# Hash Identifier Manual — Kali Linux

**Author:** Subhan Ahmed  
**Date:** 2025-08-29

## Table of contents

1. Introduction
2. Who this manual is for
3. Quick concepts: what is a hash and why identify it
4. Tools included in Kali (overview)
5. Installing the tools (one-liners)
6. Step-by-step: Using hash-identifier (interactive)
7. Step-by-step: Using hashid / hashID (modern, scriptable)
8. Parsing files & batch identification
9. Examples (real hashes) and interpreting results
10. Common hash patterns & quick reference table
11. Tips, caveats and common pitfalls
12. Automating identification in scripts and workflows
13. Integrating with cracking tools (workflow, not cracking instructions)
14. Legal / ethical note
15. Appendix: quick commands cheat sheet

# 1. Introduction

This manual explains, in a practical and beginner-friendly way, how to identify the *type* of a cryptographic hash in Kali Linux. Correctly identifying a hash (MD5, SHA1, bcrypt, etc.) is the essential first step before any legitimate analysis — for example, preparing the right cracking strategy in a penetration test or verifying file integrity.

This guide focuses on the two common CLI tools you will find on Kali: **hash-identifier** (classic, interactive) and **hashid / hashID** (modern, scriptable). It contains step-by-step examples, a quick reference table of patterns, and reusable scripts.

# 2. Who this manual is for

* Beginners learning penetration testing or digital forensics.
* Red team / blue team practitioners who need a fast way to fingerprint hashes.
* Students preparing for CTFs or labs where unknown hash strings appear.

# 3. Quick concepts: what is a hash and why identify it

* A *hash* is the fixed-length output produced by a hashing algorithm when you feed it some input. Hashes are **not** encryption — they are (normally) one-way.
* Hashes have distinguishing features (length, character set, known prefixes, encoding like base64) that let us *fingerprint* the algorithm used to produce them.
* Correct fingerprinting tells you which algorithm produced the hash (MD5, SHA256, bcrypt, etc.), which helps choose the right tools and parameters for further analysis.

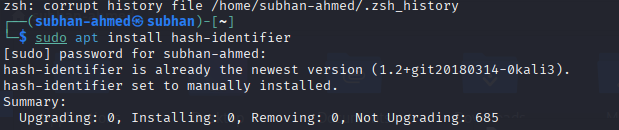
# 4. Tools included in Kali (overview)

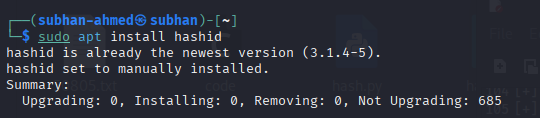
* **hash-identifier** — a classic Python-based interactive script for identifying hash types. Good for quick manual checks.
* **hashid / hashID** — a more modern Python3-based tool that recognizes many more hash patterns and can operate on files/programmatically.
* There are other GUI and online services (e.g., hashes.com or Name-That-Hash) — useful for cross-checking but avoid sending sensitive material to third-party sites.

# 5. Installing the tools (one-liners)

Most recent Kali images already include these tools. If not, you can install:

# Update package lists  
sudo apt update  
  
# Install classic interactive hash-identifier  
sudo apt install hash-identifier

  
  
# Install modern hashid (Python3-based)  
sudo apt install hashid



If you prefer, you can also install hashID from its GitHub source (useful if you want latest releases):

# Example (clone & run locally) — requires git and python3  
git clone https://github.com/psypanda/hashID.git  
cd hashID  
python3 hashid.py <hash>

# 6. Step-by-step: Using hash-identifier (interactive)

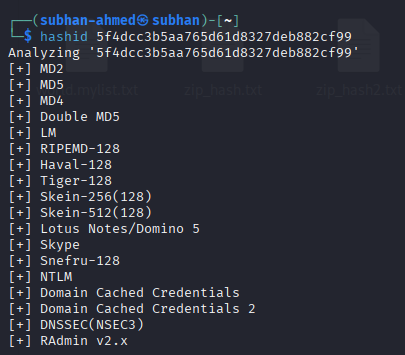
hash-identifier is simple and good for one-off checks.

