**7-2: Project Two Security Report**

Dylan Jeffery

SNHU

CS-410

Brian West

2/20/2022

# CS 410 Project Two Security Report

## 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** |
| --- | --- |
| getline(cin, user); | Input should be verified as there are some ways to trigger a buffer overflow in the user input. I would like to see this tied to an SQL database with protection from injection in the final build. |
| if (password == 123) | It is improper to store a password in source like this, in addition to the password being the same for all users, a log in with appropriate user privileges linked to a database would be a great way to handle this, protection from SQL injection must also be considered. |
| cin >> option;  ……  cin >> client;  cout << "Please enter the client's new service choice (1 = Brokerage, 2 = Retirement)" << endl;  cin >> service;  changeCustomerChoice(serviceChoice, client, service); | These cin statements are vulnerable to overflow. |
| else {  cout << "Invalid Password. Please try again\n";  password = '\0';  checkUserPermission(); | This is actually a measure of protection I implemented against invalid password entry previously. This handles all issues related to handing the program password information with bad values. |
|  |  |

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

Regarding the getline() block, much is accounted for by utilizing the getline() function here, however we could run into issues if the user inputs a value that passes this function information without a delimiting char. I am not 100% sure how to fix it but this may have to do with std::getline overloads for std::istream… std::getline(std::cin, str) and std::getline(std::cin, str, delim)… to store data into str (a std::string), overwriting its existing contents. This string would then require scrubbing/verification before it is passed to any database or function.

As for the password block, it is improper storing a password in the source code, and additionally it is simply bad practice to have a single password for all users. A policy of least privilege is important in any system that other people have access to.

Plain cin statements can be dangerous, as once the user logs in, they may pass values too large for our buffer. To account for this, I have implemented simple logic to clear the buffer and ignore the previous input. (Like so)

if (!(cin >> client)) {

cin.clear();

cin.ignore(INT8\_MAX, '\n');

cout << "invalid input" << endl;

loopMenu = 1;

}

Finally, our last block is a new addition focused on handling bad values a user could be sending. This prevents characters being passed to int variables etc.

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

Many of these have been fixed to some degree already. The first two vulnerabilities are both tied to the fact that our login isn’t tied to any real validation. To resolve these, I would like to see user logins tied to an SQL database. Storing the value of a password plainly in the code is generally bad form, as having the same login credentials for everyone is also bad form. Each unique user should have unique credentials, that are tied to unique privileges. In the case of an SQL database implementation, strings will then require validation to prevent injection attacks. The main issue with the first getline() is that in some cases a user may pass information lacking any delimiting char at the end. This may result in buffer issues.

The major changes in this deployment are modifications to the cin lines taking user values for {option, service, client}. These input statements were vulnerable to overflow and so we must account for this event.

cin >> client

becomes

if (!(cin >> client)) {

cin.clear();

cin.ignore(INT8\_MAX, '\n');

cout << "invalid input" << endl;

loopMenu = 1;

}

And likewise for other cin statements. This provides us a few extra steps in our control flow. In the case that a user inputs values greater than INT\_MAX, this logic is triggered and the cin buffer is cleared, and the last input is ignored. The console then informs the user they are inputting invalid data and re-enters our while loop at the menu. This is done for each dangerous cin statement.