

Theory: Results Analysis & Post-Processing

Mastering output formats, session management, and Python-based password analysis for professional security audits.

Introduction

Recovering passwords with hashcat is only the first step in a security audit. The real value comes from analyzing the results: identifying password patterns, measuring policy compliance, and producing actionable reports that help organizations strengthen their defenses. This lesson covers the complete post-cracking workflow, from configuring hashcat's output to building automated analysis pipelines with Python.

Output File Formats

Hashcat's `--outfile-format` flag determines what information is saved to the output file. The value is a numeric code from 1 to 6:

Format	Contents	Use Case
1	Hash only	Feeding cracked hashes into other tools
2	Plain password only	Building wordlists from cracked passwords
3	Hash:Password	Default format; general-purpose auditing
4	Login:Password	Credential reports (requires hash file with usernames)
5	Login:Password:Hash	Comprehensive audit reports with full traceability
6	Login:Password:Hash:Salt	Full data export for salted hash types

The default format (3) is suitable for most scenarios. Format 2 is useful when you want to extract just the plaintext passwords to build custom wordlists. Formats 4-6 require that the input hash file includes usernames in the format `username:hash` and that you pass the `--username` flag.

Saving Results

Use the `-o` flag to write results to a file:

```
# Default format (hash:password)
hashcat -m 0 hashes.txt wordlist.txt -o results.txt

# Login:Password format (requires --username)
hashcat -m 0 hashes.txt wordlist.txt -o results.txt --outfile-format 4 --username

# Append to an existing file instead of overwriting
hashcat -m 0 hashes.txt wordlist.txt -o results.txt --outfile-format 3
```

Without `-o`, hashcat still saves cracked passwords in the potfile (`~/.local/share/hashcat/hashcat.potfile` on Linux). The potfile is always in hash:password format and accumulates results across all sessions.

Potfile behavior: Hashcat automatically checks the potfile before starting an attack. Any hash that has already been cracked is skipped. To force re-cracking, use `--potfile-disable` or delete the potfile.

Session Management

Long-running attacks often need to be paused and resumed. Hashcat's session system handles this transparently.

Starting a Named Session

```
# Name your session so you can resume it later
hashcat -m 0 hashes.txt wordlist.txt --session audit_2024

# Session files are saved in ~/.local/share/hashcat/sessions/
```

Restoring a Session

```
# Resume from where it stopped
hashcat --session audit_2024 --restore
```

Session state includes the current position in the wordlist or keyspace, the attack mode, all options, and the potfile path. If the system loses power during a long attack, you lose only the progress since the last automatic checkpoint (hashcat saves checkpoints periodically).

Session Workflow

1. **Start:** `hashcat --session name [options]`
2. **Pause:** Press `p` during execution, or `Ctrl+C` (graceful quit with checkpoint save)
3. **Resume:** `hashcat --session name --restore`
4. **Check status:** Press `s` during execution to see progress

Important: When using `--restore`, do not pass any other attack options. Hashcat loads all parameters from the saved session file. Passing conflicting options will cause an error.

Inspecting Results: --show and --left

After running one or more attacks, you need to know what was cracked and what remains. Hashcat provides two complementary commands:

--show: View Cracked Hashes

```
# Show all cracked hashes from the potfile
hashcat -m 0 --show hashes.txt

# Output (format: hash:password)
# 5f4dcc3b5aa765d61d8327deb882cf99:password
# e10adc3949ba59abbe56e057f20f883e:123456
# d8578edf8458ce06fbc5bb76a58c5ca4:qwerty
```

The `--show` command reads the potfile and matches entries against the provided hash file. It does not run any cracking; it simply reports what has already been found.

--left: View Uncracked Hashes

```
# Show hashes that have NOT been cracked
hashcat -m 0 --left hashes.txt

# Redirect to a new file for further attacks
hashcat -m 0 --left hashes.txt > remaining.txt
```

This is essential for iterative attack strategies: run a dictionary attack, check what is left, then run a mask attack on the remaining hashes, and so on. Each iteration targets a smaller set, making more expensive attacks feasible.

Post-Processing with Python

Hashcat output files are plain text, making them easy to process with Python. The two essential libraries are `pandas` for data manipulation and `matplotlib` for visualization.

