Chapter 6

Bypassing Client-Side Controls:

Authentication appears simple but is a critical and often weak point in web application security. If compromised, attackers can gain full access to an application and its data, rendering other security measures ineffective. Despite its importance, secure authentication is difficult to implement and frequently flawed, ranging from basic weaknesses like password guessing to subtle issues in complex login processes.

Design Flaws in Authentication Mechanisms:

Some major design flaws in authentication mechanisms are:

- Bad Passwords
- Brute-Forcible Login
- Verbose Failure Messages
- Vulnerable Transmission of Credentials
- Password Change Functionality
- Forgotten Password Functionality
- “Remember Me” Functionality
- User Impersonation Functionality
- Incomplete Validation of Credentials
- Non-Unique Usernames
- Predictable Usernames
- Predictable Initial Passwords
- Insecure Distribution of Credentials

Bad Passwords:

Many web applications allow weak passwords, such as short, common, or default passwords. Since users often choose easy passwords, attackers can easily guess them and gain unauthorized access.

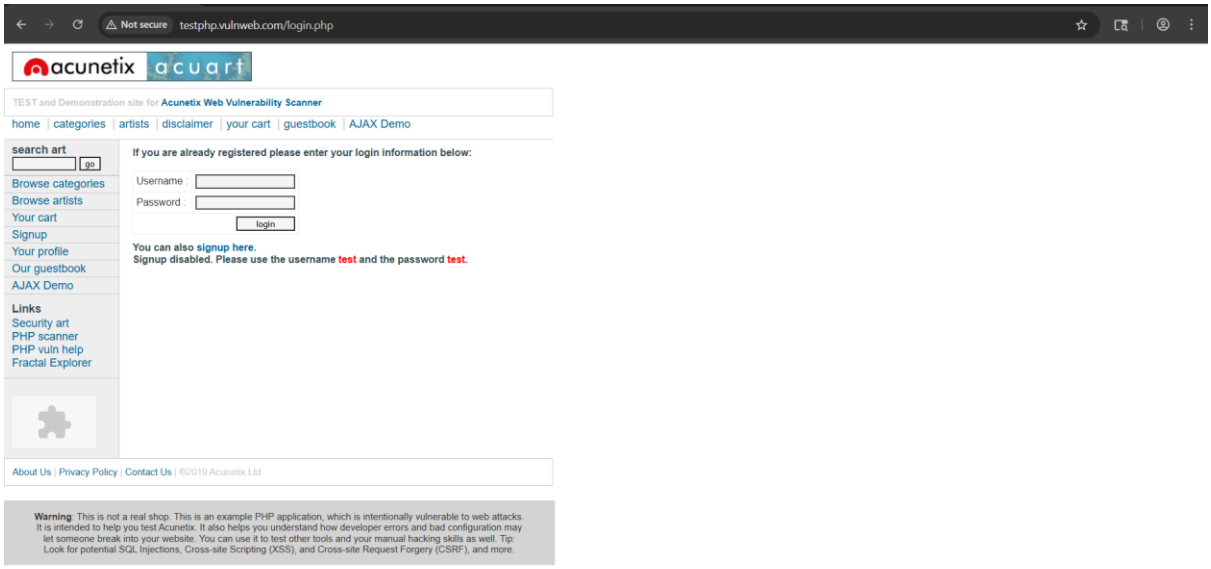
Brute-Forcible login:

If a login system allows unlimited attempts, attackers can repeatedly try common usernames and passwords until they succeed. Using automated tools, thousands of guesses can be made per minute, making even strong passwords vulnerable. Simple client-side protections can be easily bypassed and do not effectively stop these attacks.

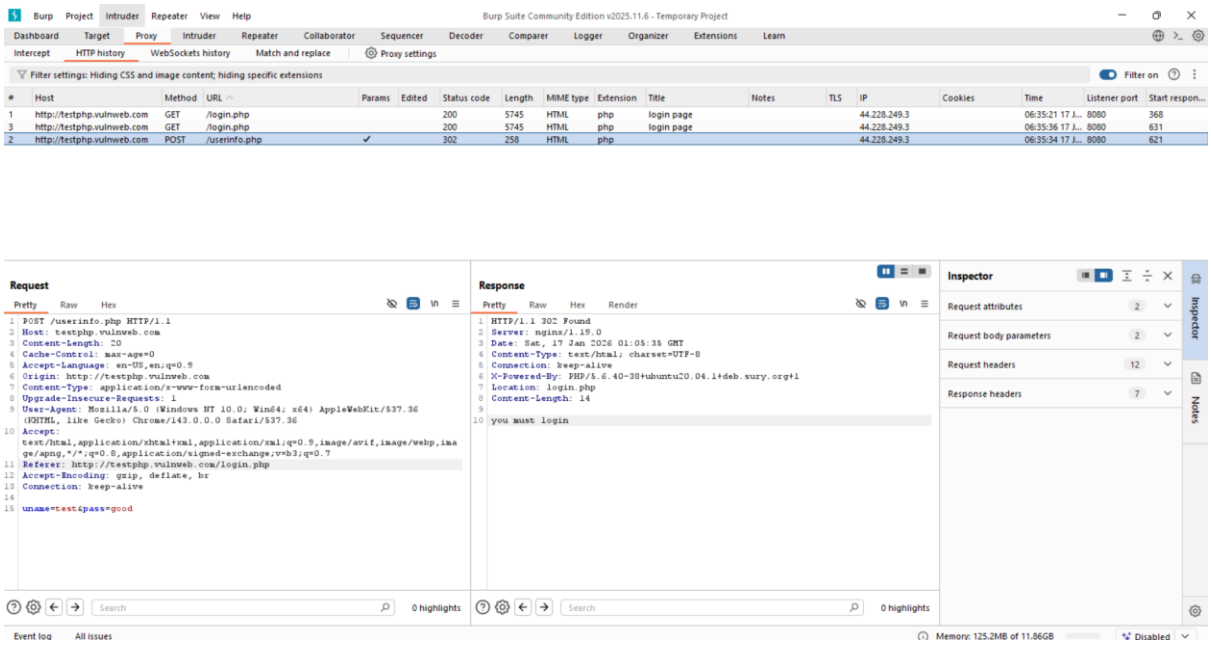
Observing Brute-Forcible login attempts:

Step1: Monitor several bad login attempts. Open the Burp Suite, and its inbuilt browser, visit the login page and try to submit any random set of id and password manually.

Browser:



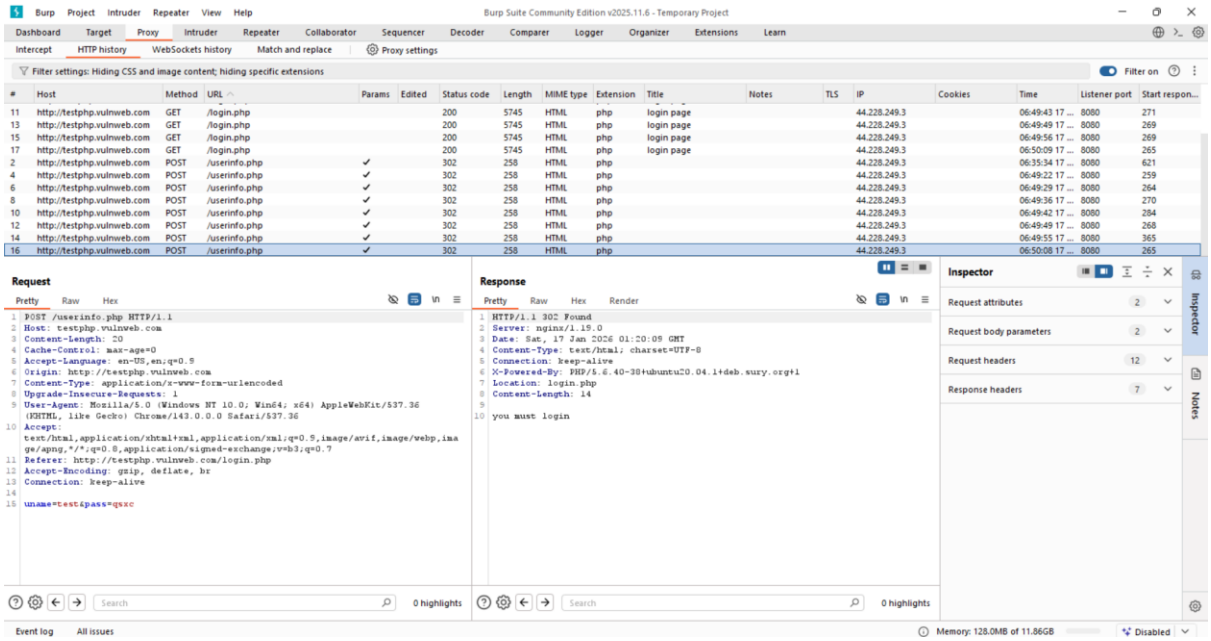
Burp:



Multiple failed login attempts were manually submitted using Burp Suite. For each invalid attempt, the server responded with an HTTP 302 redirect to login.php and a generic message stating “you must login.” The error message did not disclose whether the username or password was incorrect, indicating that the application does not leak authentication details through verbose error messages.

Step2: Approximately ten consecutive failed login attempts were submitted using invalid credentials.

Burp:



The application did not return any warning or account lockout message. A subsequent login attempt using valid credentials succeeded, indicating that no account lockout policy is implemented.

Verbose Failure Messages:

Verbose login failure messages weaken security by revealing too much information when a login attempt fails. If a system tells users whether a username or password is incorrect, attackers can use this to identify valid usernames through automated testing. Even subtle differences in error messages, hidden HTML, or response times can leak this information. Once attackers know valid usernames, it becomes much easier to carry out further attacks such as password guessing or social engineering.

Observing Verbose Failure messages on a local website:

App.py:

app.py > login

```python
from flask import Flask, request, render_template, redirect
from time import time
app = Flask(__name__)
# Fake user database
users = {
    "alice": {"password": "alice123", "fails": 0, "locked": False},
    "bob": {"password": "bob123", "fails": 0, "locked": False}
}
LOCKOUT_THRESHOLD = 3
@app.route("/", methods=["GET"])
def index():
    return render_template("login.html")
@app.route("/login", methods=["POST"])
def login():
    username = request.form.get("username", "")
    password = request.form.get("password", "")
    # Case 1: username does NOT exist
    if username not in users:
        return render_template(
            "login.html",
            error="User does not exist"
        ), 404   # <- Different status code (leak)
    user = users[username]
    # Account lockout check
    if user["locked"]:
        return render_template(
            "login.html",
            error="Account locked"
        ), 403
    # Case 2: wrong password
    if password != user["password"]:
        user["fails"] += 1
        if user["fails"] >= LOCKOUT_THRESHOLD:
            user["locked"] = True

        return render_template(
            "login.html",
            error="Incorrect password"
        ), 401   # <- Different message & code
    # Successful login
    user["fails"] = 0
    return redirect("/dashboard")
@app.route("/dashboard")
def dashboard():
    return "Welcome! Login successful."
if __name__ == "__main__":
    app.run(debug=True)
```

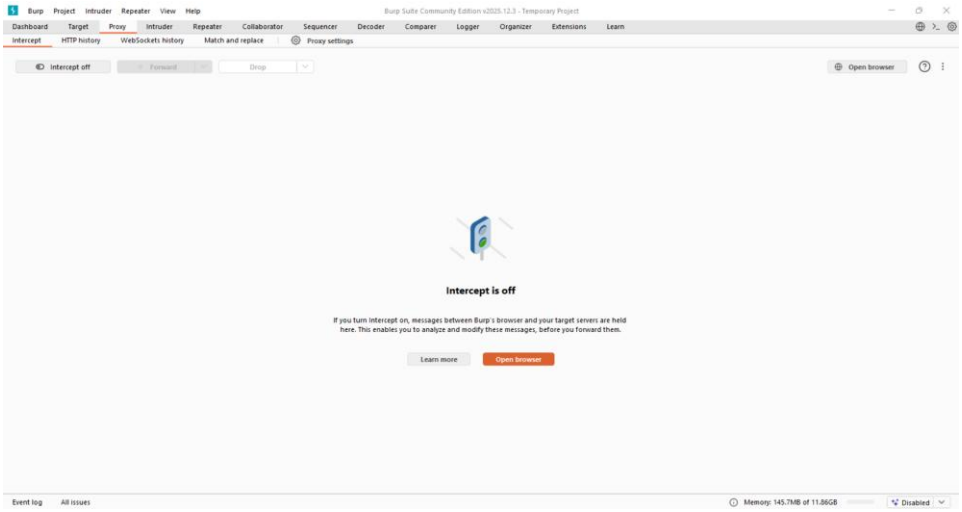
Templates/login.html:

templates > login.html > ...

