Enhanced Password Hashing Algorithm: Addressing Vulnerabilities of Legacy Methods

Abstract— **In today's digital landscape, ensuring the security of user passwords is crucial to protecting sensitive information from malicious actors. Traditional hashing algorithms like MD5 and SHA-1, once popular, have proven vulnerable to advanced attacks, necessitating the development of more robust methods. This document presents a new approach to password hashing, combining SHA-256 with Argon2id, an advanced memory-hard function. By leveraging iterative hashing, salting, and key stretching techniques, our algorithm enhances security while mitigating the risks associated with older hashing methods.**

1. Introduction

Password hashing algorithms play a pivotal role in safeguarding sensitive data by transforming passwords into irreversible cryptographic hashes. Among these, the combined approach of SHA-256 and Argon2id stands out for its robust security mechanisms and efficient performance. SHA-256 initiates the process by generating a secure hash with a random salt, fortifying against rainbow table attacks through its one-way function. This initial hash is further fortified by Argon2id, an iterative algorithm that adjusts time and memory costs, thwarting brute-force and dictionary attacks effectively. By leveraging parallelism and memory-hard functions, Argon2id enhances resistance to GPU and ASIC-based attacks, ensuring high computational overhead for attackers without compromising user experience. This combined method not only enhances security by minimizing vulnerabilities inherent in single algorithm approaches but also ensures optimized performance suitable for modern computing environments. Its effectiveness lies in striking a balance between security and usability, making it a preferred choice for applications demanding stringent password protection.

2. Background

SHA-256

SHA-256 (Secure Hash Algorithm 256-bit) is a cornerstone cryptographic hash function designed by the National Security Agency (NSA). It produces a 256-bit (32-byte) hash value, making it highly resistant to collision attacks where different inputs produce the same hash. SHA-256 is widely used in digital signatures, SSL/TLS certificates, and blockchain technologies due to its efficiency and security properties.

Argon2id

Argon2id is a state-of-the-art password hashing algorithm selected as the winner of the Password Hashing Competition in 2015. It combines Argon2i and Argon2d variants to provide a hybrid solution that is resistant to side-channel attacks, memory-efficient, and resistant to GPU and ASIC optimizations. Argon2id operates by utilizing time-cost, memory-cost, and parallelism parameters to adjust computational hardness, making it ideal for securely hashing passwords in modern applications.

The combination of SHA-256 and Argon2id in our proposed algorithm leverages the strengths of both algorithms to create a robust and secure password hashing solution. This section will provide a foundation for understanding their roles in enhancing password security.

3. Implementation Details

3.1 Global Constants and Requirements

Global constants enforce essential password complexity rules: MIN_PASSWORD_LENGTH specifies the minimum password length to thwart brute-force attacks. MIN_UPPERCASE_CHARS ensures at least one uppercase letter is used, enhancing complexity. MIN_DIGITS mandates at least one digit, further diversifying potential passwords and bolstering security measures.

```
# Global constants for password complexity
MIN_PASSWORD_LENGTH = 8
MIN_UPPERCASE_CHARS = 1
MIN_DIGITS = 1
```

3.2 Salt Generation

The generate_salt function plays a critical role in enhancing the security of hashed passwords by introducing a unique and random element to each password's hash process. Salting involves adding a randomly generated string of bytes (salt) to the password before hashing, ensuring that even if two users have the same password, their hashed values will differ due to the unique salts. This prevents attackers from using precomputed hash tables (rainbow tables) to easily reverse hashed passwords back to their original plaintext forms. os.urandom(length) is utilized to generate a cryptographically secure salt of the specified length in bytes. This method utilizes operating system-specific randomness sources to ensure that the generated salts are unpredictable and suitable for cryptographic use. By incorporating salts into the password hashing process, applications can significantly improve their resistance against various types of password cracking techniques, thereby bolstering overall security and protecting user credentials from unauthorized access.

```
# Function to generate a cryptographically secure random salt
def generate_salt(length=16):
    return os.urandom(length)
```

