Aaron J Walls

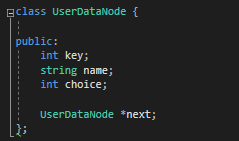
Professor Brooke Goggin

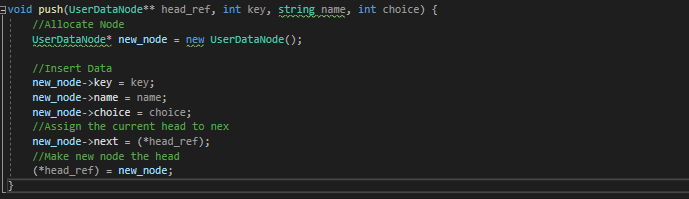
CS 499

November 21, 2021

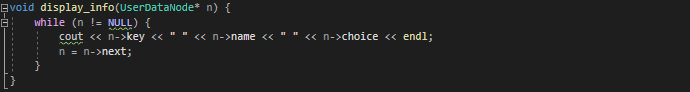
MS3: Data Structures and Algorithms

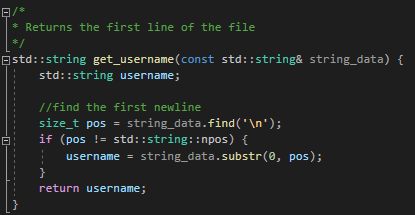
The software being modified today was simple accounting software that was reverse-engineered from a binary. The goal of the assignment was to recreate the software and identify the security vulnerabilities in the application. The application used hardcoded values to store the password and account information. The application also did not validate user input causing the application to crash.

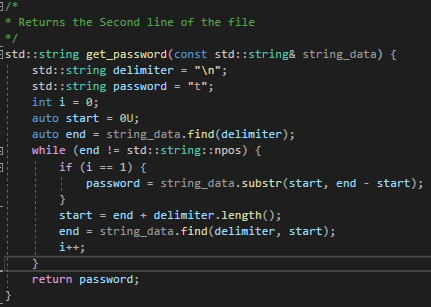
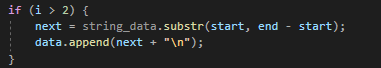
The enhancements made to the application replaced the hardcoded values with information pulled from an encrypted file. It also stores the data in a link-list data structure, rather than in global variables. The UserDataNode class stores the key, name, and choice along with a reference to the next node. The key is the number associated with the account entry, the name is the name of the account owner, and the choice represents Brokerage (1) or Retirement (2). Using the linked list allows the data to be stored, sorted, and pushed easily. However, given the complexity of the program, the only features needed are a push function, and a display function for the nodes. 



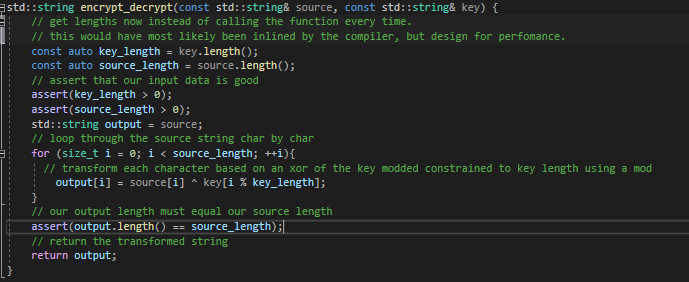
The Display function is simply looping through the linked list until the referenced Node is null.

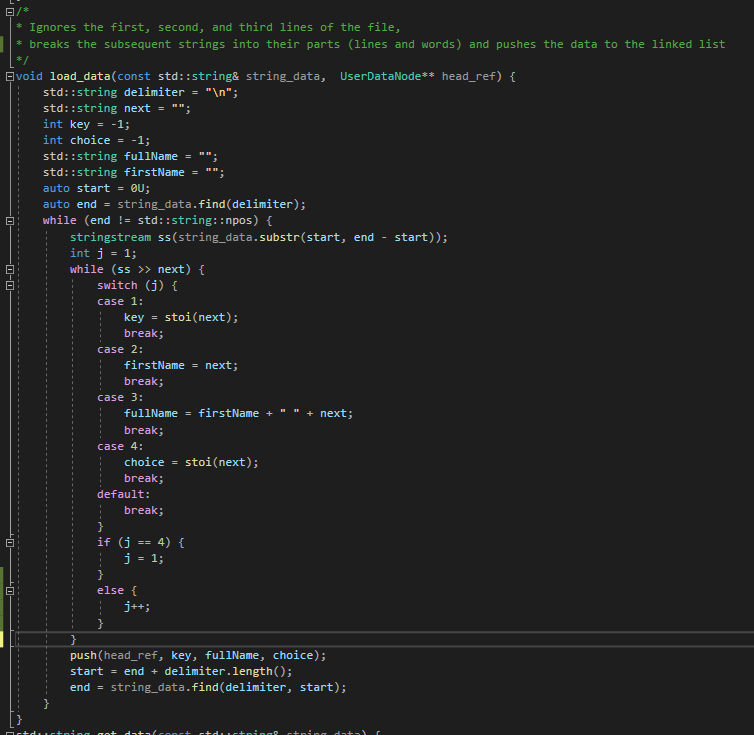


Creating the linked list was the easy part. Retrieving the data from a file required a little thought about file formats and string logic. The format of the file is; the username on line one, password and salt hash on line two, salt on line three, and user information on line five and below. The user information is written the same order each time: key, first name, last name, and choice. To read the file, I have separate functions to retrieve the username, password, salt, and data. 

