Web Hacking – Exercise 02

A02:2025 Security Misconfiguration:
Harden the Deployment of a Service

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

Learning Objectives

At the end of this exercise you are able to:

- tell apart a **code** bug from a **configuration** bug, and recognise that the same code can be secure or insecure depending on how it is deployed;
- **find misconfigurations** across the six themes of the lecture: attack surface, secrets in configuration, debug/error handling, browser security directives, insecure parser defaults (XXE), infrastructure;
- **verify** them against your local instance with a browser and `curl`;
- **harden** the application: reduce the attack surface, move secrets out of the code, fail safely, send security directives, fix insecure defaults;
- **prevent regressions** with automated checks.

Rules of engagement. ConfigBox is a training application that is deployed with intentional misconfigurations. Run it **on your own machine only**, bound to `127.0.0.1`. The XXE task reads files *on your own computer*; do not point it at anyone else's system. 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 --debug       # http://127.0.0.1:5000
pytest -v                                     # runs the test skeleton
```

Option B – Docker

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

The application keeps its data in memory, so restarting it gives you back the initial state. The team runs it exactly as shown above – in particular `--debug` is on, because "it is handy".

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

ConfigBox is a small internal service where staff store and share text snippets. The code was written carefully and reviewed; the application passed a first functional test. It is now deployed, and the operations team runs it the same way the developers did on their laptops. A new security engineer (you) is asked to perform a **configuration review** before the service is opened to the whole company, and to harden it.

Test Accounts

User	Password	Note
`alice`	`alice123`	a normal staff account
`admin`	`?`	created "for the demo" – find the password

API Overview

Method	Path	Description
GET	`/`	Front end (snippet list)
POST	`/api/login`	Log in (`{"username":..,"password":..}`)
POST	`/api/logout`	Log out
GET	`/api/snippets`	List snippets
POST	`/api/snippets`	Add a snippet (login required)
POST	`/api/import`	Import snippets from the legacy XML format
GET	`/admin/config`	Show the configuration (needs a token)
GET	`/debug/routes`	List every route (left from development)

3 Tasks

Task 1 – Map the Attack Surface (10 min)

Before fixing anything, list what the running application exposes.

(a) Enumerate the endpoints that answer *without* authentication. Which ones should not exist in production at all?

(b) Look at how the application is **started** (the `flask` command and the `Dockerfile`). Which run-time flag changes what an error reveals to the client? On which interface does the container listen, and as which user?

(c) For each of the six lecture themes (hardening, secrets, debug/errors, browser directives, XXE, infrastructure), write one line: *is this theme present in ConfigBox, and where do I expect to look?*

Task 2 – Find and Verify the Misconfigurations (25 min)

Confirm each suspicion against *your local instance*. Suggested probes (adapt the host/port if needed):

```
# What does the server tell the browser? (headers, cookie flags)
curl -i http://127.0.0.1:5000/

# What happens on an unexpected error, with --debug on?
# (log in first: the endpoint needs a session; -c/-b keep the cookie)
curl -c jar -X POST http://127.0.0.1:5000/api/login \
     -H 'Content-Type: application/json' \
     -d '{"username":"alice","password":"alice123"}'
curl -i -b jar -X POST http://127.0.0.1:5000/api/snippets \
     -H 'Content-Type: application/json' -d '{"title":"x"}'

# The legacy XML import -- try a harmless file on YOUR machine first:
echo 'local-marker' > /tmp/marker.txt
curl -s -X POST http://127.0.0.1:5000/api/import --data-binary @- <<'XML'
<?xml version="1.0"?>
<!DOCTYPE s [ <!ENTITY e SYSTEM "file:///tmp/marker.txt"> ]>
<snippets><snippet><title>t</title><body>&e;</body></snippet></snippets>
XML
```

Record each finding in a table:

ID	Location	Theme	CWE	Severity	Observed vs. expected
F1	...	...	...	...	...

Theme is one of the six lecture concepts. Estimate the severity from what an attacker gains (information, a secret, a file, code execution) and from how easily the issue is reached. Beware of false positives: not every default is a vulnerability – justify each finding.

Task 3 – The XXE in Detail (10 min)

For the XML import specifically, answer:

(a) *Why* does the import resolve your entity? Which parser option is responsible (read `app.py`)?

(b) Besides reading a local file, what else could the same feature be abused for in a real deployment (think of the `http://` scheme and the *Capital One* slide)?

(c) What is the smallest change that stops *all* of these abuses at once, without breaking the legitimate import of `<snippets><snippet>...</snippet></snippets>`?

Task 4 – Harden the Application (30 min)

Fix every finding. Follow the principles of the lecture; a change in the front end is *not* a fix. In order of priority:

- **Reduce the attack surface**: remove development-only endpoints and the default `admin` account; the application must still work for `alice`.
- **Get secrets out of the code**: read `SECRET_KEY` and the internal token from the environment (fail to start if they are missing); make sure no endpoint returns them.
- **Fail safely**: turn debug off for the served app, add a central error handler that returns a **generic** message (details to the logs only), and reject malformed input with `400` instead of crashing.
- **Send security directives**: add the response headers
  - `Content-Security-Policy`, `X-Content-Type-Options`,
  - `X-Frame-Options`, `Referrer-Policy`, `Strict-Transport-Security`,

and set the session cookie flags `HttpOnly`, `Secure` and `SameSite`.

- **Fix the parser default**: disable DTDs / external entities / network access in the XML parser.
- **The image**: review the `Dockerfile` – run as a non-root user, do not bake secrets into the image, and do not ship the development server with `--debug`.

Task 5 – Lock it Down with Tests (15 min)

Turn your findings into automated checks in `app/tests/test_hardening.py` (a skeleton with two tests is provided; the fixture `client` returns a test client). Write at least one test per finding: security headers present, cookie flags set, XXE no longer reads a file, the default `admin` account is gone, a crash returns a generic error, the development endpoints are removed.

Your suite must **fail on the original code** and **pass on your hardened 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 findings, fixes and tests for yourself, they are a good basis to prepare the exam. Before the debriefing, check your work:

- **Surface:** did you cover all six themes, and remove what should not be in production at all?
- **Secrets:** are `SECRET_KEY` and the token out of the code, and impossible to read through any endpoint? If they leaked once, what must you additionally do?
- **Fail safe:** does an unexpected error now reveal nothing? Is debug off in the way the app is actually started?
- **Directives:** are the headers and cookie flags present on every response, verified with `curl -i`?
- **XXE:** does the import still work for legitimate XML but refuse entities, DTDs and network access?
- **Reflection:** which real-world case of the lecture (Capital One, Codecov, Tesla, the Meow database wipes, the `.git` kill chain…) is closest to one of your findings, and which single setting would have prevented it? We discuss this together at the end of the lesson.

References

- Slides of chapter 02, in particular *Hardening*, *Security Headers and Cookie Flags*, *XML External Entities* and *Configuration as Code, Checked Automatically*
- OWASP Cheat Sheets – Configuration, XML External Entity Prevention, HTTP Security Response Headers
  `https://cheatsheetseries.owasp.org/`
- OWASP Top 10:2025 – A02 Security Misconfiguration
  `https://top10.owasp.org/2025/A02_2025-Security_Misconfiguration/`
- Mozilla Observatory, `securityheaders.com` (header checks)