CS 410 Project Two Security Report Template

Matthew Bass

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	Identified Security Vulnerability

// Login Globals
std::string username;
int answer;
 	
// Action Global
int choice;;
 	
// Choice Global
int changechoice;
int newservice;
 	
// Customer Globals
std::string name1 = "Bob Jones";
int num1 = 1;
std::string name2 = "Sarah Davis";
int num2 = 2;
std::string name3 = "Amy Friendly";
	// Security vulnerability having all these variables in globals when they can be encapsulated to the functions that use them
	// Global variable can be accessed and potentially modified from anywhere in the program.
	// Recommendation: Limit the scope of the variables

int num3 = 1;
std::string name4 = "Johnny Smith";
int num4 = 1;
std::string name5 = "Carol Spears";
int num5 = 2;	

// Customer Globals
std::string name1 = "Bob Jones";
int num1 = 1;
std::string name2 = "Sarah Davis";
int num2 = 2;
std::string name3 = "Amy Friendly";
int num3 = 1;
std::string name4 = "Johnny Smith";
int num4 = 1;
std::string name5 = "Carol Spears";
int num5 = 2;
	// Customer Globals
	// Storing customer data in plain text could be a security risk if the data is sensitive.
	// Recommendation: Use a struct or class to group related data together, and limit the scope of the customer data.
	// If the data is sensitive, ensure it is properly protected.

std::cin >> choice;	// Security vulnerability: The choice variable is not validated before use.
	// this can lead to a stack overflow or other security vulnerabilities such as a buffer overflow or format string vulnerability.
	// Recommendation: Validate the choice variable before use.
	// one way to do this is to make sure the user enters an integer, and that the integer is within the expected range.
	// example code:
	// std::string input;
	// std::cin >> input;
	// int choice = std::stoi(input);
	// if(choice < 1 || choice > 3){
	// std::cout << "Invalid choice. Please try again.\n";
	// continue;
	// }

std::cin >> username;	// No input validation. If the user enters a string longer than 255 characters, the program will crash. // can cause a stack overflow or other security vulnerabilities such as a buffer overflow or format string vulnerability.

	// Also the username variable is a global variable, which can be accessed and potentially modified from anywhere in the program.
	// Also maybe hash the username before storing it in the database, to prevent the username from being exposed if the database is compromised.
	// Recommendation: Limit the scope of the username variable, and ensure it is properly sanitized before use.
	// Recommendation: Add input validation to ensure the user enters a valid username.
	// example code:
	// std::string input;
	// std::cin >> input;
	// std::string username = hash(input);
	// if(username.length() > 255){
	// std::cout << "Invalid username. Please try again.\n";
	// return;
	// }

std::cin >> user_passwd;	// No input validation. If the user enters a string longer than 255 characters, the program will crash.
	// can cause a stack overflow or other security vulnerabilities such as a buffer overflow or format string vulnerability.
	// Also the user_passwd variable is a global variable, which can be accessed and potentially modified from anywhere in the program.
	// Also maybe hash the password before storing it in the database, to prevent the password from being exposed if the database is compromised.
	// Recommendation: Limit the scope of the user_passwd variable, and ensure it is properly sanitized before use.
	// Recommendation: Add input validation to ensure the user enters a valid password.
	// example code:
	// std::string input;
	// std::cin >> input;
	// std::string user_passwd = hash(input);
	// if(user_passwd.length() > 255){
	// std::cout << "Invalid password. Please try again.\n";

	// return; // }

No code but username needs to be checked and validated	// Security vulnerability: username is not validated before use.
	// There is no point in having a username if it is not validated before use.
	// Recommendation: Validate the username before use.
	// example code:
	// std::string stored_hash = "admin";
	// std::string input_hash = hash(username);
	// if(!input_hash.compare(stored_hash)){
	// return 1;
	// }
	// Usernames should also not be hardcoded and in plain text.

if(!user_passwd.compare("123")){
return 1;
}
return 2;
	// Password is hardcoded and in plain text, which is a security risk.
	// Recommendation: Store hashed passwords and compare the hash of the input password against the stored hash.
	// This will prevent the password from being exposed if the database is compromised.
	// example code:
	// std::string stored_hash = "123";
	// std::string input_hash = hash(user_passwd);
	// if(!input_hash.compare(stored_hash)){
	// return 1;
	// }
	// return 2;
	// Passwords should also not be hardcoded and in plain text.

