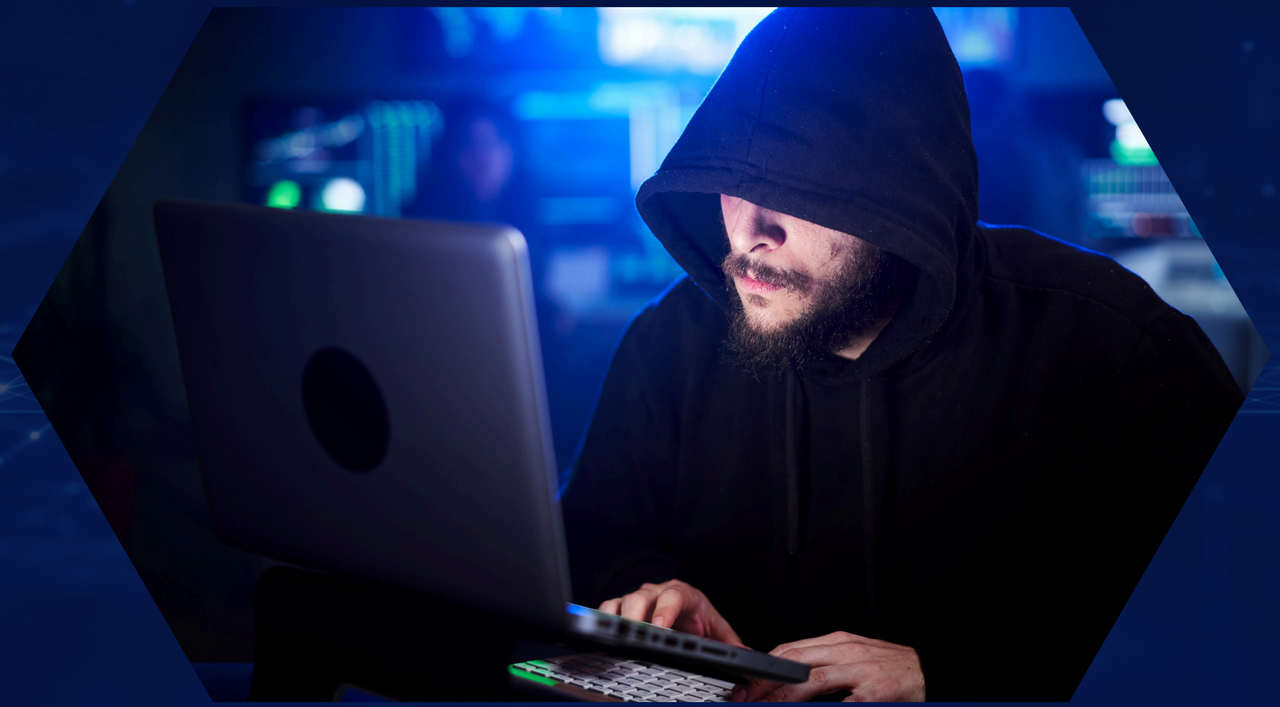
CODEALPHA INTERNSHIP

CYBER SECURITY

SECURE CODING REVIEW

Zwivhuya Tshutshu

Introduction



This review evaluates a Python-based login system for security vulnerabilities. The insecure version demonstrates poor practices that expose applications to exploitation. This report identifies weaknesses, provides remediation, and aligns recommendations with OWASP Top 10 standards.

Insecure Code

python

```python
# Insecure login code (example)
username = input("Enter username: ")
password = input("Enter password: ")

# Hardcoded credentials (bad practice!)
if username == "admin" and password == "12345":
    print("Login successful")
else:
    print("Invalid credentials")
```

Findings in Insecure Code

- Hardcoded Credentials → attacker can guess credentials from source.
- Plaintext Passwords → no encryption or hashing applied.
- Weak Authentication → vulnerable to brute-force attacks, no lockout.

- No Input Validation → unsanitized input may lead to injection risks.
- No Secure Storage → sensitive data directly in codebase.

