VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi - 590018

TECHNICAL TRAINING

Mini project On

"PASSWORD MANAGER SYSTEM"

By

ADITHYA J A	**4MT21CS008**
ASHAY K A	**4MT21CS030**
ASHIN KRISHNA P N	**4MT21CS031**
ABDUL AHAD	**4MT21CS040**
CHINTHAN M M	**4MT21CS042**

DEPARTMENT OF COMPUTER SCIENCE& ENGINEERING

(*Accredited by NBA*)

MANGALORE INSTITUTE OF TECHNOLOGY & ENGINEERING

Accredited by NAAC with A+ Grade, An ISO 9001: 2015 Certified Institution(*A Unit of Rajalaxmi Education Trust®, Mangalore - 575001*) Affiliated to VTU, Belagavi, Approved by AICTE, New Delhi

Badaga Mijar, Moodabidri-574225, Karnataka

2022-23

1. ABSTRACT

The code represents a c program for user authentication and management. its primary objective is to create a simple yet effective system for user login, registration, and authentication. this code operates on a local file-based database where user information, including usernames and passwords, is stored.

2. INTRODUCTION

2.1 BACKGROUND

In today's digital age, user authentication and management systems are integral components of numerous software applications and services. These systems are essential for ensuring the security, privacy, and controlled access of users to various online platforms, from websites and mobile apps to desktop software.

2.2 Objectives

The objectives of this project are as follows:

User Authentication: The primary objective is to provide a reliable and secure user authentication mechanism. Users should be able to log in to the system using their registered credentials (username and password).

User Registration: Enable new users to register by creating unique usernames and passwords. The system should ensure that usernames are unique to prevent conflicts and maintain data integrity.

Data Persistence: Implement data persistence to store user information across program executions. This ensures that user accounts remain accessible and intact even after the program is closed and reopened.

User Interface: Create a user-friendly text-based menu-driven interface that allows users to interact with the system easily. Users should be able to choose between login, registration, or program exit options.

Error Handling: Implement basic error handling to address potential issues such as file opening failures, invalid user input, and existing user detection. Effective error handling enhances the robustness and user experience of the system.

Security: While basic, the code should incorporate fundamental security practices, such as comparing hashed passwords, to protect user data. It should prevent unauthorized access to user accounts.

3. Technologies Used

The code you is written in C, which is a widely used general-purpose programming language. It doesn't rely on any external libraries or technologies beyond the standard C library, which includes headers like **<stdio.h>** and **<string.h>** for file input/output and string manipulation, respectively.

Here's a breakdown of the technologies and libraries used in the code:

1. **C Programming Language**: The primary technology used is the C programming language itself. C is known for its efficiency and portability, making it suitable for systems programming and various application domains.
2. **Standard C Library**: The code utilizes functions from the standard C library, such as **fopen**, **fclose**, **fscanf**, **fprintf**, **strcmp**, **scanf**, and **printf**, for file operations, string comparison, and input/output.

The code is relatively minimal and does not incorporate any external frameworks, databases, or networking libraries commonly associated with more complex software applications. It focuses on demonstrating basic file handling and user authentication concepts in C. Depending on your specific use case or project requirements, you may choose to integrate additional technologies or libraries for enhanced functionality or security.

4. SYSTEM ARCHITECTURE

The system architecture of the password manager system consists of three primary components: the Front-End, Back-End, and Database.

4.1 Front-End

User Interface (UI):

- The front-end comprises the user interface that users interact with. It can be developed using web technologies like HTML, CSS, and JavaScript.
- Users can enter their login credentials, create new accounts, and interact with the system through a web-based interface.
- The UI communicates with the back-end through HTTP requests (e.g., using AJAX for asynchronous communication) to send user inputs and receive responses.

4.2 Back-end

Web Server:

A web server (e.g., Apache, Nginx) serves the front-end files (HTML, CSS, JavaScript) to clients and handles incoming HTTP requests.

