Introduction

This report presentes the results of a Secure Coding Review performed on a simple Python application, Which was interntionally developed with security flaws for educational purposes.

The objective of the analysis was to identify common vulnerabilities, assesstheir risks, and propose recommendations based on secure development best pratices.

Analysis Scope

Laguage: Python

Analysis Type: Manual Static Code Analysis

Files Analyzed:

- main.py
- database.py
- file_handler.py
- system_utils.py

Methodology

The analysis was conducted through:

- Manual source code inspection
- Identification of user-controlled inputs
- Data flow analysis
- Verification of common insecure practices
- Risk classification based on impact and exploitability

Executive Summary

Severity	Quantity
High	**3**
Medium	**2**
Low	**1**

Overall Impact:

The application exhibits critical flaws that allow arbitrary command execution, reading of sensitive files, and database compromise, making it unsafe for use in real-world environments.

Identified Vulnerabilities

1. **Command Injection**
    - File: system_utils.py
    - Risk: High

Description: the **run_command()** function directly executes user-supplied commands using **os.popen()**, whitout any validation or restriction.

Impact: Na attacker can execute any command on the operating system, such as:

- Deleting files
- Creating users
- Installing malware
- Exfltrating data

Exploitation: User input examples:

- dir
- whoami
- del importante_file.txt

Recommendation:

- Never execute commands directly supplied by the user
- Use **subprocess.run()** with controlled lists (allowlist)
- Completely remove this functionality if it is not essential.

2. **SQL Injection**
    - File: datatabe.py
    - Risk: High

Description: The add_**user()** and **search_user()** functions build SQL queries using string concatenation with user-supplied data.

Impact allows:

- Data leakage
- Database alteration
- Table deletion
- Authentication by-pass

Exploitation: User input example

```
' OR '1'='1
```

Recommendation:

- Use parameterized queries: cursor.execute("SELECT * FROM users WHERE username LIKE ?", (term,))

3. **Path Traversal / Arbitrary File Read**
   - File: file_handler.py
   - Risk: High

Description: The **read_file()** function opens any file whose path is provided by the user, without validation.

```
open(filename, "r")
```

Impact Na attacker can read sensitive system files:

- /etc/passwd
- C:\Windows\System32\config
- Configuration files and credentials

Exploitation: Input example:

```
../../../../etc/passwd
```

Recommendation:

- Restrict reading to a specific directory.
- Validate paths with os.**path.abspath().**
- Use a list of allowed files **(allowlist).**

4. **Hardcoded Credentials**
   - **File:** database.py
   - **Risk:** Medium

Description: Database access credentials are hardcoded directly in the source code.

```
DB_USER = "admin"
DB_PASSWORD = "123456"
```

Impact:

- Exposure of secrets
- Compromise of the environment
- Facilitates reverse engineering

Recommendation:

- Use environment variables
- Use **.env** files
- Utilize secret management services

5. **Lack of Input Validation**
    - **File:** main.py
    - **Risk:** Medium

Description: No user input is validated regarding:

- **Type**
- **Size**
- **Expected contente**

Impact:

- Facilitates the exploitation of other vulnerabilities
- Can cause execution failures
- Increases the attack surface

Recommendation:

- Validate inputs
- Use regex
- Limit input size
- Handle exceptions correctly

6. **Lack of Error Handling**
    - **File:** General
    - **Risk:** Low

Description: The code does not use **try/except** to handle failures, potentially exposing internal errors.

Impact:

- Disclosure of internal information
- Facilitates application enumeration

Recommendation:

- Implement exception handling
- Create generic error messages for the user

Best Practices (Recommended)

- Input validation and sanitization
- Principle of Least Privilege
- Removal of dangerous code
- Secure credential management
- Secure logging
- Continuous code review

Conclusion

The analysis demonstrated that common development mistakes can result in critical vulnerabilities when secure coding practices are not applied.

This project reinforces the importance of the Secure Coding Review as an essential part of the Secure Development Lifecycle.

Legal Disclaimer This project is for educational purposes only.