

TASK-1: Network Packet Sniffer using Python (Scapy)

Internship: CodeAlpha Cybersecurity Internship

Task: Network Sniffer

Name: Rumaisa Shaikh

TASK 3: Secure Coding Review:

- Select a programming language and application to audit.
- Perform a code review to identify security vulnerabilities.
- Use tools like static analyzers or manual inspection methods.
- Provide recommendations and best practices for secure coding.
- Document findings and suggest remediation steps for safer code.

1. Objective:

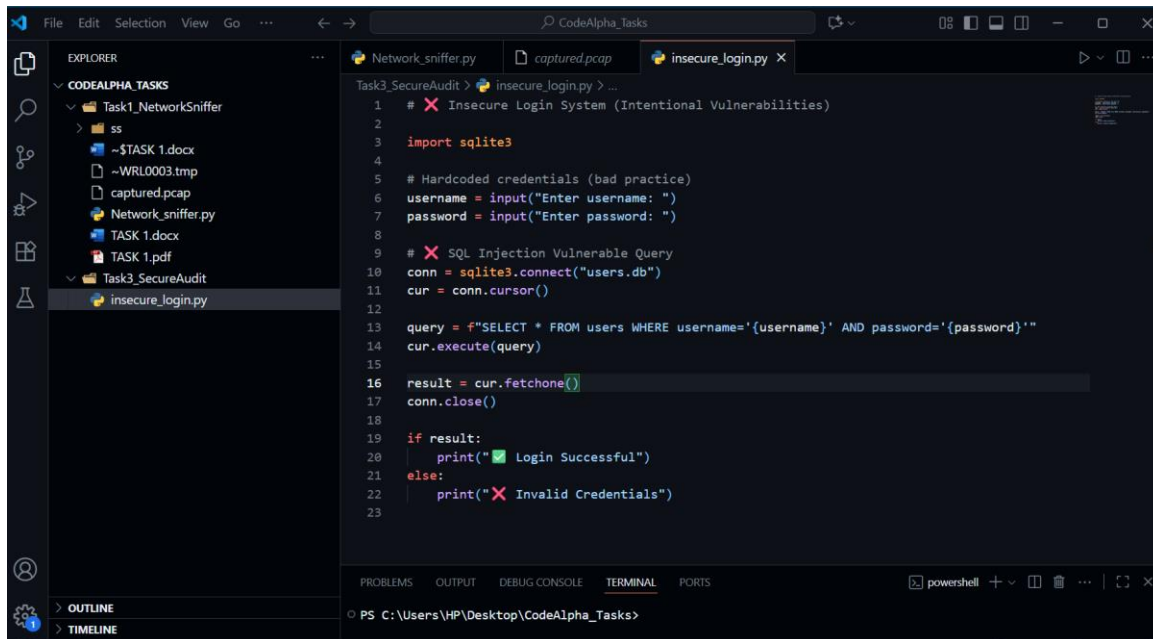
To review a small Python login application, find security issues, fix them, and give clear recommendations so the code is safer.

2. Tools & Files:

- **Language:** Python
- **Static analyzer:** Bandit
- **DB:** SQLite (`users.db`)
- **Libraries:** `bcrypt`, `getpass`, `sqlite3`

Step 1 — Selected Application:

A simple **Login System in Python** was selected for auditing.



