Secure Password Policy Testing on a Sample Application

Student: Yeddula Chakradhar Reddy

Course: Cyber Security

Duration: 3 Days

Date: 09-10-2025

Table of Contents

Table of Contents 2
Abstract 3
Objective 3
Environment & Tools 4
Design & Architecture 4
Implementation 5
Code Structure 5
Important Modules 5
Testing / Evidence 6
Overview 6
Testing of Phase-1 6
Testing of Phase-2 10
Results & Conclusion 13
Key findings: 13
Next steps / recommendations: 13
References 14

Abstract

This project evaluates password security controls through a reproducible laboratory built with two Flask + SQLite web applications. The Phase-1 application demonstrates insecure defaults: it accepts weak passwords, stores credentials with single-round MD5, and contains deliberately vulnerable SQL query patterns that expose stored hashes. The Phase-2 application implements mitigations by enforcing server-side password complexity (minimum length, mixed case, digits and special characters), storing passwords using a salted, iterated PBKDF2 scheme, and applying account lockouts after repeated failed login attempts.

Using Burp Suite to capture HTTP interactions, and John the Ripper / Hashcat for offline cracking, I reproduced common attack paths: extraction of stored hashes via SQLi, offline cracking of fast hashes with commodity wordlists, and attempted brute-force against the live login endpoint.

Results show that MD5-hashed credentials are quickly recovered with rockyou/Hashcat, whereas PBKDF2 with a high iteration count greatly increases attacker cost and makes offline recovery impractical in a lab setting. The study also demonstrates that server-side complexity checks prevent trivial passwords at creation time and that lockout policies effectively mitigate rapid online guessing.

Based on these findings I recommend using slow, salted hash functions (Argon2/bcrypt/PBKDF2 with tuned work factors), enforcing robust server-side password policies, adding MFA, and implementing IP/rate limiting and logging to further reduce compromise risk. All experiments were conducted in an isolated VM lab and are fully reproducible with supplied source code and test scripts.

Objective

To demonstrate and measure how server-side password policies, slow salted hashing, and account lockouts prevent credential compromise compared to insecure password practices.

Environment & Tools

- Operating System: Kali Linux 2025.1 (64-bit)
- Programming Language & Framework: Python 3.11 & Flask 2.x
- Database: SQLite 3.x
- Password Hashing Library: Werkzeug (generate_password_hash, check_password_hash)
- Testing / Attack Tools:
    - John the Ripper (for hash cracking)
    - Hashcat (for hash cracking and testing weak passwords)
    - Burp Suite (for web requests, brute force attacks)
    - Browser for Manual Testing: Firefox

