

Lesson 01: Introduction to Hashcat

Introduction to Hashcat

A comprehensive introduction to the world's most popular open-source password recovery tool — from basic concepts to your first hash crack.

Learning Objectives

- Understand **what Hashcat is** and why it exists
 - Learn the **history and evolution** of the tool
 - Explore the **main capabilities** and supported attack types
 - Install Hashcat and run **basic commands**
 - Perform a **test dictionary attack** on a known MD5 hash
-

What Is Hashcat?

Hashcat is an **open-source password recovery tool** that uses modern hash function recognition methods to recover lost passwords or verify the security of existing ones.

Key Characteristics

- Free and open-source (MIT license)
- Extremely fast — GPU-accelerated
- Supports 350+ hash types
- Cross-platform (Windows, Linux, macOS)

Who Uses It?

- **Security researchers** — studying password security
- **Penetration testers** — auditing organizations
- **System administrators** — verifying password policies
- **Forensic analysts** — recovering evidence

Notes: Hashcat's primary goal is to recover lost passwords or to verify the security of existing password hashes. It should only be used on systems you own or have written authorization to test.

History and Development

Hashcat was created to improve and accelerate the password recovery process. Over the years, it has grown from a simple CPU-based cracker into the most powerful hash recovery tool available.

Key Milestones

- **Early versions** — CPU-only password recovery tool with limited hash support
- **oclHashcat era** — Introduction of GPU acceleration using OpenCL, dramatically increasing speed
- **Hashcat 3.0+ (2016)** — CPU and GPU versions merged into a single unified tool
- **Ongoing development** — Constant updates adding new hash types and optimizing recognition processes

Open-Source Milestone

In 2015, Hashcat was made fully open-source, allowing the community to contribute hash algorithms, performance optimizations, and bug fixes. This accelerated development

significantly.

Main Capabilities

Hashcat stands out due to its extensive feature set and broad compatibility across platforms and hardware.

Platform Support

- Windows
- Linux
- macOS

Hardware Acceleration

- CPU (all platforms)
- NVIDIA GPU (CUDA)
- AMD GPU (OpenCL)
- Apple Metal (macOS)

Core Features

- **350+ hash types** — MD5, SHA-1, SHA-256, bcrypt, NTLM, WPA, and many more
- **Multiple attack modes** — dictionary, mask, combinator, rule-based, and hybrid attacks
- **Session management** — pause, resume, and restore cracking sessions

- **Distributed cracking** — split workloads across multiple machines
-

Attack Types Overview

Hashcat supports several attack modes, each suited for different scenarios.

Mode	Flag	Description
Dictionary Attack	<code>-a 0</code>	Tries every word from a wordlist file
Combinator Attack	<code>-a 1</code>	Combines words from two dictionaries
Brute-Force / Mask	<code>-a 3</code>	Tries all combinations matching a pattern (mask)
Hybrid (Dict + Mask)	<code>-a 6</code>	Appends mask patterns to dictionary words
Hybrid (Mask + Dict)	<code>-a 7</code>	Prepends mask patterns to dictionary words

Notes: The dictionary attack (-a 0) is the most commonly used starting point. Rule-based attacks add transformations (capitalize, append digits, etc.) to dictionary words, making them extremely powerful.

Quick Demo: Your First Crack

Let's walk through cracking a known MD5 hash using a dictionary attack.

Step 1: Create a hash file

```
# The MD5 hash of "password"
echo "5f4dcc3b5aa765d61d8327deb882cf99" > hashes.txt
```

Step 2: Run the attack

```
hashcat -m 0 -a 0 -o cracked.txt hashes.txt example.dict
```

Step 3: Check results

```
cat cracked.txt
# Output: 5f4dcc3b5aa765d61d8327deb882cf99:password
```

Command Breakdown

- `-m 0` — Hash type: MD5
 - `-a 0` — Attack mode: Dictionary
 - `-o cracked.txt` — Output file for recovered passwords
 - `hashes.txt` — File containing target hashes
 - `example.dict` — Wordlist / dictionary file
-

Essential Commands

Before diving deeper, familiarize yourself with these fundamental Hashcat commands.

System Information

```
# Show detected compute devices (GPUs, CPUs)
hashcat -I

# Display full help with all options
hashcat --help
```

Common Flags Reference

Flag	Purpose	Example
<code>-m</code>	Hash type	<code>-m 0</code> (MD5)
<code>-a</code>	Attack mode	<code>-a 0</code> (dictionary)
<code>-o</code>	Output file	<code>-o results.txt</code>

Flag	Purpose	Example
<code>--show</code>	Show already cracked	<code>hashcat -m 0 hash.txt --show</code>
<code>-w</code>	Workload profile	<code>-w 3</code> (high performance)

Lab & Homework

Lab Exercises

1. **Install Hashcat** on your operating system and verify the installation
2. **Run basic commands** — `hashcat -I` and `hashcat --help` to explore options
3. **Crack an MD5 hash** — perform a dictionary attack on a known hash and verify the output

Homework Tasks

1. Research and list 5 hash types supported by Hashcat with their `-m` codes
2. Explain the difference between dictionary and mask attacks in your own words
3. Create and crack your own MD5 hash using a custom wordlist

Reference

Official Hashcat FAQ: [hashcat.net/wiki — FAQ](https://hashcat.net/wiki/FAQ)

Theory

Reading time: ~15 minutes · Prerequisites: Basic command-line familiarity

1. What Is Hashcat?

Hashcat is an open-source password recovery tool that uses modern hash function recognition methods. It is one of the most popular and powerful tools in the cybersecurity field, widely used by security researchers, penetration testers, and system administrators.

The main goal of Hashcat is twofold: to **recover lost passwords** and to **verify the security of existing ones**. By attempting to reverse password hashes back to their plaintext form, security professionals can assess whether an organization's password policies are strong enough to withstand real-world attacks.

How Password Hashing Works

When a user creates a password, the system does not store the password itself. Instead, it runs the password through a **hash function** — a one-way mathematical operation that produces a fixed-length string of characters (the "hash"). For example, the MD5 hash of the word "password" is:

```
5f4dcc3b5aa765d61d8327deb882cf99
```

Hash functions are designed to be **irreversible** — you cannot mathematically derive the original password from the hash. However, you *can* try hashing many possible passwords and comparing the results. This is exactly what Hashcat does, but at extraordinary speed.

Key Insight

Hashcat does not "decrypt" passwords. Hashes are not encrypted data — they are one-way transformations. Hashcat works by generating candidate passwords,

hashing them, and comparing the result against target hashes. When a match is found, the original password is recovered.

2. History and Development of Hashcat

Hashcat was originally created to improve the password recovery process. In its early days, password crackers were slow, limited in the hash types they supported, and ran exclusively on CPUs.

The CPU Era

The earliest versions of Hashcat were CPU-based tools. While functional, CPUs are not well suited for the massively parallel computation that password cracking requires. A typical CPU has 4-16 cores, which limits how many hash computations can be performed simultaneously.

The GPU Revolution

The introduction of GPU acceleration was a turning point. Modern GPUs contain thousands of cores designed for parallel processing. What might take a CPU hours to compute, a GPU can accomplish in minutes or seconds. The GPU-accelerated version, originally called **oclHashcat**, used OpenCL to leverage graphics cards for hash computation.

Unification and Open Source

In 2015, Hashcat became fully open-source. In 2016, with version 3.0, the CPU and GPU versions were merged into a single unified tool. This simplified usage and allowed Hashcat to automatically detect and use the best available hardware.

Development continues actively, with constant updates adding new hash types and optimizing recognition processes. The community contributes new algorithms, performance improvements, and platform support on an ongoing basis.

3. Main Capabilities and Applications

Supported Attack Types

Hashcat supports multiple attack strategies, each effective in different scenarios:

Attack Type	Flag	How It Works	Best For
Dictionary Attack	<code>-a 0</code>	Hashes every word from a wordlist and compares against target	Common passwords, leaked password lists
Combinator Attack	<code>-a 1</code>	Concatenates words from two different dictionaries	Passwords made of two common words
Mask (Brute-Force) Attack	<code>-a 3</code>	Tries all combinations matching a specified pattern	Known password structure (e.g., 8 lowercase letters)
Hybrid Attack (Dict + Mask)	<code>-a 6</code>	Appends mask-generated suffixes to dictionary words	Passwords like "summer2024" or "admin123"
Hybrid Attack (Mask + Dict)	<code>-a 7</code>	Prepends mask-generated prefixes to dictionary words	Passwords like "123admin" or "!password"

Rule-Based Attacks

In addition to the core attack modes, Hashcat supports **rules** — transformation functions applied to dictionary words before hashing. Rules can capitalize letters, append numbers, substitute characters (e.g., "a" to "@"), reverse strings, and much more. A single dictionary of 100,000 words combined with a rule file containing 1,000 rules effectively tests 100 million password candidates.

Cross-Platform Support

Hashcat runs on all major operating systems:

- **Windows** — Pre-compiled binaries available
- **Linux** — Available through package managers or compiled from source
- **macOS** — Supported with Metal backend for Apple Silicon GPUs

Hash Type Support

Hashcat supports over 350 hash types, organized by category. Some of the most commonly encountered types include:

Hash Type	Mode (-m)	Common Use
MD5	0	Legacy web applications, file checksums
SHA-1	100	Legacy systems, some version control
SHA-256	1400	Modern applications, certificates
NTLM	1000	Windows authentication
bcrypt	3200	Modern web applications
WPA/WPA2	22000	Wi-Fi network security

4. Practical Applications

Penetration Testing

During authorized security assessments, penetration testers often extract password hashes from compromised systems. Hashcat is used to recover these passwords, demonstrating the real-world risk of weak password policies to the client.

Password Policy Auditing

Organizations can proactively test their own password databases against common wordlists and attack patterns. If a significant percentage of employee passwords can be cracked within minutes, the password policy needs strengthening.

Forensic Analysis

In digital forensic investigations, encrypted files or password-protected systems may need to be accessed. Hashcat can assist in recovering passwords when authorized by legal process.

Legal and Ethical Warning

Using Hashcat against systems or data you do not own or have explicit written authorization to test is **illegal** in most jurisdictions. Always ensure you have proper authorization before performing any password recovery or security testing activities. This course is for educational purposes and authorized security testing only.

5. Key Concepts Summary

Concept	What It Means	Why It Matters
Hash function	One-way transformation of data to fixed-length string	Passwords are stored as hashes, not plaintext
Dictionary attack	Testing passwords from a pre-built wordlist	Most effective against common/leaked passwords
Mask attack	Brute-forcing all combinations matching a pattern	Covers the entire keyspace for known structures
GPU acceleration	Using graphics card for parallel hash computation	Orders of magnitude faster than CPU-only cracking
Hash type mode (-m)	Numeric identifier for the target hash algorithm	Must match the actual hash format for correct cracking

Further Reading

- [Hashcat FAQ](#) — Official frequently asked questions covering installation, usage, and troubleshooting.
- [Hashcat Wiki](#) — Comprehensive documentation covering all hash types, attack modes, and configuration options.
- [Hashcat Downloads](#) — Official download page for the latest release.
- [Example Hashes](#) — Reference list of example hashes for every supported hash type, useful for testing.

Next lesson: **Lesson 02 — Installation & Setup** — detailed installation instructions for all platforms and hardware configuration.

Lab Guide

Time estimate: 30–45 minutes · Goals: Install Hashcat, learn basic commands and parameters, run a test attack on a known hash

Lab Goals

- Learn basic Hashcat functions and interface
- Install Hashcat on your operating system
- Learn basic commands and parameters
- Run a test dictionary attack on a known MD5 hash

Task 1: Install Hashcat

Installation Instructions

Choose the installation method for your operating system:

macOS

```
# Using Homebrew
brew install hashcat

# Verify installation
hashcat --version
```

Linux (Debian/Ubuntu)

```
# Using apt
sudo apt update
sudo apt install hashcat

# Verify installation
hashcat --version
```

Linux (Arch)

```
# Using pacman
sudo pacman -S hashcat

# Verify installation
hashcat --version
```

Windows

1. Download the latest release from <https://hashcat.net/hashcat/>
2. Extract the archive to a folder (e.g., C:\hashcat)
3. Open Command Prompt or PowerShell
4. Navigate to the folder: `cd C:\hashcat`
5. Run: `hashcat.exe --version`

✓ **Check:** Running `hashcat --version` prints a version number (e.g., `v6.2.6`).

Task 2: Explore Basic Commands

Step 2.1 — Check Available Devices

The `-I` flag displays information about compute devices detected by Hashcat (CPUs, GPUs, etc.):

```
hashcat -I
```

You should see output listing your available devices. Example:

```
hashcat (v6.2.6) starting in backend information mode
```

```
OpenCL Info:
```

```
=====
```

```
OpenCL Platform ID #1
```

```
Vendor...: Apple
```

```
Name.....: Apple
```

```
Version..: OpenCL 1.2
```

```
Backend Device ID #1
Type.....: GPU
Vendor.ID.....: 32
Vendor.....: Apple
Name.....: Apple M1
...
```

✓ **Check:** At least one compute device is listed. If no GPU is detected, Hashcat will fall back to CPU mode.

Step 2.2 — View Help

The `--help` flag displays the full list of options, attack modes, and hash types:

```
hashcat --help
```

This produces extensive output. Take note of these key sections:

- **Attack Modes** — the different `-a` options (0 = dictionary, 3 = mask, etc.)
- **Hash Types** — the full list of `-m` values and their corresponding hash algorithms
- **Output Options** — flags for controlling where and how results are saved

✓ **Check:** You can identify the `-m` value for MD5 (it is `0`).

Task 3: Crack a Known MD5 Hash

Goal

Perform a dictionary attack on a known MD5 hash to verify that your Hashcat installation works correctly.

Step 3.1 — Create the Hash File

Create a file called `hashes.txt` containing the MD5 hash of the word "password":

```
echo "5f4dcc3b5aa765d61d8327deb882cf99" > hashes.txt
```

About This Hash

The string `5f4dcc3b5aa765d61d8327deb882cf99` is the MD5 hash of the plaintext word "password". This is one of the most commonly found hashes in leaked databases, because "password" is one of the most commonly used passwords worldwide.

Step 3.2 — Create a Simple Wordlist

Create a small dictionary file called `example.dict` with a few common passwords:

```
cat <<EOF > example.dict
123456
password
12345678
qwerty
abc123
monkey
master
dragon
letmein
football
EOF
```

Step 3.3 — Run the Dictionary Attack

Execute the following command to crack the hash:

```
hashcat -m 0 -a 0 -o cracked.txt hashes.txt example.dict
```

Let's break down each part of this command:

Part	Meaning
<code>hashcat</code>	The Hashcat executable
<code>-m 0</code>	Hash type: MD5 (mode 0)
<code>-a 0</code>	Attack mode: Dictionary (mode 0)

Part	Meaning
<code>-o cracked.txt</code>	Write recovered passwords to this file
<code>hashes.txt</code>	Input file containing target hashes
<code>example.dict</code>	Wordlist file to use for the attack

Step 3.4 — Verify the Results

Check the output file to see the cracked password:

```
cat cracked.txt
```

Expected output:

```
5f4dcc3b5aa765d61d8327deb882cf99:password
```

The format is `hash:plaintext`, showing that the hash `5f4dcc3b...` corresponds to the password "password".

✓ **Check:** The file `cracked.txt` contains `5f4dcc3b5aa765d61d8327deb882cf99:password`.

Troubleshooting

- **"Hash not found" or "Already cracked"** — Hashcat caches results in a potfile. If you've already cracked this hash, use `--potfile-disable` to ignore the cache, or use `hashcat -m 0 hashes.txt --show` to display previously cracked hashes.
- **"No devices found"** — Ensure you have appropriate GPU drivers installed. On macOS, Metal is used automatically. On Linux/Windows, you need OpenCL or CUDA drivers.
- **"Token length exception"** — The hash format doesn't match the specified `-m` mode. Double-check that your hash file contains only the hash with no extra spaces or characters.

Finishing Up

Once you have completed all three tasks, verify the following:

Self-Check

- ✓ Hashcat is installed and `hashcat --version` works
- ✓ `hashcat -I` lists at least one compute device
- ✓ You can locate MD5 in the `hashcat --help` output and know its mode number
- ✓ You successfully cracked the MD5 hash and `cracked.txt` contains the correct result
- ✓ You can explain what `-m 0`, `-a 0`, and `-o` mean

Homework

Complete these tasks before Lesson 02. They reinforce the concepts from this lesson and prepare you for the next one.

Task 1: Research Hash Types

Goal: Familiarize yourself with the variety of hash types Hashcat supports.

Instructions

Using the `hashcat --help` output or the [Hashcat Example Hashes](#) page, research and document 5 different hash types.

For each hash type, record:

1. The hash type name (e.g., "MD5")
2. The `-m` mode number (e.g., `0`)
3. A brief description of where this hash type is commonly used
4. An example hash from the Hashcat example hashes page

Deliverable

A table in this format:

Hash Type	Mode (-m)	Common Use	Example Hash
MD5	0	Legacy web apps, checksums	8743b52063cd8409...
...	...	...	...

Self-check

- ✓ You have 5 distinct hash types in your table.
- ✓ Each entry includes all four requested fields.

- ✓ At least one hash type is something other than MD5, SHA-1, or SHA-256 (explore less common types).

Task 2: Dictionary vs. Mask Attacks

Goal: Understand the conceptual difference between the two most common attack types.

Instructions

Write a short explanation (150–250 words) comparing dictionary attacks and mask attacks. Your explanation should address:

1. **How each attack works** — what input does each require, and how does it generate password candidates?
2. **Strengths** — when is each attack type most effective?
3. **Weaknesses** — when does each attack type fail or become impractical?
4. **A practical example** — describe a scenario where you would choose one over the other, and explain why.

Self-check

- ✓ Your explanation is 150–250 words.
- ✓ You describe both attacks, not just one.
- ✓ You include a concrete scenario with a clear justification for your choice.
- ✓ A classmate who missed this lesson could understand your explanation.

Task 3: Create and Crack Your Own Hash

Goal: Practice the full workflow of creating a hash, building a wordlist, and running Hashcat.

Instructions

1. Choose a simple password (e.g., "sunshine", "iloveyou", "football").
2. Generate the MD5 hash of your chosen password. You can use any of these methods:

```
# macOS / Linux
echo -n "yourpassword" | md5sum

# macOS alternative
echo -n "yourpassword" | md5

# Python
python3 -c "import hashlib; print(hashlib.md5(b'yourpassword').hexdigest())"
```

3. Save the hash to a new file (e.g., `myhash.txt`).
4. Create a custom wordlist file (at least 20 words) that includes your chosen password somewhere in the list.
5. Run Hashcat to crack your hash using your custom wordlist.
6. Verify the result.

Hint

Make sure to use `echo -n` (no newline) when generating the hash. Without the `-n` flag, `echo` adds a trailing newline character, which changes the hash entirely.

Deliverable

A short write-up that includes:

- The password you chose
- The MD5 hash you generated
- The Hashcat command you ran
- The contents of your output file confirming the crack

Self-check

- ✓ You generated the hash yourself (not copied from this guide).
- ✓ Your custom wordlist has at least 20 entries.
- ✓ Hashcat successfully cracked the hash.
- ✓ The output file shows the correct `hash:password` pair.

Submission

There is no formal submission for this course. These tasks are for your own development. However:

- Keep your homework answers — you will refer back to them in later lessons.
- If you are working through this course with a group or mentor, share your Task 2 explanation for discussion.
- Ensure Task 3 is complete before Lesson 02, as the next lesson builds on these skills.

*Next lesson: **Lesson 02 — Installation & Setup** — detailed platform-specific installation, driver configuration, and performance benchmarking.*

Lesson 02: Installation & Setup

Installation & Setup

Everything you need to get hashcat running on your system — from hardware requirements to environment configuration and your first benchmark.

Learning Objectives

- Understand the **hardware requirements** for CPU and GPU cracking
 - Install hashcat on **macOS, Linux, and Windows**
 - Configure **environment variables** for optimal operation
 - Verify your installation with **hashcat --version** and **hashcat -l**
 - Run your first **benchmark** to measure cracking speed
-

CPU Requirements

While hashcat is primarily a GPU tool, the CPU still matters for coordination, rule processing, and fallback cracking modes.

Component	Minimum	Recommended
Architecture	x86_64 with SSE2	AVX2 / AVX-512 support
Intel	Core i5 (8th gen+)	Core i7 / i9 (10th gen+)
AMD	Ryzen 5	Ryzen 7 / Ryzen 9
Cores	4 cores	8+ cores
RAM	4 GB	16 GB+

Why AVX/AVX2 Matters

Hashcat uses SIMD instructions (AVX, AVX2, AVX-512) to accelerate CPU-based operations. Without AVX support, some hash modes will be significantly slower or unavailable in CPU-only

mode.

Notes: Check AVX support with `lscpu | grep avx` on Linux or `sysctl -a | grep machdep.cpu.features` on macOS.

GPU Requirements

The GPU is the heart of hashcat's performance. A modern GPU can be **100–1000x faster** than a CPU for most hash types.

NVIDIA (CUDA / OpenCL)

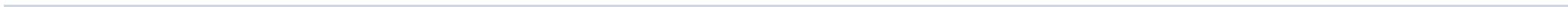
- **Minimum:** GTX 1060 6GB
- **Recommended:** RTX 3080 / RTX 3090
- **Best:** RTX 4090 / A100
- Driver: 440.0+ (CUDA 10.1+)
- VRAM: 6 GB min, 10 GB+ recommended

AMD (OpenCL / ROCm)

- **Minimum:** RX 580 8GB
- **Recommended:** RX 6800 XT / RX 6900 XT
- **Best:** RX 7900 XTX / MI250
- Driver: Adrenalin 21.5+ or ROCm 4.0+
- VRAM: 8 GB min, 16 GB+ recommended

Storage & OS

Storage: 20 GB free space minimum (SSD strongly recommended for large wordlists). **OS:** 64-bit Windows 10/11, Linux (Ubuntu 20.04+, Kali), or macOS 11+.



Installation: macOS

Two paths: Homebrew (simplest) or building from source for the latest features.

Option A: Homebrew

```
# Install via Homebrew
brew install hashcat

# Verify installation
hashcat --version
hashcat -I          # List detected compute devices
```

Option B: Build from Source

```
# Clone the repository
git clone https://github.com/hashcat/hashcat.git
cd hashcat

# Build
make -j$(sysctl -n hw.ncpu)
```

```
# Test  
./hashcat --version
```

macOS Note

Apple removed OpenCL support in macOS 12+. Hashcat on Apple Silicon uses Metal via OpenCL compatibility layer. Performance is limited compared to dedicated NVIDIA/AMD GPUs. For serious work, consider a Linux machine with a discrete GPU.

Installation: Linux & Windows

Linux (Debian/Ubuntu)

```
# Package manager
sudo apt update
sudo apt install hashcat

# Or build from source
git clone https://github.com/hashcat/hashcat.git
cd hashcat
make -j$(nproc)
sudo make install
```

On Kali Linux, hashcat is **pre-installed**.

Windows

```
# 1. Download binary from
#   https://hashcat.net/hashcat/

# 2. Extract the archive
# 3. Open CMD or PowerShell
cd C:\hashcat

# 4. Verify
.\hashcat.exe --version
.\hashcat.exe -I
```

No compilation needed — binaries are provided.

Notes: On Linux, ensure GPU drivers are installed first: `nvidia-smi` for NVIDIA or `rocm-smi` for AMD. Without proper drivers, hashcat will fall back to CPU-only mode.

Environment Variables

Hashcat can be configured via environment variables for repeatable setups and automation scripts.

Variable	Purpose	Example
<code>HASHCAT_HOME</code>	Base directory for hashcat files	<code>/opt/hashcat</code>
<code>HASHCAT_LOG</code>	Log file output path	<code>/var/log/hashcat.log</code>
<code>HASHCAT_GPU</code>	Specify GPU device(s) to use	<code>1,2</code>
<code>HASHCAT_CPU</code>	Force CPU-only mode	<code>1</code>
<code>HASHCAT_FORCE</code>	Bypass warnings (use with caution)	<code>1</code>
<code>HASHCAT_WORKLOAD_PROFILE</code>	Default workload profile (1-4)	<code>3</code>

```
# Add to ~/.bashrc or ~/.zshrc
export HASHCAT_HOME="/opt/hashcat"
```

```
export HASHCAT_LOG="$HOME/.hashcat/logs"  
export HASHCAT_WORKLOAD_PROFILE=3
```

Benchmarking

After installation, run a benchmark to confirm hashcat can use your hardware and to establish a performance baseline.

```
# Full benchmark across all hash types
hashcat -b

# Benchmark a specific hash type (e.g., MD5 = 0)
hashcat -b -m 0

# Benchmark with specific device
hashcat -b -d 1

# Show detected devices
hashcat -I
```

Sample Benchmark Output (RTX 3080)

```
Hashmode: 0 - MD5
Speed.#1.....: 52314.3 MH/s

Hashmode: 1000 - NTLM
```



```
Speed.#1.....: 93827.1 MH/s
```

```
Hashmode: 1800 - sha512crypt
```

```
Speed.#1.....: 412.8 kH/s
```

Notes: MH/s = millions of hashes per second, kH/s = thousands. The difference between MD5 and sha512crypt highlights why algorithm choice matters for password security.

Lab & Homework

Lab Exercises

1. **Install hashcat** — follow the installation steps for your OS and verify with `hashcat --version`
2. **Device detection** — run `hashcat -I` and document your available compute devices
3. **First benchmark** — run `hashcat -b -m 0` and `hashcat -b -m 1000` to compare MD5 vs NTLM speeds
4. **Environment setup** — configure environment variables and create a hashcat working directory

Homework Tasks

1. Run a full benchmark (`hashcat -b`) and identify the 3 fastest and 3 slowest hash types on your hardware.
 2. Set up all recommended environment variables and verify they persist across terminal sessions.
 3. Research why sha512crypt is orders of magnitude slower than MD5 for cracking. Write a short explanation.
-

Theory

Reading time: ~20 minutes · Prerequisites: Lesson 01 (Introduction to Hashcat), basic command-line familiarity

1. System Requirements Overview

Hashcat is an advanced password recovery tool designed to leverage massively parallel computing hardware. While it can run on modest hardware, its true potential is unlocked with modern GPUs. Understanding the hardware requirements helps you plan your setup and set realistic expectations for cracking performance.

CPU Requirements

Hashcat primarily uses the GPU for hash computation, but the CPU handles important tasks: managing the attack workflow, applying rules, distributing work across devices, and serving as a fallback compute device when no GPU is available.

- **Architecture:** x86_64 (64-bit) is required. 32-bit systems are not supported.
- **Instruction sets:** AVX and AVX2 support significantly accelerates CPU-based operations. AVX-512 provides additional gains on supported processors. Without at least SSE2, hashcat will not run.
- **Recommended CPUs:** Intel Core i7/i9 (10th generation or later) or AMD Ryzen 7/9. These provide sufficient cores and modern instruction set extensions.
- **Core count:** At least 4 cores recommended. More cores help when running multi-device attacks or when the CPU itself is used as a compute device.

GPU Requirements

The GPU is the primary compute engine in hashcat. Modern GPUs contain thousands of shader cores that can compute hashes in parallel, making them orders of magnitude faster than CPUs for most hash types.

NVIDIA GPUs

NVIDIA GPUs are supported through both CUDA and OpenCL. CUDA generally provides better performance with hashcat on NVIDIA hardware.

- **Minimum:** GTX 1060 6GB (Pascal architecture) with driver version 440.0+
- **Recommended:** RTX 3080 10GB or RTX 3090 24GB (Ampere architecture)
- **Professional:** RTX 4090 24GB or A100 80GB for maximum throughput
- **VRAM:** 6 GB minimum; 10 GB+ recommended. Some hash types and large wordlist-based attacks require significant VRAM.

