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.

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

void ChangeClientChoice(std::vector<Client >& clients) {
int clientNumber;
std::cout << "Enter the number of the client that you wish to change" << std::endl;
std::cin >> clientNumber;
if (clientNumber >= 1 && clientNumber <= clients.size()) {
int newChoice;
std::cout << "Please enter the client's new service choice (1 = Brokerage, 2 = Retirement)" << std::endl;
std::cin >> newChoice;
if (newChoice == 1 || newChoice == 2) {
clients[clientNumber - 1].serviceChoice = newChoice;
}
else {
std::cout << "Invalid choice. Please try again." << std::endl;
}
}
else {
std::cout << "Invalid client number. Please try again." << std::endl;
}
}
	- There is a potential buffer overflow issue in ChangeClientChoice when reading input for clientNumber.
	- There's a potential integer overflow issue when converting clientNumber to an index for accessing the clients vector
	- There's no error handling for invalid input (e.g., non-numeric input) when reading clientNumber and newChoice

std::string username; std::cout << "Enter your username: " << std::endl; std::cin >> username; std::string password; std::cout << "Enter your password: " << std::endl; std::cin >> password;	- There's no input sanitization for username and password inputs. This leaves the program vulnerable to injection attacks. - Usernames and passwords are transmitted and stored in plain text, which is insecure. - The password input is not masked during entry

if (password != "123") {
std::cout << "Invalid Password. Exiting program." << std::endl;
return 1;
}
	- The password is hard coded as "123", which is a security risk.
	- The lack of any password policy also weakens overall security.
	- The program only relies on a simple username-password authentication mechanism, more robust authentication recommended.
	- There's no mechanism for rate limiting failed login attempts or locking out user accounts after a certain number of failures.
	- There's no delay implemented after an authentication failure.

int choice;
do {
DisplayMenu();
std::cin >> choice;
 	
switch (choice) {
case 1:
std::cout << "You chose 1" << std::endl;
DisplayClients(clients);
break;
case 2:
std::cout << "You chose 2" << std::endl;
ChangeClientChoice(clients);
break;
case 3:
std::cout << "You chose 3" << std::endl;
break;
default:
std::cout << "Invalid choice. Please try again." << std::endl;
break;
}
} while (choice != 3);
 	
return 0;
	- There's no session management mechanism in place.
	- Proper logging is not implemented.

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

Buffer Overflow:

A buffer overflow condition is present when a program attempts to place more data in a buffer than it can hold, or when a program attempts to place data in a memory area past a buffer's boundary (OWASP, 2024). There's a potential buffer overflow issue in the ChangeClientChoice function when reading input for clientNumber. If the user enters a value that is larger than the size of the 'clients' vector, it could lead to accessing out-of-bounds memory.

Integer Overflow:

An integer overflow is a type of arithmetic overflow error in which the result of an integer operation does not fit in the allocated memory space (Nidecki, 2024). There's a potential integer overflow issue when converting clientNumber to an

index for accessing the clients vector. If clientNumber is negative, it could lead to unexpected behavior or crashes.

Lack of Error Handling for Invalid Input:

Error handling is a technique used to manage and respond to unexpected or erroneous behavior during the execution of a program. In this program, there's no error handling for invalid input (e.g., non-numeric input) when reading clientNumber and newChoice. This could lead to unexpected behavior or crashes if the user enters invalid input.

Lack of Input Sanitization and Validation for User and Pass:

Input sanitization is the process of ensuring that data conforms to the requirements of the subsystem to which it is passed (Seacord, 2024). Essentially, input sanitization removes any unsafe characters from user input, while validation will ensure the data is in the expected format and type. In the program, there's no input sanitization for username and password inputs. This leaves the program vulnerable to injection attacks, where malicious users could input special characters or strings to exploit vulnerabilities in the program.

Plain Text Transmission and Storage:

Plaintext transmission and storage is the act of transmitting or storing information in its raw form. For sensitive data, this is an extremely insecure practice. In addition to transmission concerns, storing passwords in plain text within the program's memory poses a security risk. Passwords should be securely hashed before storage to mitigate the impact of data breaches.

No Masking on Pass Input:

Password masking is the act of hiding password input as a designated character (such as *) when the user is attempting to login. On this program, the password input is not masked during entry, making it susceptible to shoulder surfing attacks where an unauthorized person could see the password being typed.

Password Hard Coded as 123:

Hard coding is the practice of embedding data directly into the source code of a program or executable object (Wikimedia Foundation, 2023). The password in the program is hard coded as "123", which is a security risk. Ideally, passwords should not be hardcoded within the source code.

Lack of Secure Password Policy:

A password policy dictates the length, characters, and other attributes of the user's password choice. The lack of any password policy (such as minimum length, complexity requirements, etc.) within the program also weakens overall security and readability.

Lack of Robust Authentication:

Simple username-password authentication schemes do not provide ample security, introducing layered security like including 2FA is recommended. As it stands, the program only relies on a simple username-password authentication mechanism.

No Rate Limiting/Lockout for Failed Logins:

Rate limiting is a strategy for limiting network traffic that puts a cap on how often a user can repeat an action within a certain timeframe, while lockout will prevent a user from attempting to access their account after a set number of failed attempts. In the program, there's no mechanism for rate limiting failed login attempts or locking out user accounts after a certain number of unsuccessful login attempts. This makes the program susceptible to brute-force attacks.

Lack of Delay after Auth Failure:

Typically, after an authentication failure a delay is implemented to prevent malicious users from garnering information about the system's execution time and thereby operational logic. With no delay implemented after an authentication failure, the system is vulnerable to timing attacks.

