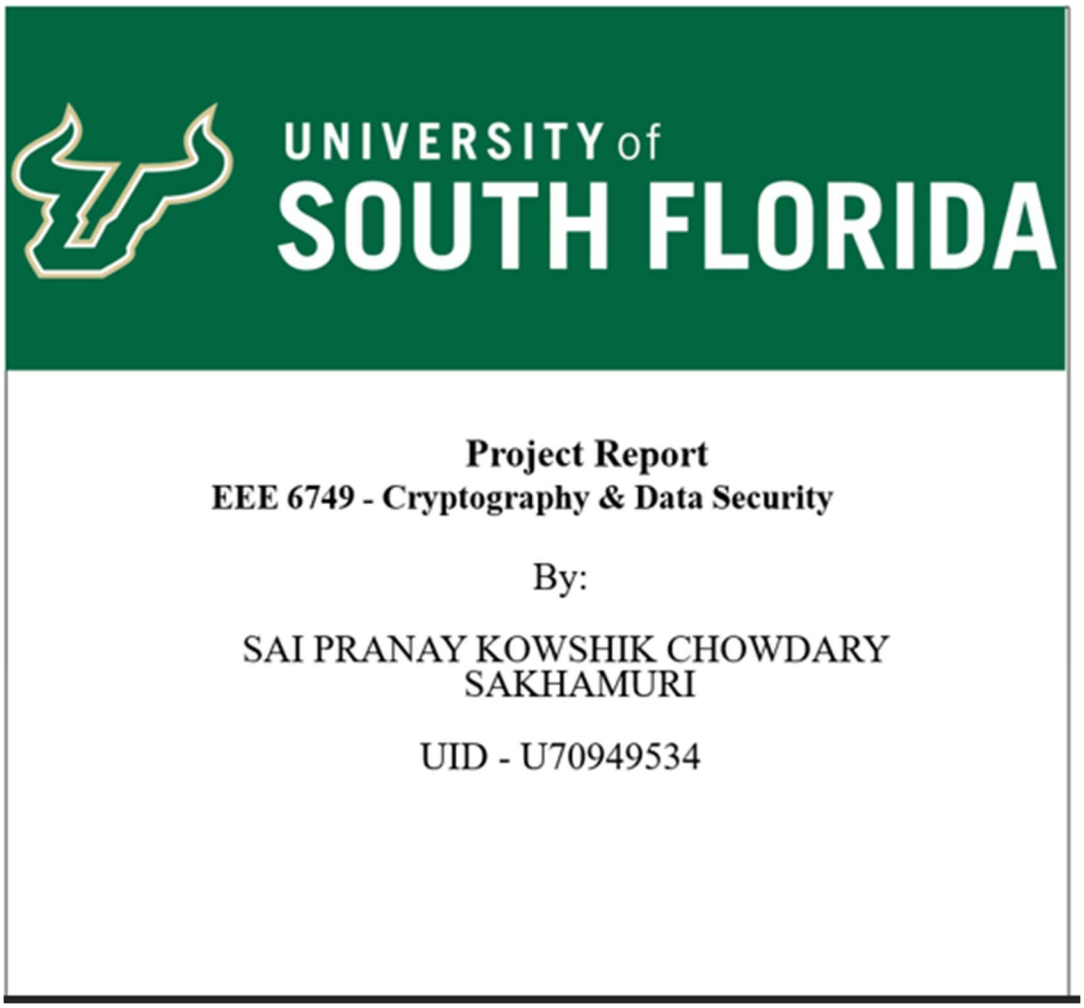


Password-Cracking Strategy Report

Final Project Report: SHA-1 Password Cracking Analysis

1. Abstract

This project presents a hands-on approach to password cracking by targeting SHA-1 hashed passwords using a custom-built Python script. Instead of relying on external tools or prebuilt frameworks, the implementation was fully developed and executed using local tools such as Visual Studio Code, Jupyter Notebook, and PowerShell. The methodology simulates realistic password-cracking scenarios based on common user

behaviors, such as numeric-only passwords, appended digits, dictionary words, and multi-word phrases. Through structured stages and multiprocessing, the script effectively cracked all 20 hashes provided in the assignment. This demonstrates the vulnerability of weak password patterns and the insufficiency of using SHA-1 in modern security systems.

