

Wordlist Generation

Build, customize, and optimize wordlists for effective password cracking.

Introduction

A dictionary attack is only as good as its dictionary. The single most impactful thing you can do to improve your password cracking success rate is to use better wordlists. This lesson covers the full lifecycle: finding existing wordlists, generating custom ones, manipulating them with rules, and combining multiple sources into a single optimized attack dictionary.

Understanding wordlist generation is essential because real-world password cracking is rarely about brute-forcing every possible combination. Instead, it is about exploiting the predictable patterns that humans use when choosing passwords. Good wordlists encode these patterns.

Sources for Wordlists

Wordlists come from several categories, each with distinct strengths:

Data Breach Leaks

When a service is compromised and its password database is leaked, security researchers can extract the plaintext passwords (either directly or after cracking weak hashes). These leaked passwords form the most realistic wordlists because they represent actual human password choices.

- **RockYou (2009)** — 14.3 million passwords from a social gaming platform. Stored in plaintext. This is the most widely used wordlist in the security community and ships with Kali Linux at `/usr/share/wordlists/rockyou.txt.gz`.
- **LinkedIn (2012)** — 117 million SHA-1 hashes. After cracking, researchers recovered the vast majority as plaintext. Revealed strong bias toward professional-sounding passwords.

- **Ashley Madison (2015)** — 36 million bcrypt hashes. Because bcrypt is slow to crack, only a fraction were recovered, but those revealed common patterns like "password1" and date-based passwords.
- **Collection #1-5 (2019)** — Compilations of billions of credentials from numerous breaches. Useful for building frequency-sorted mega-lists.

Legal notice: Possessing breach data may be illegal in some jurisdictions. Always ensure you are operating within the law and using wordlists only for authorized security testing.

Curated Common Password Lists

Several projects maintain high-quality, deduplicated wordlists:

- **SecLists** — A collection maintained by Daniel Miessler, containing passwords, usernames, URLs, fuzzing payloads, and more. Available on GitHub.
- **Probable Wordlists** — Sorted by frequency of occurrence across multiple breaches. The "Top" lists (top 1000, top 10000, etc.) are excellent for quick attacks.
- **CrackStation** — Offers a 15GB wordlist containing every word from Wikipedia, Project Gutenberg, and multiple breach lists combined.

Personalized Wordlists

When targeting a specific individual (in an authorized engagement), personalized wordlists are highly effective. Tools like CUPP (Common User Passwords Profiler) generate candidates based on personal information: names, birthdates, pet names, favorite sports teams, and common permutations of these.

```
# CUPP interactive mode — asks questions about the target
cupp -i

# Example output generates: John1990, john1990!, John90, j0hn, etc.
```

Multilingual Wordlists

Password patterns vary significantly across languages and cultures. A Spanish-speaking user might use "contrasena123" while a German user might choose "passwort2024". When targeting organizations with international employees, include

wordlists in relevant languages. Projects like the Universal Password List and language-specific breach data address this need.

Crunch — Systematic Wordlist Generation

Crunch is a command-line tool that generates wordlists based on specified criteria. Unlike breach-based lists, Crunch creates every possible combination within your parameters — making it exhaustive but potentially enormous.

Installation

```
# Debian/Ubuntu/Kali
sudo apt-get install crunch

# macOS (Homebrew)
brew install crunch
```

Basic Usage

The core syntax is `crunch <min-len> <max-len> [charset] [options]`:

```
# All combinations from 8 to 10 characters using the default charset
crunch 8 10 -o custom_dict.txt

# Only lowercase letters and digits
crunch 8 10 abcdefghijklmnopqrstuvwxyz0123456789 -o custom_dict2.txt

# Only digits, 4 to 6 characters (PINs)
crunch 4 6 0123456789 -o pins.txt
```

Pattern-Based Generation

The `-t` flag lets you define patterns using placeholder characters:

Placeholder	Matches
<code>@</code>	Lowercase letters (a-z)
<code>,</code>	Uppercase letters (A-Z)
<code>%</code>	Digits (0-9)



Special characters

```
# Pattern: 4 letters followed by 4 digits (e.g., "pass1234")
crunch 8 8 -t @@@@%-% -o pattern_dict.txt

# Pattern: Uppercase first, lowercase rest, two digits, one symbol
crunch 8 8 -t ,@@@@@%^ -o complex_pattern.txt
```

Size Considerations

Crunch can generate astronomically large files. Before running a command, estimate the output size:

- 8-character lowercase (26 chars): $26^8 = 208$ billion combinations = ~1.8 TB
- 4-digit PIN: $10^4 = 10,000$ combinations = ~50 KB
- 6-character lowercase+digits: $36^6 = 2.17$ billion = ~15 GB

Tip: Use Crunch's `-b` flag to split output into multiple files of a specified size:

```
crunch 8 8 -t @@@@%-% -b 1gb -o START
```

CeWL — Website-Based Wordlist Generation

CeWL (Custom Word List generator) spiders a website and extracts unique words to build a wordlist. This is particularly effective when targeting an organization, because employees frequently use company jargon, product names, project codenames, and other domain-specific terms as passwords or password components.

