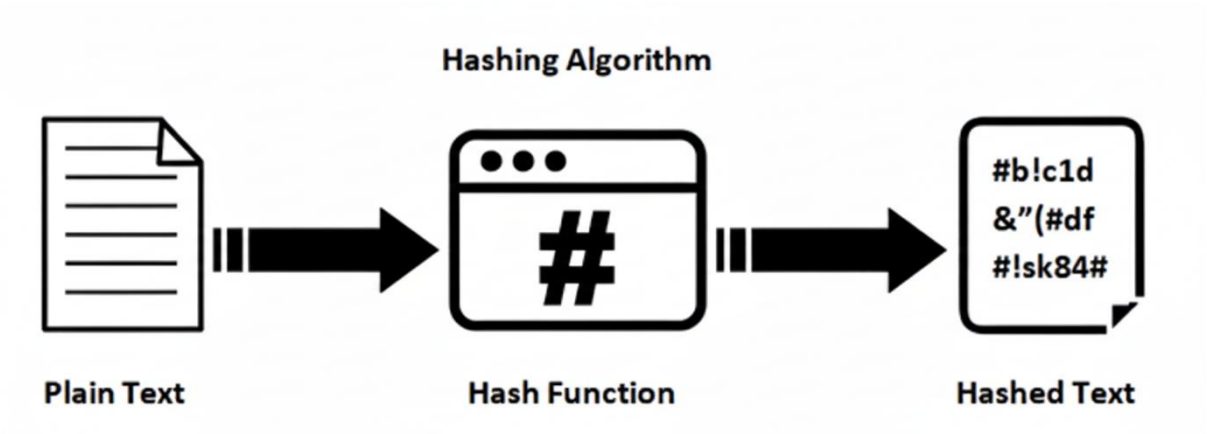
Hashing Fundamentals: From Password Security to Data Integrity

A Comprehensive Guide to One-Way Cryptographic Functions



Introduction

Imagine you've just spent hours downloading a 6GB Linux distribution ISO file. How do you know the file downloaded correctly without corruption? What if someone tampered with it during download? Or consider receiving this file on a USB drive from a colleague-how can you verify it's identical to the official release?

The elegant solution lies in comparing hash values. If two hash values match, you can confidently conclude the files are identical, bit-for-bit. But what exactly is a hash value, and how does this magical verification work?

A hash value is a fixed-size string of characters computed by a hash function-a mathematical algorithm that takes input of any size and produces output of fixed length. This fundamental cryptographic tool enables applications from password security to blockchain technology, operating invisibly yet critically in everyday digital interactions.

Learning Objectives

This comprehensive guide explores:

- Hash functions, their properties, and collision prevention mechanisms
- The critical role of hashing in modern authentication systems
- Recognizing and identifying different stored hash formats
- Hash cracking techniques, tools, and practical methodologies
- Using hashing for file integrity protection and verification

Understanding Hash Functions

Hash functions fundamentally differ from encryption in a crucial way: there is no key, and the process is designed to be computationally impossible (or impractical) to reverse. You cannot decrypt a hash-you can only attempt to recreate it by hashing different inputs until you find a match.

Core Characteristics of Good Hash Functions

A cryptographic hash function takes input data of any size and creates a summary or digest with these essential properties:

- **Deterministic:** The same input always produces the same output. Hash 'password123' and you'll always get the same result
- **Fixed Output Size:** Whether you hash a single character or an entire encyclopedia, the output length remains constant. MD5 always produces 128 bits, SHA-256 always produces 256 bits
- **Fast to Compute:** Calculating a hash should be quick and efficient
- **Computationally Infeasible to Reverse:** Given a hash, you should not be able to determine the original input (one-way function)
- **Collision Resistant:** It should be extremely difficult to find two different inputs that produce the same hash
- **Avalanche Effect:** Any slight change in input, even a single bit, should cause a dramatic, unpredictable change in output

The Avalanche Effect: A Real Example

To truly understand the avalanche effect, let's examine two files that differ by exactly one bit. Consider the letters T and U in ASCII:

- Letter T = 54 in hexadecimal = 01010100 in binary
- Letter U = 55 in hexadecimal = 01010101 in binary

These letters differ by only the last bit (0 vs. 1). Now watch what happens when we hash them with different algorithms:

