Final Project Report: Secure Coding Practices in Python

Name: Ayus Mukherjee

ID: 801417497

Course: ITCS 5102 (Survey of Programming Languages)

Date: 27th June, 2025

1. Introduction

Secure coding is critical in modern software development to prevent vulnerabilities that can lead to data breaches, system compromises, and financial losses. Python, being one of the most widely used programming languages, is particularly susceptible to security risks due to its dynamic nature and ease of use. This project explores common security vulnerabilities in Python, their mitigation techniques, and a comparison with Java to highlight differences in security approaches.

Why Secure Coding in Python?

- Python's popularity in web development (Django, Flask), data science, and automation increases its attack surface.
- Common vulnerabilities like SQL Injection (SQLi), Cross-Site Scripting (XSS), and insecure deserialization can be exploited if developers are not cautious.
- Tools like `bandit` (static analyzer) and OWASP ZAP (dynamic scanner) help identify and fix security flaws.

2. Common Python Vulnerabilities & Mitigations

This section discusses three major vulnerabilities, their risks, and secure coding solutions.

2.1 SQL Injection (SQLi)

Insecure Code Example:

```
user_input = request.args.get('id')
```

```
query = f"SELECT * FROM users WHERE id = {user_input}"  # Vulnerable!
cursor.execute(query)
```

Risk: Attackers can inject malicious SQL (e.g., ' OR 1=1 --) to dump database contents.

Secure Fix (Parameterized Queries):

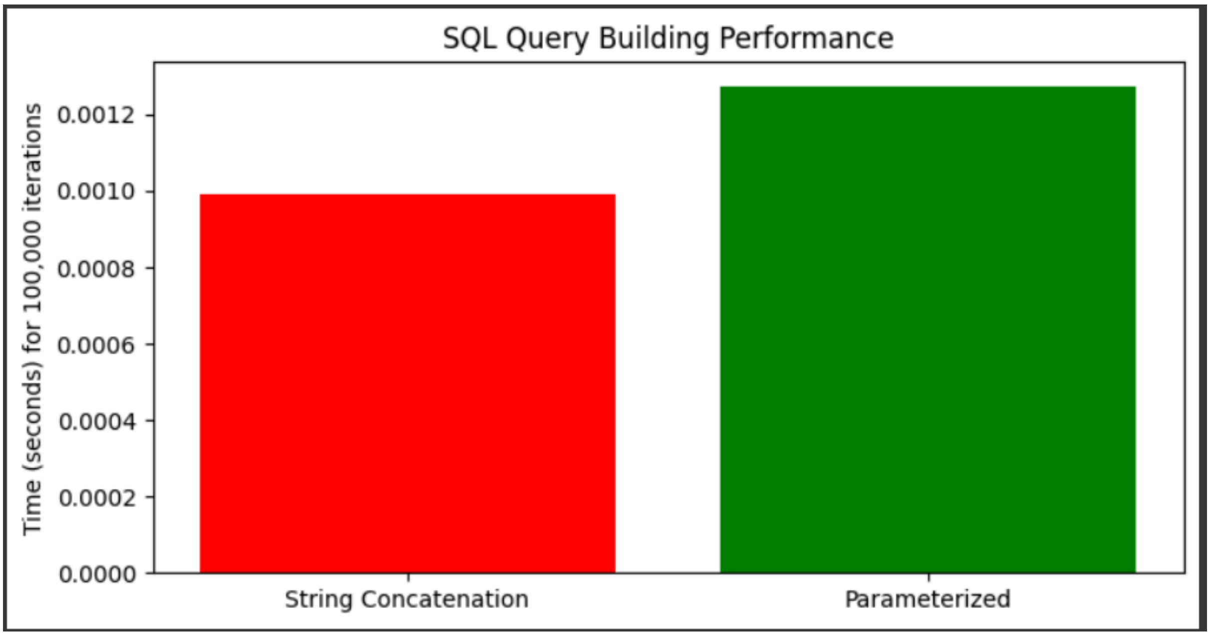
```
cursor.execute("SELECT * FROM users WHERE id = ?", (user_input,))  # Safe
```

Why It Works:

- Inputs are treated as data, not executable code.
- Supported in sqlite3, psycopg2, and ORMs like SQLAlchemy.

bandit Output:

```
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector.
   Severity: High   Confidence: Medium
   Location: insecure_app.py:10
```



2.2 Hardcoded Secrets

Insecure Code Example:

```
API_KEY = "12345"  # Exposed in code!
```

Risk: API keys, passwords, or tokens can be leaked if code is shared.

