Password Security Assessment Tool With Breach Detection

Naveen Daniel M[1],Livin Jospher [2],Kamalesh P[3],Anbuselvan B[4],Devika R[5]
[1, 2, 3, 4] Dept of Cybers ecurity
[5]Assistant Professor, Dept of Cybers ecurity
[1, 2, 3, 4, 5] Dhanalakshmi Srinivasan University

Abstract- *With the increasing risk of account takeovers and credential misuse, passwords remain the primary factor that gets compromised in data breaches. This project presents PSAWBD — a Password Security Assessment Tool with Breach Detection, developed entirely in Python. The tool evaluates password strength using common criteria including length, character diversity. It checks the user's password against a locally stored database of previously breached password hashes using SHA-1 hashing to protect the user's credentials during comparison. Additionally, it simulates real-world attack methods — dictionary attacks and rockyou-list-based attacks — using multi-threaded execution to test the actual resilience of the password against offline cracking. The results are delivered through an animated, text-based command-line interface, making the tool lightweight, portable, and easy to use. This project promotes better password hygiene and raises cybersecurity awareness among everyday users.*

Keywords: password security, strength analysis, strength analysis with breach detection, breach detection tool

I. INTRODUCTION

PSAWBD is a Python-based command-line tool designed to assess the real-world security of user passwords through three complementary modules: Strength Analysis, Breach Detection, and Attack Simulation. Unlike superficial password meters found in most web applications, this tool performs multi-dimensional analysis — evaluating passwords against standard criteria, checking them against breached password databases, and stress-testing them using actual wordlists such as rockyou.txt

II. PROBLEM STATEMENT

Most password strength indicators provide only surface-level feedback such as "weak," "medium," or "strong," without explaining the underlying vulnerabilities or verifying whether the password has already been exposed in a real-world breach. Users have no easy way to understand how quickly their password could be cracked using commonly available attack tools and wordlists.

There is a clear need for a tool that performs structured, multi-criteria password strength evaluation, verifies passwords against a local breached password hash database using secure hashing, simulates dictionary and wordlist-based attacks using threading to reflect realistic cracking scenarios

III. OBJECTIVES

- To evaluate password strength using common criteria including length, character types, and presence in weak password databases
- To detect whether a given password appears in a known data breach dataset by comparing SHA-1 hashes against a local database
- To simulate dictionary and rockyou-list attacks in parallel using Python threading and measure the time taken
- To deliver an overall password security score and verdict through a clean, animated CLI output
- To build the entire tool in Python without any external web frameworks or paid APIs

IV. LITERATURE SURVEY

Password Strength Metrics: Research by Komanduri et al. (2011) established that password length and character diversity strongly predict resistance to cracking. NIST Special Publication 800-63B recommends checking passwords against breach corpora as a primary security measure rather than enforcing complex composition rules alone.

Shannon Entropy in Password Analysis: Shannon entropy quantifies the unpredictability of a password. A password with higher entropy is harder to guess. Research by Shay et al. (2016) in *Designing Password Policies for Strength and Memorability* validated entropy as a practical metric for password security assessment.

Breached Password Databases: Troy Hunt's Have I Been Pwned (HIBP) service has aggregated billions of breached credentials from public data dumps. Local copies of the HIBP hash database are widely used in security tools to allow offline breach checking without transmitting the actual password.

Attack Simulation: Dictionary attacks and wordlist-based attacks (e.g., using rockyou.txt) represent the most common real-world offline cracking methods. Tools like Hashcat and John the Ripper demonstrate that weak passwords can be cracked within seconds using standard hardware.

V. SYSTEM DESIGN AND METHODOLOGY

SYSTEM ARCHITECTURE

The system is composed of four modules that run sequentially on the user's password:

Fig 1 : Sequential process of modules

MODULE DESIGN:

MODULE 1 – USER INPUT

The user input module is responsible for collecting the password from the user. It first displays the title banner and then prompts the user to enter their password through the command-line interface (CLI).

MODULE 2 – STRENGTH ANALYSIS

The strength module evaluates the password across three sub-checks:

Common Password Criteria :

Checks five properties and awards one point each:

Criterion	Points
Length > 12 characters	+1
Contains special characters	+1
Contains uppercase letters	+1
Contains lowercase letters	+1
Contains digits	+1

Score interpretation: 0–1 = Weak, 2–3 = Medium, 4 = Strong, 5 = Very Strong

Weak Password Database:

The weak password database module stores a collection of globally known weak passwords, including entries from sources like the RockYou password list. During execution, the module scans the rockyou.txt file line by line to check whether the user's password matches any of the commonly used passwords. If the password is found in the list, it is considered unsafe and a score of 0 is returned. If the password is not found, indicating it is less common and potentially safer, the module returns a score of 1. This process helps users understand whether their password is widely known or easily guessable, promoting stronger password choices.

