

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

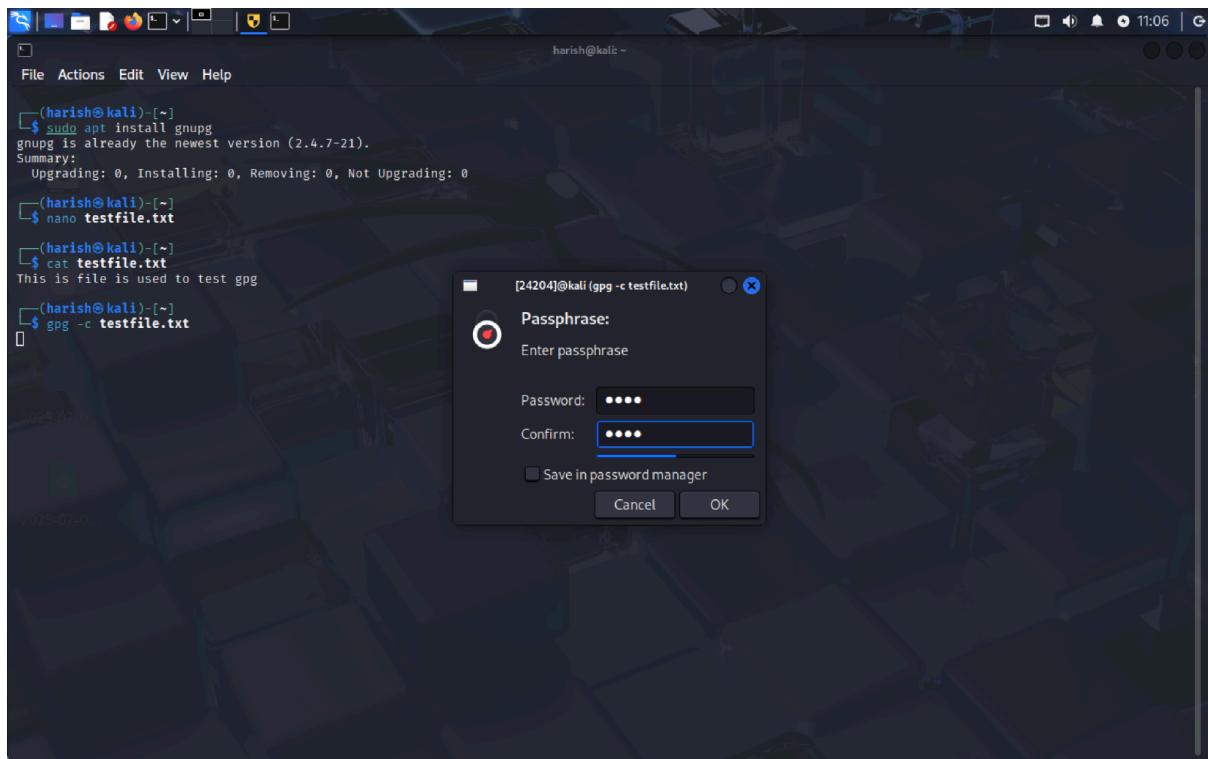
It's an **open-source** tool that implements the **OpenPGP** standard for:

- **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



```

File Actions Edit View Help
Fileish@kali:[~]
$ sudo apt install gnupg
gnupg is already the newest version (2.4.7-21).
Summary:
Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 0
Fileish@kali:[~]
$ nano testfile.txt
Fileish@kali:[~]
$ cat testfile.txt
this is the test file of gpg

Fileish@kali:[~]
$ gpg -c testfile.txt
Fileish@kali:[~]
$ ls
2025-07-07 webappclouds-ZAP-Report-.html' extentions Music Public spiderfoot Videos webappinfo.txt
Desktop hast.txt payload.exe secret.txt Templates web webappclouds_emails.xml
Documents holehe Photon secret.zip testfile.txt webappclouds_full.xml
doubleaccounts.txt metagoofil Pictures sherlock testfile.txt.gpg webappinfo.pdf
Downloads modulus.bin proxychains-ng singlefile.txt venvs
Fileish@kali:[~]
$ cat testfile.txt.gpg
Fileish@kali:[~]
$ ls
Fileish@kali:[~]

