Pupipat Singkhorn 6532142421

Activity I : Hacking Password

Created by : Krerk Piromsopa, Ph.D Objectives To understand the concepts of hashing and salting. Overviews This activity demonstrates the fundamentals of password security. Several hacking techniques will be demonstrated throughout the exercises. In particular, we will learn: brute-force attack, rainbow-table attack, and password analysis. We will use a free password dictionary from the given url as our dictionary.
https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10k-most-common.txt

Prerequisite

Please prepare a computer with Python installed. Please also install hashlib and bcrypt (e.g pip install hashlib bcrypt). Here is a sample code that might be useful in this activity.

In [ ]:
```python
import hashlib
import bcrypt

# SHA1
m = hashlib.sha1(b"Chulalongkorn").hexdigest()
print(m)
# MD5
m = hashlib.md5(b"Chulalongkorn").hexdigest()
print(m)
# BCRYPT
salt = bcrypt.gensalt()
m = bcrypt.hashpw(b"Chulalongkorn", salt).decode()
print(m)
```

```
ca8a68498ae67cd14c15f5ebf043633224005759
46fa3b56c660faff420190c18c98a56b
$2b$12$EMXmE8ujdSSQQydBjtHHXeUqtirj7zQ2UKl/zHfexN01AHZYIRM2q
```

Exercises

Q1.

Objective: Understand how attackers use pre-built word lists (dictionaries) to crack hashes of common passwords. Scenario: You have discovered a SHA-1 hash in a compromised system: d54cc1fe76f5186380a0939d2fc1723c44e8a5f7. You suspect the password is a simple, common word, possibly with some character substitutions. Task: Write a Python program that reads a list of words, applies common substitutions, hashes the result, and checks if it matches the target hash. Note that you might want to include substitution in your code (lowercase, uppercase, number for letter ['o' => 0 , 'l' => 1, 'i' => 1]).

In [ ]:
```
import hashlib
import itertools
from pathlib import Path
from typing import Dict, Iterator, List, Sequence, Set, Optional


TARGET_SHA1: str = "d54cc1fe76f5186380a0939d2fc1723c44e8a5f7"
WORDLIST_PATH: Path = Path("10k-most-common.txt")
MAX_VARIANTS_PER_WORD: int = 250_000  # guard against combinatorial blowu


# lowercase/uppercase plus: o->0, l->1, i->1
LEET_MAP: Dict[str, Sequence[str]] = {
    "o": ["0"],
    "l": ["1"],
    "i": ["1"],
}


def sha1_hex(text: str) -> str:
    """Return the SHA-1 hex digest of the given text (UTF-8)."""
    return hashlib.sha1(text.encode("utf-8")).hexdigest()


def per_char_options(ch: str) -> Sequence[str]:
    """
    For a single character, return possible variants:
    - lowercase and uppercase
    - leet substitutions if defined for the lowercase char
    Deduplicated per position to keep the product small.
    """
    base = ch.lower()
    opts: Set[str] = {base, base.upper()}
    if base in LEET_MAP:
        opts.update(LEET_MAP[base])
    return list(opts)


def generate_variants(
    word: str, max_variants: int = MAX_VARIANTS_PER_WORD
) -> Iterator[str]:
    """
    Yield mixed case + leetspeak variants for `word`.
    Uses a soft guard to avoid exponential blowups on long words.
    """
    choices: List[Sequence[str]] = [per_char_options(c) for c in word]
    est: int = 1
    for opts in choices:
        est *= len(opts)
        if est > max_variants:
            # Fall back to a small, high-probability set.
            yield word
            yield word.lower()
            yield word.upper()
            yield word.capitalize()
            return
    for combo in itertools.product(*choices):
        yield "".join(combo)


def crack_sha1_from_dictionary(
    target_hex: str,
    dictionary_path: Path,
    *,
    max_variants_per_word: int = MAX_VARIANTS_PER_WORD,
) -> Optional[str]:
    """
    Attempt to recover the plaintext for `target_hex` using words from `d
    Returns the matching plaintext if found, otherwise None.
    """
    if not dictionary_path.exists():
        raise FileNotFoundError(f"Wordlist not found: {dictionary_path.re

    with dictionary_path.open("r", encoding="utf-8", errors="ignore") as
        for raw in f:
            base = raw.strip()
            if not base:
                continue

            # Quick fast-path checks
            for quick in (base, base.lower(), base.upper(), base.capitali
                if sha1_hex(quick) == target_hex:
                    return quick

            # Full variant enumeration with leet + case
            for candidate in generate_variants(
                base, max_variants=max_variants_per_word
            ):
                if sha1_hex(candidate) == target_hex:
                    return candidate
    return None


match = crack_sha1_from_dictionary(TARGET_SHA1, WORDLIST_PATH)
print(f"Match found: {match}" if match else "No match found.")
```

```
Match found: ThaiLanD
```