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

    {% if error %}
        <p style="color: red;">{{ error }}</p>
    {% endif %}

    <form method="POST" action="/login">
        <label>Username:</label><br>
        <input type="text" name="username"><br><br>

        <label>Password:</label><br>
        <input type="password" name="password"><br><br>

        <button type="submit">Login</button>
    </form>
</body>
</html>
```

Step1: Open burp, go to http://127.0.0.1:5000. And in burp Turn Proxy → Intercept → ON (in step2)

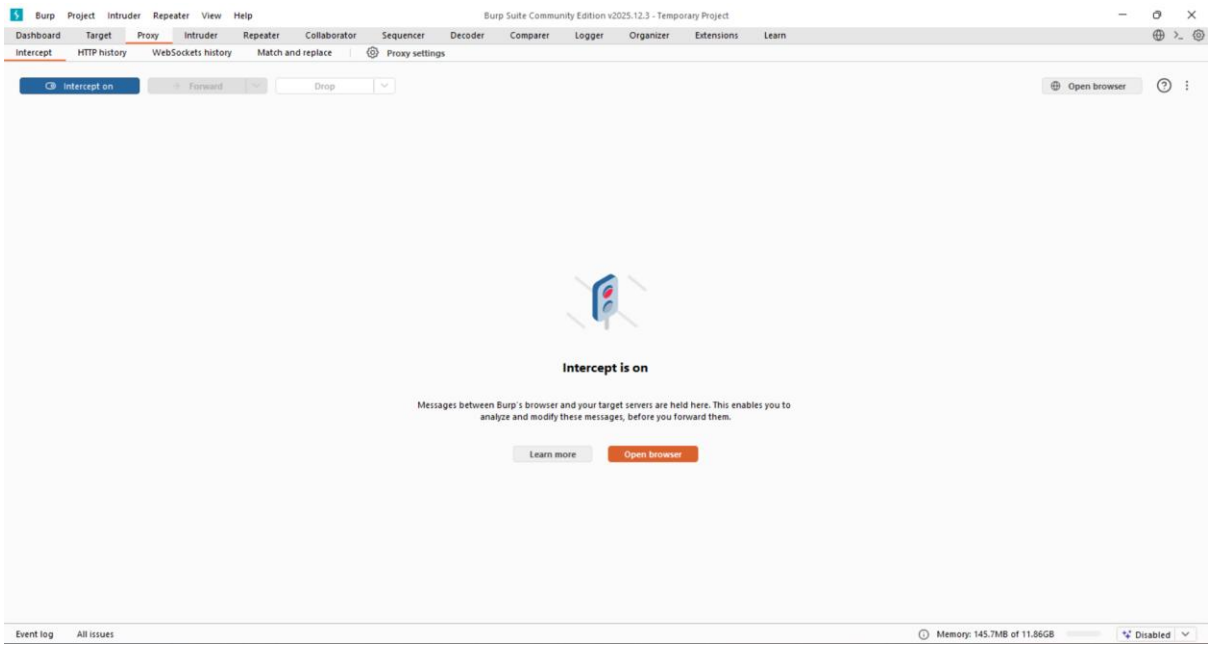
Burp:



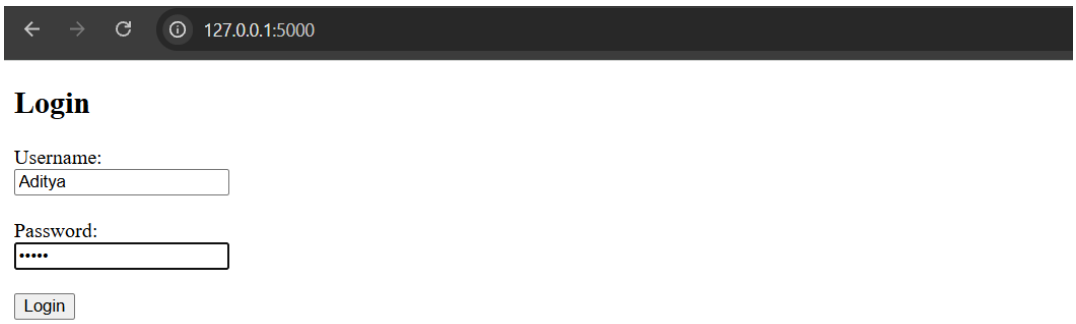
Browser:

Step2: Turn the intercept on. Fill details and click on “login” button.

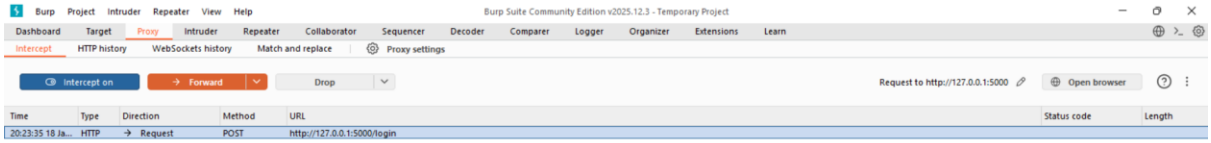
Burp:



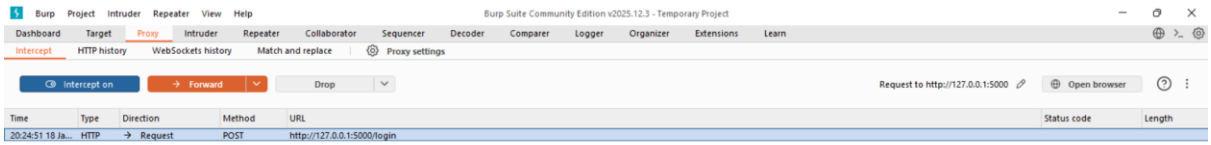
Browser: fill the details

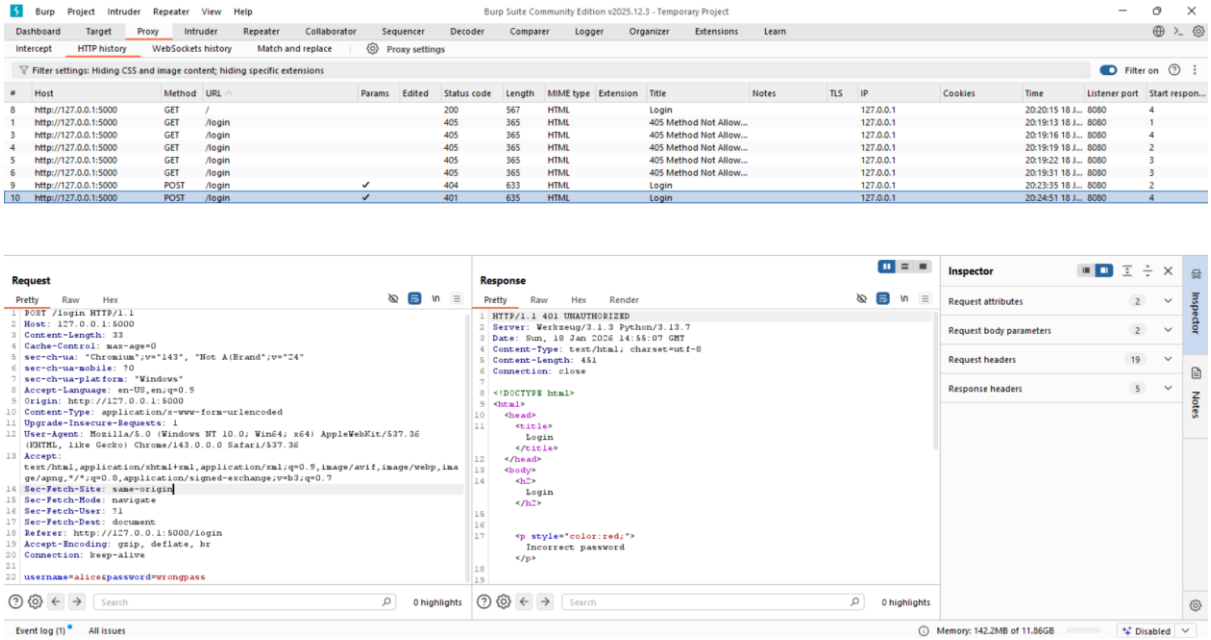


Step3: See the request in the burp.



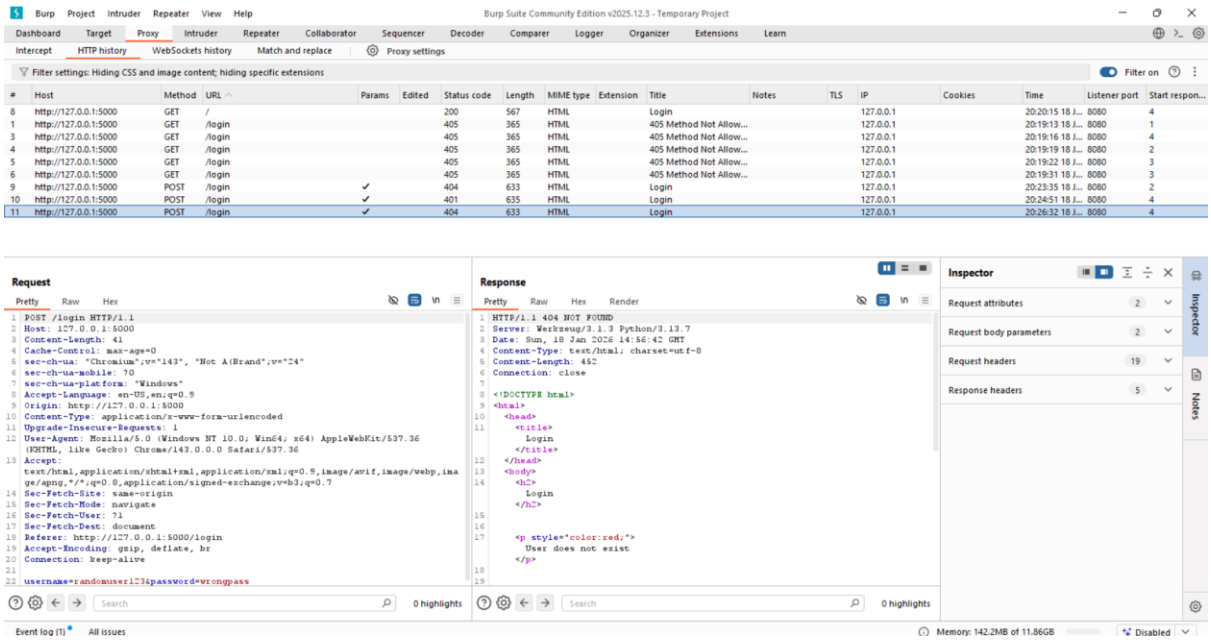
Now, test for valid username:





- Status: 401
- Message: Incorrect password

Now, test for invalid username:



- Status: 404
- Message: User does not exist

Enumeration vulnerability confirmed.

Vulnerable Transmission of Credentials:

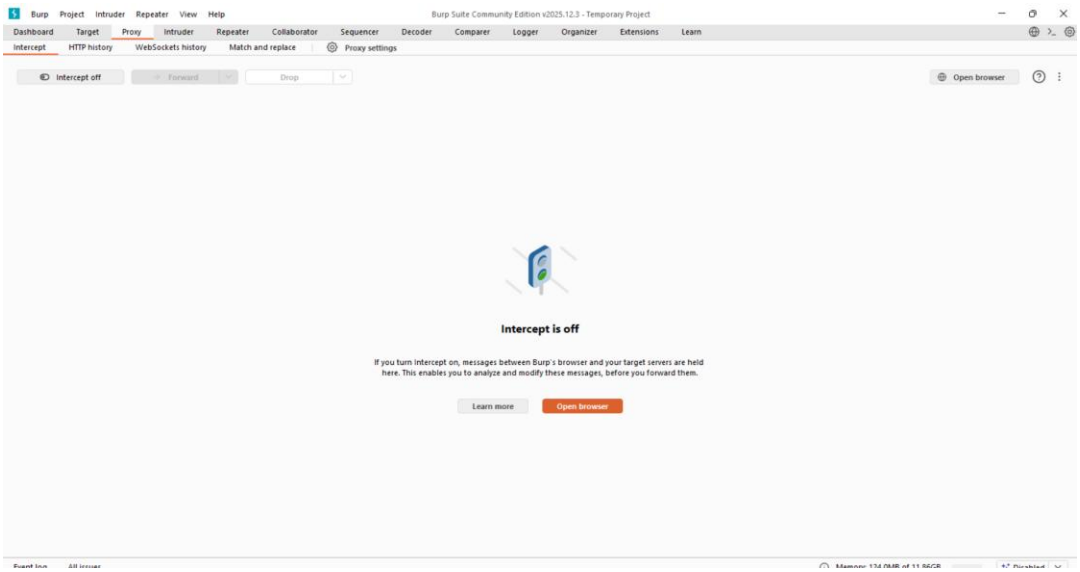
- Sending login details over HTTP lets attackers on the network easily steal them.
- Even with HTTPS, passwords can leak if they're put in URLs, logs, redirects, or cookies.
- Storing credentials in cookies or unsafe redirects allows attackers to log in as users.
- Login pages must load over HTTPS from the start, or attackers can fake them and steal credentials.

Vulnerable Transmission of Credentials (Log in while capturing traffic):

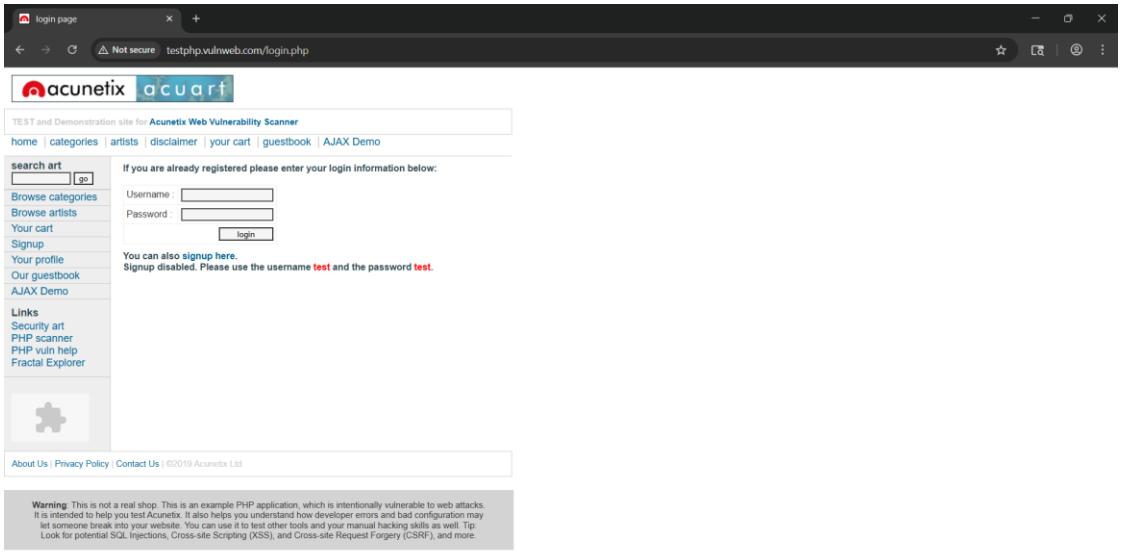
Testing on http://testphp.vulnweb.com/login.php

Step1: Open the Burp and visit the page, and login while recording all requests and responses.

Burp:

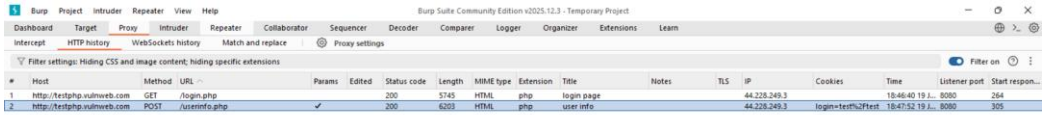


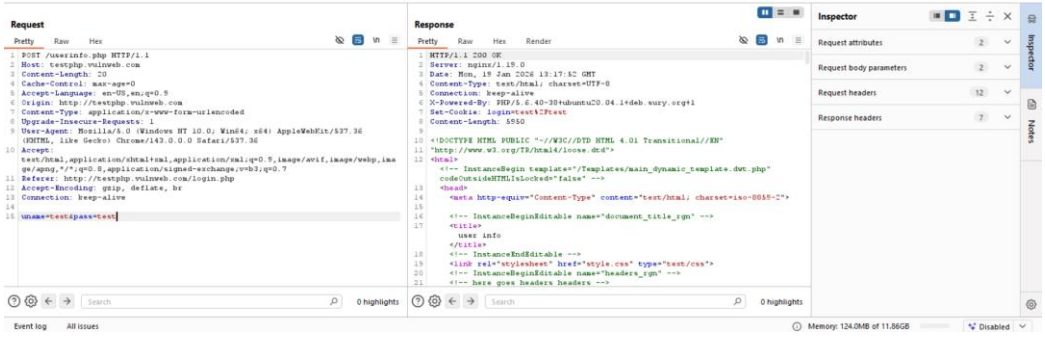
Browser:



Step2: Enter the credentials, and observe.

Burp:





- The username and password (uname=test&pass=test) are sent in clear text over HTTP, making them easy to intercept by anyone on the network.
- The server sets a cookie (login=test/test) containing credentials in an unsafe form, which could be reused by an attacker to impersonate the user.

Password Change Functionality:

There are two main reasons why password change features are important:

1. Regular password changes improve security
2. Users need a quick response if a password is stolen

Password change functionality is essential for security because it lets users regularly update passwords and quickly respond if a password is compromised. However, many applications implement it poorly. Common weaknesses include allowing access without login, revealing whether usernames exist, permitting unlimited password guesses, and improper validation order. These flaws can let attackers discover usernames or passwords and even target other users' accounts.

Password Change Functionality Testing on a local website:

App.py:

```python
from flask import Flask, request, jsonify
app = Flask(__name__)
# Fake user database (plaintext passwords = BAD on purpose)
users = {
    "alice": "password123",
    "bob": "qwerty",
    "admin": "adminpass"
}
@app.route("/change-password", methods=["POST"])
def change_password():
    username = request.form.get("username")
    old_password = request.form.get("old_password")
    new_password = request.form.get("new_password")
    confirm_password = request.form.get("confirm_password")
    # ❌ Verbose error message (username enumeration)
    if username not in users:
        return jsonify({"error": "Username does not exist"}), 400
    # ❌ Old password checked first (brute-forceable)
    if users[username] != old_password:
        return jsonify({"error": "Current password is incorrect"}), 400
    # ❌ New password validation done last
    if new_password != confirm_password:
        return jsonify({"error": "New passwords do not match"}), 400
    users[username] = new_password
    return jsonify({"message": "Password changed successfully"}), 200
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

Step1: We sent a manual HTTP POST request to the password-change endpoint with supplied form data to test how the application responds to an incorrect current password.

```
Windows PowerShell
PS C:\Users\Aditya> curl.exe -X POST http://192.168.0.108:5000/change-password -d "username=alice&old_password=wrongpass&new_password=123&confirm_password=1
23"
{
  "error": "Current password is incorrect"
}
PS C:\Users\Aditya>
```

Step2: Username Enumeration by trying a fake username.

```
PS C:\Users\Aditya> curl.exe -X POST http://192.168.0.108:5000/change-password -d "username=notreal&old_password=test&new_password=1&confirm_password=2"
{
  "error": "Username does not exist"
}
PS C:\Users\Aditya>
```

Tests for username enumeration by checking whether the application reveals that a non-existent username (notreal) is invalid. Confirms username enumeration vulnerability.

Step3: Non-Invasive Password Discovery. Try correct password but mismatched new passwords:

```
PS C:\Users\Aditya> curl.exe -X POST http://192.168.0.108:5000/change-password -d "username=alice&old_password=password123&new_password=1&confirm_password=2
"
{
  "error": "New passwords do not match"
}
PS C:\Users\Aditya>
```

Confirms the correct old password for alice without changing it by exploiting mismatched new passwords (non-invasive password discovery). Confirms the old password is correct without changing it.

Step4: Brute-Force Simulation

```
PS C:\Users\Aditya> curl.exe -X POST http://192.168.0.108:5000/change-password -d "username=alice&old_password=qwerty&new_password=1&confirm_password=2"
{
  "error": "Current password is incorrect"
}
PS C:\Users\Aditya>
```

Attempts a password guess for alice, demonstrating that the old-password field can be brute-forced. No lockout, no delay → brute-force possible.

This response means the application checked the old password (qwerty) first, found it incorrect for user alice, and rejected the request without changing anything, confirming that the password-guessing check is working and can be brute-forced.

