snhu

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

Vulnerability # 1:
 	
int32_t num1 = 1;
int32_t num2 = 2;
int32_t num3 = 1;
int32_t num4 = 1;
int32_t num5 = 2;
 	
std::string name1 = "Bob Jones";
std::string name2 = "Sarah Davis";
std::string name3 = "Amy Friendly";
std::string name4 = "Johnny Smith";
std::string name5 = "Carol Spears";
	Raw personal data for clients should be encapsulated within a class for better security, / placed into a data structure such as a hash map/unordered map) or other similar data structure.
	
	Current state of variables could lead to confusion and incorrect information being used to represent a client.
	
	No way to easily add more clients without hard coding additional sets of variables.

Vulnerability # 2: std::string username; std::string password; int64_t changechoice; int64_t newservice;	Variable declarations not initialized with default values. Can cause memory issues when address is accessed but no value has been assigned yet.

Vulnerability # 3:
 	
if (password == "123") {
localAnswer = 1;
} else {
localAnswer = 2;
}
	Hard coding of authentication data / passwords should never be done. Use password hashes instead of raw passwords, hash the user input password and compare the hashes.

Block of C++ Code	Identified Security Vulnerability

Vulnerability # 4:
 	
do { // authentication loop
 	
answer = CheckUserPermissionAccess();
 	
if (answer != 1) {
std::cout << "Invalid Password. Please try again\n";
}
} while (answer != 1);
	No rate limit or attempt limit enforced for attempts at authentication, program is very vulnerable to brute forcing.

Vulnerability # 5:
 	
if (answer != 1) {
std::cout << "Invalid Password. Please try again\n";
}
	Printing detailed authentication failure message is a security vulnerability. Makes brute forcing easier and can open you to side-channel attack. Use a generalized authentication/login attempt failure message.

Vulnerability # 6:
 	
do { // authentication loop
 	
answer = CheckUserPermissionAccess();
 	
if (answer != 1) {
std::cout << "Invalid Password. Please try again\n";
}
 	
} while (answer != 1);
	Authentication process is a public function accessed in the main function instead of an authentication class that encapsulates methods.

Vulnerability # 7:
if (password == "123") {
localAnswer = 1;
} else {
localAnswer = 2;
}
	Generic string comparisons create insecurity to timing attacks. Using values to compare directly in if conditional not best practice.

Block of C++ Code	Identified Security Vulnerability

Vulnerability # 8:
 	
if (password == "123") {
localAnswer = 1;
} else {
localAnswer = 2;
}
	Using “magic numbers” for program control instead of const values (read only), calculated at compile time. Values that don’t need to change should not be able to change.

Vulnerability # 9: std::cout << "Enter your username: \n"; std::cin >> username; std::cout << "Enter your password: \n"; std::cin >> password;	No input validation here, no sanitization of inputs.

Vulnerability # 10:
 	
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;
}
	Not following DRY / best practices. No need to create a new output line for each client, clients should be stored in a data structure, and the data structure iterated through to output client list

Block of C++ Code	Identified Security Vulnerability

Vulnerability # 11:
switch (changechoice) {
case 1:
// set client 1's service to equal newservice
num1 = newservice;
break;
case 2:
// set client 2's service to equal newservice
num2 = newservice;
break;
case 3:
// set client 3's service to equal newservice
num3 = newservice;
break;
case 4:
// set client 4's service to equal newservice
num4 = newservice;
break;
case 5:
	Again, it is bad coding practice to have this much duplicate code without an express need.

Vulnerability # 12: std::cin >> newservice; std::cin >> changechoice; std::cin >> username; std::cin >> password;	Use exception handling for reading inputs / for handling operation failures. Program currently will not handle invalid types etc in a graceful manner.
Vulnerability # 13: std::cout << "Enter the number of the client that you wish to change" std::cin >> changechoice; std::cout << "Please enter the client's new service choice (1 = Brokerage, 2 = Retirement)" std::cin >> newservice;	No input validation here, no sanitization of inputs, no exception handling.

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

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

I. Vulnerability # 1:

- Sensitive data, such as raw client information should always be properly stored. Proper storage includes using an appropriate data structure, as well as encapsulating the data / data structure within a class when using an object oriented language such as C++. In the initial application code, client names, id numbers, and service selections are all given variables, but none of these variables are directly tied to each other in any way other than a number reference within the variable name. These should be paired up with matching values assigned to a class object instance, assigned a private status, and given mutator / accessor methods to modify or retrieve the values for a specific instance. This ensures that there is no mess, or any confusion due to having a long list of assorted variables in the general program structure. Each client will have their own instance, and the data of those clients will only be accessible by ways that are defined in the class setup. This also provides a modular way to add additional clients in the future without the need to hard code a new full set of matching general variables. An interface can be used to access the addClient() method, and passed the relevant Client data.