Design & Architecture

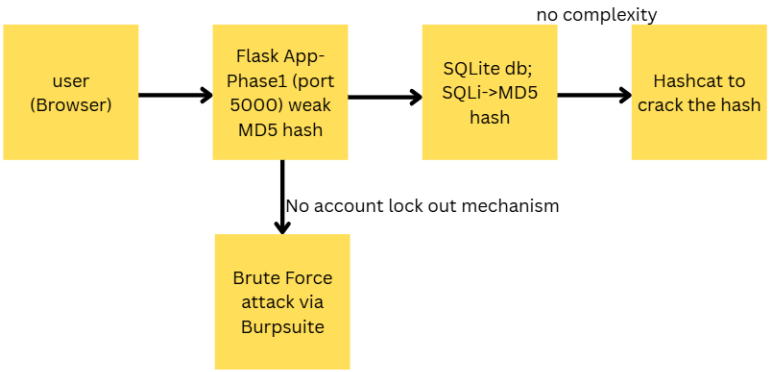

Fig 1: Lab Architecture and data flow for Phase-1

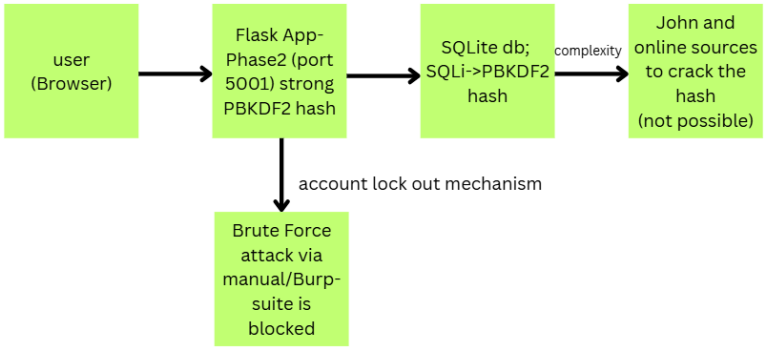

Fig 2: Lab Architecture and data flow for Phase-2

Implementation

The project was implemented in two phases using the Flask micro-framework with a backend SQLite database. Each phase demonstrates different levels of password security and storage mechanisms.

Code Structure

- project_root/
  - src/
    - app_phase1_vuln.py — Phase-1 vulnerable Flask app (MD5, unsafe SQL)
    - app_phase2_fixed.py — Phase-2 hardened Flask app (PBKDF2, complexity, lockout)
  - phase1.db — SQLite DB for Phase-1
  - phase2.db — SQLite DB for Phase-2

Important Modules

- **Flask:** To create the web server, define routes like `/login`, `/set_password`, and handle user input.
- **SQLite3:** Used as a lightweight database to store usernames and password hashes.
- **hashlib:** Used in Phase-1 to generate insecure MD5 hashes.
- **werkzeug.security:** Used in Phase-2 for secure PBKDF2 hashing (generate_password_hash, check_password_hash).
- **re (Regular Expressions):** Implements password complexity checks (uppercase, lowercase, digit, special character).
- **time:** Used in lockout mechanism to temporarily disable login after multiple failed attempts.

The complete implementation source code for this project is provided in the repository under the src/ directory. It includes two main files — app_phase1_vuln.py and app_phase2_fixed.py — representing the vulnerable and secured versions of the Flask web application, respectively. These files contain the full logic for user authentication, password storage, and policy enforcement.

Controlled extraction mechanism: For the purposes of directly comparing stored credential protection schemes, I deliberately included an extraction vector (an SQLi-enabled endpoint) in both Phase-1 and Phase-2 test builds solely to obtain stored password hashes for offline analysis. The extraction mechanism was intentionally added to the isolated lab builds to enable reproducible measurement of hashing and account protections; exploitation of SQL injection vulnerabilities per se is outside the primary scope of this project.

Testing / Evidence

Overview

The testing suite demonstrates three core areas required by the assignment:

1. **Online brute-force via Burp (Phase-1)** — Phase-1 has no lockout or rate limiting, so Burp Suite (Intruder) can be used to automate rapid login attempts against /login to demonstrate how an attacker could guess credentials online; Phase-2 shows the mitigation by preventing such automated successful attempts.

2. **Weak hashing & offline cracking** — Phase-1 stores MD5 hashes; these are extracted and cracked with John/Hashcat; Phase-2 stores PBKDF2 hashes and resists cracking.

3. **Password complexity enforcement** (server-side) — Phase-2 /set_password rejects weak passwords and returns all missing rules.

4. **Brute-force / lockout** — Phase-2 /login locks an account after LOCK_THRESHOLD failed attempts.

All tests were executed in an isolated Kali VM. The commands below are exactly what was used; include the resulting screenshots / text files as deliverables.

Testing of Phase-1

I created and activated a Python virtual environment and launched the Phase-1 web application from src/app_phase1_vuln.py. I used **source venv/bin/activate** to activate the python virtual environment. Then, **python src/app_phase1_vuln.py** to launch the Phase-1 web application.