Read more:

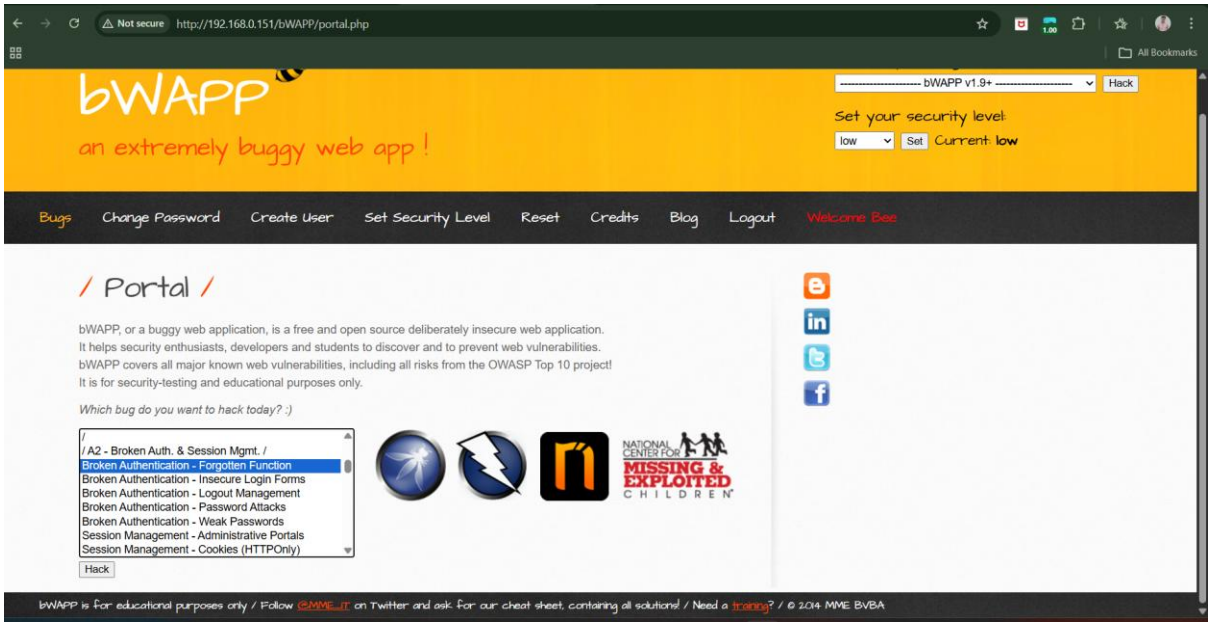
1. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/04-Authentication_Testing/09-Testing_for_Weak_Password_Change_or_Reset_Functionalities?utm_source=chatgpt.com
2. https://www.acunetix.com/blog/web-security-zone/common-password-vulnerabilities/?utm_source=chatgpt.com
3. https://www.authgear.com/post/authentication-security-password-reset-best-practices-and-more?utm_source=chatgpt.com
4. https://owasp.org/www-community/attacks/Password_Spraying_Attack?utm_source=chatgpt.com

Forgotten Password Functionality:

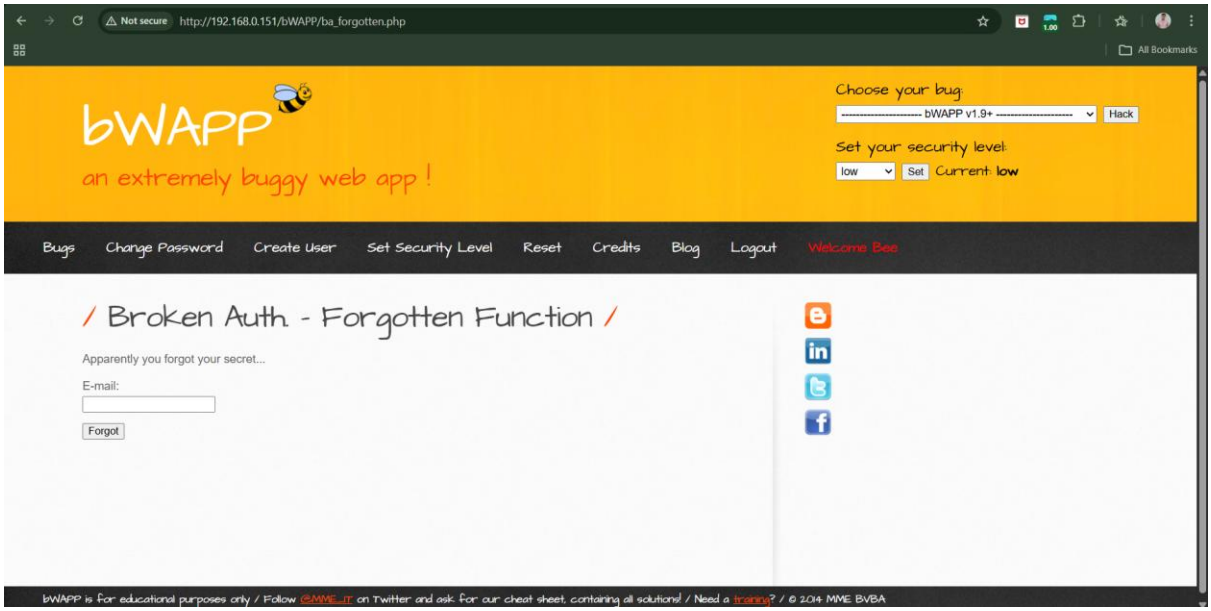
A forgotten password feature is meant to help users, but if designed poorly, it becomes the easiest way for attackers to break into accounts without ever guessing a password. Think of the forgot-password feature as a side entrance to the account.

Forgotten Password Functionality on the bWAPP:

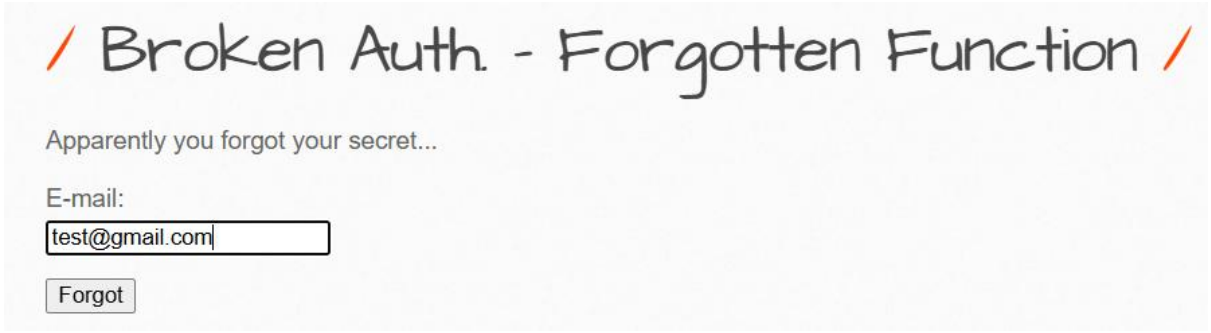
Step1: Open the local website, and visit the “Broken Authentication – Forgotten Function”.



Following screen will appear:



Step2: enter the mail id to test.



When clicked on “Forgot” button: following enumeration came “Invalid user!”.

Basically, it tells whether the “test@gmail.com” exists or not. This allowed to know which email accounts are using this website.

“Remember Me” Functionality:

“Remember Me” features save login details in browser cookies so users don’t have to log in again, but they’re often insecure. Some sites store the username or a simple ID in the cookie and then trust it to log the user in, which attackers can guess or change to access other accounts without a password. Even when the data is encrypted, bugs like cross-site scripting can let attackers steal the cookie and hijack the account. In short, poorly designed “Remember Me” cookies can allow attackers to log in as users without proper authentication.

User Impersonation Functionality:

User impersonation lets one user act as another, but if poorly designed it can allow attackers to switch accounts without permission. Common mistakes include trusting editable data like cookies, exposing hidden impersonation URLs, and using shared “backdoor” passwords. These flaws can let attackers access other users’ data or even take full admin control of an application.

User Impersonation Functionality testing on a local website:

Main.py:

main.py > ...

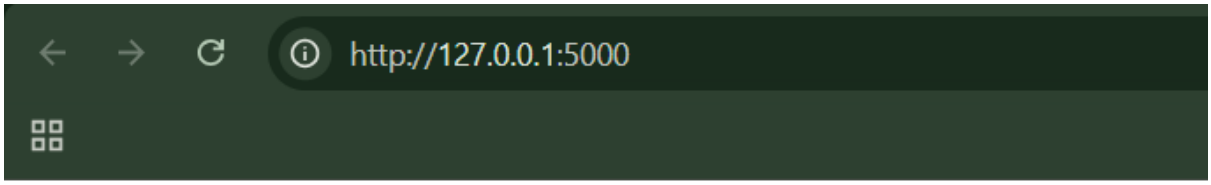
```python
from flask import Flask, request, redirect, make_response

app = Flask(__name__)

# Fake user database
users = {
    "alice": {"password": "alice123", "role": "user"},
    "bob": {"password": "bob123", "role": "user"},
    "admin": {"password": "admin123", "role": "admin"},
}

BACKDOOR_PASSWORD = "letmein"  # 🔥 Vulnerable backdoor password


# ------------------------------
# Home
# ------------------------------
@app.route("/")
def home():
    return """
    <h2>Vulnerable Demo App</h2>
    <a href="/login">Login</a><br>
    <a href="/dashboard">Dashboard</a>
    """

# ------------------------------
# Login (with backdoor password)
# ------------------------------
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form.get("username")
        password = request.form.get("password")

        # 🔥 Backdoor password vulnerability
        if password == BACKDOOR_PASSWORD:
```

main.py > ...

```python
def login():
        if password == BACKDOOR_PASSWORD:
            resp = make_response(redirect("/dashboard"))
            resp.set_cookie("user", username)
            return resp

        # Normal login
        if username in users and users[username]["password"] == password:
            resp = make_response(redirect("/dashboard"))
            resp.set_cookie("user", username)
            return resp

        return "Login failed"

    return """
    <h3>Login</h3>
    <form method="POST">
        Username: <input name="username"><br>
        Password: <input name="password"><br>
        <button type="submit">Login</button>
    </form>
    """


# ------------------------------
# Dashboard (cookie trust)
# ------------------------------
@app.route("/dashboard")
def dashboard():
    user = request.cookies.get("user")

    if not user:
        return redirect("/login")

    role = users.get(user, {}).get("role", "unknown")

    return f"""
```

```python
main.py > ...
64  def dashboard():
72      return f"""
73      <h3>Dashboard</h3>
74      Logged in as: <b>{user}</b><br>
75      Role: <b>{role}</b><br><br>
76
77      <a href="/admin/impersonate?user=alice">Impersonate Alice</a><br>
78      <a href="/admin/impersonate?user=admin">Impersonate Admin</a><br>
79      """
80
81
82  # ------------------------------
83  # Hidden impersonation endpoint
84  # ------------------------------
85  @app.route("/admin/impersonate")
86  def impersonate():
87      target_user = request.args.get("user")
88
89      # 🔥 No access control at all
90      resp = make_response(redirect("/dashboard"))
91      resp.set_cookie("user", target_user)
92      return resp
93
94
95  # ------------------------------
96  # Logout
97  # ------------------------------
98  @app.route("/logout")
99  def logout():
100     resp = make_response(redirect("/"))
101     resp.delete_cookie("user")
102     return resp
103
104
105 if __name__ == "__main__":
106     app.run(debug=True)
```