Installation

```
# Ruby gem
gem install cewl

# Kali Linux (pre-installed)
which cewl
```

Basic Usage

```
# Spider a website and save unique words
cewl -w web_dict.txt http://example.com

# Set minimum word length to 6 and spider depth to 3 levels
cewl -m 6 -d 3 -w web_dict.txt http://example.com

# Include email addresses found on the site
cewl -e --email_file emails.txt -w web_dict.txt http://example.com

# Extract words from document metadata (PDF, DOCX, etc.)
cewl --meta -w web_dict.txt http://example.com

# Verbose output to see progress
cewl -v -w web_dict.txt http://example.com
```

Why CeWL Works

Consider a company called "NovaTech Solutions" that makes a product called "StarBridge". CeWL would extract words like "NovaTech", "StarBridge", "Solutions", "innovation", and other terms from the website. Combined with rules, these become candidates like "NovaTech2024!", "starbridge123", "Solutions!", and so on — exactly the kind of passwords employees tend to choose.

Hashcat Rules for Wordlist Manipulation

Hashcat rules are the most powerful way to multiply the effectiveness of any wordlist. Rather than storing every variation on disk, rules generate candidates on the fly during an attack. A wordlist of 10 million words combined with 64 rules produces 640 million candidates — with zero additional disk space.

Using Rule Files

```
# Apply the best64 rule set
hashcat -m 0 hashes.txt wordlist.txt -r /usr/share/hashcat/rules/best64.rule

# Stack multiple rule files (each word gets rules from BOTH files applied)
hashcat -m 0 hashes.txt wordlist.txt -r rules1.rule -r rules2.rule

# Generate rule candidates to stdout (for testing)
hashcat --stdout wordlist.txt -r best64.rule | head -20
```

Built-in Rule Files

Rule File	Rule Count	Best For
<code>best64.rule</code>	64	Fast initial attacks. The 64 most statistically effective rules.
<code>d3ad0ne.rule</code>	~34,000	Thorough attacks when you have time. Aggressive mutations.
<code>dive.rule</code>	~99,000	Maximum coverage. Use overnight or when other methods fail.
<code>toggles1.rule</code> - <code>toggles5.rule</code>	varies	Case toggling at specific positions. Good for mixed-case policies.
<code>leetspeak.rule</code>	varies	Common character substitutions (a->@, e->3, etc.).

Rule Syntax Reference

Each line in a rule file contains one or more operations. Multiple operations on the same line are applied sequentially to each word:

Rule	Description	Input	Output
<code>:</code>	Do nothing (passthrough)	password	password
<code>l</code>	Lowercase all characters	PassWord	password
<code>u</code>	Uppercase all characters	password	PASSWORD
<code>c</code>	Capitalize first, lowercase rest	password	Password
<code>C</code>	Lowercase first, uppercase rest	password	pASSWORD
<code>t</code>	Toggle case of all characters	PassWord	pASSwORD
<code>TN</code>	Toggle case at position N	password	Password (T0)

<code>r</code>	Reverse the word	password	drowssap
<code>d</code>	Duplicate the word	pass	passpass
<code>\$x</code>	Append character X	password	password1 (\$1)
<code>^x</code>	Prepend character X	password	1password (^1)
<code>sXY</code>	Replace character X with Y	password	p@ssword (sa@)
<code>[</code>	Delete first character	password	assword
<code>]</code>	Delete last character	password	passwor
<code>'N</code>	Truncate at position N	password	pass ('4)

Writing Custom Rules

You can combine operations on a single line. They are applied left to right:

```
# Rule: capitalize first letter, append "123"
# c $1 $2 $3
# "password" → "Password" → "Password1" → "Password12" → "Password123"

# Rule: replace 'a' with '@', append '!'
# sa@ $!
# "password" → "p@ssword" → "p@ssword!"

# Rule: uppercase all, prepend "!" and append "!"
# u ^! $!
# "password" → "PASSWORD" → "!PASSWORD" → "!PASSWORD!"
```

Save your custom rules to a `.rule` file (one rule per line) and use it with the `-r` flag.

Combining Wordlists

In practice, the most effective approach uses multiple wordlists from different sources. Combining them ensures broad coverage while deduplication keeps the file manageable.

```
# Basic combination with deduplication
cat dict1.txt dict2.txt | sort -u > combined.txt

# Combine multiple sources
cat rockyou.txt web_dict.txt custom_dict.txt personal.txt | sort -u > mega_dict.txt

# Check the result
wc -l mega_dict.txt
head -20 mega_dict.txt
```

Optimization Strategies

- **Sort by probability.** Place the most likely passwords first. Breach-based frequency lists naturally have this property. This matters because Hashcat processes words sequentially — putting common passwords first means faster cracks.
- **Remove unlikely candidates.** If you know the target enforces a minimum length of 8, remove all words shorter than 8 characters: `awk 'length >= 8' combined.txt > filtered.txt`
- **Layer your attack.** Start with a small, high-probability wordlist + best64 rules. Then escalate to larger wordlists and more aggressive rules. This gives you quick wins first.
- **Use Hashcat's combinator attack.** Instead of just concatenating files, Hashcat's combinator mode (`-a 1`) joins every word from wordlist 1 with every word from wordlist 2: "pass" + "word" = "password".

Attack Strategy: Putting It All Together

A methodical approach to wordlist-based cracking follows this escalation pattern:

1. **Quick win:** Top 1000 passwords + best64.rule
2. **Standard:** RockYou + best64.rule
3. **Targeted:** CeWL output from target website + d3ad0ne.rule
4. **Personalized:** CUPP-generated list + best64.rule
5. **Combined:** All sources merged + dive.rule
6. **Exhaustive:** CrackStation mega-list + dive.rule (may take days)

Further Reading

- [Hashcat Wiki: Rule-Based Attack](#)
- [SecLists on GitHub](#)
- [CeWL on GitHub](#)
- [CUPP on GitHub](#)
- [CrackStation Wordlist](#)