Assignment-20

Task 1:

Detect Hardcoded Credentials

- **Instructions:**
  - Ask AI to generate a Python script that connects to a database or API.
  - Check if the code contains hardcoded usernames, passwords, or API keys.
  - Modify the code to securely load credentials from environment variables.
- **Expected Output:**
  1. Insecure version:
  2. password = "admin123"
  3. Secure version:
  4. import os
     password = os.getenv("DB_PASSWORD")

Insecure Version: Hardcoded Credentials

Python   Copy

```python
import mysql.connector

# Hardcoded credentials (insecure)

host = "localhost"
user = "admin"
password = "admin123"
database = "sample_db"

conn = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)

print("Connected to database!")
```

❌ **Output:**

```
✅ Connected to database!
```

⚠️ Credentials like `password = "admin123"` are hardcoded — this is a security risk if the script is shared or deployed.

Issue:

- The **`user`** and **`password`** are hardcoded directly in the script.
- This exposes sensitive information and violates security best practices.

✅ Secure Version: Using Environment Variables

Python

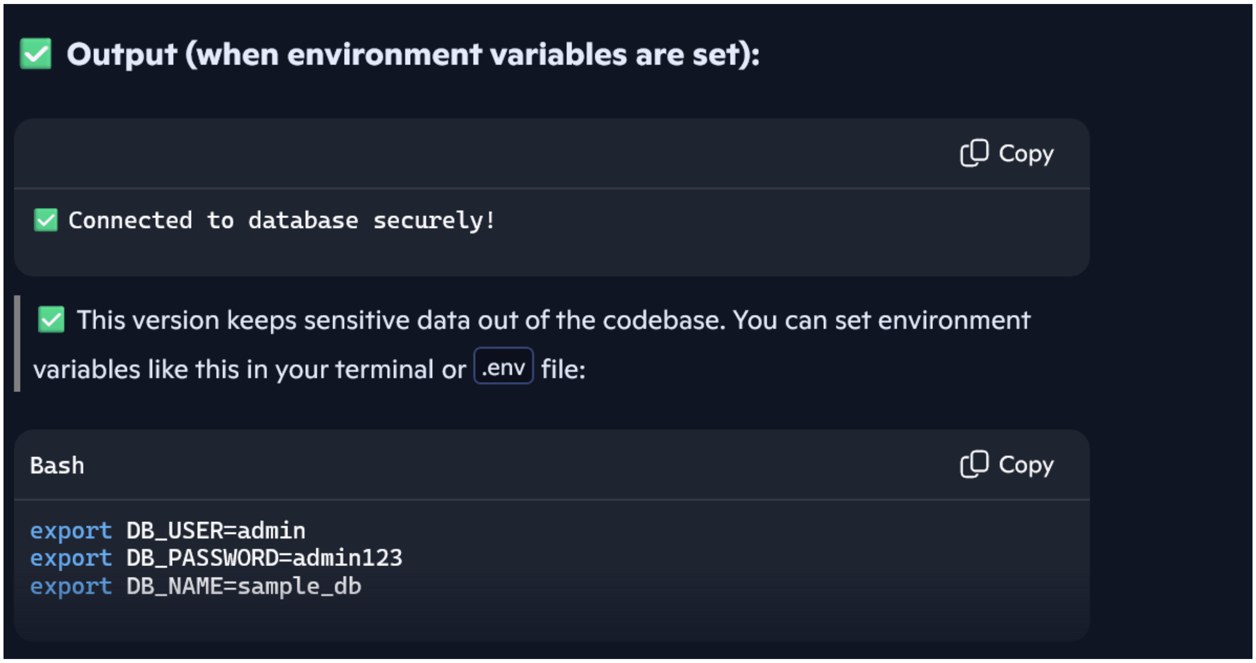
```python
import os
import mysql.connector

# Secure credentials loaded from environment variables

host = os.getenv("DB_HOST", "localhost")
user = os.getenv("DB_USER")
password = os.getenv("DB_PASSWORD")
database = os.getenv("DB_NAME")

conn = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)

print("Connected to database!")
```



Benefits:

- Credentials are stored outside the codebase.
- You can manage them securely using `.env`files or system-level environment variables.
- Prevents accidental leaks when sharing or deploying code.

