Cryptography Basics, Key Management, and Encryption Tools with Practical Implementation

Overview

In the digital era where **data breaches, ransomware, phishing**, and **spoofing** attacks dominate the cyber world, **cryptography** stands as the silent guardian. It ensures **confidentiality, integrity, authenticity, and non-repudiation** of data — whether in transit or at rest. This report dives into the **basics of cryptography**, types of encryption, **key management**, and the **most-used tools**, topped off with a practical understanding of how these elements work in the real world.

Objectives

This report covers:

- **Symmetric & Asymmetric Encryption**
- **Hashing Techniques (MD5, SHA-256, Bcrypt)**
- **Public Key Infrastructure (PKI)**
- **Digital Certificates**
- **TLS/SSL Handshake**
- **Encryption Tools (GPG, OpenSSL, etc.)**
- **Practical Implementation & Real-world Relevance**

1. What is Cryptography?

Q: Why do we need cryptography in cybersecurity?

> Cryptography is the art of **securing communication** in the presence of adversaries. It transforms **plaintext** → **ciphertext** to prevent unauthorized access.

It primarily solves these 4 problems:

- **Confidentiality:** Keeps data secret
- **Integrity:** Makes sure data isn't altered
- **Authentication:** Confirms identities
- **Non-repudiation:** No one can deny sending the data

2. Symmetric Encryption

Q: What is symmetric encryption and why is it fast?

- **One key** is used to **encrypt and decrypt** data.
- It's super fast — used in **bulk data transfer** like file systems, VPNs, etc.

Common Algorithms:

Algorithm	Key Length	Use Case
AES (Advanced Encryption Standard)	128/192/256 bits	Secure WiFi, Disk Encryption
DES (Data Encryption Standard)	56 bits (weak)	Deprecated
3DES (Triple DES)	168 bits	Legacy systems

3. Asymmetric Encryption

Q: How does asymmetric encryption solve key sharing problems?

- Uses a **key pair**: **Public Key** (to encrypt) + **Private Key** (to decrypt)
- Ideal for **secure communications**, **digital signatures**, **PKI**, etc.

Common Algorithms:

Algorithm	Key Size	Use Case
RSA (Rivest-Shamir-Adleman)	1024–4096 bits	Email, HTTPS
ECC (Elliptic Curve Cryptography)	256 bits+	Mobile Devices, Bitcoin
Diffie-Hellman	Key Exchange Protocol	VPN, SSH

4. Hashing Algorithms

Q: What is hashing and why is it one-way?

- **Hashing** creates a **fixed-size digest** from input.
- It's **irreversible** — used for **password storage, data integrity, digital signatures.**

Common Hash Functions:

Hash	Size	Usage
MD5	128-bit	Deprecated (collisions)

SHA-256	256-bit	Blockchain, Secure Hashing
Bcrypt	Variable (adaptive)	Password hashing with salt & cost factor

5. What is PKI (Public Key Infrastructure)?

Q: How does PKI enable secure identity verification over the internet?

- PKI is a **framework** that manages **digital certificates** and **key pairs**.
- Enables secure **email, websites (HTTPS), VPNs, etc.**

Components of PKI:

- **Certificate Authority (CA)** – Issues and signs certificates
- **Registration Authority (RA)** – Verifies identity before certificate issuance
- **Public/Private Keys** – Core of encryption
- **Digital Certificates** – ID card of entities online
- **CRL / OCSP** – Certificate Revocation mechanisms

6. What are Digital Certificates?

Q: How do digital certificates prove someone's identity online?

A **Digital Certificate** is a file that uses PKI to:

- Bind a **public key** to an **identity**
- Prove that a **website/server is trusted**
- Issued by a **CA**

Formats:

- **X.509** → Standard format for SSL/TLS
- **.crt, .cer, .pem** → Common file types

7. TLS/SSL Handshake

Q: How does your browser know it's safe to connect to a website?

TLS/SSL handshake happens before any data is exchanged.

Key Steps:

1. Client says "Hello" 🖐
2. Server shares its **digital certificate**
3. Client validates certificate using **CA's public key**
4. Key exchange happens (RSA, DH, or ECDHE)
5. Secure session is established using **symmetric encryption**

🔐 HTTPS = HTTP + TLS

8. Key Management Tools

Q: Why is managing encryption keys more important than encryption itself?

If keys are leaked, encryption is useless. Key Management ensures:

- Secure **generation**
- Secure **storage**
- Controlled **access**
- Secure **distribution**

9. GPG Full Practical Flow:

GPG stands for GNU Privacy Guard

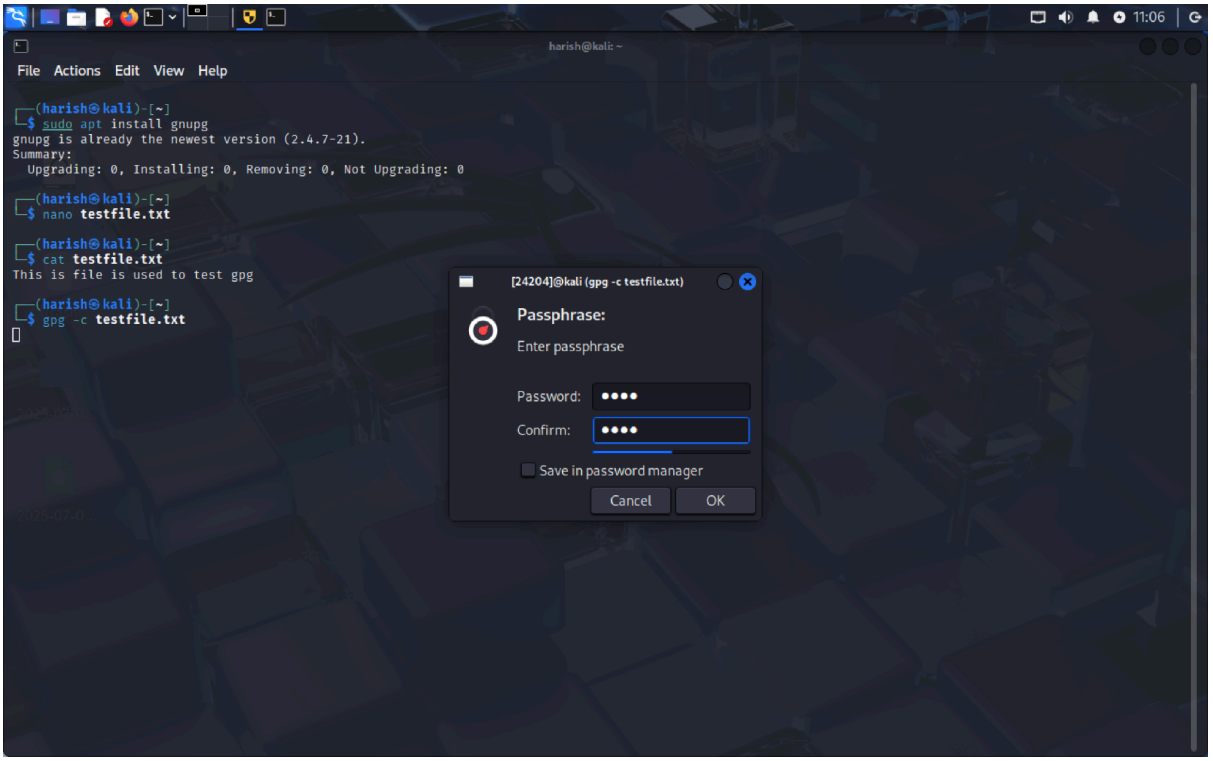
It's an **open-source** tool that implements the **OpenPGP** standard for:

- **Encrypting data** (so only intended receivers can read it)
- **Signing data** (so receivers can verify it came from *you*)
- **Verifying digital signatures**
- **Managing cryptographic keys** (key pairs, revocation, trust, etc.)

