**Day 19 — Password Cracking (Ethical Auditing & Defense)**

This guide is intended solely for lawful security auditing, education, and red‑team exercises on systems you own or are explicitly authorized to test. Do not attempt any activity without written permission and a defined scope. Misuse can be illegal and harmful.

# 1. Key Concepts

## 1.1 Password

A password is a secret string used to authenticate a user and control access to systems, accounts, or data.

## 1.2 Hash

A hash is the output of a one‑way mathematical function that maps input data (e.g., a password) to a fixed‑length value. It is designed to be computationally infeasible to invert (derive the original input from the hash). Key properties include determinism, preimage resistance, and collision resistance.

# 2. Hashing vs. Encryption

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Hashing** | **Encryption** |
| Process | Transforms input into a fixed‑length digest using a one‑way function. | Transforms plaintext into ciphertext using a reversible algorithm and a key. |
| Primary Purpose | Integrity verification, password storage (with salts/KDFs). | Confidentiality of data during storage or transit. |
| Direction | One‑way (non‑reversible). | Two‑way (decryptable with the correct key). |
| Key Required | No key for basic hashing; modern password storage uses salts and KDF parameters. | Yes—symmetric keys or public/private key pairs. |
| Output Form | Fixed‑length hash (digest). | Ciphertext similar in size to input (often larger). |
| Typical Use Cases | Password storage, integrity checks, digital signatures (within MAC schemes). | Secure messaging, file/database encryption, TLS/HTTPS. |
| Examples | SHA‑256, SHA‑3, BLAKE2, bcrypt, scrypt, Argon2. | AES, ChaCha20, RSA, ECC. |
| Reversibility | Infeasible to reverse; guessing attempts compare candidate → hash. | Decryption restores original plaintext with key. |
| Security Considerations | Use salts and slow KDFs (bcrypt/scrypt/Argon2) to resist cracking. | Protect keys; use authenticated modes (e.g., AEAD) and rotate keys. |

# 3. What is Password Cracking?

Password cracking is the process of attempting to recover the original password from stored representations (hashes). In an ethical auditing context, this helps assess the strength of password policies and storage practices. Common techniques include dictionary, brute‑force, mask, and hybrid attacks. GPU acceleration can dramatically speed up guessing, which is why modern defenses rely on slow, memory‑hard KDFs.

# 4. Ethical Audit Workflow (High‑Level)

1. Define scope and obtain written authorization.  
   Inventory and classify password data within scope; never remove or exfiltrate sensitive data.  
   Identify hash/KDF types and parameters (e.g., salt, iteration count).  
   Prepare test data and approved wordlists in a secure environment.  
   Run controlled audits; record parameters, durations, and findings.  
   Report defensively: recommend stronger KDFs, password length, MFA, and rate-limiting.

# 5. Tool Overviews (High‑Level)

## 5.1 Hashcat (GPU‑accelerated)

Hashcat is a GPU‑accelerated password auditing tool. It supports many hash types and attack modes. It benefits from discrete GPUs and may not run or may be very slow without one.

Note: If your system lacks a supported GPU, you may encounter errors or poor performance. Consider CPU‑based auditing or use a lab with compatible hardware.

Generic pattern for a dictionary attack (non‑executable template):

hashcat -a <ATTACK\_MODE> -m <HASH\_MODE> <HASH\_FILE> <WORDLIST>

Where:  
• <ATTACK\_MODE> is typically 0 for a straight dictionary mode.  
• <HASH\_MODE> identifies the hash/KDF type (consult the tool’s help to locate the correct mode).  
• <HASH\_FILE> is a text file containing hashes in the expected format.  
• <WORDLIST> is an approved dictionary of candidate passwords.

To look up supported modes, use the built‑in help and search for your target format (do not guess):

hashcat --help  
# (Search within the help output for the relevant hash/KDF type.)

Online lookup services can sometimes identify trivial hashes. Never upload proprietary or sensitive hashes to third‑party sites.

## 5.2 John the Ripper (CPU‑based)

John the Ripper (JtR) is a CPU‑focused auditing tool with broad format support. It provides various cracking modes and integrates format‑specific ‘\*2john’ utilities to extract hash material from files.

Generic pattern for a dictionary mode audit (non‑executable template):

john <HASH\_FILE> --format=<FORMAT\_NAME> --wordlist=<WORDLIST>

• <FORMAT\_NAME> specifies the hash algorithm (check JtR help for valid options).  
• <WORDLIST> points to the approved candidate list.  
Use `john --help` to review available formats and options.

# 6. Format‑Specific Extractors (\*2john Utilities)

Many file types store credentials indirectly. The \*2john utilities transform these files into hash representations suitable for auditing.

## 6.1 ZIP Archives → zip2john

Concept: Convert a password‑protected ZIP into a hash line that an auditing tool can evaluate.

zip2john <ZIP\_FILE> > <HASH\_FILE>

Note: Only process files that are in scope and that you have authorization to assess.

## 6.2 SSH Private Keys → ssh2john

Concept: Convert an encrypted SSH private key (e.g., id\_rsa) into a hash for auditing the passphrase’s strength.

ssh2john <SSH\_PRIVATE\_KEY> > <HASH\_FILE>

## 6.3 KeePass Databases (.kdbx) → keepass2john

Concept: KeePass uses strong encryption with a composite master key. The database file (.kdbx) can be transformed into a hash representation for auditing the master passphrase.

keepass2john <KDBX\_FILE> > <HASH\_FILE>

Note: Never upload .kdbx files to third‑party sites. Handle them in a secure lab environment only.

# 7. Hash Identification (High‑Level)

Before auditing, identify the hash/KDF type to avoid false results. Utilities exist that can suggest likely formats based on hash shape. However, the most reliable method is consulting system/application documentation or configuration to determine exactly how passwords are stored.

# Concept: run a hash identification utility, paste a hash, review suggested formats.  
# Always verify using official documentation.

# 8. Defensive Recommendations

* Use modern, slow, memory‑hard KDFs for password storage (e.g., Argon2, scrypt, or bcrypt with strong parameters).
* Require long, unique passphrases; deploy password managers for users.
* Enforce MFA for privileged and remote access.
* Implement lockouts, rate‑limits, and monitoring on authentication endpoints.
* Salt every password; consider an application‑level ‘pepper’ stored separately from the database.
* Disable legacy protocols where possible and segment/monitor authentication infrastructure.
* Regularly audit password policies and educate users on phishing and reuse risks.

# 9. Reporting & Ethics

Document scope, methods, parameters, and time/cost settings. Record only what is necessary, protect sensitive data, and disclose findings responsibly. Provide clear, prioritized remediation steps and tracking for follow‑up.