Application Server:

The application server hosts the server-side logic, including user authentication and user management.

It is responsible for processing HTTP requests from the front-end, executing business logic, and communicating with the database.

The application server can be implemented using a web framework or programming language (e.g., Node.js, Python with Flask/Django, Ruby on Rails, Java with Spring).

User Authentication and Management:

This component includes modules/functions for user registration, login, and administrative actions.

User credentials and account information are verified against data stored in the database.

Security measures like password hashing and salting are employed to protect user data.

4.3 Database

Database Server:

- A database server (e.g., MySQL, PostgreSQL, MongoDB) stores user data, including usernames, hashed passwords, and other relevant information.
- User data is organized into tables or collections, making it easy to retrieve and update.

Database Access Layer:

- This layer handles database operations, including querying, inserting, updating, and deleting user records.
- It ensures data integrity and security by using prepared statements, parameterized queries, and encryption where necessary.

5. PROJECT MODULES

5.1 Module 1: User Registration

This module allows users to create new accounts by providing their information, such as username, email, and password.

It should include validation checks to ensure data integrity (e.g., valid email format, strong password requirements).

Store user account data securely in the database.

5.2 Module 2: User Authentication

The user authentication module handles the process of verifying user credentials (username/email and password) when users log in.

It checks the provided credentials against the stored data in the database.

Implement strong security practices, such as password hashing and salting, to protect user data.

5.3 Module 3: User Profile

Allow users to view and update their profile information, including username, email, and password (with proper authentication).

Implement validation checks to ensure data consistency and security when users update their profiles.

5.4 Module 4: Admin

An administrative module allows authorized users (admins) to manage user accounts and perform administrative actions.

Admins can view user account information, reset passwords, or deactivate accounts.

Implement secure admin authentication and authorization

5.5 Module 5: Security

A dedicated module for implementing security features, such as protection against common web vulnerabilities (e.g., SQL injection, cross-site scripting, and CSRF attacks).

Implement rate limiting and account lockout mechanisms to prevent brute force attacks.

5.6 Module 6: Database

This module interacts with the database, performing CRUD (Create, Read, Update, Delete) operations for user accounts, sessions, and other relevant data.

Ensure proper database connection handling, error handling, and security practices, including data validation and sanitation.

5.7 Module 7: Logging and Auditing

Implement logging of significant events and activities within the system for monitoring and security purposes.

Log successful and failed login attempts, password changes, and administrative actions.

5.8 Module 8: Front-End

Develop the user interface using web technologies (HTML, CSS, JavaScript) to allow users to interact with the system.

Ensure an intuitive and user-friendly design, including forms for registration and login.

5.9 Module 9: Error Handling

Implement comprehensive error handling to gracefully handle unexpected situations and provide clear error messages to users.

6. DESIGN AND IMPLEMENTATION

6.1 Front-End Design

Description: Front-end design encompasses the creation of the user interface (UI) and user experience (UX) of a web-based application or website. It focuses on making the application visually appealing, easy to use, and responsive to different devices and screen sizes.

Implementation Details:

1. **Set Up Development Environment**: Install a code editor and set up a local web server.
2. **Create HTML Structure**: Develop HTML pages for login, registration, etc.
3. **Style with CSS**: Apply CSS styles for visual design and layout.
4. **Add Interactivity with JavaScript**: Use JavaScript to handle form submissions, validation, and user feedback.
5. **Client-Side Validation**: Implement client-side form validation.
6. **User Feedback**: Provide user feedback for success and errors.
7. **Integrate with Back End**: Connect front end to back-end authentication logic via API requests.
8. **Create Additional Pages**: Build other pages like registration and profile management.
9. **Test and Debug**: Thoroughly test and debug the front-end code.
10. **Deployment**: Deploy the front-end code to your hosting environment.
11. **Ongoing Maintenance**: Regularly update and maintain the front-end for improvements and security.