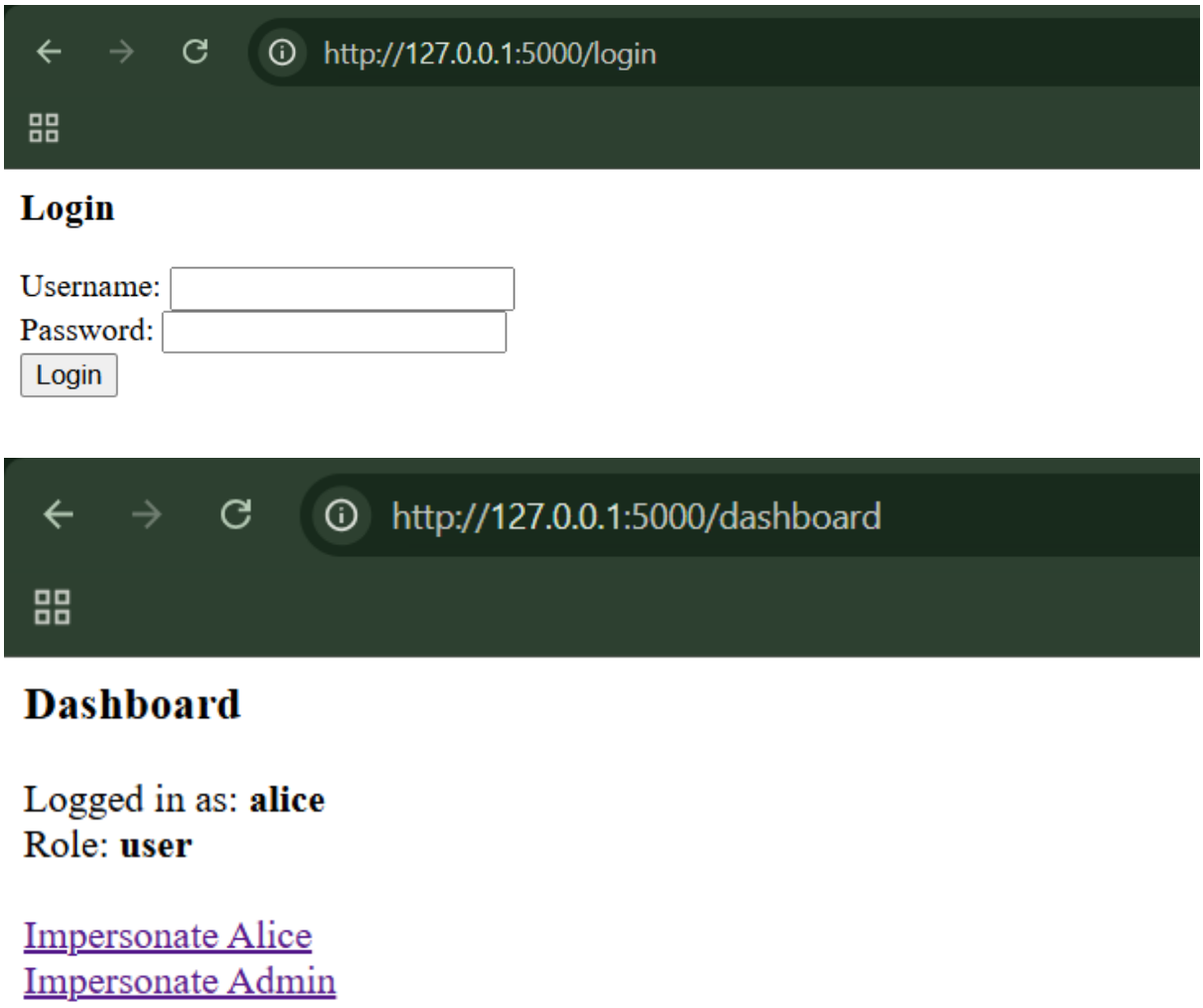
Browser:



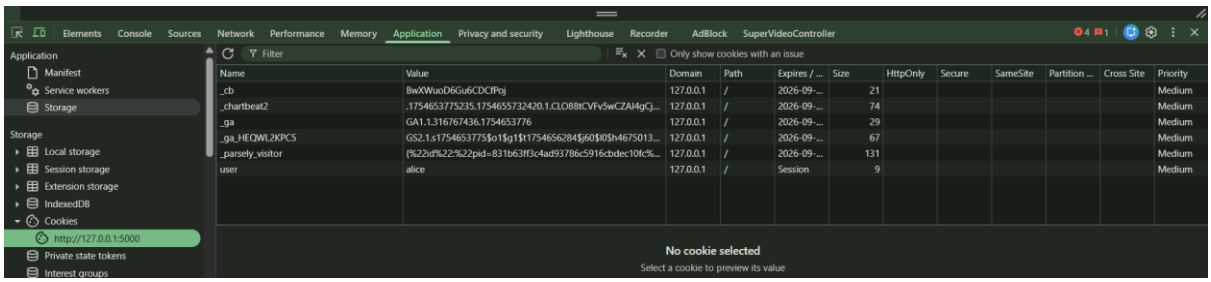
Vulnerable Demo App

Login
Dashboard

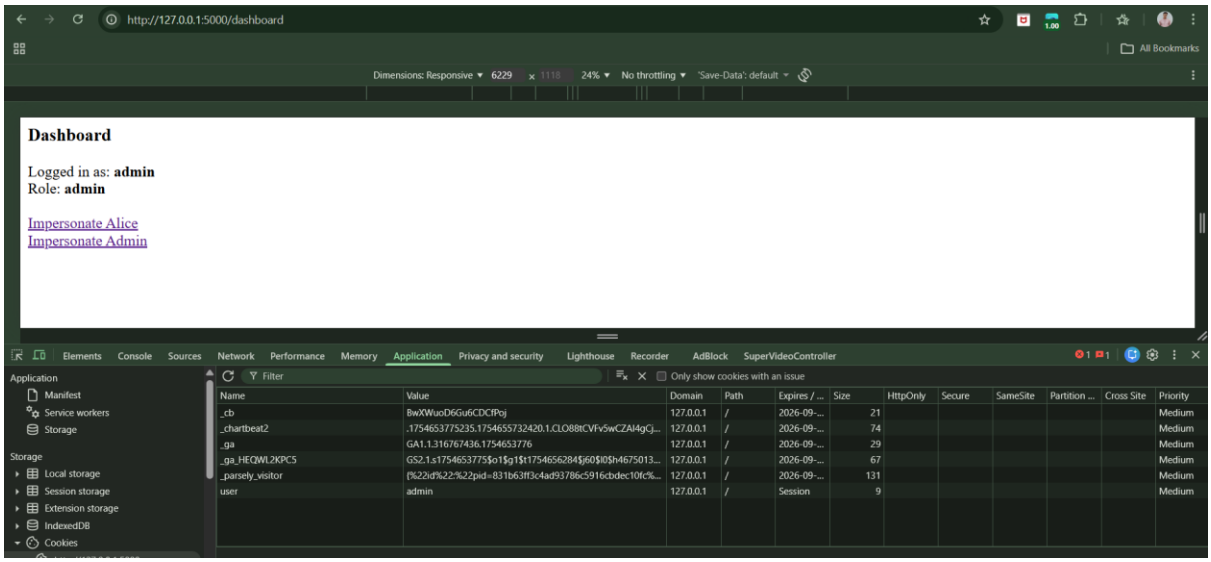
Click on “login”: enter the credentials as per the code alice:alice123



Step1: Cookie tampering. Open DevTools → Storage → Cookies. Following cookie can be seen:



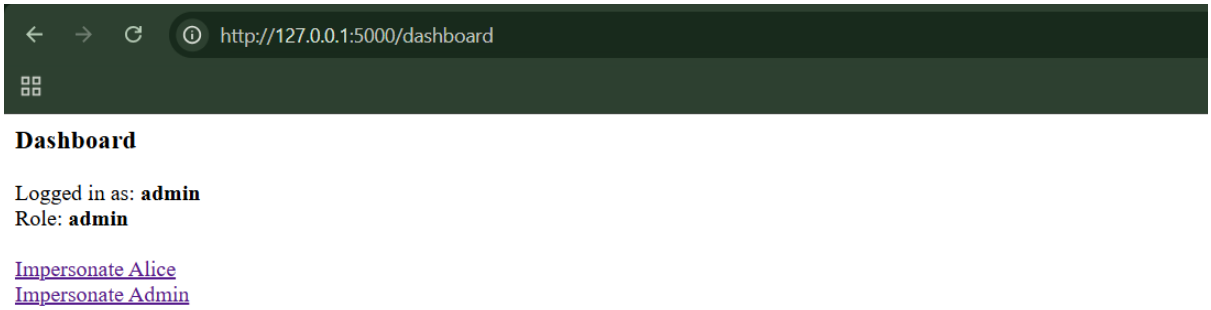
Step2: Change “alice” to “admin”. Then just refresh the page.



This shows that we can tamper the cookie.

Step2: Hidden impersonation URL. Visit directly:

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



Clearly, No login required, we gained admin account without entering any credentials.

Step3: Backdoor password discovery. When tried with password “letmein”, we can see that we managed to login to both alice and bob account, which means this is the common password of all.

Incomplete Validation of Credentials:

Incomplete validation of credentials means a website doesn’t properly check passwords the way it should, which secretly makes passwords much easier to guess or break.

Common broken behaviours:

1. Password truncation (only first N characters checked)
2. Case-insensitive passwords
3. Special characters are stripped out

Why this is dangerous?

That means:

- Instead of millions or billions of possible passwords,
- You might end up with thousands or hundreds

That makes automated guessing attacks MUCH easier.

Non-Unique Usernames:

Non-unique usernames means a website lets multiple people register with the *same username*, which can leak passwords or let people access the wrong account.

Predictable Usernames:

Predictable usernames means a website creates usernames in a pattern that's easy to guess, letting attackers figure out all valid usernames without much effort.

- Auto-generated usernames can follow obvious patterns
- Attackers only need a few examples to figure them out
- This reveals all valid usernames
- Once usernames are known, many attacks become easier
- It's quiet, fast, and hard to detect

Predictable initial passwords:

Predictable initial passwords means a system gives users "temporary" passwords that follow an obvious pattern, making it easy to guess other users' passwords.

- Some systems generate passwords instead of letting users choose
- If those passwords follow patterns, they're guessable
- Attackers only need a few examples
- This is especially common in corporate/internal systems
- Predictable passwords + predictable usernames = serious risk

Insecure distribution of credentials:

Insecure distribution of credentials means a system gives users their login details (or activation links) in unsafe ways that can be reused, guessed, or intercepted later.

- Sending credentials by email or post is risky
- If passwords don't expire, they live forever
- Activation links must be:
  - Random
  - Unpredictable
  - Single-use
  - Time-limited
- Otherwise, old messages become future vulnerabilities

Implementation Flaws in Authentication:

Authentication systems can fail due to small coding mistakes, even if their design is strong. These hidden errors are hard to detect and are often exploited in critical systems like banks.

Some majors are:

- Fail-Open Login Mechanisms
- Defects in Multistage Login Mechanisms
- Insecure Storage of Credentials

Fail-Open Login Mechanisms:

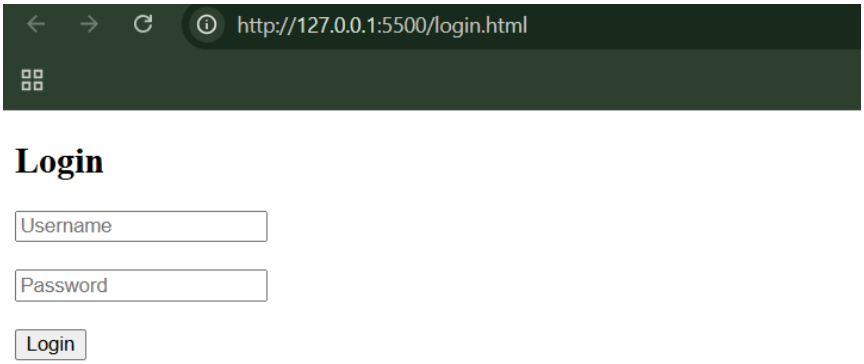
Fail-open login happens when a system allows access after an error instead of blocking it. Attackers exploit this by causing errors during login to bypass authentication.