AMD GPUs

AMD GPUs are supported through OpenCL and ROCm (on Linux). Performance has improved significantly with recent driver updates.

- **Minimum:** RX 580 8GB (Polaris architecture)
- **Recommended:** RX 6800 XT 16GB or RX 6900 XT 16GB (RDNA 2)
- **Professional:** RX 7900 XTX 24GB or Instinct MI250
- **Drivers:** Adrenalin 21.5+ on Windows, ROCm 4.0+ on Linux

Apple Silicon (M1/M2/M3)

Hashcat has experimental Metal support on Apple Silicon, but performance is significantly lower than dedicated NVIDIA or AMD GPUs. Apple's unified memory architecture is not optimised for the massively parallel workloads hashcat requires. For production cracking, a dedicated GPU rig running Linux is strongly recommended.

RAM and Storage

- **RAM:** 4 GB minimum, 16 GB+ recommended. Large wordlists and rule files are loaded into memory. A 10 GB wordlist (like the full RockYou2024) will require proportional RAM.
- **Storage:** 20 GB free space minimum. SSD is strongly recommended — wordlist loading from an HDD can take minutes instead of seconds for large files. NVMe is ideal for multi-gigabyte wordlists.

Operating System

Hashcat requires a 64-bit operating system:

- **Linux:** Ubuntu 20.04+, Kali Linux, Parrot OS, or any distro with modern GPU drivers. Linux generally provides the best performance and driver support.
- **Windows:** Windows 10 or 11 (64-bit). Ensure GPU drivers are current.
- **macOS:** macOS 11 (Big Sur) or later. Limited GPU acceleration due to Apple's deprecation of OpenCL.

2. Installation on macOS

macOS offers two installation methods: Homebrew (recommended for most users) and building from source (for access to the latest development features).

Method 1: Homebrew

Homebrew is the simplest approach. It handles dependencies automatically and makes updates straightforward.

```
# Install Homebrew if you don't have it
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/

# Install hashcat
brew install hashcat

# Verify the installation
hashcat --version
# Expected output: v6.2.6 (or later)

# List available compute devices
hashcat -I
```

Method 2: Build from Source

Building from source gives you access to the latest commits and development features before they are packaged into a release.

```
# Ensure build tools are available
xcode-select --install
```

```
# Clone the repository
git clone https://github.com/hashcat/hashcat.git
cd hashcat

# Checkout a specific version (optional, for stability)
git checkout v6.2.6

# Build using all available cores
make -j$(sysctl -n hw.ncpu)

# Verify
./hashcat --version

# Optionally install system-wide
sudo make install
```

macOS OpenCL Deprecation

Apple deprecated OpenCL in macOS 10.14 and removed it from the default SDK in macOS 12. Hashcat uses OpenCL as its primary compute API, which means GPU acceleration on macOS is increasingly limited. The hashcat team has been working on Metal backend support, but it is not yet feature-complete. If you need full GPU performance, consider dual-booting Linux or using a dedicated Linux machine.

3. Installation on Linux

Linux is the recommended platform for serious hashcat use. It provides the best GPU driver support and performance.

Method 1: Package Manager

```
# Debian / Ubuntu
sudo apt update
sudo apt install hashcat

# Fedora / RHEL
sudo dnf install hashcat
```

```
# Arch Linux
sudo pacman -S hashcat

# Verify
hashcat --version
```

Kali Linux

Hashcat comes pre-installed on Kali Linux. Simply update to the latest version with `sudo apt update && sudo apt upgrade hashcat`.

Method 2: Build from Source

```
# Install build dependencies
sudo apt install build-essential git

# Clone and build
git clone https://github.com/hashcat/hashcat.git
cd hashcat
make -j$(nproc)
sudo make install

# Verify
hashcat --version
```

GPU Driver Setup (Linux)

GPU drivers must be installed before hashcat can use GPU acceleration. Without drivers, hashcat will only see the CPU as a compute device.

NVIDIA Drivers

```
# Ubuntu: Use the recommended driver
sudo ubuntu-drivers autoinstall
sudo reboot

# Verify NVIDIA driver
nvidia-smi
```



```
# Install CUDA toolkit (optional, for CUDA backend)
sudo apt install nvidia-cuda-toolkit
```

AMD Drivers (ROCm)

```
# Add ROCm repository (Ubuntu 22.04 example)
wget -q -O - https://repo.radeon.com/rocm/rocm.gpg.key | sudo apt-key add -
echo 'deb [arch=amd64] https://repo.radeon.com/rocm/apt/latest/ ubuntu main' |
sudo apt update
sudo apt install rocm-ocl-runtime

# Verify
rocm-smi
hashcat -I
```

4. Installation on Windows

Windows installation uses pre-compiled binaries downloaded from the official hashcat website. No compilation is required.

```
# 1. Download the latest release from:
#   https://hashcat.net/hashcat/
#   (Choose the .7z or .zip archive)

# 2. Extract the archive to a permanent location
#   Example: C:\Tools\hashcat

# 3. Open Command Prompt or PowerShell
cd C:\Tools\hashcat

# 4. Verify installation
.\hashcat.exe --version

# 5. List compute devices
.\hashcat.exe -I
```

Adding to PATH (Windows)

To use hashcat from any directory, add the hashcat folder to your system PATH:

1. Open **System Properties** → **Advanced** → **Environment Variables**
2. Under **System variables**, select **Path** and click **Edit**
3. Add `C:\Tools\hashcat` (or wherever you extracted it)
4. Click **OK** and restart your terminal

GPU Drivers on Windows

Ensure you have the latest GPU drivers installed:

- **NVIDIA:** Download from [nvidia.com/drivers](https://www.nvidia.com/drivers). Use the Game Ready or Studio driver. Verify with `nvidia-smi` in Command Prompt.
- **AMD:** Download Adrenalin drivers from [amd.com/en/support](https://www.amd.com/en/support). Ensure OpenCL runtime is included in the installation.

5. Verifying Your Installation

After installation, run these commands to confirm everything is working correctly:

```
# Check hashcat version
hashcat --version

# List all detected compute devices
hashcat -I

# Expected output (example with NVIDIA GPU):
# OpenCL Platform ID #1
#   Vendor...: NVIDIA Corporation
#   Name.....: NVIDIA CUDA
#   Version..: OpenCL 3.0 CUDA 12.2.138
#
# Backend Device ID #1
#   Type.....: GPU
#   Vendor.ID.....: 32
#   Name.....: NVIDIA GeForce RTX 3080
#   Memory.Total....: 10240 MB
#   Driver.Version.: 535.129.03

# Run a quick benchmark on MD5
hashcat -b -m 0
```

```
# Run a quick test crack
echo -n "password" | md5sum
# Output: 5f4dcc3b5aa765d61d8327deb882cf99
echo "5f4dcc3b5aa765d61d8327deb882cf99" > /tmp/test.hash
```

6. Environment Configuration

Environment variables allow you to set default behaviours without passing flags on every command. This is especially useful for automation and scripting.

Variable	Purpose	Example Value
<code>HASHCAT_HOME</code>	Base directory where hashcat looks for modules, rules, and charsets	<code>/opt/hashcat</code>
<code>HASHCAT_LOG</code>	Directory for hashcat log files	<code>\$HOME/.hashcat/logs</code>
<code>HASHCAT_GPU</code>	Default GPU device ID(s) to use	<code>1</code> or <code>1,2</code>
<code>HASHCAT_CPU</code>	Enable CPU-only mode (set to 1)	<code>0</code> (default)
<code>HASHCAT_FORCE</code>	Bypass device type and capability warnings	<code>1</code> (use with caution)
<code>HASHCAT_WORKLOAD_PROFILE</code>	Default workload tuning profile (1=low, 2=default, 3=high, 4=nightmare)	<code>3</code>

Setting Up on Linux / macOS

```
# Add to ~/.bashrc, ~/.zshrc, or ~/.profile
export HASHCAT_HOME="/opt/hashcat"
export HASHCAT_LOG="$HOME/.hashcat/logs"
export HASHCAT_WORKLOAD_PROFILE=3

# Create the log directory
mkdir -p "$HOME/.hashcat/logs"
```

```
# Reload your shell configuration
source ~/.bashrc # or source ~/.zshrc
```

Setting Up on Windows

```
# PowerShell (persistent, user-level)
[Environment]::SetEnvironmentVariable("HASHCAT_HOME", "C:\Tools\hashcat", "User")
[Environment]::SetEnvironmentVariable("HASHCAT_LOG", "$env:USERPROFILE\.hashcat", "User")
[Environment]::SetEnvironmentVariable("HASHCAT_WORKLOAD_PROFILE", "3", "User")

# Create the log directory
New-Item -ItemType Directory -Force -Path "$env:USERPROFILE\.hashcat\logs"
```

Workload Profiles Explained

Profile	Name	GPU Impact	Use Case
1	Low	Minimal — desktop remains responsive	Background cracking while using the computer
2	Default	Moderate	General use on a workstation
3	High	Heavy — desktop may stutter	Dedicated cracking machine
4	Nightmare	Maximum — system nearly unusable	Headless server / dedicated rig

The --force Flag

Setting `HASHCAT_FORCE=1` or using `--force` on the command line bypasses all device warnings and capability checks. This can lead to **incorrect results**, **hardware damage** (overheating), or **crashes**. Only use it when you understand exactly why hashcat is warning you and have confirmed the warning does not apply to your situation.

7. Directory Structure

A well-organised hashcat working directory will save you time as your projects grow:

```
~/hashcat-workspace/
hashes/           # Hash files to crack
wordlists/        # Dictionary files (rockyou.txt, etc.)
rules/            # Rule files (best64.rule, etc.)
masks/            # Custom mask files
potfiles/         # Cracked hash results
sessions/         # Session restore files
logs/             # Benchmark and run logs
```

```
# Create the workspace
mkdir -p ~/hashcat-workspace/{hashes,wordlists,rules,masks,potfiles,sessions,logs}
```

8. Troubleshooting Common Issues

hashcat: command not found

The hashcat binary is not in your PATH. Either add the installation directory to PATH, use the full path to the binary, or create a symlink:

```
sudo ln -s /opt/hashcat/hashcat /usr/local/bin/hashcat
```

No devices found

GPU drivers are not installed or not loading correctly. Run `nvidia-smi` (NVIDIA) or `rocm-smi` (AMD) to check driver status. On macOS, ensure you have not removed the OpenCL framework.

CL_OUT_OF_RESOURCES or similar GPU errors

The GPU ran out of memory or hit a compute timeout. Try reducing the workload profile (`-w 1`), using a smaller wordlist, or increasing the GPU's TDR (timeout detection recovery) limit on Windows.

Benchmark speeds are lower than expected

- Ensure no other GPU-intensive applications are running (games, other compute workloads)

- Check GPU temperatures — thermal throttling reduces performance. Target below 80 degrees C.
- Verify you are using the correct workload profile (`-w 3` or `-w 4` for maximum speed)
- Update GPU drivers to the latest version

Further Reading

- [Hashcat FAQ](#) — Official FAQ covering common installation and runtime issues.
- [Hashcat Wiki](#) — Comprehensive documentation for all features and options.
- [Hashcat on GitHub](#) — Source code, issue tracker, and build instructions.
- [Benchmark Reference](#) — Community-submitted benchmark results for various hardware configurations.

Next lesson: **Lesson 03 — Command Structure & Parameters** — understanding hashcat's command-line syntax and core options.

Lab Guide

Time estimate: 30–45 minutes · Prerequisites: Terminal/command-line access, internet connection for downloads

Setup

This lab walks you through installing hashcat, verifying the installation, detecting your hardware, and running your first benchmark. Follow the steps for your operating system.

Before You Begin

Ensure your GPU drivers are up to date. Run `nvidia-smi` (NVIDIA) or `rocm-smi` (AMD) to verify. If you do not have a discrete GPU, hashcat will use CPU-only mode — this is fine for learning, but cracking speeds will be significantly lower.

Exercise 1: Install Hashcat

Goal

Install hashcat on your system and verify it runs correctly.

Steps — macOS

1. Open Terminal.
2. Install hashcat using Homebrew:

```
brew install hashcat
```

3. Verify the installation:

```
hashcat --version
```

4. Confirm the binary location:

```
which hashcat
```

Steps — Linux (Debian/Ubuntu)

1. Update your package list and install:

```
sudo apt update  
sudo apt install hashcat
```

2. Verify:

```
hashcat --version
```

Steps — Windows

1. Download the latest release from hashcat.net/hashcat/.
2. Extract the archive to `C:\Tools\hashcat` (or your preferred location).
3. Open PowerShell and navigate to the directory:

```
cd C:\Tools\hashcat  
.\hashcat.exe --version
```

Expected Output

```
$ hashcat --version  
v6.2.6
```

(Version number may vary — any version 6.x or later is fine.)

Troubleshooting

- If `hashcat: command not found` — ensure the installation directory is in your PATH.
- On macOS, if Homebrew is not installed, install it first: `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`

- On Linux, if the package is outdated, consider building from source (see Theory page).

Exercise 2: Device Detection

Goal

Identify all compute devices available to hashcat on your system.

Steps

1. Run the device information command:

```
hashcat -I
```

2. Read the output carefully and note:
 - How many compute platforms were detected?
 - What type is each device (CPU or GPU)?
 - What is the device name and VRAM (for GPUs)?
 - What driver version is reported?
3. Record this information — you will need it for the homework.

Expected Output (Example)

```
$ hashcat -I

hashcat (v6.2.6) starting in backend information mode

OpenCL Info:
=====

OpenCL Platform ID #1
  Vendor...: NVIDIA Corporation
  Name.....: NVIDIA CUDA
  Version..: OpenCL 3.0 CUDA 12.2.138

Backend Device ID #1
  Type.....: GPU
```

```
Vendor.ID.....: 32
Vendor.....: NVIDIA Corporation
Name.....: NVIDIA GeForce RTX 3080
Version.....: OpenCL 3.0 CUDA
Processor(s)....: 68
Clock.....: 1710 MHz
Memory.Total....: 10240 MB (10 GB)
Memory.Free.....: 9856 MB
Local.Memory....: 48 KB
OpenCL.Version..: OpenCL C 1.2
Driver.Version..: 535.129.03
```

CPU-Only Systems

If you only see a CPU device, that is normal for systems without a discrete GPU. You can still complete all exercises. Performance will be lower, but the learning experience is the same. Use the `--force` flag if hashcat warns about running on CPU-only.

Exercise 3: Run Your First Benchmark

Goal

Measure your system's hash cracking speed across common hash types and establish a performance baseline.

Steps

1. Run a targeted benchmark on MD5 (hash mode 0):

```
hashcat -b -m 0
```

2. Run a benchmark on NTLM (hash mode 1000):

```
hashcat -b -m 1000
```

3. Run a benchmark on sha512crypt (hash mode 1800):

```
hashcat -b -m 1800
```

4. Record the speed for each hash type in a table:

Hash Type	Mode	Speed
MD5	0	(your result)
NTLM	1000	(your result)
sha512crypt	1800	(your result)

5. Calculate the ratio: how many times faster is MD5 compared to sha512crypt?

Key Observations to Note

- MD5 and NTLM are very fast because they are simple, single-iteration hash functions.
- sha512crypt is orders of magnitude slower because it uses 5000 rounds by default — this is by design to resist brute-force attacks.
- The speed difference illustrates why modern systems use slow hash functions (bcrypt, scrypt, Argon2) for password storage.

Exercise 4: Environment Setup

Goal

Configure your hashcat working environment with a proper directory structure and environment variables.

Steps

1. Create a hashcat workspace directory:

```
mkdir -p ~/hashcat-workspace/{hashes,wordlists,rules,masks,potfiles,session}
```

2. Verify the structure:

```
ls -la ~/hashcat-workspace/
```

3. Add environment variables to your shell configuration. For Bash:

```
cat >> ~/.bashrc << 'EOF'

# Hashcat environment
export HASHCAT_HOME="$(which hashcat | xargs dirname)"
export HASHCAT_LOG="$HOME/hashcat-workspace/logs"
export HASHCAT_WORKLOAD_PROFILE=3
EOF

source ~/.bashrc
```

For Zsh (macOS default):

```
cat >> ~/.zshrc << 'EOF'

# Hashcat environment
export HASHCAT_HOME="$(which hashcat | xargs dirname)"
export HASHCAT_LOG="$HOME/hashcat-workspace/logs"
export HASHCAT_WORKLOAD_PROFILE=3
EOF

source ~/.zshrc
```

4. Verify the variables are set:

```
echo $HASHCAT_HOME
echo $HASHCAT_LOG
echo $HASHCAT_WORKLOAD_PROFILE
```

Exercise 5: Test Crack

Goal

Perform a simple test crack to confirm that your entire pipeline works end to end.

Steps

1. Create a test hash file with a known MD5 hash:

```
# MD5 hash of "password123"
echo "482c811da5d5b4bc6d497ffa98491e38" > ~/hashcat-workspace/hashes/test.h
```

2. Create a small test wordlist:

```
cat > ~/hashcat-workspace/wordlists/test.dict << 'EOF'
admin
letmein
password
password123
qwerty
123456
EOF
```

3. Run hashcat in dictionary attack mode:

```
hashcat -m 0 -a 0 ~/hashcat-workspace/hashes/test.hash ~/hashcat-workspace/
```

4. Verify the result:

```
hashcat -m 0 ~/hashcat-workspace/hashes/test.hash --show --potfile-path ~/r
```

Expected Output

```
482c811da5d5b4bc6d497ffa98491e38:password123

Session.....: hashcat
Status.....: Cracked
Hash.Mode.....: 0 (MD5)
Hash.Target.....: 482c811da5d5b4bc6d497ffa98491e38
Time.Started.....: ...
Time.Estimated....: ...
Recovered.....: 1/1 (100.00%) Digests
```

Finishing Up

Once you have completed all five exercises:

1. Confirm hashcat is installed and reports a version number.
2. Confirm `hashcat -I` shows at least one compute device.
3. Confirm benchmarks ran successfully and you recorded the speeds.
4. Confirm your workspace directory exists with all subdirectories.
5. Confirm the test crack found the correct password.

Self-Check

- ✓ `hashcat --version` returns a version number.
- ✓ `hashcat -I` lists at least one backend device (CPU or GPU).
- ✓ You can explain why MD5 benchmarks are orders of magnitude faster than sha512crypt.
- ✓ Environment variables are set and persist after opening a new terminal.
- ✓ The test crack successfully recovered `password123` from the MD5 hash.

Homework

Complete these tasks before Lesson 03. Focus on understanding your hardware capabilities and establishing a reliable working environment.

Task 1: Full Benchmark Report

Goal: Understand how cracking speed varies dramatically across different hash algorithms and learn to read benchmark output.

Instructions

1. Run a full benchmark across all hash types:

```
hashcat -b
```

This may take 10–20 minutes depending on your hardware.

2. From the output, identify and record:
 - The **3 fastest** hash types (highest speed in H/s, MH/s, or GH/s)
 - The **3 slowest** hash types (lowest speed)
3. Calculate the ratio between the fastest and slowest hash type. How many orders of magnitude separate them?
4. Present your findings in a table:

Deliverable

Category	Hash Type	Mode Number	Speed
Fastest #1	(your result)	...	...
Fastest #2	(your result)	...	...
Fastest #3	(your result)	...	...
Slowest #1	(your result)	...	...

Category	Hash Type	Mode Number	Speed
Slowest #2	(your result)	...	...
Slowest #3	(your result)	...	...

Plus a one-sentence answer: "The fastest hash type is approximately ____x faster than the slowest."

Self-check

- ✓ You ran the full benchmark to completion.
- ✓ You correctly identified 3 fastest and 3 slowest hash types.
- ✓ You calculated the speed ratio and it makes sense (typically 10,000x to 1,000,000x difference).
- ✓ You can explain why some hashes are intentionally slow (key derivation functions).

Task 2: Environment Verification

Goal: Ensure your hashcat environment is properly configured and will persist across sessions.

Instructions

1. Add the following environment variables to your shell configuration file

(`~/.bashrc`, `~/.zshrc`, or Windows environment variables):

```
export HASHCAT_HOME="/path/to/hashcat"
export HASHCAT_LOG="$HOME/hashcat-workspace/logs"
export HASHCAT_WORKLOAD_PROFILE=3
```

2. Close your terminal completely and open a new one.
3. Verify each variable is set:

```
echo "HASHCAT_HOME=$HASHCAT_HOME"
echo "HASHCAT_LOG=$HASHCAT_LOG"
echo "HASHCAT_WORKLOAD_PROFILE=$HASHCAT_WORKLOAD_PROFILE"
```


4. Verify the workspace directory structure exists:

```
ls -la ~/hashcat-workspace/
```

5. Create a simple test script that checks your setup:

```
#!/bin/bash
echo "=== Hashcat Environment Check ==="
echo "Version: $(hashcat --version)"
echo "HASHCAT_HOME: ${HASHCAT_HOME:-NOT SET}"
echo "HASHCAT_LOG: ${HASHCAT_LOG:-NOT SET}"
echo "HASHCAT_WORKLOAD_PROFILE: ${HASHCAT_WORKLOAD_PROFILE:-NOT SET}"
echo ""
echo "=== Workspace Directories ==="
for dir in hashes wordlists rules masks potfiles sessions logs; do
    if [ -d "$HOME/hashcat-workspace/$dir" ]; then
        echo "    [OK] $dir"
    else
        echo "    [MISSING] $dir"
    fi
done
echo ""
echo "=== Compute Devices ==="
hashcat -I 2>/dev/null | grep -E "(Name\\.\\.\\.|Type\\.\\.\\.|Memory\\.Total)"
```

6. Run the script and confirm all checks pass.

Deliverable

The output of your environment check script showing all variables set and all directories present.

Self-check

- ✓ All three environment variables are set after opening a new terminal.
- ✓ All workspace directories exist (hashes, wordlists, rules, masks, potfiles, sessions, logs).
- ✓ The check script runs without errors.
- ✓ At least one compute device is detected by `hashcat -I`.

Task 3: Why Are Some Hashes Slow?

Goal: Understand the security implications of hash algorithm speed from both the attacker's and defender's perspective.

Instructions

In your benchmark results, you observed that some hash types (MD5, NTLM) are cracked at billions of hashes per second, while others (sha512crypt, bcrypt, Argon2) are cracked at only thousands or hundreds of hashes per second. This is not a hashcat limitation — it is by design.

Research and write a short explanation (200–300 words) that addresses:

1. **Purpose:** Why were fast hashes like MD5 and SHA-1 created? What were they designed for?
2. **Problem:** Why is a fast hash function a poor choice for storing passwords?
3. **Solution:** How do key derivation functions (bcrypt, scrypt, Argon2, sha512crypt) solve this problem? What specific techniques do they use (iteration count, memory hardness, etc.)?
4. **Attacker perspective:** As someone learning hashcat, why does this distinction matter for your work?

Deliverable

A written explanation (200–300 words) that a fellow security student could understand.

Self-check

- ✓ You correctly distinguish between general-purpose hash functions and key derivation functions.
- ✓ You mention at least two specific techniques that slow hashes use (e.g., iteration count, memory hardness, salting).
- ✓ You connect the concept to practical hashcat usage (time estimates, attack feasibility).
- ✓ Your explanation is clear enough for a non-expert to follow.

Bonus Challenge

Bonus: Multi-Device Comparison

Goal: Compare cracking performance across different compute devices on your system.

Instructions

If your system has both a CPU and a GPU (check with `hashcat -I`):

1. Run an MD5 benchmark using only the GPU:

```
hashcat -b -m 0 -d 1
```

2. Run the same benchmark using only the CPU:

```
hashcat -b -m 0 -d 2
```

(Device IDs may differ on your system — check `hashcat -I` for the correct numbers.)

3. Calculate the GPU-to-CPU speed ratio for MD5.
4. Repeat for sha512crypt (`-m 1800`). Is the ratio different? Why?

Self-check

- ✓ You benchmarked both devices individually.
- ✓ You calculated the speed ratio for at least two hash types.
- ✓ You can explain why the GPU advantage varies across hash types.

Submission

There is no formal submission for this course. These tasks are for your own development. However:

- Save your benchmark results — you will reference them in later lessons when estimating crack times.

- Keep your environment check script — it is useful for quickly verifying your setup on new machines.
- Your Task 3 explanation will be relevant again in Lesson 04 (Hash Types & Identification).

*Next lesson: **Lesson 03 — Command Structure & Parameters***

Lesson 03: Command Structure & Parameters

Command Structure & Parameters

Every hashcat command follows the same pattern: the binary, the flags, and the arguments. Master the structure once and every attack type becomes a variation of the same template.

Learning Objectives

- Understand the **general syntax** of a hashcat command
 - Know the **key parameters**: `-m`, `-a`, `-o`, `-r`, `-w`, `-D`
 - Distinguish between **flags** (options) and **positional arguments** (hash file, dictionary, mask)
 - Build complete commands for **dictionary, mask, and hybrid attacks**
 - Use **sessions, incremental mode, and workload tuning** for long-running jobs
-

The Basic Command

The hashcat binary name depends on your platform, but the structure is always the same.

Linux / macOS

```
hashcat [options] hash_file [dict|mask]
```

Or explicitly: `./hashcat.bin`

Windows

```
hashcat.exe [options] hash_file [dict|mask]
```

Anatomy of a Command

```
hashcat -m 0 -a 0 -o results.txt hashes.txt rockyou.txt
#      |      |      |      |      |      |
#      |      |      |      |      |      | dictionary (argument)
#      |      |      |      |      |      | hash file (argument)
#      |      |      |      |      |      | output file (flag + value)
#      |      |      |      |      |      | attack mode (flag + value)
#      |      |      |      |      |      | hash type (flag + value)
```

Notes: Emphasize that order of flags does not matter, but the hash file must always come before the dictionary/mask arguments.

Hash Type: `-m`

The `-m` flag tells hashcat what hash algorithm to crack. Choosing the wrong mode means zero results.

Value	Algorithm	Example Use Case
0	MD5	Web app databases, legacy systems
100	SHA-1	Older web applications
1000	NTLM	Windows password hashes
1800	SHA-512 (Unix)	Linux /etc/shadow
3200	bcrypt	Modern web apps
22000	WPA-PBKDF2-PMKID+EAPOL	Wi-Fi auditing

Full list: `hashcat --help` or `hashcat --example-hashes`

Notes: Run `hashcat --example-hashes | head -60` live to show how many modes exist (~350+). Students should bookmark the wiki page.

Attack Mode: `-a`

The `-a` flag selects the attack strategy. Each mode determines what arguments follow the hash file.

Value	Mode	Arguments	Description
0	Dictionary	wordlist	Tries every word in a wordlist
1	Combinator	wordlist1 wordlist2	Concatenates words from two lists
3	Mask / Brute-force	mask pattern	Generates candidates from a character mask
6	Hybrid: dict + mask	wordlist mask	Appends mask patterns to dictionary words
7	Hybrid: mask + dict	mask wordlist	Prepends mask patterns to dictionary words

Key insight: The attack mode dictates the arguments. `-a 0` needs a wordlist. `-a 3` needs a mask. `-a 6` needs both. Get these wrong and hashcat will throw an error.

Output & Rules

Control where results go and how candidates are mutated.

Output: `-o` & `--outfile-format`

- `-o results.txt` — save cracked hashes to file
- `--outfile-format 1` — hash only
- `--outfile-format 2` — plain password only
- `--outfile-format 3` — hash:password (default)
- `--outfile-format 5` — hash:password:hex

Rules: `-r`

- `-r rules/best64.rule` — apply rule file
- Mutates dictionary words (capitalize, append digits, leet-speak, etc.)
- Multiple `-r` flags = chained rules
- Dramatically expands effective wordlist size

```
# Dictionary attack with rules, saving hash:password pairs
hashcat -m 0 -a 0 -o cracked.txt --outfile-format 3 -r rules/best64.rule hashes.txt rockyou.txt
```

Performance & Session Control

Long-running cracks need workload tuning and the ability to pause/resume.