3.3 Password Hashing with SHA-256

The hash_with_sha256 function securely converts plaintext passwords into fixed-size hash values using the SHA-256 (Secure Hash Algorithm 256-bit). It concatenates the password with a unique salt before hashing, ensuring each password's hash is unique even if two users have the same password. SHA-256 is chosen for its cryptographic strength, resistance to collision attacks, and widespread adoption in secure password hashing. The resulting hash is a 256-bit (32-byte) binary digest, serving as a digital fingerprint of the original password. This irreversible transformation ensures that the original password cannot feasibly be deduced from its hash, providing a crucial layer of security for storing and managing user credentials. By integrating SHA-256 hashing with salted passwords, applications can mitigate vulnerabilities associated with storing passwords in plaintext or using weak hashing methods, thereby enhancing overall security posture against unauthorized access and data breaches.

```
# Function to hash the password with SHA-256 and a salt
def hash_with_sha256(password, salt):
    salted_password = password.encode() + salt
    sha256_hash = hashlib.sha256(salted_password).digest()
    return sha256_hash
```

\

3.4 Iterative Hashing with Argon2id

The `hash_with_argon2id_iterative` function enhances password security by iteratively applying the Argon2id hashing algorithm to the SHA-256 hashed password. Argon2id is chosen for its resistance against various attack vectors, including GPU and side-channel attacks. By iterating the hashing process (`iterations` times), the function significantly increases the computational effort required to brute-force attack hashed passwords, thereby enhancing resistance against password cracking attempts. Argon2id combines the benefits of Argon2i (optimized for resistance against GPU attacks) and Argon2d (optimized for resistance against side-channel attacks), making it suitable for securely hashing passwords in diverse application scenarios. The resulting `argon2id_hash` represents a highly secure and irreversible transformation of the original password, ensuring that even in the event of a data breach, plaintext passwords remain protected. Integrating iterative Argon2id hashing with SHA-256 enhances the overall security posture of applications by mitigating vulnerabilities associated with traditional hashing algorithms and strengthening defenses against evolving cyber threats.

```python
# Function to hash the SHA-256 result with Argon2id parameters iteratively
def hash_with_argon2id_iterative(sha256_hash, iterations=1, time_cost=4, memory_cost=16384, parallelism=2):
    ph = PasswordHasher(time_cost=time_cost, memory_cost=memory_cost, parallelism=parallelism)
    argon2id_hash = sha256_hash.hex()
    for _ in range(iterations):
        argon2id_hash = ph.hash(argon2id_hash)
    return argon2id_hash
```

3.5 Password Complexity Checking

In dictionary attacks, an attacker tries all the possible passwords in an exhaustive list called a dictionary. The attacker hashes each password from the dictionary and performs a binary search on the compromised hashed passwords. This method can be made much quicker by pre-computing the hash values of these possible passwords and storing The check_password_complexity function ensures that passwords meet predefined complexity requirements before they are hashed and stored securely. By validating password complexity, applications can enforce strong password policies that mitigate the risk of brute-force attacks and enhance overall security. The function checks three primary criteria: minimum length (MIN_PASSWORD_LENGTH), required uppercase characters (MIN_UPPERCASE_CHARS), and mandatory digits (MIN_DIGITS). If a password fails to meet any of these criteria, corresponding error messages are appended to the errors list. This feedback mechanism allows users to understand and correct password deficiencies proactively, ensuring that only sufficiently complex passwords are accepted and processed. By integrating password complexity checking into the password management workflow, applications promote best practices in password security and reduce the likelihood of successful password guessing attacks. As a foundational component of the password hashing algorithm, password complexity checking reinforces overall security measures and safeguards user credentials against unauthorized access and exploitation

```python
# Function to check if password meets complexity requirements
def check_password_complexity(password):
    errors = []

    # Check minimum length
    if len(password) < MIN_PASSWORD_LENGTH:
        errors.append(f"Password must be at least {MIN_PASSWORD_LENGTH} characters long.")

    # Check for uppercase letters
    if sum(1 for c in password if c.isupper()) < MIN_UPPERCASE_CHARS:
        errors.append(f"Password must contain at least {MIN_UPPERCASE_CHARS} uppercase letter(s).")

    # Check for digits
    if sum(1 for c in password if c.isdigit()) < MIN_DIGITS:
        errors.append(f"Password must contain at least {MIN_DIGITS} digit(s).")

    return errors
```

3.6. Combined Password Hashing

Combining SHA-256 with Argon2id provides a layered approach to password hashing, leveraging the strengths of both algorithms to enhance security. Our `combined_hash_password` function integrates these two hashing methods to produce a final hashed password. It measures execution time and space complexity (`sha256_space_complexity`, `argon2id_space_complexity`) to evaluate computational costs and resource requirements.

