Assignment # 1
(CS-2001 Data Structures – Fall-2023)

Due Date and Time: Thursday, 21st September, 2023 (11:59 pm)
Total Marks: 280 (including bonus marks)

Instructions:

- *Late assignment will not be accepted*
- *Upload all your files (cpp files, header files and any data files) as a **single zip**, with the naming convention (Assignment1_Section_rollNumber1_rollNumber2.zip)*
- *Your solution will be evaluated in comparison with the best solution*
- *This assignment can be done in **pairs.***

Hash-Chain Logging for Access Violation in a Management System

It is well known that simple management systems such as those you have implemented in Object Oriented Programming, especially those containing sensitive data, can be misused or illegally accessed. A typical management system, implemented in C++, will contain a number of classes representing various real-world entities, e.g. a library management system will contain classes like Library, Book, Author, Member etc. Users interact with the management system through a graphical or command line interface to view, add, edit or delete data and perform related functionality such as fine calculation for late book return. Data associated with the system e.g. book lists or member lists is usually stored in binary files. Anyone who gains access to this system, even if unauthorised (e.g. someone who has a stolen password of an administrator) can perform operations on sensitive data.

In this assignment you will implement an attack detection system for any management system of your choice by maintaining and monitoring a *hash-chain*. A hash chain is not a real data structure but a dummy one that we will create for this assignment – it is loosely based on the concept of blockchains and is essentially a linked list.

Part 1: Implement Management System with Access Control:
Start by implementing a simple management system of your choice. It should have at least 20 different functionalities (such as the fine calculation functionality referred to earlier). It should also have a number of user roles. Every user logs in and the system identifies their role – e.g. in a library management system, the users can be administrator, librarian, library member, premium member, author, auditor etc. You have to present a menu-driven interface to the user in which they first log in (check validity of credentials against a password file you should have saved to disk), and then perform various actions that are allowed for their role. For example, library members are allowed to request books available in other branches of the library to be shipped to their local branch, they can get books issued, return books, pay fine etc. However they are not allowed to change the librarian's duty roster or charge fines to other members. You should maintain a list of the functionalities each kind of user is allowed

to perform (maintain a text file for this) and match a user's request against the requested action. If a library member tries to call the change_duty_roster function, you should not allow it, but this is to be decided at run time based on the file. These rules can change on file at run time and your system should check the latest rules every time. There should be a superuser that is allowed to do anything on the system, i.e. call all possible functions and change any data. Now you have a basic management system that implements access control functionality.

Part 2: Implement Action Logging

In such a system, there are two main security risks. One is that the superuser can be malicious and abuse their privilege to carry out data stealing, destructive or fraudulent activities such as charging too high a fine, deleting important data, or accessing and selling users' confidential details such as phone number or address. This kind of attack is called insider attack. The other risk is that a non-authorised user (e.g. someone who is not a library member) can hack into the system using, for example, a stolen password of an existing librarian's account, and carry out the activities allowed for that user (the librarian in this example). To prevent against such attacks, usually systems have a logfile that records every action performed by every user on the system along with its time of occurrence (current system time). When a user adds data for example, or calls the fine calculation function, or requests to view a user's home address, each action is logged – e.g. the log file may have an entry as follows:

```
Time: 15:25:03 10/10/2023, User: Ahmad Tauseef, Requested action: Edit
home address, Result: Home address edit successful.
```

If the request is denied, you may see "action denied" in the Result column.

Your job is to create a logging system that logs every action that any user performs, starting from login, through the menu driven interface. The logs should be saved to a file. This way illegal accesses can be detected by someone who goes through the logs.

Part 3: Log Security: Storing Logs in a Hash-Chain

An administrator or other attacker who has access to your file storage can carry out malicious actions and then modify the logs so that they remain undetected. For example, they may edit a user's home address to cause inconvenience in book shipping, but then change the log file entry by putting another user's name in place of their own. Insider attackers can usually do this easily. The logs need to be in something other than a simple text file to detect such manipulation.

The solution for this is to maintain logs in a structure that we will call a hash-chain. A hash-chain is essentially a chain of connected blocks (just like a linked list is a chain of connected nodes) that contain some data. In this assignment we assume that when a new action occurs, it is stored on the hashchain as a new block. A hashchain should be tamper-proof, i.e. no one can change or delete any data that has become part of the hashchain without being detected. This is accomplished through *hashing*. Each block contains *hashes* of all previous blocks. A hash is a one-way mathematical function that is applied to some input data to get a fixed-size, usually small output. A block contains the hash of the previous blocks so that if any block is removed or changed, the hash value of all subsequent blocks will not be correct, and it can be detected that something was changed. (More details of the hashing function are found in the end of this section).