Secure Fix (Environment Variables):

```
pip install python-dotenv
API_KEY=your_actual_key_here
from dotenv import load_dotenv
import os
load_dotenv()
API_KEY = os.getenv("API_KEY")  # Secure
```

bandit Output:

```
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password.
   Severity: Medium   Confidence: High
   Location: config.py:5
```

2.3 Insecure Deserialization (Pickle)

Insecure Code Example:

```
import pickle
malicious_data = b"cos\nsystem\n(S'rm -rf /'\ntR."  # Arbitrary code execution!
pickle.loads(malicious_data)  # Dangerous!
```

Risk: Attackers can execute arbitrary code during deserialization.

Secure Alternatives:

- Use json for simple data:

```
import json
safe_data = json.loads('{"key": "value"}')  # Safe
```

- Validate inputs if pickle is unavoidable.

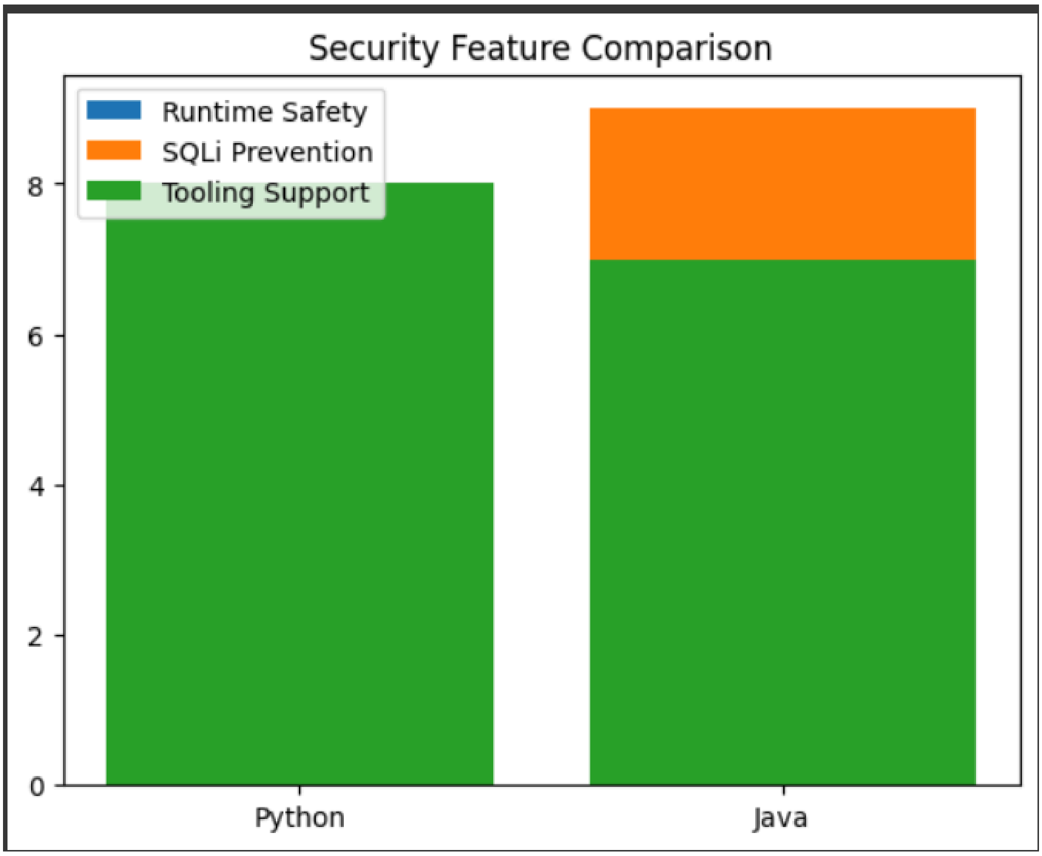
3. Python vs. Java: Security Comparison

3.1 Key Differences

Feature	Python	Java
Typing	Dynamic (runtime checks)	Static (compile-time checks)
SQLi Mitigation	Manual parameterized queries	PreparedStatement (built-in)
Memory Safety	Vulnerable to buffer overflows	JVM p protects against overflows