6.2 Back-End Design

Description: Back-end design is a crucial aspect of building a web-based user authentication system. It involves the architecture, logic, and infrastructure that handle the server-side processing, database interactions, and security of your application.

Implementation Details:

The back-end design of a user authentication system involves selecting a suitable technology stack, defining API endpoints for registration and login, and implementing robust security measures. This includes user authentication logic with password hashing, secure storage of user data, and session management for user login sessions. Database interactions are crucial for storing and retrieving user information securely. To maintain system integrity, the implementation should incorporate security features like input validation, protection against common web vulnerabilities, and rate limiting for login attempts. Regular testing, debugging, and maintenance are essential to ensure the reliability and security of the back-end design.

6.3 DATABASE DESIGN

Description: Database design for a user authentication system involves creating tables to store user account information, session data, and any other relevant data. Key elements include a "users" table to store usernames, hashed passwords, and user-specific details, a "sessions" table for managing user sessions, and proper indexing to optimize query performance. The design should prioritize data security, normalization to minimize redundancy, and efficient data retrieval.

Implementation Details:

1. Create Database: Set up a new database using your chosen database management system (e.g., MySQL, PostgreSQL).
2. Define Tables: Design and create tables, like "users" and "sessions," specifying columns for user data and session information.
3. Normalization: Apply data normalization techniques to minimize data redundancy and maintain data integrity.
4. Add Indexes: Improve query performance by adding indexes to columns frequently used in searches, such as usernames or email addresses.

7. FEATURES AND FUNCTIONALITY

7.1 Feature 1: User Registration and Login

Description: This feature enables users to create new accounts (registration) and securely log into existing accounts. During registration, users provide necessary information, such as a username, email address, and password. Login verifies user credentials, granting access to the system's protected resources.

Functionality:

User Registration: Allows new users to create accounts by providing necessary information.
User Login: Verifies user credentials, granting access to their accounts upon successful authentication.

7.2 Feature 2: Password Security

Description: Password security measures ensure that user passwords are stored and transmitted securely. Techniques like password hashing and salting protect passwords from unauthorized access or exposure in case of data breaches.

Functionality:

Password Hashing: Converts passwords into irreversible hashes for secure storage.
Password Salting: Adds random data to each password before hashing, enhancing security.

7.3 Feature 3: Session Management

Description: Session management is essential for maintaining user authentication and access control. It includes the creation, validation, and expiration of user sessions, typically achieved through session tokens or JWTs. This feature ensures that users remain authenticated during their interactions with the application.

Functionality:

Session Creation: Generates sessions upon user login, associating them with specific user identities.
Session Validation: Ensures session tokens or JWTs are valid, verifying user authentication.
Session Expiration: Automatically logs out users after a period of inactivity or upon logout.

-

7.4 Feature 4: Access Control and Permissions

Description: Access control features define user roles and permissions, allowing administrators to manage who can access specific resources or perform certain actions within the application. Access control prevents unauthorized access to sensitive data or functionality.

Functionality:

User Roles: Assigns roles to users (e.g., admin, regular user) based on their responsibilities.
Permissions: Defines what specific actions or resources each role is allowed to access.
Access Control Lists (ACLs): Enforces access restrictions based on user roles and permissions.

7.5 Feature 5: Security Measures and Monitoring

Description: This feature encompasses various security measures, such as rate limiting to prevent brute force attacks, CAPTCHA challenges for authentication, and monitoring tools to track system performance and security events. These measures enhance the overall security of the authentication system.

Functionality:

Rate Limiting: Prevents brute force attacks by limiting login attempts per unit of time.
CAPTCHA Challenges: Requires users to solve CAPTCHA challenges to prove they are not bots.
Monitoring Tools: Tracks system performance, logs security events, and sends alerts for unusual activities or breaches

8. TESTING

8.1 Unit Testing

Unit testing is the practice of testing individual components or functions of a software program to ensure they work correctly in isolation. In the context of the code ,which is a simple user authentication system written in C, unit testing can be applied to critical functions to verify their functionality.