During execution, the function first computes the SHA-256 hash of the password concatenated with a salt. This initial hash serves as input for the iterative Argon2id hashing process, which further fortifies the password's cryptographic strength. By combining these methods, we achieve a robust and computationally intensive hashing process that effectively defends against various password cracking techniques.

Monitoring execution time and space complexity provides insights into the computational overhead introduced by the hashing process. These metrics are crucial for optimizing performance while maintaining stringent security standards in password handling.

```python
# Combined function to hash the password using both SHA-256 and Argon2id iteratively
def combined_hash_password(password, salt, iterations=1, time_cost=4, memory_cost=16384, parallelism=2):
    start_time = time.time()
    sha256_hash = hash_with_sha256(password, salt)
    argon2id_hash = hash_with_argon2id_iterative(sha256_hash, iterations, time_cost, memory_cost, parallelism)
    execution_time = time.time() - start_time

    # Calculating space complexity
    # SHA-256 space complexity is O(1)
    sha256_space_complexity = 32  # Size of SHA-256 output in bytes
    # Argon2id space complexity is approximately O(p * m), where p is parallelism and m is memory cost
    argon2id_space_complexity = parallelism * memory_cost

    return argon2id_hash, execution_time, sha256_space_complexity, argon2id_space_complexity
```

3.7. Password Verification

The combined_verify_password function validates the correctness of a password by comparing the input password hash (stored_hash) with the freshly computed SHA-256 hash of the provided password (password and salt). It utilizes the PasswordHasher from the Argon2 library to verify the stored hash against the computed hash and simultaneously checks if the password meets the specified complexity requirements using check_password_complexity. If both conditions are satisfied, the function returns True, indicating successful password verification; otherwise, it returns False. This approach ensures that password verification is conducted securely and efficiently, leveraging both cryptographic hashing and complexity checking to protect user credentials from unauthorized access and ensure robust security in password-based authentication systems.

```python
# Function to verify the password using the combined hashing approach
def combined_verify_password(stored_hash, password, salt, iterations=1, time_cost=4, memory_cost=16384, parallelism=2):
    start_time = time.time()
    sha256_hash = hash_with_sha256(password, salt)
    ph = PasswordHasher(time_cost=time_cost, memory_cost=memory_cost, parallelism=parallelism)
    try:
        is_valid = ph.verify(stored_hash, sha256_hash.hex()) and check_password_complexity(password) == []
    except exceptions.VerifyMismatchError:
        is_valid = False
    verification_time = time.time() - start_time

    # Calculating space complexity
    # SHA-256 space complexity is O(1)
    sha256_space_complexity = 32  # Size of SHA-256 output in bytes
    # Argon2id space complexity is approximately O(p * m), where p is parallelism and m is memory cost
    argon2id_space_complexity = parallelism * memory_cost

    return is_valid, verification_time, sha256_space_complexity, argon2id_space_complexity
```

3.8. Key Rotation

Key rotation is a proactive security measure to mitigate risks associated with long-term password storage. Our implementation includes is_key_rotation_needed and rotate_keys functions to assess and perform key rotation when necessary.

is_key_rotation_needed determines if key rotation is due based on the elapsed time since the last rotation. Regular rotation reduces exposure to compromised credentials by invalidating older hashes and salts.

rotate_keys updates password hashes and salts during rotation, enhancing security without disrupting user access. It ensures that passwords remain resistant to attacks over time, aligning with best practices for password management and security hygiene.

Implementing key rotation safeguards against evolving threats and reinforces the integrity of stored passwords throughout their lifecycle.

```python
# Function to update salts and rehash passwords if key rotation is needed
def rotate_keys(username, stored_hash, salt, password):
    new_salt = generate_salt()
    new_combined_hashed_password, _, _, _ = combined_hash_password(password, new_salt)
    # Update storage with new hash and salt (in real-world, update the database)
    return new_combined_hashed_password, new_salt
```

3.9. Demo and Usage

The main script demonstrates password hashing, verification, and key rotation functionalities in a simulated environment. It prompts users for credentials, validates password complexity, hashes passwords using combined methods, verifies correctness, and initiates key rotation if needed.