Workload: `-w`

- `-w 1` — Low (desktop responsive)
- `-w 2` — Default
- `-w 3` — High (recommended for dedicated rigs)
- `-w 4` — Nightmare (may freeze UI)

Device: `-D`

- `-D 1` — CPU only
- `-D 2` — GPU only

Sessions: `--session` / `--restore`

```
# Start a named session
hashcat --session mysession -m 0 ...

# Resume later
hashcat --session mysession --restore
```

Sessions save progress to disk. Essential for multi-hour cracks.

- -D 1,2 — Both
-

Incremental Mode: `-i`

When using mask attacks, incremental mode tries all lengths from `--increment-min` to `--increment-max` instead of only the exact mask length.

Without increment

```
# Only tries exactly 8-character combinations  
hashcat -m 1000 -a 3 hashes.txt ?a?a?a?a?a?a?a
```

With increment

```
# Tries 4-char, 5-char, 6-char, 7-char, then 8-char  
hashcat -m 1000 -a 3 --increment --increment-min 4 --increment-max 8 hashes.txt ?a?a?a?a?a?a?a
```

This is powerful but dramatically increases the total keyspace. Use `--increment-min` to skip trivially short passwords and save time.

Notes: Show the keyspace calculation difference: 8-char only vs 4-to-8. The numbers are eye-opening.

Putting It All Together

A full-featured command combining multiple parameters for an NTLM mask attack.

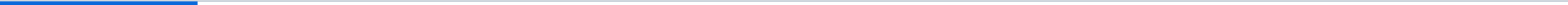
```
hashcat -m 1000 -a 3 -o results.txt --session mysession \  
--increment --increment-min 4 --increment-max 8 \  
-w 3 hashes.txt ?a?a?a?a?a?a?a
```

Breaking it down

- `-m 1000` — NTLM hashes
- `-a 3` — Mask attack
- `-o results.txt` — Output cracked hashes to file
- `--session mysession` — Named session for pause/resume
- `--increment --increment-min 4 --increment-max 8` — Try lengths 4 through 8
- `-w 3` — High-performance workload
- `hashes.txt` — Input hash file
- `?a?a?a?a?a?a?a` — Mask: all printable characters, up to 8 positions



Notes: Have students build this command parameter by parameter in the lab. Each flag should be added one at a time so they understand what each piece does.



Command Structure & Parameters

Complete reference for hashcat command-line syntax, flags, and arguments.

General Syntax

Every hashcat invocation follows the same pattern:

```
hashcat [options...] hash_file [dictionary|mask|dictionary+mask]
```

The binary name varies by platform:

- **Linux / macOS:** `hashcat` (or `./hashcat.bin` if running from the extracted directory)
- **Windows:** `hashcat.exe`

Options (flags) can appear in any order before or after the hash file, but the positional arguments — hash file, wordlists, and masks — must appear in the correct sequence after all flags.

Hash Type: `-m`

The `-m` parameter specifies the hash algorithm hashcat should use for cracking. Using the wrong mode is the most common beginner mistake — hashcat will run but find nothing because it is interpreting the hash bytes incorrectly.

Common Hash Modes

Mode	Algorithm	Typical Source
<code>0</code>	MD5	Web application databases, legacy systems
<code>100</code>	SHA-1	Older web applications, Git commits
<code>1000</code>	NTLM	Windows SAM database, Active Directory

Mode	Algorithm	Typical Source
1800	sha512crypt (\$6\$)	Linux /etc/shadow
2500	WPA/WPA2 (legacy)	Wi-Fi capture files
3200	bcrypt (\$2*\$)	Modern web applications
13100	Kerberos TGS-REP (RC4)	Kerberoasting attacks
22000	WPA-PBKDF2-PMKID+EAPOL	Modern Wi-Fi auditing

Tip: Use `hashcat --example-hashes` to see sample hashes for every supported mode. This helps you identify unknown hash types by visual comparison. There are over 350 supported modes.

Attack Mode: `-a`

The `-a` parameter selects the attack strategy. It determines what kind of arguments (wordlists, masks, or both) must follow the hash file.

Mode	Name	Required Arguments	Description
0	Dictionary (Straight)	wordlist	Tests every word from a wordlist against the hashes. The most common starting point.
1	Combinator	wordlist1 wordlist2	Combines every word from list 1 with every word from list 2. Good for compound passwords like "sunflower123".
3	Mask (Brute-force)	mask pattern	Generates candidates based on character position masks. Full brute-force when using <code>?a</code> (all printable).
6	Hybrid: Dictionary +	wordlist mask	Appends mask-generated characters to each dictionary word. Example: <code>password</code>

Mode	Name	Required Arguments	Description
	Mask		+ <code>?d?d</code> = <code>password00</code> through <code>password99</code> .
7	Hybrid: Mask + Dictionary	mask wordlist	Prepends mask-generated characters to each dictionary word. Example: <code>?d?d</code> + <code>password</code> = <code>00password</code> through <code>99password</code> .

Mask Charsets

In mask mode (`-a 3`), each position in the mask is defined by a charset placeholder:

Placeholder	Characters	Count
<code>?l</code>	Lowercase letters (a-z)	26
<code>?u</code>	Uppercase letters (A-Z)	26
<code>?d</code>	Digits (0-9)	10
<code>?s</code>	Special characters (!@#\$... etc.)	33
<code>?a</code>	All printable ASCII (?!u?d?s)	95
<code>?b</code>	All bytes (0x00-0xff)	256

Output Control

Output File: `-o`

By default, hashcat displays cracked hashes on screen and stores them in `hashcat.potfile`. Use `-o` to write results to a specific file:

```
hashcat -m 0 -a 0 -o cracked.txt hashes.txt rockyou.txt
```

Output Format: `--outfile-format`

Controls what gets written to the output file. The value is a bitmask — add the numbers together to combine fields:

Value	Content	Example Output
1	Hash only	5f4dcc3b5aa765d61d8327deb882cf99
2	Plain password only	password
3	Hash:password (default)	5f4dcc3b...cf99:password
4	Hex-encoded plain	70617373776f7264
5	Hash:hex-plain	5f4dcc3b...cf99:70617373776f7264

Rules: `-r`

Rules transform dictionary words on the fly, dramatically expanding the effective wordlist without increasing file size. Each rule defines a mutation (capitalize, reverse, append digits, leet-speak substitution, etc.).

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

# Multiple rule files (chained — every combination is applied)
hashcat -m 0 -a 0 -r rules/best64.rule -r rules/toggles1.rule hashes.txt rockyou.txt
```

Included rule files: Hashcat ships with several rule files in the `rules/` directory. Popular choices: `best64.rule` (64 most effective rules), `rockyou-30000.rule` (30k rules extracted from the RockYou breach), `d3ad0ne.rule` (34k+ aggressive rules).

Incremental Mode: `-i`

By default, a mask attack only tries candidates matching the exact mask length. The `-i` (or `--increment`) flag tells hashcat to iterate through increasing lengths.

```
# Without increment: only tries 8-character combinations
hashcat -m 0 -a 3 hashes.txt ?a?a?a?a?a?a?a

# With increment: tries 1-char, 2-char, ..., up to 8-char
hashcat -m 0 -a 3 -i hashes.txt ?a?a?a?a?a?a?a?a

# With min/max bounds: tries 4-char through 8-char only
hashcat -m 0 -a 3 -i --increment-min 4 --increment-max 8 hashes.txt ?a?a?a?a?a?
```

Warning: Incremental mode with `?a` (all printable) generates enormous keyspaces. An 8-character `?a` mask alone produces $95^8 = 6.6$ quadrillion candidates. Adding lengths 4 through 7 adds another ~7.4 billion on top. Always set `--increment-min` to skip trivially short passwords.

Workload Profile: `-w`

The workload profile controls how aggressively hashcat uses the GPU. Higher values yield faster cracking but may cause system unresponsiveness.

Level	Name	Use Case
1	Low	Desktop use — system remains responsive
2	Default	General purpose
3	High	Dedicated cracking rig, headless server
4	Nightmare	Maximum throughput; may freeze GUI/display

Device Selection: `-D`

By default, hashcat auto-detects available compute devices. Use `-D` to restrict to specific device types:

- `-D 1` — CPU only
- `-D 2` — GPU only
- `-D 1,2` — Both CPU and GPU

Use `-I` (or `--backend-info`) to list all detected devices and their capabilities before choosing.

Session Management

Long-running attacks can be paused and resumed using sessions.

```
# Start a named session
hashcat --session audit_q1 -m 1000 -a 0 hashes.txt rockyou.txt

# Pause: press 'c' during execution (checkpoint), or 'q' to quit gracefully

# Resume the session later
hashcat --session audit_q1 --restore
```

Session data is stored in `*.restore` and `*.induct` files in the hashcat working directory. Without `--session`, hashcat uses a default session name and will refuse to start a second instance.

Complete Command Examples

Example 1: Simple MD5 dictionary attack

```
hashcat -m 0 -a 0 -o results.txt hashes.txt rockyou.txt
```

Cracks MD5 hashes using the RockYou wordlist. Results written as hash:password pairs to `results.txt`.

Example 2: NTLM mask attack with incremental mode

```
hashcat -m 1000 -a 3 -o results.txt --session mysession \
--increment --increment-min 4 --increment-max 8 \
-w 3 hashes.txt ?a?a?a?a?a?a?a
```

Cracks NTLM hashes using a brute-force mask attack. Tries all printable character combinations from 4 to 8 characters. Uses high workload on GPU. Named session allows pause/resume.

Example 3: Dictionary + rules attack on SHA-512

```
hashcat -m 1800 -a 0 -r rules/best64.rule -o cracked.txt \
-w 2 shadow_hashes.txt rockyou.txt
```

Cracks Linux shadow file hashes (sha512crypt) using a dictionary with the best64 rule file applied to each word.

Example 4: Hybrid dictionary + mask

```
hashcat -m 0 -a 6 -o results.txt hashes.txt rockyou.txt ?d?d?d?d
```

Appends 4 digits to every word in the dictionary. Catches patterns like `password2024` or `admin1234`.

Example 5: Combinator attack

```
hashcat -m 0 -a 1 -o results.txt hashes.txt words1.txt words2.txt
```

Concatenates every word from `words1.txt` with every word from `words2.txt`. Effective for compound passwords.

Quick Reference Card

Flag	Purpose	Common Values
<code>-m</code>	Hash type	0 (MD5), 1000 (NTLM), 1800 (SHA-512), 3200 (bcrypt)
<code>-a</code>	Attack mode	0 (dict), 1 (combo), 3 (mask), 6 (hybrid d+m), 7 (hybrid m+d)
<code>-o</code>	Output file	Any file path
<code>--outfile-format</code>	Output content	1 (hash), 2 (plain), 3 (hash:plain)
<code>-r</code>	Rules file	rules/best64.rule, rules/rockyou-30000.rule

Flag	Purpose	Common Values
<code>-i</code>	Incremental mode	(no value — it is a toggle)
<code>--increment-min</code>	Min length for increment	1–N
<code>--increment-max</code>	Max length for increment	1–N
<code>-w</code>	Workload profile	1 (low), 2 (default), 3 (high), 4 (nightmare)
<code>-D</code>	Device type	1 (CPU), 2 (GPU), 1,2 (both)
<code>--session</code>	Session name	Any string
<code>--restore</code>	Resume session	(no value — it is a toggle)

Further Reading

- [Hashcat Wiki](#) — official documentation and hash mode reference
- [Mask Attack Reference](#) — detailed mask charset documentation
- [Rule-Based Attack Reference](#) — rule syntax and examples
- `hashcat --help` — the built-in help is comprehensive and always up to date

Lab Guide

Hands-on exercises: build and run hashcat commands with different parameters.

Prerequisites: Hashcat installed and working (verified in Lesson 02). A terminal open in the hashcat directory. The `rockyou.txt` wordlist available (or any wordlist of your choice).

Important: All exercises use test hashes that you generate yourself. Never use hashcat against hashes you do not own or have explicit authorization to test. This lab is for educational purposes only.

Setup: Create Test Hashes

Before starting the exercises, generate some known hashes so you can verify that your commands actually work.

Generate MD5 test hashes

Create a file called `test_md5.txt` with MD5 hashes of known passwords:

```
# Linux / macOS
echo -n "password" | md5sum | awk '{print $1}' > test_md5.txt
echo -n "123456" | md5sum | awk '{print $1}' >> test_md5.txt
echo -n "letmein" | md5sum | awk '{print $1}' >> test_md5.txt
echo -n "dragon" | md5sum | awk '{print $1}' >> test_md5.txt
echo -n "sunshine" | md5sum | awk '{print $1}' >> test_md5.txt
```

On macOS, replace `md5sum` with `md5 -r`:

```
echo -n "password" | md5 -r | awk '{print $1}' > test_md5.txt
echo -n "123456" | md5 -r | awk '{print $1}' >> test_md5.txt
echo -n "letmein" | md5 -r | awk '{print $1}' >> test_md5.txt
```

```
echo -n "dragon" | md5 -r | awk '{print $1}' >> test_md5.txt
echo -n "sunshine" | md5 -r | awk '{print $1}' >> test_md5.txt
```

Create a small wordlist

Create `test_wordlist.txt` with a few hundred common passwords (or use `rockyou.txt`). For quick testing, a small list is fine:

```
printf "password\n123456\nletmein\ndragon\nsunshine\nqwerty\nabc123\nmon
```

Exercise 1 — Basic Dictionary Attack

EX 1 Run your first hashcat command with `-m`, `-a`, and `-o`

Step 1: Run a basic MD5 dictionary attack

```
hashcat -m 0 -a 0 -o results_ex1.txt test_md5.txt test_wordlist.txt
```

Observe the output. Hashcat should crack all 5 hashes almost instantly.

Step 2: Inspect the results

```
cat results_ex1.txt
```

You should see lines in the format `hash:password`.

Step 3: Change the output format

Re-run with different `--outfile-format` values and compare:

```
# Password only
hashcat -m 0 -a 0 -o results_plain.txt --outfile-format 2 test_md5.txt test_wordlist.txt

# Hash only
hashcat -m 0 -a 0 -o results_hash.txt --outfile-format 1 test_md5.txt test_wordlist.txt
```

Note: Use `--potfile-disable` to prevent hashcat from skipping already-cracked hashes from the potfile.

Self-check: You should have three output files with different formats. Compare them to confirm you understand what each `--outfile-format` value produces.

Exercise 2 — Mask Attack with Incremental Mode

EX 2 Use `-a 3` with masks and the `-i` flag

Step 1: Create a hash of a short numeric password

```
# Hash the password "4521"
echo -n "4521" | md5sum | awk '{print $1}' > test_mask.txt
```

Step 2: Crack with a fixed-length mask

```
hashcat -m 0 -a 3 -o results_ex2a.txt test_mask.txt ?d?d?d?d
```

This tries all 4-digit combinations (0000–9999). Should crack in under a second.

Step 3: Use incremental mode

Now hash a password of unknown length and use increment:

```
# Hash "83"
echo -n "83" | md5sum | awk '{print $1}' > test_incr.txt

# Crack with increment from 1 to 6 digits
hashcat -m 0 -a 3 -i --increment-min 1 --increment-max 6 \
-o results_ex2b.txt test_incr.txt ?d?d?d?d?d?d
```

Step 4: Observe the keyspace

Watch the hashcat status output. Notice how it processes length 1 first (?d), then length 2 (?d?d), and so on. Press `s` during execution to see status.

Self-check: Both hashes should be cracked. Confirm that the incremental mode found "83" by trying lengths 1 and then 2.

Exercise 3 — Dictionary Attack with Rules

EX 3 Apply rule files with `-r` to expand dictionary coverage

Step 1: Create a hash that a plain dictionary won't crack

```
# "Password1" — capitalized with a digit suffix
echo -n "Password1" | md5sum | awk '{print $1}' > test_rules.txt

# "DRAGON" — all uppercase
echo -n "DRAGON" | md5sum | awk '{print $1}' >> test_rules.txt

# "sunshine123" — word + digits
echo -n "sunshine123" | md5sum | awk '{print $1}' >> test_rules.txt
```

Step 2: Try without rules (will likely fail)

```
hashcat -m 0 -a 0 -o results_ex3a.txt test_rules.txt test_wordlist.txt --
```

Check: did it crack all three? Probably not — the wordlist has "password" but not "Password1".

Step 3: Add the best64 rule file

```
hashcat -m 0 -a 0 -r rules/best64.rule -o results_ex3b.txt \
test_rules.txt test_wordlist.txt --potfile-disable
```

The rules mutate each word (capitalize, append digits, toggle case, etc.). Check if more hashes were cracked.

Step 4: Compare results

```
echo "=== Without rules ==="
cat results_ex3a.txt
echo "=== With rules ==="
cat results_ex3b.txt
```

Self-check: The rules-based attack should crack more (or all) of the hashes. Note which mutations were needed.

Exercise 4 — Hybrid Attack

EX 4Combine dictionary and mask with `-a 6`

Step 1: Create a hash with a word+digits pattern

```
echo -n "monkey99" | md5sum | awk '{print $1}' > test_hybrid.txt
echo -n "master42" | md5sum | awk '{print $1}' >> test_hybrid.txt
```

Step 2: Run a hybrid dictionary + mask attack

```
hashcat -m 0 -a 6 -o results_ex4.txt test_hybrid.txt test_wordlist.txt ?c
```

This appends every 2-digit combination (00–99) to every word in the wordlist.

Step 3: Verify results

```
cat results_ex4.txt
```

Both "monkey99" and "master42" should appear in the output.

Self-check: Both hashes cracked. Think about how this is more efficient than brute-forcing the entire password.

Exercise 5 — Session Management and Workload

EX 5Use `--session`, `--restore`, and `-w`

Step 1: Start a named session with a longer-running task

```
hashcat -m 0 -a 3 --session lab03test -w 1 \
-o results_ex5.txt test_md5.txt ?1?1?1?1?1
```


This tries all 6-character lowercase combinations (~309 million candidates). It may take a few seconds to a minute depending on your hardware.

Step 2: Press `q` to gracefully quit

While the attack is running, press `q` to save progress and exit.

Step 3: Resume the session

```
hashcat --session lab03test --restore
```

Notice that it picks up where it left off, not from the beginning.

Step 4: Experiment with workload profiles

Run the same attack with `-w 3` instead of `-w 1` and compare the speed (hashes/second) shown in the status output.

Self-check: Session restore should continue from where it stopped. Higher workload should show increased speed.

Exercise 6 — Build a Complete Command

EX 6 Combine all parameters into one production-quality command

Challenge

Write a single hashcat command that meets all of these requirements:

- Cracks NTLM hashes (`-m 1000`)
- Uses a mask attack (`-a 3`)
- Tries lengths 4 through 8 (incremental mode)
- Uses all printable characters (`?a`)
- Saves output as hash:password format
- Uses a named session called `ntlm_audit`
- Sets workload to high (`-w 3`)

Expected command

Try to build it yourself before revealing:

```
hashcat -m 1000 -a 3 -o ntlm_results.txt --outfile-format 3 \  
--session ntlm_audit --increment --increment-min 4 --increment-max 8 \  
-w 3 ntlm_hashes.txt ?a?a?a?a?a?a?a
```

Note: You can generate NTLM test hashes using Python:

```
import hashlib  
password = "Test"  
ntlm_hash = hashlib.new('md4', password.encode('utf-16le')).hexdigest()  
print(ntlm_hash)
```

Self-check: Your command should contain all 8 required elements. Count them off one by one.

Clean up: After completing the lab, remove your test files: `rm -f test_*.txt results_*.txt`. Also clear the potfile if desired: `rm -f hashcat.potfile`.

Homework

Practice tasks to solidify your understanding of hashcat command structure — to be completed before Lesson 04.

Submission: Document your commands, outputs, and answers in a text file or markdown document. Be ready to discuss your findings during the next class.

Task 1 — Command Construction

What to do

For each of the following scenarios, write the complete hashcat command you would use. Do not run them yet — just write the commands.

Scenarios

1. **MD5 dictionary attack** — Crack MD5 hashes in `dump.txt` using `rockyou.txt`. Save results to `cracked.txt` in hash:password format.
2. **NTLM with rules** — Crack NTLM hashes in `ad_hashes.txt` using `rockyou.txt` with the `best64.rule` rule file. Save only the plaintext passwords to `passwords.txt`.
3. **SHA-512 mask attack** — Crack SHA-512 (Unix) hashes in `shadow.txt` using a mask attack with all lowercase characters, trying lengths 5 through 10. Use high workload on GPU only.
4. **Hybrid attack** — Crack MD5 hashes in `web_hashes.txt` using a hybrid dictionary+mask attack. Use `common_words.txt` as the wordlist and append 4 digits to each word. Name the session `web_audit`.
5. **Combinator attack** — Crack NTLM hashes in `corp_hashes.txt` by combining words from `first_names.txt` and `years.txt`. Output hash:password pairs to `combined_results.txt`.

Requirements

- Each command must be syntactically correct (runnable without errors)
- Include all necessary flags for the scenario described
- Use the correct `-m` and `-a` values for each scenario

Self-check: For each command, verify: correct hash mode? Correct attack mode? Arguments in the right order? Output format specified where needed?

Task 2 — Parameter Exploration

What to do

Run `hashcat --help` and answer the following questions. Include the relevant flag or value in your answer.

Questions

1. What is the hash mode number for **bcrypt**?
2. What flag would you use to set the **maximum number of threads** on the CPU?
3. What does the `--force` flag do, and why is it generally discouraged?
4. What is the difference between `--outfile-format 3` and `--outfile-format 5`?
5. What flag shows you all detected **backend devices** (GPUs, CPUs) and their capabilities?
6. How do you tell hashcat to **skip** the first N words in a wordlist, and how do you **limit** it to process only N words?
7. What is the flag for running hashcat in **benchmark mode**?

Self-check: Each answer should reference a specific flag (e.g., `-b`, `--skip`, etc.) found in the `--help` output.

Task 3 — Keyspace Calculation

What to do

Calculate the total keyspace (number of candidate passwords) for each of the following mask configurations. Show your math.

Masks

1. `?d?d?d?d` — 4 digits
2. `?l?l?l?l?l?l` — 6 lowercase letters
3. `?u?l?l?l?l?d?d` — uppercase + 4 lowercase + 2 digits
4. `?a?a?a?a?a?a?a` — 8 printable characters
5. `?d?d?d?d?d?d` with `--increment --increment-min 4 --increment-max 6` — digits, lengths 4 through 6

Reference

- `?d` = 10 characters (0–9)
- `?l` = 26 characters (a–z)
- `?u` = 26 characters (A–Z)
- `?s` = 33 characters (special)
- `?a` = 95 characters (all printable)

Self-check: Verify your answers using `hashcat -m 0 -a 3 --keyspace [mask]`.
The `--keyspace` flag prints the total candidate count without cracking.

Task 4 — Error Diagnosis

What to do

Each of the following commands has an error. Identify the mistake and write the corrected command.

Broken commands

```
# Command A: Dictionary attack on NTLM hashes
hashcat -m 0 -a 0 -o results.txt ntlm_hashes.txt rockyou.txt
```

```
# Command B: Mask attack missing the mask argument
hashcat -m 1000 -a 3 -o results.txt hashes.txt

# Command C: Hybrid attack with wrong argument order
hashcat -m 0 -a 6 -o results.txt hashes.txt ?d?d rockyou.txt

# Command D: Incremental mode without a mask long enough
hashcat -m 0 -a 3 -i --increment-min 4 --increment-max 8 hashes.txt ?a?a?

# Command E: Restoring a session that was never named
hashcat --restore
```

For each command

1. Explain what is wrong
2. Write the corrected version
3. Explain why the fix works

Self-check: Test your corrected commands against test hashes (using the setup from the lab) to verify they work.

Grading Criteria

Task	Points	Key Criterion
Task 1 — Command Construction	25	All 5 commands are syntactically correct with proper flags
Task 2 — Parameter Exploration	25	All 7 questions answered with specific flag references
Task 3 — Keyspace Calculation	25	Correct math shown; answers verified with --keyspace

Task	Points	Key Criterion
Task 4 — Error Diagnosis	25	All 5 errors identified with corrected commands and explanations

Lesson 04: Hash Types & Identification

Hash Types & Identification

Recognizing hash formats, choosing the correct hashcat mode, and understanding the performance implications of different algorithms.

Learning Objectives

- Identify the **most common hash types** by length, format, and prefix
 - Map each hash type to the correct **hashcat -m mode**
 - Use **hashid**, **hash-identifier**, and **hashcat --example-hashes** for identification
 - Explain the difference between **salted and unsalted** hashes
 - Understand **performance differences** between fast and slow hash algorithms
-

Common Hash Types at a Glance

Algorithm	Mode (-m)	Length	Example Prefix
MD5	0	32 hex chars	5d41402abc4b...
SHA-1	100	40 hex chars	aaf4c61ddcc5...
SHA-256	1400	64 hex chars	2cf24dba5fb0...
SHA-512	1800	128 hex chars	\$6\$rounds=5000\$...
NTLM	1000	32 hex chars	a4f49c406510...
bcrypt	3200	60 chars	\$2a\$12\$... or \$2b\$...
WPA/WPA2	22000	varies	WPA*02*...

Key insight: Hash length alone can narrow possibilities dramatically. 32 hex chars = MD5 or NTLM. 40 hex chars = almost certainly SHA-1.

Identifying Hashes by Format

Many hash formats include **prefixes** that make identification trivial. Look for these patterns before reaching for any tool.

Signature Prefixes

- `$2a$` / `$2b$` / `$2y$` — bcrypt (mode 3200). The number after the second `$` is the cost factor.
- `$6$` — SHA-512crypt (mode 1800). Common in Linux `/etc/shadow`.
- `$5$` — SHA-256crypt (mode 7400).
- `$1$` — MD5crypt (mode 500). Legacy Linux password hash.
- `$apr1$` — Apache MD5 (mode 1600).
- `WPA*02*` — WPA/WPA2 PMKID or handshake (mode 22000).

When there is no prefix and the hash is pure hex, count the characters to determine the algorithm family.

Identification Tools

When manual inspection is not enough, use dedicated tools to narrow down the hash type.

hashid

```
# Install
pip install hashid

# Identify a hash
hashid '5d41402abc4b2a76b9719d911017c592'
# Output: [+] MD5, [+] MD4, [+] NTLM ...
```

hash-identifier

```
# Included in Kali Linux, or install from GitHub
hash-identifier
# Paste the hash when prompted — it suggests possible algorithms
```

hashcat --example-hashes

```
# Search for a specific pattern
hashcat --example-hashes | grep -i 'bcrypt' -B2 -A3

# List all modes with example hashes
hashcat --example-hashes | less
```

Pro tip: `hashcat --example-hashes` is the most authoritative source. It shows the exact format hashcat expects for each mode number.

Salted vs Unsalted Hashes

A **salt** is random data appended or prepended to the password before hashing. It ensures identical passwords produce different hashes.

Why Salts Matter

- **Unsalted:** `MD5("password")` always produces the same hash. Vulnerable to rainbow tables and precomputed lookups.
- **Salted:** `MD5(salt + "password")` produces a unique hash per salt value. Rainbow tables become useless.

How Hashcat Handles Salts

```
# Unsalted MD5 (-m 0): just the hash
5d41402abc4b2a76b9719d911017c592

# Salted MD5 (-m 10): hash:salt format
67b3dba8bc6e4f2a9e1b7c23f8d14e5a:s4lt

# SHA-512crypt (-m 1800): salt embedded in the hash string
$6$rounds=5000$saltsalt$hashvalue...
```

Important: You must supply the salt in the correct format, or hashcat will fail silently. Always check `--example-hashes` for the expected format of salted modes.

Performance: Fast vs Slow Hashes

Hash algorithm choice has an **enormous** impact on cracking speed. This is the single biggest factor in attack feasibility.

Algorithm	Approximate Speed (RTX 4090)	Category
MD5	~164 GH/s	Fast
NTLM	~300 GH/s	Fast
SHA-1	~26 GH/s	Fast
SHA-256	~10 GH/s	Medium
SHA-512crypt	~2.4 MH/s	Slow
bcrypt (cost 12)	~105 kH/s	Very Slow
WPA/WPA2	~1.6 MH/s	Slow

Takeaway: MD5 and NTLM are "fast hashes" — billions of guesses per second. bcrypt is deliberately slow — purpose-built to resist brute-force attacks. Adjust your wordlists and strategy accordingly.

Choosing Your Attack Strategy by Hash Type

The hash type dictates the feasible attack space. Plan your strategy accordingly.

Fast Hashes (MD5, NTLM, SHA-1)

- Full brute-force feasible up to ~8 chars
- Large wordlists + complex rules are practical
- Combinator and hybrid attacks are fast
- Mask attacks with full charset work well

Slow Hashes (bcrypt, SHA-512crypt, WPA)

- Brute-force beyond 6 chars is impractical
- Use small, targeted wordlists
- Focus on common passwords and patterns
- Rules should be minimal and precise

Notes: Always benchmark with `hashcat -b -m MODE` before committing to a long attack. Know your speed before choosing your strategy.

Lab & Homework

Lab Exercises

1. **Hash Identification:** Identify 10 different hashes by type using manual inspection, hashid, and hashcat --example-hashes
2. **Crack MD5 & SHA-1:** Run dictionary attacks on unsalted fast hashes and compare speeds
3. **Crack bcrypt:** Attack a bcrypt hash with a small wordlist and observe the speed difference

Homework Tasks

1. Research and document **5 additional hash types** not covered in class, with their hashcat modes
2. Build a **hash identification cheat sheet** covering length, prefix, and recommended attack strategy
3. Benchmark **3 different hash types** on your hardware and calculate maximum brute-force keyspace

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

Lab Guide

Three exercises: identify hash types, crack fast hashes, and attack a slow hash.

Prerequisites: Hashcat installed and working (verify with `hashcat --version`).
A wordlist available — `rockyou.txt` recommended. Install `hashid` with `pip install hashid`.

Exercise 1 — Hash Identification Challenge

EX 1 Identify 10 hashes using manual inspection and tools

Below are 10 hashes. For each one, determine the algorithm, the hashcat mode number, and whether it is salted. Use a combination of manual inspection (length, prefix) and automated tools.

The hashes

```
# Hash 1
5d41402abc4b2a76b9719d911017c592

# Hash 2
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

# Hash 3
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy

# Hash 4
$6$rounds=5000$saltsalt$kpa26CjNg2wMvYSGPHPNeq2VJoxTkN/0fPDJFSyYRwOuAgiNI

# Hash 5
a4f49c406510bdcab6824ee7c30fd852

# Hash 6
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```



```
# Hash 7
$1$salt$qJH7.N4xYta3aEG/dfqo/0

# Hash 8
482c811da5d5b4bc6d497ffa98491e38:salt123

# Hash 9
$apr1$71850310$qh9m4xcAn3MGxoqwX/ztb.
```

Step 1: Manual inspection

For each hash, note the following:

- Does it have a prefix (`$2a$` , `$6$` , `$1$` , `$apr1$` , etc.)?
- How many characters is it? (Count only the hex/base64 portion)
- Is it pure hex, or does it contain other characters?
- Does it include a salt (colon-separated or embedded)?