1. Install GnuPG

Command: sudo apt install gnupg -y && nano testfile.txt

gpg-Command: gpg -c testfile.txt => testfile.txt.gpg



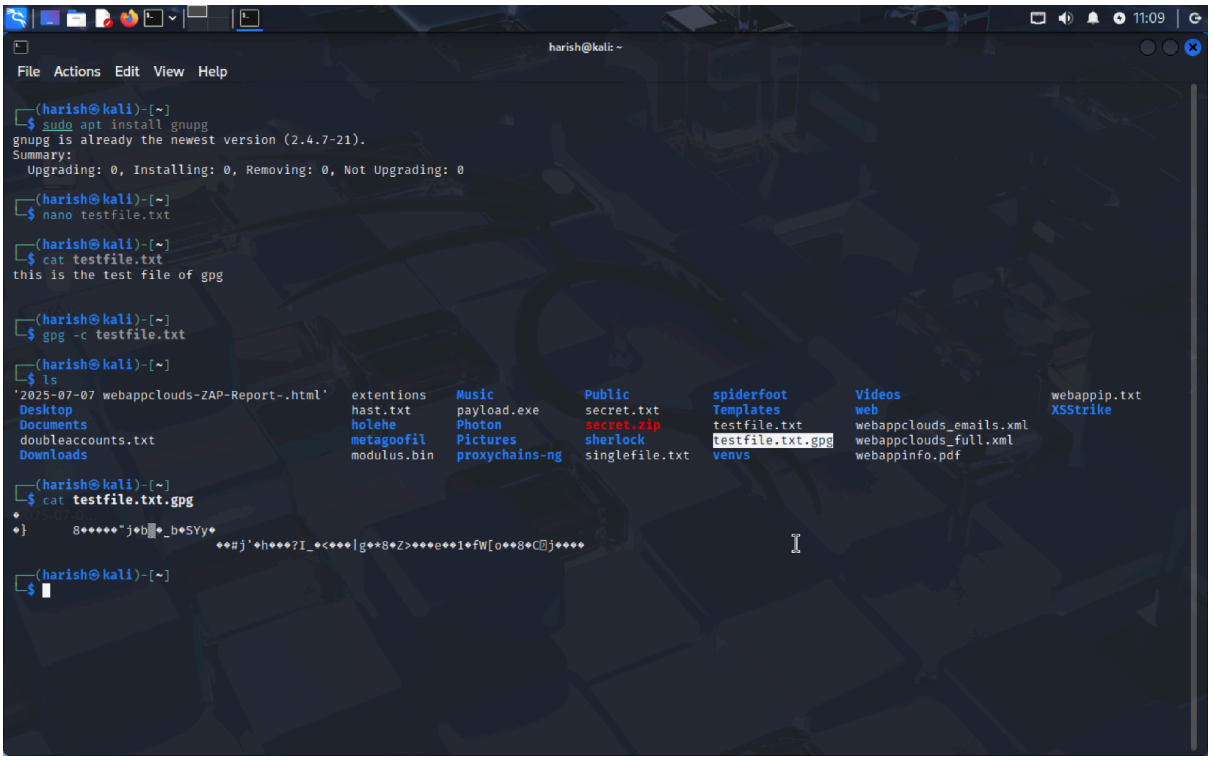


Fig: Installation and encryption

Key Point:

- `gpg -c` = **symmetric encryption**
- No public/private key involved
- Quick, simple, password-protected

GPG-Command to decrypt: gpg -d testfile.txt.gpg

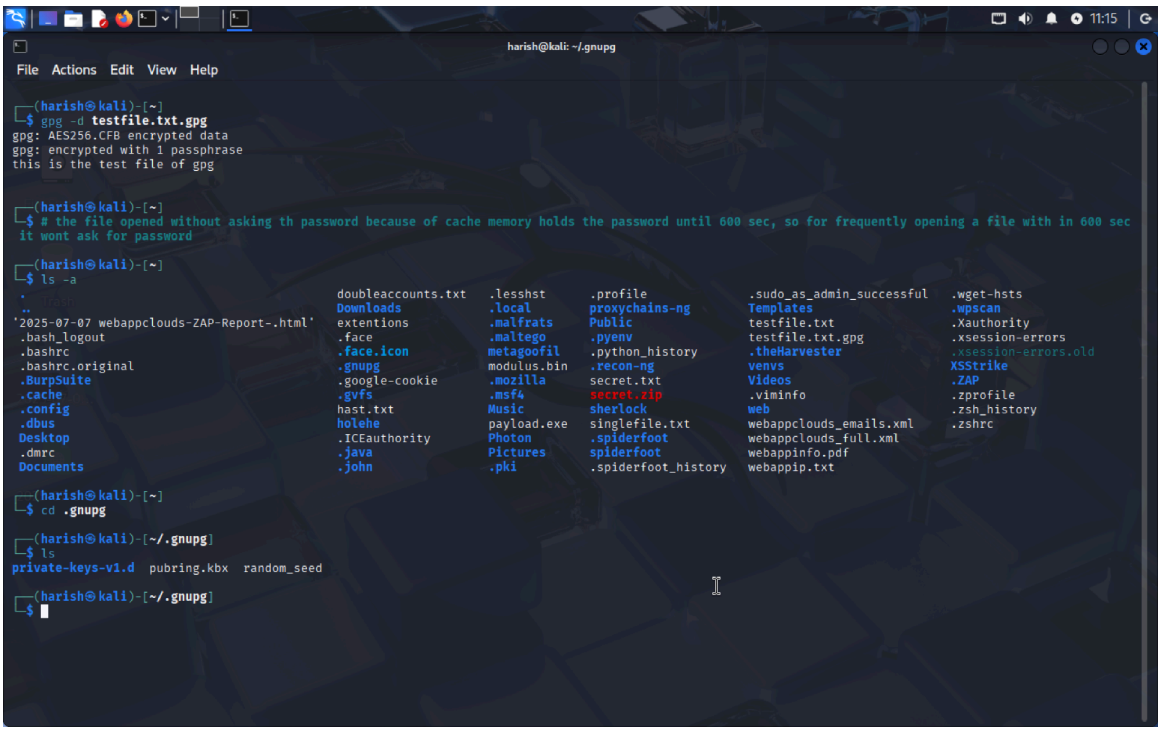


Fig: Decrypting the file but not asking for password

GPG Passphrase Caching Note:

After encrypting or decrypting a file using GPG, the **passphrase is temporarily cached** in memory by the gpg-agent.

By **default**, this cache **lasts for 600 seconds (10 minutes)**. So, if you try to decrypt multiple times within that window, GPG won't ask for the passphrase again.