```

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

```

File Actions Edit View Help
Fileish@kali:[~]
$ gpg -d testfile.txt.gpg
gpg: AES256.CFB encrypted data
gpg: encrypted with 1 passphrase
this is the test file of gpg

Fileish@kali:[~]
$ # the file opened without asking th password because of cache memory holds the password until 600 sec
it wont ask for password
Fileish@kali:[~]
$ ls -a
.. doubleaccounts.txt .lesshst .profile .sudo_as_admin_successful .wget-hsts
.. Downloads .local proxychains-ng Templates testfile.txt .wpSCAN
.. extentions .malfrats .maltego .pyenv testfile.txt.gpg .xauthority
.. .face .metagoofil .modulus.bin .python_history .theHarvester .xsession-errors
.. .gnupg .google-cookie .msf4 secret.txt venvs .xsession-errors.old
.. .gvfs hast.txt .Music sherlock .Videos .ZAP
.. holehe payload.exe singlefile.txt secret.zip .viminfo .zprofile
.. .ICEauthority .Photon .Pictures spiderfoot .web .zsh_history
.. .Java .john .pki .spiderfoot_history webappclouds_emails.xml .zshrc
.. .dmrc
.. Documents
Fileish@kali:[~]
$ cd .gnupg
Fileish@kali:[~/.gnupg]
$ ls
private-keys-v1.d pubring.kbx random_seed
Fileish@kali:[~/.gnupg]
$ ls