Step 2: Verify with hashid

```
# Run hashid with -m flag to see hashcat modes
hashid -m '5d41402abc4b2a76b9719d911017c592'
hashid -m '$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcf17p92ldGxad68LJZdL17lhWy'
hashid -m '2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824'
```

Step 3: Confirm with hashcat

```
# Verify the format matches what hashcat expects
hashcat --example-hashes | grep -A4 'MODE: 0'
hashcat --example-hashes | grep -A4 'MODE: 3200'
```

Fill in the answer table

#	Algorithm	Mode (-m)	Salted?	How You Know
1				
2				
3				

#	Algorithm	Mode (-m)	Salted?	How You Know
4				
5				
6				
7				
8				
9				
10				

Expected answers: 1=MD5 (0), 2=SHA-1 (100), 3=bcrypt (3200), 4=SHA-512crypt (1800), 5=MD5 or NTLM (0 or 1000), 6=SHA-256 (1400), 7=MD5crypt (500), 8=md5(\$pass.\$salt) (10), 9=Apache MD5 (1600), 10=SHA-512 (1700).

Exercise 2 — Cracking Fast Hashes

EX 2 Run dictionary attacks on MD5 and SHA-1, compare speeds

Create test hashes and crack them with hashcat. Compare the cracking speed between hash types.

Step 1: Generate test hashes

```
# Generate MD5 hashes of known passwords
echo -n "password" | md5sum | cut -d' ' -f1 > md5_hashes.txt
echo -n "123456" | md5sum | cut -d' ' -f1 >> md5_hashes.txt
echo -n "letmein" | md5sum | cut -d' ' -f1 >> md5_hashes.txt
echo -n "dragon" | md5sum | cut -d' ' -f1 >> md5_hashes.txt
echo -n "monkey" | md5sum | cut -d' ' -f1 >> md5_hashes.txt

# Generate SHA-1 hashes of the same passwords
echo -n "password" | shasum | cut -d' ' -f1 > sha1_hashes.txt
echo -n "123456" | shasum | cut -d' ' -f1 >> sha1_hashes.txt
echo -n "letmein" | shasum | cut -d' ' -f1 >> sha1_hashes.txt
```

```

echo -n "dragon" | shasum | cut -d' ' -f1 >> sha1_hashes.txt
echo -n "monkey" | shasum | cut -d' ' -f1 >> sha1_hashes.txt

# Generate SHA-256 hashes
echo -n "password" | sha256sum | cut -d' ' -f1 > sha256_hashes.txt
echo -n "123456" | sha256sum | cut -d' ' -f1 >> sha256_hashes.txt
echo -n "letmein" | sha256sum | cut -d' ' -f1 >> sha256_hashes.txt
echo -n "dragon" | sha256sum | cut -d' ' -f1 >> sha256_hashes.txt
echo -n "monkey" | sha256sum | cut -d' ' -f1 >> sha256_hashes.txt

```

Step 2: Crack each set with rockyou.txt

```

# Crack MD5
hashcat -m 0 -a 0 md5_hashes.txt /path/to/rockyou.txt --force
# Note the speed (Speed.#1) and time to complete

# Crack SHA-1
hashcat -m 100 -a 0 sha1_hashes.txt /path/to/rockyou.txt --force
# Note the speed and time to complete

# Crack SHA-256
hashcat -m 1400 -a 0 sha256_hashes.txt /path/to/rockyou.txt --force
# Note the speed and time to complete

```

Step 3: Record and compare

Hash Type	Mode	Speed (H/s)	Time to Crack All 5	All Found?
MD5	0			
SHA-1	100			
SHA-256	1400			

Self-check: All 5 passwords should be found for each hash type. MD5 should be fastest, SHA-256 slowest. How many times faster is MD5 compared to SHA-256?

Exercise 3 — Cracking a Slow Hash (bcrypt)

EX 3 Attack a bcrypt hash and observe the dramatic speed difference

Generate a bcrypt hash and attempt to crack it. This exercise demonstrates why slow hashes are effective at resisting brute-force attacks.

Step 1: Generate a bcrypt hash

```
# Using Python to generate a bcrypt hash of "sunshine"
python3 -c "
import bcrypt
password = b'sunshine'
salt = bcrypt.gensalt(rounds=10)
hashed = bcrypt.hashpw(password, salt)
print(hashed.decode())
" > bcrypt_hash.txt

cat bcrypt_hash.txt
# Output: $2b$10$... (60-character bcrypt hash)
```

Step 2: Create a small targeted wordlist

```
# Create a small wordlist (bcrypt is too slow for rockyou.txt)
cat > small_wordlist.txt <<'EOF'
password
123456
sunshine
letmein
football
monkey
shadow
master
dragon
qwerty
EOF
```

Step 3: Crack the bcrypt hash

```
# Crack bcrypt with the small wordlist
hashcat -m 3200 -a 0 bcrypt_hash.txt small_wordlist.txt --force

# Compare: how long would rockyou.txt take?
# Check the estimated time if you started with the full list:
hashcat -m 3200 -a 0 bcrypt_hash.txt /path/to/rockyou.txt --force
```

Step 4: Benchmark comparison

```
# Run benchmarks for direct comparison
hashcat -b -m 0      # MD5 benchmark
hashcat -b -m 3200   # bcrypt benchmark

# Calculate the speed ratio
# MD5 speed / bcrypt speed = how many times faster MD5 is
```

Expected result: MD5 will be roughly 1,000,000x faster than bcrypt. The bcrypt hash should crack in seconds with the small wordlist but would take hours or days with rockyou.txt. This is exactly why bcrypt exists — it makes brute-force impractical.

Self-check: Did hashcat find "sunshine"? What was the bcrypt speed in H/s compared to the MD5 speed? Calculate the ratio and think about what this means for password security.

Homework

Research, identification, and benchmarking tasks to deepen your hash type knowledge.

Submission: Document your findings in a text file or PDF. Include screenshots of hashcat output where requested. Be ready to present your cheat sheet in the next class.

Task 1 — Research 5 Additional Hash Types

What to do

Research and document 5 hash types **not covered in the lecture**. For each hash type, provide all the information listed below.

Requirements

For each of the 5 hash types, document:

- **Algorithm name** (e.g., Keccak-256, RIPEMD-160, Whirlpool)
- **Hashcat mode number** (`-m` value)
- **Hash length and format** (hex length, any prefix, encoding)
- **Where it is commonly found** (which applications, operating systems, or protocols use it)
- **Whether it is salted** and the salt format if applicable
- **An example hash** (from `hashcat --example-hashes`)

Suggested hash types to research

- PBKDF2-HMAC-SHA256 (mode 10900)
- Kerberos 5 TGS-REP (mode 13100)
- MySQL 4.1+ (mode 300)
- MSSQL 2012/2014 (mode 1731)

- Drupal7 (mode 7900)
- LM (mode 3000)
- NetNTLMv2 (mode 5600)
- scrypt (mode 8900)

```
# Use this command to find example hashes for any mode
hashcat --example-hashes | grep -A6 'MODE: 10900'
```

```
# Or search by name
```

```
hashcat --example-hashes | grep -i 'kerberos' -B2 -A6
```

Self-check: Can you explain to a teammate why each hash type is used in its specific context? Do you know how salting works for each one?

Task 2 — Build a Hash Identification Cheat Sheet

What to do

Create a one-page reference document (table format recommended) that you can use during CTF competitions or penetration tests to quickly identify an unknown hash.

Requirements

Your cheat sheet must include at least 15 hash types and these columns:

Algorithm	Mode	Length	Prefix	Salted	Speed	Recommended Attack
MD5	0	32 hex	none	No	Fast	Large wordlist + rules
bcrypt	3200	60 chars	\$2a\$ / \$2b\$	Yes	Very Slow	Small targeted wordlist

... continue for at least 13 more types ...

Bonus: Decision flowchart

Optionally, create a text-based decision flowchart for hash identification:

```
Unknown hash
├─ Starts with $?
│   ├── $2a$ / $2b$ / $2y$ → bcrypt (3200)
│   ├── $6$ → SHA-512crypt (1800)
│   ├── $5$ → SHA-256crypt (7400)
│   ├── $1$ → MD5crypt (500)
│   └── $apr1$ → Apache MD5 (1600)
├─ Starts with WPA*?
│   └── WPA/WPA2 (22000)
├─ Pure hex?
│   ├── 32 chars → MD5 (0) or NTLM (1000)
│   ├── 40 chars → SHA-1 (100)
│   ├── 64 chars → SHA-256 (1400)
│   └── 128 chars → SHA-512 (1700)
└─ Contains colon (:)?
    └── hash:salt → check salted variants
```

Self-check: Does your cheat sheet cover every hash type from the lecture plus 5 from Task 1? Could a teammate use it without any additional explanation?

Task 3 — Benchmark and Feasibility Analysis

What to do

Run hashcat benchmarks on your hardware for 3 different hash types (one fast, one medium, one slow). Then calculate the maximum brute-force keyspace you can exhaust in 24 hours for each.

Step 1: Run benchmarks

```
# Benchmark three hash types
hashcat -b -m 0          # MD5 (fast)
hashcat -b -m 1400       # SHA-256 (medium)
hashcat -b -m 3200       # bcrypt (slow)
```



```
# Record the Speed.#1 value for each
```

Step 2: Calculate keyspace feasibility

For each hash type, calculate the maximum brute-force keyspace you can cover in 24 hours (86,400 seconds):

```
# Formula
```

```
Max keyspace in 24h = Speed (H/s) × 86,400
```

```
# Then determine the maximum password length you can fully brute-force:
```

```
# Lowercase only (26 chars): 26^N candidates for N-length passwords
```

```
# Lowercase + digits (36 chars): 36^N candidates
```

```
# Full printable ASCII (95 chars): 95^N candidates
```

Fill in the results table

Hash Type	Speed (H/s)	24h Keyspace	Max Length (a-z)	Max Length (a-z0-9)	Max Length (all printable)
MD5					
SHA-256					
bcrypt					

Step 3: Write a brief analysis

Answer these questions in 2-3 sentences each:

1. For MD5, what is the longest all-lowercase password you can brute-force in 24 hours? What about with the full printable charset?
2. For bcrypt, is brute-force of even a 6-character lowercase password feasible in 24 hours? Show the math.
3. Based on your results, what is the minimum password length you would recommend for systems using bcrypt? For systems using unsalted MD5?

Self-check: Do your numbers make sense? MD5 should allow brute-forcing much longer passwords than bcrypt. If bcrypt brute-force of 8+ chars is

feasible on your hardware in 24h, double-check your math.

Grading Criteria

Task	Points	Key Criterion
Task 1 — Research	30	5 hash types fully documented with all 6 required fields
Task 2 — Cheat sheet	35	15+ hash types, all columns filled, usable as a quick reference
Task 3 — Benchmarks	35	Correct calculations, reasonable analysis, screenshots of benchmark output

Lesson 05: Dictionary & Combinator Attacks

Dictionary & Combinator Attacks

The most practical and widely-used hashcat attack modes. Dictionary attacks leverage real-world wordlists, while combinator attacks multiply their power by joining words from two lists.

Learning Objectives

- Run a **dictionary attack** (`-a 0`) using popular wordlists
 - Apply **rule-based mutations** to expand wordlist coverage
 - Execute a **combinator attack** (`-a 1`) joining two dictionaries
 - Identify and obtain **popular wordlists** for security auditing
 - Understand when to choose dictionary vs. combinator vs. other attack modes
-

Dictionary Attack (`-a 0`)

The dictionary (or "straight") attack tries every word in a wordlist file against the target hashes. It is attack mode `0` and is hashcat's default.

```
# Basic dictionary attack against MD5 hashes  
hashcat -m 0 -a 0 -o results.txt hashes.txt rockyou.txt
```

Parameter Breakdown

- `-m 0` — hash type: MD5
- `-a 0` — attack mode: dictionary (straight)
- `-o results.txt` — output cracked results to file
- `hashes.txt` — file containing target hashes
- `rockyou.txt` — the wordlist to use

Speed: Dictionary attacks are fast because they only check words that real humans actually use as passwords. With a GPU, hashcat can check billions of MD5 candidates per second.

Rule-Based Attacks

Rules transform each dictionary word into multiple candidates — capitalizing, appending numbers, substituting characters, reversing, and more. This dramatically increases coverage without a larger wordlist.

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

How Rules Work

For each word in the dictionary, hashcat applies every rule in the file. If `rockyou.txt` has 14 million words and `best64.rule` has 64 rules, hashcat generates **~900 million candidates**.

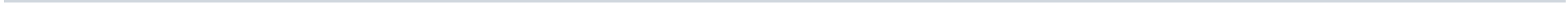
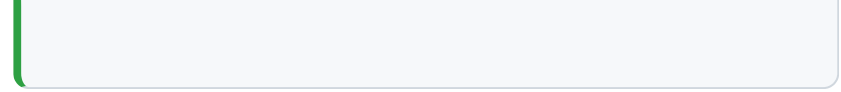
Example Mutations

- `password` → `Password` (capitalize)
- `password` → `password1` (append digit)
- `password` → `p@ssword` (leet-speak)

Built-in Rule Files

- `best64.rule` — 64 high-yield rules
- `d3ad0ne.rule` — ~35K rules
- `dive.rule` — ~99K rules

- `password` → `drowssap` (reverse)



Combinator Attack (`-a 1`)

The combinator attack concatenates every word from the first dictionary with every word from the second dictionary. This catches multi-word passwords like `solarwind` or `bluedragon`.

```
# Combinator attack: dict1 x dict2
hashcat -m 0 -a 1 hashes.txt dict1.txt dict2.txt
```

Candidate Generation

dict1.txt	dict2.txt	Candidates Generated
sun	flower	sunflower
sun	light	sunlight
blue	flower	blueflower
blue	light	bluelight

Candidate count: If dict1 has N words and dict2 has M words, the combinator produces $N \times M$ candidates. $10,000 \times 10,000 = 100$ million candidates.

Popular Wordlists

The quality of a dictionary attack depends entirely on the wordlist. These are the most widely used in the security community.

Wordlist	Size	Source	Notes
rockyou.txt	~14M passwords	2009 RockYou breach	De facto standard; ships with Kali Linux
SecLists	Varies	Daniel Miessler (GitHub)	Curated collection of many wordlists
CrackStation	~1.5B entries	crackstation.net	Massive; 15 GB uncompressed
Custom lists	Varies	You create them	Target-specific: company names, keywords

Tip: Always start with `rockyou.txt` + `best64.rule`. This combination cracks a surprising percentage of weak passwords in minutes.

Attack Strategy Flowchart

Choose your attack mode based on what you know about the target passwords.

Decision Flow

1. **Start simple:** Dictionary attack with `rockyou.txt` (`-a 0`)
2. **Add mutations:** Re-run with `-r rules/best64.rule`
3. **Try combinations:** Combinator attack (`-a 1`) with themed word lists
4. **Escalate rules:** Use `d3ad0ne.rule` or `dive.rule` for deeper coverage
5. **Go broader:** Larger wordlists (CrackStation) or mask/hybrid attacks (Lesson 06)

Real-world stat: Dictionary attacks with rules typically crack 60-80% of password hashes from typical database breaches. Most users pick common words with predictable modifications.

Live Demo

Cracking two well-known MD5 hashes with a dictionary attack.

Demo Steps

1. Create `hashes.txt` with two MD5 hashes:

```
5f4dcc3b5aa765d61d8327deb882cf99  
e10adc3949ba59abbe56e057f20f883e
```

2. Run: `hashcat -m 0 -a 0 -o results.txt hashes.txt rockyou.txt`

3. Check `results.txt` — should find **"password"** and **"123456"**

4. Re-run with rules: `hashcat -m 0 -a 0 -r rules/best64.rule hashes.txt rockyou.txt`

Note: These hashes are intentionally trivial — they exist in every wordlist. In real audits, the easy cracks come first and harder passwords require rules and combinator attacks.

Lab & Homework

Lab 2 — Dictionary Attack

1. Download `rockyou.txt` wordlist
2. Create `hashes.txt` with two known MD5 hashes
3. Run a straight dictionary attack and verify results
4. Re-run with `best64.rule` and compare performance

Homework Tasks

1. Benchmark dictionary vs. rules vs. combinator on provided hashes
2. Create a custom wordlist from a topic and test it
3. Research and document the rule syntax for 5 custom rules

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 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](#)

Lab Guide — Lab 2

Hands-on dictionary attack: download a wordlist, crack known hashes, and explore rule-based mutations.

Prerequisites

- Hashcat installed and working (verified in Lab 1)
- Terminal / command-line access
- Completed the [Theory](#) reading for this lesson
- Internet connection (to download rockyou.txt if needed)

Ethics reminder: Only crack hashes that you have created yourself or have explicit authorization to test. The hashes in this lab are well-known test values.

Step 1 — Download the rockyou.txt Wordlist

Task 1.1 — Obtain rockyou.txt

The rockyou.txt wordlist is the standard starting point for dictionary attacks. Choose one of the following methods:

Option A: Kali Linux (pre-installed)

```
# rockyou.txt ships compressed with Kali
sudo gunzip /usr/share/wordlists/rockyou.txt.gz
ls -lh /usr/share/wordlists/rockyou.txt
# Expected: ~139 MB, ~14.3 million lines
```

Option B: Download from the internet

```
# Download from a common mirror
wget https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt
```

```
# Or use curl:
curl -L -o rockyou.txt https://github.com/brannondorsey/naive-hashcat/rel
```

Option C: Create a minimal test wordlist

If you cannot download rockyou.txt, create a small test wordlist that includes the target passwords:

```
cat << 'EOF' > test_wordlist.txt
password
123456
admin
letmein
welcome
monkey
dragon
master
qwerty
login
EOF
```

Checkpoint: Verify your wordlist exists and has content.

```
wc -l rockyou.txt
# Expected: 14344391 (or 10 lines if using the mini test list)
```

Step 2 — Create the Target Hash File

Task 2.1 — Create hashes.txt

Create a file containing two MD5 hashes. These are the MD5 hashes of commonly known passwords:

```
cat << 'EOF' > hashes.txt
5f4dcc3b5aa765d61d8327deb882cf99
e10adc3949ba59abbe56e057f20f883e
EOF
```

For reference, these hashes correspond to:

Hash	Expected Password
5f4dcc3b5aa765d61d8327deb882cf99	password
e10adc3949ba59abbe56e057f20f883e	123456

Checkpoint: Verify the hash file.

```
cat hashes.txt
# Should show exactly 2 lines, each a 32-character hex string
```

Step 3 — Run the Dictionary Attack

Task 3.1 — Execute the attack

Run hashcat in dictionary mode against your hashes:

```
hashcat -m 0 -a 0 -o results.txt hashes.txt rockyou.txt
```

Parameter review:

- `-m 0` — MD5 hash type
- `-a 0` — Dictionary (straight) attack mode
- `-o results.txt` — Write cracked pairs to this file
- `hashes.txt` — Input hashes
- `rockyou.txt` — Wordlist

If you get a message that the hashes have already been cracked (from hashcat's potfile cache), use:

```
# Force re-crack by ignoring the potfile
hashcat -m 0 -a 0 -o results.txt --potfile-disable hashes.txt rockyou.txt
```

Checkpoint: The attack should complete in seconds. Both hashes should be cracked.

Task 3.2 — Examine the results

```
cat results.txt
```

Expected output:

```
5f4dcc3b5aa765d61d8327deb882cf99:password  
e10adc3949ba59abbe56e057f20f883e:123456
```

The format is `hash:password`. Both passwords are among the most common in any breach dataset, so they are found almost instantly.

Task 3.3 — Review hashcat status output

While hashcat runs (or after, using `--show`), note these key metrics:

- **Speed:** How many hashes per second (H/s) is your system achieving?
- **Progress:** How many words from the wordlist were tested?
- **Recovered:** How many of the 2 hashes were cracked?

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

Step 4 — Run with Rules

Task 4.1 — Add best64.rule

Re-run the attack with the `best64.rule` file to see the effect of rule-based mutations:

```
# Clear the potfile first so hashes are re-cracked
hashcat -m 0 -a 0 -r rules/best64.rule -o results_rules.txt --potfile-dis
```

Observe the difference in candidate count and speed compared to the plain dictionary attack.

Where are the rules? Rule files are in hashcat's installation directory under `rules/`. On Kali Linux: `/usr/share/hashcat/rules/`. On macOS with Homebrew: `/opt/homebrew/share/hashcat/rules/` or `/usr/local/share/hashcat/rules/`. You can also find the path with: `find / -name "best64.rule" 2>/dev/null`.

Task 4.2 — Compare performance

Fill in this comparison based on your observations:

Metric	Plain Dictionary	Dictionary + best64.rule
Total candidates	~14.3M	~14.3M x 64 = ~916M
Time to crack both hashes	(your value)	(your value)
Hash speed (H/s)	(your value)	(your value)

For these trivial hashes, both approaches crack them instantly. The real advantage of rules appears with more complex passwords like "Password1!" or "p@ssw0rd".

Step 5 (Bonus) — Combinator Attack

Task 5.1 — Create test dictionaries

```
# Create two small dictionaries
cat << 'EOF' > colors.txt
red
blue
green
dark
light
EOF

cat << 'EOF' > things.txt
sky
dragon
fire
wolf
star
EOF
```

Task 5.2 — Generate a test hash

```
# Generate MD5 hash of "bluedragon"
echo -n "bluedragon" | md5sum
# Expected: 53153a856798e842d84e5be22a4b8e37
```

```
# On macOS, use md5 instead:
echo -n "bluedragon" | md5
# Or: md5 -s "bluedragon"
```

```
# Save the hash
echo "53153a856798e842d84e5be22a4b8e37" > combo_hash.txt
```

Task 5.3 — Run the combinator attack

```
hashcat -m 0 -a 1 -o combo_results.txt --potfile-disable combo_hash.txt c
cat combo_results.txt
# Expected: 53153a856798e842d84e5be22a4b8e37:bluedragon
```

Checkpoint: The combinator attack should find "bluedragon" by joining "blue" from the first dictionary with "dragon" from the second.

Completion Checklist

- rockyou.txt wordlist obtained and verified
- hashes.txt created with 2 MD5 hashes
- Dictionary attack executed successfully; found "password" and "123456"
- results.txt contains both cracked hash:password pairs
- Rule-based attack executed with best64.rule
- Performance comparison table filled in
- (Bonus) Combinator attack cracked "bluedragon"

Homework

Three tasks to deepen your understanding of dictionary attacks, rule writing, and combinator strategies.

Complete [Lab 2](#) before starting the homework. These tasks build on the skills and files from the lab.

Task 1 — Attack Mode Benchmark

Objective

Benchmark three attack approaches against a set of hashes and compare their effectiveness and speed.

Setup

Create a file `hw_hashes.txt` with the following MD5 hashes:

```
5f4dcc3b5aa765d61d8327deb882cf99
e10adc3949ba59abbe56e057f20f883e
d8578edf8458ce06fbc5bb76a58c5ca4
0d107d09f5bbe40cade3de5c71e9e9b7
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
```

The last hash is SHA-1 (`-m 100`). The others are MD5 (`-m 0`). Run each attack type separately for MD5 hashes first.

Instructions

- Plain dictionary:** Run `hashcat -m 0 -a 0 --potfile-disable hw_hashes.txt rockyou.txt`. Record: time, speed, hashes cracked.
- Dictionary + rules:** Run with `-r rules/best64.rule`. Record the same metrics.

3. **Combinator:** Split `rockyou.txt` into two halves (first 7M lines and last 7M lines) and run a combinator attack. Record the metrics. Note: this generates a very large candidate space — you may want to use smaller subsets.

Deliverable

A comparison table:

Attack Mode	Candidates	Time	Speed (H/s)	Hashes Cracked
Dictionary (<code>-a 0</code>)	(your value)	(your value)	(your value)	(your value)
Dictionary + best64.rule	(your value)	(your value)	(your value)	(your value)
Combinator (<code>-a 1</code>)	(your value)	(your value)	(your value)	(your value)

Write a short paragraph (100-150 words) explaining which approach you would use first in a real audit and why.

Task 2 — Custom Wordlist Creation

Objective

Create a targeted custom wordlist for a fictional company and test it against generated hashes.

Instructions

1. Choose a fictional company name (e.g., "SolarTech") and create a wordlist of 50+ entries that an employee might use as a password. Include:
 - Company name variations (solartech, SolarTech, SOLARTECH, solar, tech)
 - Product names (make up 2-3)
 - City where the company is "based"

- Common suffixes: numbers (1-99, 2024, 2025, 2026), special characters (!, @, #)
- Seasonal patterns (Summer2025, Winter2026)

2. Generate MD5 hashes of 5 passwords from your wordlist:

```
echo -n "SolarTech2025" | md5sum  
echo -n "solartech!" | md5sum  
# ... etc.
```

3. Run a dictionary attack using your custom wordlist against those hashes.

4. Run the same hashes against `rockyou.txt`. How many does rockyou crack?

Deliverable

Your custom wordlist file, the generated hashes, and a written comparison: how does a targeted wordlist perform vs. a generic one for organization-specific passwords?