Observing Fail-Open Login Mechanisms on a local Webiste:

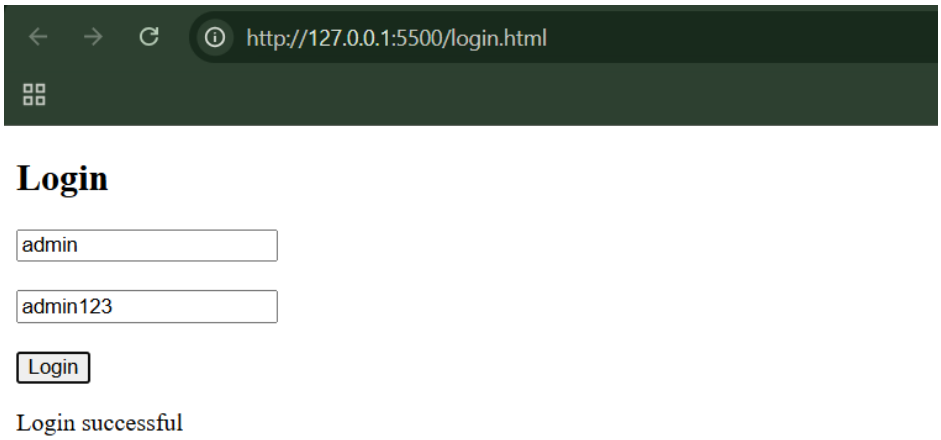
Index.html:

```
login.html > html > body > script > login
<html>
<body>
  <h2>Login</h2>
  <input id="username" placeholder="Username"><br><br>
  <input id="password" placeholder="Password"><br><br>
  <button onclick="login()">Login</button>
  <p id="msg"></p>
  <script>
    function login() {
      try {
        let uname = document.getElementById("username").value;
        let pass = document.getElementById("password").value;
        // Simulated database check
        if (uname === "admin" && pass === "admin123") {
          document.getElementById("msg").innerText = "Login successful";
          return;
        }
        if (uname === "" || pass === "") {
          throw "Missing input";
        }
        document.getElementById("msg").innerText = "Login failed";
      }
      catch (e) {
        // ❌ FAIL-OPEN BUG
      }
      // ❌ User is logged in even after error
      document.getElementById("msg").innerText = "Login successful";
    }
  </script>
</body>
</html>
```

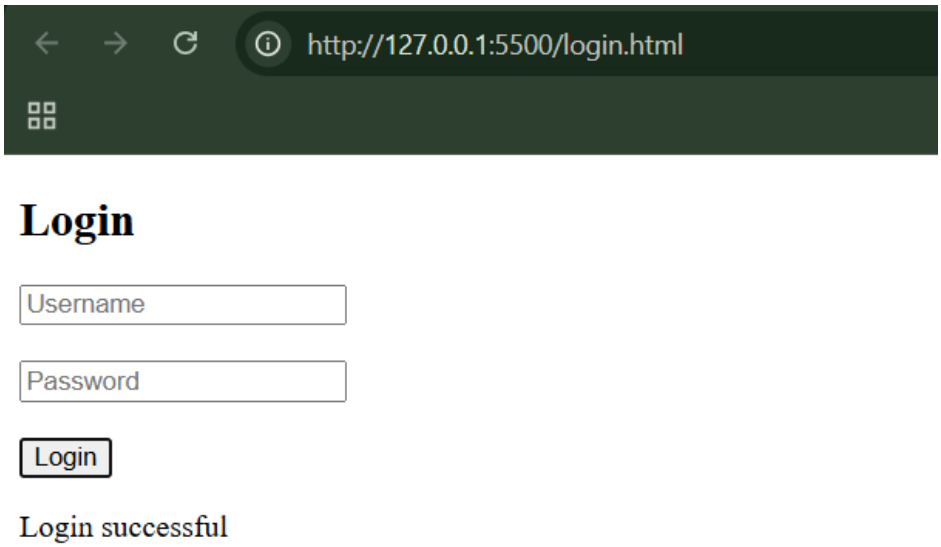
Output:



Enter the correct credentials:



Enter the empty credentials:



Empty input causes an error → error is ignored → login succeeds.

Secured version:

login.html > html > body > button

```html
<!DOCTYPE html>
<html>
<head>
  <title>Secure Login Demo</title>
</head>
<body>
  <h2>Login</h2>
  <input id="username" placeholder="Username"><br><br>
  <input id="password" placeholder="Password"><br><br>
  <button onclick="login()">Login</button>
  <p id="msg"></p>
  <script>
    function login() {
      try {
        let uname = document.getElementById("username").value;
        let pass = document.getElementById("password").value;
        // Input validation
        if (uname === "" || pass === "") {
          throw "Username or password missing";
        }
        // Simulated database check
        if (uname === "admin" && pass === "admin123") {
          document.getElementById("msg").innerText = "Login successful";
          return;
        }
        // Invalid credentials
        document.getElementById("msg").innerText = "Login failed";
      }
      catch (e) {
        // ✅ FAIL-CLOSED
        document.getElementById("msg").innerText = "Login error. Access denied.";
        return;
      }
    }
  </script>
</body>
</html>
```

Defects in Multistage Login Mechanisms:

1. Multistage login looks more secure, but is often more fragile
2. Developers wrongly assume:
   - Stages cannot be skipped
   - Data stays unchanged
   - Same user is used in all stages
3. Attackers exploit:
   - Stage skipping
   - Parameter manipulation
   - Identity mixing
   - Broken random questions
4. Poor multistage design can be less secure than a simple login
5. Always validate:
   - Each stage on the server
   - User identity consistency
   - State using server-side sessions

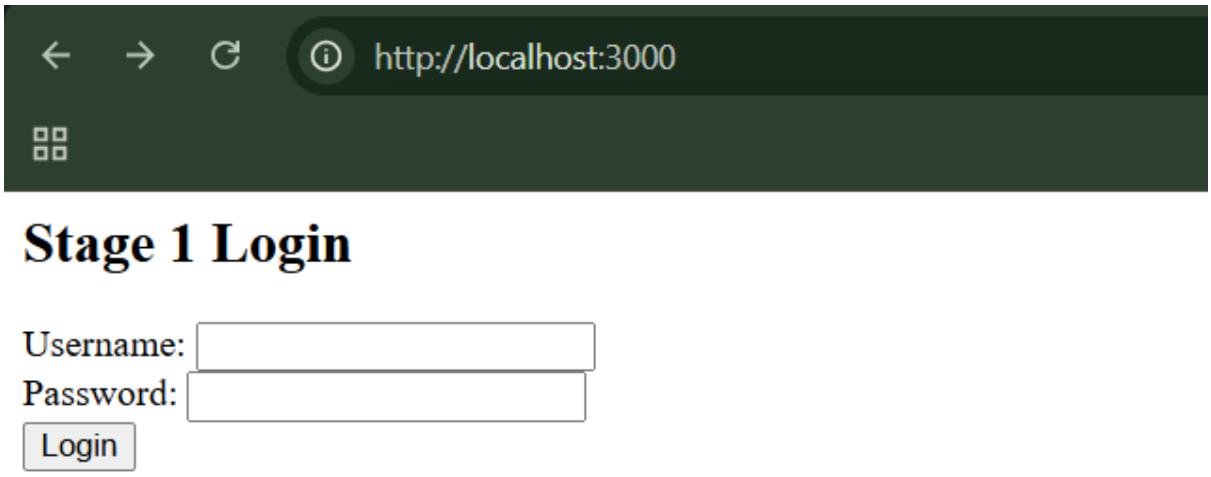
Observing Defects in Multistage Login Mechanisms on a local website:

App.js:

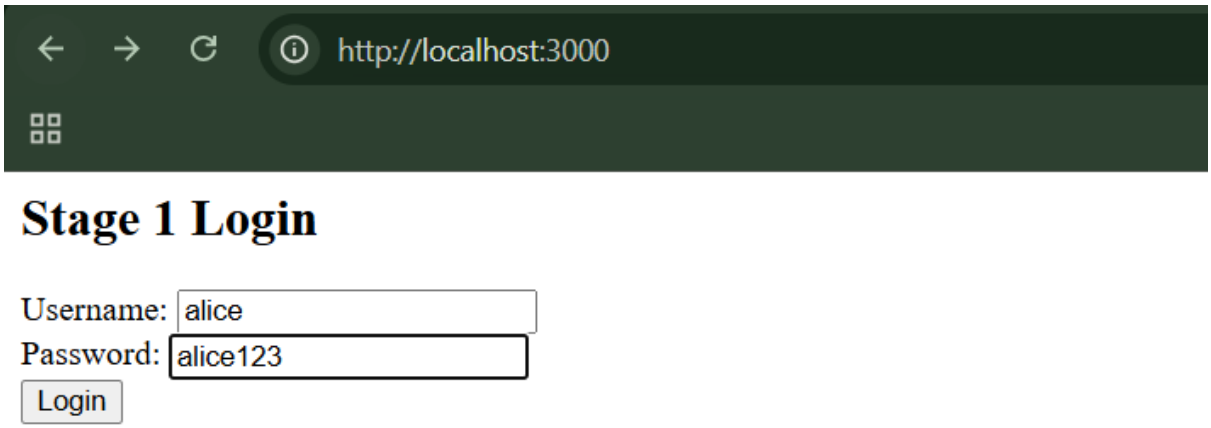
```
JS app.js > users > bob
1   const express = require("express");
2   const bodyParser = require("body-parser");
3   const app = express();
4   app.use(bodyParser.urlencoded({ extended: false }));
5   // Fake user database
6   const users = {
7     alice: {
8       password: "alice123",
9       secret: {
10        pet: "cat",
11        city: "paris"
12      }
13    },
14    bob: {
15      password: "bob123",
16      secret: {
17        pet: "dog",
18        city: "london"
19      }
20    }
21  };
22  // ---------- STAGE 1 ----------
23  app.post("/stage1", (req, res) => {
24    const { username, password } = req.body;
25    if (users[username] && users[username].password === password) {
26      // VULNERABILITY: trusting client-side state
27      res.send(`
28        <form method="POST" action="/stage2">
29          <input type="hidden" name="username" value="${username}">
30          <input type="hidden" name="stage1complete" value="true">
31          <button>Continue to Stage 2</button>
32        </form>
33      `);
34    } else {
35      res.send("Invalid username or password");
36    }
37  });
38  // ---------- STAGE 2 ----------
39  app.post("/stage2", (req, res) => {
40    const { username } = req.body;
41    // Random question (BROKEN)
42    const questions = ["pet", "city"];
43    const question = questions[Math.floor(Math.random() * questions.length)];
44    res.send(`
45      <form method="POST" action="/stage3">
46        <p>What is your favorite ${question}?</p>
47        <input name="answer">
48        <!-- VULNERABILITY: question sent to client -->
49        <input type="hidden" name="question" value="${question}">
50        <input type="hidden" name="username" value="${username}">
51        <input type="hidden" name="stage2complete" value="true">
52        <button>Login</button>
53      </form>
54    `);
55  });
56  // ---------- STAGE 3 ----------
57  app.post("/stage3", (req, res) => {
58    const { username, question, answer, stage2complete } = req.body;
```

```
JS app.js > users > bob
57  app.post("/stage3", (req, res) => {
58    const { username, question, answer, stage2complete } = req.body;
59    // VULNERABILITY: trusts stage2complete flag
60    if (stage2complete === "true") {
61      if (users[username] && users[username].secret[question] === answer) {
62        res.send(`✅ Logged in as ${username}`);
63      } else {
64        res.send("❌ Wrong answer");
65      }
66    } else {
67      res.send("❌ Stage 2 not completed");
68    }
69  });
70  // ---------- HOME ----------
71  app.get("/", (req, res) => {
72    res.send(`
73      <h2>Stage 1 Login</h2>
74      <form method="POST" action="/stage1">
75        Username: <input name="username"><br>
76        Password: <input name="password"><br>
77        <button>Login</button>
78      </form>
79    `);
80  });
81  app.listen(3000, () => {
82    console.log("Vulnerable app running at http://localhost:3000");
83  });
```