```

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

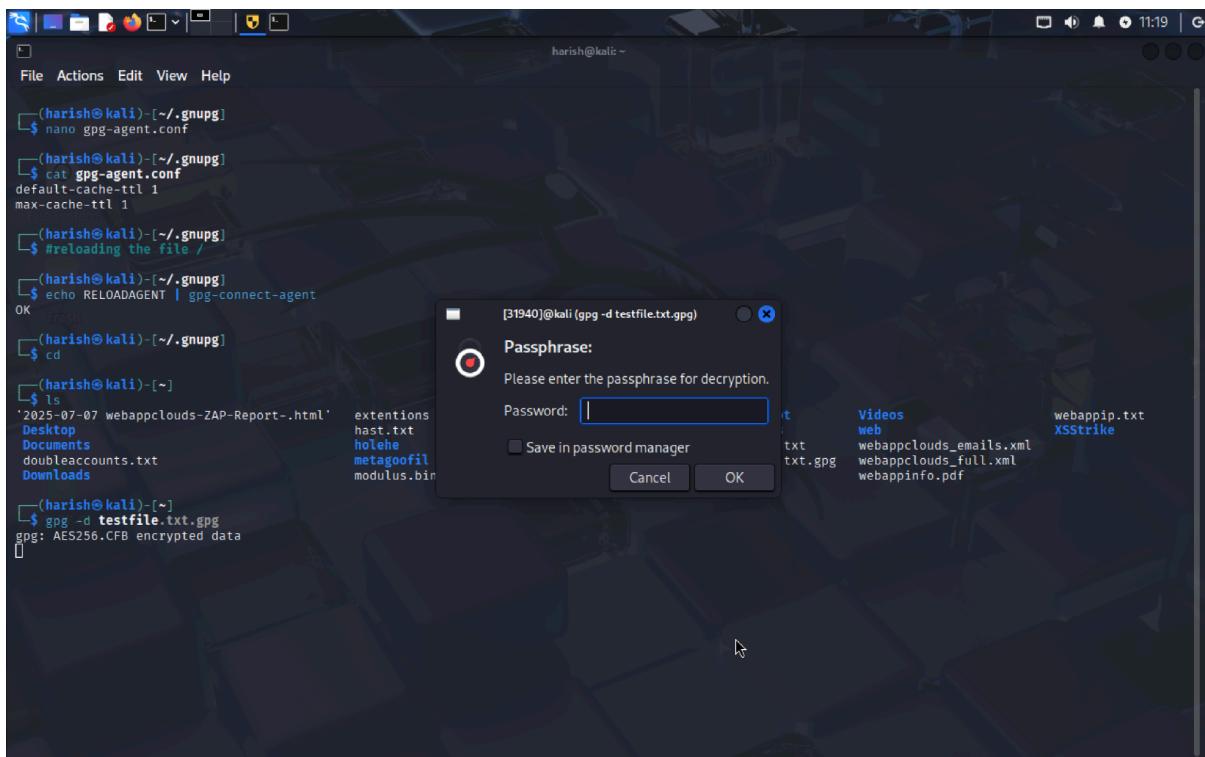


Fig: Cache config && Reload the file

GPG Encryption for Folders

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

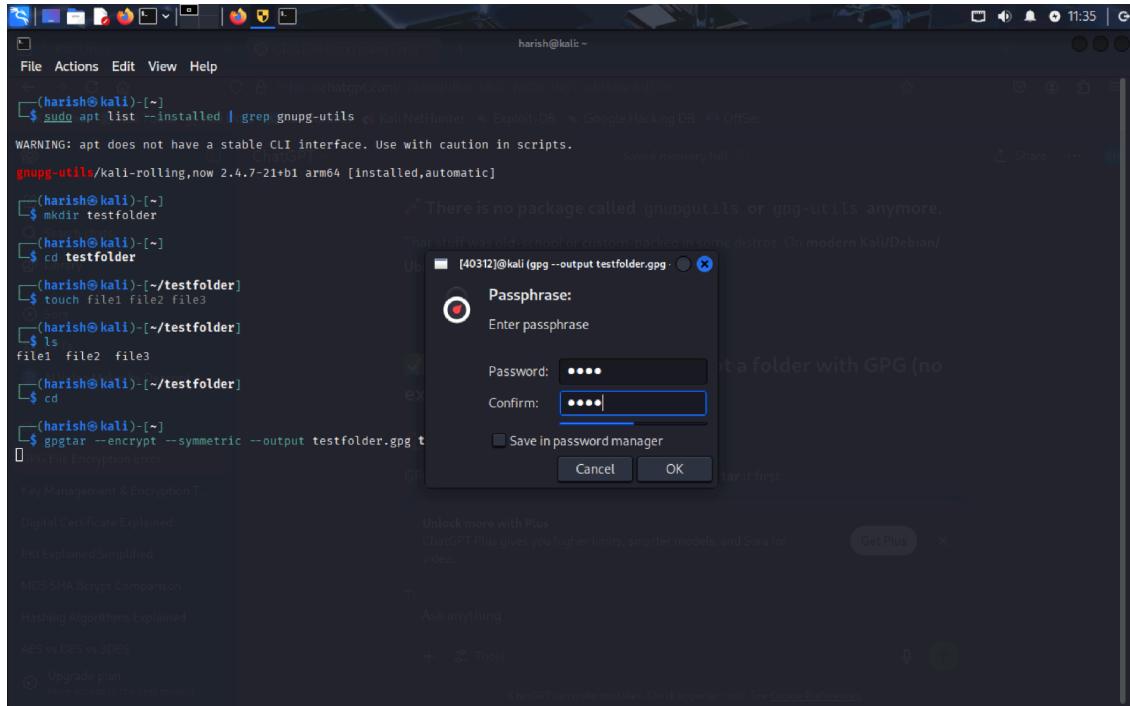


Fig: symmetric encryption to folder

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

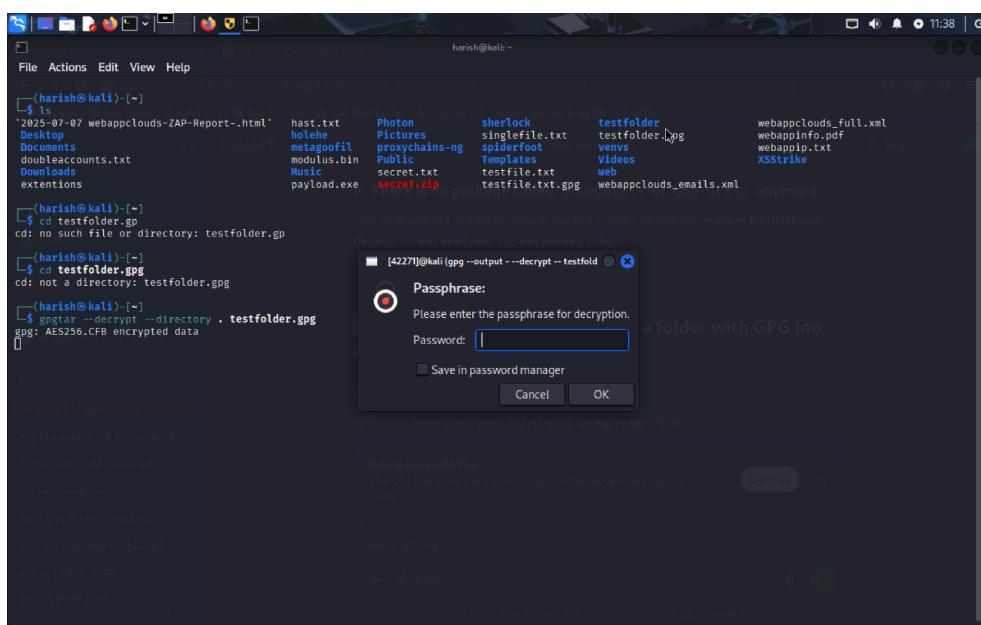


Fig: Decrypting the folder

The screenshot shows a terminal window on a Kali Linux system. The user, harish, is navigating through their home directory (~) and a testfolder. They attempt to change into the testfolder directory but receive an error message stating 'cd: no such file or directory: testfolder'. They then use the gpgtar command to decrypt a file named testfolder.gpg, which contains AES256-CFB encrypted data. The terminal also shows the creation of files file1, file2, and file3. A ChatGPT interface is visible in the background, providing a solution to extract the .tar file.

```
(harish@kali)-[~]
$ ls
2025-07-07 webappclouds-ZAP-Report-.html' extentions Music Public spiderfoot Videos webappinfo.pdf
Desktop hast.txt payload.exe secret.txt Templates web webappclouds_emails.xml
Documents holehe Photon secret.zip testfile.txt.gpg webappclouds_full.xml
doubleaccounts.txt metagoofil Pictures sherlock testfolder.gpg
