Internship Report: Password Cracker Using Python

Intern Name: Prem Kumar

Project Title: Password Cracker Using Python

Internship Organization: Inlighn Tech

Abstract

The “Password Cracker Using Python” project was designed as part of my internship at **Inlighn Tech**, focusing on understanding the real-world implications of password hashing and cracking mechanisms. The goal was to develop a command-line tool in Python capable of recovering plaintext passwords from cryptographic hashes using **dictionary attacks** and **brute-force techniques**. The tool supports multiple hash types (like MD5, SHA-1, SHA-256) and incorporates **multithreading** to optimize speed. This project allowed me to explore essential cybersecurity concepts including cryptographic hash functions, password entropy, and ethical password auditing.

1. Introduction

In today’s digital world, passwords are the most common method of user authentication. They are often stored in a **hashed format** to prevent unauthorized access in case of a data breach. However, weak passwords or predictable hashes can still be cracked using tools like this password cracker.

This project aims to simulate how attackers or penetration testers may try to recover passwords from their hashed representations using **wordlists** or **brute-force methods**. The project also provides a hands-on understanding of cryptographic practices, hash functions, threading for speed optimization, and ethical use of such tools.

2. Objectives

- Create a Python tool to recover plaintext passwords from hash values.
- Implement both **dictionary attack** and **brute-force attack** strategies.
- Support commonly used hash algorithms (e.g., MD5, SHA-1, SHA-256).
- Optimize the tool using **multithreading** for faster cracking.
- Display results clearly, along with performance statistics.
- Understand the ethical boundaries of password cracking tools.

3. Tools and Technologies Used

Tool/Library	Description
Python 3.x	Main programming language
`hashlib`	For generating cryptographic hash values
`itertools`	To generate all possible combinations of passwords
`threading` / `concurrent.futures`	For parallel password testing
Wordlist (e.g., `rockyou.txt`)	Dictionary for common password guessing

4. Project Workflow

Step 1: Input Parameters

The user provides:

- Target hashed password
- Hash algorithm (e.g., `md5`, `sha1`, `sha256`)
- Either a **wordlist path** or character set + min/max lengths for brute-force

Step 2: Hash Matching

For each generated or guessed password, the tool:

- Hashes the guess using the chosen algorithm
- Compares the resulting hash to the target hash
- Stops when a match is found or all options are exhausted

Step 3: Modes of Attack

1. **Dictionary Attack:**

Checks each word from a wordlist against the hash.

2. **Brute-force Attack:**

Generates all possible combinations of characters within a length range.

Step 4: Multi-threaded Execution

Each password guess is run in parallel threads using Python's `ThreadPoolExecutor` to improve speed.

5. Code Implementation

Install Required Modules

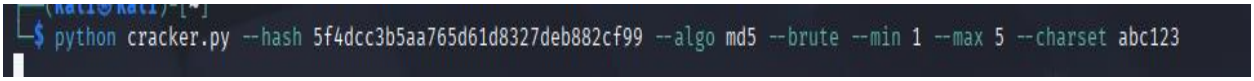
Full Python Code Example

```
└─$ import hashlib
import itertools
import argparse
from concurrent.futures import ThreadPoolExecutor
from tqdm import tqdm

# Generate hash for given password and algorithm
def generate_hash(password, algo):
    h = hashlib.new(algo)
    h.update(password.encode())
    return h.hexdigest()

# Attempt password match
def check_password(password, target_hash, algo):
    return generate_hash(password, algo) == target_hash

# Dictionary attack
def dictionary_attack(target_hash, hash_algo, wordlist):
    with open(wordlist, 'r') as f:
        words = [line.strip() for line in f]

    with ThreadPoolExecutor(max_workers=10) as executor:
        for password in tqdm(words, desc="Trying passwords"):
            if executor.submit(check_password, password, target_hash, hash_algo).result():
                print(f"\n✅ Password cracked: {password}")
                return
    print("❌ Password not found in wordlist.")

# Brute-force attack
def brute_force_attack(target_hash, hash_algo, charset, min_len, max_len):
    for length in range(min_len, max_len + 1):
        combinations = itertools.product(charset, repeat=length)
        for combo in tqdm(combinations, desc=f"Length {length}"):
            password = ''.join(combo)
            if check_password(password, target_hash, hash_algo):
                print(f"\n✅ Password cracked: {password}")
                return
    print("❌ Password not found using brute-force.")

# CLI parser
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Password Cracker Tool")
    parser.add_argument("--hash", required=True, help="Target hashed password")
    parser.add_argument("--algo", required=True, help="Hash algorithm (md5, sha1, sha256)")
    parser.add_argument("--wordlist", help="Path to wordlist file")
    parser.add_argument("--brute", action="store_true", help="Use brute-force attack")
    parser.add_argument("--min", type=int, default=1, help="Min password length for brute-force")
    parser.add_argument("--max", type=int, default=4, help="Max password length for brute-force")
    parser.add_argument("--charset", default="abcdefghijklmnopqrstuvwxyz0123456789", help="Charset for brute-force")

    args = parser.parse_args()

    if args.wordlist:
        dictionary_attack(args.hash, args.algo, args.wordlist)
    elif args.brute:
        brute_force_attack(args.hash, args.algo, args.charset, args.min, args.max)
    else:
        print("⚠<fe0f> Provide either a wordlist or brute-force option.")
```

6. Usage Examples

Dictionary Attack

Brute-Force Attack



7. Key Concepts Demonstrated

Cybersecurity Concept	Explanation
Hash Functions	One-way encryption for storing passwords
Dictionary Attack	Uses prebuilt list of common passwords
Brute-Force Attack	Tries all combinations of characters
Entropy	Measure of password strength
Multi-threading	Enhances speed by parallel processing
Ethical Hacking	The tool is intended for education and awareness only

8. Learning Outcomes

- Strong understanding of how **hashing algorithms** work
- Ability to simulate and defend against **real-world attack strategies**
- Improved **Python scripting and performance optimization**
- Knowledge of the legal boundaries of penetration testing
- Hands-on experience with threading and automation tools

9. Limitations

- Very long passwords take considerable time with brute-force
- Doesn't support salt hashing or peppered hashes
- Can’t crack password hashes using rainbow tables
- Doesn’t support hash validation via external API or live websites

10. Future Enhancements

- Add support for salted password hashes
- Integrate GPU acceleration (using PyCUDA or Hashcat API)
- Save cracked passwords and attempts to logs
- Add support for online hash verification (API lookup)
- □ Develop a GUI using Tkinter or PyQt for usability

11. Ethical Considerations

This project was implemented in a controlled lab environment. Any form of password cracking should be done **only with permission** or for **educational/testing** purposes. Unauthorized cracking of real-world accounts is illegal and punishable by cybercrime laws.

12. Conclusion

The Password Cracker Using Python project was a rewarding learning experience that helped me understand the workings of hashing, password vulnerabilities, and automated penetration testing tools. By simulating real-world attacks, I gained valuable insight into how attackers exploit weak security, and how ethical hackers defend against it.

This tool has practical applications in red teaming, password audit simulations, and educational labs, and sets the foundation for deeper work in cryptography, secure authentication systems, and ethical hacking.