This can be a **security risk** if you're on a shared system.

To modify this behavior:

1. **Find the hidden GPG directory** (usually ~/.gnupg/)
2. Create or edit the config file:

   ~/.gnupg/gpg-agent.conf
3. Set your desired cache timeout

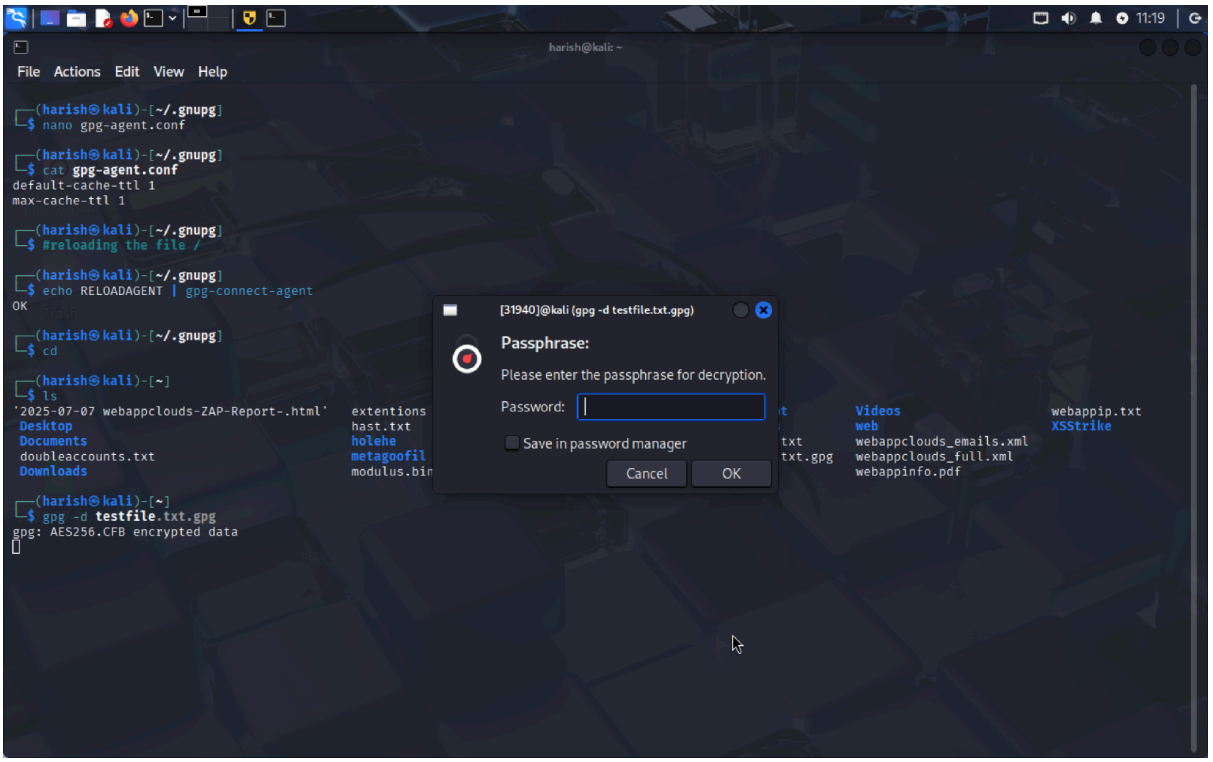


Fig: Cache config && Reload the file

GPG Encryption for Folders

Command: gpgtar --encrypt --symmetric - -ouptut testfolder.gpg testfolder

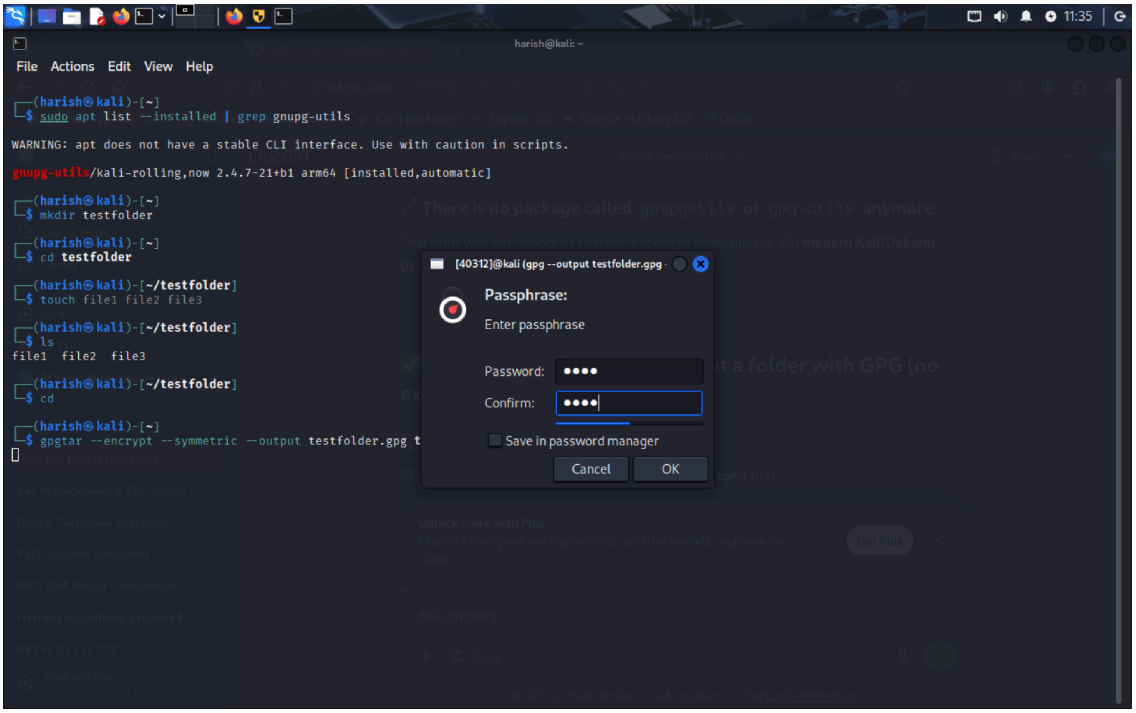


Fig: symmetric encryption to folder