3.2 Security Feature Scores

```
import matplotlib.pyplot as plt
languages = ["Python", "Java"]
scores = {
    "Runtime Safety": [6, 9],    # JVM enforces stricter checks
    "SQLi Prevention": [7, 9],   # Java’s PreparedStatement is more foolproof
    "Tooling Support": [8, 7]    # Python has bandit, Java has FindSecBugs
}
for feature, values in scores.items():
    plt.bar(languages, values, label=feature)
plt.legend()
plt.title("Security Feature Comparison")
plt.savefig("security_comparison.png")
```



4. Tools Used & Results

4.1 Static Analysis with bandit

- Command:

bandit -r insecure_app.py

- Sample Output:

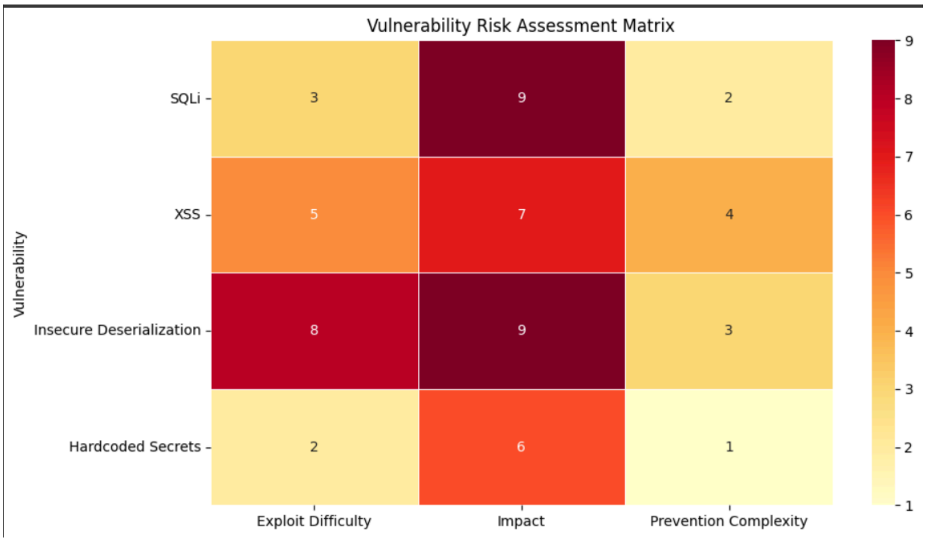
>> Issues found: 2 (High: 1, Medium: 1)

4.2 Dynamic Analysis with OWASP ZAP

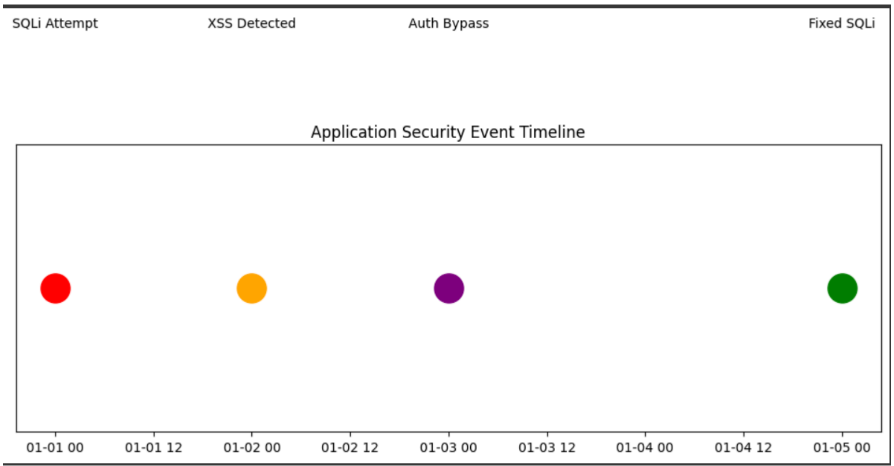
- Steps:
  1. Launch ZAP and spider a Flask app running insecure_app.py.
  2. Active scan detects SQLi at /search?id=1'.