std::cin >> changechoice;	// No input validation. If the user enters a non-integer, the program will crash.
	// can cause a stack overflow or other security vulnerabilities such as a buffer overflow or format string vulnerability.
	// Recommendation: Add input validation to ensure the user enters a valid integer.
	// example code:
	// std::string input;
	// std::cin >> input;
	// int changechoice = std::stoi(input);

	// if(changechoice < 1 || changechoice > 5){
	// std::cout << "Invalid choice. Please try again.\n";
	// return;
	// }

std::cin >> newservice;	// No input validation. If the user enters a non-integer, the program will crash.
	// can cause a stack overflow or other security vulnerabilities such as a buffer overflow or format string vulnerability.
	// Recommendation: Add input validation to ensure the user enters a valid integer.
	// example code:
	// std::string input;
	// std::cin >> input;
	// int newservice = std::stoi(input);
	// if(newservice < 1 || newservice > 2){
	// std::cout << "Invalid choice. Please try again.\n";
	// return;
	// }

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:
	// No validation of the changechoice or newservice variables. If the user enters an invalid value,
	// the program behavior may be unpredictable.
	// Recommendation: Add validation to ensure the changechoice and newservice variables are within expected ranges.
	// This should have been taken care of in the input validation step above.

break; }	

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;
	// Security vulnerability: The customer data is hardcoded and in plain text, which is a security risk.
	// Recommendation: Store customer data in a database, and retrieve it from the database when needed.
	// Example code:
	// void get_customer_data(int customer_id){
	// // connect to database
	// // query database for customer data
	// // return customer data
	// }

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

[In a paragraph or two for each security vulnerability, explain in detail how and why these are security vulnerabilities.]

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

[In a paragraph or two for each recommendation, describe how you would fix these vulnerabilities.]

Vulnerabilities 1 and 2: Global Vars and User data in plain text.

Global Variables for Login Information:
The use of global variables for login information, such as username and answer, poses a significant security risk. Global variables can be accessed and modified from any part of the program, making it easier for malicious actors to manipulate sensitive login data. To address this, it is recommended to limit the scope of these variables. Additionally, proper sanitization of the username before use is crucial to prevent injection attacks.

Global Variable for User Choice:

The global variable choice for user selection suffers from the same issue. Limiting its scope to the relevant part of the program is essential to prevent unintended modifications that could lead to unauthorized actions.

Global Variables for Service and Action:
Similar to the user choice, the global variables changechoice and newservice should have restricted scopes to ensure that only the intended sections of the program can modify them. This prevents potential misuse and unauthorized changes.

Storage of Customer Data in Plain Text:
Storing customer data, such as names and numbers, in plain text can expose sensitive information. To enhance security, it is recommended to use a struct or class to encapsulate customer data. This not only organizes the information logically but also allows for better control over the scope of the data. If the data is sensitive, additional measures like encryption or hashing should be employed to protect it from unauthorized access.

In summary, limiting the scope of global variables and organizing sensitive data using appropriate data structures are crucial steps in securing the C++ code. Additionally, implementing proper sanitization and protection measures for sensitive information further strengthens the overall security of the program. Mitigations for all of these can be seen in the table above where they are identified.

Vulnerability 3: Not validating user input for choice

The vulnerability in the given code arises from the lack of validation for user input assigned to the choice variable. This poses a significant security risk as it opens the door to various exploits, including stack overflow, buffer overflow, and format string vulnerabilities. Without proper validation, an attacker could input malicious data, leading to unexpected behavior or even compromising the program's execution. For instance, if the program assumes a valid integer input but receives something else, it may result in unintended consequences, such as memory corruption or manipulation.

Potential Consequences of the Vulnerability:

Stack Overflow: Unvalidated input could cause a stack overflow if the input data is larger than expected, potentially leading to a crash or execution of arbitrary code.
Buffer Overflow: If the input exceeds the allocated buffer size, it may overwrite adjacent memory, leading to unpredictable behavior and potential exploitation.
Format String Vulnerability: If the input contains format specifiers, it might lead to format string vulnerabilities, allowing attackers to read or write to arbitrary memory locations.

