Web Hacking – Exercise 01

A01:2025 Broken Access Control:
Audit and Harden an Invoicing API

Duration	90 minutes
Organisation	Teams of two
Tools	Python 3.12 or Docker, an editor, Burp Suite Community or OWASP ZAP
Material	Folder `app/` (source code, front end, test skeleton)

Learning Objectives

At the end of this exercise you are able to:

- write the access control policy of an application as an explicit **access control matrix**, starting from business requirements;
- **review code** to find where the policy is enforced (Policy Enforcement Points) and where it is missing;
- **verify** the actual behaviour of an application with several accounts and an intercepting proxy;
- **fix** access control weaknesses with the principles from the lecture: deny by default, server-side enforcement, ownership checks, allow lists;
- **prevent regressions** with automated access control tests.

Rules of engagement. InvoiceDesk is a training application that contains intentional weaknesses. Run it **on your own machine only**, bound to `127.0.0.1`. Never test any system you are not explicitly authorised to test: the techniques of this course applied to third-party systems are illegal (Swiss Criminal Code, art. 143^{bis}).

1 Setup (before the lesson, 5 minutes)

Option A – Python virtual environment

```
cd app
python3 -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
flask --app app:create_app run          # http://127.0.0.1:5000
pytest -v                               # runs the test skeleton
```

Option B – Docker

```
cd app
docker build -t invoicedesk .
docker run --rm -p 127.0.0.1:5000:5000 invoicedesk
```

Configure your browser to use Burp or ZAP as a proxy and open `http://127.0.0.1:5000`. The database is re-created at every start, so you can restart the application to get back to the initial state.

macOS: port 5000 is often taken by the *AirPlay Receiver* (it answers `403` to every request, which is easy to mistake for the application). Either switch it off (System Settings → General → AirDrop & Handoff) or use another port, e.g. `flask ... run --port 5001` / `-p 127.0.0.1:5001:5000`, and adapt the URLs below.

2 Scenario

InvoiceDesk is a small invoicing service used by a hosting company. Customers log in to see and pay their invoices; the back office (clerks) follows the payments; an administrator manages the user accounts. The application is about to go live. Your team is asked to perform a **security review focused on access control**, to fix the weaknesses and to add tests before the release.

Test Accounts

User	Password	Role	Invoices
`alice`	`alice123`	customer	1001, 1002
`bob`	`bob123`	customer	1003, 1004
`carol`	`carol123`	clerk	–
`dave`	`dave123`	admin	–

Business Requirements (the Policy)

R1 Anonymous visitors can only see the login page and log in.

R2 A **customer** can list and view *their own* invoices and pay *their own* unpaid invoices.

R3 A customer can change *only* their display name and e-mail address.

R4 A **clerk** can list and view *all* invoices and export them as CSV. A clerk cannot pay invoices on behalf of customers.

R5 An **administrator** has all the rights of a clerk and can additionally list and delete user accounts.

R6 The identity and the role of a user are determined *only* by the server (server-side session and user database).

R7 The API never returns secrets (password hashes, keys) to the client.

R8 Status codes: `401` when not logged in, `403` when the role is insufficient, `404` when the object does not exist *or* is not visible to the user (do not reveal the existence of other users' objects).

API Overview

Method	Path	Description
POST	`/api/login`	Log in (`{"username":..,"password":..}`)
POST	`/api/logout`	Log out
GET	`/api/me`	Profile of the current user
PATCH	`/api/me`	Update the profile (JSON body)
GET	`/api/invoices`	List invoices
GET	`/api/invoices/{id}`	Invoice details
POST	`/api/invoices/{id}/pay`	Pay an invoice
GET	`/api/reports/export`	Export all invoices (CSV)
GET	`/api/admin/users`	List users
DELETE	`/api/admin/users/{id}`	Delete a user

3 Tasks

Task 1 – Write the Access Control Matrix (15 min)

Before reading the code, translate the requirements R1–R8 into an access control matrix: one row per endpoint (and per kind of object where it matters: *own* vs. *other's* invoice), one column per subject (anonymous, customer, clerk, admin). Each cell contains the **expected** status code.