II. Vulnerability # 2:

- There are a number of variables that are not initialized with a value in their declarations. They can be found within the main method, within the CheckUserPermissionAcces function, and the ChangeCustomerChoice function. By declaring a variable but not initializing it, you may open yourself and your application up to undefined behaviors. Attempting to access a variable that hasn't been initialized can cause a host of problems such as memory related access violations, exceptions, or application clashes. The application may end up exposing sensitive data by accessing incorrect memory, and can leave your application compromised. Always give declared variables a default / initialization value to avoid these problems.

III. Vulnerability # 3:

- The application directly stores the password "123" in plain text within the code for the program. This is a severe violation of secure, best coding practices. One of the easiest ways you can compromise your application is by hard-coding security or other authentication related data. Never use hard coded passwords or authentication credentials, as doing so provides individuals familiar in reverse engineering direct access to those values. It is very easy to access hard-coded strings or other values, so

they should never be written into the code. Passwords should always be stored as hash values, ideally as salted hash values. They should be stored on an encrypted drive, or in a similar secure manner whenever possible. To authenticate a user password, you should hash the input password and compare the hash to the stored hash.

- In this application we will implement an openssl hash function using SHA256, and this function will be used to validate entered passwords by comparing the input password to the stored hash for the original password of the application "123". For the purpose of this application, we will store the valid password hash within the constructor of the singleton authentication class.

IV. Vulnerability # 4:

- The application does not implement any form of rate limiting for authentication attempts, or any form of attempt limits. Rate limits limit the speed at which a user can attempt to connect to the system, and attempt limits will limit the quantity of authentication attempts that a user can make. Rate limiting is important, especially in the context of networking infrastructure, as too many attempts to connect in too short of a time frame can effectively lock up a system. Methods such as denial of service, or distributed denial of service attacks can cause damage to a system, and can cause undefined behavior to occur. This is especially true in systems that are not set up to lock out connection attempts that have passed a threshold, or with a method to "fail with grace" when large volumes of traffic / attempts to connect are made.
- Not limiting the connection attempts also leaves your system open to brute forcing, as there is nothing preventing a malicious user from making an unlimited amount of connection attempts.
- For this system specifically, it would be helpful to implement a limit to the quantity of valid connection attempts that can be made to the system. This would be straight forward to implement using an authentication class object that handles providing connections to users. A straightforward implementation of this would be to create an authentication class that handles authentication attempts with a private member variable. It will increment the attemptsMade variable whenever the user provides a valid authentication request but fails to connect due to invalid password etc. For the purpose of this application it will be a singleton method. But for a networked version of the application, instances and connection attempts should be assigned by user account (per username/password).

V. Vulnerability # 5:

- Printing detailed authentication failure message is a security vulnerability. Use a generalized authentication/login attempt failure message.
- The system initially provides the user with feedback in the form of an "Invalid password" message when the user enters an incorrect password. Giving feedback for

authentication in this manner makes brute forcing easier and can open your system up to side-channel attacks. Side channel attacks can occur when a system inadvertently leaks information about the way that it operates, and that data can be used to glean enough information to gain unauthorized access to the system. We will use a generic failed authentication attempt message when a user does not make a successful login attempt but provides valid inputs.

VI. Vulnerability # 6:

- I mention the implementation of an authentication class in vulnerability #4, and the corrections regarding #6 will also be built into the implementation of the authentication class.
- Currently, the authentication method is very simple, and is performed below the main method within the program file. This is not best secure coding practices, as the variables are open and accessible, and the method itself is public. To correct this implementation, we will be implementing a public authenticate method within the authentication class. This authenticate method will be what takes the input password, calls the private hashing function to create a hash of the input password, and then calls the private constant time comparison function with the stored hash value and the input password hash value to determine if the passwords match. This will encapsulate all of the inner functions and variables of the authentication process within private member methods and variables, and will provide a modular interface for the authentication process to be called from.

VII. Vulnerability # 7:

- Generic string comparisons create insecurity to timing attacks. Using values to compare directly in simple comparisons such as with comparing strings does not follow best, secure coding standards.
- In vulnerability #5, I reference the fact that providing detailed error messages regarding authentication states can open your system up to side-channel attacks. One of the most common types of side-channel attacks is known as a timing attack. Different operations within a system can take varying time frames to perform, and analyzing the time that the system takes to perform it's operations can allow attackers to gather valuable data about what operations are being performed. This greatly depends on the implementation that a programmer uses. If for instance you are directly comparing a stored string value against an input string value, and are doing so in a way that returns when an incorrect character is reached, the time of those returns will vary. There are a number of ways this can be performed but at least with a simple timing attack, analyzing this side-channel of information can allow an attacker to deduce the correct password with enough connection attempts.
- To resolve this issue, we will implement a comparison method for the password hashes that utilizes constant time to complete the comparison. The comparison

ensures that we take the same amount of time every time a comparison attempt is made. In this instance, I will iterate over the length of characters in the stored hash, and compare each to the corresponding character at the same position in the user input password hash. I perform an XOR (exclusive or) operation on the characters at each position and assign that value using an OR assignment operator to an answer variable. The answer will be initially set to the value 0, so that if any of the characters are different within the hash, we will end up with a non-zero answer value. We can then declare that the answer is zero, and return the boolean of that statement. If the answer is not zero, the values of the hashes were not the same, and the passwords did not match. Because we iterate over the stored hash, the amount of time the operations take remains constant.

