# CS 410 Project Two Security Report - Joseph Dengler

## Instructions

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

1. **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**  **[Project1Unsecure.cpp]** | **Identified Security Vulnerability** | **Block of Secured C++ Code**  **[Project2Secure.cpp]** |
| --- | --- | --- |
| // Global user information (Unchanged structure, altered strings)  string client1 = "Vincent Vega"; int service1 = 1;  string client2 = "Jules Davis"; int service2 = 2;  string client3 = "Joey Frank"; int service3 = 1;  string client4 = "Johnny Apple"; int service4 = 1;  string client5 = "Beatrix Kiddo"; int service5 = 2; | **Hardcoded user data**: User information is directly stored in variables instead of using a secure storage method like a database or a file. This exposes sensitive information in the source code. | struct Client {  string name;  int service;  };  // Store client data dynamically in a vector  vector<Client> clients = {  {"Vincent Vega", 1},  {"Jules Davis", 2},  {"Joey Frank", 1},  {"Johnny Apple", 1},  {"Beatrix Kiddo", 2}  }; |
| // "Super secure" password  string passcode = "123"; | **Hardcoded password**: Storing passwords in plaintext inside the source code makes it easy for attackers to retrieve them. Use hashed and salted passwords with secure authentication. | // Function to hash passwords using std::hash (not cryptographically secure)  string hashPassword(const string &password) {  hash<string> hasher;  return to\_string(hasher(password)); // Converts hash to string  }  // Secure user database (hashed passwords)  unordered\_map<string, string> userDatabase = {  {"admin", hashPassword("abc123")},  {"user1", hashPassword("test")}  }; |
| int CheckUserPermissionAccess() {  string userEntry, passEntry;  cout << "Enter username:\n";  cin >> userEntry;  cout << "Enter password:\n";  cin >> passEntry;  if (passcode.compare(passEntry) == 0) {  return 1;  } else {  return 2;  }  } | **Username is ignored**: The program asks for a username but does not verify or check it.  Insecure password comparison: Uses simple string comparison without hashing, making it vulnerable to brute-force and dictionary attacks.  Implement proper authentication with password hashing. | int CheckUserPermissionAccess() {  string userEntry, passEntry;  int attempts = 0;  while (attempts < 3) {  cout << "Enter username:\n";  cin >> userEntry;  cout << "Enter password:\n";  cin >> passEntry;  string hashedPass = hashPassword(passEntry); // Hash the entered password before comparison  // Validate username and password  if (userDatabase.find(userEntry) != userDatabase.end() &&  userDatabase[userEntry] == hashedPass) {  return 1;  } else {  cout << "Incorrect username or password. Try again.\n";  attempts++;}}  cout << "Too many failed login attempts. Exiting program.\n";  exit(1);} |
| void DisplayInfo() {  cout << " Client Name | Service Selected" << endl;  cout << " ---------------|------------------" << endl;  cout << "1. " << client1 << " - Plan " << service1 << endl;  cout << "2. " << client2 << " - Plan " << service2 << endl;  cout << "3. " << client3 << " - Plan " << service3 << endl;  cout << "4. " << client4 << " - Plan " << service4 << endl;  cout << "5. " << client5 << " - Plan " << service5 << endl;  } | **No input validation**: The program does not check if clientNum or newPlan are within valid ranges or if the input is a valid integer. This can lead to unintended behavior or crashes. | void DisplayInfo() {  cout << " Client Name | Service Selected" << endl;  cout << " ---------------|------------------" << endl;  for (size\_t i = 0; i < clients.size(); i++) {  cout << i + 1 << ". " << clients[i].name << " - Plan " << clients[i].service << endl;  }  } |
| void ChangeCustomerChoice() {  int clientNum, newPlan;  cout << "Enter client number to modify:\n";  cin >> clientNum;  cout << "New service choice (1 = Brokerage, 2 = Retirement)\n";  cin >> newPlan;  switch (clientNum) {  case 1: service1 = newPlan; break;  case 2: service2 = newPlan; break;  case 3: service3 = newPlan; break;  case 4: service4 = newPlan; break;  case 5: service5 = newPlan; break;  default: cout << "Invalid client number.\n"; break;  }  } | **Hardcoded indices**: Clients are manually referenced instead of being stored in a dynamic structure (e.g., vector or map), making the program less scalable.  **No input validation**: If the user enters a non-integer or out-of-range value, it could cause crashes or unexpected behavior. Use input validation and store data dynamically. | void ChangeCustomerChoice() {  int clientNum, newPlan;  cout << "Enter client number to modify (1-" << clients.size() << "): ";  while (!(cin >> clientNum) || clientNum < 1 || clientNum > clients.size()) {  cout << "Invalid input. Enter a number between 1 and " << clients.size() << ":\n";  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n');}  cout << "New service choice (1 = Brokerage, 2 = Retirement): ";  while (!(cin >> newPlan) || (newPlan != 1 && newPlan != 2)) {  cout << "Invalid input. Enter 1 or 2:\n";  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n'); }  clients[clientNum - 1].service = newPlan;  cout << "Service updated successfully.\n";} |
| int main() {  cout << "Welcome to the unsecure Investment Portal!\n";  int authResult;  do {  authResult = CheckUserPermissionAccess();  if (authResult != 1) {  cout << "Incorrect Password. Try again!\n";  }  } while (authResult != 1); | **No brute-force protection**: There is no limit on login attempts, allowing infinite retries for brute-force attacks.  Implement a max attempt limit with a cooldown period. | int main() {  cout << "Welcome to the Secure Investment Portal!\n";  int authResult = CheckUserPermissionAccess(); // Now has a max attempt limit |
| int option;  do {  cout << "\nMENU:\n";  cout << "1 - Show Clients\n";  cout << "2 - Modify Client Service\n";  cout << "3 - Exit\n";  cout << "Choice: ";  cin >> option;  cout << "Processing...\n";  if (option == 1) {  DisplayInfo();  } else if (option == 2) {  ChangeCustomerChoice();  }  } while (option != 3);  cout << "Goodbye.\n";  return 0;  } | **No menu input validation**: If the user enters a non-integer value, it could cause unexpected behavior or crash the program. Validate input to ensure only expected values are processed. | int option;  do {  cout << "\nMENU:\n";  cout << "1 - Show Clients\n";  cout << "2 - Modify Client Service\n";  cout << "3 - Exit\n";  cout << "Choice: ";  while (!(cin >> option) || (option < 1 || option > 3)) {  cout << "Invalid choice. Please select a valid menu option (1-3):\n";  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  } |