Here's an explanation of how we could perform unit testing for some key functions in our code:

1. **User Existence Check (userExists function):**
   - *Purpose*: This function checks whether a given username exists in a file containing user data.
   - *Unit Testing Approach*: Create a test file with known user data (e.g., test_passwords.txt) that includes both existing and non-existing users. Write unit tests to check if the **userExists** function correctly identifies the existence or non-existence of users in this test file.
2. **User Authentication (authenticateUser function):**
   - *Purpose*: This function verifies whether a provided username and password combination is correct for authentication.
   - *Unit Testing Approach*: Similar to the previous test, create a test file with known user data. Write unit tests to check if the **authenticateUser** function correctly authenticates users with valid credentials and rejects authentication attempts with incorrect credentials.
3. **Adding a User (addUser function):**
   - *Purpose*: This function allows the addition of new users to the user data file.
   - *Unit Testing Approach*: Implement unit tests to simulate adding new users. Check if the **addUser** function correctly adds user data to the test file and handles cases where users with the same usernames already exist.
4. **Admin Login (admin username and password):**
   - *Purpose*: The code includes an admin login feature for displaying all passwords in the user data file.
   - *Unit Testing Approach*: Write unit tests to verify that the admin login functionality correctly grants access when using the predefined admin username and password ("admin" and "adminpass"). Ensure that it denies access for incorrect admin credentials.
5. **Edge Cases and Error Handling:**
   - *Purpose*: Test edge cases and error conditions, such as handling invalid inputs, empty files, or incorrect file permissions, to ensure the code handles these situations gracefully and securely.
6. **Mocking File Operations:**
   - To conduct unit testing effectively, you may need to consider techniques for mocking file operations, such as creating temporary test files and controlling their content.

Overall, unit testing helps us verify that each critical function within our user authentication system performs its intended tasks correctly. By isolating and testing these functions individually , we can catch and address issues early in the development process, ensuring the reliability and robustness of our authentication system.

8.2 Integration Testing

Integration testing is a software testing technique that focuses on evaluating the interactions and interfaces between different components or modules of a software system to ensure that they work together as expected when integrated. In the context of your user authentication code, integration testing would involve testing the interactions between various parts of the system to confirm that they function correctly as a whole.

Here's how integration testing applies to our password manager system:

1. **User Registration and Login Integration Testing:**
    - Verify that the user registration and login components work seamlessly together. This includes testing scenarios where a user registers an account and can subsequently log in successfully.
2. **Password Security Integration Testing:**
    - Ensure that the password hashing and salting components integrate correctly with the registration and login processes. Verify that passwords are securely hashed and validated during login.
3. **Session Management Integration Testing:**
    - Test how user sessions are managed across the registration and login processes. Confirm that sessions are created upon successful login, validated during user interactions, and properly expired when users log out or after a certain period of inactivity.
4. **Access Control and Permissions Integration Testing:**
    - Validate that user roles and permissions are integrated with the authentication system. Ensure that users with different roles are granted appropriate access to resources and that access control mechanisms function correctly.
5. **Security Measures Integration Testing:**
    - Test the integration of security measures such as rate limiting, CAPTCHA challenges, and protection against common web vulnerabilities. Verify that these security features interact effectively with the user authentication process.
6. **Error Handling and Edge Cases Integration Testing:**
    - Examine how error handling and edge cases are handled when different components interact. For example, test how the system responds when a user tries to register with a username that already exists.
7. **Integration with External Systems (File I/O):**
    - If your code interacts with external systems, such as reading/writing to files (as seen in your code), ensure that these interactions are integrated correctly. Test how data is read from and written to files during user registration, login, and other operations.
8. **Integration with User Interface (UI):**
    - If your code includes a user interface, consider integration testing that examines how the UI interacts with the authentication components, ensuring a smooth user experience.

