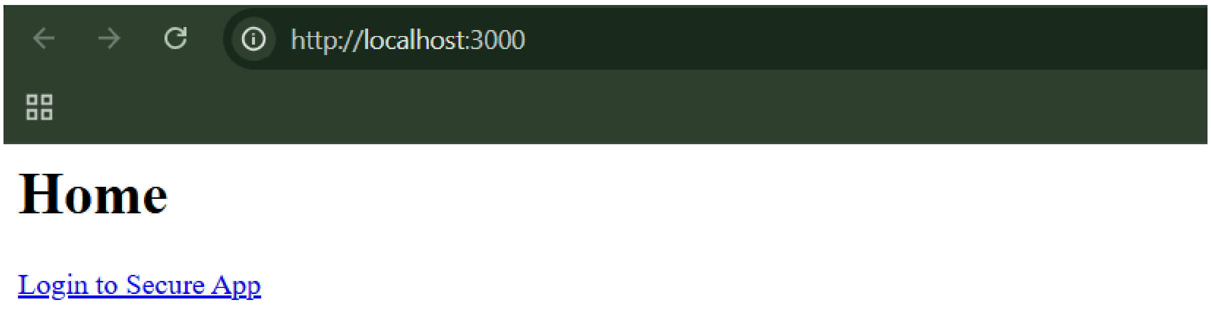
```
    </script>
  `);
});

/*
 * NORMAL HOME PAGE
 */
app.get("/", (req, res) => {
  res.send(`
    <h1>Home</h1>
    <a href="/apps/secure/login">Login to Secure App</a>
  `);
});

app.listen(3000, () => {
  console.log("Vulnerable app running at http://localhost:3000");
});
```

Terminal:



```
PS E:\Tounderstnad\cookies_demo> node server.js
>>
Vulnerable app running at http://localhost:3000
```

Browser:

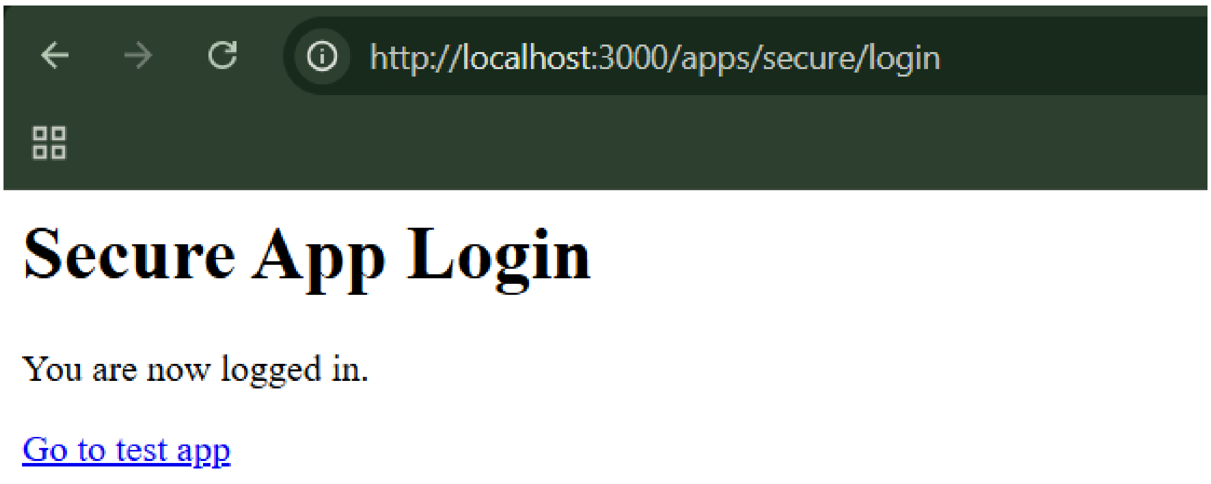


http://localhost:3000

Home

Login to Secure App

Step1: (Goal: to see if it handles the session id carefully or not). Click Login to Secure App. Open DevTools → Application → Cookies



