

Hash Types & Identification

A comprehensive reference for recognizing, identifying, and understanding the performance characteristics of common hash algorithms.

Introduction

Before you can crack a hash, you must know what algorithm produced it. Choosing the wrong hashcat mode wastes time and produces no results. This module covers the most common hash types you will encounter in penetration testing, CTF competitions, and real-world credential recovery — and teaches you how to identify them reliably.

Hash identification is both an art and a science. Many hash formats are self-identifying through prefixes or fixed lengths. Others require context clues: where was the hash found? What application or operating system produced it? The combination of visual inspection, tooling, and contextual knowledge will make you effective.

Common Hash Types Reference

MD5 (mode 0)

MD5 produces a 128-bit (32 hex character) digest. It is the most commonly encountered hash in the wild, found in legacy web applications, database dumps, and file integrity checks. MD5 has been cryptographically broken since 2004 — collision attacks are trivial — but it remains widespread.

```
# Example MD5 hash
5d411402abc4b2a76b9719d911017c592

# Crack with hashcat
hashcat -m 0 -a 0 hash.txt wordlist.txt
```

Characteristics: Always 32 hex characters. No prefix. Pure lowercase hex (`a-f`, `0-9`). On modern GPUs, cracking speed exceeds 100 billion hashes per second.

SHA-1 (mode 100)

SHA-1 produces a 160-bit (40 hex character) digest. It was the standard for digital signatures and certificates until 2017 when Google demonstrated a practical collision attack (SHAttered). Still found in older Git commits, some web applications, and legacy systems.

```
# Example SHA-1 hash
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

# Crack with hashcat
hashcat -m 100 -a 0 hash.txt wordlist.txt
```

Characteristics: Always 40 hex characters. No prefix. Distinguishable from MD5 purely by length.

SHA-256 (mode 1400)

SHA-256 is part of the SHA-2 family and produces a 256-bit (64 hex character) digest. It is currently the most widely used secure hash function, found in Bitcoin mining, TLS certificates, and modern password storage systems (when used with proper salting).

```
# Example SHA-256 hash
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

# Crack with hashcat
hashcat -m 1400 -a 0 hash.txt wordlist.txt
```

Characteristics: Always 64 hex characters. No prefix in raw form. Roughly 10x slower to crack than MD5.

SHA-512crypt (mode 1800)

SHA-512crypt is the default password hashing scheme on most modern Linux distributions. It uses SHA-512 internally but applies multiple rounds of hashing (default 5000) and includes a salt. Found in `/etc/shadow`.

```
# Example SHA-512crypt hash (from /etc/shadow)
$6$rounds=5000$salt$Uf3GYdFhPGb8w8e4KiRB5TkPJJBxr0vMXUbwpjthLCjvX...
```

Characteristics: Always starts with `$6$`. May include `rounds=NNNN$`. The salt follows the second `$`. The hash value is Base64-encoded (not hex). Cracking speed is in the

low millions of hashes per second.

NTLM (mode 1000)

NTLM is the password hash format used by Windows. It is an unsalted MD4 hash of the Unicode password. Despite being extremely fast to crack (faster than MD5 on GPUs), it remains the primary credential format in Active Directory environments.

```
# Example NTLM hash
a4f49c406510bdcab6824ee7c30fd852

# Crack with hashcat
hashcat -m 1000 -a 0 hash.txt wordlist.txt
```

Characteristics: 32 hex characters — same length as MD5. Distinguishing between MD5 and NTLM requires context: if it came from a Windows SAM database, `secretsdump.py`, or mimikatz, it is NTLM. If from a web application, it is likely MD5.

MD5 vs NTLM: Both are 32 hex characters. You cannot distinguish them visually. Context is everything. If unsure, try both modes — one will succeed if the hash matches.

bcrypt (mode 3200)

bcrypt is a deliberately slow password hashing algorithm based on the Blowfish cipher. It is the gold standard for password storage in modern web applications. The cost factor (work factor) determines how many rounds of hashing are applied — each increment doubles the computation time.

```
# Example bcrypt hash
$2a$12$WApznUPhDubN0oeveSXHp.CLGK5EIG0I3gJGWYUzPm/K3yBR3Mfvy

# Structure: $2a$ [cost] $ [22-char salt] [31-char hash]
# Variants: $2a$, $2b$, $2y$ — all use mode 3200
```

Characteristics: Always starts with `$2a$`, `$2b$`, or `$2y$`. The two digits after the prefix are the cost factor (typically 10-14). Total length is always 60 characters. Cracking speed on high-end GPUs is roughly 100,000 hashes per second at cost 12.

