

Dictionary & Combinator Attacks

Leveraging real-world wordlists, rule-based mutations, and word combination strategies to crack password hashes efficiently.

Introduction

Password cracking with hashcat is only as effective as the candidate passwords you feed it. Brute-force attacks that enumerate every possible character combination are thorough but painfully slow for anything beyond short passwords. Dictionary attacks take the opposite approach: instead of trying every combination, they try passwords that humans actually use. Since most users pick predictable passwords based on common words, names, dates, and simple patterns, this strategy is devastatingly effective.

This lesson covers two core hashcat attack modes: the dictionary (straight) attack and the combinator attack, along with the rule engine that multiplies their effectiveness.

Dictionary Attack (`-a 0`)

The dictionary attack, also known as a "straight" attack, is hashcat attack mode 0. It reads a wordlist file line by line, hashes each word using the specified algorithm, and compares the result against the target hashes. If a match is found, the password is cracked.

Basic Syntax

```
hashcat -m <hash_type> -a 0 [options] <hash_file> <wordlist>
```

Practical Example

```
# Crack MD5 hashes using the rockyou wordlist
hashcat -m 0 -a 0 -o results.txt hashes.txt rockyou.txt
```

Here `-m 0` specifies MD5, `-a 0` selects dictionary mode, `-o results.txt` writes cracked hash:password pairs to a file, `hashes.txt` contains the target hashes (one per line), and `rockyou.txt` is the wordlist.

How It Works Internally

1. Hashcat loads the target hashes into an optimized lookup table in GPU memory.
2. Words are read from the wordlist in batches and transferred to the GPU.
3. The GPU hashes each candidate and compares against all target hashes simultaneously.
4. Matches are reported and written to the output file.

For fast hash algorithms like MD5 and SHA-1, a modern GPU can process billions of candidates per second. This means even a 14-million-word wordlist like `rockyou.txt` completes in seconds.

Why dictionary attacks work: Analysis of major password breaches consistently shows that the top 1,000 passwords account for a significant percentage of all user passwords. The word "password" and the sequence "123456" appear in nearly every breach, often as the top two most common passwords.

Rule-Based Attacks

A plain dictionary attack only finds exact matches. If a user's password is "Password1" but your wordlist only contains "password", the straight attack misses it. Rule-based attacks solve this by applying transformations to each dictionary word, generating multiple candidates from each entry.

Enabling Rules

```
# Add a rule file with -r
hashcat -m 0 -a 0 -r rules/best64.rule hashes.txt rockyou.txt

# Stack multiple rule files (rules are combined)
hashcat -m 0 -a 0 -r rules/best64.rule -r rules/toggles1.rule hashes.txt rocky
```

When you specify `-r rules/best64.rule`, hashcat applies every rule in that file to every word in the dictionary. If the wordlist has 14 million entries and the rule file has 64 rules, hashcat generates approximately 896 million candidates.

Common Rule Operations

Rule	Function	Example (input: <code>password</code>)
<code>:</code>	Do nothing (pass-through)	password
<code>l</code>	Lowercase all	password
<code>u</code>	Uppercase all	PASSWORD
<code>c</code>	Capitalize first, lowercase rest	Password
<code>t</code>	Toggle case of all characters	PASSWORD (from password)
<code>r</code>	Reverse the word	drowssap
<code>d</code>	Duplicate the word	passwordpassword
<code>\$1</code>	Append character "1"	password1
<code>\$!</code>	Append character "!"	password!
<code>^1</code>	Prepend character "1"	1password
<code>sa@</code>	Replace "a" with "@"	p@ssword
<code>se3</code>	Replace "e" with "3"	password (no "e")

Built-in Rule Files

Hashcat ships with several pre-built rule files in the `rules/` directory:

Rule File	Rules	Description
<code>best64.rule</code>	64	The 64 most statistically effective rules. Best starting point for any attack.

Rule File	Rules	Description
<code>d3ad0ne.rule</code>	~34,000	Large rule set by d3ad0ne. Good second-pass coverage.
<code>dive.rule</code>	~99,000	Very comprehensive. Use when other rule sets fail. Significantly increases runtime.
<code>toggles1.rule</code>	15	Toggles case of individual character positions.
<code>leetspeak.rule</code>	Varies	Common leet-speak substitutions (a->@, e->3, etc.).

Rule stacking: You can specify `-r` multiple times. Hashcat combines them multiplicatively: if rule file A has 64 rules and rule file B has 15 rules, the combination produces $64 \times 15 = 960$ transformations per word. This is powerful but increases the candidate count significantly.