Algorithm	Hash Value
File 1 (T)	
MD5	`b9ece18c950afbfa6b0fdbfa4ff731d3`
SHA-1	`c2c53d66948214258a26ca9ca845d7ac0c17f8e7`
SHA-256	`e632b7095b0bf32c260fa4c539e9fd7b852d0de454e9be26f24d0d6f91d069d3`
File 2 (U)	
MD5	`4c614360da93c0a041b22e537de151eb`
SHA-1	`b2c7c0caa10a0cca5ea7d69e54018ae0c0389dd6`
SHA-256	`a25513c7e0f6eaa80a3337ee18081b9e2ed09e00af8531c8f7bb2542764027e7`

Key Observation: Despite changing only ONE bit out of eight, the hash values are completely, utterly different. There's no pattern, no similarity-the entire hash changed dramatically. This is the avalanche effect in action, and it's what makes hash functions so powerful for detecting even the tiniest changes.

Encoding Note: Hash functions output raw bytes, typically displayed in hexadecimal format (where each byte is represented by two hexadecimal digits) or base64 encoding. The commands md5sum, sha1sum, and sha256sum produce hexadecimal output by default.

Why Hashing Is Everywhere

Hashing operates invisibly throughout the internet, protecting data integrity and password confidentiality in ways most users never see. Every time you log into a website, download software, or verify a file, hashing works behind the scenes.

Real-World Daily Applications

- **Website Login:** When you log into TryHackMe, the server hashes your password and compares it to the stored hash. The server never sees or stores your actual password
- **Computer Login:** Your Windows or Linux login password is verified using hashed values stored in the system
- **Software Downloads:** When downloading Linux ISOs, Fedora or Ubuntu provides SHA-256 checksums to verify file integrity
- **Git Version Control:** Git uses SHA-1 hashes to identify every commit, ensuring code integrity
- **Blockchain:** Cryptocurrencies like Bitcoin use SHA-256 hashing extensively for proof-of-work
- **Digital Forensics:** Investigators hash evidence files to prove they haven't been tampered with

Hash Collisions: The Pigeonhole Problem

A hash collision occurs when two different inputs produce the exact same hash output. While hash functions are designed to minimize collision probability, mathematics guarantees their existence through the pigeonhole principle.

Understanding the Pigeonhole Principle

Imagine you have a hash function that produces 4-bit output. With 4 bits, you can create 2^4 = 16 different values (0000 through 1111). But the number of possible inputs is unlimited-you could hash millions, billions, or trillions of different messages.

It's like having 21 pigeons and only 16 pigeonholes. No matter how you arrange them, at least one pigeonhole must contain more than one pigeon. Similarly, with unlimited inputs and limited outputs, some inputs must share the same hash value.

Real-World Impact: MD5 (128-bit output = 2^128 possible hashes) and SHA-1 (160-bit output = 2^160 possible hashes) have both suffered successful collision attacks. Researchers can now intentionally create two different files with identical MD5 or SHA-1 hashes. This is why these algorithms are considered broken for security purposes.

External Resources on Collisions

→ **MD5 Collision Demo:** Visit https://www.mscs.dal.ca/~selinger/md5collision/ to see two different files with identical MD5 hashes
→ **SHAttered Attack:** Read about the SHA-1 collision at https://shattered.io/

Password Storage: Lessons from Major Breaches

Understanding how NOT to store passwords is just as important as knowing the correct method. Let's examine three catastrophic real-world failures that exposed millions of user credentials.

Case Study 1: RockYou (2009) - Plaintext Disaster

RockYou, a company developing social media applications and widgets, stored over 32 million user passwords in plaintext-completely unencrypted, unhashed, just readable text in a database. When their database was breached in December 2009, attackers gained immediate access to every password.

The compromised password list became the infamous rockyou.txt file, containing 14,344,392 unique passwords. This file lives on every penetration testing distribution (like Kali Linux) in /usr/share/wordlists/ and remains one of the most effective wordlists for password cracking today.

Top 10 most common passwords from RockYou:

- 123456 (used by 290,729 people!)
- 12345
- 123456789
- password
- iloveyou

The Lesson: Never, ever store passwords in plaintext. Period. This is security malpractice of the highest order.