Downloads modulus.bin proxychains-ng singlefile.txt venvs webappinfo.pdf

(harish@kali)-[~]
$ cd testfolder
cd: no such file or directory: testfolder

(harish@kali)-[~]
$ gpgtar --decrypt --directory . testfolder.gpg
gpg: AES256-CFB encrypted data
gpg: encrypted with 1 passphrase

(harish@kali)-[~]
$ cd testfolder
(harish@kali)-[~/testfolder]
$ ls
file1 file2 file3

(harish@kali)-[~/testfolder]
$
```

✓ SOLUTION 2: Want the .tar file back first?
If you only want the .tar and then extract it manually:

```
both
gpgtar --decrypt --output testfolder.tar testfolder.gpg
tar -xvf testfolder.tar
```

Ask anything

Get Plus

Fig: Reading the folder

GPG Asymmetric Encryption Practical

1. Generate GPG Key Pair:

Command: gpg --full-generate-key

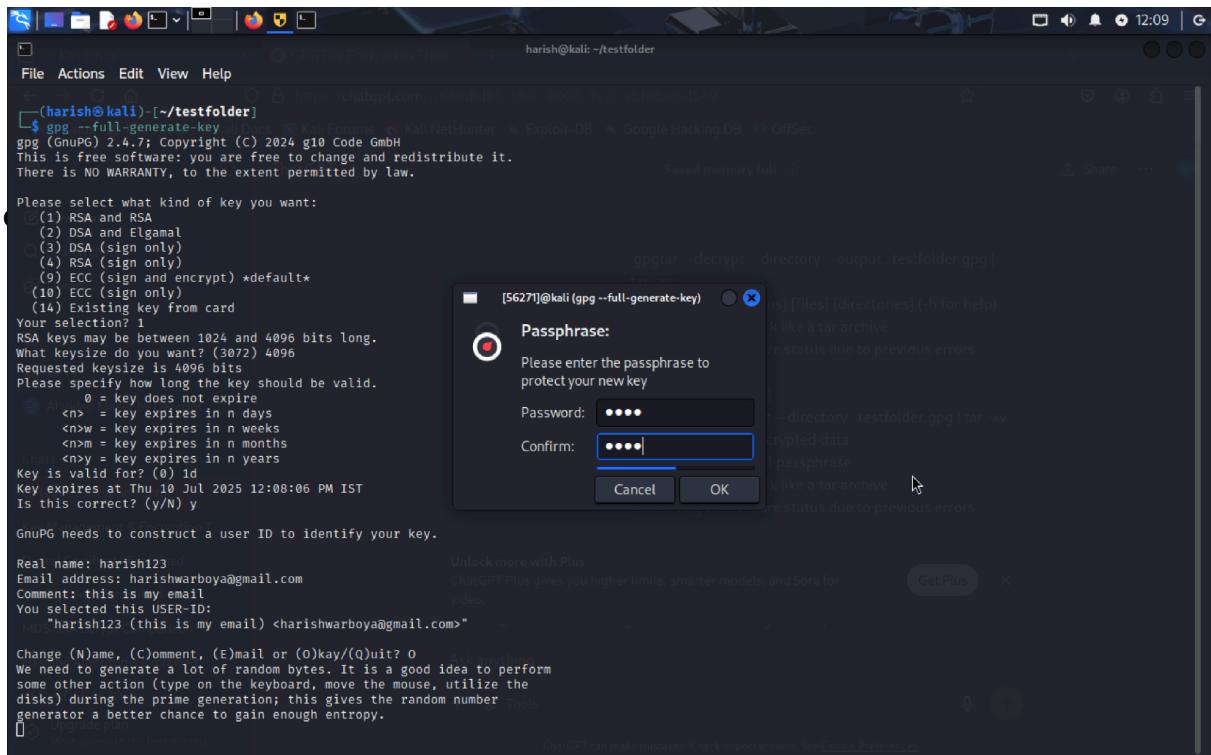


Fig: Keys Generating

Chose: (1) RSA and RSA

Key size: 4096

Name: harish123

Email: harishwaroya@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**

The screenshot shows a terminal window titled "GPG Suite Encryption Software". The command entered is:

```
cat harish_pub.asc
```

The output of the command is a long string of ASCII text representing a PGP public key. The text starts with:

```
BEGIN PGP PUBLIC KEY BLOCK
```

... (The rest of the key is too long to show here)

At the bottom of the terminal window, there is a message: "Check if it exists".

