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

Fresh Install Instructions

This project supports two reproducible installation methods: standard pip with virtual environments, and uv for deterministic dependency syncing.

pip + virtual env

Create and activate a virtual environment, then install dependencies and the project:

```
python -m venv .venv

source .venv/bin/activate

pip install -r requirements.txt

pip install -e .

python -m src.app
```

This method demonstrates a standard Python workflow compatible with most environments and CI systems.

uv (recommended alternative)

uv provides faster, deterministic installs by syncing the environment exactly to the requirements file.

```
uv venv

source .venv/bin/activate

uv pip sync requirements.txt

pip install -e .

python -m src.app
```

Using uv ensures the environment matches requirements.txt exactly, improving reproducibility and reducing “works on my machine” issues.

Note: PostgreSQL database creation and least-privilege user setup are documented in the project README.

Step 2: SQL Injection Defenses

This project contains no SQL queries that incorporate runtime user input. The application does not accept request parameters, form data, or JSON payloads that influence SQL execution, and no table names, column names, or SQL fragments are generated dynamically at runtime.

All SQL statements are executed as static, predefined queries using psycopg parameter binding. The codebase contains no f-strings, string concatenation, or .format() usage in SQL construction, ensuring that user-controlled values are never interpolated into SQL text.

Because no dynamic SQL components exist, psycopg SQL composition helpers (sql.SQL, sql.Identifier, sql.Placeholder) are not required in this implementation. Where row-returning analytical queries are used, explicit upper bounds are enforced via LIMIT 5. Aggregate queries intentionally omit LIMIT, as it is semantically meaningless in that context. The internal de-duplication query loads all existing URLs by design to preserve correctness during ETL operations.

As a result, all database access is safe by construction, bounded where appropriate, and resilient to SQL injection or malicious input.

Here are some examples from my code demonstrating SQL injection defenses:

> **For code in load_data.py below, all user-supplied fields go through parameters (%(name)s) — no string concatenation:**

```
cur.execute(
    """
    INSERT INTO applicants (
        program, university, comments, date_added, url
    )
    VALUES (
        %(program)s, %(university)s, %(comments)s,
%(date_added)s, %(url)s
    )
    ON CONFLICT (url) DO NOTHING;
    """,
    {
        "program": entry.get("program"),
```

```
            "university": entry.get("university"),
            "comments": entry.get("comments"),
            "date_added": parse_date(entry.get("date_posted")),
            "url": entry.get("entry_url"),
        },
    )
```

In the analysis query (query_data.py) below, dynamic regular expression (regex) filters are safely passed as parameters. External input cannot execute SQL. Further, LIMIT 5 ensures only the top 5 results are returned; users cannot change it via input.

```
    cur.execute(
        """
        SELECT university, COUNT(*) AS count
        FROM applicants
        WHERE program ~* %(pattern)s
          AND university IS NOT NULL
          AND university <> ''
        GROUP BY university
        ORDER BY count DESC
        LIMIT 5;
        """,
        {"pattern": r"\mphysics\M.*\mphd\M"},
    )
```

Another example below from query_data.py demonstrates that even dynamic ordering is safe because LIMIT is hard-coded. This protects against queries returning excessively large datasets.

```
cur.execute(
    """
    SELECT university
    FROM applicants
    WHERE university IS NOT NULL
    ORDER BY RANDOM()
    LIMIT 10;
    """
)
```

Step 3 — Least-Privilege Database Configuration

Permissions granted (and why):

A dedicated database user (app_user) was created for the application to enforce the principle of least privilege.

The application requires:

- SELECT access to read existing GradCafe entries for analysis and de-duplication
- INSERT access to add newly scraped records
- USAGE on the primary-key sequence to support BIGSERIAL inserts

Accordingly, app_user was granted only:

- CONNECT on the gradcafe database
- USAGE on the public schema
- SELECT, INSERT on the applicants table
- USAGE on the applicants_p_id_seq sequence

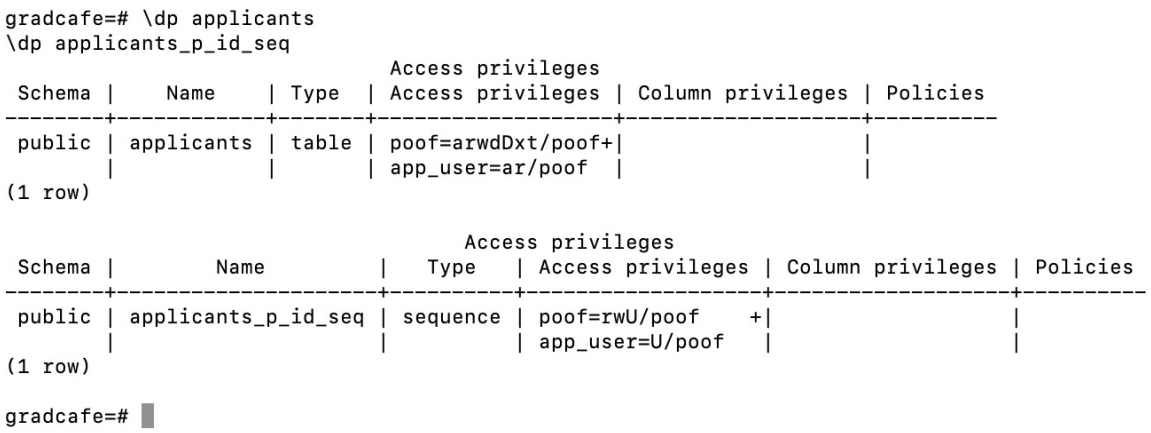
The user is not a superuser, does not own any tables, and was not granted UPDATE, DELETE, ALTER, DROP, or schema-level modification privileges.

