INLIGHN TECH

EXPERIENCE. LEARN. THRIVE

Prerequisites for the Password Cracker Project

By: **Inlighn Tech**

Overview

The Password Cracker project requires students to develop a Python script that cracks a given hash by either testing passwords from a wordlist or generating them on the fly, using multithreading for efficiency. The solution leverages the hashlib library for hash functions, itertools for password generation, concurrent.futures for parallel execution, tqdm for progress tracking, and argparse for command-line arguments. To write this script independently, students must master several key concepts and skills from the curriculum. These prerequisites, paired with practical Python code examples, ensure students understand how hash cracking works and can implement the script effectively.

Prerequisite Knowledge and Skills

1. **Cryptographic Hashes – Understanding Hash Functions**
    - **What to Know**:

        - Hash functions (e.g., MD5, SHA-256) convert input (e.g., a password) into a fixed-length string (hash) that's unique to the input.
        - Cracking involves guessing passwords, hashing them, and comparing them to a target hash.
        - Common hash types include md5, sha1, sha256, etc., supported by hashlib.
    - **How It Applies**:
        - The script uses hashlib to compute hashes of passwords and compare them to the target hash in check_hash().
    - **Curriculum Source**:
        - Not explicitly listed; assumes basic cybersecurity knowledge (implied in "NETWORKING, SOCKETS, AND CYBERSECURITY" context).
    - **Practical Prep**:
        - Test hashing manually with a tool like echo -n "password" | md5sum (Linux).
    - **Python Code Block**:

```
# Compute a simple hash
import hashlib

password = "test123"
hash_fn = hashlib.md5  # Example hash function
hashed = hash_fn(password.encode()).hexdigest()
print(f"Password: {password}, MD5 Hash: {hashed}")
```

        - **Run It**: Save as hash_test.py, run with python hash_test.py. Shows how to hash a password, preparing for check_hash().

2. **File Input/Output (I/O) Basics**
    - **What to Know**:
        - Read text files (e.g., wordlists) using open() in text mode ('r').
        - Use list comprehension or loops to process lines.

- **How It Applies**:
  - The script reads passwords from a wordlist file in crack_hash() for testing against the hash.
- **Curriculum Source**:
  - "PYTHON BASICS" Lesson 11 (IO with Basic Files).
- **Practical Prep**:
  - Practice reading a wordlist.
- **Python Code Block**:

```
# Read a wordlist file
with open("wordlist.txt", "w") as f:
    f.write("pass1\npass2\npass3\n")  # Create sample wordlist

with open("wordlist.txt", "r") as f:
    passwords = [line.strip() for line in f.readlines()]
    print("Loaded passwords:", passwords)
```

  - **Run It**: Save as file_io_test.py, run with python file_io_test.py. Mimics wordlist loading in crack_hash().

3. **Functions, Generators, and Itertools**
   - **What to Know**:
     - Define functions with def and use yield for generators to produce values lazily (e.g., generate_passwords).
     - Use itertools.product() to generate all possible combinations of characters.
   - **How It Applies**:
     - generate_passwords() yields password combinations, enabling brute-force cracking without storing all possibilities in memory.
   - **Curriculum Source**:
     - "PYTHON BASICS" Lessons 17-19 (Introduction to Functions, def Keyword, Basics of Functions).
     - Note: Generators and itertools may need a brief intro (implied in advanced Python use).

- **Practical Prep**:
    - Test password generation.
- **Python Code Block**:

```
# Generate passwords with itertools
import itertools

def gen_passwords(chars, length):
    for combo in itertools.product(chars, repeat=length):
        yield ''.join(combo)

chars = "ab"
for pwd in gen_passwords(chars, 2):
    print(pwd)  # Outputs: aa, ab, ba, bb
```

-
    - **Run It**: Save as generator_test.py, run with python generator_test.py. Prepares for generate_passwords().

4. **Command-Line Arguments with argparse**
    - **What to Know**:
        - argparse parses command-line inputs with required (e.g., hash) and optional arguments (e.g., --wordlist).
        - Set defaults and access via args.attribute.
    - **How It Applies**:
        - The script uses argparse to handle hash, wordlist, hash type, and generation options.
    - **Curriculum Source**:
        - Extends "SSH Cracker" Lesson 4 (User Input); assumes basic Python knowledge.
    - **Practical Prep**:
        - Practice argument parsing.
    - **Python Code Block**:

```
# Parse command-line arguments
import argparse
```

```
parser = argparse.ArgumentParser(description="Test parser")
parser.add_argument("input", help="Input string")
parser.add_argument("--option", default="default", help="Optional value")
args = parser.parse_args()

print(f"Input: {args.input}, Option: {args.option}")
```

-
    - **Run It**: Save as args_test.py, run with python args_test.py test --option custom. Shows argparse usage, like the script's setup.