Tip: In real-world audits, custom wordlists combined with rules are often more effective than massive generic wordlists for cracking organization-specific passwords.

Task 3 — Rule Syntax Deep Dive

Objective

Learn hashcat's rule syntax by writing and testing 5 custom rules.

Instructions

1. Read the [hashcat rule syntax documentation](#).
2. Create a file called `custom5.rule` containing exactly 5 rules. Each rule must use a different operation. Suggested operations:
 - Capitalize first letter + append a year (`c $2 $0 $2 $6`)
 - Leet-speak substitutions (`sa@ se3 sil so0`)

- Reverse + uppercase all (`r u`)
- Duplicate the word + append "!" (`d $!`)
- Truncate to first 6 characters + append "123" (`]]]]] $1 $2 $3` — note: `]` removes the last character; adjust count based on word length, or use `'6` to truncate at position 6)

3. Test your rules by running:

```
# Preview what rules generate (without cracking)
echo "password" | hashcat -r custom5.rule --stdout
```

4. Document each rule: explain what it does and give an example of the input/output transformation.

Deliverable

The `custom5.rule` file and a table documenting each rule:

#	Rule Code	Description	Input	Output
1	<code>c \$2 \$0 \$2</code> <code>\$6</code>	Capitalize + append "2026"	password	Password2026
2	(your rule)	(your description)	(input)	(output)
3	(your rule)	(your description)	(input)	(output)
4	(your rule)	(your description)	(input)	(output)
5	(your rule)	(your description)	(input)	(output)

Submission Checklist

- Task 1: Benchmark table with 3 attack modes + written analysis
- Task 2: Custom wordlist (50+ entries) + hash comparison results + written comparison
- Task 3: `custom5.rule` file created + documentation table with 5 rules + tested with `--stdout`

- All commands ran successfully and results are documented

Lesson 06: Mask & Hybrid Attacks

Mask & Hybrid Attacks

Move beyond wordlists. When you know the password structure, mask attacks let you define exact character patterns. Hybrid attacks combine wordlists with masks for even more power.

Learning Objectives

- Understand **mask attack mode** (`-a 3`) and built-in charset placeholders
 - Write masks that match known password patterns (PINs, names+digits, etc.)
 - Use **incremental mode** (`-i`) to sweep across multiple lengths
 - Define **custom charsets** with `-1`, `-2`, `-3`, `-4`
 - Run **hybrid attacks** (`-a 6` and `-a 7`) combining wordlists with masks
-

Mask Attack Basics (`-a 3`)

A mask attack generates candidates based on character position patterns. Instead of reading from a wordlist, Hashcat builds every possible combination matching the mask.

Built-in Charset Placeholders

Placeholder	Character Set	Example Characters
<code>?l</code>	Lowercase letters	<code>a-z</code>
<code>?u</code>	Uppercase letters	<code>A-Z</code>
<code>?d</code>	Digits	<code>0-9</code>
<code>?s</code>	Special characters	<code>!@#\$%^&*()...</code>
<code>?a</code>	All printable ASCII	<code>?l?u?d?s</code> combined
<code>?b</code>	Full byte range	<code>0x00-0xff</code> (256 values)

Each placeholder represents **one character position** in the candidate password.

Mask Examples

Each position in the mask defines what characters are allowed at that position.

4-Digit PIN

Mask: `?d?d?d?d`

Generates: 0000 through 9999

Keyspace: **10,000** candidates

Name + Digits Pattern

Mask: `?u?l?l?l?d?d`

Example: `Alex42`, `John99`

Keyspace: **26 x 26 x 26 x 26 x 10 x 10 =
45,697,600**

Any 4 Characters

Mask: `?a?a?a?a`

All printable 4-char passwords

Command

```
hashcat -m 0 -a 3 -o results.txt \  
hashes.txt ?d?d?d?d
```

Keyspace: $95^4 = 81,450,625$

Incremental Mode (`-i`)

When you don't know the exact password length, incremental mode gradually increases the mask length from a minimum to a maximum.

```
# Try all printable passwords from 4 to 8 characters
hashcat -m 0 -a 3 -i --increment-min 4 --increment-max 8 \
hashes.txt ?a?a?a?a?a?a?a
```

How It Works

- Starts at length `--increment-min` (default: 1)
- Runs the mask truncated to that length
- Increments by 1 and repeats until `--increment-max`
- The mask you provide must be at least as long as `--increment-max`

Warning: Incremental mode with `?a` grows exponentially. At 8 characters that is $95^8 = 6.6$ trillion

candidates. Use more constrained charsets when possible.

Custom Charsets

When built-in placeholders are too broad or too narrow, define your own charsets with `-1` through `-4`.

```
# Custom charset 1: lowercase + digits
hashcat -m 0 -a 3 -1 ?l?d hashes.txt ?1?1?1?1?1?1

# Custom charset mixing: uppercase first, then custom
hashcat -m 0 -a 3 -1 ?l?d -2 ?u?s \
hashes.txt ?2?1?1?1?1?d

# Literal characters in a custom charset
hashcat -m 0 -a 3 -1 abc123 hashes.txt ?1?1?1?1
```

Custom Charset Slots

- `-1 <charset>` — referenced as `?1`
- `-2 <charset>` — referenced as `?2`
- `-3 <charset>` — referenced as `?3`
- `-4 <charset>` — referenced as `?4`

You can combine built-in placeholders and literal characters in a single charset definition.

Hybrid Attacks

Hybrid attacks combine a wordlist with a mask, covering the common pattern of users appending or prepending characters to dictionary words.

Mode 6: Wordlist + Mask

Appends mask characters **after** each dictionary word.

```
# password123, summer99, etc.  
hashcat -m 0 -a 6 \  
    hashes.txt rockyou.txt ?d?d?d
```

If rockyou has "password", generates:
password000 through password999

Mode 7: Mask + Wordlist

Prepends mask characters **before** each dictionary word.

```
# 123password, 42summer, etc.  
hashcat -m 0 -a 7 \  
    hashes.txt ?d?d?d rockyou.txt
```

If rockyou has "password", generates:
000password through 999password

Tip: Hybrid attacks are extremely effective because many users simply add digits or special characters to the end of a common word.

Attack Mode Comparison

Mode	Flag	Source	Best For
Dictionary	<code>-a 0</code>	Wordlist	Common passwords, leaked lists
Combinator	<code>-a 1</code>	Wordlist + Wordlist	Two-word passwords
Mask / Brute-Force	<code>-a 3</code>	Pattern	Known structure (PINs, patterns)
Hybrid (dict+mask)	<code>-a 6</code>	Wordlist + Pattern	Words with appended chars
Hybrid (mask+dict)	<code>-a 7</code>	Pattern + Wordlist	Words with prepended chars

Strategy: Start with dictionary (`-a 0` + rules), then hybrid (`-a 6` / `-a 7`), then targeted masks (`-a 3`). Full brute-force is a last resort.

Lab & Homework

Lab Exercises

1. Create `hashes.txt` with MD5 hashes of known test passwords
2. Run a **4-digit PIN mask** attack with `?d?d?d?d`
3. Run a **complex mask** attack: `?u?1?1?1?d?d`
4. Run **incremental mode** from 4 to 8 characters
5. Run a **hybrid attack** appending 3 digits to dictionary words

Homework Tasks

1. Design custom charsets for a locale-specific password policy
 2. Benchmark mask vs hybrid vs dictionary attack speeds
 3. Create a multi-stage cracking script that chains attack modes
-

Theory: Mask & Hybrid Attacks

Pattern-based password cracking: when you know the structure but not the exact characters.

Introduction

Dictionary attacks are powerful when users choose common words, but many passwords follow predictable structural patterns: a 4-digit PIN, a capitalized name followed by digits, a word with appended special characters. Mask attacks let you describe these patterns precisely and exhaustively test every combination that matches.

Hybrid attacks combine the best of both worlds: the real-world vocabulary of a wordlist with the systematic character generation of masks. Together, these two modes cover a massive portion of real-world passwords that pure dictionary attacks miss.

Mask Attack Fundamentals (`-a 3`)

A mask attack (attack mode 3) generates password candidates based on a positional character pattern. Each position in the mask specifies which characters are allowed at that position. Hashcat then generates every possible combination.

The mask is read left to right. Each `?x` placeholder represents one character position. The total keyspace is the product of the number of characters in each position's charset.

Built-in Charset Placeholders

Placeholder	Charset	Size	Description
<code>?l</code>	<code>abcdefghijklmnopqrstuvwxyz</code>	26	Lowercase Latin letters
<code>?u</code>	<code>ABCDEFGHIJKLMNOPQRSTUVWXYZ</code>	26	Uppercase Latin letters

Placeholder	Charset	Size	Description
<code>?d</code>	<code>0123456789</code>	10	Decimal digits
<code>?s</code>	<code>!"#\$%&'()*+,-./:;<=>? @[\\]^_`{ }~</code>	33	Special / printable symbols
<code>?a</code>	<code>?l?u?d?s</code>	95	All printable ASCII
<code>?b</code>	<code>0x00 - 0xff</code>	256	Full byte range

Mask Examples

Here are common patterns and how to express them as masks:

- `?d?d?d?d` — 4-digit PIN (0000–9999). Keyspace: 10,000.
- `?u?l?l?l?l?d?d` — Capitalized 4-letter name + 2 digits (e.g., Alex42). Keyspace: 45,697,600.
- `?a?a?a?a` — Any 4 printable characters. Keyspace: 81,450,625.
- `?u?l?l?l?l?l?s?d?d` — Name + special + 2 digits (e.g., Summer!23). Keyspace: ~25 billion.

Basic Command

```
# Crack MD5 hashes using a 4-digit PIN mask
hashcat -m 0 -a 3 -o results.txt hashes.txt ?d?d?d?d
```

Incremental Mode (`-i`)

Often you don't know the exact password length. Incremental mode tells Hashcat to start at a shorter mask and gradually increase the length up to the full mask you provide.

```
# Try lengths 4 through 8 using all printable characters
hashcat -m 0 -a 3 -i --increment-min 4 --increment-max 8 \
hashes.txt ?a?a?a?a?a?a?a
```

The process works as follows:

1. Hashcat starts at length `--increment-min` (4 in this example)
2. It runs the mask truncated to that length: `?a?a?a?a`
3. When that keyspace is exhausted, it increments to length 5: `?a?a?a?a?a`
4. This continues until `--increment-max` (8) is reached

Keyspace explosion: With `?a` (95 chars per position), each additional character multiplies the keyspace by 95. Length 4 = 81 million, length 5 = 7.7 billion, length 6 = 735 billion, length 7 = 69 trillion, length 8 = 6.6 quadrillion. Plan your time and resources accordingly.

If you omit `--increment-min`, Hashcat defaults to starting at length 1. If you omit `--increment-max`, it defaults to the length of your mask.

Custom Charsets

The built-in placeholders are convenient but sometimes too broad or too narrow.

Hashcat supports up to four custom charset definitions using the flags `-1`, `-2`, `-3`, and `-4`.

```
# Define custom charset 1 as lowercase + digits
# Then use ?1 in the mask
hashcat -m 0 -a 3 -1 ?l?d hashes.txt ?1?1?1?1?1?1

# Two custom charsets: ?1 = lowercase+digits, ?2 = uppercase+special
hashcat -m 0 -a 3 -1 ?l?d -2 ?u?s hashes.txt ?2?1?1?1?1?d

# Literal characters in a custom charset
hashcat -m 0 -a 3 -1 aeiou hashes.txt ?1?1?1?1?1
```

Custom charsets let you model real-world password policies precisely. For example, if a policy requires at least one uppercase letter, you can put `?u` in specific positions and `?l?d` in the rest.

Combining charsets: A custom charset definition can include both built-in placeholders and literal characters. For example, `-1 ?l?d_.` creates a charset

containing all lowercase letters, all digits, underscore, hyphen, and period.

Hybrid Attacks

Hybrid attacks combine a wordlist with a mask. They are extremely effective because users commonly take a dictionary word and append or prepend numbers, years, or symbols to meet password requirements.

Mode 6: Wordlist + Mask (Append)

Attack mode 6 takes each word from the dictionary and appends every combination generated by the mask.

```
# Append 3 digits to each word in rockyou.txt
hashcat -m 0 -a 6 hashes.txt rockyou.txt ?d?d?d

# Example: "password" becomes password000, password001, ... password999
# Example: "summer" becomes summer000, summer001, ... summer999
```

The total keyspace is: (number of words in dictionary) x (mask keyspace). For rockyou.txt (~14 million words) with `?d?d?d` (1,000), that is 14 billion candidates.

Mode 7: Mask + Wordlist (Prepend)

Attack mode 7 is the reverse: the mask generates a prefix that is prepended to each dictionary word.

```
# Prepend 3 digits before each word in rockyou.txt
hashcat -m 0 -a 7 hashes.txt ?d?d?d rockyou.txt

# Example: "password" becomes 000password, 001password, ... 999password
```

Argument order matters. In `-a 6`, the wordlist comes first, then the mask. In `-a 7`, the mask comes first, then the wordlist. This matches the order in which they appear in the generated candidate: left-to-right.

Common Hybrid Patterns

Pattern	Mode	Command Fragment
word + 2 digits	<code>-a 6</code>	<code>rockyou.txt ?d?d</code>
word + year	<code>-a 6</code>	<code>rockyou.txt 20?d?d</code>
word + special + digit	<code>-a 6</code>	<code>rockyou.txt ?s?d</code>
digit + word	<code>-a 7</code>	<code>?d rockyou.txt</code>
year + word	<code>-a 7</code>	<code>20?d?d rockyou.txt</code>

Choosing Between Attack Modes

A practical cracking session typically chains multiple attack modes in order of efficiency:

1. **Dictionary attack** (`-a 0`) with rules — catches the most common passwords quickly
2. **Hybrid append** (`-a 6`) — catches "word + digits" patterns
3. **Hybrid prepend** (`-a 7`) — catches "digits + word" patterns
4. **Targeted mask** (`-a 3`) — specific known patterns (PINs, date formats)
5. **Incremental mask** (`-a 3 -i`) — broad search across multiple lengths, last resort

Real-world tip: Before running a mask attack, check the password policy of the target system. If passwords must be 8+ characters with at least one uppercase, one digit, and one special character, you can craft a mask that enforces those constraints and dramatically reduce the keyspace.

Keyspace Calculation

Understanding keyspace helps you estimate how long an attack will take. The formula is straightforward:

$\text{Keyspace} = (\text{chars in position 1}) \times (\text{chars in position 2}) \times \dots \times (\text{chars in position } N)$

At a speed of 10 billion hashes per second (typical for MD5 on a modern GPU):

Mask	Keyspace	Time at 10 GH/s
<code>?d?d?d?d</code>	10,000	Instant
<code>?d?d?d?d?d?d</code>	1,000,000	Instant
<code>?1?1?1?1?1?1</code>	308,915,776	~0.03 seconds
<code>?a?a?a?a?a?a</code>	735,091,890,625	~73 seconds
<code>?a?a?a?a?a?a?a?a</code>	6,634,204,312,890,625	~7.7 days

Further Reading

- [Hashcat Wiki — Mask Attack](#)
- [Hashcat Wiki — Hybrid Attack](#)
- [Hashcat Wiki — Custom Charsets](#)

Lab Guide

Five hands-on exercises covering mask attacks, incremental mode, custom charsets, and hybrid attacks.

Prerequisites

- Hashcat installed and working (verify with `hashcat --version`)
- A working directory for this lab (e.g., `~/hashcat-labs/lab06`)
- A small wordlist (e.g., `rockyou.txt` or a custom test wordlist)

Exercise 1 — Create Test Hashes

Goal

Generate MD5 hashes of known passwords so you can verify that your attacks are working correctly.

Step 1 — Generate Hashes

Create MD5 hashes for the following test passwords using `echo -n` and `md5sum` (or `md5` on macOS):

```
# Linux
echo -n "1234" | md5sum | awk '{print $1}' >> hashes.txt
echo -n "5678" | md5sum | awk '{print $1}' >> hashes.txt
echo -n "Alex42" | md5sum | awk '{print $1}' >> hashes.txt
echo -n "John99" | md5sum | awk '{print $1}' >> hashes.txt
echo -n "pass" | md5sum | awk '{print $1}' >> hashes.txt
echo -n "password123" | md5sum | awk '{print $1}' >> hashes.txt

# macOS
echo -n "1234" | md5 >> hashes.txt
echo -n "5678" | md5 >> hashes.txt
echo -n "Alex42" | md5 >> hashes.txt
echo -n "John99" | md5 >> hashes.txt
```

```
echo -n "pass" | md5 >> hashes.txt
echo -n "password123" | md5 >> hashes.txt
```

Step 2 — Verify

Check that `hashes.txt` contains 6 lines, each a 32-character hex string:

```
cat hashes.txt
wc -l hashes.txt
```

Tip: Use `echo -n` (no trailing newline) to get the correct hash. Without `-n`, the newline character is included in the hash input, producing a different result.

Exercise 2 — 4-Digit PIN Mask Attack

Goal

Crack the 4-digit PIN hashes ("1234" and "5678") using a simple digit mask.

Step 1 — Run the Attack

```
hashcat -m 0 -a 3 -o cracked_pins.txt hashes.txt ?d?d?d?d
```

Step 2 — Examine Results

```
cat cracked_pins.txt
```

You should see the hashes for "1234" and "5678" with their cracked plaintext values.

Step 3 — Review the Output

Note in the Hashcat output:

- **Keyspace:** should be 10,000 (10^4)
- **Speed:** how many hashes per second your hardware achieves
- **Recovered:** at least 2 out of 6 hashes should be cracked

Note: If Hashcat reports that some hashes were already found in a potfile, add `--potfile-disable` to force re-cracking, or remove the potfile with `rm ~/.local/share/hashcat/hashcat.potfile`.

Exercise 3 — Complex Mask Attack

Goal

Crack the "Alex42" and "John99" hashes using the pattern: uppercase + 3 lowercase + 2 digits.

Step 1 — Run the Attack

```
hashcat -m 0 -a 3 -o cracked_names.txt hashes.txt ?u?l?l?l?d?d
```

Step 2 — Examine Results

```
cat cracked_names.txt
```

Both "Alex42" and "John99" should appear in the results.

Step 3 — Calculate the Keyspace

Verify the keyspace manually:

- Position 1: `?u` = 26 choices
- Positions 2-4: `?l` = 26 choices each
- Positions 5-6: `?d` = 10 choices each
- Total: $26 \times 26 \times 26 \times 26 \times 10 \times 10 = 45,697,600$

Compare this with the keyspace shown in the Hashcat status output.

Exercise 4 — Incremental Mode

Goal

Use incremental mode to crack the "pass" hash (4 characters) without knowing the exact length in advance.

Step 1 — Run the Incremental Attack

```
hashcat -m 0 -a 3 -i --increment-min 4 --increment-max 8 \  
-o cracked_inc.txt hashes.txt ?a?a?a?a?a?a?a
```

Step 2 — Monitor Progress

Press **s** during the attack to see the status. Observe:

- The current mask length being tested
- The progress percentage for the current length
- Estimated time remaining

Warning: This attack will take a very long time beyond length 5 with **?a**. After "pass" is found at length 4, press **q** to quit gracefully, or let it run if you have time. The point is to observe the incremental behavior.

Step 3 — Verify

```
cat cracked_inc.txt
```

The hash for "pass" should be cracked. It will be found during the length-4 pass.

Exercise 5 — Hybrid Attack

Goal

Crack "password123" using a hybrid attack that appends digits to dictionary words.

Step 1 — Prepare a Small Wordlist

If you don't have rockyou.txt, create a small test wordlist:

```
cat > wordlist.txt <<EOF
password
admin
summer
welcome
dragon
EOF
```

Step 2 — Run the Hybrid Attack (Mode 6)

```
hashcat -m 0 -a 6 -o cracked_hybrid.txt \
hashes.txt wordlist.txt ?d?d?d
```

Step 3 — Examine Results

```
cat cracked_hybrid.txt
```

"password123" should appear in the cracked results. Hashcat combined "password" from the wordlist with the mask `?d?d?d` which generated "123" among its candidates.

Step 4 — Try Prepend Mode (Mode 7)

For comparison, try a prepend attack:

```
hashcat -m 0 -a 7 -o cracked_hybrid7.txt \
hashes.txt ?d?d?d wordlist.txt
```

This generates candidates like "000password", "001password", etc. It won't crack "password123" (since the digits are at the end, not the beginning), but it demonstrates the mode 7 syntax.

Tip: In real-world scenarios, run both `-a 6` and `-a 7` since you often don't know whether the user appended or prepended characters.

Completion Checklist

- Ex 1: hashes.txt created with 6 MD5 hashes
- Ex 2: "1234" and "5678" cracked with `?d?d?d?d` mask
- Ex 3: "Alex42" and "John99" cracked with `?u?1?1?1?d?d` mask
- Ex 4: "pass" cracked using incremental mode
- Ex 5: "password123" cracked using hybrid mode 6

Homework

Three tasks that deepen your understanding of mask design, attack benchmarking, and multi-stage cracking strategies.

Complete all five lab exercises before starting the homework. These tasks build on the skills you practiced in the lab.

Task 1 — Custom Charset Design

Objective

Design a set of custom charsets tailored to a specific password policy, then use them to crack test hashes.

Instructions

1. Choose a password policy (real or hypothetical). Example: "8 characters minimum, must contain at least one uppercase letter, one lowercase letter, one digit, and one of the following special characters: `!@#$`"
2. Create 5 test passwords that comply with this policy. Generate their MD5 hashes.
3. Design masks using custom charsets (`-1`, `-2`, etc.) that model the policy constraints. Write at least 3 different masks targeting different structural patterns:
 - Example mask 1: `-1 !@#$ ?u?l?l?l?l?d?d?1` (uppercase + 4 lower + 2 digits + special)
 - Example mask 2: `-1 !@#$ ?1?u?l?l?l?l?l?d` (special + uppercase + 5 lower + digit)
 - Example mask 3: `-1 !@#$ ?d?d?u?l?l?l?l?l?1` (2 digits + uppercase + 4 lower + special)

4. Run each mask against your test hashes. Record which passwords each mask cracks.
5. Write a brief explanation (3-5 sentences) of why multiple masks are needed even with custom charsets.

Deliverable

A text file or document containing: the password policy, your 5 test passwords, the 3+ masks with their commands, results, and your explanation.

Task 2 — Attack Speed Benchmarking

Objective

Compare the performance of mask, hybrid, and dictionary attacks on identical hash sets to understand when each mode is most efficient.

Instructions

1. Create a set of 10 test passwords that include a mix of:
 - Pure dictionary words (e.g., "sunshine")
 - Words with appended digits (e.g., "sunshine123")
 - Pattern-based passwords (e.g., "Abcd1234")
 - Words with prepended digits (e.g., "99sunshine")
2. Generate their MD5 hashes into `bench_hashes.txt`.
3. Run the following attacks and record the time, speed, and number of passwords cracked for each:

```
# Dictionary attack
hashcat -m 0 -a 0 bench_hashes.txt rockyou.txt

# Hybrid append
hashcat -m 0 -a 6 bench_hashes.txt rockyou.txt ?d?d?d

# Hybrid prepend
hashcat -m 0 -a 7 bench_hashes.txt ?d?d rockyou.txt
```

```
# Mask attack (8-char mixed)
hashcat -m 0 -a 3 bench_hashes.txt ?u?l?l?l?d?d?d?d
```

4. Create a results table:

Attack Mode	Passwords Cracked	Time	Speed (H/s)
Dictionary (-a 0)	?/10	?	?
Hybrid append (-a 6)	?/10	?	?
Hybrid prepend (-a 7)	?/10	?	?
Mask (-a 3)	?/10	?	?

5. Write a brief analysis (5-8 sentences): Which attack mode cracked the most passwords? Which was fastest? Were there passwords that only one mode could crack? What does this tell you about real-world cracking strategy?

Tip: Use `--potfile-disable` between runs so previously cracked hashes don't get skipped. Alternatively, clear the potfile before each run.

Task 3 — Multi-Stage Cracking Script

Objective

Write a shell script that chains multiple attack modes in order of efficiency, stopping when all hashes are cracked or all attacks are exhausted.

Instructions

1. Create a Bash script called `crack_chain.sh` that accepts a hash file as an argument.
2. The script should run the following attacks in sequence:
 1. **Stage 1:** Dictionary attack with `rockyou.txt`
 2. **Stage 2:** Hybrid append — wordlist + `?d?d`

3. **Stage 3:** Hybrid append — wordlist + `?d?d?d`
 4. **Stage 4:** Hybrid prepend — `?d?d` + wordlist
 5. **Stage 5:** Mask for 4-digit PINs: `?d?d?d?d`
 6. **Stage 6:** Mask for 6-digit PINs: `?d?d?d?d?d?d`
 7. **Stage 7:** Mask for name+digits: `?u?l?l?l?l?d?d`
3. Between each stage, print a status message showing which stage just completed and how many hashes have been cracked so far.
 4. Test the script against a set of 10+ test hashes covering diverse password patterns.

Starter Template

```
#!/bin/bash
# crack_chain.sh - Multi-stage hashcat cracking script

HASHFILE="$1"
WORDLIST="rockyou.txt"
HASHTYPE=0 # MD5

if [ -z "$HASHFILE" ]; then
    echo "Usage: $0 <hashfile>"
    exit 1
fi

echo "=== Stage 1: Dictionary Attack ==="
hashcat -m $HASHTYPE -a 0 "$HASHFILE" "$WORDLIST"
echo "Stage 1 complete."

echo "=== Stage 2: Hybrid Append ?d?d ==="
hashcat -m $HASHTYPE -a 6 "$HASHFILE" "$WORDLIST" ?d?d
echo "Stage 2 complete."

echo "=== Stage 3: Hybrid Append ?d?d?d ==="
hashcat -m $HASHTYPE -a 6 "$HASHFILE" "$WORDLIST" ?d?d?d
echo "Stage 3 complete."

# Continue with remaining stages...
echo "=== Stage 4: Hybrid Prepend ?d?d ==="
```

Enhancement ideas: Add a function that counts remaining uncracked hashes after each stage and skips further stages if all are cracked. Add a `-quiet` mode that suppresses Hashcat's verbose output. Add timing information for each stage.

Submission Checklist

- Task 1: Password policy defined; 5 test passwords + hashes; 3+ custom masks; results and explanation
- Task 2: 10 test passwords benchmarked across 4 attack modes; results table; written analysis
- Task 3: `crack_chain.sh` script created; tested against 10+ hashes; stage output visible

Lesson 07: Wordlist Generation

LESSON 07

Wordlist Generation

The quality of a dictionary attack depends entirely on the quality of the wordlist. In this lesson we explore where to find wordlists, how to generate custom ones with Crunch and CeWL, and how to supercharge them with Hashcat rules.

Objectives

- **Identify wordlist sources** — data breach leaks, generators, and common password lists
 - **Generate custom wordlists** using Crunch and CeWL
 - **Apply Hashcat rule files** to multiply the effectiveness of any wordlist
 - **Combine and deduplicate** multiple wordlists into optimized attack dictionaries
-

SOURCES

Where Do Wordlists Come From?

Not all wordlists are created equal. The best ones are derived from **real-world password leaks**, because they reflect how actual humans choose passwords.