Dictionary Word Check:

The dictionary word database module contains a collection of common English words used to test the user's password against simple dictionary-based attacks. During execution, the module loads the dictionary_words.txt file and checks whether the entered password exactly matches any plain dictionary word. If the password is found in the list, it is considered weak and a score of 0 is returned. If the password is not found, indicating better resistance to dictionary attacks, the module returns a score of 1. This helps users avoid easily predictable passwords that are commonly targeted by attackers.

Overall Score:

Sum of the three sub-scores (0–3) maps to: Very Weak → Weak → Medium → Strong.

MODULE 3 – BREACH DETECTION :

Steps performed:

1. The password is hashed using SHA-1 (hashlib.sha1) and converted to uppercase
2. The local pwnedpasswords.txt database is searched using **binary search** (seek-based) for efficiency
3. If the hash is found, the breach count is displayed; otherwise, the password is reported as safe

Binary search is used because the breached password database can contain billions of entries — a linear scan would be impractical. The f.seek() method is used to jump to the midpoint of the file, skip the partial line, and compare hashes until a match or exhaustion.

MODULE 4 – ATTACK SIMULATION:
Two attack threads are launched simultaneously using Python's threading:

The Dictionary Attack module reads the dictionary_words.txt file line by line and computes the SHA-1 hash of each word. The generated hash is then compared with the target password hash to determine whether the password can be cracked using common dictionary words. The module also records the total time taken for the attack process.

The RockYou Attack module scans the rockyou.txt file line by line, generates SHA-1 hashes for each password entry, and compares them against the target hash. This simulates a real-world password cracking attempt using one of the most widely known leaked password datasets. The execution time for the attack is also measured and stored.

Both attack modules run simultaneously using multi-threaded execution, improving efficiency and reducing processing time. After completion, the results — including whether the password was found and the time taken for each attack — are displayed through the terminal interface.

KEY IMPLEMENTATION DETAILS

Entry Point (psawbd.py): The main function Init() displays the animated PSAWBD banner, collects the user's password, and calls strengthCheck(), checkBreach(), and simulateAttack() in sequence. A user_option() function is also defined for optional individual module selection.

Animation (animations.py): The TextAnimation() function prints each character of a string one at a time with a configurable delay (default 0.09 seconds), creating a typewriter animation effect in the terminal.

Binary Search in Breach Detection (breachcheck.py): Rather than scanning the entire database linearly, the breach checker uses f.seek() to implement file-level binary search — dramatically reducing lookup time on a database of millions of SHA-1 hashes.

Threaded Attack Simulation (attackpassword.py): Two threading.Thread objects are created targeting dictionaryAttack and Rockyoulist functions respectively. Both are started simultaneously and compare SHA-1 hashes of candidate words against the target hash, printing the cracked word and elapsed time if found.

VI. TESTING

TESTING STRATEGY
Unit testing was performed on each module independently before integration. The following test cases were used:

STRENGTH ANALYSIS TEST CASES

PASSWORD INPUT	COMMON CRITERIA	WEAK DB	DICTIONARY	OVERALL SCORE
password	Medium	Found	Not Found	Weak
Password	Medium	Found	Not Found	Weak
P@ssw0rd!	Very Strong	Found	Not Found	Medium
Xk9#mLp2!qZ	Very Strong	Not Found	Not Found	Strong

BREACH DETECTION TEST CASES

Password	SHA-1 Hash Prefix	Result
password	5BAA6	Found in breach (3.8M times)
123456	7C4A8	Found in breach
Xk9#mLp2!qZ	(custom)	Not found in breach

ATTACK SIMULATION TEST CASES

Password	Dictionary Attack	Rockyou Attack
password	Found	Found
hello	Found	Found
Xk9#mLp2!qZ	Not Found	Not Found

PERFORMANCE TESTING

Module	Operation	Observed Time
Strength Check	Full evaluation	< 2 seconds
Breach Detection	Binary search in local DB	~1–3 seconds
Attack Simulation (Dictionary)	Line-by-line SHA-1 comparison	Varies with list size

Module	Operation	Observed Time
Attack Simulation (Rockyou)	Line-by-line comparison SHA-1	Varies with list size

VII. RESULTS

SAMPLE RUN OUTPUT:

Input password: password
----------PSAWBD----------

Enter your password : password

Common Password Criteria : Medium
Weak password Database : Found
Dictionary Word : Not Found
Overall Score : Weak

Checking database....
Found in data breach 3861493 times!