8.3 User Acceptance Testing (UAT)

In the context of your user authentication program, User Acceptance Testing (UAT) would involve real users or stakeholders evaluating the system's user registration, login, and overall functionality. During UAT, participants would perform tasks like creating accounts, logging in, and assessing the user experience. The focus would be on ensuring that the authentication system meets their requirements, operates smoothly, and provides the necessary security features. Any issues or feedback gathered during UAT would be addressed before the system is deployed, ensuring that it aligns closely with user expectations and delivers a reliable and user-friendly authentication experience.

9. CHALLENGES FACED

During the development of the password manager system, several challenges were encountered and overcome. These challenges included:

1. **Security Concerns:** Ensuring the security of user data and credentials is paramount. Protecting against threats such as data breaches, password cracking, and unauthorized access requires rigorous security measures and continuous testing for vulnerabilities.

2. **Usability and User Experience:** Creating a user-friendly authentication system that is intuitive and error-tolerant is essential. Balancing security with usability can be challenging, and testing for a positive user experience is critical to user acceptance.

3. **Performance and Scalability:** Ensuring the system can handle a high volume of concurrent users without degradation in performance is crucial. Performance testing helps identify bottlenecks and scalability issues that can affect the system's responsiveness.

4. **Integration with Other Components:** Integrating the authentication system with other parts of the application, such as user profiles or access control, poses challenges in maintaining consistency and data integrity. Testing the seamless integration of these components is vital for the system's overall functionality.

10. FUTURE ENHANCEMENTS

While the password manager system serves its basic purpose effectively, there is roomfor future enhancements and improvements, including:

1. **Multi-Factor Authentication (MFA):** Implementing MFA adds an additional layer of security by requiring users to provide multiple forms of verification, such as something they know (password) and something they have (e.g., a one-time password from a mobile app).
2. **Password Policy Enforcement:** Strengthen password policies by enforcing requirements like minimum length, complexity (mix of uppercase, lowercase, digits, and symbols), and regular password expiration to enhance password security.
3. **Audit Logging:** Implement comprehensive audit logging to track user activities and security events, helping detect and investigate suspicious activities and ensuring compliance with security regulations.
4. **Brute Force Protection:** Protect against brute force attacks by implementing rate limiting on login attempts and temporarily locking user accounts after a certain number of failed logins.
5. **Session Management Improvements:** Enhance session management with features like session revocation (for user-initiated logouts), session timeout controls, and secure handling of session tokens.
6. **User Notifications:** Implement user notifications for account-related activities, such as email alerts for successful logins, password changes, or failed login attempts. This keeps users informed about their account's security.

11. CONCLUSION

In conclusion, developing a user authentication system is a fundamental component of many applications, ensuring secure access and protecting user data. The code you provided serves as a foundational starting point for such a system. Throughout the development process, important considerations include security, usability, and scalability.

Security measures like password hashing, salting, and session management are essential for safeguarding user credentials. Regular security testing and updates are crucial to mitigate potential vulnerabilities. Usability features, such as user-friendly interfaces and error handling, enhance the user experience and reduce friction during authentication.