WPA/WPA2 (mode 22000)

WPA/WPA2 hashes are captured from wireless network handshakes or PMKID exchanges. The hash format used by hashcat is a specially structured line containing the network SSID, MAC addresses, and cryptographic material from the capture.

```
# Example WPA hash (PMKID format)
WPA*02*aabb...ccdd*aabbccddeeff*112233445566*MyNetwork*...
```

Characteristics: Starts with `WPA*`. Contains MAC addresses and SSID in the hash line. Captured using tools like hcxdumpool or converted from .cap files using hcxpcapngtool. Cracking speed is comparable to SHA-512crypt — around 1-2 million hashes per second.

Hash Identification Methodology

Follow this systematic approach when you encounter an unknown hash:

1. **Check for a prefix.** If the hash starts with `$`, `WPA*`, or another recognizable marker, identification is usually immediate.
2. **Count the characters.** 32 hex = MD5 or NTLM. 40 hex = SHA-1. 64 hex = SHA-256. 128 hex = SHA-512.
3. **Consider the context.** Where did the hash come from? Windows = NTLM. Linux shadow = SHA-512crypt or bcrypt. Web app database = MD5, SHA-1, or bcrypt.
4. **Use identification tools.** Run `hashid` or `hash-identifier` for automated suggestions.
5. **Verify with hashcat.** Use `hashcat --example-hashes` to confirm the format matches your target mode.

Using hashid

`hashid` is a Python tool that analyzes a hash string and suggests possible algorithms, optionally including the corresponding hashcat mode numbers.

```
# Install hashid
pip install hashid

# Basic usage
hashid '5d41402abc4b2a76b9719d911017c592'

# Analyzing '5d41402abc4b2a76b9719d911017c592'
```

```
# [+] MD2
# [+] MD5
# [+] MD4
# [+] Double MD5
# [+] NTLM
# ...

# Show hashcat mode numbers
hashid -m '5d41402abc4b2a76b9719d911017c592'
# [+] MD5 [Hashcat Mode: 0]
# [+] NTLM [Hashcat Mode: 1000]

# Identify from a file
hashid -m -f hashes.txt
```

Using hash-identifier

`hash-identifier` is an interactive tool pre-installed on Kali Linux. It prompts you to paste a hash and returns a ranked list of possible algorithms. It is less precise than `hashid` but useful as a quick sanity check.

```
# Launch the tool
hash-identifier

# Paste your hash at the prompt
HASH: 5d41402abc4b2a76b9719d911017c592
# Possible Hashs:
# [+] MD5
# [+] Domain Cached Credentials - MD4(MD4(($pass)).(strtolower($username)))
```

Using hashcat --example-hashes

This is the definitive reference. It shows the exact hash format hashcat expects for every supported mode. Use `grep` to filter results.

```
# Search by algorithm name
hashcat --example-hashes | grep -i 'sha-512' -B2 -A5

# Search by mode number
hashcat --example-hashes | grep -A4 'MODE: 3200'

# Dump everything to a file for reference
hashcat --example-hashes > all_modes.txt
```

Salted vs Unsalted Hashes

Understanding salting is critical for both identification and attack planning.

What is a salt?

A salt is a random value generated for each password before hashing. The salt is stored alongside the hash (it is not secret). Its purpose is to ensure that two users with the same password produce different hash values.

Why salts defeat precomputation

Without salts, an attacker can build a rainbow table — a massive precomputed lookup table mapping hashes to passwords. With a 16-byte salt, each rainbow table would only work for one specific salt value, making the approach computationally infeasible.