Dictionary attack: Dictionary word : password
[Time] Dictionary: 0.0031 sec

Rockyou attack: Hash Found : password
[Time] Rockyou: 0.0012 sec
Input password: Xk9#mLp2!qZ
Common Password Criteria : Very strong
Weak password Database : Not Found
Dictionary Word : Not Found
OverallStrength : Strong

Checking database....
Not found in a breach!

Dictionary attack failed
[Time] Dictionary: 0.87 sec
Rockyou attack failed
[Time] Rockyou: 4.23 sec

Fig 2 Sample output

Discussion:

The results clearly demonstrate that simple, common passwords like password are immediately identified as weak across all three analysis dimensions — they score poorly on criteria, appear in the breach database millions of times, and are cracked by dictionary and rockyou attacks in milliseconds. In contrast, complex passwords with length, mixed characters, and no dictionary basis survive all three checks, highlighting the real security benefit of strong password choices. The threaded attack simulation effectively demonstrates the danger of offline cracking to users in a practical and relatable way.

VIII. CONCLUSION AND FUTURE ENHANCEMENT

Conclusion

PSAWBD successfully delivers a multi-dimensional password security assessment entirely within a Python CLI environment. By combining rule-based strength analysis, local breached hash lookup with binary search, and parallel dictionary/rockyou attack simulation, the tool provides users with a realistic, data-driven picture of their password's security posture. The animated terminal interface makes the tool engaging and accessible, while the modular architecture keeps each analysis function independent and maintainable. This project reinforces the importance of evaluating passwords not just structurally, but against real-world attack data.

Future Enhancements

- Integrate the HIBP k-anonymity API for real-time online breach checking without storing a local database
- Add Shannon entropy calculation to the strength analysis for a more rigorous mathematical assessment
- Implement a brute-force simulation module with estimated crack time based on character set and password length
- Develop a web-based front end using Flask or Streamlit for broader, non-technical user accessibility
- Add a secure password generator module that creates strong, memorable passphrases
- Support for batch password auditing (CSV input) for organizational security audits
- Add bcrypt/Argon2 hashing support in addition to SHA-1 for breach checks

IX. APPENDIX

SOURCE CODE

animations.py

```
from time import sleep, time
loading = ['|', '/', '-', '\\', '|']

def TextAnimation(text_input, seconds=0.09):
   text = text_input
   print('\n')
   for i in text:
print(i, end="", flush=True)
      sleep(seconds)
   print('\n')
```

psawbd.py

```
from time import sleep
from strengthcheck import strengthCheck
from breachcheck import checkBreach
from attackpassword import simulateAttack
from animations import TextAnimation as Aprint

def Init():
   try:
Aprint("----------------------------------PSAWBD----------------------------------", 0.01)
user_password = input("Enter your password : ")
sleep(0.5)
   except Exception as e:
      print(e)

   def user_option():
user_option = int(input("'Choose a option \n 1.Strength Analysis \n 2.BreachDetection  \n 3.Attack simulation \n \n \n'"))
      if user_option == 1:
strengthCheck(user_password)
elifuser_option == 2:
checkBreach(user_password)
elifuser_option == 3:
simulateAttack(user_password)
      else:
print("Choose a valid option")
user_option()

strengthCheck(user_password)
checkBreach(user_password)
simulateAttack(user_password)

if __name__ == "__main__":
Init()
```

strengthcheck.py

```
import string
import os
from time import sleep
from animations import TextAnimation as Aprint

os.chdir("C:/Users/user/psawbd/")

def strengthCheck(password):

   def CommonPasswordCriteria(user_password):
      score = 0
passwordLength = 1 if len(user_password) > 12 else 0
specialchars = 1 if any(char in string.punctuation for char in user_password) else 0
      upper = 1 if any(char in string.ascii_uppercase for char in user_password) else 0
      lower = 1 if any(char in string.ascii_lowercase for char in user_password) else 0
      numeric = 1 if any(char in string.digits for char in user_password) else 0
calculativePoints = passwordLength + specialchars + upper + lower + numeric
sleep(1)
      if calculativePoints< 2:
print("Common Password Criteria : ", "Weak")
elifcalculativePoints<= 3:
print("Common Password Criteria : ", "Medium")
elifcalculativePoints< 5:
print("Common Password Criteria : ", "Strong")
elifcalculativePoints == 5:
print("Common Password Criteria : ", "Very strong")
         return score + 1
      return score

   def WeakPassDatabase(user_password):
      score = 0
      with open("rockyou.txt", "r", encoding="utf-8", errors="ignore") as rockyou:
         found = False
         for line in rockyou:
            if user_password == line.strip():
               found = True
               break
      if found:
print("Weak password Database : ", "Found \n")
         return score
      else:
print("Weak password Database : ", "Not Found \n")
         return score + 1

   def DictionaryWords(user_password):
```

```
        score = 0
        found = False
        with open("dictionary_words.txt", "r") as dic_words:
            words = dic_words.read()
            wordlist = list(words.split())
            for word in wordlist:
                if user_password == word:
                    found = True
        if found:
print("Dictionary Word : ", "Found \n")
            return score
        else:
print("Dictionary Word : ", "Not Found \n")
            return score + 1

    def OverallStrength():
cpc_score = CommonPasswordCriteria(password)
        _sleep()
wpd_score = WeakPassDatabase(password)
        _sleep()
dw_score = DictionaryWords(password)
        _sleep()
        overall = cpc_score + wpd_score + dw_score
sleep(1)
        if overall == 0:
Aprint("Overall Score : Very Weak")
elif overall == 1:
Aprint("Overall Score : Weak")
elif overall == 2:
Aprint("Overall Score : Medium")
elif overall == 3:
Aprint("Overall Strength :  Strong")

OverallStrength()
sleep(2)

def _sleep():
sleep(0.5)
```

