**13. Develop a system to enforce strong password policies, including minimum length, complexity requirements, and regular password changes, to prevent unauthorized access to network resources.**

**Steps to Implement a Password Policy Enforcement System:**

1. **Define Password Rules**:
   * Minimum password length (e.g., 8 characters).
   * Minimum number of character types (uppercase, lowercase, digits, special characters).
   * Password expiration policy (e.g., every 30 days).
   * Password history to prevent reuse.
2. **User Input**: Prompt the user to create or update their password based on these rules.
3. **Password Validation**: Check the password against the defined rules before allowing the user to change it or authenticate.
4. **Password Expiry Check**: Track when the password was last changed and enforce a regular change.
5. **Password Storage**: Store passwords securely, typically using a hashing algorithm (e.g., SHA-256 or bcrypt) so that plaintext passwords are not stored.

Below is a simple implementation of such a system in **C**. This implementation will cover the following features:

* **Password Length**: Ensures the password meets the minimum length.
* **Password Complexity**: Ensures the password contains at least one uppercase letter, one lowercase letter, one digit, and one special character.
* **Password Expiry**: Simulates password expiration after a certain number of days.
* **Password History**: Ensures the user doesn't reuse recent passwords.

**C Program to Enforce Strong Password Policies**

c

Copy code

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <time.h>

#define MIN\_PASSWORD\_LENGTH 8

#define PASSWORD\_EXPIRY\_DAYS 30

#define MAX\_HISTORY 5

// Structure to store user password and history

struct User {

char current\_password[100];

char password\_history[MAX\_HISTORY][100];

time\_t last\_changed;

};

// Function to check password length

int checkPasswordLength(char \*password) {

return strlen(password) >= MIN\_PASSWORD\_LENGTH;

}

// Function to check password complexity (contains uppercase, lowercase, digit, special char)

int checkPasswordComplexity(char \*password) {

int has\_upper = 0, has\_lower = 0, has\_digit = 0, has\_special = 0;

for (int i = 0; password[i] != '\0'; i++) {

if (isupper(password[i])) has\_upper = 1;

if (islower(password[i])) has\_lower = 1;

if (isdigit(password[i])) has\_digit = 1;

if (ispunct(password[i])) has\_special = 1;

}

return has\_upper && has\_lower && has\_digit && has\_special;

}

// Function to check if the password is in history

int checkPasswordHistory(char \*password, struct User \*user) {

for (int i = 0; i < MAX\_HISTORY; i++) {

if (strcmp(password, user->password\_history[i]) == 0) {

return 1; // Password found in history

}

}

return 0;

}

// Function to update password history

void updatePasswordHistory(struct User \*user, char \*new\_password) {

for (int i = MAX\_HISTORY - 1; i > 0; i--) {

strcpy(user->password\_history[i], user->password\_history[i - 1]);

}

strcpy(user->password\_history[0], new\_password); // Add new password to history

}

// Function to check if the password has expired

int isPasswordExpired(struct User \*user) {

time\_t now = time(NULL);

double seconds = difftime(now, user->last\_changed);

return (seconds / (60 \* 60 \* 24)) > PASSWORD\_EXPIRY\_DAYS; // Compare days

}

// Function to set a new password

int setPassword(struct User \*user, char \*new\_password) {

// Check if password length is sufficient

if (!checkPasswordLength(new\_password)) {

printf("Error: Password must be at least %d characters long.\n", MIN\_PASSWORD\_LENGTH);

return 0;

}

// Check if password meets complexity requirements

if (!checkPasswordComplexity(new\_password)) {

printf("Error: Password must contain an uppercase letter, a lowercase letter, a digit, and a special character.\n");

return 0;

}

// Check if password is in history

if (checkPasswordHistory(new\_password, user)) {

printf("Error: You cannot reuse your previous passwords.\n");

return 0;

}

// Update password history and set new password

updatePasswordHistory(user, new\_password);

strcpy(user->current\_password, new\_password);

user->last\_changed = time(NULL);

printf("Password successfully changed.\n");

return 1;

}

// Function to authenticate user

int authenticateUser(struct User \*user, char \*password) {

if (strcmp(user->current\_password, password) == 0) {

printf("Authentication successful.\n");

return 1;

} else {

printf("Authentication failed.\n");

return 0;

}

}

int main() {

struct User user = {0}; // Initialize user structure

char password[100];

char new\_password[100];

printf("Set your initial password:\n");

while (1) {

printf("Enter password: ");

scanf("%s", password);

// Try to set the password

if (setPassword(&user, password)) {

break; // Successfully set password

}

}

// Simulate checking password expiration

if (isPasswordExpired(&user)) {

printf("Your password has expired. Please change it.\n");

}

// Authentication simulation

printf("Authenticate with your password:\n");

printf("Enter password: ");

scanf("%s", password);

authenticateUser(&user, password);

// Change password after expiration (for simulation)

printf("\nChange your password:\n");

while (1) {

printf("Enter new password: ");

scanf("%s", new\_password);

// Try to set the new password

if (setPassword(&user, new\_password)) {

break; // Successfully set new password

}

}

return 0;

}

**Explanation of the Program:**

1. **Password Length Check**: Ensures the password has at least MIN\_PASSWORD\_LENGTH characters.
2. **Password Complexity Check**: Ensures the password contains at least one:
   * Uppercase letter
   * Lowercase letter
   * Digit
   * Special character (e.g., !@#$%)
3. **Password History**: Prevents users from reusing their last 5 passwords.
4. **Password Expiry**: Simulates checking if a password has expired based on the last change time (PASSWORD\_EXPIRY\_DAYS).
5. **User Authentication**: Authenticates users by comparing the input password with the stored password.
6. **Password Change**: Allows users to change their password, updating history and expiration time.

**Compilation and Execution:**

**Compilation:**

bash

Copy code

gcc -o password\_policy password\_policy.c

**Execution:**

bash

Copy code

./password\_policy

**Sample Output:**

plaintext

Copy code

Set your initial password:

Enter password: Password123!

Password successfully changed.

Authenticate with your password:

Enter password: Password123!

Authentication successful.

Change your password:

Enter new password: NewPass123!

Password successfully changed.

**Key Features:**

* **Password Policy Enforcement**: Ensures that passwords are long, complex, and unique by checking length, complexity, and history.
* **Password Expiry**: Enforces password expiration after a certain period (e.g., 30 days).
* **Password History**: Prevents users from using old passwords by maintaining a history of the last 5 passwords.

**Potential Enhancements:**

* **Secure Password Storage**: Store hashed passwords instead of plaintext (using bcrypt, PBKDF2, etc.).
* **User Interface**: Develop a more user-friendly interface (e.g., web or GUI-based).
* **Integration with Network Systems**: Integrate with authentication systems like LDAP, Kerberos, or Active Directory.