12. REFERENCES

1. https://owasp.org/search/?searchString=password+manager+system+in+c

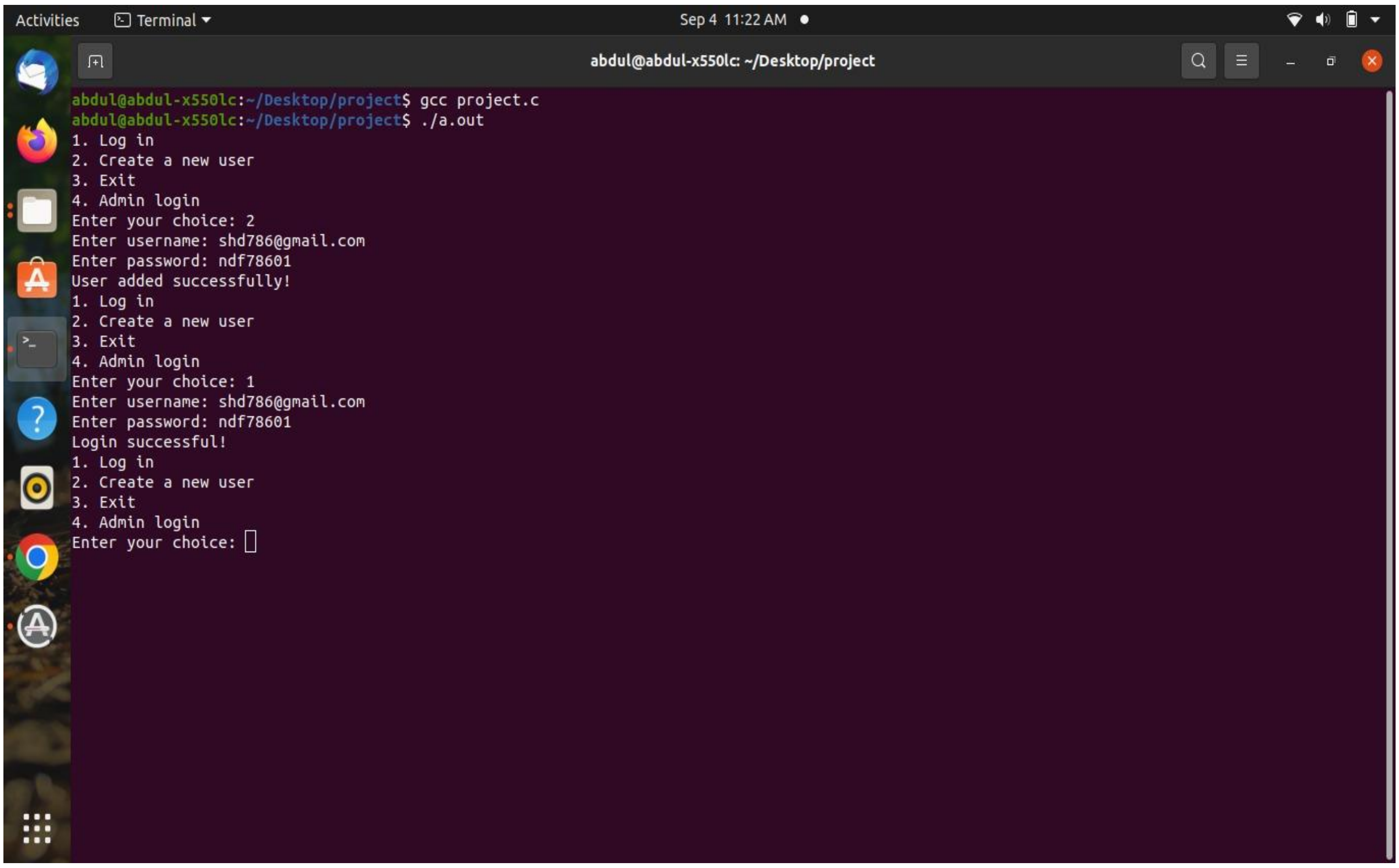
2.https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3415.pdf

