# 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** |
| --- | --- |
| #include <iostream>  #include <vector>  using namespace std; | No issues |
| string username, password;  string name1 = "Bob Jones";  string name2 = "Sarah Davis";  string name3 = "Amy Friendly";  string name4 = "Johnny Smith";  string name5 = "Carol Spears";  int num1 = 1;  int num2 = 2;  int num3 = 1;  int num4 = 1;  int num5 = 2;  int choice = 0;  int answer = 0;  int changechoice = 0;  int newservice = 0;  int completed = 0; | Global variables can be potential security issues due to their access control. |
| void ChangeCustomerChoice() {    cout << "Enter the number of the  client that you wish to  change\n";    cin >> changechoice;    cout << "Please enter the client's new  service choice (1 =  Brokerage, 2 =  Retirement)\n";    cin >> newservice;    if (changechoice == 1) {      num1 = newservice;    }    else if (changechoice == 2) {      num2 = newservice;    }    else if (changechoice == 3) {      num3 = newservice;    }    else if (changechoice == 4) {      num4 = newservice;    }    else if (changechoice == 5) {      num5 = newservice;    }    return;  } | Input restrictions and validation for the user’s responses  A switch statement could have been used for cleaner code. Best practice is to have a final ‘else’ statement for data validation and error handling. |
| int CheckUserPermissionAccess() {    password = "";    cout << "Enter your username:\n";    cin >> username;    cout << "Enter your password:\n";    cin >> password;    if (password == "123"){      completed = 1;    }    else{      completed = 2;    }    return completed;  } | Usernames and passwords should not be hard coded in a program.  Lack of Input restrictions/requirements for the username and password.  No login limits for failed attempts |
| void DisplayInfo() {    cout << "  Client's Name    Service  Selected (1 = Brokerage, 2 =  Retirement)" << endl;    cout << "1. " << name1 << " selected  option " << num1  << endl;    cout << "2. " << name2 << " selected  option " << num2 << endl;    cout << "3. " << name3 << " selected  option " << num3 << endl;    cout << "4. " << name4 << " selected  option " << num4 << endl;    cout << "5. " << name5 << " selected  option " << num5 << endl;    return;  } | Information should be stored in a database or a data structure for efficient access. |
| int main() {    cout << "Reverse Engineered by Raphael  Coloma" << endl;    cout << "Hello! Welcome to our  Investment Company\n";    while(answer != 1){      answer = CheckUserPermissionAccess();      if(answer != 1){        cout << "Invalid Password. Please  try again\n";      }    }    while(choice != 3){      cout << "What would you like to  do?\n";      cout << "DISPLAY the client list  (enter 1)\n";      cout << "CHANGE a client's choice  (enter 2)\n";      cout << "Exit the program.. (enter  3)\n";        cin >> choice;        cout << "You chose " << choice  << endl;      if (choice == 1) {        DisplayInfo();      }      else if (choice == 2) {        ChangeCustomerChoice();      }    }    return 0;  } | The variable 'answer' should be local  Input validation should be done to ensure only integers 1-3 can be entered  A switch statement could have been used for cleaner code. Best practice is to have a final ‘else’ statement for data validation and error handling. No hard code exit written for option 3. |

Explain the *security vulnerabilities* that are found in the blocks of C++ code.

Global Variables

These can be potential security issues due to the access control they are given. They also make it difficult to keep track of the flow of data. Since they are global, any function can change the data they hold and thus make it harder to track when/where the data is changed. Variables should be limited to the scope of where they are used/needed and if another function requires that data, it should be passed to the function to maintain abstraction.

No Input Validation

There are a couple of instances in the application where the user is prompted for input. The application should validate the information received by the user as improper input received by the application can lead to unexpected behavior or even crashing. As developers, we cannot expect that the correct information will always be inputted by the user.

Lack of Input Restrictions

Input fields are notorious for vulnerabilities and exploits. Hackers use input fields to manipulate applications by injecting malicious code after overflowing the buffer. All input fields have a buffer limit and once it is exceeded, the hacker will have access to the executable code. Input restrictions ensure the user only inputs data in the expect format with defined limits. This safeguards the application from injection attacks through input fields.

