# 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** |
| --- | --- |
| void ChangeCustomerChoice() {  int clientNumber, serviceNumber;  cout << "Enter the number of the client that you wish to change.\n";  cin >> clientNumber;  cout << "Please enter the client's new service choice (1 = Brokerage, 2 = Retirement)\n";  cin >> serviceNumber;  if (clientNumber == 1) {  clientChoices[clientNumber - 1] = serviceNumber;  }  else if (clientNumber == 2) {  clientChoices[clientNumber - 1] = serviceNumber;  }  else if (clientNumber == 3) {  clientChoices[clientNumber - 1] = serviceNumber;  }  else if (clientNumber == 4) {  clientChoices[clientNumber - 1] = serviceNumber;  }  else if (clientNumber == 5) {  clientChoices[clientNumber - 1] = serviceNumber;  }  } | Variables should be initialized before use.  Missing input validation for the variable clientNumber.  Missing input validation for the variable serviceNumber.  Potential for Buffer overflow if the value taken from input for the variable clientNumber is larger than the size of clientChoices.  The code should also ensure that the serviceNumber variable can only be assigned a value of 1 or 2. (This isn’t directly a security vulnerability, however, it is a flaw in the design and should be addressed). |
| The DisplayInfo() function does not contain any security vulnerabilities, since the size of clients and clientChoices never changes. When displaying client information, memory locations that would be considered out-of-bounds are never accessed. |  |
| int CheckUserPermissionAccess() {  string password = "";  string correctPassword = "123";  cout << "Enter your username: ";  cin >> username;  cout << "Enter your password: ";  cin >> password;  if (password.compare(correctPassword) == 0) {  return 1;  }  else {  cout << "Invalid Password." << endl;  cout << "Please try again." << endl;  return 2;  }  } | The password should not be hardcoded into the application.  The code does not check the input length for username and password variables.  Error messages should be more generic. |
| int main() {  int validUser = 0;  cout << "Created by: Justin Starr" << endl << endl;  cout << "Welcome to our Investment Company." << endl;  while (validUser != 1) {  validUser = CheckUserPermissionAccess();  }  int choice = 0;  while (choice != 3) {  cout << "What would you like to do?" << endl;  cout << "DISPLAY the client's list (enter 1)" << endl;  cout << "CHANGE a client's choice (enter 2)" << endl;  cout << "EXIT the program (enter 3)" << endl;  cin >> choice;  cout << "You chose " << choice << "." << endl;  if (choice == 1) {  DisplayInfo();  }  else if (choice == 2) {  ChangeCustomerChoice();  }  }  return 0;  } | There is no input validation for the variable choice.  This should also be checked to ensure that the user only inputs a value of 1, 2, or 3. (This isn’t directly a security vulnerability, however, it is a flaw in the design and should be addressed). |

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

Not initializing variables – This can lead to unexpected or undefined behavior, even if we are planning on receiving values from input and storing them in these variables at a later point in time. Using uninitialized variables can cause a program to abruptly terminate or other unexpected behavior, and a developer should never assume that variables automatically start with a zero or null value. Initializing variables when they are declared helps reduce the risk of encountering these problems, which can become hard to diagnose.

Missing input validation for integer variables – Not validating user input can lead to unexpected or undefined behavior. For example, when reading input from a user and storing it into an integer variable, entering things such as integer values that are too large (greater than the variable type’s numerical limits) will cause unexpected behavior. This could potentially lead to issues such as causing a program to crash, data corruption, or even a Denial-of-Service due to resource consumption. They could also cause buffer overflows or logic errors that make the program behave unpredictably.

Not validating user input length – This fits into the category of not validating user input because entering too many characters can also lead to undefined or unexpected behavior. Entering too many characters can lead to buffer overflow attacks because more data could be written to a buffer than it can possibly handle, which can overwrite the memory that is stored next to it.