breachcheck.py

```
import hashlib
import os
from animations import TextAnimation as Aprint
from time import sleep

def checkBreach(user_password):

    def hashing(password):
pswdHash = hashlib.sha1(password.encode()).hexdigest().upper()
        return pswdHash

    def searchDatabase(paswdhash):
os.chdir("C:/Users/user/")
Aprint("Checking database....")
        with open("pwnedpasswords.txt", "r", encoding="utf-8", errors="ignore") as f:
            low = 0
f.seek(0, 2)
            high = f.tell()
            while low <= high:
                mid = (low + high) // 2
f.seek(mid)
f.readline()
pos = f.tell()
                line = f.readline()
                if not line:
                    high = mid - 1
                    continue
current_hash, *rest = line.strip().split(":")
                if current_hash == paswdhash:
                    count = rest[0] if rest else "0"
sleep(1)
Aprint(f"Found in data breach {count} times!")
                    return
elifcurrent_hash<paswdhash:
                    low = pos
                else:
                    high = mid - 1
Aprint("Not found in a breach!")

searchDatabase(hashing(user_password))
```

attackpassword.py

```
import threading
import hashlib
import time
import os
from animations import TextAnimation as Aprint

def simulateAttack(user_password):
os.chdir("C:/Users/user/psawbd/")
hashedpassword = hashlib.sha1(user_password.encode()).hexdigest()

    def dictionaryAttack(password_hash):
starttime = time.perf_counter()
        found = False
        with open("dictionary_words.txt", "r", encoding='utf-8', errors="ignore") as dictext:
            for word in dictext:
                word = word.strip()
```

```
            if hashlib.sha1(word.encode()).hexdigest() == password_hash:
                found = True
print(f"Dictionaryword : {word}")
                break
endtime = time.perf_counter()
      if not found:
print("Dictionary attack failed")
      print(f"[Time] Dictionary: {endtime - starttime:.4f} sec\n")

   def Rockyoulist(password_hash):
starttime = time.perf_counter()
      found = False
      with open("rockyou.txt", "r", encoding='utf-8', errors="ignore") as dictext:
          for word in dictext:
             word = word.strip()
             if hashlib.sha1(word.encode()).hexdigest() == password_hash:
                found = True
print(f"HashFound : {word}")
                break
endtime = time.perf_counter()
      if not found:
print("Rockyou attack failed")
      print(f"[Time] Rockyou: {endtime - starttime:.4f} sec\n")

   t1 = threading.Thread(target=dictionaryAttack, args=(hashedpassword,))
   t2 = threading.Thread(target=Rockyoulist, args=(hashedpassword,))
   t1.start()
   t2.start()
```

REFERENCES

[1] NIST Special Publication 800-63B – Digital Identity Guidelines, National Institute of Standards and Technology, 2017.

[2] Komanduri, S. et al., Of Passwords and People: Measuring the Effect of Password-Composition Policies, CHI 2011.

[3] Shay, R. et al., Designing Password Policies for Strength and Memorability, CHI 2016.

[4] Troy Hunt, Have I Been Pwned: Pwned Passwords, https://haveibeenpwned.com/Passwords

[5] A Password Strength Evaluation Algorithm Based on Sensitive Personal Information, IEEE.

[6] On Password Strength Measurements: Password Entropy and Password Quality, IEEE.

[7] Python Software Foundation, Python 3 Documentation, https://docs.python.org/3/