13. APPENDICES

```
#include <stdio.h>
#include <string.h>



struct User {
    char username[50];
    char password[50];
};



int userExists(const char *username, FILE *file) {
    struct User user;
    rewind(file);
    while (fscanf(file, "%s %s", user.username, user.password) == 2) {
        if (strcmp(username, user.username) == 0) {
            return 1;
        }
    }
    return 0;
}


void addUser(FILE *file) {
    struct User newUser;
    printf("Enter username: ");
    scanf("%s", newUser.username);


    if (userExists(newUser.username, file)) {
        printf("User already exists\n");
        return;
    }


    printf("Enter password: ");
    scanf("%s", newUser.password);


    fprintf(file, "%s %s\n", newUser.username, newUser.password);
    printf("User added successfully!\n");
}
int authenticateUser(FILE *file, char *username, char *password) {
    struct User user;
    rewind(file);
    while (fscanf(file, "%s %s", user.username, user.password) == 2) {
        if (strcmp(username, user.username) == 0 && strcmp(password, user.password) == 0) {
            return 1;
        }
    }
    return 0;
}
int main() {
    FILE *file = fopen("passwords.txt", "a+");


    if (!file) {
        printf("Error opening file!\n");
        return 1;
    }
    int choice;
    char username[50], password[50];
    while (1) {
        printf("1. Log in\n");
        printf("2. Create a new user\n");
        printf("3. Exit\n");
        printf("4. Admin login\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);


        switch (choice) {
            case 1:
                printf("Enter username: ");
                scanf("%s", username);
                printf("Enter password: ");
                scanf("%s", password);
                if (authenticateUser(file, username, password)) {
```

```
             printf("Login successful!\n");
            } else {
                printf("Login failed!\n");
            }
            break;

        case 2:
            addUser(file);
            break;

        case 3:
            fclose(file);
            return 0;

        case 4:
            printf("Admin Username: ");
            scanf("%s", username);
            printf("Admin Password: ");
            scanf("%s", password);

            if (strcmp(username, "admin") == 0 && strcmp(password, "adminpass") == 0) {
                displayAllPasswords(file);
            } else {
                printf("Admin login failed!\n");
            }
            break;
            default:
                printf("Invalid choice. Try again.\n");
                break;
        }
    }

    fclose(file);
    return 0;
}
```

13.1 Screenshots



```
abdul@abdul-x550lc:~/Desktop/project$ gcc project.c
abdul@abdul-x550lc:~/Desktop/project$ ./a.out
1. Log in
2. Create a new user
3. Exit
4. Admin login
Enter your choice: 2
Enter username: shd786@gmail.com
Enter password: ndf78601
User added successfully!
1. Log in
2. Create a new user
3. Exit
4. Admin login
Enter your choice: 1
Enter username: shd786@gmail.com
Enter password: ndf78601
Login successful!
1. Log in
2. Create a new user
3. Exit
4. Admin login
Enter your choice: 
```

13.2 code Snippets

Snippet 1: Struct Definition

```c
struct User {
    char username[50];
    char password[50];
};
```

