Chapter 8

Attacking Session Management:

Access controls decide what actions and data a logged-in user is allowed to access in a web application. Even when authentication and session handling are strong, weak or missing access controls can let attackers perform unauthorized actions, such as accessing sensitive data or admin features. These flaws are very common and conceptually simple—the app allows something it shouldn't—but they appear in many different forms, requiring different techniques to find and exploit.

Common Vulnerabilities:

Access controls manage what users can do and what data they can access in an application, using vertical controls (based on roles) and horizontal controls (based on ownership). When these controls fail, users may gain unauthorized abilities through vertical or horizontal privilege escalation. These flaws can combine and become especially dangerous, sometimes allowing attackers—even unauthenticated ones—to access sensitive data or administrative functionality.

Some common ones are:

- Completely Unprotected Functionality
- Identifier-Based Functions
- Multistage Functions
- Static Files
- Insecure Access Control Methods

Completely Unprotected Functionality:

Completely unprotected functionality occurs when sensitive features are accessible to anyone who knows the correct URL, with no real access checks in place. Hiding links or using obscure URLs does not provide true security, because URLs are easily discovered through logs, browser history, or client-side code like JavaScript. Without proper server-side access controls, attackers can find and directly access administrative or sensitive functions, leading to serious security risks.

Identifier-Based Functions:

Identifier-based vulnerabilities occur when an application uses client-supplied IDs to access resources or functions without properly checking authorization. Even if identifiers are hard to guess, they are not secrets and can often be discovered through logs or other disclosures. When access controls rely only on identifiers instead of verifying user permissions, attackers can view other users' data or trigger restricted functionality.

Multistage Functions:

Multistage function vulnerabilities occur when applications check authorization only in early steps and assume later steps are safe. Attackers can bypass initial checks by directly accessing or modifying later requests, allowing them to perform unauthorized actions such as creating admin accounts or transferring funds from other users. Proper access control and data validation must be enforced at every stage of a multistep process.

Static Files:

Static file vulnerabilities occur when sensitive resources are stored as directly accessible files that cannot enforce access controls. Because static files do not execute server-side logic, anyone who knows or guesses the URL can retrieve them. Predictable naming schemes make this especially dangerous, allowing attackers to access or mass-download protected content such as paid ebooks, financial documents, or internal logs.

Insecure Access Control Methods:

Insecure access control methods occur when applications rely on client-supplied data—such as URL parameters, hidden fields, cookies, or the HTTP Referer header—to decide what a user is allowed to do. Because users can modify these values, attackers can easily bypass restrictions and gain unauthorized access. Access control decisions must always be enforced on the server using trusted data, not information provided by the client.

Attacking Access Controls:

App.py:

app.py > ...

```python
from flask import Flask, request, session, redirect, url_for, send_from_directory

app = Flask(__name__)
app.secret_key = "supersecret"

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

documents = {
    1: {"owner": 2, "content": "Alice's private document"},
    2: {"owner": 3, "content": "Bob's private document"},
}

# ---------- AUTH ----------
@app.route("/", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        for uid, user in users.items():
            if user["username"] == request.form["username"] and user["password"] == request.form["password"]:
                session["user_id"] = uid
                session["role"] = user["role"]
                return redirect("/home")
        return "Invalid login"
    return """
        <form method="POST">
            Username: <input name="username"><br>
            Password: <input name="password"><br>
            <input type="submit">
        </form>
    """

@app.route("/home")
def home():
```

app.py > ...

```python
@app.route("/home")
def home():
    if "user_id" not in session:
        return redirect("/")
    return f"""
        <h1>Home</h1>
        <p>User ID: {session['user_id']}</p>
        <p>Role: {session['role']}</p>
        <a href="/profile">My Profile</a><br>
        <a href="/documents">My Documents</a><br>
        <a href="/admin">Admin Panel</a><br>
    """

# ---------- IDOR ----------
@app.route("/profile")
def profile():
    uid = request.args.get("id", session["user_id"])
    uid = int(uid)
    user = users.get(uid)
    return f"""
        <h1>Profile</h1>
        Username: {user['username']}<br>
        Password: {user['password']}<br>
        Role: {user['role']}
    """

# ---------- HORIZONTAL PRIV ESC ----------
@app.route("/documents")
def list_documents():
    return """
        <h1>Documents</h1>
        <a href="/document?id=1">Document 1</a><br>
        <a href="/document?id=2">Document 2</a>
    """

@app.route("/document")
def document():
```

