Web Hacking – Exercise 05

A05:2025 Injection:
Four Interpreters, One Bug Class

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

Learning Objectives

At the end of this exercise you are able to:

- recognise the single root cause behind injection – untrusted *data* parsed as *code* – across four different interpreters;
- **exploit** a SQL injection (authentication-independent data theft via `UNION`), an OS command injection, a stored XSS and a server-side template injection, each against your local instance;
- **fix** each one with the *structural* defence of the lecture: parameterized queries, no shell / argument lists, context-aware output encoding, and never evaluating user input as a template;
- **tell apart** a real vulnerability from a safe pattern (not every query here is vulnerable);
- **prevent regressions** with automated tests that carry classic payloads.

Rules of engagement. HelpDesk is a training application with intentional injection flaws. Run it **on your own machine only**, bound to `127.0.0.1`. The command-injection task runs commands *on your own computer*; keep the payloads harmless (`echo`, `id`). Never test any system you are not explicitly authorised to test (Swiss Criminal Code, art. 143bis).

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 helpdesk .
docker run --rm -p 127.0.0.1:5000:5000 helpdesk
```

The SQLite database is re-created at every start. Open `http://127.0.0.1:5000` in a browser for the XSS task; use `curl` for the API tasks.

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

HelpDesk is a small internal support-ticket tool. Agents search tickets, read and comment on them; an admin has a small "diagnostics" page and a signature preview. The tool was written quickly by gluing strings together. Your team is asked to **find the injection flaws, exploit them to prove the impact, and fix them**.

Test Accounts

User	Password	Role
`alice`	`alice123`	agent
`admin`	`admin123`	admin

API / Page Overview

Method	Path	Description
GET	`/`	Ticket list (HTML)
POST	`/api/login`	Log in
GET	`/api/tickets?q=...`	Search tickets by subject
GET	`/ticket/{id}`	Ticket detail page with comments (HTML)
POST	`/api/tickets/{id}/comments`	Add a comment
GET	`/api/diagnostics?host=...`	Ping a host (admin only)
GET	`/api/preview?name=...`	Render an e-mail signature

3 Tasks

Task 1 – Find the Interpreters (10 min)

Read `app.py`. For each endpoint, note *which interpreter* the user input eventually reaches (SQL database, the shell, the browser's HTML/JS, the template engine) and *how* the input gets there (parameter, concatenation, `|safe`, `render_template_string`...). Mark each as *probably safe* or *probably injectable*. **Careful:** at least one endpoint that looks suspicious is actually safe – say which and why.

Task 2 – SQL Injection (20 min)

Target `GET /api/tickets?q=...`. Payloads contain spaces, quotes and braces: let curl URL-encode them with `-G --data-urlencode` (pasting them raw into the URL fails silently).

```
URL=http://127.0.0.1:5000/api/tickets
# a normal search
curl -G $URL --data-urlencode "q=Invoice"
# make the condition always true
curl -G $URL --data-urlencode "q=' OR '1'='1"
# read another table with UNION (match the column count!)
curl -G $URL --data-urlencode \
     "q=' UNION SELECT username, password_hash, role FROM users -- "
```

(a) Why does `UNION` work here, and how did you find the number of columns to select? What sensitive data did you reach?

(b) This endpoint needs no login. Which real-world breach from the lecture does this remind you of?

(c) What is the *one* change that fixes it, and why is "escaping quotes" or a blocklist not a reliable fix?

Task 3 – OS Command and Template Injection (20 min)

(a) Command injection – log in as `admin`, then:

```
curl -c jar -X POST http://127.0.0.1:5000/api/login \
     -H 'Content-Type: application/json' -d '{"username":"admin","password":"admin123"}'
curl -b jar -G http://127.0.0.1:5000/api/diagnostics \
     --data-urlencode "host=127.0.0.1; echo INJECTED"
```

Why does `echo INJECTED` run? Which shell metacharacters are at play? What would `$(id)` show?

(b) Template injection – try the signature preview:

```
curl -G http://127.0.0.1:5000/api/preview --data-urlencode "name={{7*7}}"
```

If the response contains `49`, the input was *evaluated*. Explain the difference between building the template *source* from user input and passing the user input *as data* to a fixed template.

Task 4 – Stored XSS (15 min)

The ticket page has no comment form; comments are posted through the API. Add a comment containing a harmless script, then open the ticket in the browser (`http://127.0.0.1:5000/ticket/1`):

```
curl -X POST http://127.0.0.1:5000/api/tickets/1/comments \
     -H 'Content-Type: application/json' \
     -d '{"body":"<script>alert(document.domain)</script>"}'
```

(a) Does your script run? Why – what does the template do with the comment body (look for `|safe`)?

(b) Why is *stored* XSS more dangerous than *reflected* XSS here?

(c) What is the correct fix, and what does a `Content-Security-Policy` add on top of it?

Task 5 – Fix Everything and Test It (20 min)

Fix every flaw in the server code, using the structural defences:

- **SQL**: parameterized query (bind the search term as data).
- **Shell**: do not use a shell; pass arguments as a list; validate the host against a strict allow-list pattern.
- **Browser**: let the template auto-escape (remove `|safe` on untrusted content); add a `Content-Security-Policy` as defence in depth.
- **Template engine**: pass the name as a *variable*, never build the template source from it.

Then turn each exploit into a test in `app/tests/test_injection.py` (a skeleton and the fixtures `client` and `admin` are provided). 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 payloads, fixes and tests to prepare the exam. Before the debriefing:

- **Coverage:** did you fix all four interpreters, and correctly leave the safe endpoint untouched?
- **Structural fix:** is each fix about *separating data from code*, not about filtering bad characters?

- **XSS:** is the comment now shown as text, and does the CSP block inline script?
- **Reflection:** which real-world case of the lecture (TalkTalk, the Samy worm, command-injection botnets, SSTI-to-RCE…) matches each finding, and which single control would have prevented it? We discuss this together at the end.

References

- Slides of chapter 05, in particular *Data vs Code*, *SQL Injection*, *OS Command Injection*, *Cross-site Scripting* and *The Defense, Summarised*
- OWASP Cheat Sheets – SQL Injection Prevention, Query Parameterization, Cross-Site Scripting Prevention, OS Command Injection Defense
  `https://cheatsheetseries.owasp.org/`
- OWASP Top 10:2025 – A05 Injection
  `https://top10.owasp.org/2025/A05_2025-Injection/`
- PortSwigger Web Security Academy – SQL injection, XSS, SSTI, OS command injection
  `https://portswigger.net/web-security`