Recommendation: Validating User Input for the Choice Variable

To mitigate this vulnerability, it is crucial to implement input validation for the choice variable. One effective approach is to ensure that the user enters a valid integer within the expected range. The provided example code utilizes std::cin to read user input as a string and then attempts to convert it to an integer using std::stoi. It further checks if the obtained integer falls within the expected range. If the input is invalid, the program provides an error message, prompting the user to input a correct choice. Example of code to do this can be seen in the table above.

Vulnerability 4: Not validating user input for user_name

The absence of input validation for the username variable poses a risk of a buffer overflow if the user enters a string longer than 255 characters. This could lead to a stack overflow or other security vulnerabilities, including buffer overflow or format string vulnerabilities.

Moreover, the use of a global variable for the username introduces another vulnerability. Global variables can be accessed and modified from any part of the program, increasing the risk of unintended modifications or unauthorized access to sensitive user information.

Potential Consequences of the Vulnerabilities:

Buffer Overflow: If the user enters a string longer than 255 characters without proper validation, it can overflow the buffer allocated for the username variable, potentially leading to unpredictable behavior or crashes.
Global Variable Access: The global scope of the username variable makes it susceptible to unauthorized modifications, posing a risk to the confidentiality and integrity of user data.
Username Exposure: Storing the username without hashing in the database exposes it to potential compromise. Hashing the username before storage enhances security by protecting user identities even if the database is breached.
Recommendations: Input Validation and Scope Limitation for Username

To address these vulnerabilities, it is recommended to implement input validation for the username variable, ensuring that it is of a valid length. Additionally, limiting the scope of the username variable by avoiding global declarations and ensuring proper sanitation before use are crucial steps to enhance security. Code to fix this can be seen in the table above.

Vulnerability 5: Not validating user input for user_passwd

The code exposes the program to potential security vulnerabilities due to the absence of input validation for the user_passwd variable. If the user enters a string longer than 255 characters, it could result in a buffer overflow, stack overflow, or other security issues, such as format string vulnerabilities. Furthermore, the use of a global variable for the user_passwd introduces the risk of unauthorized access and modifications, compromising the confidentiality and integrity of user passwords.

Potential Consequences of the Vulnerabilities:

Buffer Overflow: Unvalidated input with a length exceeding 255 characters may lead to a buffer overflow, potentially causing crashes or unintended behavior in the program.
Global Variable Access: The global scope of the user_passwd variable makes it susceptible to unauthorized modifications, jeopardizing the security of stored passwords.
Password Exposure: Storing passwords without hashing in the database exposes them to potential compromise. Hashing passwords before storage is a best practice to protect sensitive user information in the event of a database breach.
Recommendations: Input Validation and Scope Limitation for Password

To mitigate these vulnerabilities, it is crucial to implement input validation for the user_passwd variable, ensuring that it adheres to a valid length. Additionally, limiting the scope of the user_passwd variable by avoiding global declarations and ensuring proper sanitation before use are essential steps to enhance security. Code to fix it can be seen in the table above.

Vulnerability 6: Not validating username /checking it

The identified security vulnerability stems from the absence of validation for the username before its use in the program. Without proper validation, the program becomes susceptible to various security risks, and the purpose of having a username is compromised. The provided recommendation emphasizes the need to validate the username before use to enhance the overall security of the system.

Potential Consequences of the Vulnerability:

Unauthorized Access: Without username validation, there is a risk of unauthorized access if the username is not verified before critical operations.
Security Bypass: Lack of validation may lead to security bypass scenarios, allowing attackers to manipulate or impersonate user identities.
Hardcoded and Plain Text Usernames: Hardcoding usernames in plain text exposes sensitive information and makes it easier for attackers to identify potential targets.

Recommendation: Username Validation and Security Best Practices

To address this vulnerability, it is recommended to implement username validation before using it in any critical parts of the program. The example code provided demonstrates a simple approach by hashing the input username and comparing it with a stored hash. This ensures that only validated usernames are accepted, preventing unauthorized access or manipulation. Example code to fix it can be seen in the table above.

Vulnerability 7: Storing password in plain text.