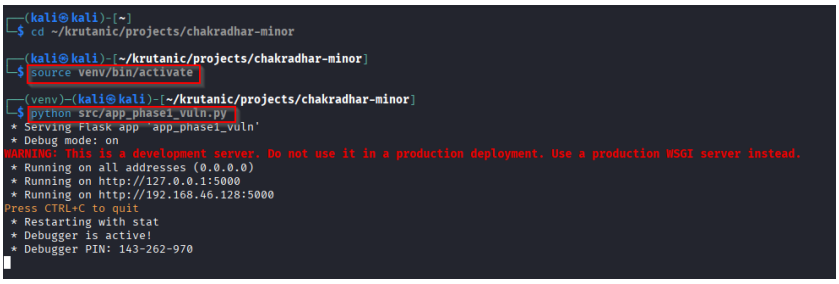

Fig 3: commands in Kali terminal

I opened the app in Firefox at http://192.168.46.128:5000 and verified a successful login using the seeded credentials username: ‘testuser’ and password: ‘Password123!’

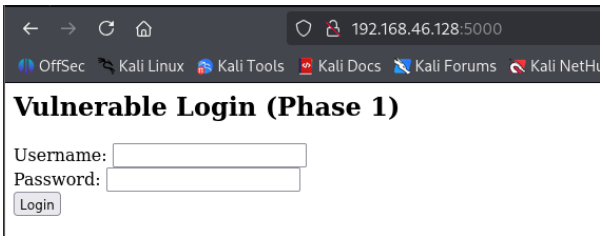

Fig 4: Login page of phase-1 web app

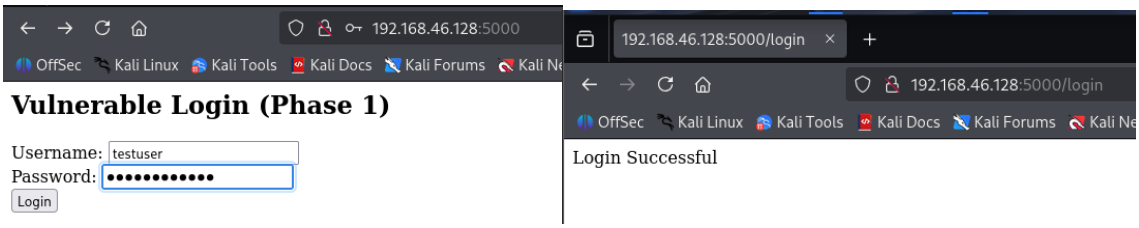

Fig 5: Successful authentication of the login page

I then used the app's web form to set a deliberately weak password (for example admin) to demonstrate lack of complexity enforcement. I used the curl command to set the password: 'admin' for username: 'test'

```
┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=test&password=admin' http://192.168.46.128:5000/set_password
ACCEPTED:password_set
```

Fig 6: Curl command to set weak password: 'admin'

Using the lab-only SQL-extraction endpoint I retrieved the stored credential and observed it was a single-round MD5 hash. The SQLi query used is sqli?name=' OR '1'='1

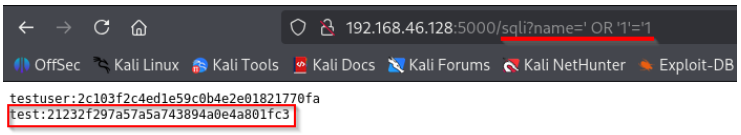

Fig 7: SQLi query and extraction of weak MD5 Hash

I exported that hash to a file named weak_hash.txt and ran Hashcat/John with a common wordlist (rockyou.txt), which quickly recovered the plaintext password.