Answer: ThaiLanD

Q2.

Objective: To understand why modern password hashing algorithms like bcrypt are more secure than older ones like MD5 and SHA-1. Task: Design and run an experiment to measure how many hashes each algorithm can compute in a fixed amount of time. The code must test atleast MD5 , SHA-1 , and bcrypt . (You may also try additional algorithms like SHA256, SHA512, scrypt, Argon2.) Hint: Use time function in python.

In [5]:
```
import time
import hashlib
```

```python
# Test message
PLAINTEXT = b"password123"


def benchmark(func, duration=2.0):
    """Run `func` repeatedly for ~duration seconds. Return iterations/sec
    start = time.perf_counter()
    end = start + duration
    count = 0
    while time.perf_counter() < end:
        func()
        count += 1
    elapsed = time.perf_counter() - start
    return count / elapsed


# Workers
def md5_worker():
    hashlib.md5(PLAINTEXT).digest()


def sha1_worker():
    hashlib.sha1(PLAINTEXT).digest()


salt = bcrypt.gensalt(rounds=12)  # cost factor 12


def bcrypt_worker():
    bcrypt.hashpw(PLAINTEXT, salt)


# Run benchmarks
results = {
    "MD5": benchmark(md5_worker),
    "SHA-1": benchmark(sha1_worker),
    "bcrypt (cost=12)": benchmark(bcrypt_worker),
}

for algo, hps in results.items():
    print(f"{algo}: {hps:,.0f} hashes/sec")
```

```
MD5: 2,198,399 hashes/sec
SHA-1: 2,472,814 hashes/sec
bcrypt (cost=12): 5 hashes/sec
```

Q3.

Objective: To apply the performance measurements to understand the importance of password length and algorithm choice. Task: Based on the measurements from Exercise 2, estimate how long it would take an attacker to brute-force a password of a given length You may assume that the password contains only upper-case, lower-case, numbers and symbols. What does it suggest about the length of a proper password. (ie. Use more than a year to brute force.)

