Computer Security

Assignment 1, Hacking Password

Yanapat Patcharawiwatpong

1 Introduction

This activity demonstrates the fundamentals of password security. Several hacking techniques will be explored throughout the exercises, including brute-force attacks, rainbow-table attacks, and password analysis.

2 Exercise

2.1 Hash Cracking

Task: Write a Python program to find the original value of the SHA-1 hash using the provided dictionary, including substitutions.

```python
import hashlib
import itertools

def case_permutations(word):
    substitutions = {"a": "@", "i": "1!", "l": "1Ii", "o": "0", "s": "$5"}

    def variations(char):
        lower = char.lower()
        return lower + char.upper() + substitutions.get(lower, '')

    perms = map(variations, word)
    return ("".join(p) for p in itertools.product(*perms))

def check_word(word, target_hash):
    for perm in case_permutations(word):
        if hashlib.sha1(perm.encode()).hexdigest() == target_hash:
            return perm
    return None

def main():
    target_hash = "d54cc1fe76f5186380a0939d2fc1723c44e8a5f7"

    with open("10k-most-common.txt", "r") as file:
        for line in file:
            word = line.strip()
```

```
            result = check_word(word, target_hash)
            if result:
                print(f"Password found: {result}")
                return

    print("Password not found in the dictionary.")

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

The Python program is designed to recover the original value of a SHA-1 hash by utilizing a dictionary of common passwords and applying various character substitutions to each password. It first generates permutations of each word by considering substitutions for instance, *'@' for 'a', '1!' for 'i', '0' for 'o', and other variations*. Each permutation is hashed using SHA-1 and compared against the target hash. By iterating through all possible permutations of each word in the dictionary, the program successfully identifies the password that matches the given hash. In this case, the result is **"ThaiLanD"**, demonstrating that the program correctly found the original password associated with the SHA-1 hash *d54cc1fe76f5186380a0939d2fc1723c44e8a5f7*.

2.2 Rainbow Table Creation

Task: Create a rainbow table using the SHA-1 algorithm, including substituted strings. Measure the time and size of the table.

```
# cython: boundscheck=False, wraparound=False, cdivision=True

import hashlib, itertools, time, sys
from cpython cimport array
import array

cdef dict subs = {"a": "@", "i": "1!", "l": "1Ii", "o": "0", "s": "$5"}

cdef bytes sha1_hash(str s):
    return hashlib.sha1(s.encode()).hexdigest().encode()

cdef dict hash_permutations(str word):
    cdef dict result = {}
    if not word.strip():
        return result

    cdef list chars = [c.lower() + c.upper() + subs.get(c.lower(), "") for c in word.strip()]
    for perm in itertools.product(*chars):
        perm_str = ''.join(perm)
        result[perm_str] = sha1_hash(perm_str)

    return result

cdef dict create_table(list words):
    cdef dict table = {}
```

```
    for word in words:
        table.update(hash_permutations(word))
    return table

def main():
    cdef list words = [line.strip() for line in open("../10k-most-common.txt")]
    cdef int num_trials = 10
    cdef double[:] times = array.array('d', [0] * num_trials)
    cdef long long[:] sizes = array.array('q', [0] * num_trials)
    cdef long long[:] mem_sizes = array.array('q', [0] * num_trials)

    for i in range(num_trials):
        print(f"Running trial {i + 1}/{num_trials}")
        start_time = time.time()
        table = create_table(words)
        end_time = time.time()

        times[i] = end_time - start_time
        sizes[i] = len(table)
        mem_sizes[i] = sys.getsizeof(table)

    avg_time = sum(times) / num_trials
    avg_size = sum(sizes) / num_trials
    avg_mem_size = sum(mem_sizes) / num_trials
    avg_hash_time = sum(times) / sum(sizes)

    print("\nAverage Results:")
    print(f"Average creation time: {avg_time:.4f} seconds")
    print(f"Average table size: {avg_size:.0f} entries")
    print(f"Average memory size: {avg_mem_size:.0f} bytes")
    print(f"Average hash time: {avg_hash_time:.9f} seconds")

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

This program measures the time required to create the rainbow table, the total number of entries generated, and the memory usage over multiple trials. By averaging the results across 10 trials, we gain valuable insights into the performance and resource demands of constructing such a table.

```
Average Results over 10 trials:
Average creation time: 68.9122 seconds
Average table size: 28212780 entries
Average memory size: 1480.14 MB
Average hash time : 0.000002443 seconds
```

These findings underscore the computational complexity and resource-intensive nature of password-cracking techniques like rainbow tables, emphasizing their effectiveness but also the considerable resources required for their implementation.

2.3 Hashing Performance Analysis

Task: Measure how long it takes to perform a SHA-1 hash on a password string. Analyze the performance.

In Section 2.2, the performance metrics for creating a rainbow table were discussed. For hashing individual passwords, the average hash time is:

- **Average hash time**: 0.000002443 seconds

This average hash time highlights the efficiency of the SHA-1 hashing process, emphasizing its low computational overhead per hash operation.

2.4 Brute-Force Password Cracking Time Estimate

Task: Estimate how long it takes to break a password with brute force using your computer.

The total number of possible characters is given by:

$$C = 26 \text{ (lowercase letters)} + 26 \text{ (uppercase letters)} + 10 \text{ (digits)} + 32 \text{ (special characters)} = 94 \text{ characters}$$

Given a password length $L = 10$, the number of possible permutations of the password is C^L:

$$94^{10} \approx 53.86 \times 10^{18} \text{ permutations}$$

The average time to perform a SHA-1 hash per character is $H = 2.43 \times 10^{-6}$ seconds (as mentioned in section 2.2). Therefore, the total time required to hash all possible permutations is calculated as:

$$\text{Total Time} = C^L \times H \approx 131,583,672,373,384.0 \text{ seconds}$$

This total time can be converted into different time units:

- **Minutes**: $\frac{131583672373384.0}{60} \approx 2,193,061,206,223.07$ minutes
- **Hours**: $\frac{131583672373384.0}{3600} \approx 36,551,020,103.72$ hours
- **Days**: $\frac{131583672373384.0}{86400} \approx 1,522,959,170.99$ days
- **Years**: $\frac{131583672373384.0}{31536000} \approx 4,172,490.88$ years

Analysis: The analysis reveals that even with a relatively small password length of 10 and a hash time of just 2.43 microseconds per character, the computational effort required to hash all possible permutations is immense. The total time required to hash all combinations would span over 4 million years, highlighting the robustness of modern cryptographic hash functions like SHA-1 in securing passwords through sheer computational complexity.

2.5 Proper Password Length

Task: Determine the minimum password length to ensure it takes at least one year to crack.

Given:

$$94^L \times 0.000002443 = 31{,}536{,}000 \text{ seconds}$$

Solve for L:

$$94^L = \frac{31{,}536{,}000}{0.000002443} \approx 1.29 \times 10^{13}$$

$$L \geq \frac{\log_{10}(1.29 \times 10^{13})}{\log_{10}(94)} \approx \frac{13.112}{1.973} \approx 6.65$$

Thus, a password length of **at least 7 characters** is recommended to ensure it takes at least one year to crack.

2.6 What is Salt?

Task: Explain the concept of salt and its role in protecting a password hash.

Salt is a random value added to the password before hashing. It ensures that even if two users have the same password, their hashes will be different. This protects against rainbow-table attacks because the salt changes the hash output, making precomputed hash tables ineffective. Salt increases the complexity for attackers, ensuring that each password hash is unique.