Command: **hashcat –m 0 weak_hash.txt /usr/share/wordlists/rockyou.txt**

```
┌──(kali㉿kali)-[~/krutanic/projects/chakradhar-minor]
└─$ hashcat -m 0 weak_hash.txt /usr/share/wordlists/rockyou.txt
hashcat (v6.2.6) starting

OpenCL API (OpenCL 3.0 PoCL 6.0+debian  Linux, None+Asserts, RELOC, SPIR-V, LLVM 18.1.8, SLEEF, DISTRO, POCL_DEBUG) - Platform #1 [The pocl project]
====================================================================================================================================
* Device #1: cpu-sandybridge-11th Gen Intel(R) Core(TM) i5-1155G7 @ 2.50GHz, 1435/2934 MB (512 MB allocatable), 6MCU
```

Fig 8: Hashcat command

```
Dictionary cache hit:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keyspace..: 14344385

21232f297a57a5a743894a0e4a801fc3:admin
```

Fig 9: the caracked md5 hash

Finally, I used Burp Suite to automate online guessing against /login; because there was no rate-limiting or account lockout in Phase-1, brute-force attempts were trivial and I reproduced the successful recovery of the testuser password (Password123!).

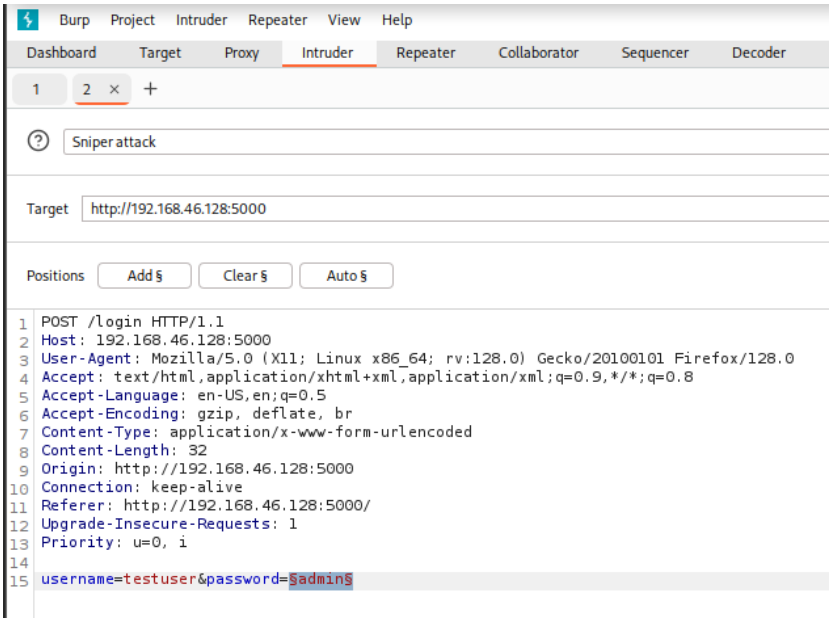

Fig 10: Intercepted request sent to the intruder

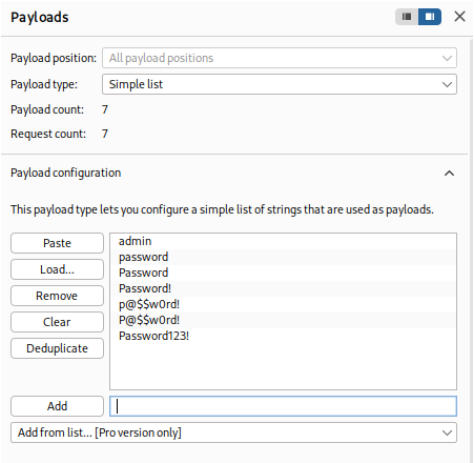

Fig 11: List of passwords as payload

The password which I got is Password123! as it has got a different length compared to the other payloads.