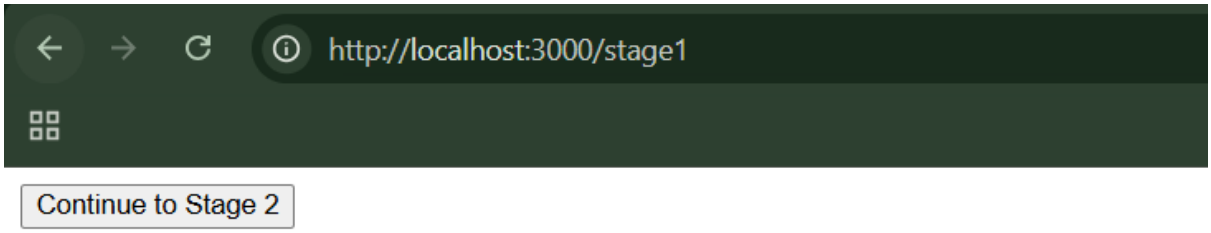
Browser:



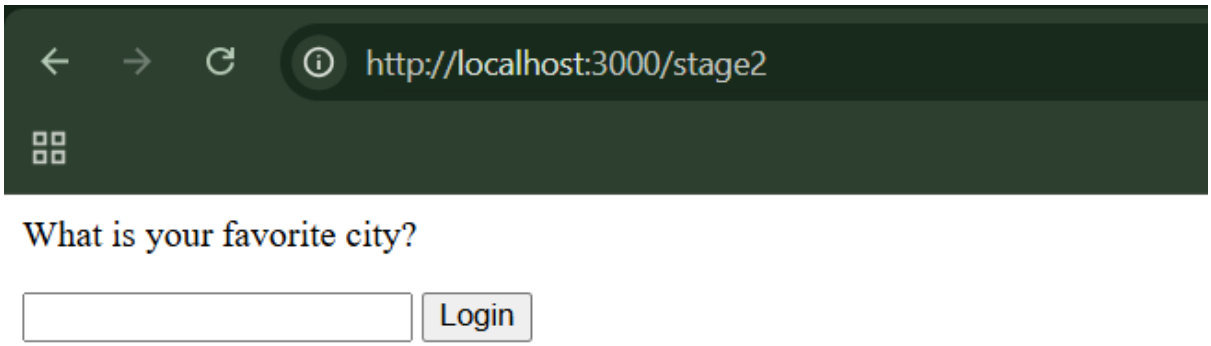
Step1: do a valid login



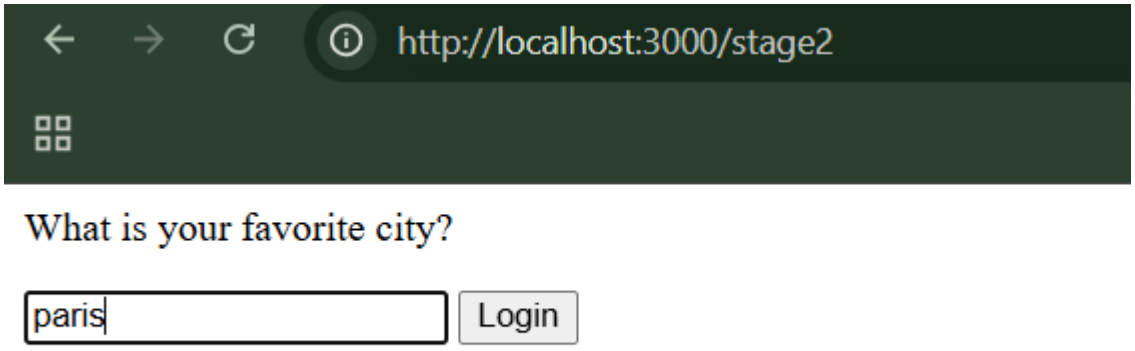
When clicked on “login” button:



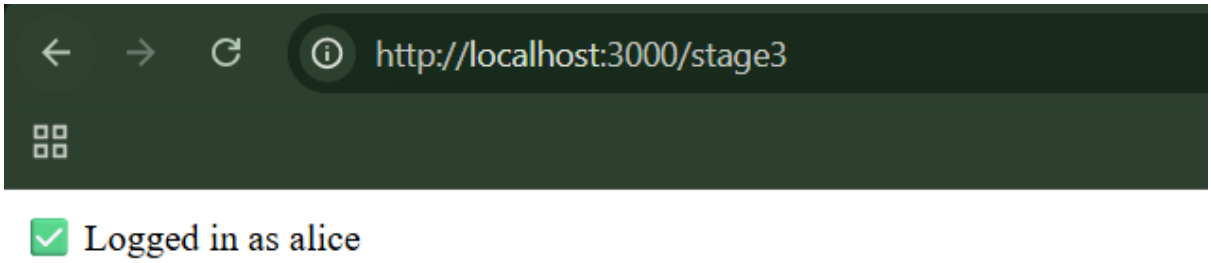
When clicked, we are sent to the stage2:



Step2: Answer the secret question

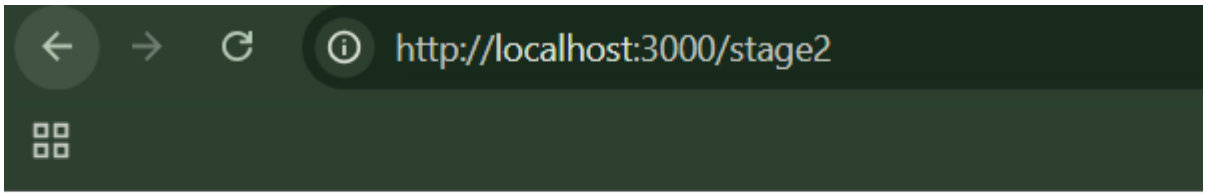


When answered:



You are logged in as alice. This confirms the app works normally

Step3: Now, we will Login without entering username + password. In new tab, visit this: http://localhost:3000/stage2



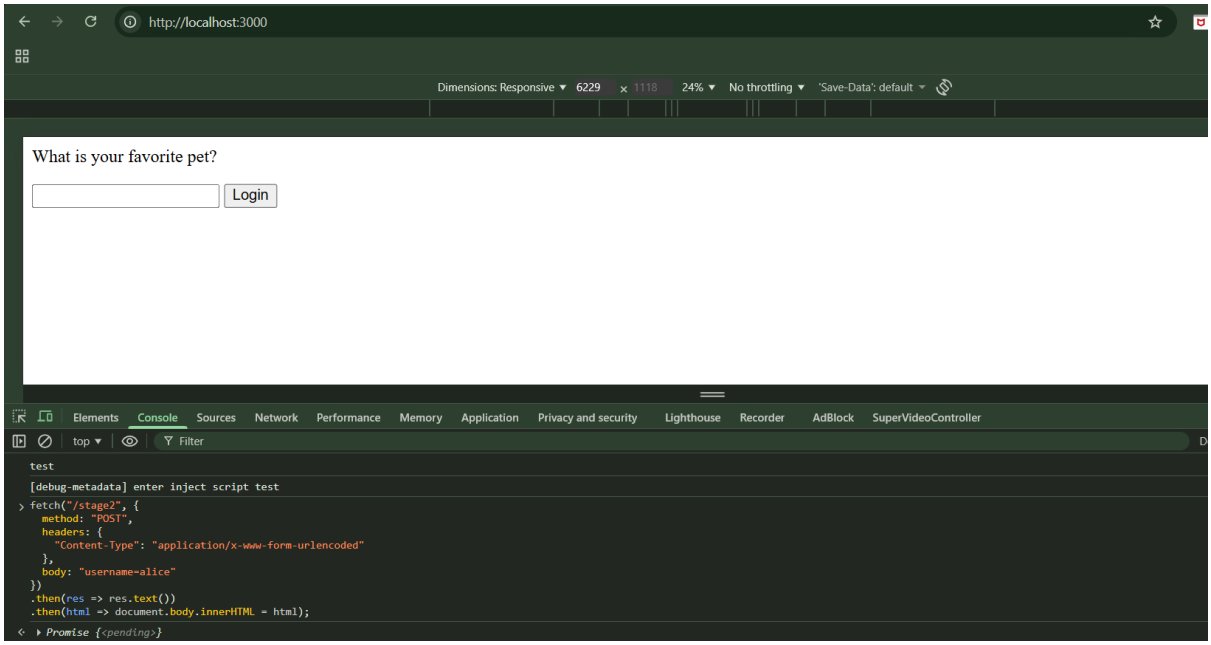
Cannot GET /stage2

As expected we see nothing, as stage2 expects a POST Request.

Step4: Go to the console of the page, and paste this code:

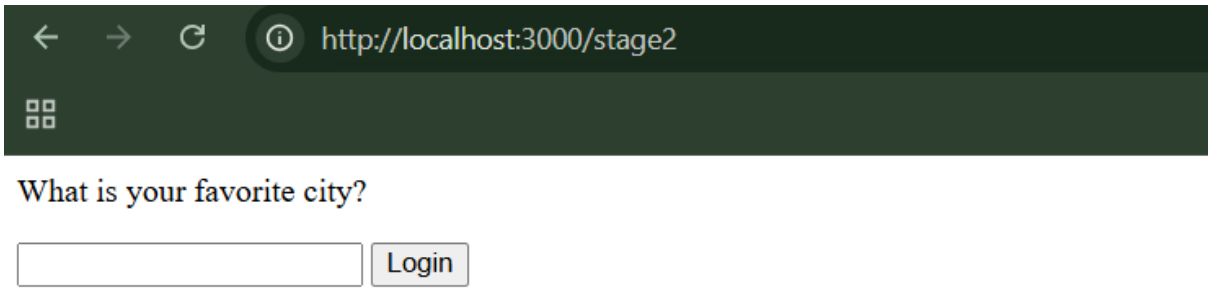
```
fetch("/stage2", {
  method: "POST",
  headers: {
    "Content-Type": "application/x-www-form-urlencoded"
  },
  body: "username=alice"
})
.then(res => res.text())
.then(html => document.body.innerHTML = html);
```

Browser:



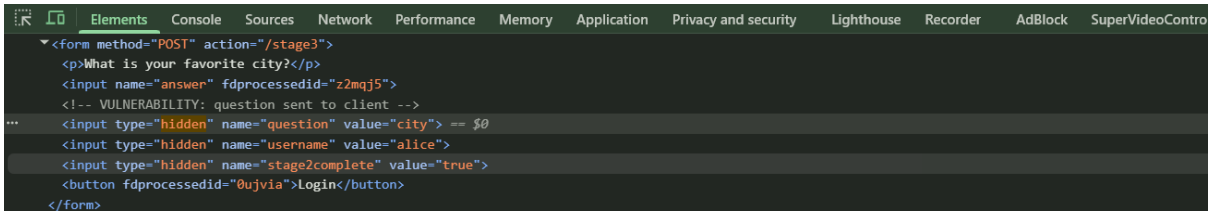
We are now on Stage 2. We never entered a password. Stage 1 is bypassed

Step5: Answer any question you want, not the one shown. For this Reach Stage 2 normally, use credentials alice/alice123.