Request	anonymous	customer	clerk	admin
`GET /api/invoices/{own id}`	401	200	n/a	n/a
`GET /api/invoices/{other id}`	...	...	...	...
...				

Question: for which requests is the role *not* sufficient to decide, so that the object itself must be taken into account?

Task 2 – Code Review (20 min)

Read `app/app.py` and `app/static/index.html`.

(a) Locate every **Policy Enforcement Point**: where is authentication checked, where is the role checked, where is ownership checked?

(b) Is the central rule table (`ACCESS_RULES`) an *allow list* or a *block list*? What happens with an endpoint that matches no rule?

(c) Where does the application get the role of the current user from? Who controls this value?

(d) Which fields can be modified through `PATCH /api/me`? Which fields are returned by `GET /api/me`?

(e) For each endpoint, mark it as *correct*, *missing check* or *check in the wrong place*. Beware: not every endpoint is vulnerable – false positives cost credibility in a real audit.

Task 3 – Dynamic Verification (15 min)

Using the browser, your proxy and the four accounts, **verify each suspicion** from Task 2 against *your local instance* and compare the observed status code with your matrix. A useful technique (see the slide *Testing Access Control*): capture a request made by a high-privilege account and replay it with the session of a low-privilege account (Burp Repeater, or the *Autorize* extension).

Record each deviation in a findings table:

ID	Endpoint / location	Concept	CWE	Severity	Observed vs. expected
F1	...	...	...	...	...

Concept refers to the five concepts of the lecture (deny by default, horizontal/vertical escalation, object/property level authorization, trust boundary, confused deputy). Estimate the severity (low / medium / high / critical) from the number of reachable objects and their sensitivity.

Task 4 – Fix the Application (25 min)

Fix every finding **in the server code**. Your fixes must follow the principles of the lecture:

- **deny by default**: a new endpoint added later must be closed until a rule explicitly opens it;
- **one central mechanism** for authentication and role checks;

- **ownership** enforced in the data access (scoped queries), not by filtering afterwards;
- **allow lists** for writable and readable properties;
- **no authorization decision** based on client-controlled data.

Hiding a button in `index.html` is *not* a fix. The front end must keep working for the four accounts.

Task 5 – Access Control Tests (15 min)

Turn your matrix into automated tests in `app/tests/test_access_control.py` (a skeleton with two rows is provided; the fixture `client_as(user)` returns a logged-in test client). Add dedicated tests where a status code is not enough (request bodies, response contents, tampered cookies).

Your test suite must **fail on the original code** and **pass on your fixed code**. Check this with `git stash` or a copy of the original folder.

4 Wrap-up and Self-Check

This exercise is **not graded** and there is **nothing to hand in**. Its purpose is practice: keep your matrix, findings, fixes and tests for yourself, they are a good basis to prepare the exam. Before the debriefing, check your work with the following questions:

- **Matrix:** does it contain a cell for every endpoint and every subject, including *own* vs. *other's* objects?
- **Findings:** is each one justified (concept, CWE, severity) and confirmed on your local instance? Did you avoid false positives?
- **Fixes:** are they server-side, centralised and deny by default? Does the front end still work for the four accounts?
- **Tests:** do they cover the matrix and every finding, and do they fail on the original code and pass on your fixed code?
- **Reflection:** which real-world case of the lecture (Capital One, First American, Optus, USPS, Dell, Power Apps. . .) is closest to one of your findings, and which single measure would have prevented it? We discuss this question together at the end of the lesson.

References

- Slides of chapter 01, in particular *Testing Access Control* and *Centralised, Declarative Access Control*
- OWASP Authorization Cheat Sheet
  `https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html`
- OWASP API Security Top 10 (2023) – API1, API3, API5
  `https://owasp.org/API-Security/`