Major Sources

- **Data breaches** — RockYou (14M passwords), LinkedIn (117M), Ashley Madison (36M). These are the gold standard for attack wordlists.
- **Common password lists** — SecLists, Probable Wordlists, CrackStation's dictionary. Curated and deduplicated.
- **Generators** — Crunch, Mentalist, CUPP. Create targeted wordlists based on patterns or personal info.
- **Web-scraped wordlists** — CeWL extracts words from websites to build target-specific dictionaries.
- **Multilingual wordlists** — Passwords from non-English speaking regions often follow different patterns and character sets.

Key insight: RockYou is included by default in Kali Linux at `/usr/share/wordlists/rockyou.txt`. It remains the most widely used wordlist in password cracking.

CRUNCH

Crunch — Custom Wordlist Generator

Crunch generates wordlists based on character sets and length ranges. It is exhaustive — it creates every possible combination within the specified parameters.

Basic Syntax

```
# Generate all passwords from 8 to 10 characters (default charset)
crunch 8 10 -o custom_dict.txt

# Specify a custom character set (lowercase + digits)
crunch 8 10 abcdefghijklmnopqrstuvwxyz0123456789 -o custom_dict2.txt

# Use a pattern: @=lowercase, =uppercase, %=digits, ^=symbols
crunch 8 8 -t @@@@%$$$ -o pattern_dict.txt
```

Warning: Crunch generates every combination. A wordlist of all 8-character lowercase+digit passwords is $36^8 = 2.8$ trillion entries. Be mindful of disk space and time.

CEWL

CeWL — Custom Word List Generator from Websites

CeWL spiders a target website and extracts unique words to build a wordlist. This is invaluable when targeting an organization — employees often use company-related terms as passwords.

```
# Basic usage — spider a website and save words
cewl -w web_dict.txt http://example.com

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

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

# Also extract words from metadata of downloadable files
cewl --meta -w web_dict.txt http://example.com
```

Pro tip: Combine CeWL output with Hashcat rules. A company name like "Acme" becomes "Acme123", "acme2024!", "ACME" and hundreds of other variants through rule-based mutations.

RULES

Hashcat Rules for Wordlist Manipulation

Rules transform each word in a wordlist into multiple candidates at runtime — without increasing the wordlist file size. This is one of Hashcat's most powerful features.

Built-in Rule Files

Rule File	Rules	Description
<code>best64.rule</code>	64	The most effective 64 rules. Great starting point.
<code>d3ad0ne.rule</code>	~34,000	Large, aggressive rule set for thorough attacks.
<code>dive.rule</code>	~99,000	Extremely comprehensive. Use when time is not a constraint.
<code>toggles1-5.rule</code>	varies	Toggle case at positions 1 through 5.

```
# Apply best64 rules to rockyou.txt  
hashcat -m 0 hashes.txt rockyou.txt -r /usr/share/hashcat/rules/best64.rule
```

SYNTAX

Rule Syntax Basics

Each line in a rule file is a sequence of operations applied to every word. Understanding the syntax lets you write your own custom rules.

Common Rule Operations

Rule	Name	Example: "password"
:	Do nothing (passthrough)	password
c	Capitalize first letter	Password
u	Uppercase all	PASSWORD
l	Lowercase all	password
t	Toggle case of all	PASSWORD
\$1	Append "1"	password1
\$!	Append "!"	password!

<code>^1</code>	Prepend "1"	1password
<code>r</code>	Reverse	drowssap
<code>d</code>	Duplicate	passwordpassword
<code>sa@</code>	Replace a with @	p@ssword

```
# Combined rule: capitalize + append "123"  
# Rule line: c $1 $2 $3  
# "password" → "Password123"
```

COMBINING

Combining & Optimizing Wordlists

Real-world attacks often use multiple wordlists. Combining them removes duplicates and produces a single, optimized file.

```
# Merge two wordlists, sort, and remove duplicates
cat dict1.txt dict2.txt | sort -u > combined.txt

# Count lines (unique passwords)
wc -l combined.txt

# Merge multiple files at once
cat rockyou.txt web_dict.txt custom_dict.txt | sort -u > mega_dict.txt

# Sort by frequency (if you have counts)
sort -t: -k2 -rn passwords_with_counts.txt | head -10000 > top10k.txt
```

Personalized wordlists

Use CUPP or Mentalist to generate wordlists from personal info (names, dates, pet names). Highly effective for targeted attacks.

Combined approach

Start with a breach list, add CeWL output for the target, mix in a personalized list, deduplicate, then apply rules during the attack.

NEXT STEPS

Lab & Homework

Lab 3 — Wordlist Generation

- **Step 1:** Install Crunch and CeWL on your system
- **Step 2:** Generate wordlists with both tools
- **Step 3:** Run Hashcat attacks using your custom wordlists with rule files
- **Step 4:** Analyze and compare results across different wordlist strategies

Homework

- Build a comprehensive attack wordlist from multiple sources
 - Write custom Hashcat rules and benchmark them against built-in rules
 - Profile CeWL against a test website and analyze the output
-

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](#)

Lab 3 — Wordlist Generation

Install wordlist generation tools, create custom dictionaries, and run attacks with them.

Prerequisites: You need a working Hashcat installation (from Lab 1) and a terminal with root/sudo access for package installation. All commands assume a Linux or macOS environment.

Exercise 1 — Install Crunch

EX 1

Install the Crunch wordlist generator

Crunch is available in most package managers. Install it for your platform:

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

# macOS (Homebrew)
brew install crunch

# Verify installation
crunch --help
```

If the installation succeeds, you should see Crunch's help output listing the available options.

Checkpoint: Running `crunch --help` displays usage information without errors.

Exercise 2 — Install CeWL

EX 2

Install the CeWL web wordlist generator

CeWL is a Ruby gem. You need Ruby installed on your system first:

```
# Check Ruby is available
ruby --version

# Install CeWL
gem install cewl

# On Kali Linux, CeWL is pre-installed
which cewl

# Verify
cewl --help
```

Note: If you encounter permission errors with `gem install`, try `sudo gem install cewl` or use a Ruby version manager like `rvm` or `rbenv`.

Checkpoint: Running `cewl --help` displays usage information.

Exercise 3 — Generate a Wordlist with Crunch

EX 3

Create custom wordlists using Crunch

Create a working directory and generate several wordlists:

```
# Create a lab directory
mkdir -p ~/hashcat_lab3
cd ~/hashcat_lab3

# Generate a small wordlist: 4-digit PINs
crunch 4 4 0123456789 -o pins.txt
wc -l pins.txt

# Expected: 10,000 lines
```

```
# Generate passwords 6-8 chars, lowercase only
crunch 6 8 abcdefghijklmnopqrstuvwxyz -o alpha_dict.txt
# WARNING: This will be very large! Cancel with Ctrl+C after a few seconds
# to see it working, then clean up

# More practical: use a pattern
# 4 lowercase letters + 4 digits (e.g., "pass1234")
crunch 8 8 -t @@@@% % -o pattern_dict.txt
wc -l pattern_dict.txt

# Generate passwords 8-10 chars with default charset
crunch 8 10 -o custom_dict.txt
# WARNING: extremely large output — cancel quickly or limit with -c flag

# Practical approach: limit line count
crunch 8 8 abcdefghijklmnopqrstuvwxyz0123456789 -c 100000 -o limited_dict.txt
```

Examine the output

```
# Check sizes
ls -lh *.txt

# Preview the first and last entries
head -5 pins.txt
tail -5 pins.txt

head -5 pattern_dict.txt
tail -5 pattern_dict.txt
```

Checkpoint: You have at least two Crunch-generated wordlists in your lab directory. `pins.txt` contains exactly 10,000 lines.

Exercise 4 — Generate a Wordlist with CeWL

EX 4

Spider a webpage and build a wordlist

Use CeWL to extract words from a public website. We will use example.com as a safe target:


```
cd ~/hashcat_lab3

# Basic CeWL run against example.com
cewl -w web_dict.txt http://example.com
wc -l web_dict.txt
cat web_dict.txt

# Try with a more content-rich page (Wikipedia)
cewl -m 6 -d 1 -w wiki_dict.txt https://en.wikipedia.org/wiki/Password
wc -l wiki_dict.txt
head -20 wiki_dict.txt

# Extract longer words (minimum 8 characters)
cewl -m 8 -d 2 -w long_words.txt https://en.wikipedia.org/wiki/Cryptograph
wc -l long_words.txt
```

Tip: In a real engagement, you would run CeWL against the target organization's website, blog, about page, and public documentation. The extracted words become high-value password candidates.

Checkpoint: `web_dict.txt` and `wiki_dict.txt` contain extracted words. The Wikipedia-sourced list should have dozens to hundreds of unique words.

Exercise 5 — Run Attacks with Custom Wordlists

EX 5

Use your custom wordlists with Hashcat

First, create some test hashes, then attack them with your wordlists:

```
cd ~/hashcat_lab3

# Create test hashes (MD5) from known passwords
echo -n "pass1234" | md5sum | awk '{print $1}' > test_hashes.txt
echo -n "word5678" | md5sum | awk '{print $1}' >> test_hashes.txt
echo -n "test0000" | md5sum | awk '{print $1}' >> test_hashes.txt
```

```
# On macOS, use md5 instead of md5sum:
# echo -n "pass1234" | md5 >> test_hashes.txt

# Attack with the pattern dictionary (no rules)
hashcat -m 0 test_hashes.txt pattern_dict.txt --potfile-disable

# Attack with CeWL wordlist + best64 rules
hashcat -m 0 test_hashes.txt wiki_dict.txt -r /usr/share/hashcat/rules/bes

# Attack with PIN dictionary
hashcat -m 0 test_hashes.txt pins.txt --potfile-disable

# Show cracked results
hashcat -m 0 test_hashes.txt --show
```

Combine wordlists and attack

```
# Merge all your wordlists
cat pins.txt pattern_dict.txt web_dict.txt wiki_dict.txt | sort -u > combined.txt
wc -l combined.txt

# Attack with combined wordlist + best64 rules
hashcat -m 0 test_hashes.txt combined.txt -r /usr/share/hashcat/rules/bes
```

Checkpoint: At least "pass1234" should be cracked by the pattern dictionary. The combined attack should crack additional hashes.

Exercise 6 — Analyze Results

EX 6

Compare wordlist effectiveness

Document your findings by answering these questions:

1. How many hashes did each wordlist crack individually (pins, pattern, CeWL, combined)?
2. What was the speed (hashes per second) for each attack?
3. How does adding rules (best64.rule) change the number of candidates and the results?

4. Which wordlist strategy was most effective for the test hashes you created?

Test rule impact

```
# See how many candidates a wordlist + rules generates
hashcat --stdout wiki_dict.txt -r /usr/share/hashcat/rules/best64.rule |

# Compare with the raw wordlist
wc -l wiki_dict.txt

# Try with d3ad0ne rules (much larger)
hashcat --stdout wiki_dict.txt -r /usr/share/hashcat/rules/d3ad0ne.rule
```

Expected observation: The best64 rule set multiplies your wordlist by 64x. A 1,000-word CeWL list becomes 64,000 candidates. The d3ad0ne rules multiply it by ~34,000x — turning 1,000 words into 34 million candidates.

Checkpoint: You have documented which wordlist/rule combination cracked the most hashes and understand the trade-offs between wordlist size, rules, and attack speed.

Clean up: Remove large generated wordlists when done to free disk space: `rm`

```
~/hashcat_lab3/alpha_dict.txt ~/hashcat_lab3/custom_dict.txt
```

Homework — Wordlist Generation

Three tasks to deepen your understanding of wordlist creation and optimization.

Document your commands, outputs, and observations for each task. Submit a text file or screenshot of your terminal session.

TASK 1

Build a Comprehensive Attack Wordlist

Create a multi-source wordlist by combining at least three different sources:

1. **Crunch-generated:** Generate a pattern-based wordlist of 8-character passwords following the pattern: one uppercase letter, five lowercase letters, two digits (e.g., "Abcdef12"). Use Crunch's `-t` flag with the pattern `,@#####%.`
2. **CeWL-sourced:** Run CeWL against two different public websites of your choice (news sites, Wikipedia articles, company pages). Use a minimum word length of 5 and a spider depth of 2.
3. **Manual additions:** Create a small file of 20-30 common password bases (e.g., "password", "admin", "welcome", "letmein", "qwerty", "dragon", "master", "monkey", "shadow").

Combine all sources, deduplicate, and report:

- How many unique words does each source contribute?
- What is the total combined wordlist size after deduplication?
- How much overlap exists between sources?

```
# Combine and analyze
cat crunch_out.txt cewl_site1.txt cewl_site2.txt manual.txt | sort -u > combined.txt
wc -l crunch_out.txt cewl_site1.txt cewl_site2.txt manual.txt combined.txt
```

TASK 2

Write and Benchmark Custom Hashcat Rules

Create a custom rule file called `my_rules.rule` with at least 10 rules targeting common password patterns:

```
# Example my_rules.rule content:
:                # passthrough (original word)
c                # capitalize first letter
c $1            # capitalize + append "1"
c $1 $2 $3      # capitalize + append "123"
$!             # append "!"
$1 $!          # append "1!"
sa@ se3 si1 so0 # leetspeak substitutions
c $2 $0 $2 $4   # capitalize + append "2024"
u              # uppercase all
^1 ^2 ^3        # prepend "321"
```

Benchmark against built-in rules

Create 50 MD5 hashes from known passwords, then compare crack rates:

1. Attack with your wordlist + **no rules**
2. Attack with your wordlist + **my_rules.rule**
3. Attack with your wordlist + **best64.rule**
4. Attack with your wordlist + **d3ad0ne.rule**

For each run, record: number of hashes cracked, time taken, and total candidates tested. Present results in a comparison table.

Tip: Use `hashcat --stdout wordlist.txt -r my_rules.rule | wc -l` to count candidates without actually attacking. This helps predict runtime.

TASK 3

CeWL Profiling and Targeted Attack

Choose a public-facing website (a company site, university, or open-source project) and perform a thorough CeWL analysis:

1. Run CeWL with varying parameters and compare the results:

```
# Minimum word length 4, depth 1
cewl -m 4 -d 1 -w short_shallow.txt http://target-site.com
wc -l short_shallow.txt

# Minimum word length 6, depth 3
cewl -m 6 -d 3 -w medium_deep.txt http://target-site.com
wc -l medium_deep.txt

# Minimum word length 8, depth 2, with emails
cewl -m 8 -d 2 -e --email_file found_emails.txt -w long_medium.txt http://target-site.com
wc -l long_medium.txt found_emails.txt
```

2. Analyze the extracted words:

- What types of words did CeWL find (technical terms, names, products)?
- Which words would make plausible password bases?
- Were any email addresses or metadata found?

3. Create a targeted attack plan:

- Select the best CeWL output and apply it with `best64.rule`
- Create 10 test hashes based on words you expect CeWL would find on the target site (e.g., company name + digits)
- Run the attack and verify your targeted wordlist cracks them

Write a short report (half a page) summarizing: how many useful password candidates CeWL extracted, which parameters gave the best results, and how the targeted approach compares to using a generic wordlist like RockYou.

Lesson 08: GPU Acceleration & Performance

GPU Acceleration & Performance

Modern GPUs transform password cracking from hours to minutes. This lesson covers how to harness GPU parallel processing in Hashcat, tune performance, manage thermals, and troubleshoot common issues.

Learning Objectives

- Understand why **GPUs outperform CPUs** for hash cracking
 - Prepare your system with the correct **drivers and compute frameworks**
 - Use **device selection** and **workload profiles** to control performance
 - Configure **temperature limits** and **runtime constraints**
 - Troubleshoot **common GPU issues** in Hashcat
-

Why GPU?

CPUs are designed for complex sequential tasks with a few powerful cores. GPUs contain **thousands of smaller cores** optimized for parallel computation — perfect for hashing.

GPU Advantages

- **Parallel computation:** Thousands of cores process hashes simultaneously
- **High throughput:** 10–100x faster than CPU for most hash types
- **Efficient for large volumes:** Millions of hash candidates tested per second

Example: An NVIDIA RTX 4090 can compute ~160 billion MD5 hashes per second. A high-end CPU manages ~5 billion. That is a 32x difference on a single GPU.

Preparation

Before using GPU acceleration, your system must have the correct drivers and compute frameworks installed.

NVIDIA GPUs

- Install the latest **NVIDIA driver** from [nvidia.com](https://www.nvidia.com)
- Install **CUDA Toolkit** (includes OpenCL support)
- Verify with `nvidia-smi`

AMD GPUs

- Install **AMD Adrenalin drivers** or ROCm
- Ensure **OpenCL runtime** is available
- Verify with `clinfo`

Note: Hashcat supports OpenCL, CUDA, and HIP backends. CUDA is preferred for NVIDIA; OpenCL works for

both vendors.

Checking Available Devices

Use `hashcat -I` to list all available compute devices on your system.

```
# List all OpenCL/CUDA devices
hashcat -I
```

Sample output:

```
OpenCL Info:
=====
Platform ID #1
  Name.....: NVIDIA CUDA
  Vendor....: NVIDIA Corporation
  Device ID #1
    Name.....: NVIDIA GeForce RTX 4090
    Type.....: GPU
    Memory.....: 24564 MB
    OpenCL Version.: OpenCL 3.0 CUDA

Platform ID #2
  Name.....: Intel(R) OpenCL
  Device ID #2
    Name.....: Intel(R) Core(TM) i9-13900K
```

Device IDs matter. Use these IDs with `-d` to target specific devices.

Using GPU in Hashcat

The `-D` flag selects the device type. Use `-D 2` to target GPUs exclusively.

```
# Dictionary attack on GPU
hashcat -m 0 -a 0 -D 2 -o results.txt hashes.txt rockyou.txt
```

Flag	Value	Description
<code>-D 1</code>	CPU	Use CPU only
<code>-D 2</code>	GPU	Use GPU only
<code>-D 1,2</code>	CPU + GPU	Use both (not always optimal)

Selecting Specific GPUs

```
# Use only GPU devices 1 and 2  
hashcat -m 0 -a 0 -D 2 -d 1,2 hashes.txt rockyou.txt
```

Workload Profiles

The `-w` flag controls how aggressively Hashcat uses the GPU. Higher profiles yield faster speeds but can make the system unresponsive.

Profile	Name	Description	Use Case
<code>-w 1</code>	Low	Minimal GPU usage	Desktop remains usable
<code>-w 2</code>	Medium	Default balance	Background cracking
<code>-w 3</code>	High	Aggressive GPU usage	Dedicated cracking rig
<code>-w 4</code>	Maximum	Full GPU saturation	Headless server, no display

```
# High workload profile
hashcat -m 0 -a 3 -D 2 -w 3 hashes.txt ?a?a?a?a?a
```


Warning: Profile `-w 4` can cause display freezes and system instability on machines with a single GPU that also drives the display.

Temperature Control

Hashcat provides built-in thermal management to protect your hardware.

```
# Abort if GPU reaches 85C, throttle to stay under 75C
hashcat -m 0 -a 0 -D 2 \
  --gpu-temp-abort=85 \
  --gpu-temp-retain=75 \
  hashes.txt rockyou.txt
```

Temperature Flags

- `--gpu-temp-abort=N` — Immediately stop if any GPU reaches N degrees Celsius
- `--gpu-temp-retain=N` — Dynamically throttle speed to keep the GPU at or below N degrees

Note: Monitor temperatures in real-time with `nvidia-smi -l 1` (NVIDIA) or `watch -n1 sensors` (Linux).

Performance Tuning

Advanced flags for fine-tuning GPU performance:

Flag	Description	Notes
<code>-n</code>	Kernel accel (threads per block)	NVIDIA tuning; higher = more parallelism
<code>-u</code>	Kernel loops (work blocks)	AMD tuning; controls work granularity
<code>-T</code>	Kernel threads (vector width)	Number of vector elements per thread

Runtime Limits

```
# Run for at most 1 hour, show status every 60 seconds
hashcat -m 0 -a 3 -D 2 \
  --runtime=3600 \
  --status-timer=60 \
  hashes.txt ?a?a?a?a?a?a
```

Tip: Let Hashcat auto-tune first (`-w 3` without `-n` / `-u` / `-T`). Only manually tune if you understand your GPU architecture and need to squeeze out the last few percent.

Troubleshooting

Common GPU issues and solutions:

Low Hash Speed

- Check workload profile (`-w 3` or `-w 4`)
- Ensure GPU drivers are up to date
- Verify you are using `-D 2` (not CPU)

GPU Overheating

- Set `--gpu-temp-retain=75`
- Improve case airflow / cooling
- Reduce workload profile

Multi-GPU Issues

- Use `-d` to select specific devices
- Check for PCIe bandwidth bottlenecks
- Ensure matching driver versions

Driver / RAM Problems

- Reinstall CUDA or OpenCL runtime
 - Check GPU memory with `nvidia-smi`
 - Close other GPU-intensive applications
-

Lab & Homework

Lab Exercises

1. **Device check** — run `hashcat -I` and system tools
2. **GPU dictionary attack** — crack hashes with `-D 2`
3. **GPU mask attack** — brute-force with GPU
4. **Workload comparison** — test `-w 1` through `-w 4`
5. **Temperature limits** — set thermal safeguards
6. **CPU vs GPU** — compare performance

Homework Tasks

1. Benchmark all hash modes on your GPU
2. Optimize workload for a specific hash type
3. Document multi-GPU or CPU-vs-GPU results

Theory: GPU Acceleration & Performance

A comprehensive guide to leveraging GPU computing power for efficient hash cracking with Hashcat.

Why GPUs Excel at Hash Cracking

Hash functions like MD5, SHA-1, and bcrypt are designed to be computed independently for each input. There are no data dependencies between one hash computation and the next. This makes hashing an **embarrassingly parallel** workload — the ideal scenario for GPU computing.

A modern CPU has 8–24 cores, each capable of complex branching and sequential logic. A modern GPU has thousands of simpler cores (NVIDIA calls them CUDA cores; AMD calls them stream processors) designed to execute the same operation on many data points simultaneously. When cracking passwords, each GPU core can compute a hash for a different password candidate at the same time.

The result is dramatic. A single NVIDIA RTX 4090 can compute approximately 160 billion MD5 hashes per second, while a top-tier Intel or AMD CPU manages roughly 5 billion. For slower hash algorithms like bcrypt, the gap narrows but GPUs still hold a significant advantage.

CPU vs GPU Architecture

Property	CPU	GPU
Core count	8–24 (high-end desktop)	5,000–16,000+ (modern GPUs)
Clock speed	3–6 GHz per core	1.5–2.5 GHz per core
Strength	Complex sequential tasks	Massively parallel simple tasks
Memory	System RAM (32–128 GB typical)	VRAM (8–24 GB typical)
Best for	General computing, OS tasks	Hash cracking, rendering, ML

System Preparation

GPU acceleration in Hashcat requires the correct drivers and compute frameworks. Without them, Hashcat cannot detect or use the GPU.

NVIDIA GPUs

1. **Install the latest NVIDIA driver** from the [NVIDIA driver page](#). On Linux, use your distribution's package manager or the official `.run` installer.
2. **Install the CUDA Toolkit** from the [NVIDIA CUDA page](#). CUDA includes OpenCL support, so you get both backends.
3. **Verify the installation:**

```
# Check NVIDIA driver and GPU status
nvidia-smi

# Expected output includes GPU name, driver version, CUDA version,
# temperature, power usage, and memory usage
```

AMD GPUs

1. **Install AMD drivers:** On Windows, install AMD Adrenalin Edition. On Linux, install ROCm or the AMDGPU-PRO driver with OpenCL support.
2. **Verify OpenCL availability:**

```
# Check OpenCL platforms and devices
clinfo

# Look for your GPU listed under a platform
# Ensure "Device Type" shows "GPU"
```

Backend support: Hashcat supports three compute backends: **CUDA** (NVIDIA only, often fastest), **OpenCL** (both vendors), and **HIP** (AMD ROCm). If you have an NVIDIA GPU, CUDA is recommended. For AMD, OpenCL or HIP depending on your driver setup.

Detecting Devices with `hashcat -I`

Before running any attack, always verify that Hashcat sees your GPU. The `-I` flag lists all detected compute devices:

```
hashcat -I
```

The output shows each platform (CUDA, OpenCL) and its devices with their IDs, names, memory, and capabilities. Pay attention to:

- **Device ID** — used with `-d` to select specific GPUs
- **Device Type** — GPU vs CPU
- **Memory** — larger VRAM allows bigger wordlists and more complex attacks
- **Driver Version** — outdated drivers can cause crashes or poor performance

Device Selection

Hashcat provides two flags for controlling which hardware is used:

Device Type (`-D`)

Selects the category of device:

- `-D 1` — CPU only
- `-D 2` — GPU only
- `-D 1,2` — Both CPU and GPU

In almost all cases, `-D 2` is the correct choice. Using both CPU and GPU simultaneously can actually slow things down because the CPU processes hashes much slower and creates synchronization overhead.

Specific Devices (`-d`)

When you have multiple GPUs, use `-d` to select which ones to use:

```
# Use only GPU #1
hashcat -m 0 -a 0 -D 2 -d 1 hashes.txt rockyou.txt

# Use GPUs #1 and #2
hashcat -m 0 -a 0 -D 2 -d 1,2 hashes.txt rockyou.txt
```

```
# Use all available GPUs (default when -D 2 is set)
hashcat -m 0 -a 0 -D 2 hashes.txt rockyou.txt
```

Tip: If your primary GPU also drives the display, running Hashcat at full load on it can freeze or crash your display. Consider using `-d 2` to target a secondary GPU, or lower the workload profile.

Workload Profiles

The `-w` flag controls how much GPU time Hashcat demands. Higher values mean faster cracking but more system impact.

Profile	Name	Behaviour	Recommended Use
<code>-w 1</code>	Low	Short GPU compute bursts with frequent pauses. Desktop remains fully responsive.	Cracking while working on the same machine
<code>-w 2</code>	Medium	Default balance between speed and responsiveness.	General-purpose cracking
<code>-w 3</code>	High	Long GPU compute bursts. System may lag noticeably.	Dedicated cracking sessions
<code>-w 4</code>	Maximum	GPU is fully saturated. Display may freeze.	Headless servers or secondary GPUs

The difference between `-w 1` and `-w 4` can be 2–5x in speed depending on the hash type and GPU. For fast hash types like MD5, the impact is smaller because the GPU is already fast. For slow hashes like bcrypt, the profile makes a bigger difference.

Temperature Management

Sustained GPU cracking generates significant heat. Without thermal management, GPUs can throttle (reducing performance) or shut down to prevent hardware damage. Hashcat provides two temperature controls:

`--gpu-temp-abort=N`

If any GPU reaches N degrees Celsius, Hashcat immediately stops the session and saves progress. This is a safety net to prevent hardware damage. A reasonable value is 85–90 degrees, depending on your GPU model's thermal limits.

`--gpu-temp-retain=N`

Hashcat dynamically adjusts the workload to keep the GPU at or below N degrees. When the temperature approaches the limit, Hashcat reduces the compute load. When it drops, Hashcat increases it again. Set this 10–15 degrees below the abort temperature.

```
# Recommended thermal configuration
hashcat -m 0 -a 0 -D 2 -w 3 \
  --gpu-temp-abort=85 \
  --gpu-temp-retain=75 \
  hashes.txt rockyou.txt
```

Cooling matters: Ensure adequate case airflow. Clean dust from GPU heatsinks regularly. In multi-GPU rigs, consider liquid cooling or leaving PCIe slot gaps between GPUs for better airflow. An overheating GPU will thermal-throttle and actually be slower than a properly cooled GPU at a lower workload profile.

Advanced Performance Tuning

For most users, `-w 3` or `-w 4` is sufficient. Advanced users can fine-tune kernel parameters for specific GPU architectures:

Kernel Acceleration (`-n`)

Controls the number of threads per compute block. Higher values increase parallelism but also increase GPU memory usage. This is most relevant for NVIDIA GPUs. Hashcat's auto-tuner usually picks a good value, but for some hash modes you may benefit from manual adjustment.

Kernel Loops (`-u`)

Controls the number of iterations per work unit. Larger values reduce overhead from kernel launches but increase the time per unit. This is more relevant for AMD GPUs and for slow hash types where each iteration is expensive.

Kernel Threads (`-T`)

Sets the vector width (number of elements processed per thread). Usually 1, 2, 4, 8, or 16. Auto-tuning handles this well for most cases.

Best practice: Run Hashcat with `-w 3` and no manual kernel flags first. Note the speed. Then experiment with `-n` and `-u` values, comparing speed each time. The Hashcat benchmark (`hashcat -b`) auto-tunes these for each hash mode.

Runtime Controls

For scheduled or time-limited cracking sessions, Hashcat supports:

- `--runtime=N` — Stop after N seconds. The session is saved automatically and can be resumed with `--restore`.
- `--status-timer=N` — Print status information every N seconds. Useful for monitoring long-running sessions.