Case Study 2: Adobe (2013) - Encryption Gone Wrong

Adobe's breach affected 153 million user accounts. Unlike RockYou, Adobe at least attempted to protect passwords-they encrypted them using 3DES (Triple DES), a symmetric encryption algorithm. However, they made critical mistakes:

1. Used encryption instead of hashing (encryption is reversible if you get the key)
2. Stored password hints in plaintext
3. Password hints often contained the actual password or gave it away completely

Once attackers obtained the encryption key, they could decrypt all passwords. Combined with helpful hints like 'mypassword123' or 'same as email,' the breach became catastrophic.

The Lesson: Encryption is NOT appropriate for password storage because you need to store the key. If the key is compromised (as it often is in breaches), all passwords are exposed. Hash functions don't use keys.

Case Study 3: LinkedIn (2012) - Unsalted Hashing

LinkedIn made a better choice by hashing passwords using SHA-1. However, they committed two critical errors:

1. Used SHA-1, which was already considered weak and vulnerable to collisions
2. Did NOT use password salting (we'll explain this shortly)

Without salts, identical passwords produced identical hashes. This meant:

- Attackers could see which users shared passwords
- Rainbow table attacks became highly effective
- Cracking one password instantly revealed all accounts using that password

The breach affected 6.5 million accounts, with millions of passwords cracked within days using rainbow tables.

The Lesson: Hashing alone isn't enough. You must use a strong algorithm (not MD5 or SHA-1) and implement salting to prevent rainbow table attacks.

Rainbow Tables: Pre-Computed Hash Databases

Rainbow tables represent one of the most powerful attacks against unsalted password hashes. Understanding how they work reveals why salting is absolutely essential.

How Rainbow Tables Work

Imagine creating a massive lookup table that maps every possible password to its hash value. For example:

Password	MD5 Hash
password	`5f4dcc3b5aa765d61d8327deb882cf99`
123456	`e10adc3949ba59abbe56e057f20f883e`
letmein	`0d107d09f5bbe40cade3de5c71e9e9b7`
qwerty	`d8578edf8458ce06fbc5bb76a58c5ca4`

Rainbow tables trade computation time for disk space. Instead of hashing millions of passwords repeatedly, you hash them once, store the results, then perform instant lookups. Modern rainbow tables contain billions of password-hash pairs.

Online Rainbow Table Services

Several websites maintain massive rainbow tables for instant hash cracking:

→ **CrackStation:** https://crackstation.net/ - Free hash cracking with 15GB+ rainbow tables
→ **Hashes.com:** https://hashes.com/en/decrypt/hash - Massive database of pre-computed hashes
→ **MD5 Decrypt:** https://md5decrypt.net/ - Specialized MD5 rainbow table lookups

Try It: Hash the password 'password123' using MD5 (you'll get 482c811da5d5b4bc6d497ffa98491e38). Search this hash on CrackStation-it will instantly return 'password123' without any computation!

Salt: The Rainbow Table Defense

Salting completely defeats rainbow table attacks through a simple but brilliant technique: add random data to each password before hashing.

How Salting Works

A salt is a random string added to a password before hashing. Each user gets a unique salt, stored alongside their hash in the database. Here's the process:

1. User creates password: `mypassword`
2. System generates random salt: `8Kx9$mP2`
3. Concatenate password + salt: `mypassword8Kx9$mP2`
4. Hash the combined string
5. Store both the salt and the hash

Example with two users:

User	Salt	Hash (both use 'password123')
Alice	`f7G2k`	`a3c4d5e6f7g8h9i0j1k2l3m4n5o6p7q8`
Bob	`9XmQ!`	`z9y8x7w6v5u4t3s2r1q0p9o8n7m6l5k4`

Critical Point: Even though Alice and Bob use the SAME password ('password123'), their hashes are completely different because of unique salts. An attacker cracking Alice's password doesn't automatically crack Bob's!

Why Salts Don't Need to Be Secret: Salts are typically stored in plaintext right next to the hash. They don't provide security through secrecy-they provide security by making rainbow tables useless. An attacker would need to create a new rainbow table for EVERY salt, which is computationally infeasible.