Buffer overflows – This occurs when invalid memory locations are accessed. This can happen from a variety of causes, such as security vulnerabilities previously mentioned. Also, it can occur by attempting to write to locations that are out-of-bounds. This results in undefined or unexpected behavior, and could potentially expose critical or sensitive information about the application or system. This can often be exploited by attackers to execute arbitrary code, which can enable them to gain control over a program’s execution or even an entire system.

Another security vulnerability is that passwords should not be hard coded into the source code. This is because anyone who has access to the source code would be able to see the password required to be granted access to the application’s primary functionality. Also, even without the source code, any experienced developer could use a hex editor to view the binary file for the application, and acquire from it, the password for the application. This would lead to further exposure of application data. Both usernames and passwords should be stored somewhere else and verified in a more secure manner.

Error messages should be more generic. This is because in doing so, you avoid giving potential attackers useful information. An example is displaying the error message “Invalid Password.” This tells attackers that the username is correct, just not the password. Instead, messages like “Invalid Username or Password.” Would be better suited because it avoids confirming if either one of the variables is correct, which makes it more difficult for an attacker to exploit weaknesses during authentication.

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

Not initializing variables is a straightforward fix. When each of our integer variables are declared, we can simply initialize them to zero. This way we aren’t assuming that variables begin with an initial value or that they start with a null value. Strings can similarly be initialized with an empty string, which helps to prevent unexpected behavior in programs. By initializing variables, we ensure we know what their starting state is which reduces the likelihood of introducing security vulnerabilities.

Not validating user input can be fixed by implementing checks anywhere we plan to receive input from the user to check whether or not the input is valid. For instance, anywhere we receive from input and expect its value to be an integer, we can take the input and assign it to either a string or a character array that will accept a large size of characters then pass this variable to a function used to parse the input. The function will then parse through the input to determine whether or not the value entered was or was not an integer value. If the input taken in is not an integer, the parser function will return a false value. Otherwise, a true value will be returned and the value can then be assigned to the integer variable. This process can be repeated in a while loop that will continue until the user enters a valid integer. This ensures that when the user does not enter only an integer, the program will not behave unexpectedly or cause undefined behavior.

Another instance where input validation is needed is when we are taking strings in as input, we should limit the size of the input we take in, or make a maximum size we are willing to accept, especially in cases where we know about the length of a string or the format of a string that the program is expecting to receive from the user. To fix this, anytime we are receiving from input a string, we can use a do while loop to prompt the user for input and tell the loop to repeat until the input the user enters is less than or equal to the maximum size of the string we are willing to accept from input.

Validating input is one way to ensure we do not access memory locations that are out of bounds, which can cause buffer overflows. For instance, in the case of this program, we are taking in from input a value for the client number that we want to change their service number. To prevent buffer overflow that could be caused when assigning the client’s new service number, we can validate the input to ensure that the user is entering a number that represents an existing client. During the loop that is checking to determine whether they have entered a valid integer, we can also tell the loop to continue if the number entered is outside the bounds of the clientChoices array. This ensures that when the input taken is assigned to clientNumber, the memory location is valid, which in turn prevents a buffer overflow from occuring.

One method of fixing instances of passwords being hard-coded into the source code could be to hash passwords as they are created using known cryptographic hashing algorithms, and only use the hash to compare it to passwords that are given at input. The hash can be stored in an external file and later read in, and when the user enters their password, we can hash their input and check to determine whether or not the two hash values are equal.

To prevent error messages from being too specific which can aid attackers by helping them narrow down what variables are incorrect, we can ensure we display error messages that are more generic. For instance, when a password is wrong, we should not state that only the password is wrong. We should use a generic statement such as, “The username or password is invalid.” This ensures that attackers aren’t easily made aware of partially correct or incorrect credentials.

We can further expand on security by ensuring that we are encapsulating when possible. This helps ensure that information is kept hidden away from the users and important business logic can only be accessed from within its class, essentially protecting the integrity of the data. This also helps to keep code more organized by keeping it modular, helps improve the maintenance process when updates or changes need to be made in code, and improves the overall readability of the code.