Note: gpgtar is designed to encrypt only to the folders into archive

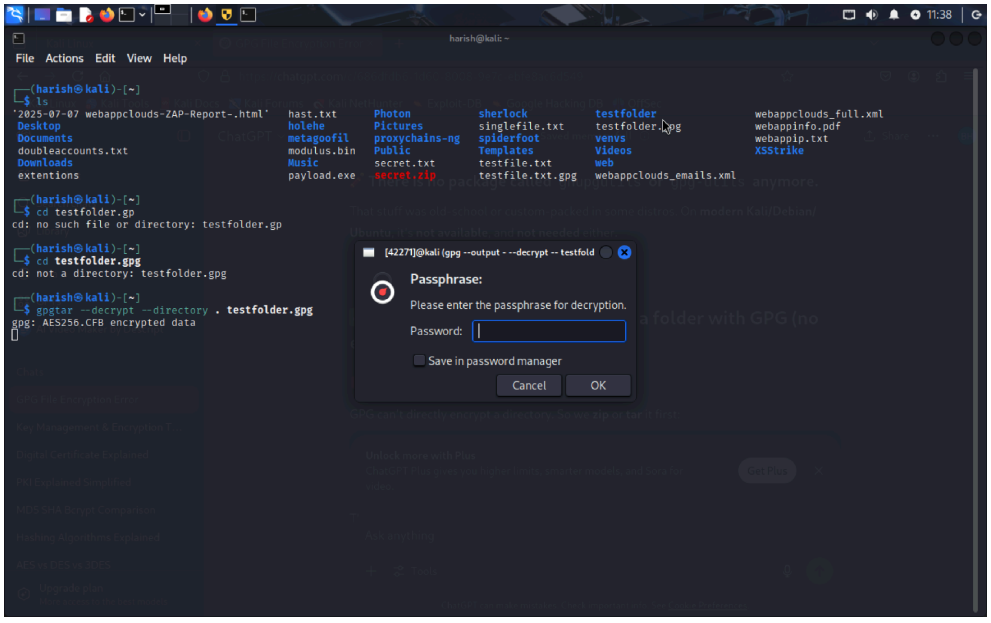


Fig: Decrypting the folder

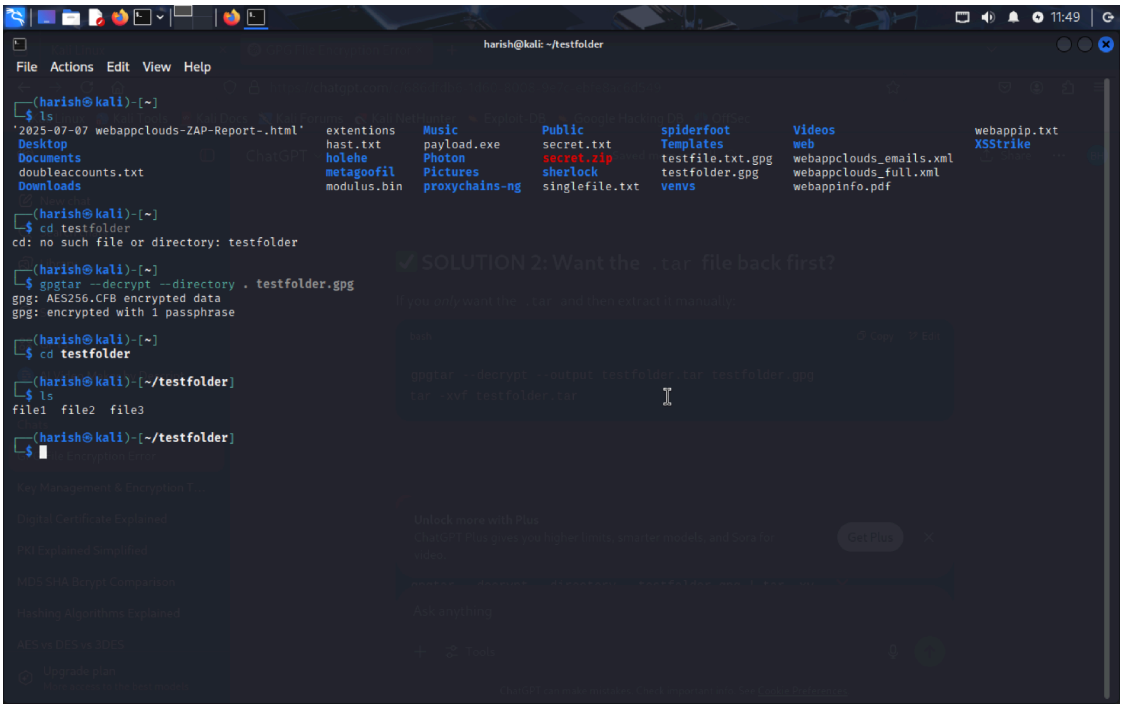


Fig: Reading the folder

GPG Asymmetric Encryption Practical

1. Generate GPG Key Pair:

Command: gpg --full-generate-key

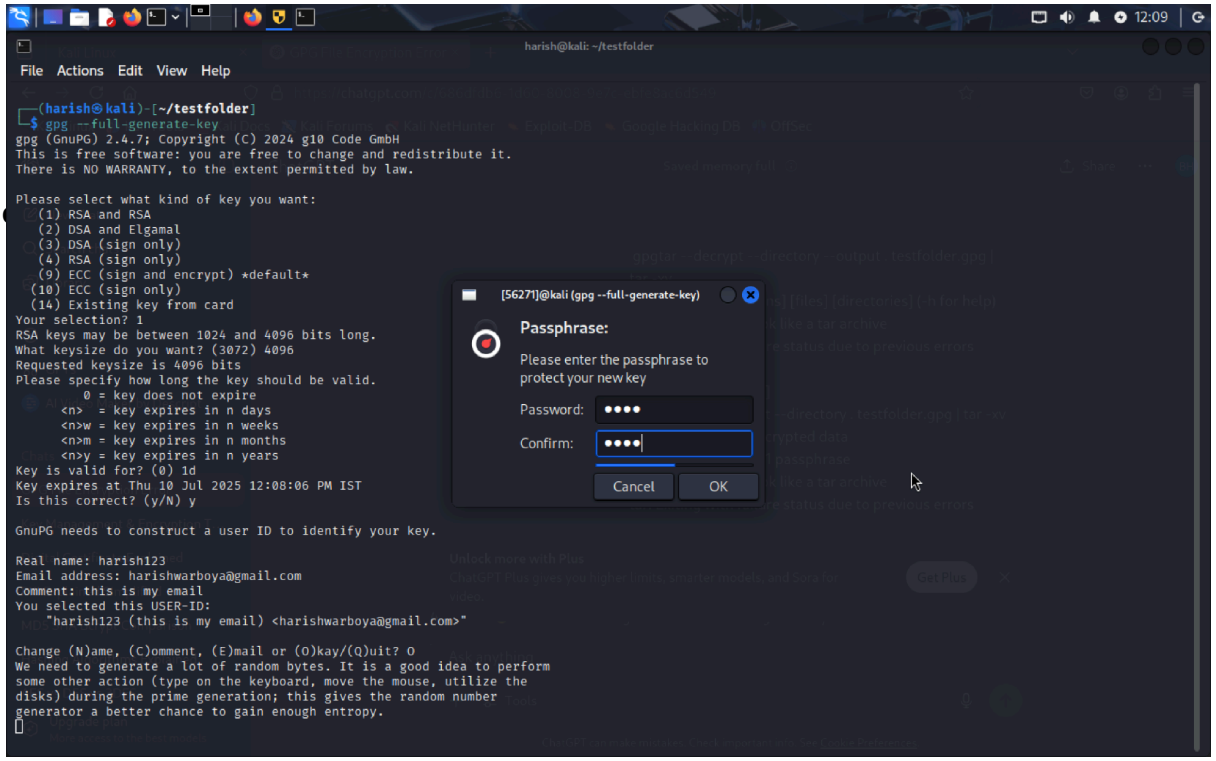


Fig: Keys Generating

Chose: `(1) RSA and RSA`

Key size: `4096`

Name: `harish123`

Email: `harishwarboya@gmail.com`

Expiry: `1d`

Passphrase:set by you to **lock your private key**

Created:

- Public key → for sharing
- Private key → for decrypting
- Passphrase → to protect the private key

Export Your Public Key:

Command: gpg --armor --export harishwarboya@gmail.com > harish_pub.asc

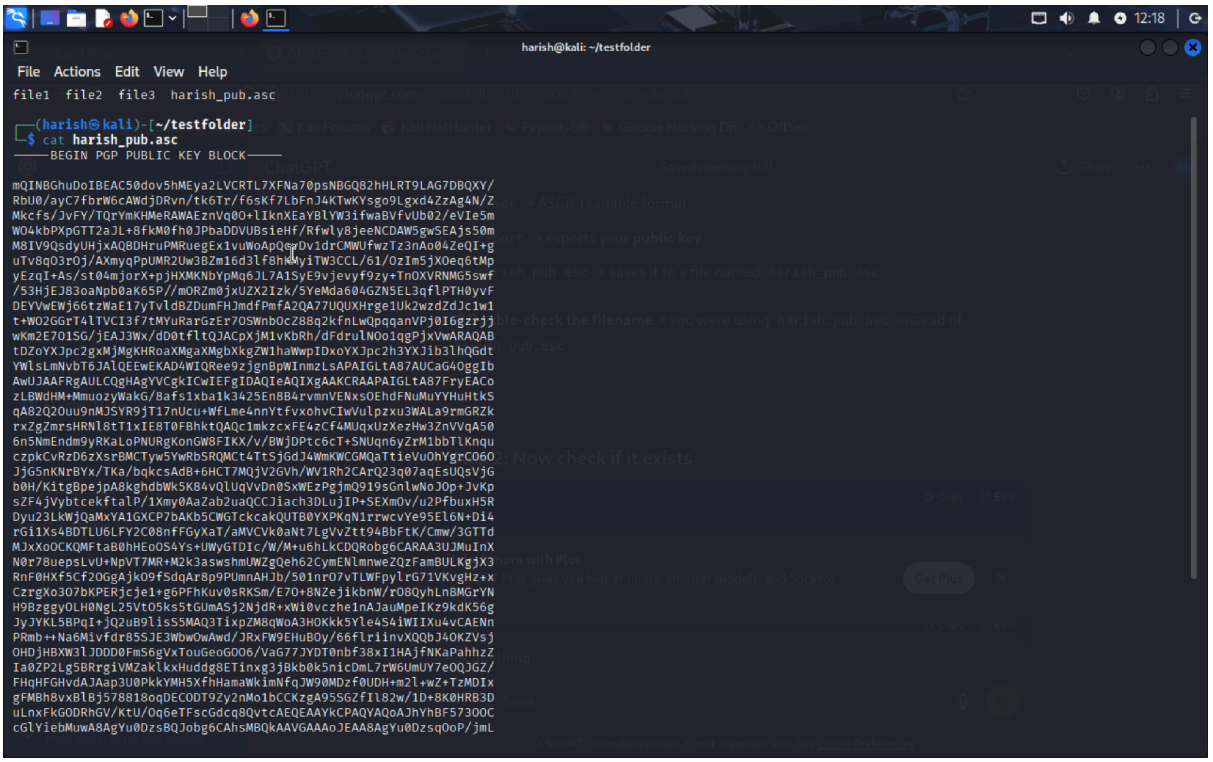


Fig: Public Key

Note: The Public key is shared with sender and sender is sending message

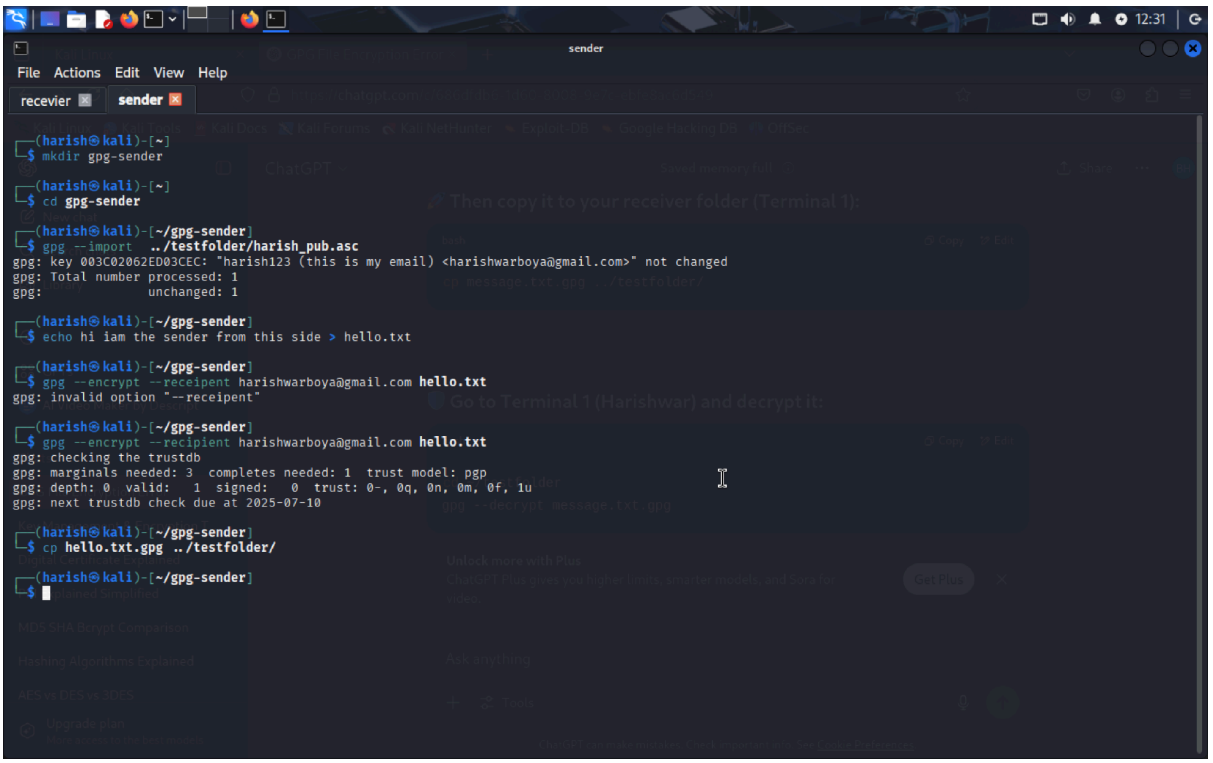


Fig: Sender View

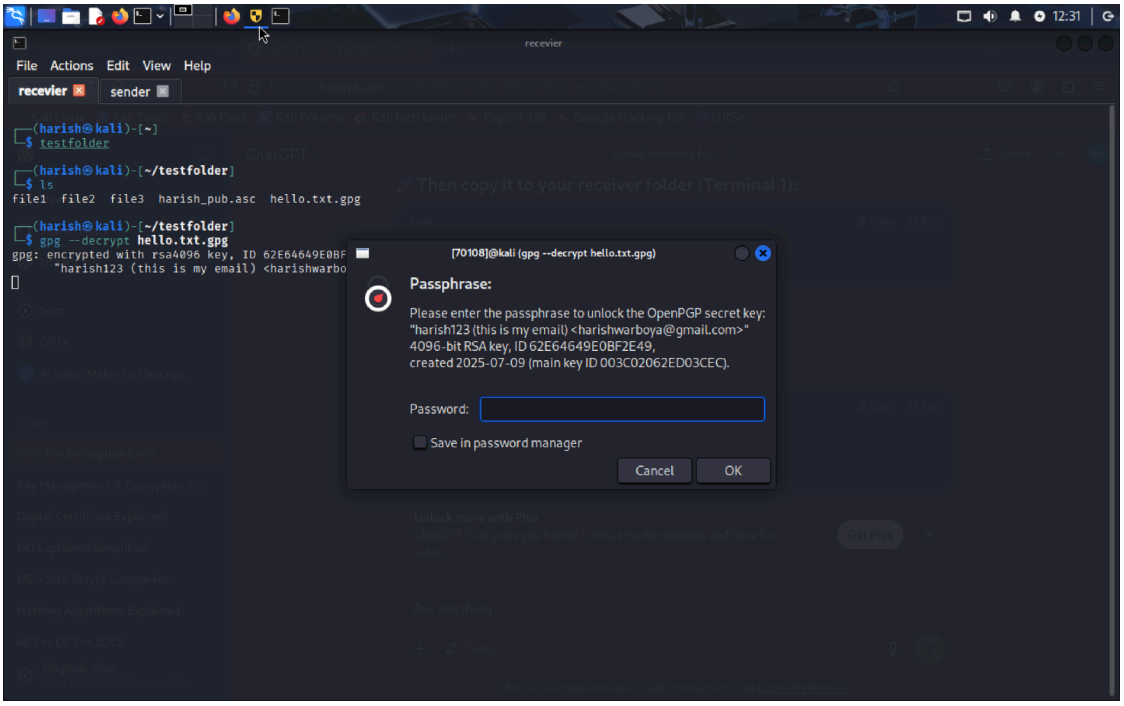


Fig: Receiver's View

OpenSSL

A powerful open-source cryptographic toolkit for encryption, key generation, hashing, SSL/TLS, and certificate management.

Key Pair Generation (RSA):

Command:

1: openssl genrsa -out rsaprivate.key (which is only used to create rsa private keys)

2:openssl genpkey -alogrithms <alogirhtms> -out ,,, (which is modern can generate different keys)

Note: This is only used to generate Asymmetric keys (no symmetric keys)

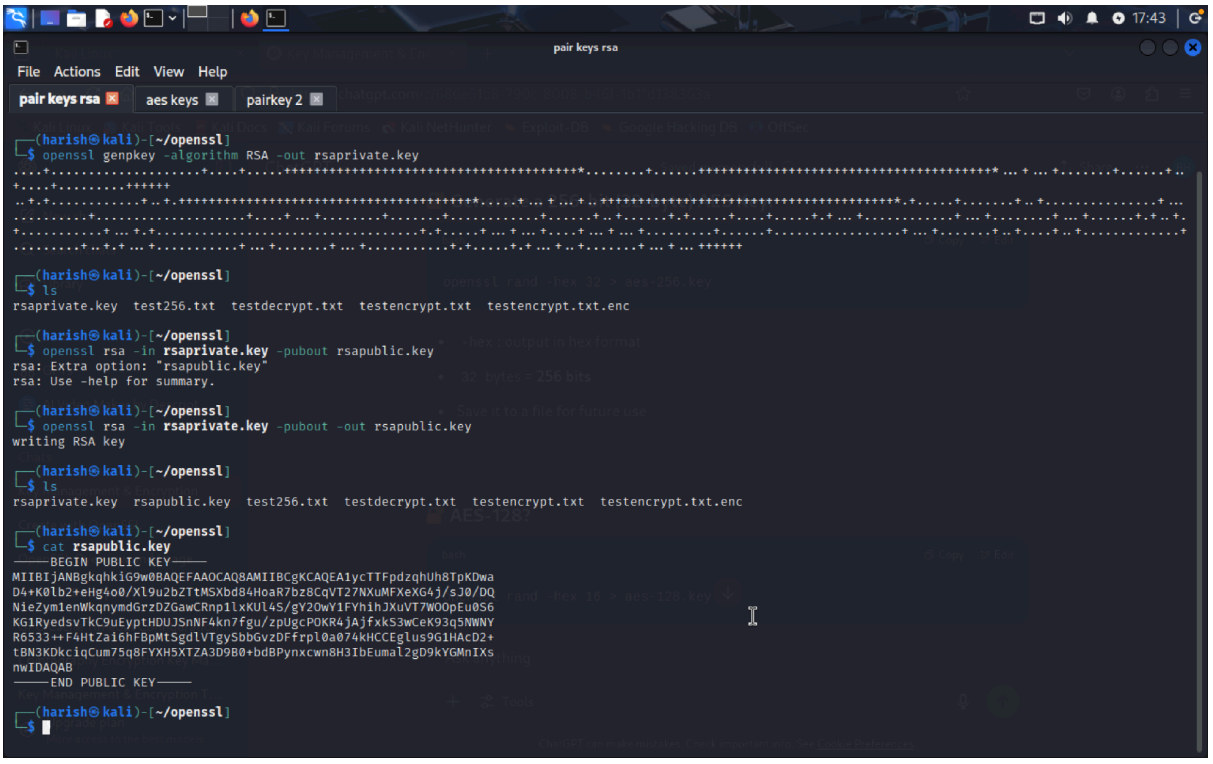


Fig: Asymmetric pair keys

Public key from a generated private key:

Command:

Openssl rsa -in rsaprivate.key -pubout -out rsapublic.key (gives you public key)

Symmetric key:

Command: Openssl rand -hex bytes >keyname.key

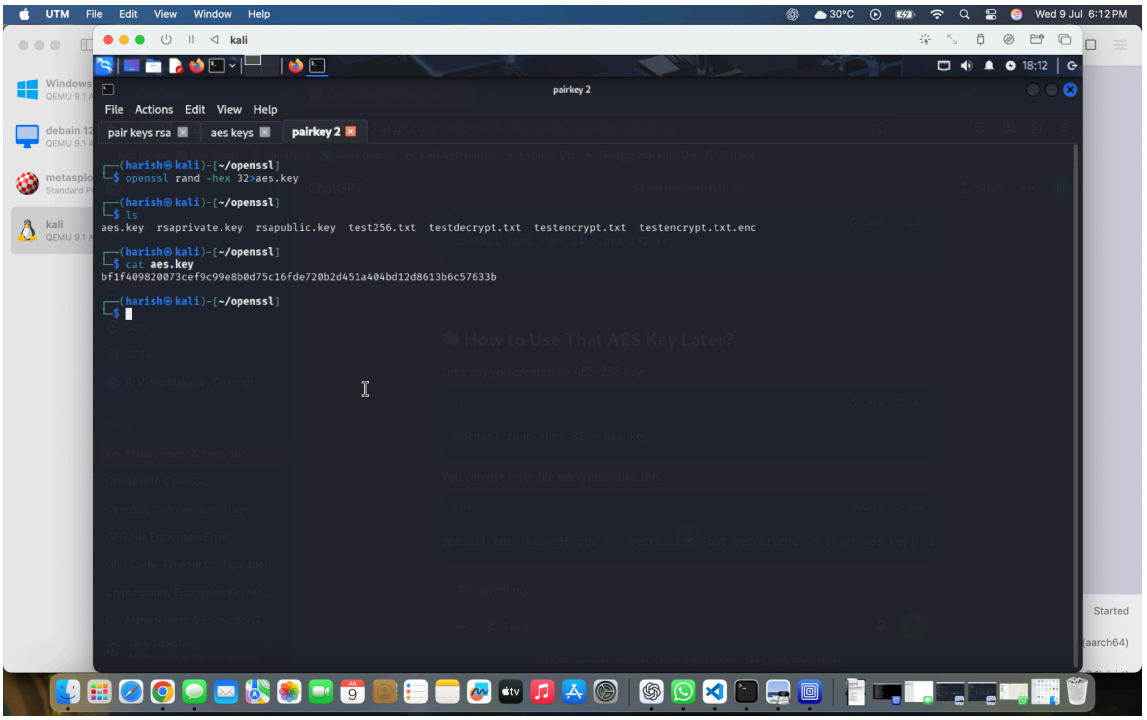


Fig: Symmetric key(AES-256=32bytes,128=16bytes,192=24bytes)