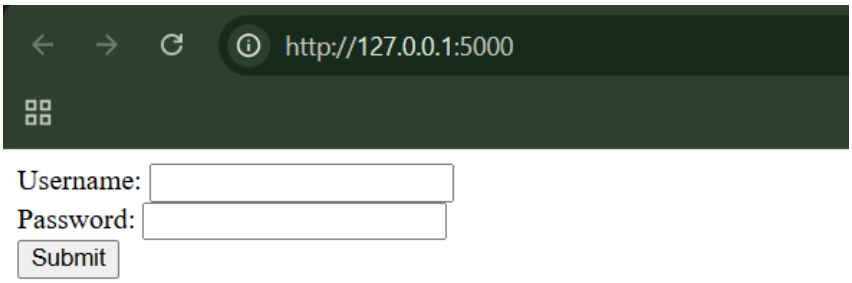
app.py > ...

```python
@app.route("/document")
def document():
    doc_id = int(request.args.get("id"))
    return documents[doc_id]["content"]

# ---------- VERTICAL PRIV ESC ----------
@app.route("/admin")
def admin_panel():
    # BAD: Access control via URL parameter
    if request.args.get("admin") == "true":
        return "<h1>Admin Panel</h1><p>All users:</p>" + str(users)
    return "Access denied"

# ---------- REFERER-BASED CONTROL ----------
@app.route("/admin/delete_user")
def delete_user():
    if request.headers.get("Referer") != "http://localhost:5000/admin":
        return "Access denied"
    uid = int(request.args.get("id"))
    users.pop(uid, None)
    return "User deleted"

# ---------- UNPROTECTED STATIC FILE ----------
@app.route("/files/<path:filename>")
def files(filename):
    return send_from_directory("protected_files", filename)

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

Terminal:

```
PS E:\Tounderstnad\Vulnerable Access Control Lab> python app.py
 >>
  * Serving Flask app 'app'
  * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
  * Running on http://127.0.0.1:5000
Press CTRL+C to quit
  * Restarting with stat
  * Debugger is active!
  * Debugger PIN: 998-674-252
```

Browser:

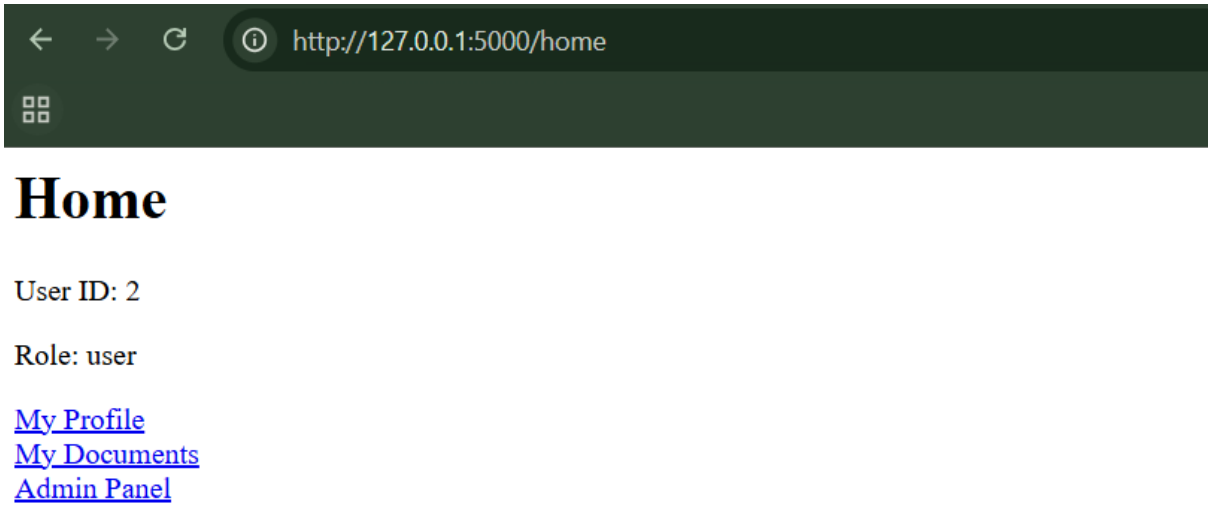


Step1: (Goal: to check for Horizontal Privilege Escalation). Login as Alice.

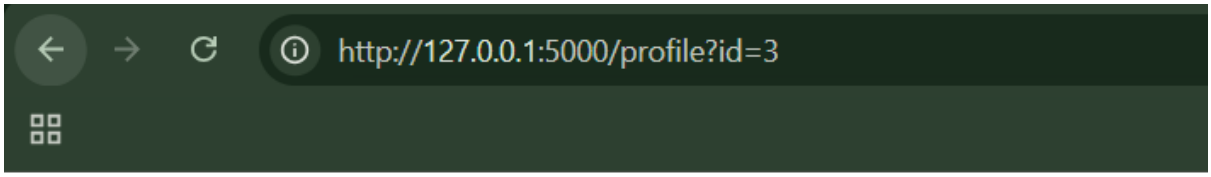
Visit:

- /profile?id=3
- /document?id=2

Normal alice login:



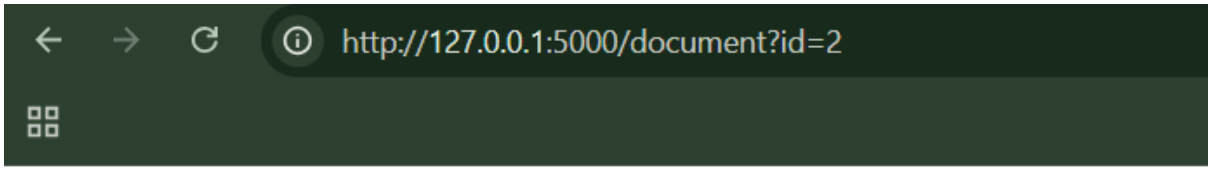
When I did this: http://127.0.0.1:5000/profile?id=3



Profile

Username: bob
Password: bob123
Role: user

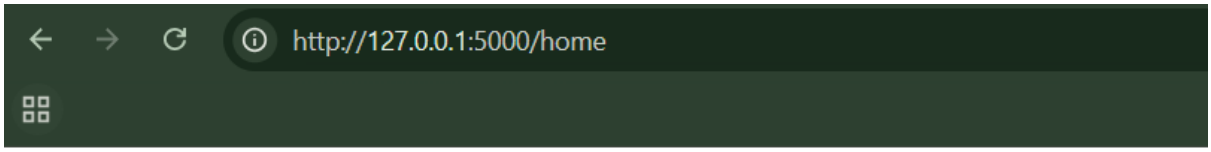
When I did this: http://127.0.0.1:5000/document?id=2



Bob's private document

Clearly, we can see Bob's data.

Step1: (Goal: to check for Vertical Privilege Escalation). Login as Alice.



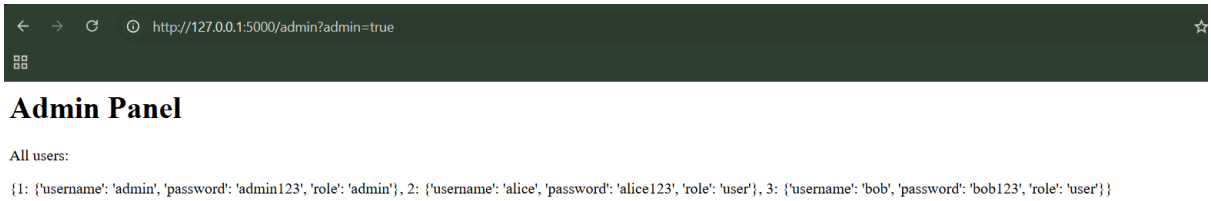
Home

User ID: 2

Role: user

My Profile
My Documents
Admin Panel

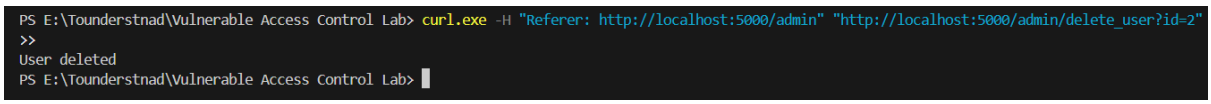
Now, change to: http://127.0.0.1:5000/admin?admin=true



Clearly, Admin panel appears, thus vertical privilege escalation is possible.

Step3: (Goal: Deletes a user without being admin). Just use this:

curl.exe -H "Referer: http://localhost:5000/admin" "http://localhost:5000/admin/delete_user?id=2"



Step4: (Goal: to reach the unprotected resources.) Visit directly: /files/admin_report.txt