Modern Algorithms with Built-in Salting

These algorithms automatically handle salting-you don't need to implement it yourself:

- **Argon2:** Winner of the Password Hashing Competition (2015), currently the best choice
- **Bcrypt:** Based on Blowfish cipher, widely used and trusted
- **Scrypt:** Memory-hard function resistant to GPU cracking
- **PBKDF2:** NIST-approved key derivation function

Recognizing Hash Formats

From an offensive security perspective, identifying hash types enables effective cracking strategies. Different algorithms require different tools and approaches.

Linux Password Hashes (/etc/shadow)

Linux stores password hashes in /etc/shadow (readable only by root). The format is: $prefix$options$salt$hash

Example line from /etc/shadow:

```
root:$y$j9T$76UzfgEM5PnymhQ7TlJey1$/OOSg64dhfF.TigVPdzqiFang6uZA4QA1pzzegKdVm4:19965:0:999
99:7:::
```

Breaking down the hash field:

- `$y$` = yescrypt algorithm
- `j9T` = algorithm parameters/cost factor
- `76UzfgEM5PnymhQ7TlJey1` = random salt (unique to this user)
- `/OOSg64dhfF.TigVPdzqiFang6uZA4QA1pzzegKdVm4` = actual password hash

Prefix	Algorithm & Description
y	yescrypt - Default and recommended for modern Linux systems
gy	gost-yescrypt - Uses GOST R 34.11-2012 hash + yescrypt
7	scrypt - Password-based key derivation function
$2b$, $2y$, $2a$	bcrypt - Blowfish-based, very common and secure
6	sha512crypt - SHA-512 based, used in older Linux systems
5	sha256crypt - SHA-256 based
1	md5crypt - MD5-based, legacy systems only (WEAK)

Windows Password Hashes (NTLM)

Windows uses NTLM (NT LAN Manager), a variant of MD4, for password hashing. These hashes are stored in the SAM (Security Accounts Manager) database. NTLM hashes look identical to MD4 and MD5 hashes (32 hexadecimal characters), so context is crucial for identification.

Example NTLM hash:

```
Admin:500:aad3b435b51404eeaad3b435b51404ee:8846f7eaee8fb117ad06bdd830b7586c:::
```

Tools like Mimikatz can extract these hashes from Windows systems, bypassing normal security protections.

External Resources for Hash Identification

→ **Hashcat Example Hashes:** https://hashcat.net/wiki/doku.php?id=example_hashes - Comprehensive hash format database
→ **hashID Tool:** Command-line tool for automated hash identification (unreliable for many formats)
→ **hash-identifier:** Alternative Python-based hash identification tool

Hash Cracking: Tools and Techniques

Password hashes cannot be decrypted-they must be cracked through computational brute-force. The process involves hashing millions of potential passwords and comparing results to target hashes.

Hashcat: GPU-Accelerated Cracking

Hashcat leverages GPU power for extremely fast hash cracking. Modern graphics cards contain thousands of cores specialized for parallel mathematical operations, making them ideal for hashing calculations.

Basic Hashcat Syntax:

```
hashcat -m <hash_type> -a <attack_mode> <hashfile> <wordlist>
```

Parameters explained:

- `-m <hash_type>` - Numeric code specifying hash algorithm
- `-a <attack_mode>` - Attack strategy (0 = straight/dictionary, 3 = brute-force)
- `<hashfile>` - Text file containing hashes to crack
- `<wordlist>` - Password dictionary file (e.g., rockyou.txt)

Common Hash Type Codes:

- `0` = MD5
- `100` = SHA-1
- `1000` = NTLM (Windows)
- `1800` = sha512crypt (Linux 6)
- `3200` = bcrypt

Practical Examples:

Example 1: Cracking MD5 hashes

```
hashcat -m 0 -a 0 hash.txt /usr/share/wordlists/rockyou.txt
```

Example 2: Cracking bcrypt hashes

```
hashcat -m 3200 -a 0 bcrypt_hash.txt rockyou.txt
```

Example 3: Cracking NTLM (Windows) hashes

```
hashcat -m 1000 -a 0 ntlm.txt rockyou.txt
```

John the Ripper: CPU-Based Cracking

John the Ripper uses CPU processing and works effectively in virtual machines. While generally slower than GPU-based Hashcat, John excels at automatic format detection and offers powerful rule-based attacks.