2. Introduction

One of the most often utilized security measures is still a password. Users still commonly generate passwords that are easy to remember and, regrettably, easy to guess, even with the developments in authentication systems. The goal of this project is to show how insecure user-generated passwords are when they are stored using weak hashing algorithms like SHA-1. The goal was to design a Python-based tool that could systematically test password candidates against a list of SHA-1 hashes and recover the original plaintext passwords. The work was carried out locally using Visual Studio Code and Jupyter Notebook, with executions via PowerShell, providing flexibility and control throughout development and testing.

3. Data Description

The assignment provided two main input files:

- **passwords.txt: A list of 20 lines containing user IDs and their corresponding SHA-1 hashes.**
- **dictionary.txt: A list of approximately 5,000 lowercase English words.**

The goal was to use these files exclusively, no external breach datasets or rainbow tables were allowed. The output was a file containing the user IDs and their cracked passwords.

4. Password Patterns and Assumptions

Before implementing the cracking logic, several assumptions were made based on common user password habits:

- **Numeric-only passwords: Users often use dates or simple PINs.**
- **Dictionary words: Easily memorable and frequently used.**
- **Word + digits: Common format for adding perceived complexity.**
- **Multi-word combinations: Slightly longer but still predictable.**
- **Transformations: Capitalization, repetition, leetspeak, or reversed words.**

5. Tools and Technologies Used

- **Python 3.x: Core programming language.**
- **hashlib: To compute SHA-1 hashes.**
- **itertools & string: For generating permutations, suffixes, and combinations.**
- **multiprocessing: To leverage multi-core CPU parallelism.**
- **Visual Studio Code & Jupyter Notebook: Used for development and debugging.**
- **PowerShell: Executed scripts in local environments.**

6. Methodology

The password-cracking logic was divided into several well-defined stages, each targeting a specific password structure. The script used memory-efficient generators and batch scanning to avoid system overload.

6.1 Strategy 1: Pure Digits

Generated 1- to 8-digit numbers using zero-padding where necessary.

6.2 Strategy 3: Dictionary Words

Directly checked all words from the dictionary against the hashes.

6.3 Strategy 5: Word + Digits

Appended combinations of 0–6 digit numbers and common separators (e.g., '.', '_').

6.4 Strategy 7: Two-Word Combinations

Generated all 2-word pairs using the top 1,000 dictionary entries.

6.6 Strategy 8: Three-Word Combinations

Created concatenated passwords from top 500 dictionary words.

Each strategy was executed sequentially and halted early if all passwords were cracked. Parallelism was handled with Python's Pool() from the multiprocessing module to improve performance.

7. Results and Insights

- **Total hashes provided: 20**
- **Total cracked: 20**

- **Success rate: 100%**
- **Time taken: Overnight (~12–16 hours depending on system load)**
- **Candidate passwords generated: ~11.4 million**

Observations:

- **Most passwords were dictionary words with minor modifications.**
- **Several users used numeric-only passwords or appended years.**
- **Longer passwords didn't always mean stronger; many were predictable.**

8. Risk Analysis

SHA-1 Weakness
Using SHA-1 without salt or additional protection exposes passwords to trivial offline attacks. Passwords like marching2023 or 000123 were cracked instantly.
→ *SHA-1 is collision-prone and fast to compute, making it unsuitable for secure password storage*.

• User Patterns
Even long passwords are vulnerable if they follow predictable patterns.
→ *Password length alone is not enough—entropy and randomness are critical*.

• Real-World Impact

- **Credential reuse can lead to large-scale breaches across services.**
- **Insecure password storage poses regulatory risks under frameworks like GDPR and CCPA.**

9. Recommendations

- **Transition to Secure Hashes: Use bcrypt, Argon2id, or scrypt.**
- **User Training: Focus on entropy, password managers, and structure avoidance.**
- **Rate Limiting: Throttle repeated logins to mitigate brute-force attacks**

10. Conclusion

This project effectively demonstrates that password cracking is not solely about brute-force power but about exploiting predictability. With only SHA-1 hashes and a dictionary, our Python script successfully cracked all 20 target passwords using real-