Limited Error Handling:

Error handling is a crucial part of secure programming, implementing robust error handling is critical for mitigating SQL injection attack and facilitating smooth program function. Although input validation is performed in certain areas, comprehensive error handling across all user inputs and system interactions is necessary to prevent unexpected behavior or exploitation.

Lack of Session Management:

As programs become increasingly complex, applications are required to retain information or status about each user for the duration of multiple requests, session management provides the ability to establish variables like access rights and localization settings (OWASP, 2024). In the program, there's no session

management mechanism in place, which could lead to session hijacking or fixation attacks.

Lack of Logging:

User session information is often taken or logged to provide developers and maintainers with the proper information to diagnose and solve problems with the program. Without proper logging, it becomes difficult to trace and investigate security incidents or anomalies effectively. Implementing a logging mechanism can aid in monitoring and detecting suspicious activities.

Buffer Overflow:

In order to fix the buffer overflow risk in the program, I would implement bounds checking when accessing elements of the 'clients' vector. This would ensure that 'clientNumber' is within a valid range between 1 and clients.size(). This could be accomplished by manually implementing bounds checking, or utilizing the at() function which inherently performs bounds checking and throws out of range exceptions.

Integer Overflow:

To fix the integer overflow vulnerability, I would implement input validation to ensure that the input is non-negative before converting it to an index. 'clientNumber' can be checked before performing the index calculation to ensure there is no error. If the size of 'clients' were to increase, usage of an unsigned integer or larger data types may be required.

Lack of Error Handling for Invalid Input:

Proper Error Handling is a bit more complicated than the specific errors above, and is relevant to multiple portions of the program. In order to institute robust error handling, input validation techniques can be utilized to catch errors, and guide the user to valid input. Prompts to the users could be placed within loops to continually prompt them until valid input is given.

Lack of Input Sanitization for User and Pass:

In order to implement proper input sanitization, a regular expression can be utilized to identify and remove harmful characters before accepting and utilizing user supplied input. This should be done for any input that has passed a trust boundary and is utilized within the program. The regular expression should be based upon the password policy that is recommended to be implemented elsewhere in the document.

Plain Text Transmission and Storage:

To remedy the transmission and storage of plaintext data within the application, such as the user's password, encryption should be utilized to secure sensitive data. Data should be secured both at rest and in transmission. Techniques like AES should be utilized to encrypt sensitive data before storing it on the database or transmitting.

No Masking on Pass Input:

In order to implement password masking within the program, platform-specific libraries or input masking techniques can be employed to conceal user input as a set character, such as a asterisk (*). Password masking should always be implemented as it prevents shoulder surfing.

Password Hard Coded as 123:

Passwords should not be hardcoded within the source code, rather passwords should be securely stored and hashed in a separate configuration file or DB. In order to fix this problem for the program, a separate configuration file or database would need to be created, and an appropriate encryption and hashing algorithm(s) chosen. I would personally recommend AES and SHA.

Lack of Password Policy:

The only way to fix a lack of a password policy is to create or adopt one. Enforcing a comprehensive password policy that included requirements for length, complexity, and expiration is an important aspect to adequate secure practices. The security policy can be enforced using input validation and sanitization, and as mentioned prior, users should be supplied feedback regarding their entry.

Lack of Robust Authentication:

To introduce a more robust authentication system, tertiary defense mechanisms like MFA and session tokens can be utilized for session management and ensure authentication tokens are securely generated, transmitted, and validated. This would require large refactoring of the current program but should be implemented for the marketed increase in security.

No Rate Limiting Failed Logins:

Rate limiting and account lockout is important for preventing brute-force attacks. User login attempts can be limited to a certain number of attempts in a specific time frame, and account lockouts (indefinite or temporary) can be set to occur

after a specific number of failed attempts. This aspect of the program should be decided within the security policy document that outlines best practices to adhere to for the development of the program.

Lack of Delay after Auth Failure:

Introducing a delay time with each authentication failure can help prevent brute force attacks, additionally, randomizing this delay can help prevent timing attacks, as mentioned previously. A progressive delay system could be implemented to address both of these attack vectors adequately, where the delay increases with each subsequent failure.

Lack of Session Management:

The implementation of session management mechanisms will allow user sessions to be established and maintained securely. This can be accomplished through the use of session tokens or cookies (with secure attributes) that can identify and authenticate users across multiple requests. Additionally, session expiration and invalidation mechanisms should be implemented to protect against hijacking or fixation attacks.

Lack of Logging:

The introduction of comprehensive logging mechanisms will help developers and maintainers track user activities, system events, and security related incidents. Information such as login attempts, authentication failures, access control decisions, and critical system operations should be logged. Secure logging libraries or frameworks can be employed to ensure the integrity and confidentiality of the log data, and implementation of log analysis tools will help monitor and detect suspicious activity effectively.

References

Nidecki, T. A. (2024, January 9). *What is integer overflow*. Acunetix. https://www.acunetix.com/blog/web-security-zone/what-is-integer-overflow/

OWASP. (2024). *Buffer overflow*. Buffer Overflow | OWASP Foundation. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

OWASP. (2024). *Session management cheat sheet*. Session Management - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

Seacord, R. (2024). *Input validation and data sanitization*. Input Validation and Data Sanitization - SEI CERT Oracle Coding Standard for Java - Confluence. https://wiki.sei.cmu.edu/confluence/display/java/Input+Validation+and+Data+Sanitization

Wikimedia Foundation. (2023, November 24). *Hard coding*. Wikipedia. https://en.wikipedia.org/wiki/Hard_coding