Results Positions

Capture filter: Capturing all items

View filter: Showing all items

Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
1	admin	200	1			185	
2	password	200	1			185	
3	Password	200	1			185	
4	Password!	200	2			185	
5	p@$$w0rd!	200	3			185	
6	P@$$w0rd!	200	2			185	
7	Password123!	200	2			189	

Fig 12: The right password

Attack Save 2. Intruder attack of http://192.168.46.128:5000

2. Intruder attack of http://192.168.46.128:5000

Results Positions

Capture filter: Capturing all items

View filter: Showing all items

Request	Payload	Status code	Response received	Error	Timeout	Length	Comment
1	admin	200	1			185	
2	password	200	1			185	
3	Password	200	1			185	
4	Password!	200	2			185	
5	p@$$w0rd!	200	3			185	
6	P@$$w0rd!	200	2			185	
7	Password123!	200	2			189	

Request Response

Pretty Raw Hex

```
1  POST /login HTTP/1.1
2  Host: 192.168.46.128:5000
3  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-US,en;q=0.5
6  Accept-Encoding: gzip, deflate, br
7  Content-Type: application/x-www-form-urlencoded
8  Content-Length: 39
9  Origin: http://192.168.46.128:5000
10 Connection: keep-alive
11 Referer: http://192.168.46.128:5000/
12 Upgrade-Insecure-Requests: 1
13 Priority: u=0, i
14
15 username=testuser&password=Password123!
```

Fig 13: The request captured

Attack Save 2. Intruder attack of http://192.168.46.128:5000

2. Intruder attack of http://192.168.46.128:5000

Results Positions

Capture filter: Capturing all items

View filter: Showing all items

Request	Payload	Status code	Response received	Error	Timeout	Length
1	admin	200	1			185
2	password	200	1			185
3	Password	200	1			185
4	Password!	200	2			185
5	p@$$w0rd!	200	3			185
6	P@$$w0rd!	200	2			185
7	Password123!	200	2			189

Request Response

Pretty Raw Hex Render

```
1  HTTP/1.1 200 OK
2  Server: Werkzeug/3.1.3 Python/3.13.7
3  Date: Mon, 06 Oct 2025 12:26:34 GMT
4  Content-Type: text/html; charset=utf-8
5  Content-Length: 16
6  Connection: close
7
8  Login Successful
```

Fig 14: The response captured with successful login

This completes the Phase-1 vulnerable app testing. In this phase I demonstrated an attacker strategy that exploits weak password policies and application vulnerabilities: a lack of server-side complexity enforcement allowed trivial passwords to be set; an intentionally exposed SQLi endpoint enabled extraction of stored hashes (which were MD5 and quickly cracked offline with Hashcat); and absence of rate-limiting or account lockout permitted automated online guessing (Burp Intruder) to succeed.

Together these findings show how weak password storage and missing protections make credential compromise straightforward for an attacker.

Testing of Phase-2

I created and activated a Python virtual environment like in the Phase-1 testing and launched the Phase-2 web application from src/app_phase2_fixed.py using **python src/app_phase2_fixed.py**

```
┌──(venv)─(kali㉿kali)-[~/krutanic/projects/chakradhar-minor]
└─$ python src/app_phase2_fixed.py
 * Serving Flask app 'app_phase2_fixed'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.46.128:5001
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 143-262-970
```

Fig 15: Launching the Phase-2 web app

I opened the app in Firefox at http://192.168.46.128:5001 and verified a successful login using the seeded credentials.

