# 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** |
| --- | --- |
| string name1 = "Bob Jones", name2 = "Sarah Davis", name3 = "Amy Friendly", name4 = "Johnny Smith", name5 = "Carol Spears";  string num1 = "1", num2 = "2", num3 = "1", num4 = "1", num5 = "2"; | Client list is hard coded. This can/should be abstracted, and client elections should not be obtainable through reverse engineering |
| cin >> choice;  cin >> username;  cin >> password;  cin >> changechoice;  cin >> newservice; | No input validation means vars can be any combination of characters and numbers |
| if ((username == "UUU") && (password == "123")) return true; | Hardcoded U and PW that can be (and were) discovered through software reverse engineering |
| while (!answer) {  answer = CheckUserPermission();  } | no log in limit means you can just run the program a billion times to find the password and username |
|  |  |
|  |  |
|  |  |
|  |  |

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

**VULNERABILITIES**

Hardcoded client list and options – This is a security vulnerability for a few reasons, but the main reason is that the client’s names and financial elections can be easily reverse engineered just by running the program and checking the memory locations. This means any malicious actor with any access to the program and memory can find the client list and elections.

Input validation – This is probably one of the easiest ways for unauthorized users to gain access to privileged information, or even to worm their way into parts of the system the application wasn’t meant to touch. Having no input validation can also allow for remote code execution, which could have an untold number of unintended consequences for the program.

Hardcoded username and password – Similar to the client list, having the username and password hard coded into the program is a huge problem because it means anyone with access to the log in screen can pretty easily access the username and password vars through storage and gain privileged access to the system. If the primary goal of security it to prevent unauthorized access, this seems like a big first oversight.

No Log in limit – This is pretty self-explanatory, but you have to have a log in limit that disallows the user from trying to log in a billion times. If there is no limit and no check on the number of log in attempts, brute force becomes a viable solution to breaking the system.

**SOLUTIONS**

BRIEF:

Input validation will be handled by simply dropping additional input in the console buffer before it reaches the program, a log in limit will be implemented, and asymmetrical encryption will be implemented for the client lists and username and password.

Input Validation – Input validation for this kind of program is fairly simple, since we can control how many characters we want in each field. The choice, changechoice and newservice variables all need to be only 1 character long, so we can just take the first character from getChar() which will take one character then flush the rest of the console buffer. For the username and password, we’ll just get the first 20 characters from cin (which has built in buffer overflow protection), and then just drop the rest of the input out of memory so there’s no risk of remote code execution or buffer overflow.

No log in limit – A very simple solution is to add an integer variable that is incremented when a log in attempt fails, and exits the program when a limit is reached. In wider implementation for this program, some kind of log will need to be updated on the server side, but for our discrete program we will just exit for now.

Hardcoded username and password – This solution is a bit more complicated than the last, but to start with, we’ll want to encrypt the password, using asymmetric encryption. Using asymmetric encryption means that if the password is recovered by reverse engineering, you only have the encrypted version, and the key built into the program cannot un-encrypt the password or username. Essentially, this solution means the program doesn’t actually know what the username or password is, but still has a way of checking whether the information provided to it is correct. This makes reverse engineering not much better than just brute force log in attempts. Proper implementation would require that un/pw validation and storage done off the local machine, which would make recovering the encrypted values and encryption key much more difficult. Therefor, this solution will not be implemented now, and those connections would need to be made later.

Hardcoded client list and options – The solution to this problem is very similar to the problem of having hardcoded usernames and passwords, except we’re going to need to recover the original values. This means we’ll either need to use symmetrical encryption, or use a key that isn’t stored in the program itself for de-encryption. We won’t want to use symmetrical encryption, because there’s a greater risk that the key could be recovered from the program and used to gain privileged information. A potential solution could be to use the user’s encrypted password as a key for the de-encryption of the client list. This solution would be under the assumption that the master client list will only be accessed by the administrator of the system, and separate encrypted client lists will be assigned to each user of the system. This solution will NOT be implemented right now, because encrypted information should also be stored off system in a secure server location to further prevent access via reverse engineering of the executables.