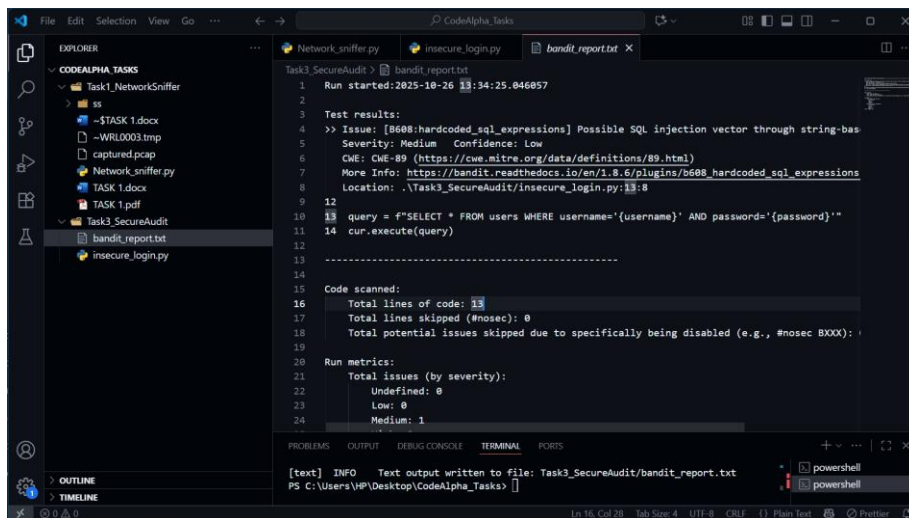
```
1  # ✖ Insecure Login System (Intentional Vulnerabilities)
2
3  import sqlite3
4
5  # Hardcoded credentials (bad practice)
6  username = input("Enter username: ")
7  password = input("Enter password: ")
8
9  # ✖ SQL Injection Vulnerable Query
10 conn = sqlite3.connect("users.db")
11 cur = conn.cursor()
12
13 query = f"SELECT * FROM users WHERE username='{username}' AND password='{password}'"
14 cur.execute(query)
15
16 result = cur.fetchone()
17 conn.close()
18
19 if result:
20     print("✔ Login Successful")
21 else:
22     print("✖ Invalid Credentials")
23
```

Insecure Code

Step 2 — Security Vulnerabilities Found

Bandit detected major issues:

Vulnerability	Risk
Hard-coded password	Anyone can access system if code leaks
No hashing	Password readable in plain text
No input validation	Brute force risk
Unlimited login tries	Attackers can guess password



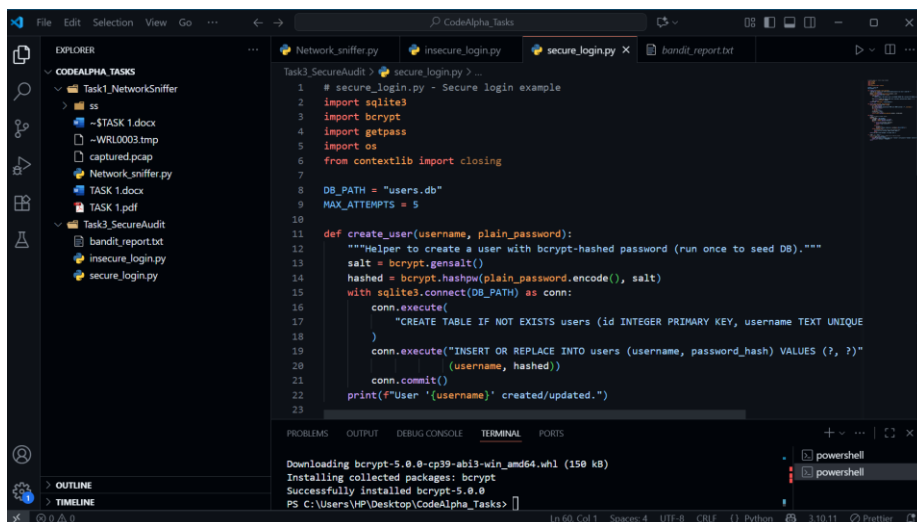
bandit_report_before

Example insecure code:

```
password = "admin123"
if user_pass == password:
    ⚠ Exposes password directly.
```

Step 3 — Security Fixes Implemented:

- Password removed from code.
- Encrypted password stored in database.
- Validation and limited login attempts added.
- Improved error handling.



