Hashing Vs Encryption

Encryption scrambles data that can be decoded with a key. The intent is to pass the information to another party, and the recipient will use keys to decipher the data.

- **Purpose:** Data confidentiality, secrecy.
- **Process:** Two-way, reversible (encrypt/decrypt).
- **Output:** Ciphertext (variable length).
- **Key:** Requires a key (symmetric or asymmetric) to decrypt.
- **Example:** Encrypting an email so only the recipient with the key can read it.

Hashing also scrambles data, but the intent is to prove its authenticity. Administrators can run a check on hashed data to determine the contents haven't been touched or altered while in storage. No deciphering key exists.

- **Purpose:** Data integrity, verification, password storage.
- **Process:** One-way function, irreversible.
- **Output:**
- Fixed-length hash (e.g., SHA-256)
- .
- **Key:** No decryption key; used for comparison.
- **Example:** Storing password123 becomes a unique hash like a1b2c3d4e5f6, which can't be turned back into the password, but can be compared to a newly generated hash for login.

Identifying Common Hash Types

1. Legacy & Weak Hashes (Avoid for Security)

These hashes were once industry standards but are now considered "broken" because modern hardware can crack them in seconds or find "collisions" (where two different inputs produce the same hash).

- **MD5 (Message Digest 5):** Output: 128-bit (32 hex characters).
  - Status: Insecure for passwords.
  - Best For: Verifying file integrity (checking if a download was corrupted) where security isn't a concern.
- **SHA-1 (Secure Hash Algorithm 1)**: * Output: 160-bit (40 hex characters).
  - Status: Deprecated. Major browsers and certificate authorities stopped supporting it years ago.
  - Best For: Legacy software compatibility or version control systems like Git.

2. The SHA-2 Family (Current Standard)

Developed by the NSA, SHA-2 is the most widely used family of hash functions today. It is significantly more secure than SHA-1.

- **SHA-256**: * Output: 256-bit (64 hex characters).
  - Status: Very Secure.

    - Best For: SSL/TLS certificates, Bitcoin and blockchain technology, and verifying software updates.
- **SHA-512**: * Output: 512-bit (128 hex characters).
    - Status: Extremely Secure.
    - Best For: High-security applications and environments where 64-bit CPUs can process the longer strings efficiently.

3. Password-Specific Hashes (Slow & Salted)

Standard hashes like SHA-256 are designed to be *fast*. However, for passwords, you want a *slow* hash to make brute-force attacks difficult. These often include a "salt" (random data added to the password) to prevent identical passwords from having the same hash.

- **bcrypt**: * Mechanism: Based on the Blowfish cipher; it includes a "cost factor" that allows you to increase the time it takes to compute the hash as hardware gets faster.
    - Best For: Modern web application password storage.
- **Argon2**: * Mechanism: The winner of the 2015 Password Hashing Competition. It is resistant to GPU and ASIC-based cracking attacks.
    - Best For: The current "gold standard" for password hashing.
- **scrypt**: * Mechanism: Designed to require large amounts of memory (RAM), making it very expensive for attackers to build custom hardware to crack it.

Generating Password Hashes in Kali Linux

1. Using OpenSSL (The Professional Standard)

OpenSSL is a robust tool that can generate hashes for almost any algorithm. It is the most common way to generate hashes that include a **salt** (random data added to the password to prevent pre-computed attacks).

- **Generate an MD5 Hash:** openssl passwd -1 "yourpassword"
- **Generate a SHA-256 Hash:** openssl passwd -6 "yourpassword"
- **Generate a SHA-512 Hash:** openssl passwd -5 "yourpassword"

Note: The -1, -5, and -6 flags correspond to specific algorithms (MD5, SHA-256, and SHA-512 respectively).

2. Using Coreutils (Quick Checksums)

If you just need a raw hash of a string without salts or formatting (often used for file integrity or basic verification), you can use the standard utility commands.

- **MD5:** echo -n "password123" | md5sum
- **SHA-1:** echo -n "password123" | sha1sum
- **SHA-256:** echo -n "password123" | sha256sum

3. Using Python (Scripting & Automation)

Since Kali comes with Python pre-installed, you can use the hashlib library to generate hashes. This is useful if you want to build a small tool to hash a list of passwords from a file.

Open your terminal and type python3, then enter:

Python

import hashlib

Generate a SHA-256 hash

password = "mysecurepassword"

hash_object = hashlib.sha256(password.encode())

print(hash_object.hexdigest())

4. Using Mkpasswd (User Management)

The mkpasswd utility is specifically designed for generating hashes in the format used by the /etc/shadow file in Linux systems.