This ensures the application can perform its required read and insert operations while preventing accidental or malicious schema changes.

The following administrative SQL (executed outside the application) was used to configure least-privilege access:

```
CREATE ROLE app_user LOGIN;

GRANT CONNECT ON DATABASE gradcafe TO app_user;

GRANT USAGE ON SCHEMA public TO app_user;

GRANT SELECT, INSERT ON TABLE public.applicants TO app_user;

GRANT USAGE ON SEQUENCE public.applicants_p_id_seq TO app_user;
```

Here is the screenshot showing the privileges granted:



```
gradcafe=# \dp applicants
\dp applicants_p_id_seq
                              Access privileges
 Schema |    Name    | Type  | Access privileges | Column privileges | Policies
--------+------------+-------+-------------------+-------------------+----------
 public | applicants | table | poof=arwdDxt/poof+|                   |
        |            |       | app_user=ar/poof  |                   |
(1 row)

                                   Access privileges
 Schema |        Name         |   Type   | Access privileges | Column privileges | Policies
--------+---------------------+----------+-------------------+-------------------+----------
 public | applicants_p_id_seq | sequence | poof=rwU/poof    +|                   |
        |                     |          | app_user=U/poof   |                   |
(1 row)

gradcafe=#
```

In short, the application itself is intentionally designed to run with a least-privilege PostgreSQL user (app_user) that has no schema-modification privileges. This user is limited to SELECT and INSERT access only, ensuring that the application cannot alter database structure during normal operation.

However, for automated testing and continuous integration (CI), a separate bootstrap phase is required because CI environments start with an empty database. In this context, an elevated database role is used temporarily to create the required schema and load test data. This bootstrap step is performed **outside** the application runtime and exists solely to support reproducible testing. The application never grants itself elevated privileges, and schema creation is explicitly excluded from runtime behavior.

Step 4: Dependency graph summary

The dependency graph (dependency.svg) identifies app.py as the top-level entry point and final consumer of application logic. app.py imports query_data.py to serve read-only analytical results and imports the update pipeline modules (scrape_update.py, load_update.py, and load_data.py) to trigger background data ingestion. All database-interacting modules depend on db.py, which acts as the single shared database access layer and enforces environment-based configuration and least-privilege access. External libraries such as psycopg provide PostgreSQL connectivity, while threading and subprocess support asynchronous execution of the pipeline. Flask and its supporting modules sit above the standard library and are used exclusively by app.py for HTTP routing. The graph shows a clean, acyclic dependency structure with app.py correctly positioned as the final consumer of all application logic.

Step 5b. Add a setup.py – why packaging matters

Adding a setup.py makes the project an installable Python package, which ensures imports behave consistently across local runs, tests, and automated grading environments. Packaging eliminates reliance on implicit paths or environment-specific behavior, reducing “works on my machine” errors. It also enables editable installs (pip install -e .), which are commonly used during development and testing. By declaring dependencies in one place, tooling such as pip or uv can reliably reproduce the runtime environment. This improves portability, test reliability, and long-term maintainability of the project.

Step 6. Snyk Code Test (Static Application Security Testing) results

Snyk Code was run against the application source directory only to avoid scanning the virtual environment or third-party packages:

snyk code test src

Scan scope: module_5/src

Test type: Static code analysis (SAST)

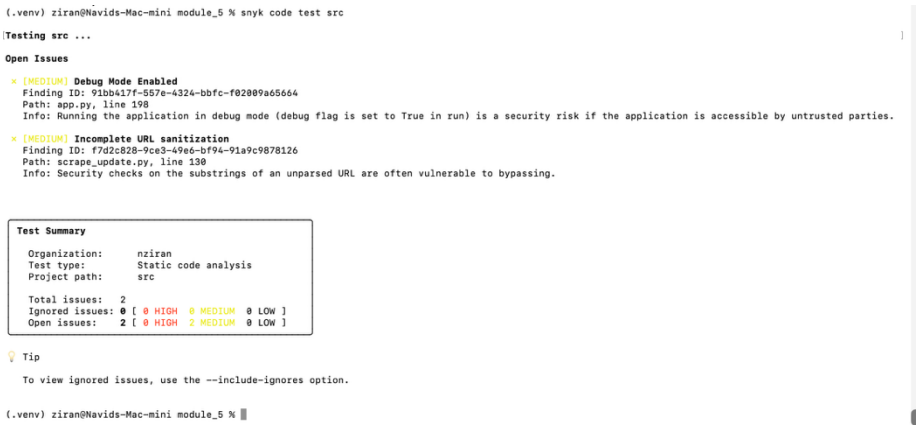
Results:

Total issues: 2

Severity: Medium (2)

High: 0

Low: 0



Findings:

1. Debug Mode Enabled

Flask is configured to run in debug mode in app.py. This is acceptable for local development and coursework but would be disabled in a production deployment.

2. Incomplete URL Sanitization

URL validation in scrape_update.py relies on substring checks rather than structured URL parsing. The data source is controlled and non-user-supplied, limiting risk in this context.

Conclusion:

No high-severity vulnerabilities were identified in the application source. All findings are development-context issues and have been documented for completeness.