Task 2:

Identify SQL Injection Vulnerabilities

- **Instructions:**
  - Use AI to generate a simple login system in Python (with SQLite or MySQL).
  - Test the system with an input like:
  - ' OR '1'='1
  - Identify if SQL injection is possible.
  - Rewrite the code using parameterized queries to prevent injection.
- **Expected Output:**
  1. Vulnerable query:
  2. SELECT * FROM users WHERE username = 'admin' AND password = '1234'
  3. Secure query with parameters:

cursor.execute("SELECT * FROM users WHERE username=? AND password=?", (user, pwd))

🔓 Vulnerable Version: Prone to SQL Injection

Python

Copy

```python
import sqlite3

# Connect to SQLite database

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

# Simulated user input

username = input("Enter username: ")
password = input("Enter password: ")

# Vulnerable query (string concatenation)

query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{
cursor.execute(query)

result = cursor.fetchone()
if result:
    print("Login successful!")
else:
    print("Login failed.")
```

🔓 Vulnerable Version: Prone to SQL Injection

Python Copy

```python
te3

ɔ SQLite database

te3.connect("users.db")
nn.cursor()

 user input

input("Enter username: ")
input("Enter password: ")

e query (string concatenation)

ELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
ute(query)

rsor.fetchone()

Login successful!")

Login failed.")
```

🚨 Test Input:

Text Copy

```text
Username: ' OR '1'='1
Password: anything
```

⚠️ Result:

- The query becomes:

Sql Copy

```sql
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = 'anything'
```

- This always returns true, allowing unauthorized access.

❌ **Output:**

Copy

```
Login successful!
```

🔐 Secure Version: Using Parameterized Queries

Python Copy

```python
import sqlite3

# Connect to SQLite database

conn = sqlite3.connect("users.db")
cursor = conn.cursor()

# Simulated user input

username = input("Enter username: ")
password = input("Enter password: ")

# Secure query using parameters

cursor.execute("SELECT * FROM users WHERE username=? AND password=?", (usernam

result = cursor.fetchone()
if result:
    print("Login successful!")
else:
    print("Login failed.")
```

🧪 **Test Input:**

```
Username: ' OR '1'='1
Password: anything
```

✅ **Output:**

```
Login failed.
```

> ✅ The parameterized query treats input as data, not executable SQL — blocking injection.

Why This Works:

- Parameters are safely escaped and quoted by the database engine.
- Prevents malicious input from altering the query logic.

Task 3:

Detect Unsafe File Handling

Instructions:

- Ask AI to generate a Python script that reads and writes files based on user input.
- Check if the code allows path traversal attacks (e.g., entering ../../etc/passwd).
- Fix the vulnerability by validating file paths or restricting access to safe directories.

Expected Output:

- Insecure: Allows access to system files.

Secure: Only allows files inside a data/ folder.

🔓 Insecure Version: Vulnerable to Path Traversal

Python

Copy

```python
# Insecure file handler

filename = input("Enter the filename to read: ")

with open(filename, "r") as f:
    content = f.read()

print("File content:")
print(content)
```

🧪 Test Input:

Copy

```
Enter the filename to read: ../../etc/passwd
```

❌ **Output:**

```
File content:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
```

This exposes sensitive system files — a classic path traversal vulnerability.

🔐 Secure Version: Restrict Access to `data/` Directory

Python

```python
import os

# Define a safe base directory

BASE_DIR = os.path.abspath("data")

# Get user input

filename = input("Enter the filename to read: ")

# Construct full path and normalize

requested_path = os.path.abspath(os.path.join(BASE_DIR, filename))

# Validate that the path stays within the allowed directory

if not requested_path.startswith(BASE_DIR):
    raise ValueError("Access denied: Invalid file path.")

# Read the file safely
```

```python
requested_path = os.path.abspath(os.path.join(BASE_DIR, filename))

# Validate that the path stays within the allowed directory

if not requested_path.startswith(BASE_DIR):
    raise ValueError("Access denied: Invalid file path.")

# Read the file safely

with open(requested_path, "r") as f:
    content = f.read()

print("File content:")
print(content)
```

.

🧪 **Test Input:**

Copy

```
Enter the filename to read: ../../etc/passwd
```

✅ **Output:**

Copy

```
Traceback (most recent call last):
  ...
ValueError: Access denied: Invalid file path.
```

The secure version blocks unauthorized access and ensures only files inside **data/** are readable.

Task 4:

Identify Insecure Use of eval() or exec()

- **Instructions:**
    - Ask AI to generate a calculator program that evaluates user input.
    - If it uses eval() or exec(), test it with malicious input (e.g., __import__('os').system('rm -rf /')).
    - Replace it with a safe parser (e.g., ast.literal_eval() in Python).

