# 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** |
| --- | --- |
| std::string userName;  std::string password = "123"; // got this from looking at the strings in the original .o file and trial and error of inputting in the program  int answer;  int choice;  int changeChoice;  int newService;  int num1 = 1; // nums are the values in the original program  int num2 = 2;  int num3 = 1;  int num4 = 1;  int num5 = 2;  std::string name1 = "Bob Jones"; // names are the values in the original program  std::string name2 = "Sarah Davis";  std::string name3 = "Amy Friendly";  std::string name4 = "Johnny Smith";  std::string name5 = "Carol Spears"; | Password should not be hard coded. Also the password should be hashed and not stored as plain text |
| int CheckUserPermissionAccess() {  int match = 0;  // using string to eliminate need to limit size  std::string inputPassword;  std::cout << "Enter your username:" << std::endl;  // using string to eliminate need to limit size  std::cin >> userName;  std::cout << "Enter your password:" << std::endl;  std::cin >> inputPassword;  match = inputPassword.compare(password);  if (match == 0) {  return 1;  }  else {  return 2;  }  } | String dynamically allocates memory and elliminates need to limit size like on a char[] to prevent overflow  We are saving a password attempt into a variable. This should be cleared when done to ensure it doesn’t stay in memory  The inputPassword would normally be hashed before comparing to the saved password since it should be hashed as well so it isn’t stored in a database as plain text |
| void DisplayInfo() {  std::cout << " Client's Name Service Selected (1 = Brokerage, 2 = Retirement)" << std::endl;  std::cout << "1. " << name1 << " selected option " << num1 << std::endl;  std::cout << "2. " << name2 << " selected option " << num2 << std::endl;  std::cout << "3. " << name3 << " selected option " << num3 << std::endl;  std::cout << "4. " << name4 << " selected option " << num4 << std::endl;  std::cout << "5. " << name5 << " selected option " << num5 << std::endl;  } | No vulnerabilities |
| void ChangeCustomerChoice() {  std::cout << "Enter the number of the client that you wish to change" << std::endl;  // security vulnerability: need to make sure user input is an integer  // while it isn't exactly a security vulnerability the integer should have a coresponding name associated with it  std::cin >> changeChoice;  std::cout << "Please enter the client's new service choice (1 = Brokerage, 2 = Retirement)" << std::endl;  // security vulnerability: need to make sure user input is an integer  // while not exactly a security vulnerability the integer should be 1 or 2  std::cin >> newService;  // used a switch because it is easier to read than if else statements  switch (changeChoice) {  case 1:  num1 = newService;  break;  case 2:  num2 = newService;  break;  case 3:  num3 = newService;  break;  case 4:  num4 = newService;  break;  case 5:  num5 = newService;  break;  default:  break;  }  } | 2 integers are entered in this function. There needs to be input validation to ensure they are integers. Also while not exactly a security vulnerability, the numbers should be in a certain range. |
| int main()  {  std::cout << "Created by Brian Engel" << std::endl;  std::cout << "Hello! Welcome to our Investment Company" << std::endl;    // check for valid password and if invalid ask again  answer = CheckUserPermissionAccess();  while (answer != 1) {  std::cout << "Invalid Password. Please try again" << std::endl;  answer = CheckUserPermissionAccess();  }  // used do while since it always runs at least once  do {  std::cout << "What would you like to do?" << std::endl;  std::cout << "DISPLAY the client list (enter 1)" << std::endl;  std::cout << "CHANGE a client's choice (enter 2)" << std::endl;  std::cout << "Exit the program.. (enter 3)" << std::endl;  // security vulnerability: need to make sure user input is an integer  // while not exactly a vulnerability the choice should be 1-3  std::cin >> choice;  std::cout << "You chose " << choice << std::endl;  // used a switch because it is easier to read than if else statements | An integer is entered in this function. There needs to be input validation to ensure it is an integer. Also while not exactly a security vulnerability, the numbers should be in a certain range. |
| switch (choice) {  case 1:  DisplayInfo();  break;  case 2:  ChangeCustomerChoice();  break;  default:  break;  }  } while (choice != 3);  } | (cont.) |

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

There are major vulnerabilities in 2 main areas: Input validation and authentication.

Input validation should be done on the username and password strings to ensure that there are no attempts at injection attacks if they are being used by a database. Just reading them from the file itself isn’t really vulnerable to injection attacks, but all I had to do was look at the original .o file and I found the password very easily so that is another security vulnerability, and should uses a database for storage of these items. Input validation also needs to be done in input the menu choices. If a wrong integer is input, it is not that bad – nothing happens. But if a non-integer char is entered the program goes into an infinite loop. The authentication is kind of a mess in the fact the password and username are both stored in the file and easy to find. Also, the input password is stored to a variable that might stick around even after exiting the function for authentication, and the username is never matched up with a password.

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

To fix authentication vulnerabilities, I would first use a database to store username and password combinations and use encryption and hashing to ensure they were not in plain text in the database. Then when entering the username and password I would use parameterized queries instead of string concatenation when writing the queries to the database to ensure no SQL injection attacks could take place. Since this is a little bit out of the scope of the assignment, I did not fix this in the code. I did however take care of input validation when it comes to entering integers for the menus. I created a getInt function that serves the purpose of ensuring that a integer is entered and that it is in between the range specified by the arguments to the function. It actually reads in the user input as a string and then uses stoi() to convert to an integer. If the number is to large or small to fit in integer stoi() throws an exception out\_of\_range that is caught and then re-enters the input loop. If the user enters non number characters first stoi() throws invalid\_argument exception and re-enters input loop. If there is invalid characters after a number or if the number is not between the arguments I throw an invalid\_argument that is caught and re-enters the loop. Right now there is no limit to the amount of incorrect inputs that can be entered, but I could set up a limit since an infinite loop could be the source of a denial of service attack. One other vulnerability I fixed was in the CheckUserPermissionAccess function. After the userinput is compared to the password I clear out the userinput since it probably contains plain text of the users password.