Salted hash formats in hashcat

Mode	Description	Format
0	MD5 (unsalted)	<code>hash</code>
10	<code>md5(\$pass.\$salt)</code>	<code>hash:salt</code>
20	<code>md5(\$salt.\$pass)</code>	<code>hash:salt</code>
100	SHA-1 (unsalted)	<code>hash</code>
110	<code>sha1(\$pass.\$salt)</code>	<code>hash:salt</code>
120	<code>sha1(\$salt.\$pass)</code>	<code>hash:salt</code>
1800	SHA-512crypt	<code>\$6\$salt\$hash</code> (salt embedded)
3200	bcrypt	<code>\$2a\$cost\$salthash</code> (salt embedded)

Tip: When you see a `hash:salt` format in a database dump, the colon-separated value after the hash is the salt. Hashcat expects this exact format — do not strip the salt.

Performance Characteristics

The speed at which you can test password candidates against a hash is determined by the algorithm's computational cost. This has profound implications for attack planning.

Fast hashes

MD5, SHA-1, NTLM, and SHA-256 (raw, unsalted) are all "fast hashes." They were designed for data integrity verification, not password storage. A single GPU can test billions of candidates per second against these algorithms. This means:

- Full brute-force of all 8-character passwords (lowercase + digits) takes minutes
- Large wordlists (hundreds of millions of entries) with complex rules are practical
- Combinator attacks and mask attacks over wide keyspaces complete quickly

Slow hashes

bcrypt, scrypt, Argon2, SHA-512crypt, and PBKDF2 are "slow hashes" or "key derivation functions." They are specifically designed to be computationally expensive, making brute-force attacks impractical. Characteristics:

- Built-in salt (prevents rainbow tables)
- Configurable cost factor (iterations can be increased as hardware improves)
- Memory-hard variants (scrypt, Argon2) resist GPU acceleration
- Cracking speed drops from billions/sec to thousands/sec

Benchmarking your hardware

Always benchmark before starting a real attack. Hashcat includes a built-in benchmarking mode:

```
# Benchmark all hash types
hashcat -b

# Benchmark a specific mode
hashcat -b -m 0      # MD5
hashcat -b -m 1000   # NTLM
hashcat -b -m 3200   # bcrypt

# Example output (RTX 4090):
```

```
# Hashmode: 0 - MD5
# Speed.#1.....: 164.3 GH/s
#
# Hashmode: 3200 - bcrypt $2*$, Blowfish (Unix)
# Speed.#1.....: 105.2 kH/s
```

Calculating attack feasibility

Use benchmark results to estimate how long an attack will take:

```
# Keyspace calculation
# All 8-char lowercase + digits: 36^8 = 2,821,109,907,456 candidates

# MD5 at 164 GH/s:
# 2.82 trillion / 164 billion/sec = ~17 seconds

# bcrypt (cost 12) at 105 kH/s:
# 2.82 trillion / 105,000/sec = ~26,867,713 seconds = ~311 days
```

Planning rule of thumb: If your estimated attack time exceeds 24 hours, consider narrowing the keyspace with targeted rules, shorter masks, or smaller wordlists. For slow hashes, prioritize quality of your wordlist over quantity.

Quick Reference Table

Algorithm	Mode	Length/Format	Salted	Speed Class
MD5	0	32 hex	No	Fast
SHA-1	100	40 hex	No	Fast
SHA-256	1400	64 hex	No	Medium
SHA-512crypt	1800	<code>\$6\$...</code>	Yes	Slow
NTLM	1000	32 hex	No	Fast
bcrypt	3200	<code>\$2a\$...</code> (60 chars)	Yes	Very Slow

Algorithm	Mode	Length/Format	Salted	Speed Class
WPA/WPA2	22000	WPA*...	Yes (SSID)	Slow
MD5crypt	500	\$1\$...	Yes	Slow
SHA-256crypt	7400	\$5\$...	Yes	Slow
Apache MD5	1600	\$apr1\$...	Yes	Slow

Further Reading

- [Hashcat Wiki: Example Hashes](#) — complete list of all supported modes with examples
- [Wikipedia: bcrypt](#) — detailed explanation of the Blowfish-based KDF
- [OWASP Password Storage Cheat Sheet](#) — recommendations for secure password hashing
- [hashID on GitHub](#) — source and documentation for the hashid tool