5. **Multithreading with concurrent.futures**
    - **What to Know**:
        - concurrent.futures.ThreadPoolExecutor runs tasks in parallel; submit() schedules tasks, and result() retrieves outcomes.
        - max_workers controls thread count.
    - **How It Applies**:
        - The script uses ThreadPoolExecutor to test multiple passwords concurrently, speeding up cracking.
    - **Curriculum Source**:
        - "Port Scanner" Lesson 3 ("Working with Threads.mp4"); builds on "Network Scanner" threading.
    - **Practical Prep**:
        - Test thread pools.
    - **Python Code Block**:

```
# Use ThreadPoolExecutor for parallel tasks
from concurrent.futures import ThreadPoolExecutor
import time

def task(n):
   time.sleep(0.5)
   return f"Task {n} done"

with ThreadPoolExecutor(max_workers=2) as executor:
   futures = [executor.submit(task, i) for i in range(4)]
   for future in futures:
      print(future.result())
```

-
    - **Run It**: Save as threadpool_test.py, run with python threadpool_test.py. Prepares for crack_hash() threading.

6. Progress Tracking with tqdm
    - **What to Know**:
        - tqdm displays progress bars for loops or iterators; total sets the expected count.
        - Update manually with pbar.update() for generators.
    - **How It Applies**:
        - The script uses tqdm to show cracking progress for wordlist or generated passwords.
    - **Curriculum Source**:
        - Not explicitly taught; assumes a brief intro (common in Python projects).
        - Note: Students need pip install tqdm.
    - **Practical Prep**:
        - Test a progress bar.
    - **Python Code Block**:

```
# Show progress with tqdm
from tqdm import tqdm
import time

for i in tqdm(range(5), desc="Processing"):
    time.sleep(0.5)
print("Done")
```

-
    - **Run It**: Install tqdm (pip install tqdm), save as tqdm_test.py, run with python tqdm_test.py. Introduces progress tracking in crack_hash().

7. String and Character Sets with string Module
    - **What to Know**:
        - The string module provides character sets (e.g., string.ascii_letters, string.digits).

        - Combine sets for custom character pools (e.g., letters + digits).
    - **How It Applies**:
        - The script uses string constants to define the default character set for password generation.
    - **Curriculum Source**:
        - "WORKING WITH DATA STRUCTURES & FILE HANDLING" Lesson 1 (String Properties and Methods).
    - **Practical Prep**:
        - Explore character sets.
    - **Python Code Block**:

```
# Use string module for character sets
import string

chars = string.ascii_letters + string.digits
print(f"Character set: {chars[:20]}... (total {len(chars)})")
```

    -
        - **Run It**: Save as string_test.py, run with python string_test.py. Shows character set creation, like in crack_hash().

How Password Cracking Works

- **Concept**:
    - Password cracking guesses a password by hashing candidates (from a wordlist or generated) and matching them against a target hash.
    - This simulates brute-force or dictionary attacks to test hash security.
- **Script Workflow**:
    - Parse command-line arguments for hash, wordlist, hash type, or generation parameters.
    - Load passwords from a file or generate them with itertools.
    - Use ThreadPoolExecutor to hash and compare passwords in parallel.
    - Track progress with tqdm and return the matching password.
    - Handle invalid hash types or no matches gracefully.

- **Why Multithreading?**:
    - Speeds up cracking by testing multiple passwords simultaneously, critical for large wordlists or brute-force attempts.

How to Write the Password Cracker Script

Using these prerequisites, students can build the script step-by-step:

1. **Setup**:
    - Import hashlib, itertools, string, concurrent.futures, tqdm, and argparse.
    - Define hash_name list and generate_passwords() generator.
2. **Check Hash**:
    - Write check_hash() to hash a password with hash_fn and compare it to target_hash.
3. **Crack Function**:
    - Implement crack_hash() to validate hash_type, handle wordlist or generation, and use threads with progress tracking.
4. **Argument Parsing**:
    - Set up argparse with required (hash) and optional arguments (e.g., --wordlist, --min_length).
5. **Main Execution**:
    - Use if __name__ == "__main__": to parse args and run crack_hash().

Notes for Students

- **Setup**: Install tqdm (pip install tqdm); no other external libraries needed beyond Python standard library. Create a small wordlist.txt (e.g., pass1, 123, test) and a test hash (e.g., MD5 of "test": 098f6bcd4621d373cade4e832627b4f6).
- **Engagement**: Run each code block to practice hashing, generation, and threading. Test the script with python script.py 098f6bcd4621d373cade4e832627b4f6 -w wordlist.txt or python script.py 098f6bcd4621d373cade4e832627b4f6 --min_length 1 --max_length 4.

- **Tips**: Use short wordlists or small max_length initially to see quick results; adjust max_workers based on your CPU.

Jump to the Password Cracker Project Description if you are ready now.
