PASSWORD SECURITY TOOL

Submitted to: Dr. Dolly Das

MADE BY: KUSHAGRA RANJAN

Sap:590025702

Angad Chaudhary

Sap:590026262

Batch 45

SCHOOL OF COMPUTER SCIENCE

INDEX

1.0 ABSTRACT

Passwords remain one of the most widely used methods for securing user accounts and digital systems. However, passwords stored in plaintext or weakly encrypted formats continue to pose major risks. This project investigates password vulnerabilities by implementing a brute-force password-cracking simulation in the C programming language.
The system reads credentials from a text file, verifies user existence, and attempts to crack the corresponding password by generating and testing character combinations until a match is found. The goal is to practically demonstrate the exponential time growth associated with brute-force attacks and to highlight the importance of using hashing, salting, and strong password policies in real-world environments.

The project includes algorithm design, flowcharts, file handling techniques, recursion for combination generation, and performance evaluation. Experimental results show that even a slight increase in password length drastically increases cracking time, emphasizing the significance of strong passwords.

This project contributes to understanding password vulnerabilities and encourages the implementation of advanced authentication techniques in modern systems.

2.0 Problem Definition

With increasing digitalization, user authentication has become a critical security component. Passwords are commonly used but often mismanaged, leading to unauthorized access, data theft, and security breaches.

Issues Observed:

1. Many systems still store passwords in plaintext.
2. Users frequently choose weak passwords (e.g., "1234", "password", "admin").
3. Brute-force tools can easily exploit such weak credentials.
4. Lack of awareness about the time complexity of brute-force attacks.

Problem Statement:

To simulate a brute-force attack using C programming that demonstrates how weak passwords can be compromised. The system should:

1. Load credentials from a file.
2. Identify a specific user.
3. Attempt to brute-force the user’s password by generating every possible combination.
4. Analyse the time and attempts required.
5. Highlight the need for secure password-storage methodologies.

3.0 System Design

4.1 System Architecture Overview

The system follows a simple but structured workflow:

1. **User Input Module** – accepts username.
2. **Credential Lookup Module** – scans through creds.txt for matching username.
3. **Brute-Force Engine** – generates and tests password combinations using recursion/loops.
4. **Time Calculation Unit** – measures execution time using <time.h>.
5. **Output Module** – displays cracked password, attempts, and total time.

4.2 Brute-Force Algorithm (Detailed)

Algorithm: BRUTE_FORCE (password)

Input: Character set, maximum length, target password
Output: Time taken + found password

1. Start timer.
2. Define character set:
   {a–z, A–Z, 0–9, special symbols}
3. For length = 1 to max length:
   - Generate all combinations of current length.
   - For each string:
     - Compare with target using strcmp().
     - If match:
       - Stop timer.
       - Return result.
4. End.

4.0 IMPLEMENTATION DETAILS

5.1 Programming Language G Tools

1. **Language:** C
2. **Compiler:** GCC
3. **Editor:** VS Code
4. **Dependencies:**
   - #include <stdio.h>
   - #include <stdlib.h>
   - #include <string.h>
   - #include <time.h>

5.2 Modules

Module 1: File Handling (creds.txt)

5. Format
6. admin pass123
7. test qwerty

Module 2: Password Lookup

Uses fscanf() to read username-password pairs.

Module 3: Brute-Force Generator

Supports:

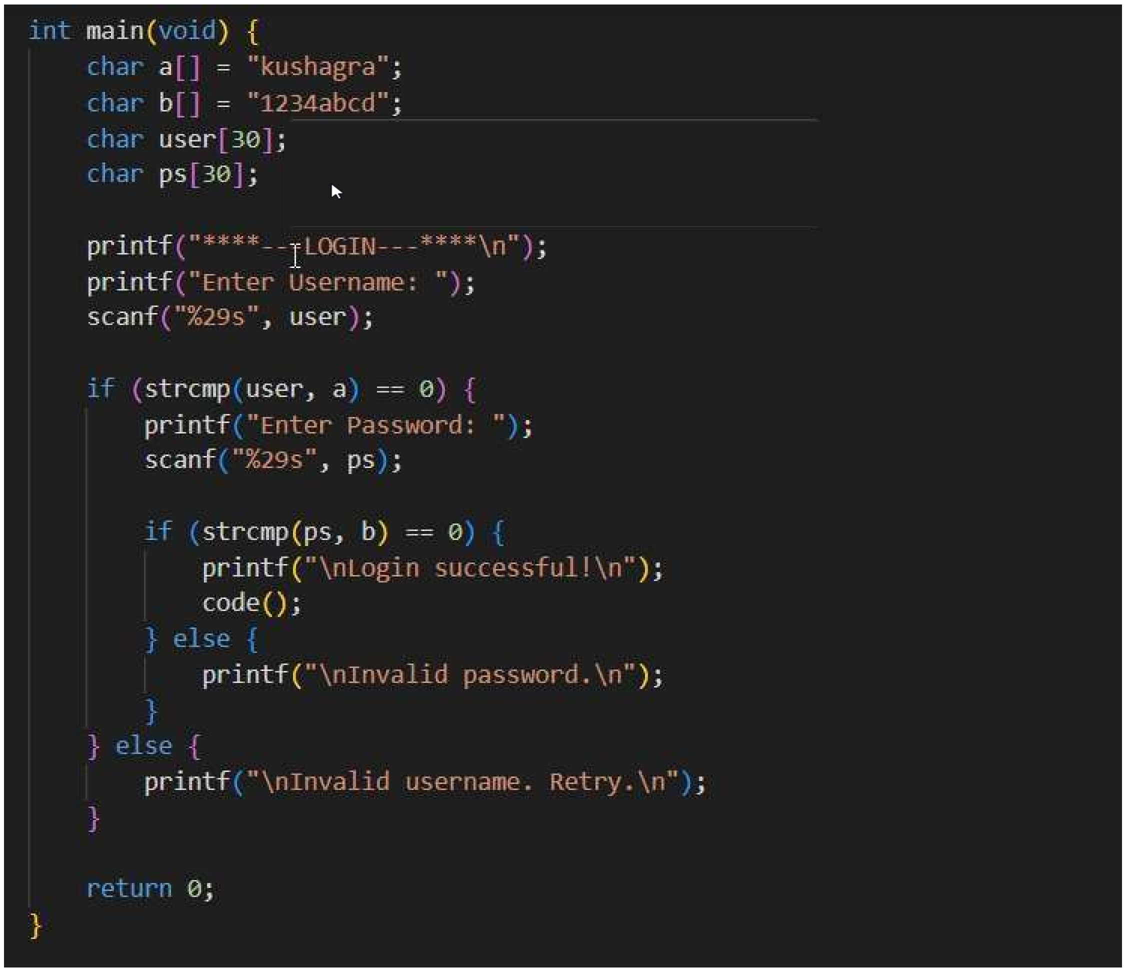
8. Increasing length
9. Multiple character sets
10. Millions of combinations
11. Recursive function call

Module 4: Time Measurement

Uses clock_t start = clock();.