Hashing: it is one way function fixed string length(irreversible)

Command: openssl dgst <hash-format> file

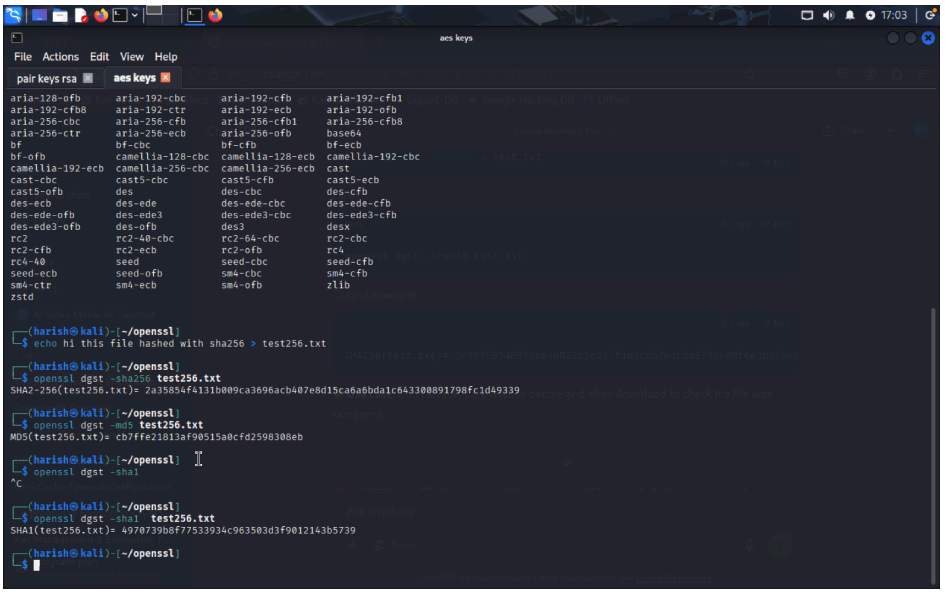


Fig: converted into hash using (sha256,…)

Encryption : Convert into cipher text (AES)=> Block cipher most widely used

Command :

Encryption: openssl enc -aes-128-CBC -salt -in testfile.txt -out textfile.txt.enc

Decryption: openssl enc -d -aes-128-CBC -salt -in testfile.txt.enc -out plain.txt

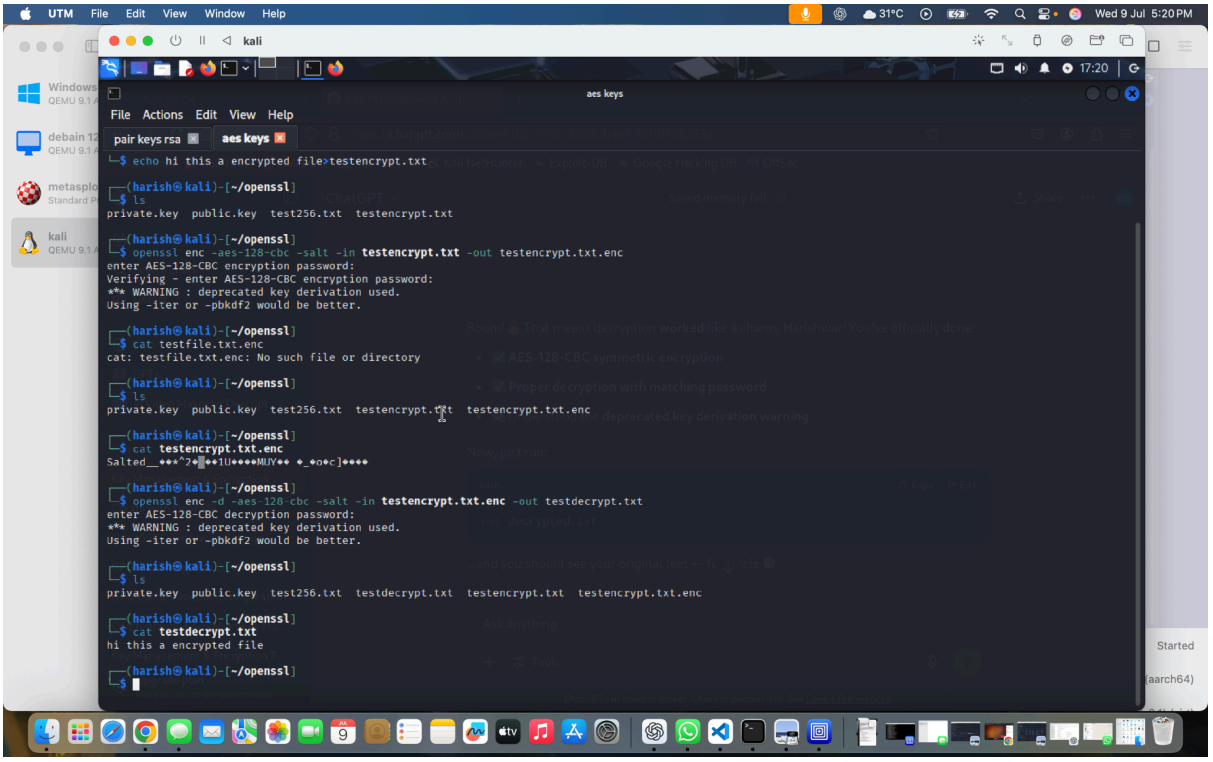


Fig: Encryption && Decryption (Openssl)

Certificate creation:

Digital Certificate

A digital certificate is like an online ID card issued by a trusted Certificate Authority (CA). It confirms the identity of a website or user and contains the public key, the owner's details, and the CA's digital signature

Used in: SSL/TLS for HTTPS, secure email, code signing.

CSR (Certificate Signing Request)

A **CSR** is a file you generate when you want to get a certificate. It includes your **public key** and identifying info (like domain name, organization, etc.), but **not** the private key.

You send the CSR to a CA, and they use it to create your digital certificate.