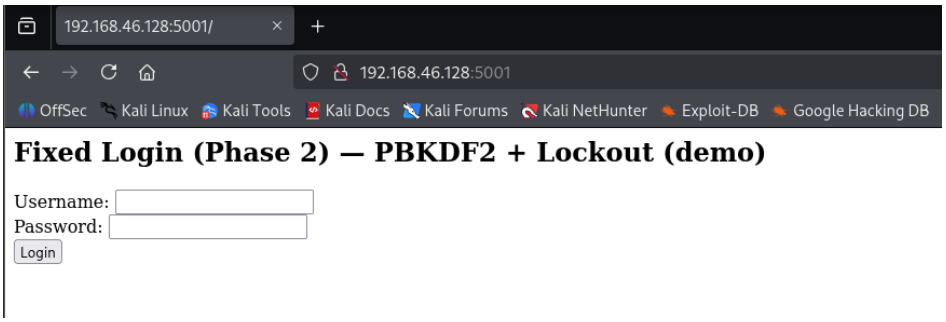

Fig 16: Login page of phase-2 web app

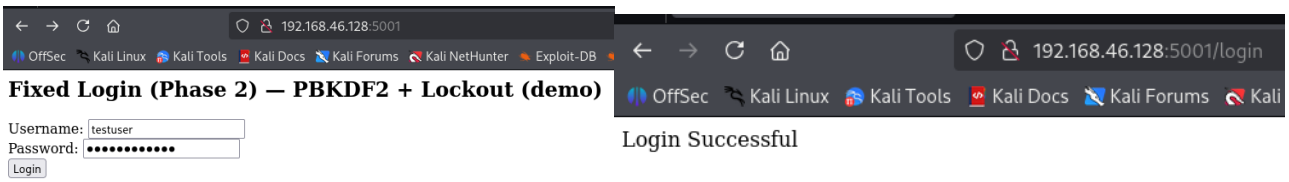

Fig 17: Successful authentication of the login page

I then attempted to set a weak password (pass); the server rejected it according to the configured requirements. I next set a compliant password (Password123!) and received ACCEPTED:password_set.

```
┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=Pass' http://192.168.46.128:5001/set_password
REJECTED:min_length,no_digit,no_special

┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=pass' http://192.168.46.128:5001/set_password
REJECTED:min_length,no_upper,no_digit,no_special

┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=Pass!' http://192.168.46.128:5001/set_password
REJECTED:min_length,no_digit
```

Fig 18: curl commands used to set passwords and getting rejected output because of the config req.

```
┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=Password123!' http://192.168.46.128:5001/set_password
ACCEPTED:password_set
```

Fig 19: curl command used and the password being accepted as it met the config req.

To test online protections, I attempted repeated logins manually: after three failed attempts I could still try, but following the third failed attempt the account was locked and subsequent login attempts returned Account locked. Try again after 300 seconds - this lockout behaved as designed and will materially increase an attacker's cost and time for brute-force attacks by limiting rapid repeated attempts.

```
┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=admin' http://192.168.46.128:5001/login
Login Failed

┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=administrator' http://192.168.46.128:5001/login
Login Failed

┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=Password123' http://192.168.46.128:5001/login
Login Failed

┌──(kali㉿kali)-[~]
└─$ curl -s -X POST -d 'username=testuser&password=P@ssword123' http://192.168.46.128:5001/login
Account locked. Try again after 287 seconds.
```

Fig 20: curl commands used to login to the web app and enforcing the lockout mechanism

Using the lab-only SQLi extraction endpoint I retrieved the stored credential hash; it was a PBKDF2-SHA256 entry in the Werkzeug format and I saved it in a file - strong_hash_john2.txt

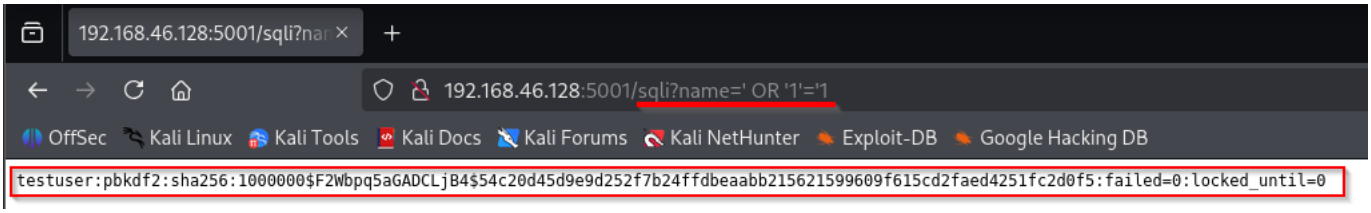

Fig 21: The PBKDF2-SHA256 hash

I attempted offline cracking with John the Ripper, but the tool did not accept or was unable to crack the hash file as provided (no successful plaintext recovered within practical time).

```
┌──(kali㉿kali)-[~/krutanic/projects/chakradhar-minor]
└─$ john --wordlist=/usr/share/wordlists/rockyou.txt --format=PBKDF2-HMAC-SHA256 strong_hash_john2.txt

Using default input encoding: UTF-8
No password hashes loaded (see FAQ)
```