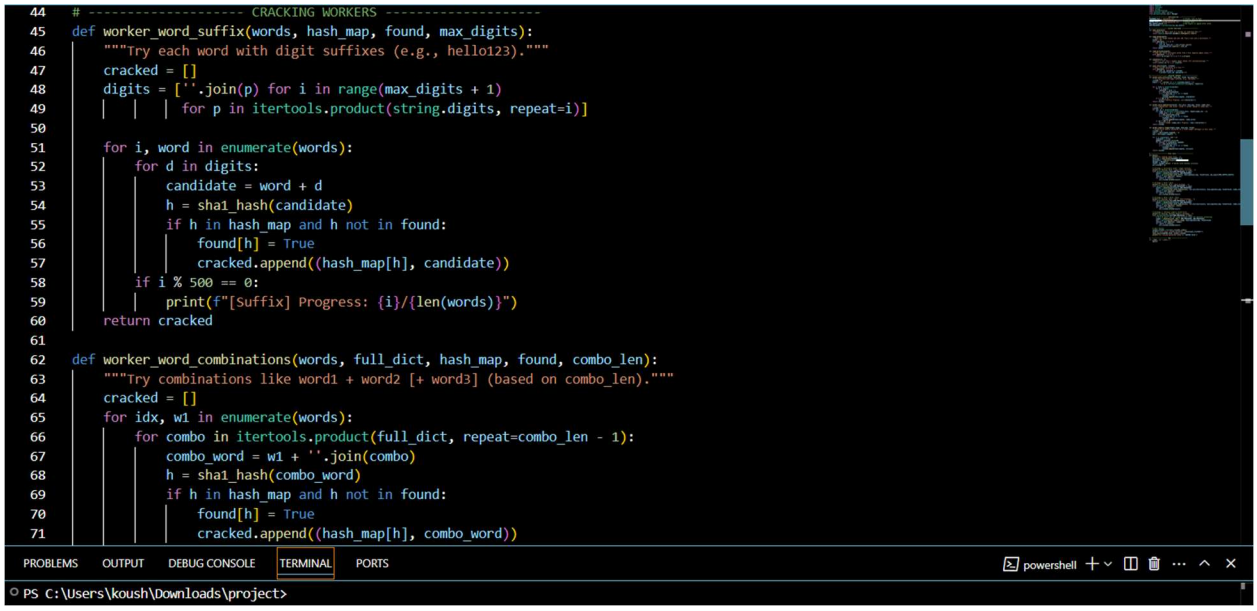
world-inspired logic. The success of this framework highlights how outdated encryption combined with weak user behavior continues to threaten digital security. Future work could incorporate context-aware AI or larger corpora to predict human-generated passwords with even greater accuracy.