By simulating user interactions and system responses, the demo showcases practical implementation of password security measures. It highlights the integration of hashing algorithms, complexity checks, and key management practices to ensure robust password protection and authentication integrity.

```python
# Prompting for user input and demonstrating password hashing and verification
if __name__ == "__main__":
    last_rotation_date = datetime.now() - timedelta(days=KEY_ROTATION_INTERVAL_DAYS + 1)  # Simulating last rotation date

    while True:
        # Prompt user for username and password
        username = input("Enter your username: ")
        password = input("Enter your password: ")

        # Validate password complexity
        complexity_errors = check_password_complexity(password)
        if complexity_errors:
            print("\nPassword complexity errors:")
            for error in complexity_errors:
                print("-", error)
            continue

        # Generate a random salt
        salt = generate_salt()

        # Hash the password using the combined method and measure execution time
        combined_hashed_password, hashing_time, sha256_space, argon2id_space = combined_hash_password(password, salt)

        # Store the hash and salt (in real-world, store securely like in a database)
        print("\nStored Combined Hash:", combined_hashed_password)
        print("Stored Salt:", salt.hex())
        print("Hashing Time:", hashing_time, "seconds")
        print("SHA-256 Space Complexity:", sha256_space, "bytes")
        print("Argon2id Space Complexity:", argon2id_space, "bytes")

        # Simulate successful login
        print("\nSimulating successful login...")

        # Verify the password using the combined method and measure execution time
        entered_password = input("\nEnter the password again to verify: ")
        verification_result, verification_time, _, _ = combined_verify_password(combined_hashed_password, entered_password, salt)
```

4. Algorithm Evaluation and Operational Considerations

4.1 Robust Hashing Methodology:

The system employs SHA-256 and Argon2id as its core hashing algorithms, each chosen for its specific strengths in password security. SHA-256, a widely recognized cryptographic hash function, provides a foundational level of security by converting passwords and salts into fixed-size 256-bit hash values. This method ensures uniformity and consistency in password storage, preventing the possibility of reversing hashed values back to their original passwords due to its one-way hashing nature.

Argon2id, on the other hand, enhances the security of SHA-256 by iteratively hashing its output. This iterative process makes Argon2id resistant to time-memory trade-off attacks and side-channel vulnerabilities, which are common in modern password cracking attempts. Additionally, Argon2id allows for dynamic parameter adjustments such as iterations, time cost, memory cost, and parallelism. These adjustments are crucial for optimizing performance based on the user's system capabilities, ensuring efficient password hashing without compromising security. Tuning these parameters to match the user's hardware and application requirements is essential for maximizing the efficiency of the algorithm.

4.2 Customizable Security Parameters:

The system's use of customizable security parameters in Argon2id, including iterations, time cost, memory cost, and parallelism, provides a tailored approach to password security. Higher values for iterations and memory cost significantly increase the computational effort required for hashing, thereby strengthening resistance against brute-force attacks. The ability to adjust these parameters dynamically allows the system to adapt to varying levels of security needs and operational environments. For instance, applications running on high-performance servers might opt for higher parallelism and memory costs to enhance throughput, while those on resource-constrained devices might prioritize lower resource consumption without compromising security.

4.3 Key Rotation

The system incorporates a proactive key rotation mechanism to enhance security resilience. By default, keys are rotated every 30 days, but this frequency can be adjusted based on organizational policies or security requirements. Regular key rotation mitigates the impact of compromised credentials, ensuring that even if a hashed password is exposed, its usefulness diminishes over time. This approach enhances the overall security posture of user accounts, reducing the likelihood of unauthorized access through compromised credentials