Fig:Public Key

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

The screenshot shows a terminal window titled "sender" on a Kali Linux desktop. The terminal content is as follows:

```
(harish㉿kali)-[~] $ mkdir gpg-sender
(harish㉿kali)-[~] $ cd gpg-sender
(harish㉿kali)-[~/gpg-sender] $ gpg --import ../../testfolder/harish_pub.asc
gpg: key 003C02062ED03CEC: "harish123 (this is my email) <harishwarboya@gmail.com>" not changed
gpg: Total number processed: 1
gpg:          unchanged: 1

(harish㉿kali)-[~/gpg-sender] $ echo hi iam the sender from this side > hello.txt
(harish㉿kali)-[~/gpg-sender] $ gpg --encrypt --recipient harishwarboya@gmail.com hello.txt
gpg: invalid option "--recipient"
(harish㉿kali)-[~/gpg-sender] $ gpg --encrypt --recipient harishwarboya@gmail.com hello.txt
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid: 1  signed: 0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2025-07-10
gpg: --decrypt message.txt.gpg
(harish㉿kali)-[~/gpg-sender] $ cp hello.txt.gpg ../../testfolder/
(harish㉿kali)-[~/gpg-sender] $ ls
named simplified
```

Below the terminal, there is a note: "Then copy it to your receiver folder (Terminal 1)". The ChatGPT interface includes a sidebar with links like "Digital Certificate Explained", "AES vs DES vs 3DES", and "Hashing Algorithms Explained". A watermark for "Ask anything" is visible.

Fig: Sender View

The screenshot shows a terminal window titled "receiver" on the same Kali Linux desktop. The terminal content is as follows:

```
(harish㉿kali)-[~] $ testfolder
(harish㉿kali)-[~/testfolder] $ ls
file1 file2 file3 harish_pub.asc hello.txt.gpg
(harish㉿kali)-[~/testfolder] $ gpg --decrypt hello.txt.gpg
gpg: decrypted with rsa4096 key, ID 62E64649E0BF
"harish123 (this is my email) <harishwarboya@gmail.com>"
```

A modal dialog box titled "Passphrase:" is displayed, asking for a password to unlock the OpenPGP secret key. The dialog includes fields for "Password:" and "Save in password manager", and buttons for "Cancel" and "OK".

Below the terminal, there is a note: "Then copy it to your receiver folder (Terminal 1)". The ChatGPT interface includes a sidebar with links like "Digital Certificate Explained", "AES vs DES vs 3DES", and "Hashing Algorithms Explained". A watermark for "Ask anything" is visible.

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 -algorithms <algorithms> -out ,,, (which is modern can generate different keys)

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

The screenshot shows a terminal window titled "pair keys rsa" running on a Kali Linux desktop environment. The terminal displays the following command sequence:

```
(harish㉿kali)-[~/openssl]$ openssl genpkey -algorithm RSA -out rsaprivate.key
(harish㉿kali)-[~/openssl]$ ls
rsaprivate.key test256.txt testdecrypt.txt testencrypt.txt testencrypt.txt.enc
(harish㉿kali)-[~/openssl]$ openssl rsa -in rsaprivate.key -pubout rsapublic.key
(harish㉿kali)-[~/openssl]$ ls
rsaprivate.key rsapublic.key test256.txt testdecrypt.txt testencrypt.txt testencrypt.txt.enc
(harish㉿kali)-[~/openssl]$ cat rsapublic.key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9wBAQEFAAOCAQ8AMIIIBCgKCAQEAIyCTFPdzqjhUi8TpKDwa
D4+K0lb2+eHg4oo/Xl9u2bzTTMSxbd84Hoar7bz8CqvT27XuMFxeG4J/sJ0/DQ
NizymlenWkanymdgrzD2GawCrnpl1xU45/gx20w1FyhinhJKuVT7W0Opfui0S6
KGIRyedsvtKCeypthDUJSnNF4kn7fgu/zpUgcPOKR4JAfxkS3nCeK93q5NWNY
R6533+H4HtzA16hfBpMtSgd1VTgySbbGvzdFFrpl0a074KHCEgIus9G1HACD2+
tBN3DKCcicUm75q8FYXH5XTZA3D9B0+bdBPynxcwn8H3IbEumal2gD9KYGMnIXs
nwIDAQAB
-----END PUBLIC KEY-----
(harish㉿kali)-[~/openssl]$
```

The terminal shows the creation of two files: rsaprivate.key and rsapublic.key. The rsapublic.key file contains the public RSA key in PEM format, starting with "-----BEGIN PUBLIC KEY-----".