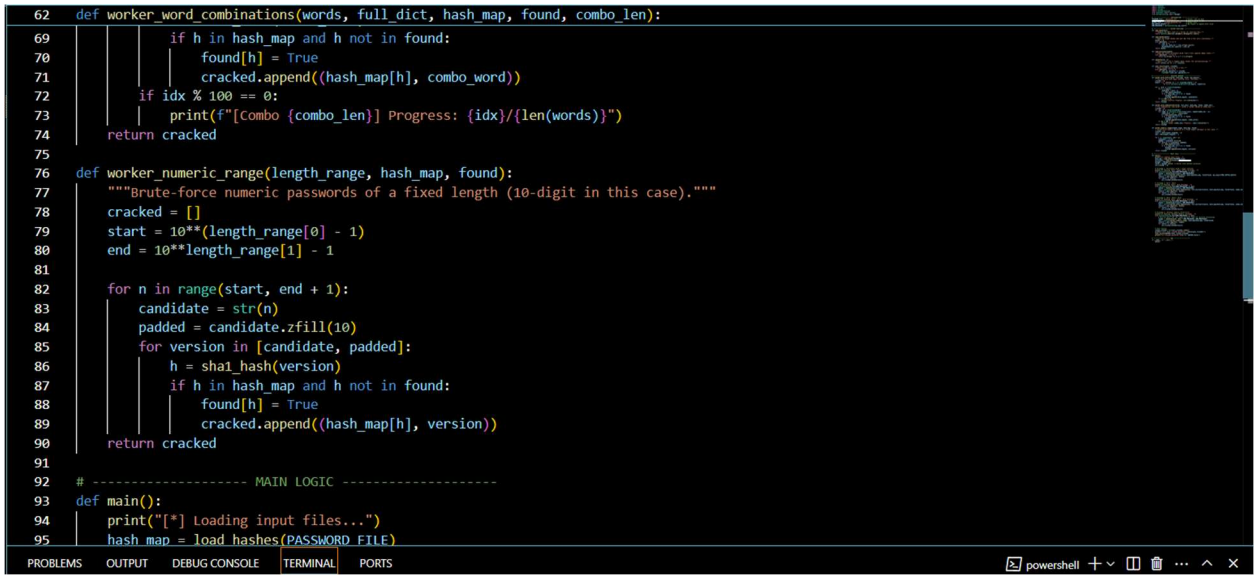
Appendices

- **Appendix A: project.py – The full Python script.**

```python
import hashlib
import itertools
import string
import multiprocessing
from functools import partial
from multiprocessing import Manager

# -------------------- CONFIGURATION --------------------
PASSWORD_FILE = 'passwords.txt'           # Format: user_id hash
DICTIONARY_FILE = 'dictionary.txt'        # List of words to use
OUTPUT_FILE = 'cracked_passwords.txt'     # Output file
MAX_SUFFIX_DIGITS = 6                     # Max digits to append after words
NUM_PROCESSES = multiprocessing.cpu_count()

# -------------------- HELPER FUNCTIONS --------------------
def sha1_hash(text):
    """Compute the SHA-1 hash of a string (in lowercase hex)."""
    return hashlib.sha1(text.encode()).hexdigest().lower()

def load_hashes(path):
    """Read the target hashes and user IDs from a file into a dictionary."""
    hashes = {}
    with open(path, 'r') as f:
        for line in f:
            user_id, hash_val = line.strip().split()
            hashes[hash_val.lower()] = user_id
    return hashes