1. Open a terminal in Kali.
2. Start the tool:

hash-identifier

1. You will see a prompt similar to: Hash: — paste a hash string and press Enter.
2. The tool prints a list of matched algorithms with brief descriptions. Example:

Hash : 5f4dcc3b5aa765d61d8327deb882cf99  
  
Possible Hashs:  
 MD5  
 NTLM  
 MySQL



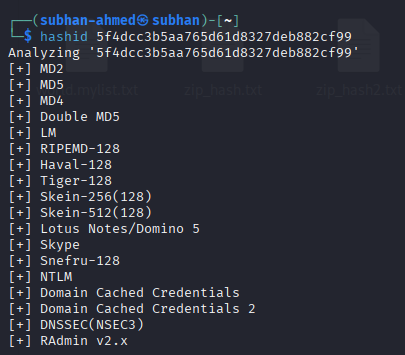
1. If the tool output shows several possibilities, use additional context (length, known application — e.g., WordPress salts, Windows SAM, database dumps) to narrow it down. See the examples section for guidance.

# 7. Step-by-step: Using hashid / hashID (modern, scriptable)

hashid (or hashID depending on the package) is more powerful and supports batch files, regex-based matching, and often recognizes more hash types. It is preferable when automating tests.

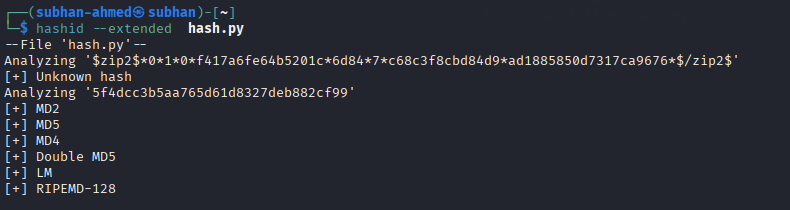
**Single hash identification:**

# using the installed binary  
hashid 5f4dcc3b5aa765d61d8327deb882cf99

  
  
# or running the Python script directly if you cloned the repo  
python3 hashid.py 5f4dcc3b5aa765d61d8327deb882cf99

**Identify hashes inside a file (one per line):**

hashid -f hashes.txt



**Useful flags:** - -f <file> : parse and identify each line as a possible hash.  
- -v : verbose output (if supported).  
- --help : show an up-to-date list of flags for your installed version.

# 8. Parsing files & batch identification

When you have many hashes (for example from a leaked database dump), put one per line in a file and run hashid -f file.txt. The tool will report the likely algorithm for each line.

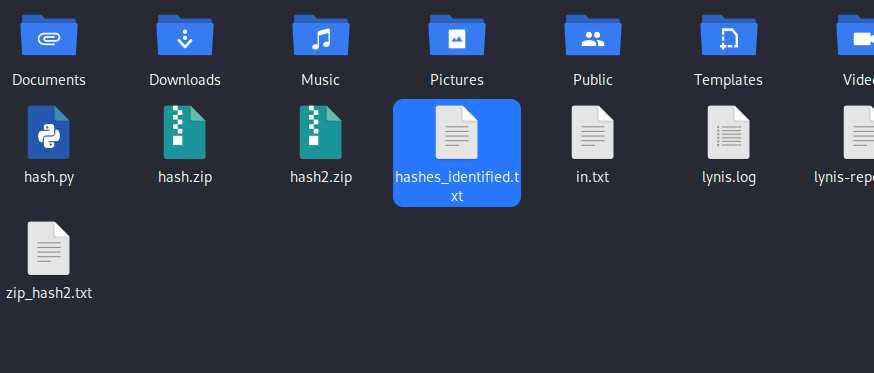
**Example file format (hashes.txt):**

5f4dcc3b5aa765d61d8327deb882cf99  
$2y$12$eImiTXuWVxfM37uY4JANjQ==  
202cb962ac59075b964b07152d234b70

**Run:**

hashid -f hashes.txt > hashes\_identified.txt





This creates an output you can grep, sort, and use to split the file into groups by algorithm for targeted processing.