VIII. Vulnerability # 8:

- Using "magic numbers" for program control instead of const values (read only), calculated at compile time. Values that don't need to change should not be able to change. Using magic numbers for program flow is not in line with best coding practices.
- Magic numbers are unique values within the application that do not have an explained meaning associated with them in the same way that a named constant value would. An example of such would be integers used in conditional branching that is used to control the program flow. Using magic numbers disrupt the readability of the code, and create complications when it comes to maintainability. They can be inadvertently changed down the road, and in doing so implement undesired behavior or vulnerabilities. It is best to always use named constant values when implementing values to control the flow of the application. All instances of magic numbers in the program will be transitioned to named constants, and new control variables will follow this same guideline.

IX. Vulnerability # 9:

- No input validation here, no sanitization of inputs
- Not implementing a system of robust input validation system is one of the greatest sources of vulnerabilities for a system. It is extremely important to make sure that a malicious actor cannot use the input of your program to gain access to sensitive data. Without proper sanitation of inputs as well, you leave the system open to attacks that can directly inject of code into your system by passing it as an input to the program. A common example of this would be passing SQL commands/arguments into the system to control the flow of information within the database that the system relies on. Another could be using the input system to pass information related to your methods of authentication. An example could be passing an input that overflows a given input value and causes a portion of the input to be treated as code instructions. This could lead to a scenario when someone is able to completely bypass the normal

authentication process. Applications must always account for all possible input conditions, and limit the system inputs to only allow those which would maintain desired functionality. This application will use a string validation function to validate string inputs, limiting only basic alphanumeric values to be entered. An integer validation function will also be implemented to ensure that only valid integers can be given to the system, and both functions will limit the the number of characters allowed for a single input.

X. Vulnerability # 10:

- Not following DRY / best practices. No need to create a new output line for each client, clients should be stored in a data structure, and the data structure iterated through to output client list.
- Numerous duplications of the same statement for variables is not best coding practice. We should always consolidate duplications such as these into a object instances when necessary, and store the instances within an appropriate data structure. Having a large amount of named variables performing the same operations creates a code-base that is burdensome to navigate. These could very easily be contained within a single data structure and into class / object instances. We will implement a Client class to manage all of our individual clients. This will encapsulate their private personal data. We will then use a ClientList singleton class to manage all operations related to the client class. This instance will contain a static vector with all Clients added to it on their creation. To create a new Client object instance, call the addClient method of the ClientList class. Passing the addClient method a client name and service choice will automatically generate the next available clientId number, assign it to the client, and then add the new Client object to the vector of Clients. These classes will provide a modular interface to access the data and operations of the program through, and will do so in a way that can later allow the addition/integration of a graphical user interface.

XI. Vulnerability # 11:

- Bad coding practice to have this much duplicate code. This is an extension of the previous code problem, #10. This is another area where code is being duplicated without the need to. This could very easily be contained within a class object, and a single accessor function that iterates over a list of the objects. Best coding practices would not be to continue adding cases to a switch statement indefinitely. The application should be designed in a way that supports maintainability and stability. The values should be defined once, within the class object, and then if a matching object is found in the list of objects, the value can be modified. This way we are not rewriting the same code multiple times throughout the application, and it is easy to find and

maintain into the future.

XII. Vulnerability # 12:

- It is not best, secure coding practices to allow inputs or operations within your program to cause an unexpected exception, or to result in the program failure / a crash. We should use exception blocks to account for all states that might allow the program to fail. If an operation fails and there is not exception handling in place it could cause undefined behavior and potentially expose vulnerable areas of the application. Branching should handle all expected edge-cases, and exception handling along with a robust logging system should catch the remaining errors.
- Exception handling will be used for reading input, and for attempting operations within the application that could fail. The current program will not handle invalid type in a graceful manner, and there is no handling or checks for the process of assigning a new service value to a client.

Sources:

Timing Attacks:

- https://www.chosenplaintext.ca/articles/beginners-guide-constant-time-cryptography.html
- https://timing.attacks.cr.yp.to/index.html
- https://stackoverflow.com/questions/8768469/guessing-a-string-with-time-comparison-is-it-possible