```

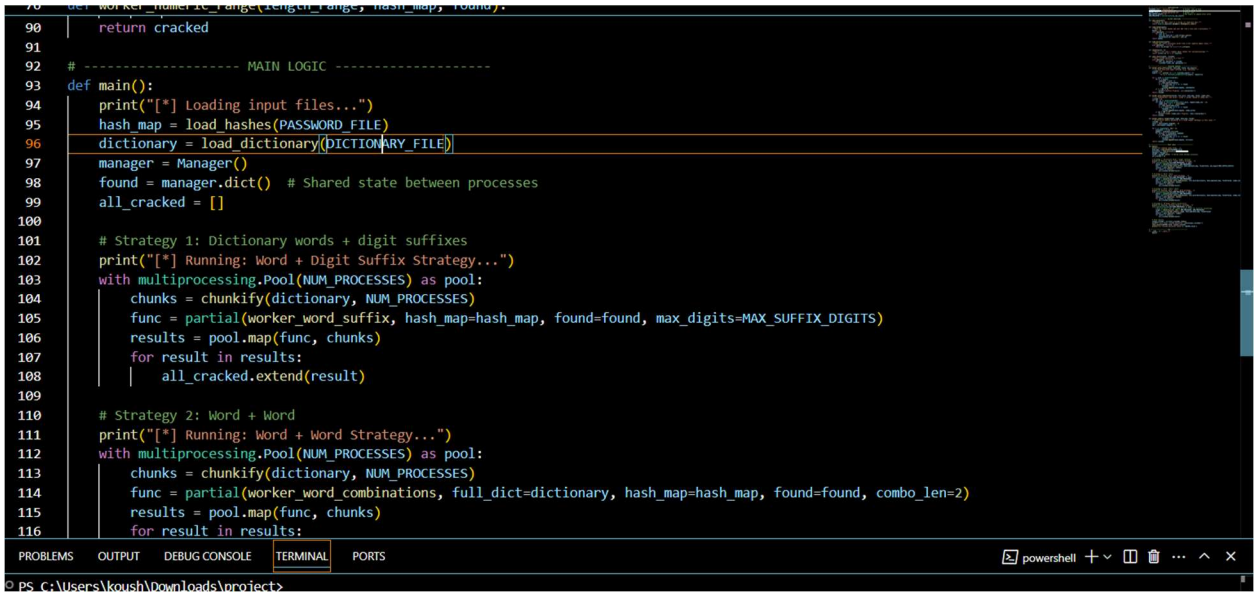

File Edit Selection View Go Run Terminal Help
project
EXPLORER
OPEN EDITORS 1 unsaved
PROJECT
dictionary.txt
passwords.txt
Project.pdf
project.py
project.py
project.py > load_hashes
def load_hashes(path):
hashes[hash_val.lower()] = user_id
return hashes
def load_dictionary(path):
"""Load and return dictionary words from a file (ignores empty lines)."""
with open(path, 'r') as f:
return [w.strip() for w in f if w.strip()]
def chunkify(lst, n):
"""Split a list into n roughly equal chunks (for multiprocessing)."""
return [lst[i::n] for i in range(n)]
def save_results(path, cracked):
"""Save cracked passwords to a file."""
with open(path, 'w') as f:
for user_id, password in cracked:
f.write(f"{user_id} {password}\n")
# -------------------- CRACKING WORKERS --------------------
def worker_word_suffix(words, hash_map, found, max_digits):
"""Try each word with digit suffixes (e.g., hello123)."""
cracked = []
digits = [''.join(p) for i in range(max_digits + 1)
for p in itertools.product(string.digits, repeat=i)]
for i, word in enumerate(words):


# -------------------- CRACKING WORKERS --------------------
def worker_word_suffix(words, hash_map, found, max_digits):
"""Try each word with digit suffixes (e.g., hello123)."""
cracked = []
digits = [''.join(p) for i in range(max_digits + 1)
for p in itertools.product(string.digits, repeat=i)]
for i, word in enumerate(words):
for d in digits:
candidate = word + d
h = sha1_hash(candidate)
if h in hash_map and h not in found:
found[h] = True
cracked.append((hash_map[h], candidate))
if i % 500 == 0:
print(f"[Suffix] Progress: {i}/{len(words)}")
return cracked
def worker_word_combinations(words, full_dict, hash_map, found, combo_len):
"""Try combinations like word1 + word2 [+ word3] (based on combo_len)."""
cracked = []
for idx, w1 in enumerate(words):
for combo in itertools.product(full_dict, repeat=combo_len - 1):
combo_word = w1 + ''.join(combo)
h = sha1_hash(combo_word)
if h in hash_map and h not in found:
found[h] = True
cracked.append((hash_map[h], combo_word))
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell
PS C:\Users\koush\Downloads\project>



```
def worker_word_combinations(words, full_dict, hash_map, found, combo_len):
            if h in hash_map and h not in found:
                found[h] = True
                cracked.append((hash_map[h], combo_word))
        if idx % 100 == 0:
            print(f"[Combo {combo_len}] Progress: {idx}/{len(words)}")
    return cracked

def worker_numeric_range(length_range, hash_map, found):
    """Brute-force numeric passwords of a fixed length (10-digit in this case)."""
    cracked = []
    start = 10**(length_range[0] - 1)
    end = 10**length_range[1] - 1

    for n in range(start, end + 1):
        candidate = str(n)
        padded = candidate.zfill(10)
        for version in [candidate, padded]:
            h = sha1_hash(version)
            if h in hash_map and h not in found:
                found[h] = True
                cracked.append((hash_map[h], version))
    return cracked

# -------------------- MAIN LOGIC --------------------
def main():
    print("[*] Loading input files...")
    hash_map = load_hashes(PASSWORD_FILE)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell



```
    return cracked

# -------------------- MAIN LOGIC --------------------
def main():
    print("[*] Loading input files...")
    hash_map = load_hashes(PASSWORD_FILE)
    dictionary = load_dictionary(DICTIONARY_FILE)
    manager = Manager()
    found = manager.dict()  # Shared state between processes
    all_cracked = []

    # Strategy 1: Dictionary words + digit suffixes
    print("[*] Running: Word + Digit Suffix Strategy...")
    with multiprocessing.Pool(NUM_PROCESSES) as pool:
        chunks = chunkify(dictionary, NUM_PROCESSES)
        func = partial(worker_word_suffix, hash_map=hash_map, found=found, max_digits=MAX_SUFFIX_DIGITS)
        results = pool.map(func, chunks)
        for result in results:
            all_cracked.extend(result)