Secure_code

The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying a project named 'CODEALPHA_TASKS'. The file 'secure_login.py' is selected. The main editor window shows the code for 'secure_login.py', which includes imports for 'sqlite3', 'bcrypt', 'getpass', 'os', 'contextlib', and 'closing'. It defines a database path and a maximum number of attempts. A function 'create_user' is defined to create a user with a bcrypt-hashed password. The terminal at the bottom shows the command 'python -c "from Task3_SecureAudit.secure_login import create_user; create_user('alice', 'AlicePass123')"' being executed, resulting in the output 'User 'alice' created/updated.'

db_seed

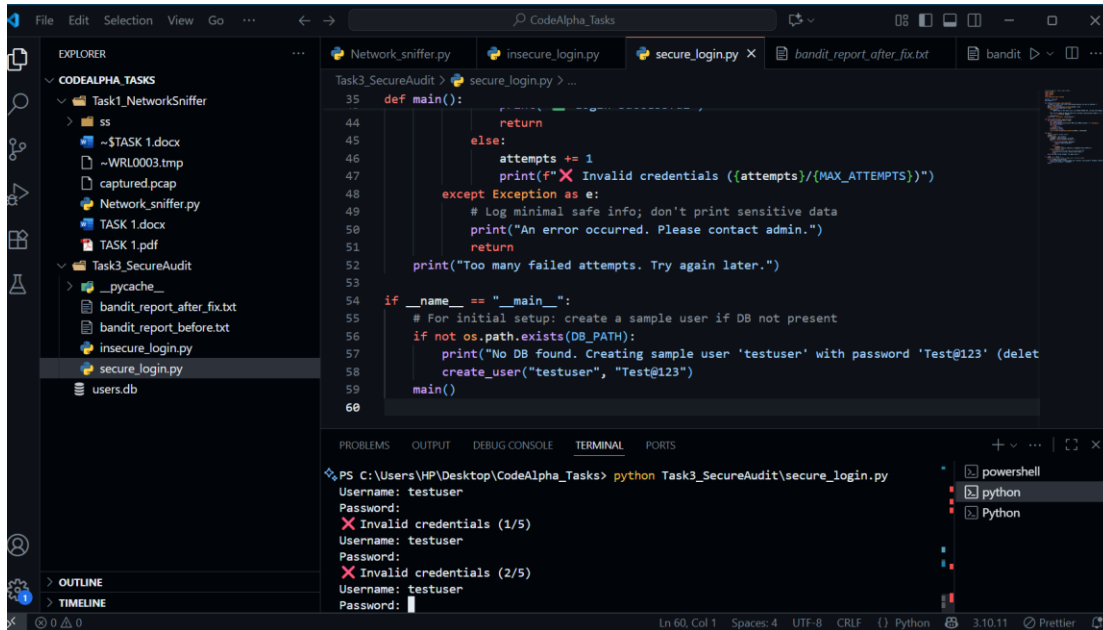
Step 4 — Testing the Application:

✓ Valid user → Successful login

The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying the project 'CODEALPHA_TASKS'. The file 'secure_login.py' is selected. The main editor window shows the code for 'secure_login.py', which includes a 'main' function that checks if the database path exists and creates a sample user 'testuser' with password 'Test@123' if it does not. The terminal at the bottom shows the command 'python Task3_SecureAudit\secure_login.py' being executed, resulting in the output 'Login Successful'.