```
# Run for 1 hour, status every 60 seconds
hashcat -m 0 -a 3 -D 2 -w 3 \
  --runtime=3600 \
  --status-timer=60 \
  hashes.txt ?a?a?a?a?a?a

# Resume a previously stopped session
hashcat --restore
```

Benchmarking

Hashcat includes a built-in benchmark mode that tests performance across all supported hash types on your hardware:

```
# Benchmark all hash types
hashcat -b

# Benchmark a specific hash type
hashcat -b -m 0      # MD5
hashcat -b -m 1000   # NTLM
hashcat -b -m 3200   # bcrypt
```

Benchmark results help you understand your hardware's capabilities and set realistic expectations for cracking times. They also help verify that your GPU is performing correctly — if benchmark speeds are significantly lower than expected for your GPU model, there may be a driver or thermal issue.

Troubleshooting Common Issues

Low Hash Speed

- Verify you are using `-D 2` (GPU mode), not defaulting to CPU
- Increase the workload profile to `-w 3` or `-w 4`
- Update GPU drivers to the latest stable version
- Check for thermal throttling with `nvidia-smi` or `sensors`
- Close other GPU-intensive applications (games, video editors, browsers with hardware acceleration)

GPU Overheating

- Set `--gpu-temp-retain` to keep temperatures in check
- Reduce the workload profile (`-w 2` or `-w 1`)
- Improve physical cooling: clean dust, add case fans, improve cable management
- Increase GPU fan speed using `nvidia-settings` or MSI Afterburner

Multi-GPU Issues

- Ensure all GPUs have the same driver version
- Check PCIe lane allocation: x16 vs x8 vs x4 affects bandwidth
- Use `hashcat -I` to verify all GPUs are detected
- Test each GPU individually with `-d 1`, `-d 2`, etc., to isolate problems

Driver Problems

- Completely uninstall old drivers before installing new ones (use DDU on Windows)
- On Linux, ensure the correct kernel module is loaded: `lsmod | grep nvidia`
- Check for OpenCL/CUDA version compatibility with your Hashcat version
- Some laptop GPUs (Optimus/hybrid graphics) require special configuration

RAM / VRAM Limitations

- Large wordlists can exceed GPU memory. Hashcat handles this by streaming, but performance may drop.
- Some hash modes with large salt counts require significant VRAM. Check Hashcat's output for memory warnings.
- If you see "out of memory" errors, reduce the number of hashes being cracked simultaneously, or use a GPU with more VRAM.

Further Reading

- [Hashcat Official Wiki](#)
- [Hashcat Benchmark Results](#)
- [Hashcat Hardware Monitoring FAQ](#)
- [NVIDIA CUDA Toolkit](#)
- [AMD ROCm Documentation](#)

Lab Guide — GPU Acceleration & Performance

Six hands-on exercises testing GPU detection, attack modes, workload tuning, thermals, and CPU vs GPU comparison.

Prerequisites: A system with a supported GPU (NVIDIA or AMD), up-to-date GPU drivers, and Hashcat installed. Prepare a test hash file with several MD5 hashes and have the `rockyou.txt` wordlist available.

Prepare test hashes: Create a file called `hashes.txt` with known MD5 hashes for testing. For example:

```
# Generate test hashes
echo -n "password" | md5sum | cut -d' ' -f1 > hashes.txt
echo -n "123456" | md5sum | cut -d' ' -f1 >> hashes.txt
echo -n "qwerty" | md5sum | cut -d' ' -f1 >> hashes.txt
echo -n "letmein" | md5sum | cut -d' ' -f1 >> hashes.txt
echo -n "welcome" | md5sum | cut -d' ' -f1 >> hashes.txt
```

Exercise 1 — Check GPU Availability

EX 1

Detect compute devices with Hashcat and system tools

Verify that your GPU is properly detected before running any attacks.

1. Run `hashcat -I` and note all detected devices, their IDs, types, and memory
2. For NVIDIA: run `nvidia-smi` and record the GPU name, driver version, CUDA version, and current temperature

3. For AMD or general OpenCL: run `clinfo` and note the platform and device details
4. Run `hashcat -b -m 0` to benchmark MD5 on your system and record the speed

```
# Step 1: Hashcat device info
hashcat -I

# Step 2: NVIDIA system info
nvidia-smi

# Step 3: OpenCL info (all vendors)
clinfo

# Step 4: Quick MD5 benchmark
hashcat -b -m 0
```

Checkpoint: You can identify your GPU by name and device ID. The benchmark completes without errors.

Exercise 2 — GPU Dictionary Attack

EX 2

Run a dictionary attack using the GPU

Crack the test hashes using `rockyou.txt` on the GPU.

1. Run the attack with GPU device type:

```
# Dictionary attack on GPU
hashcat -m 0 -a 0 -D 2 -o cracked_gpu.txt hashes.txt rockyou.txt
```

2. Observe the output: note the speed (H/s), progress, and time to completion
3. Check the results file:

```
# View cracked passwords
cat cracked_gpu.txt
```

Checkpoint: All test hashes are cracked. The speed shows millions or billions of H/s (depending on your GPU).

Exercise 3 — GPU Mask Attack

EX 3

Run a brute-force mask attack on the GPU

Use a mask to brute-force short passwords on the GPU.

1. Create a hash of a short known password (e.g., "test1"):

```
# Create a test hash for mask attack
echo -n "test1" | md5sum | cut -d' ' -f1 > mask_test.txt
```

2. Run a mask attack targeting 5-character lowercase+digit passwords:

```
# Mask attack: 4 lowercase letters + 1 digit
hashcat -m 0 -a 3 -D 2 -o cracked_mask.txt mask_test.txt ?l?l?l?l?d
```

3. Note the speed, keyspace size, and estimated time to exhaustion

4. Verify the result:

```
cat cracked_mask.txt
```

Checkpoint: The mask attack finds "test1". You can explain the keyspace size ($26^4 * 10 = 4,569,760$ candidates).

Exercise 4 — Workload Profile Comparison

EX 4

Compare speeds across all four workload profiles

Test each workload profile and record the hash rate.

1. Run the benchmark for each profile and record the speed:

```
# Test workload profile 1 (Low)
hashcat -b -m 0 -D 2 -w 1

# Test workload profile 2 (Medium)
hashcat -b -m 0 -D 2 -w 2

# Test workload profile 3 (High)
hashcat -b -m 0 -D 2 -w 3

# Test workload profile 4 (Maximum)
hashcat -b -m 0 -D 2 -w 4
```

2. Record the results in a comparison table:

Profile	Speed (H/s)	System Responsiveness
-----	-----	-----
-w 1	_____	_____
-w 2	_____	_____
-w 3	_____	_____
-w 4	_____	_____

3. Note the speed increase percentage between profiles

Checkpoint: You have a completed comparison table. Speed increases with each profile level.

Exercise 5 — Temperature Limits

EX 5

Set thermal safeguards and observe throttling behaviour

Configure temperature limits and monitor GPU thermals during an attack.

1. Open a second terminal and start monitoring GPU temperature:

```
# NVIDIA: refresh every 1 second
nvidia-smi -l 1
```

```
# Linux alternative (AMD/NVIDIA)
watch -n1 sensors
```

2. In the first terminal, run an attack with thermal limits:

```
# Attack with temperature controls
hashcat -m 0 -a 3 -D 2 -w 3 \
  --gpu-temp-abort=85 \
  --gpu-temp-retain=75 \
  hashes.txt ?a?a?a?a?a?a
```

3. Observe the Hashcat status output — look for temperature readings and any throttling messages

4. Press `s` during the attack to see the current status including temperature

5. Press `q` to quit when you have observed enough data

Checkpoint: Temperature stays at or below the retain value. You can see Hashcat adjusting speed to manage thermals.

Exercise 6 — CPU vs GPU Performance

EX 6

Compare CPU and GPU cracking speeds

Run the same attack on CPU and GPU to quantify the performance difference.

1. Benchmark MD5 on CPU only:

```
# CPU benchmark
hashcat -b -m 0 -D 1
```

2. Benchmark MD5 on GPU only:

```
# GPU benchmark
hashcat -b -m 0 -D 2
```

3. Run the same dictionary attack on both and compare wall-clock time:

```
# CPU dictionary attack (time it)
time hashcat -m 0 -a 0 -D 1 -o cracked_cpu.txt hashes.txt rockyou.txt

# GPU dictionary attack (time it)
time hashcat -m 0 -a 0 -D 2 -o cracked_gpu2.txt hashes.txt rockyou.txt
```

4. Calculate the speedup factor: GPU speed / CPU speed

5. Repeat the comparison for a slower hash type like SHA-256 (`-m 1400`) or
bcrypt (`-m 3200`)

```
# Benchmark SHA-256 on CPU vs GPU
hashcat -b -m 1400 -D 1
hashcat -b -m 1400 -D 2
```

Checkpoint: You can state the GPU speedup factor for at least two hash types.
GPU is significantly faster for both.

Tip: Clean up temporary output files between exercises to avoid Hashcat's "all hashes already cracked" message. Use `--potfile-disable` or remove the potfile between runs: `rm -f ~/.local/share/hashcat/hashcat.potfile`

Homework

Three tasks covering GPU benchmarking, workload optimization, and performance documentation.

Complete all six lab exercises before starting the homework. These tasks build on the skills and observations from the lab.

Task 1 — Comprehensive GPU Benchmark

Objective

Benchmark your GPU across multiple hash types and create a performance reference sheet for your hardware.

Instructions

1. Run `hashcat -b` to benchmark all hash types (or select at least 10 common types)
2. Record the results in a table with columns: Hash Mode, Hash Name, Speed (H/s), and Relative Speed (compared to MD5)
3. Include at minimum these hash types:
 - MD5 (`-m 0`)
 - SHA-1 (`-m 100`)
 - SHA-256 (`-m 1400`)
 - NTLM (`-m 1000`)
 - bcrypt (`-m 3200`)
 - WPA-PBKDF2 (`-m 22000`)
 - SHA-512 (`-m 1700`)
 - MD5crypt (`-m 500`)

- sha512crypt (`-m 1800`)
 - PBKDF2-HMAC-SHA256 (`-m 10900`)
4. Calculate how long each hash type would take to exhaust a 6-character alphanumeric keyspace ($62^6 = \sim 56.8$ billion candidates)
 5. Write a brief analysis: which hash types are GPU-friendly and which are GPU-resistant?

Deliverable

A completed benchmark table with speed data, estimated exhaustion times, and a written analysis of GPU-friendly vs GPU-resistant algorithms.

Self-check: Your table has at least 10 hash types. Exhaustion time calculations are correct. Your analysis correctly identifies bcrypt and scrypt as GPU-resistant.

Task 2 — Workload Optimization for a Target Hash

Objective

Find the optimal Hashcat settings for cracking a specific hash type on your hardware.

Instructions

1. Choose a hash type that takes at least a few seconds to benchmark (e.g., bcrypt `-m 3200`, or WPA `-m 22000`)
2. Test all four workload profiles (`-w 1` through `-w 4`) and record the speed for each
3. Experiment with kernel tuning flags:
 - Try different `-n` values (e.g., 64, 128, 256, 512, 1024)
 - Try different `-u` values (e.g., 64, 128, 256, 512)
 - Record the speed for each combination
4. Find the combination that gives the highest speed without causing errors or overheating
5. Compare your optimal settings against Hashcat's auto-tuned defaults

6. Document the temperature behaviour under each configuration

Deliverable

A report showing all tested configurations with their speeds and temperatures. Identify the optimal configuration and explain why it outperforms (or matches) the defaults.

Self-check: You tested at least 8 different configurations. Your optimal settings are clearly identified. Temperature data is included for each test.

Task 3 — CPU vs GPU Performance Report

Objective

Create a detailed performance comparison between CPU and GPU cracking on your system.

Instructions

1. Benchmark at least 5 hash types on both CPU (`-D 1`) and GPU (`-D 2`)
2. For each hash type, calculate the speedup factor (GPU speed / CPU speed)
3. Run a real dictionary attack on both CPU and GPU for at least 2 hash types.
Record:
 - Wall-clock time to completion
 - Hashes per second
 - Power consumption (if measurable with `nvidia-smi`)
4. If you have access to multiple GPUs, test scaling: 1 GPU vs 2 GPUs. Calculate the scaling efficiency.
5. Write a summary answering:
 - What is the average GPU speedup on your system?
 - Which hash types benefit most from GPU acceleration?
 - Are there any hash types where CPU performance is competitive?
 - What are the diminishing returns of multi-GPU setups?

Deliverable

A performance report with benchmark tables, speedup calculations, and a written summary answering the questions above.

Self-check: At least 5 hash types compared. Speedup factors are calculated correctly. Your summary is supported by the data.

Submission Checklist

- Task 1: Benchmark table with 10+ hash types, exhaustion time estimates, and GPU-friendly analysis
- Task 2: Workload optimization report with tested configurations, speeds, and temperatures
- Task 3: CPU vs GPU comparison with speedup factors and written analysis
- All commands executed without errors; results are reproducible

Lesson 09: Results Analysis & Post-Processing

Results Analysis & Post-Processing

Cracking hashes is only half the job. This lesson covers how to save, organize, analyze, and report on cracked passwords using hashcat's built-in tools and Python post-processing.

Learning Objectives

- Configure **output file formats** with `--outfile-format`
 - Save and manage results with `-o` and **session management**
 - Use `--show` and `--left` to inspect cracked and uncracked hashes
 - Build **Python scripts** for cleaning, analyzing, and visualizing password data
 - Create professional **password analysis reports**
-

Output File Formats

The `--outfile-format` flag controls the structure of saved results. Choose the format that matches your reporting needs.

Value	Format	Example Output
1	Hash only	5f4dcc3b5aa765d61d8327deb882cf99
2	Plain only	password
3	Hash:Password	5f4dcc3b...:password
4	Login:Password	admin:password
5	Login:Password:Hash	admin:password:5f4dcc3b...
6	Login:Password:Hash:Salt	admin:password:5f4dcc3b...:salt123

Default is format 3 (hash:password). Use format 2 for feeding into other tools. Use format 5 or 6 for comprehensive audit reports.

Saving Results & Session Management

Hashcat provides robust mechanisms for saving output and resuming interrupted sessions.

Saving Results

```
# Save cracked passwords to a file
hashcat -m 0 hashes.txt wordlist.txt -o results.txt

# Save with specific format (login:password)
hashcat -m 0 hashes.txt wordlist.txt -o results.txt --outfile-format 4
```

Session Management

```
# Start a named session
hashcat -m 0 hashes.txt wordlist.txt --session mysession

# Resume an interrupted session
hashcat --session mysession --restore
```

Sessions save state automatically. If a long-running attack is interrupted (power loss, Ctrl+C), use `--restore` to pick up exactly where you left off.

Inspecting Results: --show and --left

After cracking, use these flags to review what was and was not recovered.

--show (Cracked)

Displays all previously cracked hashes from the potfile.

```
hashcat -m 0 --show hashes.txt
```

Output: `hash:password` pairs for every cracked entry.

--left (Uncracked)

Shows hashes that have not been cracked yet.

```
hashcat -m 0 --left hashes.txt
```

Output: remaining hashes that need further attack strategies.

Tip: Pipe `--left` output to a new file and use it as input for more aggressive attacks (larger wordlists, rule stacking, brute-force).

Python Post-Processing

Raw hashcat output needs cleaning and analysis. Python with `pandas` and `matplotlib` is ideal for this.

Cleaning Results

```
import pandas as pd

# Read hashcat output (format 3: hash:password)
df = pd.read_csv('results.txt', sep=':', header=None,
                 names=['hash', 'password'])
df.drop_duplicates(inplace=True)
df.to_csv('clean_results.csv', index=False)
print(f"Unique passwords: {df['password'].nunique()}")
```

Frequency Analysis

```
# Top 10 most common passwords
top = df['password'].value_counts().head(10)
print(top)
```

Why automate? Real-world audits may crack thousands of passwords. Manual review is impractical — scripts make analysis repeatable and auditable.

Visualization & Reporting

Transform raw data into actionable insights with charts and summary statistics.

```
import matplotlib.pyplot as plt

# Password length distribution
df['length'] = df['password'].str.len()
df['length'].hist(bins=20, color='#1f6feb', edgecolor='#0d1117')
plt.xlabel('Password Length')
plt.ylabel('Count')
plt.title('Password Length Distribution')
plt.savefig('length_distribution.png', dpi=150, bbox_inches='tight')
plt.show()
```

Key Report Metrics

- **Crack rate:** % of hashes recovered vs total
- **Top passwords:** Most common passwords (often "password", "123456")
- **Length distribution:** How many passwords are under 8 characters

- **Complexity analysis:** % with uppercase, digits, special characters
-

Complexity Analysis

Categorize cracked passwords by their complexity to identify policy weaknesses.

```
import re

def classify(pw):
    has_upper = bool(re.search(r'[A-Z]', pw))
    has_lower = bool(re.search(r'[a-z]', pw))
    has_digit = bool(re.search(r'[0-9]', pw))
    has_special = bool(re.search(r'[!@#$%^&*~.-_`~!@#$%^&*~.-_`~]', pw))
    score = sum([has_upper, has_lower, has_digit, has_special])
    if score <= 1: return 'Weak'
    if score == 2: return 'Fair'
    if score == 3: return 'Good'
    return 'Strong'

df['complexity'] = df['password'].apply(classify)
print(df['complexity'].value_counts())
```

Common finding: In most audits, 60-80% of cracked passwords fall into the "Weak" or "Fair" categories, revealing inadequate password policies.

Lab & Homework

Lab Exercises

1. **Output formats** — run attacks with different `--outfile-format` values
2. **Session management** — use `--session` and `--restore`
3. **Python cleaning** — build `clean_results.py`
4. **Analysis** — build `analyze_passwords.py` with charts

Homework Tasks

1. Full password analysis report with visualizations
2. Complexity breakdown and policy recommendations
3. Automated report generation script

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](#)

Lab Guide — Results Analysis & Post-Processing

Four exercises covering output formats, session management, data cleaning, and password analysis with Python.

Prerequisites: You need hashcat installed and working (from previous lessons), Python 3.8+, and the `pandas` and `matplotlib` packages. Install them with: `pip install pandas matplotlib`

Setup — Create Test Hashes

Before starting, generate a set of test hashes to work with throughout the lab.

Prepare test data

Create a file called `lab_hashes.txt` with MD5 hashes. You can generate these from known passwords:

```
# Generate MD5 hashes from a list of known passwords
echo -n "password" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "123456" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "qwerty" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "letmein" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "admin" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "welcome1" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "monkey" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "dragon" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "master" | md5sum | cut -d' ' -f1 >> lab_hashes.txt
echo -n "P@ssw0rd!" | md5sum | cut -d' ' -f1 >> lab_hashes.txt

# Also create a version with usernames
echo "alice:$(echo -n 'password' | md5sum | cut -d' ' -f1)" >> lab_users.txt
echo "bob:$(echo -n '123456' | md5sum | cut -d' ' -f1)" >> lab_users.txt
```



```
echo "charlie:$(echo -n 'qwerty' | md5sum | cut -d' ' -f1)" >> lab_users.txt
echo "diana:$(echo -n 'letmein' | md5sum | cut -d' ' -f1)" >> lab_users.txt
echo "admin:$(echo -n 'admin' | md5sum | cut -d' ' -f1)" >> lab_users.txt
```

Exercise 1 — Output Formats

EX 1

Explore --outfile-format options

Run the same attack with different output formats and compare the results.

1. Run a dictionary attack saving results in format 3 (default):

```
hashcat -m 0 lab_hashes.txt /usr/share/wordlists/rockyou.txt \
-o results_fmt3.txt --outfile-format 3
```

2. Run again with format 2 (plain passwords only):

```
hashcat -m 0 lab_hashes.txt /usr/share/wordlists/rockyou.txt \
-o results_fmt2.txt --outfile-format 2 --potfile-disable
```

3. Run with format 5 using the username file:

```
hashcat -m 0 lab_users.txt /usr/share/wordlists/rockyou.txt \
-o results_fmt5.txt --outfile-format 5 --username
```

4. Compare the three output files:

```
echo "=== Format 3 (hash:password) ==="
head -5 results_fmt3.txt
echo "=== Format 2 (plain only) ==="
head -5 results_fmt2.txt
echo "=== Format 5 (login:password:hash) ==="
head -5 results_fmt5.txt
```

Checkpoint: You have three output files, each showing the same cracked passwords in different formats.

Exercise 2 — Session Management

EX 2

Practice --session and --restore

1. Start a named session with a large wordlist:

```
hashcat -m 0 lab_hashes.txt /usr/share/wordlists/rockyou.txt \  
--session lab09_session -o session_results.txt
```

2. While running, press `s` to view status, then press `p` to pause
3. Press `q` or `Ctrl+C` to quit gracefully (saves checkpoint)
4. Resume the session:

```
hashcat --session lab09_session --restore
```

5. After completion, inspect what was cracked and what remains:

```
# Show cracked hashes  
hashcat -m 0 --show lab_hashes.txt  
  
# Show uncracked hashes  
hashcat -m 0 --left lab_hashes.txt  
  
# Save uncracked to a separate file  
hashcat -m 0 --left lab_hashes.txt > remaining.txt  
echo "Remaining hashes:"  
wc -l remaining.txt
```

Checkpoint: You can pause, resume, and inspect results from a named session.

Exercise 3 — Python Data Cleaning (clean_results.py)

EX 3

Build a script to clean and normalize hashcat output

Create a file called `clean_results.py` that processes raw hashcat output.

```
#!/usr/bin/env python3
"""clean_results.py - Clean and normalize hashcat output."""
import pandas as pd
import sys

def clean_results(input_file, output_file='clean_results.csv'):
    # Read hash:password format
    data = []
    with open(input_file, 'r') as f:
        for line in f:
            line = line.strip()
            if ':' in line:
                # Split on first colon only (passwords may contain colons)
                parts = line.split(':', 1)
                if len(parts) == 2:
                    data.append({'hash': parts[0], 'password': parts[1]})

    df = pd.DataFrame(data)
    print(f"Total entries read:      {len(df)}")

    # Remove duplicate hashes (same hash cracked multiple times)
    df.drop_duplicates(subset='hash', inplace=True)
    print(f"After dedup by hash:      {len(df)}")

    # Remove empty passwords
    df = df[df['password'].str.len() > 0]
```

Run the script:

```
python3 clean_results.py results_fmt3.txt
```

Checkpoint: The script reads raw hashcat output, removes duplicates, and produces a clean CSV with metadata columns.

Exercise 4 — Password Analysis (analyze_passwords.py)

Build an analysis script with frequency charts and complexity breakdown

Create `analyze_passwords.py` that reads the cleaned CSV and produces visualizations.

```
#!/usr/bin/env python3
"""analyze_passwords.py - Password frequency analysis with charts."""
import pandas as pd
import matplotlib.pyplot as plt
import re
import sys

def classify_complexity(pw):
    """Classify password into Weak/Fair/Good/Strong."""
    checks = [
        bool(re.search(r'[a-z]', str(pw))),
        bool(re.search(r'[A-Z]', str(pw))),
        bool(re.search(r'[0-9]', str(pw))),
        bool(re.search(r'^A-Za-z0-9', str(pw))),
    ]
    score = sum(checks)
    if score <= 1: return 'Weak'
    if score == 2: return 'Fair'
    if score == 3: return 'Good'
    return 'Strong'

def analyze(csv_file):
    df = pd.read_csv(csv_file)
    print(f"Analyzing {len(df)} passwords...\n")

    # --- Top 10 passwords ---
```

Run the analysis:

```
python3 analyze_passwords.py clean_results.csv
```

Checkpoint: You have a bar chart of the top passwords, a length distribution histogram, a complexity pie/bar chart, and printed summary statistics.

Tip: If you are working in a headless environment (no display), add

`plt.switch_backend('Agg')` at the top of the script so matplotlib saves files without trying to open a window.

Homework

Three tasks building on the lab: a full analysis report, complexity policy evaluation, and an automated report generator.

Complete all four lab exercises before starting the homework. These tasks extend the analysis skills from the lab into professional audit deliverables.

Task 1 — Full Password Analysis Report

Objective

Produce a comprehensive password analysis report from a larger dataset with at least 50 cracked passwords.

Instructions

1. Create a hash file with at least 50 MD5 hashes (use common passwords from a wordlist to generate them, or use the hashes from previous lab exercises)
2. Run a dictionary attack with `-o results.txt --outfile-format 3`
3. Use your `clean_results.py` to produce a clean CSV
4. Use your `analyze_passwords.py` to generate:
 - Top 10 most common passwords (bar chart)
 - Password length distribution (histogram)
 - Complexity breakdown (bar chart)
5. Write a text report (plain text or HTML) that includes:
 - Executive summary: crack rate, key findings
 - Methodology: hash type, attack mode, wordlist used
 - Analysis section with the three charts
 - At least 3 specific recommendations for improving password policy

Deliverable

A report file (`report.txt` or `report.html`) plus the three chart image files.

Task 2 — Complexity Policy Evaluation

Objective

Evaluate cracked passwords against common organizational password policies and measure compliance.

Instructions

1. Define three password policies to test against:

```
policies = {
    'Basic':    {'min_length': 6, 'require_upper': False,
                 'require_digit': False, 'require_special': False},
    'Standard': {'min_length': 8, 'require_upper': True,
                 'require_digit': True, 'require_special': False},
    'Strong':   {'min_length': 12, 'require_upper': True,
                 'require_digit': True, 'require_special': True},
}
```

2. Write a function `check_policy(password, policy)` that returns `True` if the password meets the policy requirements
3. For each policy, calculate what percentage of cracked passwords would have been compliant
4. Create a grouped bar chart showing compliance rates across all three policies
5. Answer: Which policy would have prevented the most cracks? What is the trade-off with user experience?

Deliverable

A Python script (`policy_check.py`) that outputs compliance percentages and a comparison chart.

Task 3 — Automated Report Generator

Objective

Build a single script that takes a raw hashcat output file and produces a complete analysis report automatically.

Instructions

1. Create `generate_report.py` that accepts command-line arguments:

```
python3 generate_report.py results.txt --total-hashes 100 \
--output report.html
```

2. The script should:

- Read and clean the results (remove duplicates, handle colons in passwords)
- Generate all three charts (top passwords, length distribution, complexity)
- Calculate summary statistics (crack rate, average length, policy compliance)
- Output an HTML report with embedded charts (use base64 encoding for images)

3. The generated HTML report should be self-contained (no external dependencies) and presentable to a client

4. Include error handling: what if the input file is empty? What if it uses an unexpected format?

Starter template

```
#!/usr/bin/env python3
"""generate_report.py - Automated password audit report generator."""
import argparse
import base64
import io
import pandas as pd
import matplotlib
matplotlib.use('Agg') # Non-interactive backend
import matplotlib.pyplot as plt
```