Securing Access Controls:

Secure access control means assuming users are hostile, checking every request centrally using server-side session data, never trusting the client, and enforcing permissions consistently everywhere — without exception.

Keep in mind:

- Assume the attacker knows all URLs and all IDs. Security must come from access checks, not secrecy.
- Never trust anything that comes from the user to decide permissions.
- Every page must protect itself. Never rely on the path the user took.
- Re-validate everything when it comes back to the server.

Best-practice way to implement access controls:

1. Define access rules clearly
2. Base access decisions on the session
3. Use one central access control system (A single security guard checking every request.)
4. Check every request, no exceptions
5. Force developers to implement access checks
6. Extra protection for sensitive features

Securing static files (PDFs, images, downloads):

1. Serve files through a protected page
2. Protect files at the server level

Extra protection for critical actions:

Add more safeguards:

- Re-authentication (ask for password again)
- Dual authorization (two people must approve)

Logging and monitoring:

Every sensitive action should be logged:

- Viewing sensitive data
- Changing settings
- Admin actions

Why centralized access control is better?

Using a central access control system:

Benefits

- Clearer: Developers understand rules faster
- Easier to maintain: Change once, apply everywhere
- More flexible: New rules fit into existing system
- Fewer bugs: Less chance of missing or broken checks

A Multi-Layered Privilege Model:

What is a Multi-Layered Privilege Model?

A security approach where access control is applied at multiple layers, not just the web app.

Why it's important:

- Prevents total compromise if one layer fails
- Strong example of Defense-in-Depth

Layers involved:

1. Web application
2. Application server
3. Database
4. Operating system

Key techniques used:

- Programmatic control → code-based permission checks
- Discretionary Access Control (DAC) → admins grant/revoke access
- Role-Based Access Control (RBAC) → users assigned roles
- Declarative control → enforced by database or server, not code

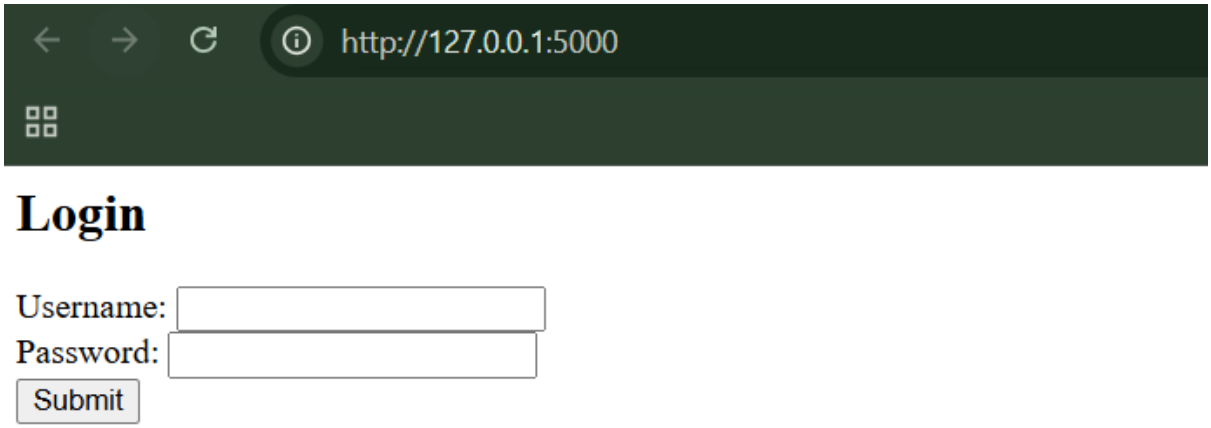
Core principle: Give only the minimum privileges required, at every layer.

Exploring vulnerable local website again:

vuln_app.py:

setup_db.py:

Browser:

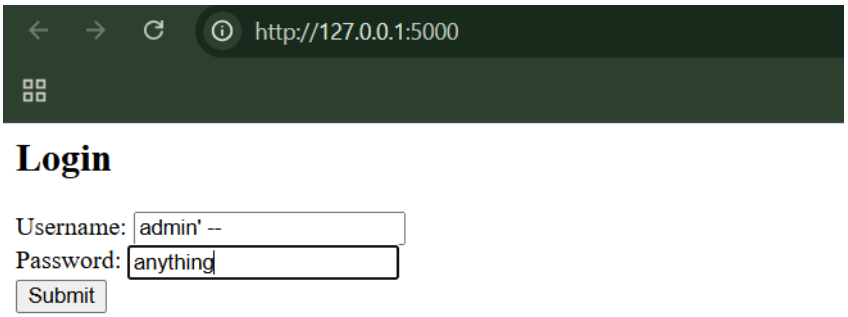


Step1: (Goal: SQL Injection (Login Bypass)).

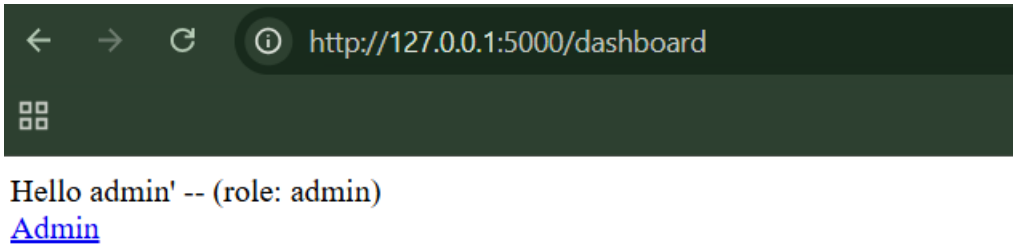
Try logging in with:

Username: admin' --

Password: anything

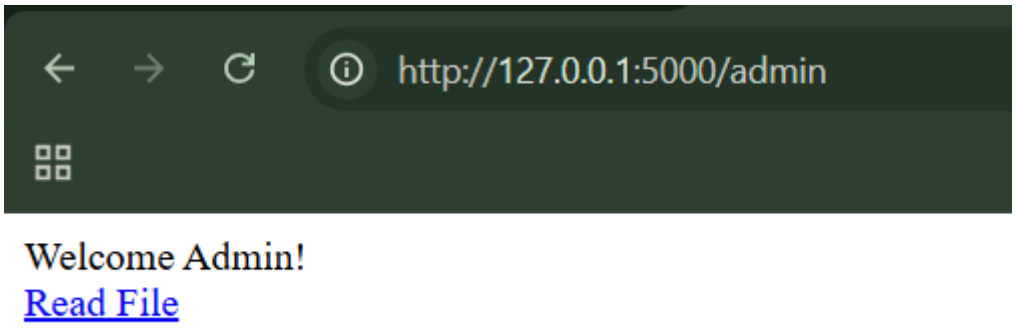


When clicked on the “Submit” button:



We managed to log in without knowing the password.

Step2: (Goal: Broken Access Control testing.) Login as a normal user. Manually access: /admin



--The End--