- **Generate a bcrypt hash:** mkpasswd -m bcrypt "yourpassword"
- **Generate a SHA-512 hash with a specific salt:** mkpasswd -m sha-512 "yourpassword" -s "randomsalt"

Cracking weak hashes using wordlists

1. Preparation: The Wordlist

A "wordlist" or "dictionary" is a text file containing millions of common passwords. Kali Linux comes pre-loaded with the industry-standard list: **rockyou.txt**.

- **Location:** /usr/share/wordlists/rockyou.txt.gz
- **Unzip it first:** Since it's compressed, you must unzip it before use: sudo gunzip /usr/share/wordlists/rockyou.txt.gz

2. Method A: Using John the Ripper (Beginner Friendly)

John the Ripper (JTR) is excellent because it often auto-detects the hash type for you.

Step 1: Create a target file Put your hash into a text file: echo "5e8648d5d1bb3344690c51496c1da911" > hash_to_crack.txt

Step 2: Run the attack Point John to your hash file and your wordlist: john --wordlist=/usr/share/wordlists/rockyou.txt hash_to_crack.txt

Step 3: View results If John finds a match, it will display it. To see previously cracked passwords: john --show hash_to_crack.txt

3. Method B: Using Hashcat (High Performance)

Hashcat is significantly faster because it can use your GPU (Graphics Card) to attempt millions of passwords per second. However, you must manually specify the **Hash Mode (-m)**.

Common Hash Modes:

- **0:** MD5
- **100:** SHA-1
- **1400:** SHA-256
- **1800:** SHA-512 (Unix)
- **3200:** bcrypt

The Command: hashcat -m 0 -a 0 hash_to_crack.txt /usr/share/wordlists/rockyou.txt

Explanation of flags:

- -m 0: Tells Hashcat the target is MD5.
- -a 0: Specifies a "Straight" attack (Dictionary attack).

4. Advanced: Using "Rules"

Sometimes a password isn't exactly in the wordlist (e.g., the word is password but the user set it to Password123!). Both tools can apply **Rules** to mutate the wordlist on the fly.

- **In John:** john --wordlist=... --rules hash.txt
- **In Hashcat:** hashcat -m 0 hash.txt wordlist.txt -r /usr/share/hashcat/rules/best64.rule

Attack Methodologies: Brute Force vs. Dictionary

- **Dictionary Attack**:
  - How it works: The tool tries every word in a pre-defined list (dictionary).
  - Pros/Cons: Extremely fast but fails if the password is not in the list (e.g., P@ssw0rd!).
- **Brute Force Attack**:
  - How it works: The tool systematically tries every possible combination of characters (a, b, c... 1, 2, 3... !@#).
  - Pros/Cons: Guaranteed to find the password eventually, but takes an exponential amount of time as password length increases.

Why Weak Passwords Fail?

Weak passwords (e.g., 123456, password, qwerty) fail because they are:

1. **Present in Wordlists**: They are the first items tested in dictionary attacks.
2. **Short in Length:** They are easily exhausted by brute force in seconds.
3. **Predictable**: Many users follow patterns (e.g., Capitalizing the first letter and adding "1" at the end) which attackers use to create "rules" for cracking.

Multi-Factor Authentication (MFA)

MFA adds layers of security by requiring two or more pieces of evidence (factors) to authenticate.

- **Something you know**: Password or PIN.

- **Something you have**: Smartphone (OTP), Security key (Yubikey).
- **Something you are**: Fingerprint, Facial recognition.

Importance: Even if an attacker successfully cracks a password hash, they cannot gain access without the second factor, effectively neutralizing the impact of the compromised password.

Recommendations for Strong Authentication

To protect users and systems, implement the following:

- **Password Complexity**: Require a minimum of 12–16 characters, including a mix of uppercase, lowercase, numbers, and symbols.
- **Use Modern Hashing**: Always use bcrypt, Argon2, or scrypt with a unique salt for every user to prevent rainbow table attacks.
- **Mandatory MFA**: Implement TOTP (Time-based One-Time Password) or hardware keys for all administrative and user accounts.
- **Rate Limiting**: Lock accounts or implement delays after multiple failed login attempts to thwart brute force attacks.
- **Transition to Passkeys**: Encourage the use of FIDO2/WebAuthn for passwordless authentication, which is phishing-resistant.