The security vulnerability in the code arises from the use of a hardcoded and plaintext password, which poses a significant security risk. Hardcoding passwords in plain text makes it easier for attackers to compromise user accounts if the source code is accessed. The recommendation here is to store hashed passwords and compare the hash of the input password against the stored hash, enhancing security and preventing exposure in case of a database compromise.

Potential Consequences of the Vulnerability:

Password Exposure: Storing passwords in plain text increases the risk of exposure, especially if the source code or database is compromised.
Security Bypass: Hardcoded passwords make it easier for attackers to gain unauthorized access by exploiting known credentials.
Recommendation: Hashed Passwords and Best Practices

To address this vulnerability, it is recommended to store hashed passwords and compare the hash of the input password against the stored hash. The example code provided demonstrates this approach by hashing the input password and comparing it with a stored hash. This practice significantly improves security by preventing the exposure of actual passwords, even if the database is compromised. Code to fix it can be seen in the table above.

Vulnerability 8: Not validating user input for changechoice

The identified security vulnerability lies in the absence of input validation for the changechoice variable, potentially leading to various security risks such as a stack overflow, buffer overflow, or format string vulnerability. Without proper validation, accepting non-integer input poses a threat to the stability and security of the program. The recommended solution is to add input validation to ensure that the user enters a valid integer within the expected range.

Potential Consequences of the Vulnerability:

Unexpected Behavior: Accepting non-integer input may lead to unexpected behavior, causing the program to crash or exhibit undefined behavior.
Security Exploits: Lack of validation opens the door to potential security exploits, including stack overflow, buffer overflow, or format string vulnerabilities.
Recommendation: Input Validation for Integer Input

To address this vulnerability, it is crucial to implement input validation for the changechoice variable. The example code provided demonstrates a simple yet effective approach using std::cin and std::stoi. It reads the user input as a string, attempts to convert it to an integer, and checks whether the obtained integer falls within the valid range. If the input is invalid, the program provides an error message and prompts the user to try again, preventing potential security issues associated with non-integer input.
Code to fix it can be seen in the table above.

Vulnerability 9: Not validating user input for newservice

The identified security vulnerability lies in the absence of input validation for the newservice variable, potentially leading to various security risks such as a stack overflow, buffer overflow, or format string vulnerability. Without proper validation, accepting non-integer input poses a threat to the stability and security of the program. The recommended solution is to add input validation to ensure that the user enters a valid integer within the expected range.

Potential Consequences of the Vulnerability:

Unexpected Behavior: Accepting non-integer input may lead to unexpected behavior, causing the program to crash or exhibit undefined behavior.
Security Exploits: Lack of validation opens the door to potential security exploits, including stack overflow, buffer overflow, or format string vulnerabilities.
Recommendation: Input Validation for Integer Input

To address this vulnerability, it is crucial to implement input validation for the newservice variable. The example code provided demonstrates a simple yet effective approach using std::cin and std::stoi. It reads the user input as a string, attempts to convert it to an integer, and checks whether the obtained integer falls within the valid range. If the input is invalid, the program provides an error message and prompts the user to try again, preventing potential security issues associated with non-integer input.
Code to fix it can be seen in the table above.

Vulnerability 10: More customer data in plain text

The security vulnerability in the code arises from the customer data being hardcoded and stored in plain text, which poses a significant security risk. Hardcoding sensitive information, such as customer names and selected options, exposes the data and makes it easier for unauthorized access or manipulation. The recommended solution is to store customer data in a secure database and retrieve it when needed, ensuring better confidentiality and integrity.

Potential Consequences of the Vulnerability:

Sensitive Information Exposure: Storing customer data in plain text makes it susceptible to exposure if an unauthorized party gains access to the source code or system.
Limited Scalability: Hardcoding customer data limits the scalability and flexibility of the system, making it difficult to manage and update information dynamically.
Recommendation: Store Customer Data in a Database

To address this vulnerability, it is strongly recommended to store customer data in a secure database and retrieve it as needed. The example code provided demonstrates a simple function get_customer_data that connects to a database, queries for customer data based on the customer ID, and returns the relevant information. This approach enhances security by centralizing and securing customer data in a dedicated storage system.
Code to fix it can be seen in the table above.