Module 5 - Software Assurance + Secure SQL (SQLi Defense) Assignment

jhu.instructure.com/courses/112002/assignments/1233755

This assignment gives you hands-on practice with **software assurance** workflows: input validation, static analysis, dependency analysis, virtual environments, supply-chain scanning, and least-privilege database configuration. Your goal is to harden a Flask + PostgreSQL web app against **SQL injection** and ship a reproducible, security-checked codebase.

Skills: Static Code Analysis, Software Assurance, SQL Injection Defenses, Virtual Environments, Dependency Analysis, CI Security

Assignment Overview

By the end of the assignment, you will:

- Copy `module_4/` into a new `module_5/`
- Achieve **10/10** linting with **Pylint**
- Refactor SQL to use **safe psycopg composition** (no raw string-built SQL)
- Enforce **LIMIT** for every query
- Generate a Python dependency graph (`dependency.svg`) using **pydeps + Graphviz**
- Ensure the project installs and runs in a clean environment using:
    - **pip + venv** (required)
    - **uv** (required as an alternate install path)
- Add a `setup.py` (and explain why) to make your project installable
- Store DB credentials in environment variables and use a **least-privilege** DB user
- Scan dependencies with **Snyk** and provide proof
- Add **GitHub Actions CI** that enforces lint/tests/dependency graph
- (Optional/Extra Credit) runs Snyk

Step 0: Setup (Folder + Environment)

1. Copy your `module_4/` folder into a new folder called `module_5/`.
2. Create and activate a virtual environment inside `module_5/`.
3. Install dependencies from `requirements.txt`.
4. Confirm the Flask web app (and analysis page + tests) still run.

(You will update `requirements.txt` later.)

Step 1: Pylint (10/10 Required)

1. Install Pylint in your environment.
2. Run Pylint on **all Python files** inside `module_5/src`. Do not lint any code outside of src.
3. Fix issues until your output includes:

```
Your code has been rated at 10.00/10
```

Requirement: No Pylint error messages or warnings at final submission.

Tip (recommended): Include a `README.md` section showing exactly how you ran Pylint (the command you used).

Step 2: SQL Injection Defenses (Required Refactor)

Update **every SQL query** in `module_5/` that uses user input.

Required changes

- Do **not** build SQL using f-strings, `+` concatenation, or `.format()` with raw SQL text.
- Use **psycopg SQL composition** (`sql.SQL`) for dynamic SQL.
- Separate:
    - **SQL statement construction** (composed SQL object)
    - from **execution** and **parameters** (`cursor.execute(stmt, params)`).

- Convert variable components correctly:
    - table/column names: `sql.Identifier(...)`
    - values: parameter placeholders (`%s` + params list) or `sql.Placeholder(...)`
- **Every query must have an inherent LIMIT**, and your code must enforce a maximum allowed limit (example: clamp to 1–100).

Hint: psycopg SQL composition patterns (SQL/Identifier/Placeholder) are the expected approach.

What we're looking for (high-level)

- Dynamic SQL parts (like column names) are **quoted safely via Identifier**
- User values never get inserted into SQL text: they go through **parameter binding**
- Your endpoints handle malicious input safely (no data leak, no crash, no "everything returned")

Step 3: Database Hardening (Least Privilege Required)

Security is not just query syntax. Your database account should have the minimum privileges needed.

Required changes

1. **No hard-coded DB credentials in code.**
   Your app must read DB connection values from environment variables, for example:

   `DB_HOST`, `DB_PORT`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`

2. Include a file named **`.env.example`** containing the variable names (with placeholder values). Do **not** commit real secrets. Ensure `.env` is in `.gitignore`.
3. Create (or configure) a **least-privilege DB user**:
    - Not a superuser
    - No `DROP`, `ALTER`, or owner-level permissions
    - Only the permissions your app needs (for many apps: `SELECT` on specific tables; possibly `INSERT/UPDATE` if your app truly writes)

4. In your PDF, include:
    - what permissions you granted and why
    - a short snippet of the SQL you used (or a screenshot of privileges)

(If your app is read-only, your DB user should be read-only.)

Step 4: Python Dependency Graph (pydeps + Graphviz)

1. Install `pydeps`.
2. Install Graphviz (ensure the `dot` command is available).
3. Run pydeps to generate an SVG graph.

A standard command looks like this (pydeps supports `-o` and `--noshow`):

```
pydeps app.py --noshow -T svg -o dependency.svg
```

Deliverables for this step

- Save as `dependency.svg` in `module_5/`.
- In **5–7 sentences**, explain the key dependencies shown and what they do.

Your dependency graph will likely look something like (but not identical to) this:

Step 5: Reproducible Environment + Packaging (pip + uv + setup.py)

5A) requirements.txt (Required)

Update `requirements.txt` so a brand new environment can run your project from scratch.

Required

- The Flask web app (and analysis page) still runs
- `requirements.txt` includes everything needed at runtime
- `requirements.txt` also includes your tools: **pylint** and **pydeps**

5B) Add a setup.py (Required)

Add a `setup.py` in `module_5/` and include a short explanation in your PDF: **why packaging matters**.

Why do this?

- It makes your project **installable**, so imports behave consistently (local runs, tests, CI).
- It supports editable installs (`pip install -e .`), reducing "it works on my machine" path issues.
- Tools like uv can also extract requirements from `setup.py` when syncing environments.

5C) Fresh install instructions (Required: pip + uv)

Add a "Fresh Install" section to your `README.md` showing **both** install methods (pip and uv).

(Note: `uv pip sync` makes the environment match the requirements file exactly, which is great for reproducibility.)

Step 6: Snyk Dependency Scan (Required) + Extra Credit

Software assurance includes supply-chain checks.

Required (everyone)

1. Install Snyk CLI and authenticate.
2. Run: `snyk test`
3. Include `snyk-analysis.png` (screenshot of results).
4. If vulnerabilities are found, document them and patch/remove packages where applicable.

Snyk's CLI supports `snyk test` for open-source dependency scanning.

Extra Credit (+5 points): Snyk Code (SAST)

Run:

```
snyk code test
```

Include a screenshot (or output) as proof and briefly summarize what it found.

Step 7: CI Enforcement with GitHub Actions

Add a GitHub Actions workflow that runs on every push/PR and enforces “shift-left security”.

CI Requirements

Your workflow should now have 4 separate actions:

1. Run **Pylint** and fail if score is below 10 (use `--fail-under=10`)

2. Generate `dependency.svg` (and fail if missing) using pydeps + Graphviz

3. Run `snyk test` (at minimum, produce output; optionally fail on issues)

4. Continue to have your working **Pytest and fail if score is below 10**

Follow the same pathway that we used for Module 4.

Final Deliverables (Updated)

Submit the following:

Canvas

Zipped `module_5/` folder

Git repo

Push `module_5/` to the required repo location (course instructions)

Inside `module_5/`, include:

- 10/10 lint score evidence
- `dependency.svg`
- `snyk-analysis.png`
- `setup.py`

- A PDF explaining:
    - how to install/run via **pip** and **uv**
    - dependency graph summary (5–7 sentences)
    - your SQL injection defenses (what changed and why it's safe)
    - least-privilege DB configuration (what permissions and why)
- Requirements met for SQL: LIMIT enforced, statements/execution separated, safe composition/parameterization used
- GitHub Actions workflow file (`.github/workflows/ci.yml`) + screenshot of a successful run

Optional/Extra Credit

(+5) `snyk code test` evidence + short summary

Please remember to submit to both Canvas and commit to your GitHub (note we are still using public GitHub, and your Read the Docs must still be available / viable!

Please let us know if you have any questions via Teams or email!