Fig 22: john the ripper command and it failing to recognise the hash

```
┌──(kali㉿kali)-[~/krutanic/projects/chakradhar-minor]
└─$ john --format=PBKDF2-HMAC-SHA256 --test

Will run 6 OpenMP threads
Benchmarking: PBKDF2-HMAC-SHA256 [PBKDF2-SHA256 128/128 AVX 4x]... (6xOMP) DONE
Speed for cost 1 (iteration count) of 1000
Raw:    6190 c/s real, 1053 c/s virtual
```

Fig 23: Computational Speed Benchmark of PBKDF2-HMAC-SHA256 Hashing

A benchmarking test was performed using John the Ripper on the PBKDF2-HMAC-SHA256 hashing algorithm configured with 1,000,000 iterations. The benchmark reported approximately 6190 c/s (checks per second) for 1,000 iterations, which scales down to ~6.19 c/s for 1,000,000 iterations [in my case]. At this rate, running a standard wordlist such as rockyou.txt (≈14 million entries) would take around 28 days to complete on the test system. This demonstrates that the PBKDF2 implementation provides strong resistance against brute-force and dictionary attacks due to its high computational cost. While weak or common passwords could still be cracked, the high iteration count significantly increases the effort required, thereby strengthening the security of stored passwords.

These results show the Phase-2 mitigations (server-side complexity enforcement, PBKDF2 salted hashing, and account lockout) are functioning and materially increase resistance to both online guessing and fast offline cracking. In short, Phase-2 raised the attacker's cost and time significantly — which is the intended outcome of these mitigations.

Results & Conclusion

This project successfully demonstrated the effectiveness of modern password-security controls using a reproducible lab environment with two Flask + SQLite applications. In Phase-1, weak passwords and MD5-hashed credentials were easily compromised through SQL injection (used here as a controlled extraction method) and offline cracking with John the Ripper and Hashcat. For Phase-2, the application implements mitigations — server-side password complexity enforcement, salted & iterated PBKDF2 storage, and account lockouts — which significantly increased attacker cost and prevented trivial compromises.

Note: SQL injection was used deliberately in a controlled, isolated lab to obtain stored hashes for offline analysis; the exploitation of SQLi vulnerabilities itself is outside the primary scope of this project and all tests were run on isolated test builds.

Key findings:

- MD5-hashed passwords are rapidly recovered, underscoring the risk of fast, unsalted hashing.
- PBKDF2 with a sufficiently high iteration count substantially increases the cost of offline attacks and made practical recovery infeasible in this lab.
- Server-side complexity checks together with account lockouts reduce the success of online guessing attacks.

Next steps / recommendations:

- Adopt modern, slow, salted hashing algorithms (Argon2, bcrypt, or PBKDF2 with tuned work factors).
- Enforce robust server-side password policies and deploy multi-factor authentication.
- Implement IP/rate limiting, consistent logging, and monitoring to detect and mitigate automated attacks.
- Extend the lab to evaluate additional scenarios (credential reuse, phishing, and real-world attacker tooling) while keeping all testing in an isolated environment.

All experiments were conducted in an isolated VM lab and are fully reproducible with the supplied source code and test scripts.

References

[1] Hashcat, “Hashcat Wiki,” https://hashcat.net/wiki/doku.php?id=hashcat, accessed Oct. 6, 2025.