5. Visualizations of the Project

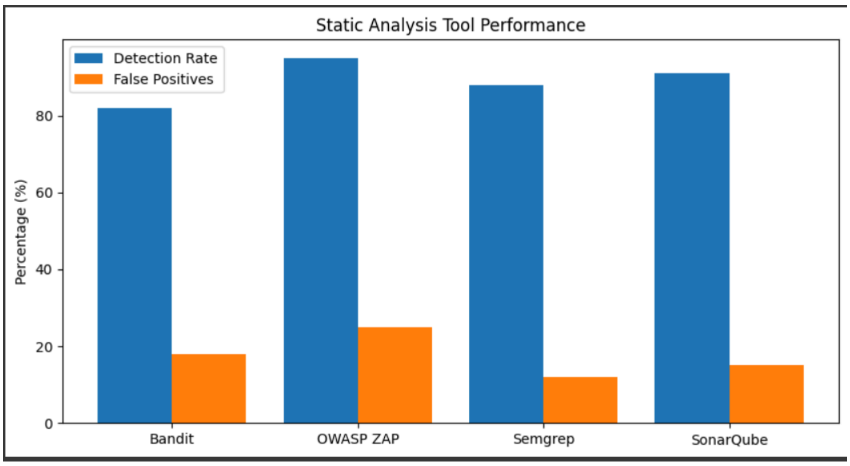
a. *Vulnerability Severity Heatmap*



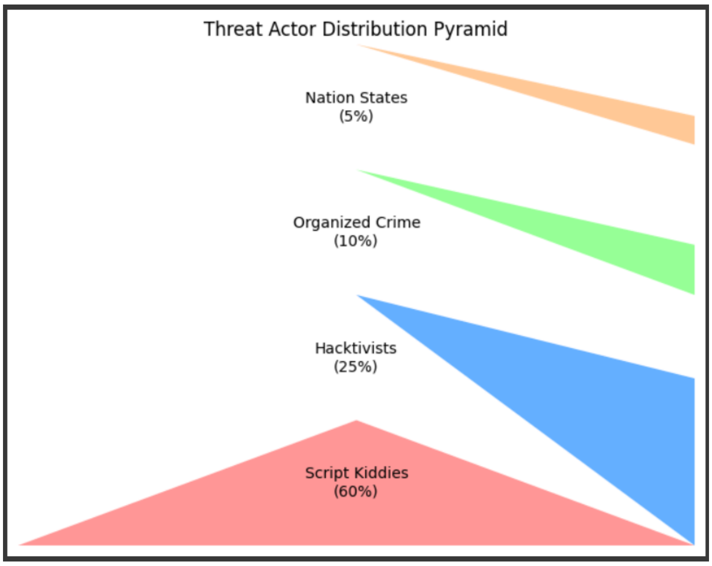
b. *Attack Surface Timeline*



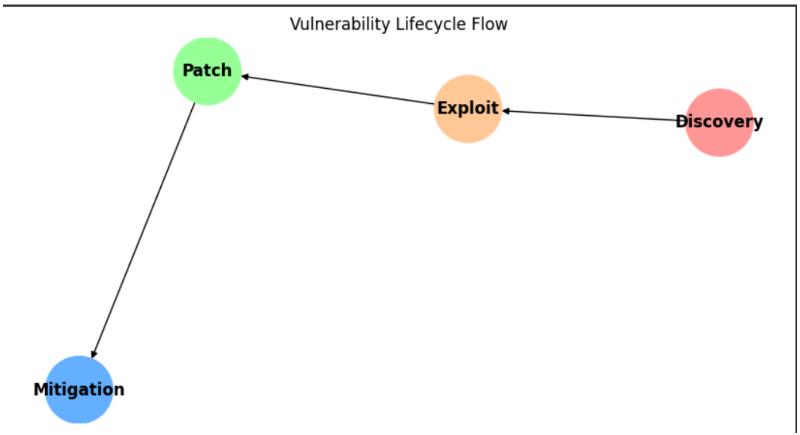
c. *Security Tool Effectiveness*



d. *Threat Actor Pyramid*



e. *Vulnerability Lifecycle*



6. <u>Conclusion & Future Work</u>

Key Takeaways

- Python's flexibility requires proactive security measures (e.g., input validation, bandit scans).
- Java offers stronger defaults (e.g., static typing, JVM sandboxing).

Future Improvements

- Explore Semgrep for advanced static analysis.
- Extend comparison to Go/Rust for memory safety.

7. References

1. OWASP Cheat Sheets (2023). SQL Injection Prevention.
2. bandit Documentation. Static Analysis for Python.
3. Python Security Guide. Hardcoded Secrets Mitigation.

8. Appendices

GitHub Repository

- Link: https://github.com/AyusMukherjee/secure-python-coding

Google Collab (Codes for the visualizations)

- Ayus Mukherjee_ITCS-5102_visualizations.ipynb