You should implement a linked list of strings to represent the hash chain. Whenever you log an action in the logfile, it should be added in the text file as described earlier, and at the same time, you should add it to the hashchain (i.e. the linked list) as a new node to be inserted in the linked list – the entire line of text will be the data of the node. As the linked list is a hashchain, each node should contain, in addition to the data and the next pointer, a hash value the previous blocks. So every time you insert a new block, calculate the hash of the previous block and add it to the new block.

After every addition to the hashchain, write the entire hashchain to a file and name the file "HASHCHAIN_<Timestamp>" where the timestamp is the current system time. You can delete previous versions of the hashchain to save space.

Now you have implemented a secure logging functionality that will ensure that if someone changes the txt log file, it will not match with the hashchain file and hence the attack can be detected.

To demonstrate that this functionality is correct, write a function for traversing the hashchain and printing it to the screen (this can be called after performing a few actions so we can check the validity) and add it to the main menu.

Hashing Function:

The hashing is done as follows. First convert each node of the linked list to a string by concatenating the string data member of the class (containing the line of the logfile) and the hash value stored in the block. Use any library of your choice to calculate the md5 OR sha-1 hash value (md5 or sha-1 function usually takes a string as input). The output of the function will be your hash value for that block. Note that the hash value is a data member of the block, but it stores the hash value of the *previous* block, not of the current block itself. Hence, the hash value (of the previous block) is treated as a string in the current block and should be included in the construction of the concatenated string itself that you will use to calculate the next block's hash value. Whenever you insert a block in the blockchain, you must populate the hash field with the hash value of the previous block. This way, the hashes are chained together securely and if someone deletes or alters a node, the hash value stored in the next block is no longer valid and the change can be detected.

Part 4: Simulating an Attack, Detection and Recovery:
The main strategy for attack detection is that the hashes become incorrect if anyone tries to tamper with the hashchain. Suppose someone carries out an in-memory attack (using some other code) to tamper the hashchain when a new action is recorded – e.g. an admin who illegally deletes important data, later realises that the hashchain captured his actions, and so he deletes the node of the hashchain and the line of the txt log file that records the action, or edits them to pin the blame on someone else. Now the hashes of the next nodes in the chain become incorrect as one node is randomly deleted from the middle.

You should simulate such an attack in your code. Carry out a malicious action, then some normal actions, then delete the node that recorded the malicious action or edit its value (the idea of a hashchain is that values cannot even be edited later) to simulate the attack. For detection, you should write a hashchain verification function that traverses the hashchain from start to end and confirms that all hashes are in order. If any hash is out of place, it generates a BREACH message and prints the incorrect node, as well as 2 nodes before and 2 nodes after the incorrect hash (for context). This indicates that there was a breach. Your system should now auto-recover from the breach by reading the correct value of the node/hashchain (before it was tampered) from a backup copy (the HASHCHAIN_<timestamp> files you have saved) and inserting the correct node back in its place. Then you should run the hash verification again to confirm that now the hashchain is restored to a valid state.

Part 5: Reporting
Also write at least five different reporting functions to explore the hashchain – e.g. list all activities of a particular user, list all activities from a particular time window (e.g. between 5 and 10 p.m. on a given date). Include these in a separate menu. For example, you can divide your main menu into "Security" or "Regular" – the regular portion can have the normal menu, that will have all data manipulation and display functionality, and the security portion can have reporting based on the hashchain logs. Use the hashchain, not the text log file, to implement the search functionality for all five of your reporting functions.

Marking Rubric:

This assignment will be marked as follows:

Item	Marks
Basic management system with menu driven interface	**Total: 30** 10: Barely working 20: Basic functionality smooth with some missing pieces 30: Perfect
Access control in management system	**Total: 30** 10: Barely there 20: Mostly there with some missing pieces (e.g. without file handling) 30: Perfectly done
Action logging:	**Total: 30** 10 marks for coverage (all actions are logged) 10 marks for descriptive message (all required details are logged for the action) 10 marks for correct file storage.
Hashchain logging	**Total: 50**

	30 marks for coverage (all actions inserted in hashchain correctly) 10 marks for correct linked list implementation (shown by traversal and printing correctly) 10 marks for correct hashing
Attack simulation, detection and recovery	**Total: 60** Attack simulated sensibly: 10 marks Attack detected correctly with correct approach: 20 marks Attack recovery correctly done with file handling: 30 marks
Reporting	**Total: 50** 10 marks for each function
Bonus	30 marks for exceptional work, with good display, clear demonstration of all functionality, and perfect working.
Total (includes bonus)	280 Marks