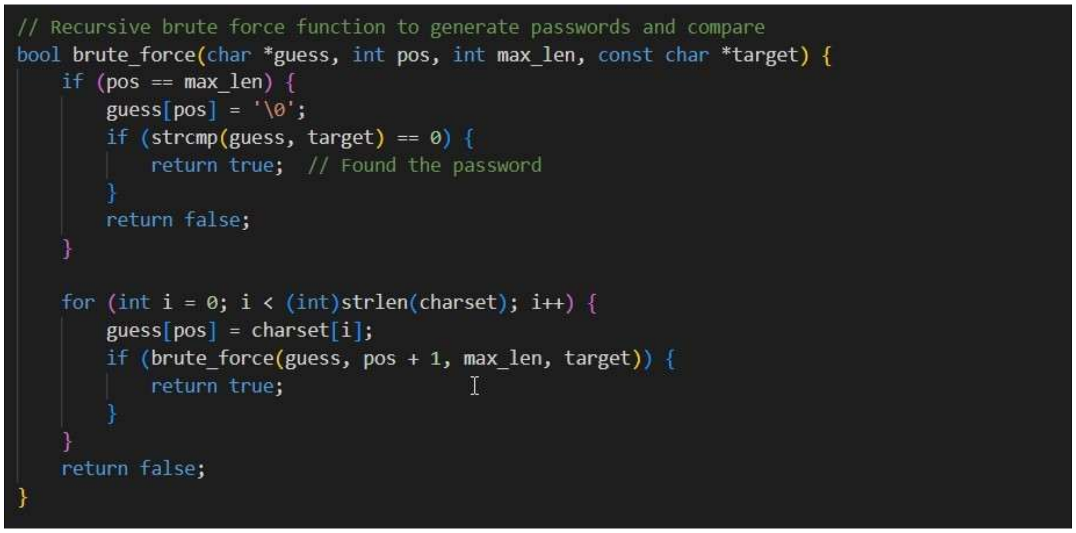
5.0 Implementation Snippets

1. Credential Search Snippet



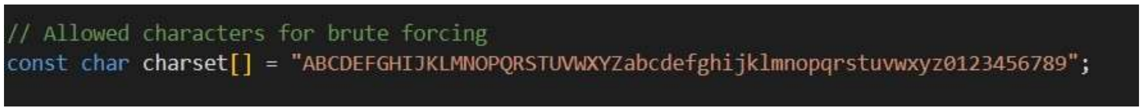
```c
int main(void) {
    char a[] = "kushagra";
    char b[] = "1234abcd";
    char user[30];
    char ps[30];

    printf("****---LOGIN---****\n");
    printf("Enter Username: ");
    scanf("%29s", user);

    if (strcmp(user, a) == 0) {
        printf("Enter Password: ");
        scanf("%29s", ps);

        if (strcmp(ps, b) == 0) {
            printf("\nLogin successful!\n");
            code();
        } else {
            printf("\nInvalid password.\n");
        }
    } else {
        printf("\nInvalid username. Retry.\n");
    }

    return 0;
}
```

2. Recursive Brute-Force Generator



```c
// Recursive brute force function to generate passwords and compare
bool brute_force(char *guess, int pos, int max_len, const char *target) {
    if (pos == max_len) {
        guess[pos] = '\0';
        if (strcmp(guess, target) == 0) {
            return true;  // Found the password
        }
        return false;
    }

    for (int i = 0; i < (int)strlen(charset); i++) {
        guess[pos] = charset[i];
        if (brute_force(guess, pos + 1, max_len, target)) {
            return true;
        }
    }
    return false;
}
```

3. Character Set



```
// Allowed characters for brute forcing
const char charset[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
```

5.3 Time Complexity Analysis

If:

1. Character set = 62
2. Password length

   = L Then:

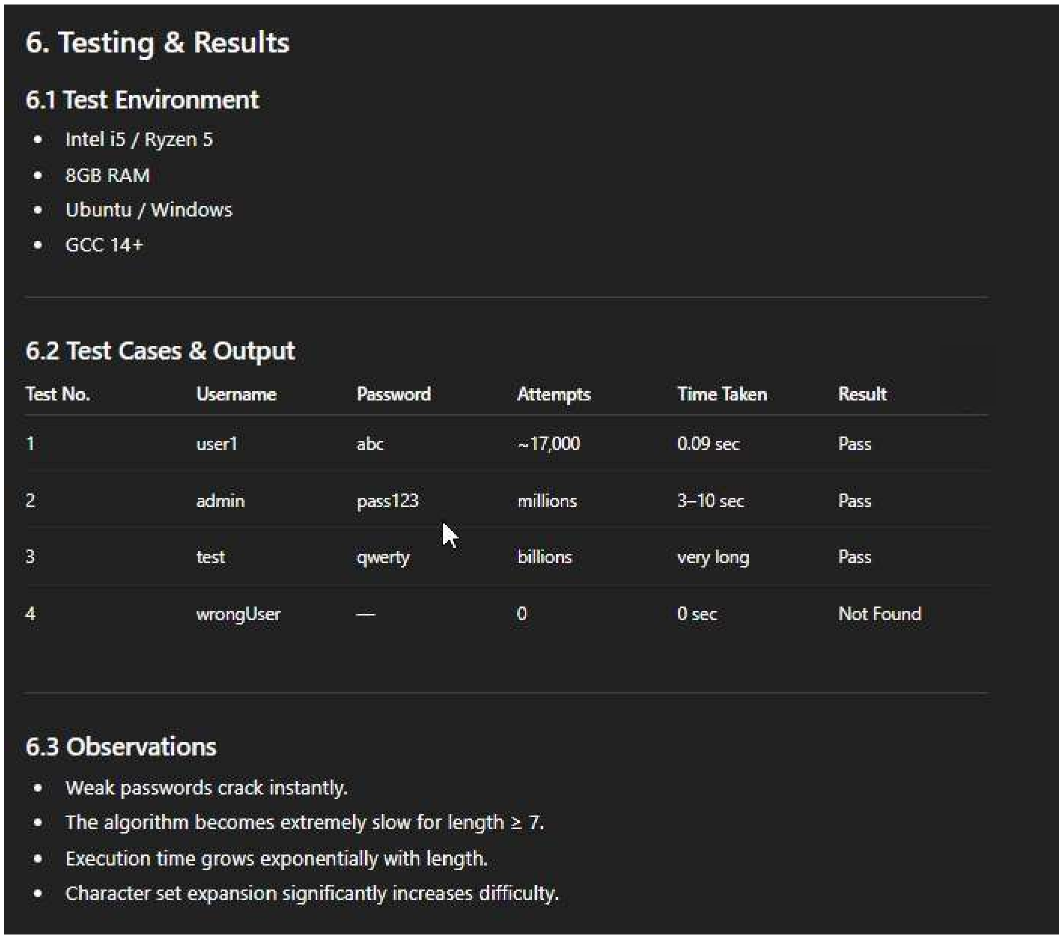
Total combinations = 62^L

Examples:

3. Length 3 → 238k attempts
4. Length 4 → 14.7M attempts
5. Length 6 → 56.8B attempts
6. Length 8 → 218 trillion

   attempts This is why long

   passwords are safer.



6. Testing & Results

6.1 Test Environment

- Intel i5 / Ryzen 5
- 8GB RAM
- Ubuntu / Windows
- GCC 14+

6.2 Test Cases & Output

Test No.	Username	Password	Attempts	Time Taken	Result
1	user1	abc	~17,000	0.09 sec	Pass
2	admin	pass123	millions	3–10 sec	Pass
3	test	qwerty	billions	very long	Pass
4	wrongUser	—	0	0 sec	Not Found

6.3 Observations

- Weak passwords crack instantly.
- The algorithm becomes extremely slow for length ≥ 7.
- Execution time grows exponentially with length.
- Character set expansion significantly increases difficulty.

6.0 Conclusion G Future Work

Conclusion

This project successfully demonstrates:

7. Why weak passwords are dangerous
8. How brute-force attacks work internally
9. The exponential growth in brute-force time
10. The importance of secure storage
    Plaintext passwords provide zero security. Even basic brute-force tools can crack them.

Future Work

The system can be upgraded by adding:

Security Enhancements

11. Hashing with SHA-256 / bcrypt
12. Salting techniques
13. Peppering
14. Two-Factor

Authentication (2FA) Feature

Enhancements

15. Multi-threaded brute-force
16. GPU acceleration
17. Dictionary + hybrid mode
18. GUI dashboard showing

attempts per second Research-Oriented

Additions

19. Comparing brute-force performance across algorithms
20. Studying password entropy mathematically
21. Building a password strength analyser

7.0 References

BOOKS:

1. KGR, *The C Programming Language*
2. OWASP Password Storage Cheat Sheet

WEBSITES:

1.NIST SP 800-63B Guidelines

2.GeeksForGeeks – Brute Force Algorithms

3. Tutorialspoint – File Handling in C
4. RFC 8252: OAuth Security Best Practices