**Secure Application Programming CA 1**

**Programme:** BSHCCYBE4, BSHCSD4\_BSHCYB4

**Date:** 31st October (Week 7)

**Weighting:** 20% of Module Grade

**Objective**

This assessment aims to evaluate your ability to identify and remediate security vulnerabilities in a Java-based authentication and session management system. You will apply secure coding practices to fix critical flaws in a provided codebase.

**Scenario**

You are a junior security developer. Your senior developer has written a foundational AuthSystem class for a new web application but has been pulled onto another project. They have left you a note:

"I've started the auth system, but I know it has security holes. I need you to harden it before we go live. Focus on password storage, timing attacks, and session security. Use the principles we discussed in the labs."

**Your Task**

You are provided with a vulnerable AuthSystem.java file. Your task is to identify the security flaws and fix them by implementing secure coding practices.

**Create a new Java project and a class named**AuthSystem.java**with the starter code.**

**Part A: The Security Audit Report (5 Marks)**

*In the PDF report, list and briefly explain the security vulnerabilities you have identified in the provided code. Classify them based on the security principles they violate (e.g., Confidentiality, Integrity, Availability, Defence in Depth).*

**Part B: The Code Hardening (15 Marks)**

Refactor the AuthSystem.java class to address the critical flaws. Your implementation must demonstrate professional-grade security. You are required to:

1. Eradicate Plain Text Passwords: Implement salted password hashing using a strong, modern algorithm. (You must research and choose an appropriate algorithm beyond basic SHA-256).
2. Prevent User Enumeration: The login process must not leak information about whether a username exists. The response time for existing and non-existing users must be indistinguishable.
3. Implement Brute-Force Defence: Introduce a mechanism that temporarily locks an account after a certain number of consecutive failed login attempts.
4. Secure Session Management: Generate cryptographically strong, unpredictable session tokens. The validation of these tokens must also be secure.
5. Use Constant-Time Comparisons: Ensure all security-critical comparisons (like password hash matching) are performed in constant time to prevent timing attacks.

**Deliverables**

1. AuthSystem.java: Your fully refactored and secure Java class.
2. Report.pdf (Max 2 pages):
   * Vulnerability Analysis: A list of the vulnerabilities you found in the original code.
   * Remediation Explanation: A brief justification for your technical choices (e.g., why you chose a specific hashing algorithm, how you prevent timing attacks).
   * Trade-off Discussion: Briefly discuss one potential trade-off between security and usability that you considered in your design (e.g., related to account lockout).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Criteria | Excellent | Good | Satisfactory | Poor | Marks |
| **Security Audit (Report)** | Identifies all flaws with precise technical explanation. | Identifies most flaws with good explanation. | Identifies basic flaws with vague explanation. | Misses major flaws. | 5 |
| **Password Hashing** | Correctly implements salted, strong hashing (e.g., bcrypt, PBKDF2). | Uses salted SHA-256. | Uses hashing but without salt or with weak RNG. | Plain text or major flaw. | 4 |
| **Timing Attack & Enumeration** | Fully mitigates user enumeration and uses constant-time comparison. | Mitigates one aspect well, the other partially. | Attempts made but implementation is flawed. | No effective mitigation. | 3 |
| **Brute-Force Defence** | Correctly implements account lockout with a sensible threshold. | Implements lockout but with a logical flaw. | Basic attempt counter without lockout. | No implementation. | 2 |
| **Session Security** | Strong, random tokens and secure validation. | Improved tokens but validation is weak. | Tokens are still predictable. | Weak, predictable tokens. | 2 |
| **Code Quality & Correctness** | Code is clean, well-commented, and fully functional. | Code is functional but lacks clarity. | Code is messy or has runtime errors. | Code does not compile. | 4 |
| **Total** | | | | | 20 |