```
def parse_args():
    parser = argparse.ArgumentParser(
        description='Generate password audit report from hashcat output')
    parser.add_argument('input_file', help='Hashcat output file (format 3)')
    parser.add_argument('--total-hashes', type=int, required=True,
                        help='Total number of hashes in the original file')
    parser.add_argument('--output', default='report.html',
                        help='Output report filename')
    return parser.parse_args()

def fig_to_base64(fig):
    """Convert matplotlib figure to base64 string for HTML embedding."""
    buf = io.BytesIO()
    fig.savefig(buf, format='png', dpi=150, bbox_inches='tight')
    buf.seek(0)
    return base64.b64encode(buf.read()).decode('utf-8')
```

Deliverable

The script `generate_report.py` plus a sample generated `report.html` file.

Tip: Use the `fig_to_base64()` helper to embed charts directly in the HTML as `` so the report is a single self-contained file.

Submission Checklist

- Task 1: Full analysis report with 3 charts, executive summary, and recommendations
- Task 2: Policy compliance script with comparison chart and written analysis
- Task 3: Automated report generator script and sample HTML output
- All Python scripts run without errors

- No raw passwords or identifying information in submitted reports (use aggregate statistics only)

Lesson 10: Security Ethics & Responsible Use

Security Ethics & Responsible Use

The final lesson in the Hashcat course focuses on the ethical, legal, and professional principles that must guide every use of password recovery tools.

Learning Objectives

- Understand the **five ethical principles**: Legality, Consent, Confidentiality, Responsibility, Transparency
 - Learn the **legal frameworks** governing password recovery tools
 - Know how to **obtain proper consent** before any testing
 - Develop an **ethical use policy** for Hashcat in your organization
 - Explore industry **penetration testing frameworks**: PTES, OWASP, NIST
-

The Five Ethical Principles

Every use of Hashcat and similar tools must be grounded in these core principles. Violating any of them can lead to legal prosecution, career damage, and harm to others.

Legality

Always operate within the law. Laws governing password recovery tools vary by country — check local regulations before any engagement.

Consent

Obtain **written authorization** from the system or data owner before any testing. Never assume implicit permission.

Confidentiality

Protect all recovered passwords. Use encryption and secure storage for results. Never disclose findings without authorization.

Responsibility & Transparency

Follow cybersecurity standards and best practices. Inform clients and management about results, risks, and remediation steps.

Legal Framework

Password recovery and cracking tools occupy a complex legal space. The legality depends on jurisdiction, intent, and authorization.

Region	Key Legislation	Key Requirement
United States	Computer Fraud and Abuse Act (CFAA)	Written authorization required
European Union	GDPR + National cyber laws	Data protection compliance
United Kingdom	Computer Misuse Act 1990	Authorized access only
Germany	§202a-c StGB (Hacker Paragraph)	Tool possession legal only with legitimate purpose
Australia	Criminal Code Act 1995 (Part 10.7)	Unauthorized access is an offence

Notes: Even in countries where possessing password recovery tools is legal, using them without authorization on systems you do not own is a criminal offence. Always consult legal counsel when in doubt.

Obtaining Proper Consent

Written authorization is the single most important legal protection for a penetration tester. A proper consent document must include:

- **Scope** — Specific systems, networks, or hash files to be tested
- **Timeframe** — Exact start and end dates/times for testing
- **Methods** — Approved attack types (dictionary, mask, brute-force, etc.)
- **Boundaries** — Explicitly excluded systems and data
- **Contact information** — Emergency contacts for both parties
- **Liability** — Indemnification clauses and limitations
- **Signatures** — Authorized signatories from the system/data owner

Consent Template Tip

Always use a formal template. Have it reviewed by legal counsel. Keep signed copies in secure storage for at least the duration of any applicable statute of limitations.

Confidentiality & Secure Handling

Recovered passwords and hash files are extremely sensitive data. Mishandling them can cause severe damage.

During Testing

- Work on encrypted drives only
- Use air-gapped or isolated systems when possible
- Never store results on shared or cloud drives
- Use full-disk encryption (LUKS, BitLocker, FileVault)

After Testing

- Encrypt all output files with GPG or similar
- Securely transmit results (encrypted email, secure portal)
- Securely delete hash files and cracked passwords
- Document the chain of custody for all data

Notes: Use `shred` on Linux or secure-erase tools on other platforms. Standard file deletion does not remove data from disk.

Ethical Dilemmas in Password Recovery

Real-world engagements often present gray areas that require careful judgment.

Scenario 1: Scope Creep

You discover additional vulnerable systems outside your authorized scope. Do you test them?

Answer: No. Report the finding and request expanded authorization before proceeding.

Scenario 2: Criminal Activity

During a password audit, you discover evidence that an employee is engaged in illegal activity.

Answer: Follow your reporting obligations and the engagement contract. Consult legal counsel immediately.

Scenario 3: Weak Executive Passwords

The CEO's password is cracked in seconds. Do you include it in the report? **Answer:** Yes, but deliver sensitive findings through a secure, confidential channel to the appropriate stakeholders only.

Penetration Testing Frameworks

Industry-standard frameworks provide structure and credibility to your testing engagements.

Framework	Focus	Key Benefit
PTES	Penetration Testing Execution Standard	End-to-end pentest methodology with ethical guidelines
OWASP	Open Web Application Security Project	Web-focused testing guides, top-10 vulnerability lists
NIST SP 800-115	Technical Guide to Information Security Testing	Government-standard methodology and reporting templates

Why Use a Framework?

- Ensures **consistency** and **repeatability** across engagements
- Provides **legal defensibility** — you followed an established standard
- Builds **client confidence** through recognized methodology

- Guides **reporting** with structured templates
-

Lab & Homework

Lab Exercise

1. **Study ethical principles** and discuss ethical dilemmas in password recovery
2. **Develop an ethical use policy** for Hashcat with sections for Introduction, Legality, Consent, Confidentiality, Responsibility, Transparency, Control & Monitoring, and Conclusion
3. **Write a report** summarizing your policy and the rationale behind each section

Homework Tasks

1. Research the password cracking laws in your country and write a summary
2. Draft a consent template for a password audit engagement
3. Compare PTES, OWASP, and NIST frameworks for password testing coverage
4. Write a 1-page ethical use policy for your team or organization

Course Complete

Congratulations on completing the Hashcat course! Remember: technical skill without ethical judgment is dangerous. Always use your knowledge responsibly.

Theory

Reading time: ~25 minutes · A comprehensive guide to the ethical, legal, and professional principles governing password recovery tools

1. Introduction: Why Ethics Matter in Password Recovery

Password recovery tools like Hashcat are powerful instruments that can be used for both legitimate security purposes and malicious activities. The difference between a security professional and a criminal often comes down to a single factor: **authorization**.

This lesson establishes the ethical framework that must govern every use of password recovery tools. Understanding these principles is not optional — it is a professional and legal necessity.

The tools themselves are neutral. A hammer can build a house or break a window. Hashcat can strengthen an organization's security posture or compromise it. Your ethical judgment determines which outcome occurs.

2. The Five Ethical Principles

Every engagement involving password recovery must satisfy five core ethical principles. These principles are interdependent — violating one typically compromises the others.

2.1 Legality

All password recovery activities must comply with applicable laws and regulations. This includes:

- **National legislation** — Most countries have computer crime laws that criminalize unauthorized access to computer systems. Using Hashcat on systems without authorization is illegal in virtually every jurisdiction.
- **Industry regulations** — Certain industries (healthcare, finance, government) have additional regulations governing security testing, such as HIPAA, PCI-DSS, and FedRAMP.

- **Contractual obligations** — Engagement contracts, non-disclosure agreements, and service-level agreements may impose additional constraints on testing methods and scope.
- **International considerations** — When testing systems that span multiple jurisdictions, you must comply with the laws of each relevant country.

Warning: "I didn't know it was illegal" is never a valid defense. It is your responsibility to understand the laws that apply to your activities before you begin any testing.

2.2 Consent

Written authorization from the system or data owner is mandatory before any password recovery work begins. Verbal agreements are insufficient — they cannot be verified, and they will not protect you in court.

A proper consent document (sometimes called a "Rules of Engagement" or "Authorization to Test") must include:

- **Parties involved** — Full legal names and roles of the authorizing party and the tester
- **Scope definition** — Specific systems, IP ranges, hash files, or user accounts to be tested
- **Exclusions** — Systems or data explicitly excluded from testing
- **Time window** — Exact dates and times during which testing is authorized
- **Approved methods** — Which attack types and tools are permitted
- **Data handling** — How recovered passwords and results will be stored, transmitted, and destroyed
- **Emergency procedures** — Contact information and escalation procedures if something goes wrong
- **Liability and indemnification** — Who bears responsibility for system disruption or data exposure
- **Signatures** — Dated signatures from authorized representatives

Consent Document Template

Authorization to Perform Password Security Testing

Date: [Date]

Authorizing Party: [Organization name], represented by [Name, Title]

Testing Party: [Tester/Company name], represented by [Name, Title]

Scope: [Detailed description of systems, hash types, and accounts to be tested]

Exclusions: [Systems, accounts, or data explicitly excluded]

Time Window: [Start date/time] to [End date/time]

Approved Methods: [Dictionary attacks, mask attacks, rule-based attacks, etc.]

Data Handling: [Encryption requirements, storage location, destruction timeline]

Emergency Contact: [Name, phone, email]

Signatures:

_____ (Authorizing Party) Date: _____

_____ (Testing Party) Date: _____

2.3 Confidentiality

Recovered passwords are among the most sensitive data you will ever handle. A single leaked password can lead to identity theft, financial loss, corporate espionage, or worse.

Confidentiality requirements include:

- **Encryption at rest** — All hash files, cracked passwords, and reports must be stored on encrypted media. Use full-disk encryption (LUKS, BitLocker, FileVault) and file-level encryption (GPG, age) for additional protection.
- **Encryption in transit** — Never transmit results over unencrypted channels. Use encrypted email (PGP/GPG), secure file-sharing portals, or hand-delivered encrypted media.

- **Access control** — Restrict access to recovered passwords to only those individuals who need them for remediation. This is typically limited to the security team lead and the system owner.
- **Secure deletion** — After the engagement concludes and the client has acknowledged receipt of the report, securely delete all hash files and recovered passwords. Use tools like `shred` (Linux), `sdelete` (Windows), or `srm` (macOS) — standard file deletion does not remove data from disk.
- **Chain of custody** — Maintain a log documenting who had access to the data, when, and why. This is essential for compliance and legal defensibility.

Best Practice: Create a dedicated encrypted partition or container for each engagement. When the engagement ends, securely wipe the entire container rather than trying to track individual files.

2.4 Responsibility

Professional responsibility means adhering to established cybersecurity standards and best practices throughout every engagement. This includes:

- **Competence** — Only perform testing that you are qualified to conduct. If an engagement requires expertise you do not have, disclose this to the client.
- **Due diligence** — Thoroughly plan each engagement. Understand the target environment, potential impacts, and fallback procedures before you begin.
- **Proportionality** — Use the minimum level of testing necessary to achieve the engagement objectives. Do not use aggressive methods when less intrusive ones will suffice.
- **Documentation** — Maintain detailed records of all activities, findings, and recommendations. Your documentation should be sufficient for another qualified professional to reproduce your work.
- **Continuous learning** — Stay current with evolving tools, techniques, laws, and ethical standards. The cybersecurity landscape changes rapidly.

2.5 Transparency

Transparency means keeping clients and management fully informed about the results, risks, and limitations of your work. This includes:

- **Honest reporting** — Report all findings accurately, including those that may be uncomfortable for the client. Never minimize or exaggerate results.
- **Risk communication** — Clearly explain the risks associated with each finding. Use language the audience can understand — technical jargon is appropriate for the security team but not for the board of directors.
- **Limitations disclosure** — Be transparent about what your testing did and did not cover. A password audit does not guarantee that all weak passwords have been found.
- **Remediation guidance** — Provide actionable recommendations for each finding. A report that says "passwords are weak" without guidance on how to fix the problem is incomplete.

3. Legal Framework: A Deeper Look

Understanding the legal landscape is essential for any security professional working with password recovery tools.

3.1 United States: Computer Fraud and Abuse Act (CFAA)

The CFAA is the primary federal law governing computer crimes in the United States. Key provisions include:

- Unauthorized access to a computer system is a federal crime
- Exceeding authorized access (going beyond the scope of your permission) is also criminal
- Penalties range from fines to imprisonment depending on the severity and intent
- Civil liability is also possible — victims can sue for damages

The 2021 Supreme Court decision in *Van Buren v. United States* narrowed the interpretation of "exceeding authorized access," but the law remains broad and prosecution-friendly.

3.2 European Union: GDPR and National Laws

In the EU, password recovery activities must comply with both the General Data Protection Regulation (GDPR) and national computer crime laws. Key considerations:

- Password hashes may constitute personal data under GDPR, especially when they can be linked to identified individuals
- Processing personal data requires a lawful basis (consent, legitimate interest, etc.)
- Data Protection Impact Assessments (DPIAs) may be required before conducting password audits
- Breach notification requirements apply if recovered passwords are compromised

3.3 Other Jurisdictions

Nearly every developed country has computer crime legislation. Common themes include:

- Unauthorized access is universally prohibited
- Possession of "hacking tools" is restricted in some jurisdictions (e.g., Germany's §202c StGB)
- Intent matters — but proving legitimate intent after the fact is difficult
- Cross-border testing requires compliance with the laws of all involved jurisdictions

Key Takeaway: Always check the laws of your specific jurisdiction before any engagement. When in doubt, consult legal counsel. The cost of a legal consultation is trivial compared to the cost of a criminal prosecution.

4. Ethical Dilemmas in Password Recovery

Real-world engagements frequently present ethical dilemmas that are not covered by simple rules. Developing the judgment to navigate these situations is a critical professional skill.

4.1 Discovery of Unauthorized Activity

During a password audit, you may discover evidence of unauthorized access, insider threats, or other criminal activity. How should you respond?

- Review your engagement contract for reporting obligations
- Consult legal counsel before taking any action
- Document the finding carefully without disturbing evidence

- Report through the agreed-upon chain of communication
- Never conduct your own investigation beyond what is authorized

4.2 Pressure to Exceed Scope

A client may ask you to test additional systems that were not included in the original authorization. Even if the request comes from the same person who signed the original consent, you should:

- Stop testing on out-of-scope systems immediately
- Request a formal amendment to the authorization document
- Have the amendment signed before proceeding
- Document the request and your response

4.3 Conflicts of Interest

You may be asked to audit the security of a system you helped design, or to test the passwords of people you know personally. In such cases:

- Disclose the conflict to the client before accepting the engagement
- Recuse yourself if the conflict could compromise your objectivity
- If you proceed, ensure additional oversight from an independent party

4.4 Responsible Disclosure

If your testing reveals vulnerabilities in third-party software or services, you face a disclosure dilemma. Follow responsible disclosure practices:

- Notify the vendor privately and give them reasonable time to patch
- Follow established disclosure timelines (typically 90 days)
- Do not publicly disclose details until the vendor has had time to respond
- Coordinate with your client on how to handle third-party findings

5. Developing an Ethical Use Policy

Every organization that uses password recovery tools should have a formal ethical use policy. This document sets expectations, provides guidance, and creates accountability.

5.1 Policy Structure

A comprehensive ethical use policy for Hashcat should include the following sections:

Section	Purpose
Introduction	State the policy's purpose, scope, and the organization's commitment to ethical security practices
Legality	Outline applicable laws and regulations; require compliance with all relevant legislation
Consent	Define authorization requirements; provide templates for consent documents
Confidentiality	Specify data handling, encryption, storage, and destruction requirements
Responsibility	Define professional standards, competency requirements, and accountability measures
Transparency	Set reporting standards, communication channels, and disclosure requirements
Monitoring & Control	Describe audit procedures, logging requirements, and oversight mechanisms
Conclusion	Summarize commitments and specify review schedule for the policy

5.2 Key Elements of Each Section

Introduction: Clearly state that the policy applies to all personnel who use or have access to password recovery tools. Define the policy's scope (which tools, which environments, which teams).

Legality: List the specific laws and regulations that apply to your organization. Require legal review before any new type of engagement. Mandate training on applicable laws.

Consent: Require written authorization for every engagement. Provide standardized templates. Specify who has the authority to approve testing and how consent documents must be stored.

Confidentiality: Mandate encryption standards (e.g., AES-256 for data at rest, TLS 1.3 for data in transit). Define access control requirements. Specify data retention and destruction timelines.

Responsibility: Define minimum competency requirements for personnel. Require adherence to a recognized penetration testing framework (PTES, OWASP, NIST). Establish a code of conduct.

Transparency: Define reporting templates and timelines. Require that all findings be reported honestly and completely. Establish communication channels for different stakeholders.

Monitoring & Control: Require logging of all password recovery activities. Define audit procedures and frequency. Establish an incident response plan for ethical violations.

Conclusion: Commit to annual policy review. Require all personnel to acknowledge the policy in writing. Specify consequences for violations.

6. Penetration Testing Frameworks

Industry-standard frameworks provide structure, credibility, and consistency to security testing engagements. Here are the three most relevant frameworks for password recovery work.

6.1 PTES (Penetration Testing Execution Standard)

PTES is the most comprehensive framework for penetration testing. It covers seven phases:

1. **Pre-engagement Interactions** — Scoping, authorization, and rules of engagement
2. **Intelligence Gathering** — Passive and active reconnaissance
3. **Threat Modeling** — Identifying potential attack vectors
4. **Vulnerability Analysis** — Finding weaknesses in the target
5. **Exploitation** — Attempting to leverage vulnerabilities (including password cracking)
6. **Post-Exploitation** — Assessing the impact of successful exploitation
7. **Reporting** — Documenting findings and recommendations

Password recovery with Hashcat typically falls within the Exploitation and Post-Exploitation phases.

6.2 OWASP (Open Web Application Security Project)

OWASP provides extensive guidance for web application security testing. Relevant resources include:

- **OWASP Testing Guide** — Comprehensive methodology for web application testing
- **OWASP Top 10** — The most critical web application security risks (includes authentication weaknesses)
- **OWASP Password Storage Cheat Sheet** — Best practices for storing passwords securely
- **OWASP Authentication Cheat Sheet** — Guidance on implementing secure authentication

6.3 NIST SP 800-115

NIST Special Publication 800-115, "Technical Guide to Information Security Testing and Assessment," provides a government-standard methodology for security testing. Key sections include:

- **Planning** — Defining objectives, scope, and approach
- **Discovery** — Identifying systems and vulnerabilities
- **Attack** — Attempting to exploit vulnerabilities (password testing falls here)
- **Reporting** — Documenting results with standardized templates

NIST is particularly relevant for organizations working with US government agencies or contractors.

Framework Comparison: PTES provides the most detailed methodology for penetration testing. OWASP is best for web application-specific testing. NIST is essential for government compliance. Many professionals combine elements from all three frameworks.

7. Summary

The ethical use of password recovery tools is not a constraint on your work — it is what makes your work valuable. Without ethics, password cracking is simply hacking. With ethics, it becomes a professional service that strengthens security.

Key takeaways:

- **Always operate within the law** — Know and comply with applicable legislation
- **Always obtain written consent** — Never test without proper authorization
- **Always protect sensitive data** — Encrypt, control access, and securely delete
- **Always act responsibly** — Follow established standards and maintain competence
- **Always be transparent** — Report honestly and provide actionable guidance
- **Adopt a recognized framework** — Use PTES, OWASP, or NIST to structure your work
- **Develop a formal policy** — Document and enforce ethical standards in your organization

"With great power comes great responsibility." This principle is not just a slogan — it is the foundation of ethical cybersecurity practice.

Lab Guide

Time estimate: 60–90 minutes · Format: Discussion, policy writing, and report ·

Prerequisites: Review the Theory page before starting

Lab Overview

In this lab, you will study ethical principles, discuss real-world dilemmas, and develop a formal ethical use policy for Hashcat within your organization or team. This is a writing and analysis exercise — no password cracking is involved.

Note: This lab is best completed in a group setting to encourage discussion. If working alone, write your responses as if presenting to a team and consider multiple perspectives for each dilemma.

Part 1: Study Ethical Principles (15 minutes)

Task 1.1: Principle Review

Goal: Internalize the five ethical principles and understand how they apply to password recovery.

Instructions

1. Read through the five ethical principles in the Theory page: **Legality, Consent, Confidentiality, Responsibility, Transparency**.
2. For each principle, write one sentence explaining why it matters specifically for Hashcat usage (not just security in general).
3. Rank the five principles in order of importance from your perspective. Be prepared to defend your ranking.

Checkpoint: You should have 5 principle descriptions and a ranked list with justification.

Part 2: Discuss Ethical Dilemmas (20 minutes)

Task 2.1: Scenario Analysis

Goal: Develop ethical judgment by analyzing realistic scenarios.

Instructions

For each scenario below, write a response that addresses: (a) What should you do? (b) Which ethical principle(s) apply? (c) What are the consequences of getting it wrong?

Scenario A: The Overeager Manager

Your manager asks you to run Hashcat against the company's Active Directory password hashes to "see how strong they are." They say verbal approval from the CTO is sufficient. No written authorization exists, and no scope has been defined.

Scenario B: The Curious Colleague

A colleague sees you running Hashcat during an authorized engagement and asks if you can "quickly check" if their ex-partner's password appears in a leaked database they found online.

Scenario C: The Time Pressure

You are conducting an authorized password audit. With one day left in the engagement window, you have only cracked 40% of the hashes. The client asks you to extend testing by a week. The original authorization document specifies the end date as tomorrow.

Scenario D: The Unexpected Discovery

During an authorized password audit of employee accounts, you crack the password of an account that appears to be a backdoor — it does not correspond to any known employee and has elevated privileges.

Task 2.2: Create Your Own Dilemma

Goal: Deepen understanding by creating a novel ethical scenario.

Instructions

1. Write a realistic ethical dilemma related to Hashcat usage that is not covered by the scenarios above.
2. Identify which ethical principles it tests.
3. Write a recommended course of action.

Checkpoint: You should have analysis of all 4 scenarios plus 1 original dilemma.

Part 3: Develop an Ethical Use Policy (30–40 minutes)

Tip: Think of this policy as a real document that your team would actually use. Be specific enough to be actionable, but general enough to cover the range of engagements your team handles.

Task 3.1: Define Policy Goals

Goal: Establish what your policy aims to achieve.

Instructions

1. Write 3–5 specific goals for your ethical use policy. Examples:
 - Ensure all Hashcat usage complies with applicable laws
 - Protect the confidentiality of recovered passwords
 - Establish clear accountability for all testing activities

2. Define the audience: Who will read and follow this policy?
3. Define the scope: What tools, activities, and environments does this policy cover?

Task 3.2: Write the Policy

Goal: Create a complete, actionable ethical use policy for Hashcat.

Required Structure

Your policy must include the following eight sections:

#	Section	Minimum Content
1	Introduction	Purpose statement, scope, definitions of key terms
2	Legality	Applicable laws, compliance requirements, legal review process
3	Consent	Authorization requirements, consent template, approval workflow
4	Confidentiality	Data classification, encryption standards, access controls, destruction procedures
5	Responsibility	Roles and responsibilities, competency requirements, code of conduct
6	Transparency	Reporting requirements, communication channels, disclosure standards
7	Control & Monitoring	Activity logging, audit procedures, violation handling, incident response
8	Conclusion	Acknowledgment requirement, review schedule, version control

Writing Guidelines

- Each section should be 100–200 words

- Use clear, direct language — avoid jargon where possible
- Include specific, measurable requirements (e.g., "AES-256 encryption" not just "strong encryption")
- Reference applicable frameworks (PTES, OWASP, NIST) where relevant

Checkpoint: Your policy should be approximately 1,000–1,500 words across all 8 sections.

Task 3.3: Review and Approve

Goal: Simulate the review and approval process for a real policy.

Instructions

1. **Self-review:** Read your policy from the perspective of a new team member. Is everything clear? Are there gaps?
2. **Peer review (if working in a group):** Exchange policies with another participant. Provide written feedback on:
 - Clarity and completeness of each section
 - Specific, actionable requirements vs. vague statements
 - Missing scenarios or edge cases
 - Compliance with the five ethical principles
3. **Revision:** Incorporate feedback and produce a final version.
4. **Sign-off:** Add a version number, date, and author name to the final document.

Part 4: Write a Report (10–15 minutes)

Task 4.1: Lab Report

Goal: Summarize your work and reflect on what you learned.

Report Structure

1. **Executive Summary** (2–3 sentences) — What did you create and why?

2. **Dilemma Analysis Summary** — Key insights from the ethical dilemma discussions
3. **Policy Overview** — Brief summary of each section of your ethical use policy
4. **Lessons Learned** — What surprised you? What was hardest to write? What would you change in a real-world policy?
5. **Framework Alignment** — Which penetration testing framework (PTES, OWASP, NIST) best aligns with your policy, and why?

Checkpoint: Your report should be 300–500 words.

Deliverables Checklist

Before you finish, verify you have completed:

- Principle review with descriptions and ranking (Part 1)
- Analysis of 4 ethical dilemmas + 1 original dilemma (Part 2)
- Complete ethical use policy with all 8 sections (Part 3)
- Peer review feedback (if applicable) (Part 3)
- Lab report with all 5 sections (Part 4)

Homework

Complete these tasks after the lab. Focus on critical thinking and real-world application of ethical principles. Written tasks should be clear and well-structured.

Task 1: Legal Research Report

Goal: Understand the legal landscape governing password recovery tools in your jurisdiction.

Instructions

1. Research the laws in your country (or a country of your choice) that regulate the use of password cracking tools. Consider:
 - Computer fraud and abuse laws (e.g., CFAA in the US, Computer Misuse Act in the UK)
 - Data protection regulations (e.g., GDPR in the EU)
 - Penetration testing exemptions or safe harbours
2. Write a 300–500 word summary covering:
 - Which specific laws apply to password recovery tools?
 - Under what conditions is their use legal?
 - What are the penalties for unauthorized use?
 - Are there any safe harbour provisions for security researchers?

✓ **Self-check: Your report cites at least 2 specific laws or regulations by name and explains how they apply to Hashcat usage.**

Task 2: Ethical Dilemma Analysis

Goal: Practice ethical reasoning in realistic cybersecurity scenarios.

Instructions

For each of the following scenarios, write 100–150 words explaining: (a) what ethical principles are at stake, (b) what you would do, and (c) why.

1. **Scenario A:** Your manager asks you to crack password hashes from a competitor's leaked database "to check if any of our employees reused passwords." You have no written authorization from the employees or the competitor.
2. **Scenario B:** During an authorized penetration test, you recover the CEO's password and discover it is used across multiple personal accounts (you can tell from the password pattern). The scope of the engagement only covers corporate systems.
3. **Scenario C:** A colleague shares a list of "practice hashes" for a training exercise. You suspect some of them may be real production hashes from a client system, not synthetic test data.

✓ **Self-check:** Each response explicitly names the ethical principles involved (legality, consent, confidentiality, responsibility, or transparency) and provides a clear course of action.

Task 3: Organizational Security Policy

Goal: Create a professional-quality policy document for ethical use of password auditing tools.

Instructions

Write a 1–2 page "Acceptable Use Policy for Password Auditing Tools" suitable for a fictional organization. Your policy must include:

1. **Purpose & Scope:** What the policy covers and why it exists
2. **Authorization Requirements:** Who can approve password auditing, what documentation is required
3. **Consent & Notification:** How affected parties are informed
4. **Data Handling:** How recovered passwords and hash files must be stored, encrypted, and destroyed
5. **Approved Tools & Methods:** Which tools are permitted, what attack types are allowed

- 6. **Reporting:** What must be documented and reported after an audit
- 7. **Violations & Consequences:** What happens if the policy is breached
- 8. **Review Schedule:** How often the policy is reviewed and updated

✓ **Self-check:** Your policy covers all 8 sections listed above and could realistically be adopted by a small to mid-size company.

Grading Criteria

Criterion	Weight	Description
Legal accuracy	25%	Correct identification and explanation of relevant laws
Ethical reasoning	25%	Clear application of ethical principles to scenarios
Policy completeness	30%	All required sections present with actionable content
Writing quality	20%	Clear, professional, well-organized writing