FINDINGS

⚠️ Vulnerabilities Identified:

1. Hardcoded Credentials
   - Storing admin/12345 directly in the code makes it easy to guess or reverse-engineer.

1. Plaintext Password Comparison
   - Passwords are stored and compared in plaintext, exposing them if compromised.

2. Weak Authentication Controls
   - No protection against brute-force login attempts.

3. Lack of Security Practices
   - No input validation, no environment-based configuration, no logging or monitoring.

Recommendations

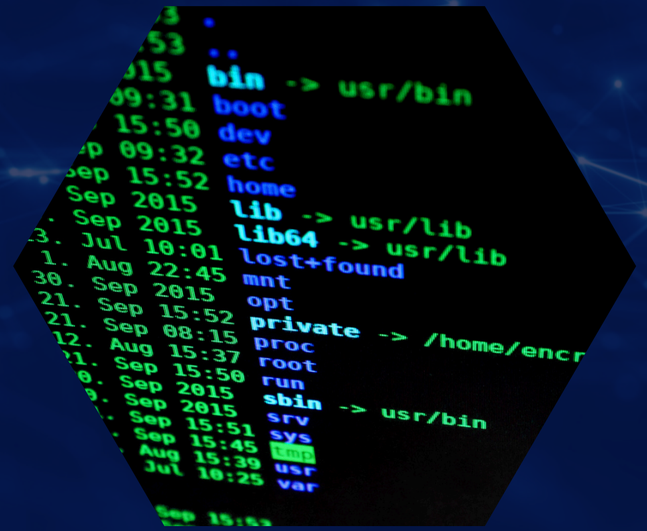
- ✅ Use Hashing for Passwords → Store only secure hashes (SHA-256 or bcrypt).
- ✅ Use Environment Variables → Prevent exposing secrets in source code.
- ✅ Brute Force Protection → Limit login attempts or introduce CAPTCHA.
- ✅ Input Validation & Logging → Track suspicious login attempts.
- ✅ Use Environment Variables for storing secrets.
- ✅ Hash Passwords with SHA-256, bcrypt, or Argon2.
- ✅ Implement Brute-Force Protection (limit attempts, add delay, or CAPTCHA).
- ✅ Input Validation to sanitize data.
- ✅ Secure Coding Practices (no hardcoding, logging suspicious activities).

SECURE CODE

```python
import hashlib
import os

# Store password securely in environment variable
# Example: export ADMIN_PASSWORD="StrongPassword123"
stored_hash = hashlib.sha256(os.environ.get("ADMIN_PASSWORD", "

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

# Hash user input before comparing
hashed_pw = hashlib.sha256(password.encode()).hexdigest()

if username == "admin" and hashed_pw == stored_hash:
    print("Login successful ✅")
else:
    print("Invalid credentials ❌")
```

The improved code:

- Removes hardcoded credentials.
- Stores admin password securely in environment variable.
- Uses SHA-256 hashing to protect password storage.
- Compares only hashed values to avoid plaintext exposure.



Benefits of Secure Version

- 🔒 No hardcoded secrets → attackers can’t extract passwords from code.
- 🔑 Hashed passwords → prevents plaintext exposure.
- 🛡️ Environment configuration → flexible & secure deployment.
- 🚫 Stronger defense → reduces brute-force and credential exposure risks.
- 📈 Aligned with OWASP best practices

FUTURE ENHANCEMENT

- Use bcrypt or Argon2 instead of SHA-256 for stronger hashing.
- Add Multi-Factor Authentication (MFA) for layered security.
- Implement secure logging for suspicious login attempts.
- Deploy account lockouts after multiple failed attempts.

Conclusion

TThe insecure login system demonstrated how poor coding practices expose applications to cyberattacks. By implementing password hashing, environment variable management, and secure authentication practices, the improved version aligns with industry best practices and significantly reduces the attack surface.

This review demonstrates that secure coding is not optional, but essential in modern software development.

THANK YOU