# CS 410 Project Two Security Report Template

## Instructions

Fill in the table in step one. In steps two and three, replace the bracketed text with your answer in your own words.

Identify where multiple security vulnerabilities are present within the blocks of C++ code. You may add columns and extend this table as you see fit.

| **Block of C++ Code** | **Identified Security Vulnerability** |
| --- | --- |
| string username, password; if (password == "123") { return 1; } | Hardcoded Password Authentication |
| using namespace std; | Unsafe Namespace Usage |
| void displayInfo() { cout << "Client Information:\\n"; // No session validation } | Unrestricted Data Access |
| void changeCustomerChoice() { cin >> clientNum; cin >> serviceChoice; // No input validation } | Unsanitized Input |
| while (choice != 3) { // Menu loop continues indefinitely } | Missing Session Management |
| const string CLIENT\_DATA\_FILE = "client\_data.txt"; | Insecure File Operations |
| Client& client = clients[clientNum - 1]; client.serviceType = newService; // No error handling } | Unsafe State Changes |
| void changeCustomerChoice() { // Changes made without audit trail } | Missing Audit Trail |

**Step 2: Explain the Security Vulnerabilities**

**1. Hardcoded Password Authentication**

Storing hardcoded credentials like "123" in source code makes the system vulnerable to reverse engineering and unauthorized access. Attackers can extract credentials by inspecting the code or binary. Additionally, plaintext passwords provide no protection against brute force attacks or password leaks.

**2. Unsafe Namespace Usage**

Using using namespace std; introduces potential naming conflicts and makes it harder to track the origins of functions. This can lead to ambiguous references and reduce the maintainability and security of the codebase.

**3. Unrestricted Data Access**

Displaying client information without validating the session or enforcing access control allows unauthorized users to view sensitive data. This is a privacy and compliance risk, as sensitive client data may be exposed without proper authorization checks.

**4. Unsanitized Input**

Accepting user input without validation (e.g., cin >> clientNum) opens the system to attacks like buffer overflows, invalid data injection, and system state corruption. This can crash the system or alter its intended behavior.

**5. Missing Session Management**

A lack of session management allows users to maintain indefinite access to the system, even after being idle. This increases the risk of session hijacking and unauthorized access through abandoned or inactive sessions.

**6. Insecure File Operations**

Storing sensitive client data (client\_data.txt) in plaintext without encryption or access restrictions leaves the data vulnerable to tampering and unauthorized access. Unsecured file paths may also lead to file inclusion attacks.

**7. Unsafe State Changes**

Directly modifying client data without error handling or transaction management can result in data corruption or loss. If an error occurs, the system lacks a rollback mechanism to revert to a safe state.

**8. Missing Audit Trail**

Without logging or tracking user actions, the system cannot detect or investigate unauthorized access, system misuse, or changes to sensitive data. This reduces accountability and limits forensic capabilities.

**Step 3: Describe Recommendations to Fix Vulnerabilities**

**1. Secure Authentication Implementation**

• Use hashed passwords (e.g., bcrypt or Argon2) instead of plaintext.

• Store credentials in a secure database or configuration file.

• Add login attempt limiting and multi-factor authentication.

• Implement proper session management to restrict access.

**2. Safe Namespace Usage**

• Avoid using namespace std; globally.

• Use explicit namespace qualifiers (e.g., std::cout).

• Implement proper scope management to prevent naming conflicts.

**3. Implement Access Controls**

• Validate session activity before displaying sensitive data.

• Enforce role-based access control (RBAC).

• Mask sensitive information in logs or outputs.

**4. Input Validation and Sanitization**

• Add input validation and sanitization to handle invalid or malicious inputs.

• Restrict the length of inputs to prevent buffer overflows.

• Validate inputs against expected formats (e.g., numeric ranges).

**5. Session Management**

• Implement session timeouts to invalidate inactive sessions.

• Require reauthentication after prolonged inactivity.

• Store session information securely and clean up expired sessions.

**6. Secure File Operations**

• Encrypt sensitive data files to prevent unauthorized access.

• Use secure directories with proper permissions.

• Implement atomic operations to reduce data corruption risks.

**7. Safe State Management**

• Add error handling and validation before modifying state.

• Implement a transaction-based approach with rollback capabilities.

• Ensure data integrity by checking for inconsistencies before saving changes.

**8. Audit Trail Implementation**

• Log all critical operations (e.g., logins, data changes) with timestamps.

• Store logs in a secure, immutable format.

• Regularly review and rotate logs to detect and prevent unauthorized activities.