In [ ]:
```python
# Hashing speeds from Q2 results
hash_rates = {
    "MD5": 2_198_399,
    "SHA-1": 2_472_814,
    "bcrypt(12)": 5,
}

CHARSET_SIZE = 94  # uppercase (26) + lowercase (26) + digits (10) + symb


def brute_force_time(length: int, rate: int) -> float:
    total = CHARSET_SIZE**length
    return total / rate


def format_time(seconds: float) -> str:
    minute, hour, day, year = 60, 3600, 86400, 31536000
    if seconds < minute:
        return f"{seconds:.2f} sec"
    elif seconds < hour:
        return f"{seconds / minute:.2f} min"
    elif seconds < day:
        return f"{seconds / hour:.2f} hrs"
    elif seconds < year:
        return f"{seconds / day:.2f} days"
    else:
        return f"{seconds / year:.2e} years"


# Build result table
for algo, rate in hash_rates.items():
    print(f"\n=== {algo} ({rate:,} hashes/sec) ===")
    for length in range(6, 13):  # lengths 6 to 12
        seconds = brute_force_time(length, rate)
        print(f"Length {length:2d}: {format_time(seconds)}")
```

```
=== MD5 (2,198,399 hashes/sec) ===
Length  6: 3.63 days
Length  7: 341.41 days
Length  8: 8.79e+01 years
Length  9: 8.26e+03 years
Length 10: 7.77e+05 years
Length 11: 7.30e+07 years
Length 12: 6.86e+09 years

=== SHA-1 (2,472,814 hashes/sec) ===
Length  6: 3.23 days
Length  7: 303.52 days
Length  8: 7.82e+01 years
Length  9: 7.35e+03 years
Length 10: 6.91e+05 years
Length 11: 6.49e+07 years
Length 12: 6.10e+09 years

=== bcrypt(12) (5 hashes/sec) ===
Length  6: 4.38e+03 years
Length  7: 4.11e+05 years
Length  8: 3.87e+07 years
Length  9: 3.63e+09 years
Length 10: 3.42e+11 years
Length 11: 3.21e+13 years
Length 12: 3.02e+15 years
```

Answer:

To exceed 1 year against your measured rates:

- Fast hashes need ≥ 8 characters.
- bcrypt(12) already makes even 6 characters take millennia, but still aim for 12–16+ for margin and usability across different threat models.

Q4.

If a given hash value is from a bcrypt algorithm, is it practical to do a brute-force attack?

Answer:

If a given hash value is produced using the bcrypt algorithm, it is generally not practical to perform a brute-force attack. Unlike older algorithms such as MD5 or SHA-1, which can be computed millions of times per second, bcrypt is intentionally slow and includes a configurable cost factor that makes each hash computation take significantly longer.

For example, in our experiment, bcrypt at cost=12 achieved only about 5 hashes per second compared to millions for MD5 and SHA-1. This means that even attempting all 8-character combinations would require millions of years, far beyond practical feasibility.

Therefore, when bcrypt is used properly with an adequate cost factor and unique salts, brute-forcing becomes computationally infeasible, shifting the focus of attackers toward weak or common passwords rather than exhaustive keyspace search.

Q5.

If a given hash value is from a bcrypt algorithm, is it practical to perform a rainbow table attack?

Answer:

If a given hash value is produced with the **bcrypt** algorithm, it is **not practical** to perform a rainbow table attack. Rainbow tables work by precomputing and storing hashes of many possible passwords so that, when a hash is stolen, an attacker can simply look it up instead of recomputing. This technique is only effective when the hash function is very fast (like MD5 or SHA-1) and deterministic. Bcrypt, however, is both **slow** and **salted by design** - every hash includes a random salt embedded in its string.

This means that even if the same password is used by two users, their bcrypt hashes will look different, and precomputed tables become useless because an attacker would need to regenerate a separate table for each unique salt. Combined with bcrypt's computational cost, building such tables is infeasible in terms of both time and storage.

Thus, rainbow table attacks are not practical against bcrypt.

Q6.

You have to store a password in a database. Please explain your design/strategy for securely storing it. (Hints: Proper hash function, Salting, Cost factor, Database Security)

Answer:

1. Use strong password hashing algorithms

- Argon2id (preferred) or bcrypt; not MD5/SHA-1.

2. Add unique salts and tune cost factors

- per-user salts plus parameters (e.g., bcrypt cost=12+) adjusted to hardware over time.

3. Optionally add a pepper

- an app-level secret stored separately from the database for extra protection.

4. Secure implementation and database

- vetted libraries, TLS in transit, encryption at rest, least-privilege DB roles, monitoring.

5. Maintain hygiene and upgrades

- strong password policy, MFA, rate limiting, rehash on login when parameters are outdated, and regular audits.