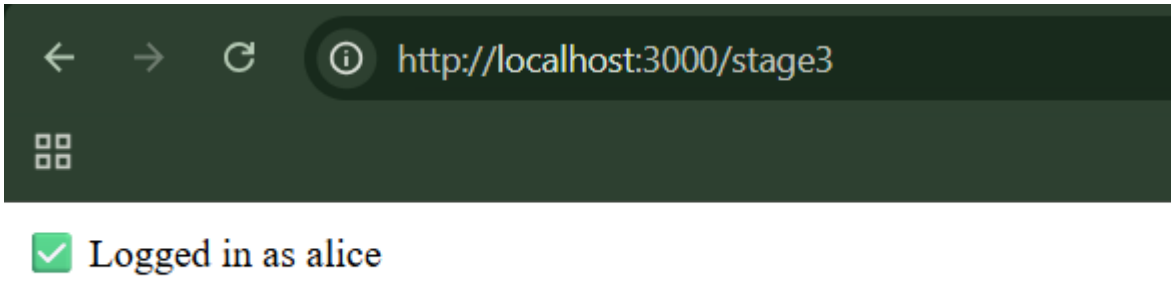


Step6: Right-click → Inspect

- Right-click on the page
- Click Inspect
- Go to Elements
- Find the hidden field



Step7: Change the value from “city” to “pet”, then enter the answer. Then click on “Login”, and done!



Login succeeds. We answered a different question than the one shown. The server trusts the client

Why All This Works?

The application:

- Trusts hidden fields
- Trusts client flags
- Does not track login progress on the server
- Does not bind stages to one user session

So: The browser controls authentication

Safe app.js:

```
JS app.js > ...
1   const express = require("express");
2   const bodyParser = require("body-parser");
3   const session = require("express-session");
4
5   const app = express();
6   app.use(bodyParser.urlencoded({ extended: false }));
7
8   // -------- SESSION SETUP --------
9   app.use(session({
10    secret: "very-secret-key",
11    resave: false,
12    saveUninitialized: false
13  }));
14
15  // -------- FAKE DATABASE --------
16  const users = {
17    alice: {
18      password: "alice123",
19      secret: {
20        pet: "cat",
21        city: "paris"
22      }
23    },
24    bob: {
25      password: "bob123",
26      secret: {
27        pet: "dog",
28        city: "london"
29      }
30    }
```

```
JS app.js > ...
33  // -------- HOME --------
34  app.get("/", (req, res) => {
35    res.send(`
36      <h2>Stage 1 Login</h2>
37      <form method="POST" action="/stage1">
38        Username: <input name="username"><br>
39        Password: <input name="password"><br>
40        <button>Login</button>
41      </form>
42    `);
43  });
44
45  // -------- STAGE 1 --------
46  app.post("/stage1", (req, res) => {
47    const { username, password } = req.body;
48
49    if (users[username] && users[username].password === password) {
50      // SERVER remembers user & stage
51      req.session.user = username;
52      req.session.stage = 1;
53
54      res.redirect("/stage2");
55    } else {
56      res.send("❌ Invalid username or password");
57    }
58  });
59
60  // -------- STAGE 2 --------
61  app.get("/stage2", (req, res) => {
62    if (req.session.stage !== 1) {
```

JS app.js > ...

```
61  app.get("/stage2", (req, res) => {
62    if (req.session.stage !== 1) {
63      return res.send("❌ Access denied");
64    }
65
66    const questions = ["pet", "city"];
67    const question = questions[Math.floor(Math.random() * questions.length)];
68
69    // SERVER stores the question
70    req.session.question = question;
71    req.session.stage = 2;
72
73    res.send(`
74      <h2>Stage 2</h2>
75      <p>What is your favorite ${question}?</p>
76      <form method="POST" action="/stage2">
77        <input name="answer">
78        <button>Continue</button>
79      </form>
80    `);
81  });
82
83  // -------- STAGE 2 VERIFY --------
84  app.post("/stage2", (req, res) => {
85    if (req.session.stage !== 2) {
86      return res.send("❌ Access denied");
87    }
88
89    const user = req.session.user;
```

JS app.js > ...

```
84  app.post("/stage2", (req, res) => {
88    const user = req.session.user;
89    const question = req.session.question;
90    const answer = req.body.answer;
91    if (users[user].secret[question] === answer) {
92      req.session.stage = 3;
93      res.redirect("/stage3");
94    } else {
95      res.send("❌ Wrong answer");
96    }
97  });
98  // -------- STAGE 3 --------
99  app.get("/stage3", (req, res) => {
100   if (req.session.stage !== 3) {
101     return res.send("❌ Access denied");
102   }
103   res.send(`✅ Logged in securely as ${req.session.user}`);
104 });
105 // -------- LOGOUT --------
106 app.get("/logout", (req, res) => {
107   req.session.destroy(() => {
108     res.send("Logged out");
109   });
110 });
111
112 app.listen(3000, () => {
113   console.log("✅ Secure app running at http://localhost:3000");
114 });
```

Insecure Storage of Credentials:

Storing passwords without proper protection means that any other security bug can turn into a full password leak, putting all users at risk.

Securing Authentication:

What does "securing authentication" mean?

Authentication is how a system checks who you are (for example, using a username, password, OTP, fingerprint, etc.).

- Authentication security must balance security, usability, and cost.
- Too much security can reduce real-world safety.
- There are many authentication attacks, so protecting against everything is hard.
- Developers usually focus on blocking the most serious threats.
- Decisions depend on:
    - How critical the app is
    - User tolerance
    - Support and financial costs
- There is no perfect solution, only appropriate trade-offs.

Some major ones are:

- Use Strong Credentials
- Handle Credentials Secretively
- Validate Credentials Properly
- Prevent Information Leakage
- Prevent Brute-Force Attacks
- Prevent Misuse of the Password Change Function
- Prevent Misuse of the Account Recovery Function
- Log, Monitor, and Notify

Use Strong Credentials:

Credentials usually mean:

- Username
- Password

Using strong credentials makes it harder for attackers to guess or break into accounts.

- Strong credentials make accounts harder to hack.
- Passwords should be:
    - Long
    - Mixed with letters, numbers, symbols
    - Not common words or names
    - Different from usernames
    - Not reused or slightly modified
- Usernames must be unique.
- System-generated credentials must be random and unpredictable.
- Users should be allowed to create very strong passwords, not blocked by weak rules.

Handle Credentials Secretively:

Credentials (username, password, tokens, etc.) are secrets. They must be protected everywhere:

- When they are created
- When they are stored
- When they are sent over the network

If credentials leak at any point, attackers can take over accounts.

- Credentials must be protected at all stages.
- Always use HTTPS, never custom encryption.
- Login pages must load over HTTPS.
- Send credentials only via POST, never URLs or cookies.
- Store passwords using strong hashing + salt.
- “Remember me” should avoid storing passwords.
- Protect against XSS attacks.
- Allow password changes and enforce periodic updates.
- Send new credentials securely and force change on first login.
- UI tricks can reduce keylogger risk, but cannot stop full system compromise.

Validate Credentials Properly:

Validating credentials properly means:

Checking login data correctly, safely, and consistently, even when something goes wrong or an attacker tries to confuse the system. Many authentication bugs happen not because of weak passwords, but because of bad validation logic.

- Passwords must be validated exactly as entered.
- Any unexpected error during login must fail securely.
- Authentication logic must be heavily reviewed.
- User impersonation is dangerous and must be tightly controlled.
- Multistage logins must:
    - Store all state on the server
    - Prevent skipping or reusing steps
    - Always verify previous stages
    - Never reveal where login failed
- Random security questions must be:
    - Server-controlled
    - Consistent
    - Protected from manipulation
- Small behaviour differences can leak usernames.

Prevent Information Leakage:

Information leakage happens when a login system reveals clues that help attackers:

- Guess usernames
- Guess passwords
- Understand how security controls work

Even small hints can be combined into a powerful attack.

- Authentication must not reveal any clues.
- Always use generic error messages.
- Handle all login failures through one code path.
- Account lockout messages must not reveal:
    - Valid usernames
    - Lockout rules
- Self-registration should not disclose existing usernames.
- Best solutions:
    - System-generated usernames
    - Email-based registration with identical responses

Prevent Brute-Force Attacks:

brute-force attack is when an attacker:

- Tries many passwords automatically
- Uses scripts or bots
- Keeps guessing until something works

The goal of defense is: Make guessing slow, difficult, and noisy.

- Brute-force attacks rely on automation and speed.
- Protect **all** authentication-related features.
- Unpredictable usernames slow attackers.
- Temporary account suspension is usually better than permanent lockout.
- Never reveal:
    - Which account is suspended
    - Lockout rules or timings
- During suspension, reject logins immediately.
- Per-account lockout alone does not stop “one password, many users” attacks.
- CAPTCHA helps block automation, even if not perfect.
- Poor CAPTCHA implementations can be bypassed.

Prevent Misuse of the Password Change Function:

The password change feature is very powerful:

- If abused, an attacker can lock users out
- Or silently take over accounts

So this function must be even more secure than login.

- Password change is a critical security feature.
- It must only work for authenticated users.
- Never allow users to specify a username.
- Require the current password again.
- Ask for the new password twice.
- Use generic error messages.
- Suspend the function after repeated failures.
- Notify users when their password changes, without revealing credentials.

Prevent Misuse of the Account Recovery Function:

The account recovery (forgot password) feature is dangerous because:

- It lets someone regain access without knowing the password
- If designed badly, it becomes the easiest way to hack accounts

So the goal is: Help real users recover access without helping attackers.

- Account recovery is a high-risk feature.
- Very secure apps use out-of-band recovery (calls, mail).
- Password hints must never be used.
- Best automated solution: Email a unique, time-limited, single-use reset link
- Old passwords should stay valid until changed.
- Security questions must:
    - Be system-defined
    - Be hard to guess
    - Be protected from brute force
- Never log users in or reveal passwords during recovery.
- Always require email confirmation for password reset.

Log, Monitor, and Notify:

- Log everything related to authentication (but never passwords).
- Protect logs because they contain sensitive information.
- Monitor logs in real time to detect brute-force and attacks.
- Alert admins when suspicious behavior occurs.
- Notify users out-of-band (email/SMS) for critical events.
- Notify users in-band (inside app) about login history and failed attempts.

--The End--