Lack of Input Requirements

Input requirements are used to protect users and their data. Input requirements are primary used bolster passwords. By setting input requirements such as a minimum character requirement or requiring the use of a special character, the user’s password becomes more difficult to brute force hack. Though this is not a vulnerability of the application itself, it is a means to protect the application’s users and give them piece of mind.

‘switch’ vs ‘if else’ Statements

Though both accomplish the same result, the use of a switch statement would be the better option. While this may not be seen as a vulnerability by some, ‘if else’ statement conditions are susceptible to programing errors and can lead to unexpected behavior if not properly coded. In this instance, ‘switch’ statements would be ideal since the comparisons being made are between the user inputs their equivalency to the option numbers. ‘Switch’ statements are also more efficient when compiled and executed.

Hard Coding Login Information

Login information should never be hard coded into the application. Hackers who gain access to the application source code can immediately know the login information for its users. Even without the source code, having login information hard coded into the application is susceptible to manipulation through malicious attacks. Login information should be at the very least stored on a separate database that is only reference when needed by the application. To increase the protection, the database should be encrypted.

Lack of Login Limits

Login limits protect the users from brute force hacking. Without login limits, hackers can write a script that can cycle through all the possible combinations of characters to figure out a user’s login information. Having a login limit to suspend a user’s account or disable account login for an extend amount of time will deter hackers to use brute force methods.

Lack of Data Structure/Database for stored information

Like the “Hard Coding Login Information” vulnerability, not using a database or at the very lease a data structure to hold stored information can be problematic. By hard coding and assigning individual variables to all the pieces of data, the application will require more memory than needed. This will also make it easier for hackers to determine variable values as I did reverse engineering the code.

1. **Describe *recommendations* for how the security vulnerabilities can be fixed.**

To resolve the global variables security vulnerability, I would use local variable and passes parameters between functions as needed. Again, variables should be limited to the scope of where they are used/needed. I am not saying that the use of global variables is bad, but this applications declares all its variables as global variables and it is not needed. The username and password should never be global variables and the are a couple of variables that are only used within a single function.

Input Validation and input restrictions can be resolved in the same manner. As the developer, you know what kind of information is needed from the user. To fixed these security vulnerabilities, establish input validation and restrictions on the input fields. Limit what kind of data the user can input – if its supposed to be a date, format the input to only receive dates. Once the input is entered, the application should validate that the information submitted correlates with what the application expects. For this application, it requires requests for integer inputs with a range from 1-5. By checking the input to see if it is an integer within the specified range and rejecting everything else, I can eliminate this vulnerability.

Again, not having input requirements is not a security vulnerability, but having them will bolster the application’s security. In the context of this application, this would not work since the password is hard coded into the application. For other applications that require creating a login account, I would first set input requirements users must have when creating a password. I would then check the password after it has been entered to determine if it meets those requirements. If it does, the password is saved, and the user can login with their specified credentials. If it does not, the user will be notified and prompted to enter to try again.

This vulnerability is an easy fix. I would just swap the ‘if else’ statements with switch statements. The prompts will stay the same and the ‘switch’ statement would use the same options for its branches. For this application, since we know that the code works and that there are no errors, it is not entirely necessary, but as mentioned, it will help to keep the code clean and promote optimization and efficiency.

For the hard coded login information, unfortunately I will not be able to fix it without doing some major overhauls on the application. First, I would need to code a means of creating user accounts. Then that information needs to be stored on a separate database file. Ideally, I would like to encrypt the information, maybe through a hash map. The application would then need to be modified to check both the username and password and verify them based on the database file before granting access to the application.

Incorporating basic login limits is easy to set up. It would be a simple counter with an ‘if’ statement. After each login attempt, the application would compare the number of failed attempts to the set limit. If that maximum is reached, the application is terminated. This is a simple fix, but the ideal implementation would be to attach a suspension Boolean variable to the created user accounts. Then when the login limit is reached the suspension variable for the user account will be set to true and the user will not be able to login until an account with high privileges unsuspends the user account.

Storing information is an important part of any application. Unfortunately, with this application the information is stored using individual variables with no relations. Again, this would require some major adjustments to incorporate, but I would create a client object class and create client objects for each of the clients with attributes to store their respective information. This would keep the code clean and easy to read as well as optimize the memory.