STEP 1**: Generate a Private Key && Generate a Certificate Signing Request (CSR):**

Command key generation :openssl genpkey -algorithm RSA -out caprivate.key -pkeyopt rsa_keygen_bits:2048

Command CSR: openssl req -new -key ca-private.key -out ca-private.csr

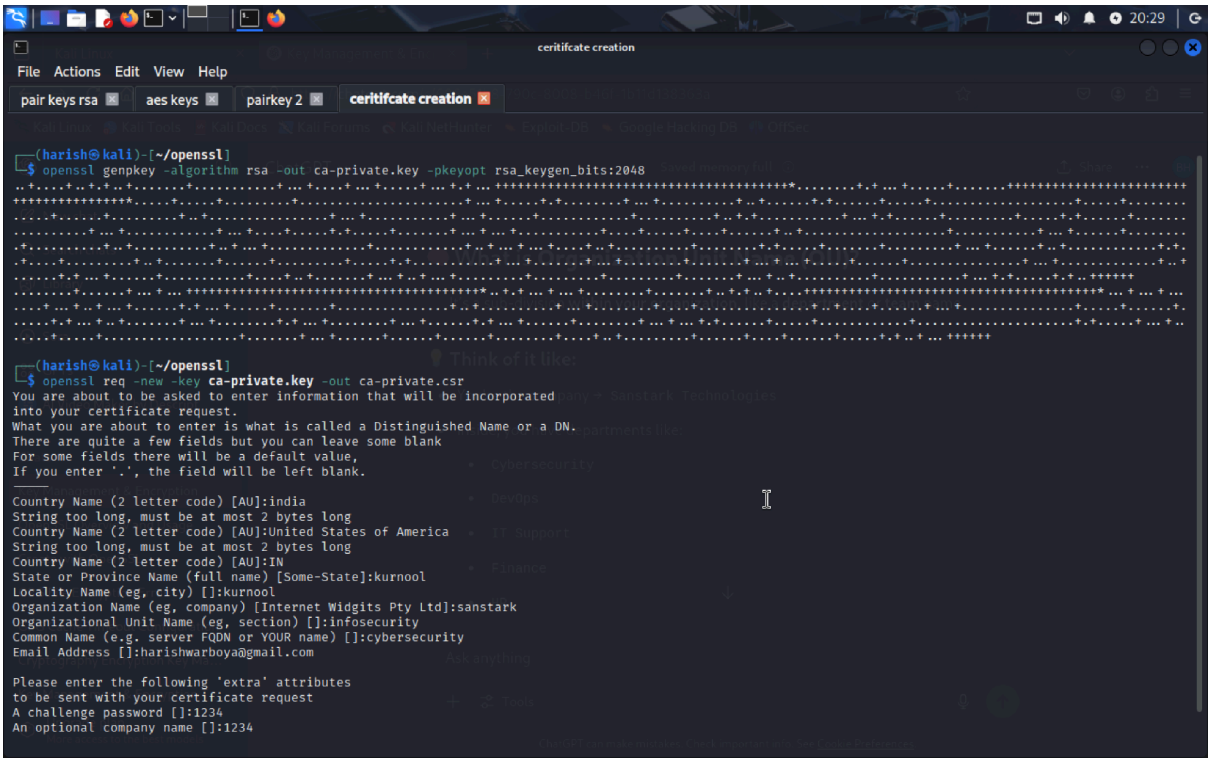


Fig: key and csr

STEP 2: Create a Self-Signed Certificate:

Command: openssl x509 -req -in harish.csr -signkey harish.key -out harish.crt -days 10

`harish.crt` – your public certificate
`ca-private.key` – your private key

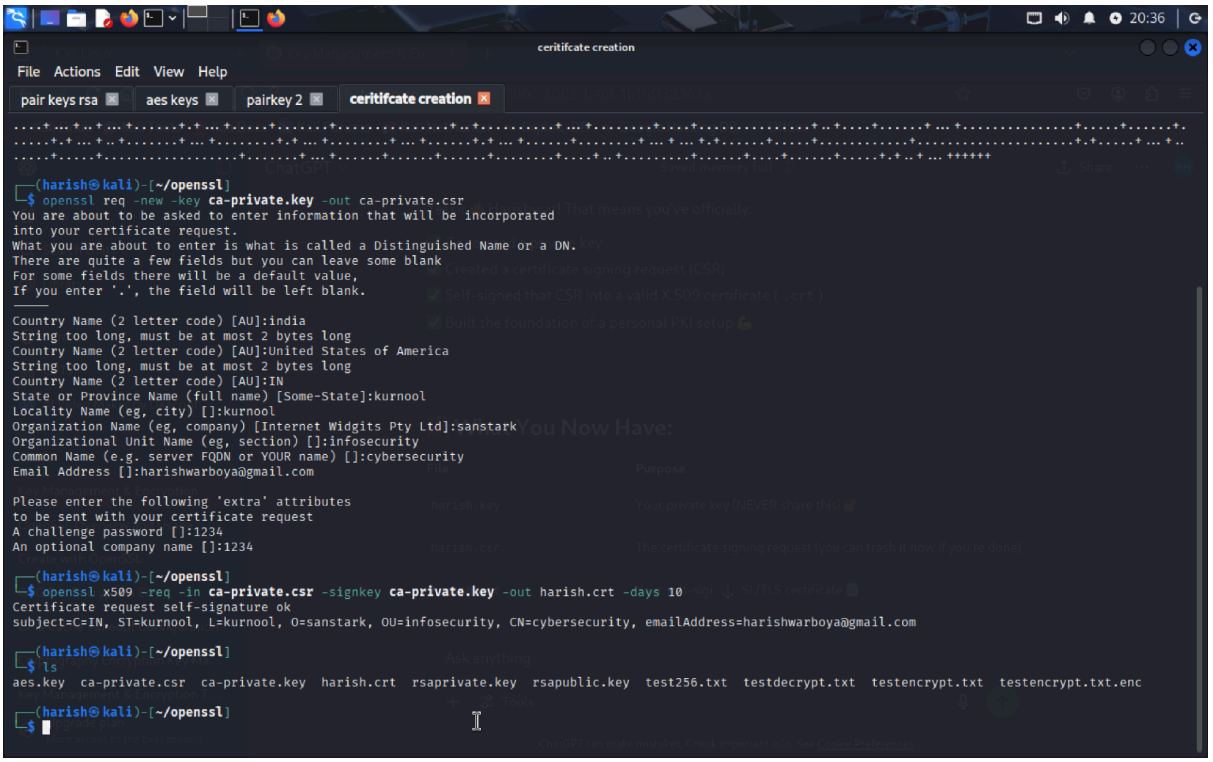


Fig: Digital Ceritificate Creation

Conclusion: Cryptography, Key Management & Encryption Tools

Cryptography is the core of cybersecurity, ensuring confidentiality, integrity, and authenticity of data through encryption, hashing, and digital signatures. Key management plays a critical role in securely generating, storing, and handling encryption keys. Without proper key management, even the strongest cryptographic algorithms become vulnerable. Tools like OpenSSL, GPG, and VeraCrypt enable secure implementation of encryption, signing, and certificate handling. Together, cryptography, key management, and encryption tools form the foundation of strong data protection and secure communication.

Reference:

https://youtu.be/PBR0Ujv1BIE?si=w4fmRsl-GlAG0p9b (GPG)

https://youtu.be/wzbf9ldvBjM?si=6SkZM7araZOE744P (open-ssl-all)