# 9. Examples (real hashes) and interpreting results

Below are common examples and how the tools report them.

* **MD5 (32 hex chars)**
  + Example: 5f4dcc3b5aa765d61d8327deb882cf99
  + Typical result: MD5
  + Notes: MD5 is unsalted and fast — often used for legacy storage.
* **SHA1 (40 hex chars)**
  + Example: 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
  + Typical result: SHA-1
* **SHA256 (64 hex chars)**
  + Example: 9d5ed678fe57bcca610140957afab571 (shorter example only) — real SHA256 is 64 hex chars.
  + Typical result: SHA-256
* **bcrypt (modular crypt format, starts with , , )**
  + Example: $2y$12$K9h3v7qS8lq4P5Q6dFhQO./xyz...
  + Typical result: bcrypt or Blowfish (bcrypt)
  + Notes: bcrypt hashes include cost parameter and salt — they are easily recognized.
* **NTLM (32 hex, similar to MD5)**
  + Example: 32ed87bdb5fdc5e9cba88547376818d4
  + Typical result: NTLM might show along with MD5 — context helps (Windows SAM/NTLM dumps).

**How to narrow ambiguous results:** - Check the **length** (hex digits count) — it’s one of the simplest clues.  
- Check for **prefixes** like $2y$, $argon2 or {SSHA}.  
- Look at the **character set** (base64 vs hex).  
- Use context: where did the hash come from (a MySQL dump, /etc/shadow, WordPress config, etc.)?

# 10. Common hash patterns & quick reference table

| Algorithm | Typical format / length | Common places seen |
| --- | --- | --- |
| MD5 | 32 hex chars | older apps, database dumps |
| NTLM | 32 hex chars | Windows SAM, NTLM hashes |
| SHA1 | 40 hex chars | older web apps, git (object ids) |
| SHA256 | 64 hex chars | modern apps, file integrity |
| bcrypt | $2[aby]$cost$... (modular crypt format) | modern password stores, frameworks |
| PBKDF2 | Often base64 plus prefix or JSON metadata | enterprise systems, salted iterative hashing |
| MD5(pass) | 32 hex but context shows salt nearby | custom-app salts |

**Tip:** When in doubt, paste the hash into a local tool (hashid) — do not paste sensitive hashes to third-party online services.

# 11. Tips, caveats and common pitfalls

* **False positives:** Hash identification is heuristic-based — multiple algorithms can match the same pattern. Use context always.
* **Base64 vs hex:** Many hashes may be base64-encoded. If length looks short for an expected algorithm, try base64 decoding the string and then re-run identification.
* **Salted & iterative schemes:** Presence of a salt or multiple rounds (PBKDF2, scrypt, Argon2) often alters the format — look for structured prefixes or surrounding metadata.
* **Custom / truncated hashes:** Some apps store truncated or custom-encoded variants — these are harder to identify reliably.

# 13. Integrating with cracking tools (workflow, not cracking instructions)

A safe and legal workflow is: 1. Identify probable hash types with hashid.  
2. Group hashes by type.  
3. For each type, choose the correct cracking tool & parameters (Hashcat mode, John the Ripper format).

**Important:** This manual does not instruct how to crack passwords — only how to identify the hash types. When performing cracking exercises, always have permission.

# 14. Legal / ethical note

Only analyze, identify, or attempt recovery of hashes when you have explicit authorization (e.g., your own systems, sanctioned penetration testing, CTF challenges). Handling or sharing leaked credentials is illegal and unethical.

# 15. Appendix: quick commands cheat sheet

# interactive classic  
hash-identifier  
  
# single hash with hashid  
hashid <hash>  
  
# batch file  
hashid -f hashes.txt  
  
# clone & run latest hashID from GitHub  
git clone https://github.com/psypanda/hashID.git  
python3 hashID/hashid.py <hash>

## Final notes

If you’d like this manual exported to **DOCX** or **PDF**, I can generate and provide a downloadable file. I can also add sample screenshots (terminal runs) and tailor the examples to hashes you provide. Would you like: **DOCX**, **PDF**, **screenshots added**, or any specific hash examples included?

*End of manual.*