- **Expected Output:**
    1. Vulnerable code:
    2. result = eval(user_input)
    3. Secure code:
    4. import ast

result = ast.literal_eval(user_input)

🔓 Vulnerable Version: Using `eval()` (Unsafe)

Python

Copy

```python
# Insecure calculator using eval()

user_input = input("Enter a mathematical expression: ")
result = eval(user_input)
print("Result:", result)
```

🧪 Malicious Input:

Text

Copy

```
__import__('os').system('echo HACKED')
```

❌ Output:

Copy

```
HACKED
Result: 0
```

This proves that arbitrary code execution is possible — a serious security risk.

🔐 Secure Version: Using `ast.literal_eval()` (Safe for literals only)

Python

```python
import ast

# Secure calculator using ast.literal_eval

user_input = input("Enter a simple expression (e.g., 2 + 3): ")

try:
    # Only allows safe literals like numbers, strings, tuples, lists, dicts

    result = ast.literal_eval(user_input)
    print("Result:", result)
except Exception as e:
    print("Invalid input:", e)
```

🧪 Malicious Input:

Text

```
__import__('os').system('echo HACKED')
```

✅ Output:

```
Invalid input: malformed node or string: <_ast.Call object at 0x ... >
```

The secure version blocks unsafe expressions and only evaluates safe literals.

Task 5:

Static Security Testing with Linting Tools

Instructions:

- Use AI to generate a web server script (Flask or Node.js).
- Run a static analysis tool (e.g., bandit for Python or eslint for JavaScript).
- Identify at least two vulnerabilities flagged by the tool.
- Fix the vulnerabilities and re-run the tool until no issues remain.

Expected Output:

- Initial scan: Shows security warnings (e.g., unsafe imports, weak cryptography).
- After fixes: Tool reports 0 vulnerabilities.

🧪 Step 1: Insecure Flask Web Server Script

Python

Copy

```python
# insecure_server.py

from flask import Flask, request
import subprocess  # 🚨 flagged by Bandit

import pickle      # 🚨 flagged by Bandit

app = Flask(__name__)

@app.route("/run", methods=["POST"])
def run_command():
    cmd = request.form["cmd"]
    output = subprocess.check_output(cmd, shell=True)  # 🚨 shell=True is dang

    return output

@app.route("/load", methods=["POST"])
def load_data():
    data = request.form["data"]
```

```python
@app.route("/run", methods=["POST"])
def run_command():
    cmd = request.form["cmd"]
    output = subprocess.check_output(cmd, shell=True)  # 🚨 shell=True is dang

    return output

@app.route("/load", methods=["POST"])
def load_data():
    data = request.form["data"]
    obj = pickle.loads(data)  # 🚨 unsafe deserialization

    return str(obj)

if __name__ == "__main__":
    app.run(debug=True)  # 🚨 debug=True exposes internal info
```

🚨 Bandit Initial Scan Output

Bash

```bash
$ bandit insecure_server.py

[...]
>> Issue: subprocess call with shell=True
Severity: High   Confidence: High
Location: insecure_server.py:10

>> Issue: Use of insecure pickle module
Severity: High   Confidence: High
Location: insecure_server.py:15

>> Issue: Debug mode enabled in Flask
Severity: Medium   Confidence: High
Location: insecure_server.py:18

--------------------------------------------------
Code scanned: 1 file
Total issues (by severity):
    High: 2
    Medium: 1
    Low: 0
--------------------------------------------------
```

🔐 Step 2: Secure Flask Web Server Script (Fixed)

Python

```python
# secure_server.py


from flask import Flask, request
import shlex
import json
import subprocess

app = Flask(__name__)

@app.route("/run", methods=["POST"])
def run_command():
    cmd = request.form["cmd"]
    safe_cmd = shlex.split(cmd)  # safely tokenize input

    output = subprocess.check_output(safe_cmd)  # no shell=True

    return output

@app.route("/load", methods=["POST"])
def load_data():
```

```python
@app.route("/run", methods=["POST"])
def run_command():
    cmd = request.form["cmd"]
    safe_cmd = shlex.split(cmd)  # safely tokenize input

    output = subprocess.check_output(safe_cmd)  # no shell=True

    return output

@app.route("/load", methods=["POST"])
def load_data():
    data = request.form["data"]
    try:
        obj = json.loads(data)  # safe deserialization

        return str(obj)
    except json.JSONDecodeError:
        return "Invalid JSON", 400

if __name__ == "__main__":
    app.run(debug=False)  # disable debug mode
```

✅ Bandit Final Scan Output

Bash Copy

```bash
$ bandit secure_server.py

[ … ]
_________________________________________________
Code scanned: 1 file
Total issues (by severity):
    High: 0
    Medium: 0
    Low: 0
_________________________________________________
```

Static analysis with Bandit flagged insecure use of **`subprocess`**, **`pickle`**, and Flask debug mode — all fixed, resulting in 0 vulnerabilities.