Reading and Cleaning Results

```
import pandas as pd

# Read hash:password format (format 3)
df = pd.read_csv('results.txt', sep=':', header=None,
                 names=['hash', 'password'], engine='python')

# Remove duplicates (same password may appear for different hashes)
print(f"Total entries: {len(df)}")
df.drop_duplicates(subset='password', inplace=True)
print(f"Unique passwords: {len(df)}")

# Basic stats
print(f"Shortest password: {df['password'].str.len().min()} chars")
print(f"Longest password: {df['password'].str.len().max()} chars")
print(f"Average length: {df['password'].str.len().mean():.1f} chars")
```

Handling colons in passwords: If passwords contain colons, the simple `sep=':'` approach will split incorrectly. Use `str.split(':', n=1)` to split only on the first colon, or use format 2 (plain only) as a workaround.

Password Frequency Analysis

```
# Top 20 most common passwords
top_passwords = df['password'].value_counts().head(20)
print("Top 20 Passwords:")
print(top_passwords.to_string())

# What percentage of users share the top password?
total_users = len(df)
top_count = top_passwords.iloc[0]
print(f"\nMost common password used by {top_count}/{total_users} "
      f"({top_count/total_users*100:.1f}%) of users")
```

Length Distribution

```
import matplotlib.pyplot as plt

df['length'] = df['password'].str.len()

fig, ax = plt.subplots(figsize=(10, 5))
df['length'].hist(bins=range(1, df['length'].max() + 2),
                  color='#1f6feb', edgecolor='#0d1117', ax=ax)
ax.set_xlabel('Password Length')
ax.set_ylabel('Count')
ax.set_title('Password Length Distribution')
ax.axvline(x=8, color='#f85149', linestyle='--', label='8-char minimum')
ax.legend()
plt.tight_layout()
plt.savefig('length_distribution.png', dpi=150)
plt.show()
```

Complexity Analysis

```
import re

def analyze_complexity(password):
    """Classify password complexity based on character types."""
    checks = {
        'has_lower': bool(re.search(r'[a-z]', password)),
        'has_upper': bool(re.search(r'[A-Z]', password)),
        'has_digit': bool(re.search(r'[0-9]', password)),
        'has_special': bool(re.search(r'[^\A-Za-z0-9]', password)),
    }
```

```
score = sum(checks.values())
if score <= 1:
    return 'Weak'
elif score == 2:
    return 'Fair'
elif score == 3:
    return 'Good'
return 'Strong'

df['complexity'] = df['password'].apply(analyze_complexity)

# Summary
print("Complexity Distribution:")
print(df['complexity'].value_counts().to_string())

# Visualize
```

Creating Reports

A professional password audit report should include the following sections:

1. **Executive Summary:** Crack rate, most critical findings, key recommendations
2. **Methodology:** Hash types, attack modes used, wordlists, rules, hardware
3. **Results Overview:** Total hashes, cracked count, crack percentage by hash type
4. **Password Analysis:**
 - Top 10 most common passwords
 - Length distribution chart
 - Complexity distribution chart
 - Percentage meeting policy requirements (e.g., 8+ chars, mixed case, digits, special)
5. **Recommendations:** Minimum length increase, mandatory complexity, MFA adoption, password manager deployment

Handling sensitive data: Cracked passwords are highly sensitive. Encrypt output files, limit access, and ensure the report does not include raw passwords for individual users. Aggregate statistics are sufficient for recommendations.

Automated Report Script

```
def generate_report(results_file, total_hashes):
    """Generate a summary report from hashcat results."""
    df = pd.read_csv(results_file, sep=':', header=None,
                     names=['hash', 'password'], engine='python')

    cracked = len(df)
    rate = cracked / total_hashes * 100

    report = []
    report.append("=" * 60)
    report.append("PASSWORD AUDIT REPORT")
    report.append("=" * 60)
    report.append(f"\nTotal hashes:      {total_hashes}")
    report.append(f"Cracked:          {cracked} ({rate:.1f}%)")
    report.append(f"Remaining:       {total_hashes - cracked}")
    report.append(f"\nAverage length:  {df['password'].str.len().mean():.1f}")
    report.append(f"Under 8 chars:   {(df['password'].str.len() < 8).sum()}")
    report.append(f"\nTop 5 passwords:")
    for pw, count in df['password'].value_counts().head(5).items():
        report.append(f"    {pw:20s}    ({count} users)")

    return '\n'.join(report)
```

Further Reading

- [Hashcat Wiki — Output Formats and Session Management](#)
- [pandas Documentation — Data Analysis](#)
- [matplotlib Tutorials — Visualization](#)
- [SANS White Papers — Password Auditing Best Practices](#)