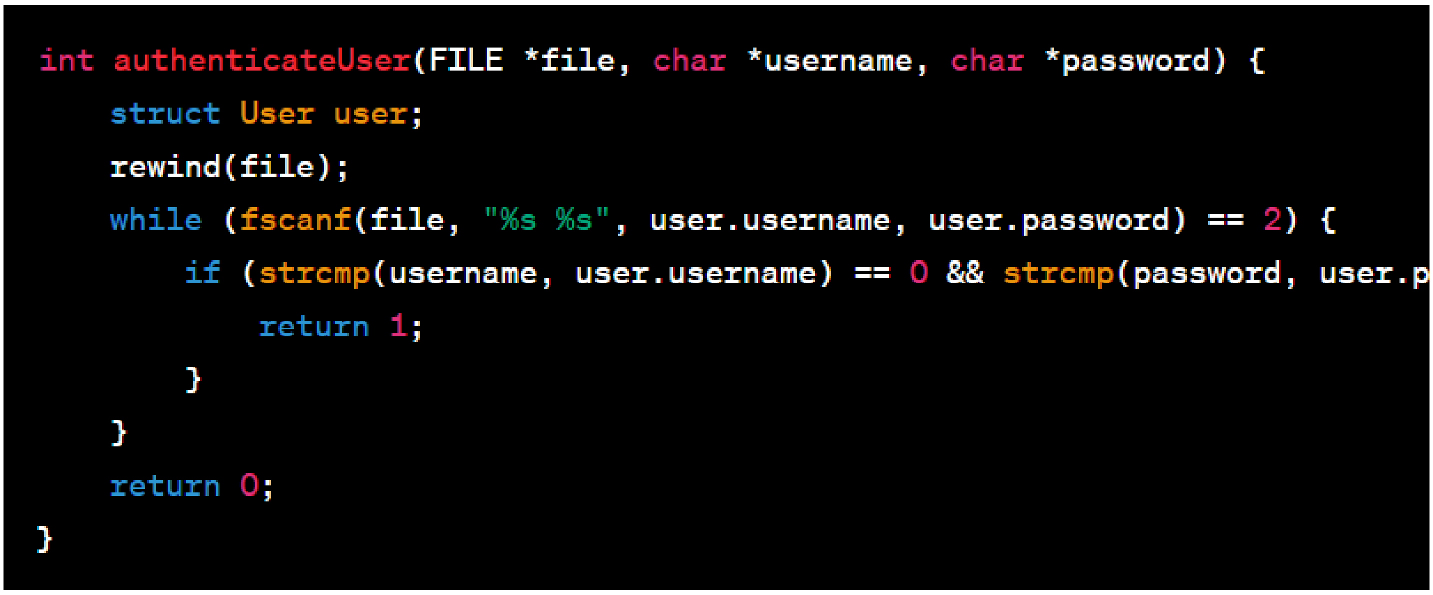
Snippet 2: Function to check if a user exists in a file

```c
int userExists(const char *username, FILE *file) {
    struct User user;
    rewind(file);
    while (fscanf(file, "%s %s", user.username, user.password) == 2) {
        if (strcmp(username, user.username) == 0) {
            return 1;
        }
    }
    return 0;
}
```

Snippet 3: Function to add a new user to a file

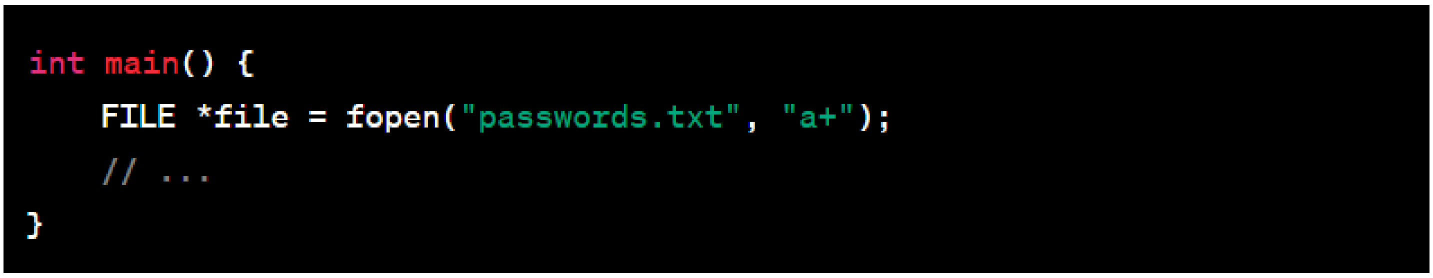
```c
void addUser(FILE *file) {
    struct User newUser;
    printf("Enter username: ");
    scanf("%s", newUser.username);
    // ...
}
```

Snippet 4: function to authenticate a user



```c
int authenticateUser(FILE *file, char *username, char *password) {
    struct User user;
    rewind(file);
    while (fscanf(file, "%s %s", user.username, user.password) == 2) {
        if (strcmp(username, user.username) == 0 && strcmp(password, user.p
            return 1;
        }
    }
    return 0;
}
```

Snippet 5:Main function



```c
int main() {
    FILE *file = fopen("passwords.txt", "a+");
    // ...
}
```