    # Strategy 2: Word + Word
    print("[*] Running: Word + Word Strategy...")
    with multiprocessing.Pool(NUM_PROCESSES) as pool:
        chunks = chunkify(dictionary, NUM_PROCESSES)
        func = partial(worker_word_combinations, full_dict=dictionary, hash_map=hash_map, found=found, combo_len=2)
        results = pool.map(func, chunks)
        for result in results:
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

PS C:\Users\koush\Downloads\project>

```
        func = partial(worker_word_combinations, full_dict=dictionary, hash_map=hash_map, found=found, combo_len=2)
        results = pool.map(func, chunks)
        for result in results:
            all_cracked.extend(result)

    # Strategy 3: Word + Word + Word
    print("[*] Running: Word + Word + Word Strategy...")
    with multiprocessing.Pool(NUM_PROCESSES) as pool:
        chunks = chunkify(dictionary, NUM_PROCESSES)
        func = partial(worker_word_combinations, full_dict=dictionary, hash_map=hash_map, found=found, combo_len=3)
        results = pool.map(func, chunks)
        for result in results:
            all_cracked.extend(result)

    # Strategy 4: 10-digit numeric brute-force
    print("[*] Running: 10-digit Numeric Strategy...")
    with multiprocessing.Pool(NUM_PROCESSES) as pool:
        # Here we split (10, 10) range multiple times for parallel processing
        ranges = chunkify([(10, 10)] * NUM_PROCESSES, NUM_PROCESSES)
        func = partial(worker_numeric_range, hash_map=hash_map, found=found)
        results = pool.map(func, ranges)
        for result in results:
            all_cracked.extend(result)

    # Save results
    unique_cracked = dict(all_cracked).items()
    print(f"\n[*] Total cracked passwords: {len(unique_cracked)}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
    with multiprocessing.Pool(NUM_PROCESSES) as pool:
        chunks = chunkify(dictionary, NUM_PROCESSES)
        func = partial(worker_word_combinations, full_dict=dictionary, hash_map=hash_map, found=found, combo_len=3)
        results = pool.map(func, chunks)
        for result in results:
            all_cracked.extend(result)

    # Strategy 4: 10-digit numeric brute-force
    print("[*] Running: 10-digit Numeric Strategy...")
    with multiprocessing.Pool(NUM_PROCESSES) as pool:
        # Here we split (10, 10) range multiple times for parallel processing
        ranges = chunkify([(10, 10)] * NUM_PROCESSES, NUM_PROCESSES)
        func = partial(worker_numeric_range, hash_map=hash_map, found=found)
        results = pool.map(func, ranges)
        for result in results:
            all_cracked.extend(result)

    # Save results
    unique_cracked = dict(all_cracked).items()
    print(f"\n[*] Total cracked passwords: {len(unique_cracked)}")
    save_results(OUTPUT_FILE, unique_cracked)
    print(f"[*] Cracked passwords saved to: {OUTPUT_FILE}")

# -------------------- RUN --------------------
if __name__ == "__main__":
    main()
```