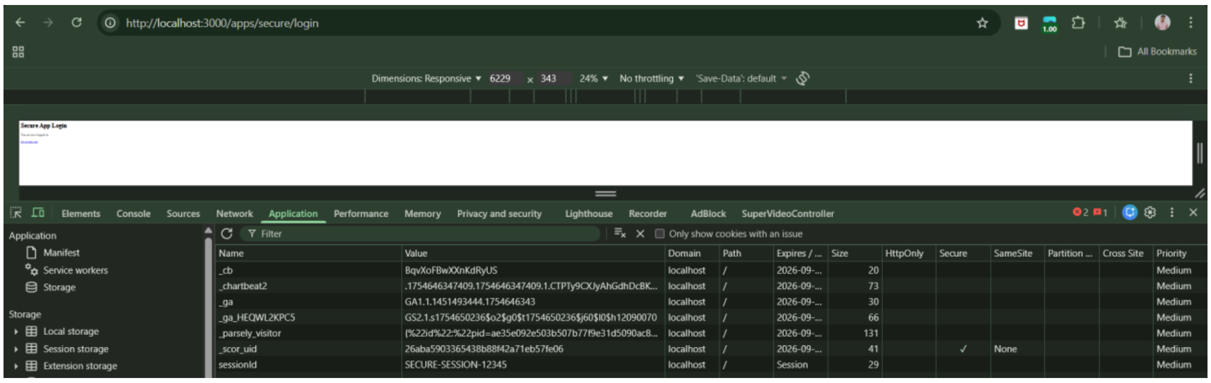
http://localhost:3000/apps/secure/login

Secure App Login

You are now logged in.

Go to test app

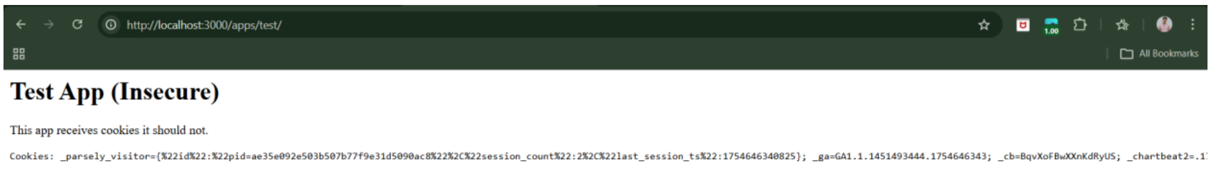
Notice: sessionId = SECURE-SESSION-12345



http://localhost:3000/apps/secure/login

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| _cb | BqvXoFBwXXnKdRyUS | localhost | / | 2026-09-... | 20 | | | | | | Medium |
| _chartbeat2 | .1754646347409.1754646347409.1.CTPTy9CXJyAhGdhDcBK... | localhost | / | 2026-09-... | 73 | | | | | | Medium |
| _ga | GA1.1.1451493444.1754646343 | localhost | / | 2026-09-... | 30 | | | | | | Medium |
| _ga_HEQWL2KPC5 | GS2.1.s1754650236$o2$g0$t1754650236$j60$l0$h12090070 | localhost | / | 2026-09-... | 66 | | | | | | Medium |
| _parsely_visitor | {%22id%22:%22pid=ae35e092e503b507b77f9e31d5090ac8... | localhost | / | 2026-09-... | 131 | | | | | | Medium |
| _scor_uid | 26aba5903365438b88f42a71eb57fe06 | localhost | / | 2026-09-... | 41 | | ✓ | None | | | Medium |
| sessionId | SECURE-SESSION-12345 | localhost | / | Session | 29 | | | | | | Medium |

Now, click on "Go to test app", following screen will appear:



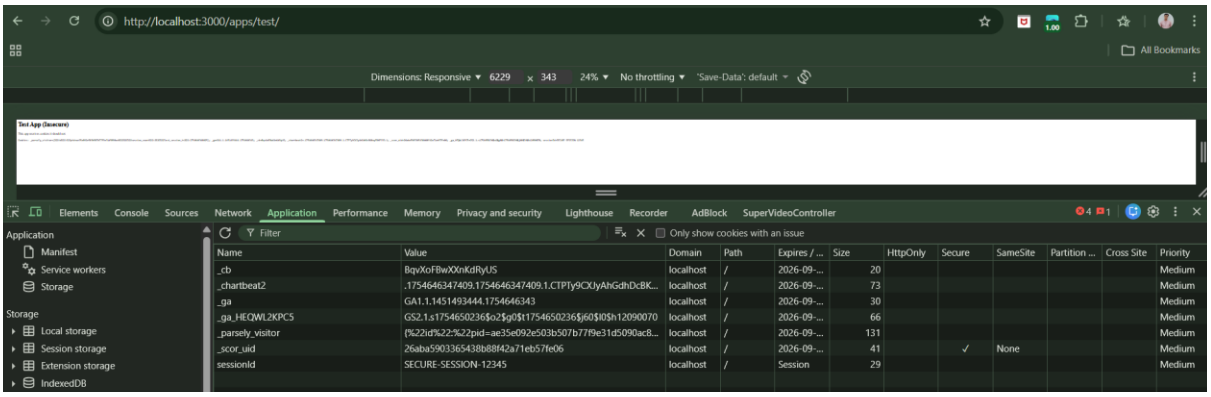
http://localhost:3000/apps/test/

Test App (Insecure)