1. **Explain the *security vulnerabilities* that are found in the blocks of C++ code.**
2. **Describe recommendations for how the security vulnerabilities can be fixed.**

**Hardcoded User Data**

In Project1Unsecure.cpp the client information is stored directly in variables and exposes sensitive data in the source code. This approach makes the program static, requiring manual updates to add or modify client details. If an attacker gains access to the source code through reverse engineering, they could extract and manipulate user information. Additionally, embedding user data within the source code prevents secure data management practices. Secure DB practices such as encryption and access controls are essential for protecting sensitive information.

The improved Project2Secure.cpp, mitigates vulnerability by dynamically storing client data in a vector. This approach allows for more flexible and scalable data management while keeping user information separate from the source code. In a real-world scenario, the client list could be retrieved from a secure database to ensure that unauthorized access to the source code does not expose data. Separating data from application logic is a fundamental principle in secure coding practices, preventing direct access to sensitive information.

**Hardcoded Password**

In Project1Unsecure.cpp, the password is stored in plain text within the code. This makes it easily retrievable to attackers who gain access to the file or decompile the binary. Hardcoded passwords create a severe security risk because they cannot be changed without modifying and recompiling the source code, and they are vulnerable to static analysis attacks. This method does not provide any encryption or hashing which could allow attackers to use the password directly if obtained.

To address this issue, Project2Secure.cpp replaces the plaintext password with a hashed and stored value in an unordered\_map. When a user enters a password, it’s hashed and compared against the stored hash rather than being checked directly. While std::hash is not the most secure hashing method, it significantly improves security over plaintext storage. A more secure approach would involve using a cryptographic hashing function like SHA-256 or bcrypt to prevent brute-force attacks and password leaks.

**Insecure Authentication**

In Project1Unsecure.cpp, the authentication function collects a username but does not verify it, making the username input redundant. The program only checks the password against a hardcoded value, meaning anyone who enters the correct password is granted access. This flaw allows unauthorized users to gain access if the password is leaked. Also the lack of login attempt limitations leaves the program vulnerable to brute-force attacks, where an attacker can repeatedly attempt passwords until access is granted.

The improved implementation in Project2Secure.cpp ensures that both the username and password are validated against a secure ‘user database’. Passwords are hashed before storage and verification to reduce risk of exposure. The authentication system also includes a limit of three failed login attempts to discourage brute-force attacks by terminating the program after multiple incorrect entries.

**No Input Validation in Client Display Function**

The original DisplayInfo function in Project1Unsecure.cpp presents static-hardcoded client information limiting the program's ability to adapt to changes dynamically. Additionally, it does not validate the input data, meaning if any unexpected values were inserted into the source code, they could be displayed incorrectly or cause unintended behavior.

In Project2Secure.cpp, the function is redesigned to iterate through a vector of Client structures, dynamically displaying client information. This modification prevents the risks associated with hardcoded data while improving flexibility and scalability. By structuring the data dynamically, the program can efficiently handle updates and modifications without requiring direct source code changes.

**No Input Validation in Service Modification**

The ChangeCustomerChoice function in Project1Unsecure.cpp lacks input validation which allows users to enter invalid client numbers or service selections. Without proper checks, an out-of-range input could cause undefined behavior, crashes, or even memory corruption. This lack of validation makes the program vulnerable to accidental misuse or intentional exploitation.

In Project2Secure.cpp input validation is introduced using cin error handling and numeric\_limits. The program ensures that users can only select valid client numbers and service plans by clearing invalid input and re-prompting the user. This prevents crashes and strengthens the program's stability, making it more resilient against incorrect or malicious input.

**No Brute-Force Protection**

The original authentication system in Project1Unsecure.cpp does not limit login attempts, allowing unlimited retries. This flaw enables brute-force attacks, where an attacker can repeatedly try different passwords until they gain access. Without a mechanism to restrict repeated failures, an attacker could automate password guessing to compromise the system.

In Project2Secure.cpp, a login attempt limit is enforced, restricting users to three failed attempts before terminating the program. This effectively prevents brute-force attacks by making it impractical for attackers to attempt numerous password combinations. Further improvements could include account lockouts or time-based cooldowns to enhance security.

**No Menu Input Validation**

In Project1Unsecure.cpp, the main menu does not validate user input, allowing users to enter invalid or non-numeric values. If a user enters a non-integer, such as a string, the program may enter an infinite loop or crash due to an unhandled input error. This flaw affects usability and security, as unintended inputs could disrupt program execution.

The revised version in Project2Secure.cpp ensures input validation for menu selections. The program checks if the input is an integer within the expected range and clears invalid entries from the input stream before re-prompting the user. This prevents crashes and ensures the program runs smoothly, improving both security and user experience