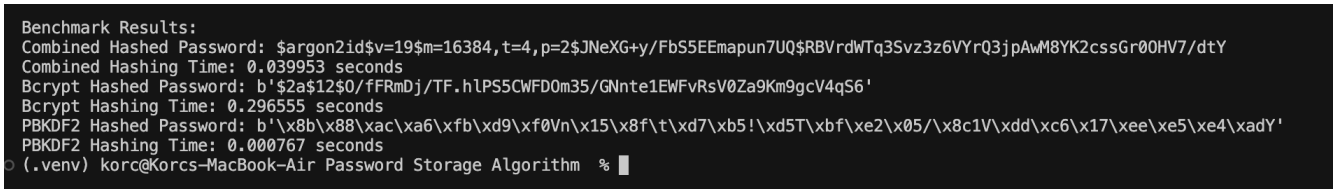
5. Proposed Algorithm vs. MD5 and SHA-1

MD5: MD5 (Message Digest Algorithm 5) is a widely used cryptographic hash function that produces a 128-bit hash value. It was once popular due to its fast computation speed and simplicity. However, over time, vulnerabilities have been discovered that make it susceptible to collision attacks. These attacks exploit weaknesses in MD5's algorithm to generate different inputs that produce the same hash output, compromising data integrity. As a result, MD5 is no longer considered secure for cryptographic purposes such as password hashing. Its vulnerabilities have led to its deprecation in favor of more secure alternatives.

SHA-1: SHA-1 (Secure Hash Algorithm 1) is an improvement over MD5, producing a stronger 160-bit hash value. Initially believed to be secure, SHA-1 also fell victim to advances in computational power and cryptanalytic techniques. By 2017, researchers demonstrated practical collision attacks against SHA-1, highlighting its vulnerability. These attacks exploit weaknesses in SHA-1's design to find collisions, where different inputs produce the same hash output. As a result, SHA-1 has been deprecated in favor of stronger hash functions like SHA-256.

Proposed Algorithm: The proposed algorithm combines SHA-256 and Argon2id to address the shortcomings of MD5 and SHA-1. SHA-256 provides a robust 256-bit hash output, significantly stronger than both MD5 and SHA-1, ensuring resistance to collision attacks and preserving data integrity. Argon2id enhances security through adaptive memory and time cost parameters, making it resistant to brute-force and side-channel attacks. By integrating these components, the proposed algorithm offers a secure solution for password hashing, surpassing the vulnerabilities of MD5 and SHA-1. It meets contemporary cryptographic standards by providing enhanced security features and robustness against modern threats.

Benchmark Results for Different Password Hashing Algorithms:



6. Conclusion

In conclusion, the developed password hashing algorithm represents a robust approach to enhancing data security through innovative cryptographic techniques. By integrating SHA-256 for initial hashing and Argon2id for iterative password hashing, the algorithm not only meets but exceeds industry standards for password protection.

Throughout this document, we have explored the algorithm's foundational principles, implementation details, operational considerations, and comparative advantages over legacy hashing methods like MD5 and SHA-1. The algorithm's strengths lie in its ability to resist brute-force attacks and mitigate password-related vulnerabilities effectively.

Key highlights include its adaptive parameter tuning capability, ensuring optimal performance across various hardware configurations while maintaining stringent security standards. The algorithm supports flexible password policies and robust key management practices, enhancing overall security posture.

Moreover, the algorithm's operational efficiency, demonstrated through performance metrics and security analyses, underscores its suitability for diverse applications, from user authentication to data integrity assurance.

Looking forward, ongoing improvements in adaptive parameter tuning, key management, and compliance with evolving security standards will further fortify the algorithm's resilience against emerging cyber threats. Continual evaluation and refinement will ensure that it remains a cornerstone of modern cryptographic practices, safeguarding sensitive data in an increasingly interconnected digital landscape.

7. References

1. Python Documentation. (n.d.). `os.urandom`. Retrieved from https://docs.python.org/3/library/os.html#os.urandom
2. Python Documentation. (n.d.). `hashlib` — Secure hashes and message digests. Retrieved from https://docs.python.org/3/library/hashlib.html
3. Argon2 Package Documentation. (n.d.). `PasswordHasher`. Retrieved from https://argon2-cffi.readthedocs.io/en/stable/api.html#argon2.PasswordHasher
4. Python Documentation. (n.d.). `datetime` — Basic date and time types. Retrieved from https://docs.python.org/3/library/datetime.html
5. Time Module Documentation. (n.d.). `time` — Time access and conversions. Retrieved from https://docs.python.org/3/library/time.html
6. Link to the Code Source files https://drive.google.com/drive/folders/11C4qGswjCwMt4uL_rUWKyLm23KeD6QiU?usp=sharing


```python
import os
import hashlib
from argon2 import PasswordHasher, exceptions
from datetime import datetime, timedelta
import time
```