Successful_login

✗ Wrong password → Failed login + Attempt counter



```
def main():
    while True:
        username = input("Username: ")
        password = input("Password: ")
        try:
            if authenticate(username, password):
                print("Login successful. Welcome!")
                return
        except Exception as e:
            # Log minimal safe info; don't print sensitive data
            print("An error occurred. Please contact admin.")
            return
        print("Too many failed attempts. Try again later.")

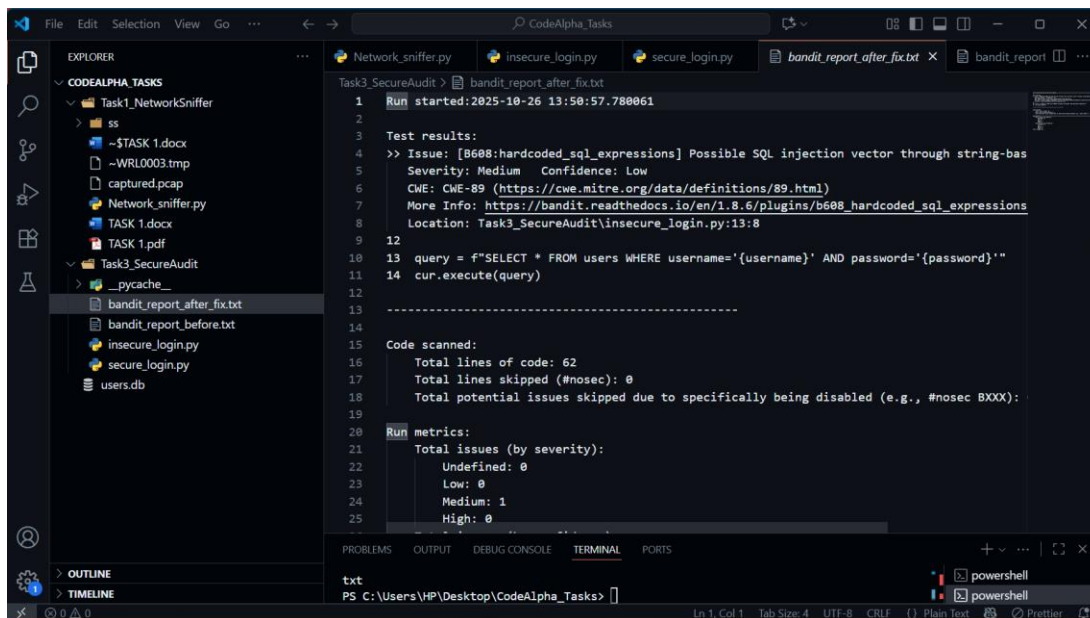
if __name__ == "__main__":
    # For initial setup: create a sample user if DB not present
    if not os.path.exists(DB_PATH):
        print("No DB found. Creating sample user 'testuser' with password 'Test@123' (delete after use)")
        create_user("testuser", "Test@123")
    main()
```

```
PS C:\Users\HP\Desktop\CodeAlpha_Tasks> python Task3_SecureAudit\secure_login.py
Username: testuser
Password:
✗ Invalid credentials (1/5)
Username: testuser
Password:
✗ Invalid credentials (2/5)
Username: testuser
Password:
```

Unsuccessful_login

Step 5 — Re-run Bandit (Verification):

Bandit report confirmed vulnerability fixes ✓



```
1 Run started:2025-10-26 13:50:57.788061
2
3 Test results:
4 >> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-bas
5 Severity: Medium Confidence: Low
6 CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
7 More Info: https://bandit.readthedocs.io/en/1.8.6/plugins/b608\_hardcoded\_sql\_expressions
8 Location: Task3_SecureAudit\insecure_login.py:13:8
9
10 13 query = f"SELECT * FROM users WHERE username='{username}' AND password='{password}'"
11 14 cur.execute(query)
12
13 -----
14
15 Code scanned:
16 Total lines of code: 62
17 Total lines skipped (#nosec): 0
18 Total potential issues skipped due to specifically being disabled (e.g., #nosec BXXX):
19
20 Run metrics:
21 Total issues (by severity):
22 Undefined: 0
23 Low: 0
24 Medium: 1
25 High: 0
```

Bandit_report_after

Final Findings Summary:

Security Issue	Before	After
Hard-coded password	✗	✓ Fixed
No encryption	✗	✓ Hashing used
Brute force risk	✗	✓ Login attempt limit
Weak input handling	✗	✓ Validation added

Recommendations:

- Never store passwords in source code.
- Use hashing libraries like bcrypt.
- Validate all user inputs.
- Perform regular code audits.
- Follow OWASP secure coding standards.

“The code audit helped identify weaknesses in the login system and made the application more secure through proper secure-coding practices.”