This app receives cookies it should not.

```
Cookies: _parsely_visitor={%22id%22:%22pid=ae35e092e503b507b77f9e31d5090ac8%22%2C%22session_count%22:2%2C%22last_session_ts%22:1754646340825}; _ga=GA1.1.1451493444.1754646343; _cb=BqvXoFBwXXnKdRyUS; _chartbeat2=.1
```

Again, see the cookie: sessionId=SECURE-SESSION-12345



http://localhost:3000/apps/test/

| Name | Value | Domain | Path | Expires / ... | Size | HttpOnly | Secure | SameSite | Partition ... | Cross Site | Priority |
|---|---|---|---|---|---|---|---|---|---|---|---|
| _cb | BqvXoFBwXXnKdRyUS | localhost | / | 2026-09-... | 20 | | | | | | Medium |
| _chartbeat2 | .1754646347409.1754646347409.1.CTPTy9CXJyAhGdhDcBK... | localhost | / | 2026-09-... | 73 | | | | | | Medium |
| _ga | GA1.1.1451493444.1754646343 | localhost | / | 2026-09-... | 30 | | | | | | Medium |
| _ga_HEQWL2KPC5 | GS2.1.s1754650236$o2$g0$t1754650236$j60$l0$h12090070 | localhost | / | 2026-09-... | 66 | | | | | | Medium |
| _parsely_visitor | {%22id%22:%22pid=ae35e092e503b507b77f9e31d5090ac8... | localhost | / | 2026-09-... | 131 | | | | | | Medium |
| _scor_uid | 26aba5903365438b88f42a71eb57fe06 | localhost | / | 2026-09-... | 41 | | ✓ | None | | | Medium |
| sessionId | SECURE-SESSION-12345 | localhost | / | Session | 29 | | | | | | Medium |

Clearly, the test app stole the secure session.

## Securing Session Management:

Web applications use session tokens to identify users after login, making them a common target for attackers. Security issues mainly arise from how tokens are created and how they are protected over time. Tokens must be random and unpredictable, and also kept safe from theft, leakage, or reuse throughout their lifetime. In short, apps need to create strong session tokens and protect them from start to finish.

### Generate Strong tokens:

- Don't put meaning inside them
- Make tokens long and random
- Use cryptographically secure randomness
- Mix in request data and a server-only secret

**Protect Tokens throughout Their Lifecycle:**

Web apps must protect session tokens at every stage of their lifecycle, because any leak lets attackers hijack user sessions. Tokens should only be sent over HTTPS, never placed in URLs, and must be properly destroyed on logout or inactivity, with old sessions invalidated on new logins.

Strong defenses include limiting cookie scope, locking down admin tools, rejecting unknown tokens, regenerating sessions after login, and preventing XSS, CSRF, and session fixation attacks. For sensitive actions, apps should require re-authentication or extra confirmation and avoid exposing or reflecting sensitive data at any point.

**Per page tokens:**

Per-page tokens are constantly changing mini-tokens that add tight control over every request, quickly stopping hijacking, fixation, and out-of-sequence attacks—even if a session token is stolen.

Per-page tokens are short-lived security codes that change every time you load a page. Each request must include the latest code, or the session is ended. This makes stolen sessions and fake requests much harder to use, but can break back buttons and multiple tabs.

**Log, Monitor, and Alert:**

A secure web app doesn't just block attacks; it watches for them and reacts. It logs invalid or fake tokens (many failures = likely attack), keeps an eye on brute-force attempts, and may temporarily block abusive traffic. Even when attacks can't be stopped right away, logs and alerts let security teams investigate and respond.

Users should also be warned about suspicious activity so they can act fast.

**Reactive Session Termination:**

Ending a session immediately when suspicious activity occurs is a strong defensive tactic that slows attackers, but it doesn't replace proper fixes and must be test-friendly.

--The End--