Basic John Syntax:

```
john --wordlist=/usr/share/wordlists/rockyou.txt hashfile.txt
```

`john --show hashfile.txt` - Display cracked passwords

GPU vs. CPU Cracking

Modern GPUs contain thousands of specialized cores for parallel computation. A high-end GPU can test billions of password combinations per second for fast algorithms like MD5 or NTLM. However, algorithms like Bcrypt and Scrypt intentionally resist GPU acceleration, maintaining comparable speeds on CPUs to slow cracking attempts.

Virtual Machine Limitations: VMs typically lack direct GPU access. For maximum Hashcat performance, run it on the host operating system with GPU drivers installed. John the Ripper works effectively in VMs but still benefits from host execution to avoid virtualization overhead.

External Resources for Hash Cracking

- → **Hashcat Official Site:** https://hashcat.net/hashcat/
- → **John the Ripper:** https://www.openwall.com/john/
- → **Wordlists Collection:** https://github.com/danielmiessler/SecLists

Data Integrity and File Verification

Beyond password security, hashing provides critical file integrity verification. Since identical inputs always produce identical outputs, hash comparison instantly confirms file authenticity.

Practical Integrity Checking

Example: Verifying a Fedora Linux Download

Official SHA-256 checksum from Fedora website:

```
8d3cb4d99f27eb932064915bc9ad34a7529d5d073a390896152a8a899518573f
```

Your downloaded file's hash:

```
sha256sum Fedora-Workstation-40-1.14.iso
```

If the hashes match perfectly, you have an exact, unmodified copy. If even one character differs, the file is corrupted or tampered with.

HMAC: Authenticated Hashing

HMAC (Keyed-Hash Message Authentication Code) combines hashing with secret keys to verify both authenticity (who created it) and integrity (hasn't been changed).

HMAC provides two guarantees:

1. **Authenticity:** Only someone with the secret key could create this HMAC
2. **Integrity:** The message hasn't been modified (any change invalidates the HMAC)

HMAC is used extensively in API authentication (like JWT tokens), secure messaging, and data transmission protocols.

Hashing vs. Encoding vs. Encryption

These terms are often confused, but they serve completely different purposes:

Type	Purpose	Reversible?
Hashing	Integrity verification, password storage	NO - One-way function
Encoding	Format conversion (UTF-8, Base64)	YES - No security
Encryption	Confidentiality protection	YES - With key

Example to clarify:

- **Hashing 'password':** → 5f4dcc3b5aa765d61d8327deb882cf99 (can't reverse)
- **Encoding 'password' (Base64):** → cGFzc3dvcmQ= (easily decoded)
- **Encrypting 'password' (AES):** → Encrypted bytes (decryptable with key)

Key Takeaways

- Hash functions create fixed-size digests from any input; avalanche effect ensures tiny changes produce completely different outputs
- Collisions are mathematically inevitable (pigeonhole principle) but good algorithms make them negligible
- MD5 and SHA-1 are cryptographically broken; use SHA-256, SHA-3, or modern algorithms
- Never store passwords in plaintext or use encryption; hash with Argon2, Bcrypt, or Scrypt
- Salt defeats rainbow tables by making each hash unique even for identical passwords
- Linux hashes use $prefix$ format (/etc/shadow); Windows uses NTLM (SAM database)
- Hashcat (GPU) and John the Ripper (CPU) crack hashes through dictionary/brute-force attacks
- Hashing ≠ Encoding ≠ Encryption; each serves distinct security purposes

Conclusion

Hashing represents a fundamental cryptographic primitive enabling password security without storage and file integrity verification through simple comparison. Its one-way nature-fast to compute forward, impossible to reverse-makes it perfect for authentication systems where you need to verify knowledge of a secret without storing the secret itself.

From understanding why the RockYou breach exposed 14 million passwords to implementing proper Bcrypt+salt storage, this knowledge spans defensive and offensive security perspectives. The avalanche effect, pigeonhole principle, rainbow table defense, and practical hash cracking with Hashcat form essential skills for cybersecurity professionals.

Whether verifying downloaded Linux ISOs, auditing password storage implementations, or cracking CTF challenges, hashing pervades modern security. Understanding these fundamentals provides the foundation for advanced cryptographic applications and security architecture decisions protecting billions of users worldwide.

Continue exploring practical implementations through hands-on hash cracking exercises and secure password storage development.