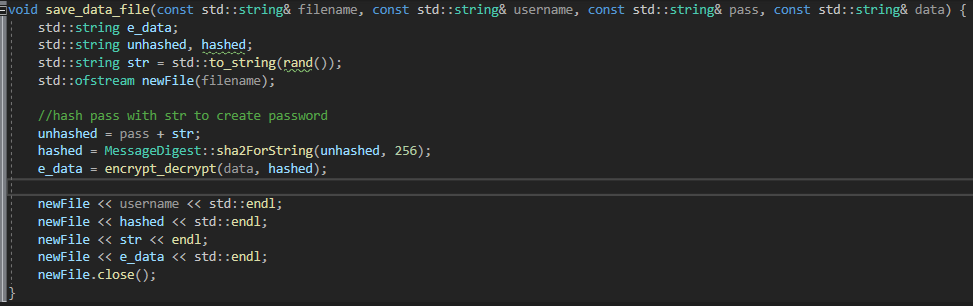
Get username just needs the first line. Using the newline as a delimiter, the last character of the line can be found. Since the username is on the first line, the beginning of the username is at zero. From this, we can get the substring between 0 and newline. Getting the subsequent lines uses similar logic, except the start is where the end was, and the length of the substring is the difference between the new end and start. The if statement ensures that the password gets exactly the line I am looking to get. In this case, it’s the second line. The same strategy can be used to grab any individual line. To grab multiple lines, as with the case for reading data, the if statement can be changed. 

It is possible to make one function to grab the username, password, salt, and data, but I have the split them into their own functions. The only string that needs to be loaded into the linked list is the data. Since the data string is encrypted, data needs to be sent to the encrypt\_decrypt function first.

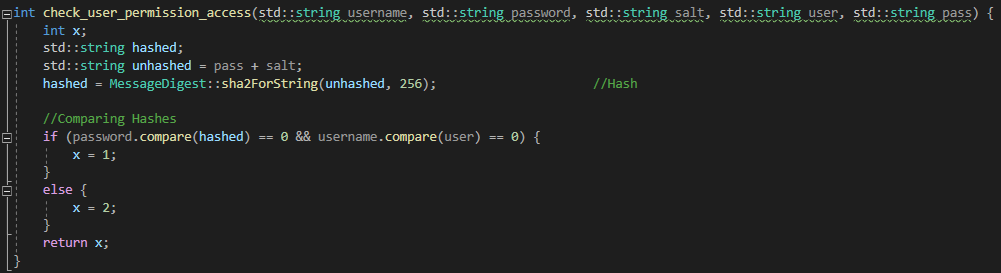


This function uses xor encryption to change plain text to encrypted text, or encrypted text to plain text. It encrypts based on the key. The key for this application is a salted password hashed with sha256. Given that the key is strong, the encryption is fairly robust. That being said encryption should be handled by cryptographically secure libraries. This algorithm is for demonstration purposes. Once decrypted it is loaded into the linked list by more string manipulation techniques. 

This strategy loops through the lines using delimiters but then breaks down those lines to the words using stringstream. Each word is saved to a variable based on their order since the order of the data is always the same. The user’s key is first, followed by the first name, last name, and choice. The first name is concatenated with the next to create the full name. All these variables are then pushed to the linked list, allowing the data to be used in the program.

When it is time to write to the file, a new salt is created, and the password is salted then hashed. This means every hash used to log in is unique to that instance. 

This function generates a random number and adds it to the end of the unhashed password as a string. Then, the unhashed password is hashed using sha256 by using the OpenSSL libraries. Once the new hash is created, the data is encrypted and saved to “e\_data”. Then the data is recorded to the file. Logging in is as simple as fetching the hash, the salt, the user input, adding the user input to the salt, and recalculating the hash based on the user input. If the two hashes match, then the user has authenticated with the application.



In this example, the password is the hashed value fetched from the file, and "pass” is the user input.

This application demonstrates the extent of care that has to be applied to securing an application. Applying secure coding concepts to a simple console application can cause the code to balloon and become a large project, especially when trying to add security after the fact. This artifact demonstrates some security concepts involving logging in, storing passwords, and dealing with encrypted data. It has added insight into how complicated ‘logging in’ can be. This workflow works, but imagine if I needed to account for multiple users, concurrent data edits, and 2FA practices? It would be impractical for me to write my own libraries for that. Given how important logging in the workflow is to security, it is not surprising that many companies employ or use third-party applications to manage authentication. That being said, anyone who is involved with application security should be familiar with how authentication works.