Writing Custom Rules

You can create your own rule file — it is a plain text file with one rule per line. Each rule is a sequence of operation codes. For example:

```
# custom.rule - example custom rules
:                # pass-through (original word)
c                # capitalize first letter
c $1            # capitalize + append "1"
c $!            # capitalize + append "!"
c $1 $2 $3      # capitalize + append "123"
sa@ se3 si1 so0 # leet-speak substitutions
u               # uppercase all
r               # reverse
```

Combinator Attack (`-a 1`)

The combinator attack concatenates every word from a first dictionary with every word from a second dictionary. This targets users who create passwords by joining two common words (e.g., "sunflower", "bluesky", "monkey123").

Syntax

```
hashcat -m 0 -a 1 hashes.txt dict1.txt dict2.txt
```

Candidate Generation

If `dict1.txt` contains:

```
sun  
blue  
dark
```

And `dict2.txt` contains:

```
flower  
sky  
night
```

Hashcat generates these 9 candidates (3 x 3):

```
sunflower    sunsky    sunnight  
blueflower   bluesky   bluenight  
darkflower   darksky   darknight
```

Candidate explosion: The candidate count grows as $N \times M$ where N and M are the sizes of each dictionary. Two dictionaries of 100,000 words each produce 10 billion candidates. Plan your dictionary sizes carefully, or use smaller themed lists.

Combinator with Rules

You can also apply rules to combinator attacks for even more coverage. The `--rule-left` and `--rule-right` flags apply a single rule to the left or right dictionary word respectively:

```
# Capitalize the left word, append "!" to the right word  
hashcat -m 0 -a 1 --rule-left c --rule-right '$!' hashes.txt dict1.txt dict2.t  
# Generates: Sunflower!, Sunsky!, ..., Darknight!
```

Popular Wordlists

The effectiveness of dictionary attacks depends heavily on wordlist quality. Here are the most important sources:

rockyou.txt

The single most widely-used wordlist in password cracking. It contains approximately 14.3 million unique passwords leaked from the RockYou breach in 2009. It ships pre-installed with Kali Linux at `/usr/share/wordlists/rockyou.txt.gz` and is available from many online sources.

Why it is so effective: these are real passwords chosen by real users. The distribution matches how humans actually pick passwords, making it statistically optimal for first-pass cracking.

SecLists

A curated GitHub repository ([danielmiessler/SecLists](https://github.com/danielmiessler/SecLists)) containing hundreds of wordlists organized by category: passwords, usernames, URLs, fuzzing payloads, and more. Key password files include:

- `Passwords/Common-Credentials/10-million-password-list-top-1000000.txt`
- `Passwords/Leaked-Databases/` — various breach-derived lists
- `Passwords/darkweb2017-top10000.txt`

CrackStation

A massive wordlist from crackstation.net containing approximately 1.5 billion entries (15 GB uncompressed). It combines real password leaks with dictionary words and Wikipedia entries. Best reserved for thorough audits due to its size.

Custom Wordlists

For targeted audits, creating a custom wordlist based on the target organization is often more effective than generic lists. Include:

- Company name and variations
- Product names, project code names
- City and location names
- Common industry terms
- Seasonal and date-based patterns (Summer2025, January2026)

When to Use Each Attack Mode

Scenario	Recommended Approach
First pass on unknown hashes	Dictionary (<code>-a 0</code>) with <code>rockyou.txt</code>
Second pass after straight dictionary	Dictionary + rules (<code>-a 0 -r</code> <code>best64.rule</code>)
Suspected two-word passwords	Combinator (<code>-a 1</code>) with themed lists
Deep coverage needed	Dictionary + <code>d3ad0ne.rule</code> or <code>dive.rule</code>
Known password pattern (e.g., "Company" + 4 digits)	Mask or hybrid attack (Lesson 06)

Performance Considerations

Dictionary and combinator attacks are I/O-bound for fast hash types (MD5, SHA-1) since the GPU can hash faster than the wordlist can be read. For slower hash types (bcrypt, scrypt), the GPU becomes the bottleneck and wordlist size matters less.

- **Fast hashes (MD5, SHA-1, NTLM):** Use large wordlists and extensive rules. GPU cycles are cheap relative to candidate generation.
- **Slow hashes (bcrypt, Argon2):** Use smaller, high-quality wordlists. Every candidate costs significant GPU time, so prioritize the most likely passwords.
- **Combinator attacks:** Monitor candidate count. Two 100K-word lists produce 10 billion candidates, which is fine for MD5 but could take days for bcrypt.

Further Reading

- [Hashcat Wiki — Rule-Based Attack](#)
- [Hashcat Wiki — Combinator Attack](#)
- [SecLists on GitHub](#)

- [CrackStation Wordlist](#)