Fig: Asymmetric pair keys

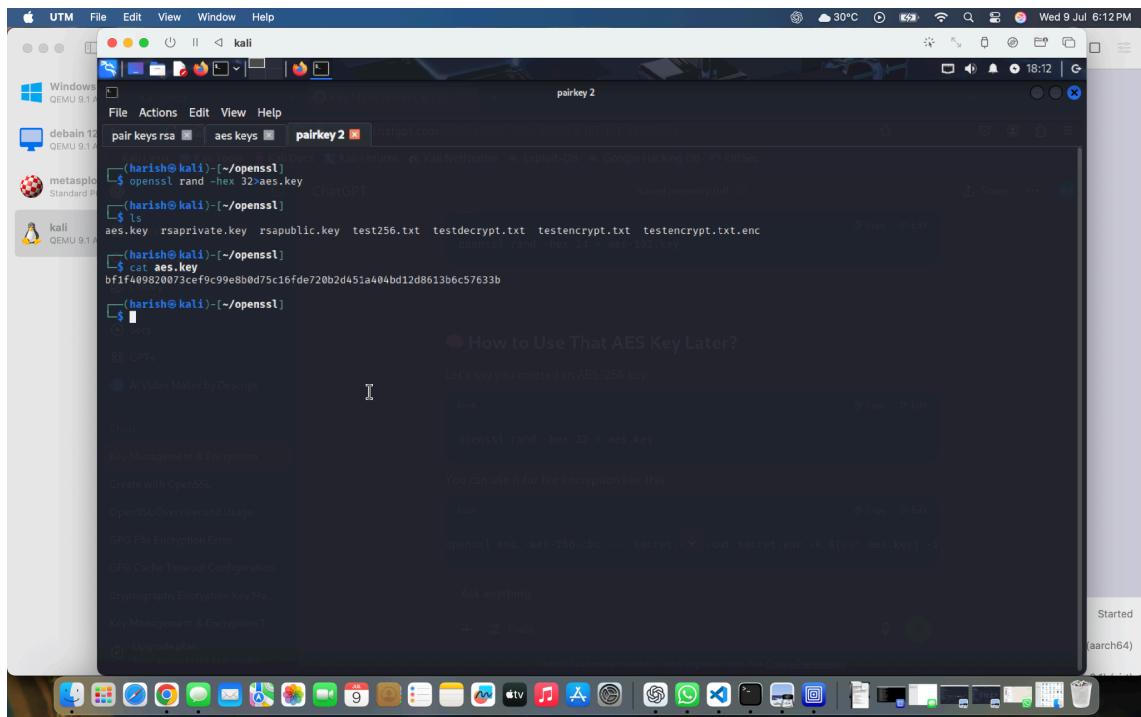
Public key from a generated private key:

Command:

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

Symmetric key:

Command: openssl rand -hex bytes >keyname.key

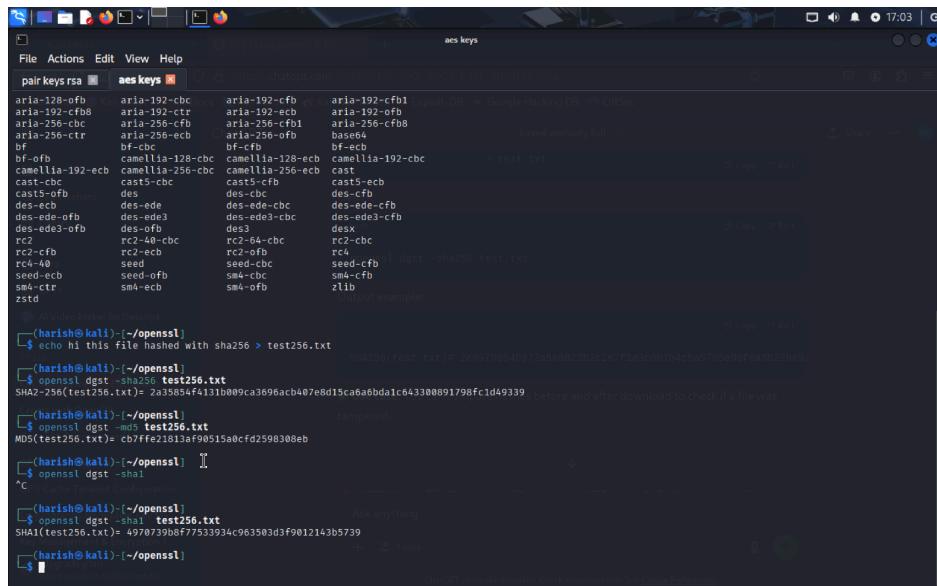


```
(harish㉿kali)-[~/openssl]$ openssl rand -hex 32 > aes.key
(harish㉿kali)-[~/openssl]$ ls
aes.key  rsaprivate.key  rsapublic.key  test256.txt  testdecrypt.txt  testencrypt.txt.enc
(harish㉿kali)-[~/openssl]$ cat aes.key
bf1f409820073cef9c99e800d75c16fde720b2d451a404bd12d8613b6c57633b
(harish㉿kali)-[~/openssl]$
```

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



```
(harish㉿kali)-[~/openssl]$ echo hi this file hashed with sha256 > test256.txt
(harish㉿kali)-[~/openssl]$ $ echo hi this file hashed with sha256 > test256.txt
SHA256(test256.txt)= 2a3585474131b009ca3698acb407e8d15ca6bda1c643308891798fc1d49339
(harish㉿kali)-[~/openssl]$ openssl dgst -md5 test256.txt
MD5(test256.txt)= cb7ffe22813a9b0515a0cf2d2598308eb
(harish㉿kali)-[~/openssl]$ openssl dgst -sha1
SHA1(test256.txt)= 4970739b8f77533934c963503d3f9012143b5739
(harish㉿kali)-[~/openssl]$
```

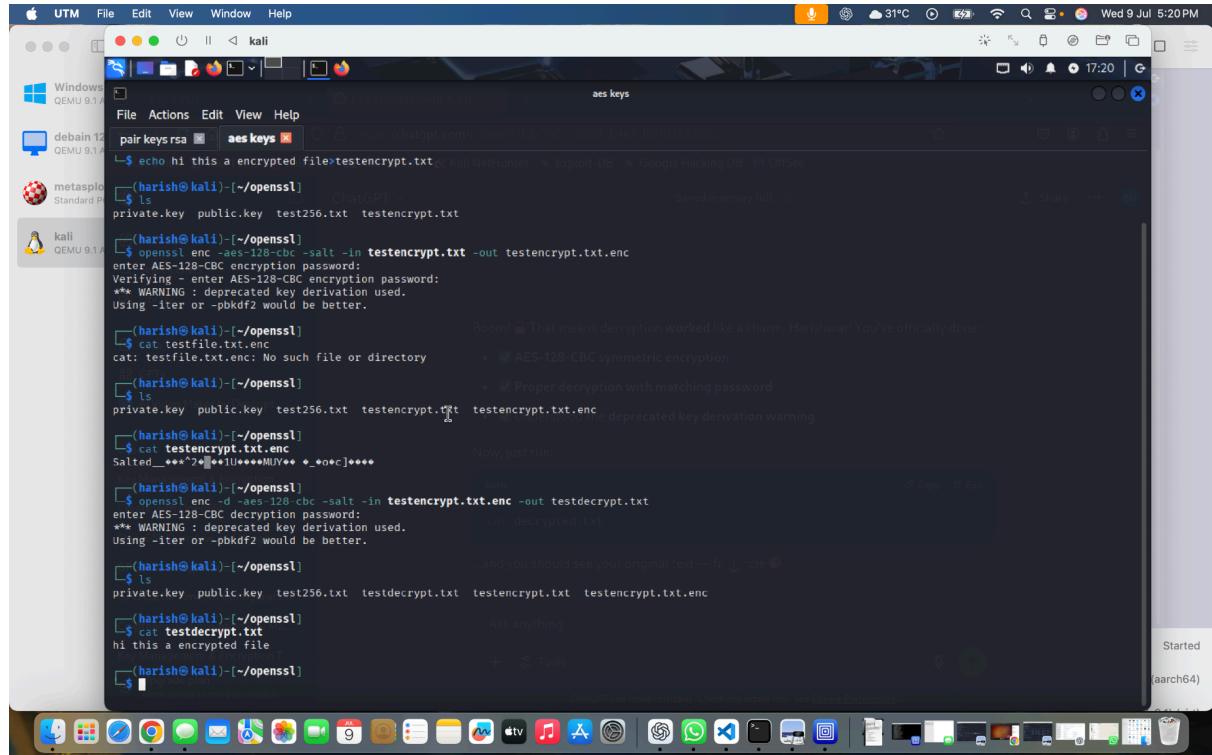
Fig: converted into hash using (sha256,...)