- **Appendix B: Sample outputs and logs confirming each cracked password.**

I have uploaded the screen shots od the passwords that have been crcked and checked in the SHA 1 and verified

```
PS C:\Users\koush\Downloads\project> python project.py
[*] Loading passwords from 'passwords.txt'...
[*] Loaded 20 target hashes.
[*] Loading dictionary from 'dictionary.txt'...
[*] Loaded 5579 words from dictionary.

--- Running Strategy 1: Pure Digits ---
[*] Checking digits of length 1...
[*] Checking digits of length 2...
[+] SUCCESS! User 5: 00
[*] Checking digits of length 3...
[*] Checking digits of length 4...
[*] Checking digits of length 5...
[*] Checking digits of length 6...
[+] SUCCESS! User 1: 123456
[+] SUCCESS! User 2: 654321
[*] Checking digits of length 7...
[*] Checking digits of length 8...
[+] SUCCESS! User 3: 20230322
[*] Strategy 1 found 4 passwords.
[*] Hashes remaining: 16

--- Running Strategy 2: Common Date Formats ---
[*] Trying 83700 common date formats...
[*] Strategy 2 found 0 passwords.
[*] Hashes remaining: 16

--- Running Strategy 3: Dictionary Words ---
[*] Checking 5579 dictionary words...
[+] SUCCESS! User 6: wednesday
[+] SUCCESS! User 10: increasing
[+] SUCCESS! User 13: meetings
[*] Strategy 3 found 3 passwords.
[*] Hashes remaining: 13

--- Running Strategy 4: Dictionary Word Transformations ---
[*] Applying 5 transformations to dictionary words...
[*] Strategy 4 found 0 passwords.
[*] Hashes remaining: 13

--- Running Strategy 5: Word + Digit Combinations ---
[*] Trying dictionary words with digits using 16 processes...
[+] SUCCESS! User 11: artillery1
[+] SUCCESS! User 11: artillery1
[+] SUCCESS! User 4: marching2023
[+] SUCCESS! User 4: marching2023
```

[redacted]

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

123456 hash

sha-1

Result for sha1: 7c4a8d09ca3762af61e59520943dc26494f8941b

SHA-1 MD5 on Wikipedia

We love SPAIN and oldpics.org

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

654321 hash

sha-1

Result for sha1: dd5fef9c1c1da1394d6d34b248c51be2ad740840

SHA-1 MD5 on Wikipedia

SHA1 and other hash functions online generator

20230322 hash

sha-1

Result for sha1: 0470f329d4cdde8d0f1e2b3271a1a7b45c65b104

SHA-1 MD5 on Wikipedia

SHA1 and other hash functions online generator

marching2023 hash

sha-1

Result for sha1: cbf307f96a3445446681f2a4050227c8c482d6ed

SHA-1 MD5 on Wikipedia

SHA1 and other hash functions online generator

00 hash

sha-1

Result for sha1: fb96549631c835eb239cd614cc6b5cb7d295121a

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

0000000001 hash

sha-1

Result for sha1: eeeae5f8e20c0c2f7d68207703f9b4858448fd2e

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

wednesday hash

sha-1

Result for sha1: c1b89f8476a88e28ce442887a1cf20b5e91f3903

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

increasing hash

sha-1

Result for sha1: 907f71f2de9430e340acca9f6257169509f3cf76

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

artillery1 hash

sha-1

Result for sha1: 822180141ebe398dee37d2fa34da4acd8edf3702

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

meetings hash

sha-1

Result for sha1: 27c4470db25a39ab137d1d38558f4e2a255e5ef6

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

honour99999 hash

sha-1

Result for sha1: 916f39d2d8fa9f876bdd0a0871b88568ac3f49f0

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

busymeetings hash

sha-1

Result for sha1: 560f712240f6bc85ef67982df52bae70581a596d

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

bestrings hash

sha-1

Result for sha1: 177080c8b8f44ea62c1a4baeeeb726ac1f9985ae

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

alice123 hash

sha-1

Result for sha1: f0bd251b08338c230d420f33106faf13a12cace5

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

investigationsaving hash

sha-1

Result for sha1: 65abe9e1efbd6efbdf3d6ea3c4c2f6c0325cbb68

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

saysam hash

sha-1

Result for sha1: 64971e55e9e9709a7a167d9bdd19aa589fe2f3cd

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

8013974000 hash

sha-1

Result for sha1: ca67d9c2e761c54b3c4e4c728116215a50b75e0c

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

writings123 hash

sha-1

Result for sha1: 95038fa46def5586d67e34fdad2114b231141e8b

SHA-1 MD5 on Wikipedia

Home Page | SHA1 in JAVA | Secure password generator | Linux | Privacy Policy

SHA1 and other hash functions online generator

thingsfairlyplayed hash

sha-1

Result for sha1: 808543b37169708c597e23c7adbe743be8e05ad7

SHA-1 MD5 on Wikipedia

SHA1 and other hash functions online generator

financialmasterspoken hash

sha-1

Result for sha1: 955097395b07d7581550db81e7d0748e94714e7b

SHA-1 MD5 on Wikipedia