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

Command :

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

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



The screenshot shows a macOS desktop environment with a terminal window open. The terminal window title is "aes keys". The terminal session shows the following commands and their output:

```
pair keys rsa
(harish@kali)-[~/openssl]
$ ls
private.key public.key test256.txt testencrypt.txt

(harish@kali)-[~/openssl]
$ openssl enc -aes-128-cbc -salt -in testencrypt.txt -out testencrypt.txt.enc
enter AES-128-CBC encryption password:
Verifying - entered AES-128-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

(harish@kali)-[~/openssl]
$ cat testfile.txt.enc
cat: testfile.txt.enc: No such file or directory
(harish@kali)-[~/openssl]
$ ls
private.key public.key test256.txt testencrypt.txt testencrypt.txt.enc

(harish@kali)-[~/openssl]
$ cat testencrypt.txt.enc
Salted__***^24@+1@***MUY** *_o*c]****

(harish@kali)-[~/openssl]
$ openssl enc -d -aes-128-cbc -salt -in testencrypt.txt.enc -out testdecrypt.txt
enter AES-128-CBC decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

(harish@kali)-[~/openssl]
$ ls
private.key public.key test256.txt testdecrypt.txt testencrypt.txt testencrypt.txt.enc

(harish@kali)-[~/openssl]
$ cat testdecrypt.txt
hi this a encrypted file

(harish@kali)-[~/openssl]
$
```

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 genkey -algorithm RSA -out caprivate.key -pkeyopt rsa_keygen_bits:2048`

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

The screenshot shows a terminal window titled "certificate creation" on a Kali Linux desktop. The terminal displays two commands being run:

```
(harish㉿kali)-[~/openssl]
$ openssl genkey -algorithm rsa -out ca-private.key -pkeyopt rsa_keygen_bits:2048
Saving memory full ...
```

```
(harish㉿kali)-[~/openssl]
$ openssl req -new -key ca-private.key -out ca-private.csr
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
Country Name (2 letter code) [AU]:india
String too long, must be at most 2 bytes long
Country Name (2 letter code) [AU]:United States of America
String too long, must be at most 2 bytes long
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:kurnool
Locality Name (eg, city) []:kurnool
Organization Name (eg, company) [Internet Widgit Pty Ltd]:sanstark
Organizational Unit Name (eg, section) []:Infosecurity
Common Name (e.g. server FQDN or YOUR name) []:cybersecurity
Email Address []:harishwarboya@gmail.com
```

The terminal also shows a list of available options for the certificate request, including:

- cybersecurity
- DevOps
- IT Support
- Finance

At the bottom of the terminal, there are prompts for extra attributes, challenge password, and optional company name, all set to "1234".

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

```

certificate creation
File Actions Edit View Help
pair keys rsa aes keys pairkey 2 certificate creation
$ openssl req -new -key ca-private.key -out ca-private.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value.
If you enter '.', the field will be left blank.
Country Name (2 letter code) [AU]:india
String too long, must be at most 2 bytes long
Country Name (2 letter code) [AU]:United States of America
String too long, must be at most 2 bytes long
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:kurnool
Locality Name (eg, city) []:kurnool
Organization Name (eg, company) [Internet Widgits Pty Ltd]:sanstark
Organizational Unit Name (eg, section) []:infosecurity
Common Name (e.g. server FQDN or YOUR name) []:cybersecurity
Email Address []:harishwarboya@gmail.com
File Purpose
please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:1234
An optional company name []:1234
$ harish.key Your private key(NEVER share this)
$ harish.csr The certificate signing request (you can trash it now if you're done)
$ openssl x509 -req -in ca-private.csr -signkey ca-private.key -out harish.crt -days 10
Certificate request self-signature ok
subject=C=IN, ST=kurnool, L=kurnool, O=sanstark, OU=infosecurity, CN=cybersecurity, emailAddress=harishwarboya@gmail.com
$ ls
aes.key ca-private.csr ca-private.key harish.crt rsaprivate.key rsapublic.key test256.txt testdecrypt.txt testencrypt.txt testencrypt.txt.enc
$ harish@kali:[~/openssl]
$ 

```

Fig: Digital Certificate 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-GIAG0p9b> (GPG)

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