THIRD PHASE

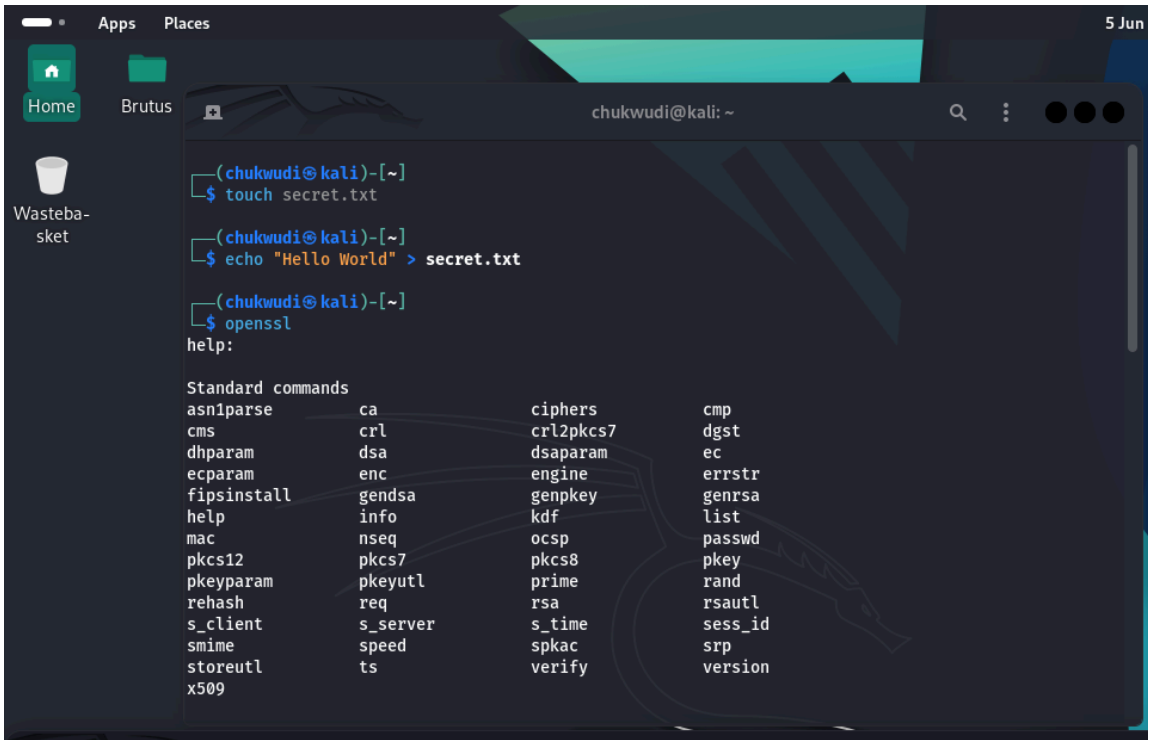
CYBERSECURITY CAPSTONE PROJECT GROUP 24

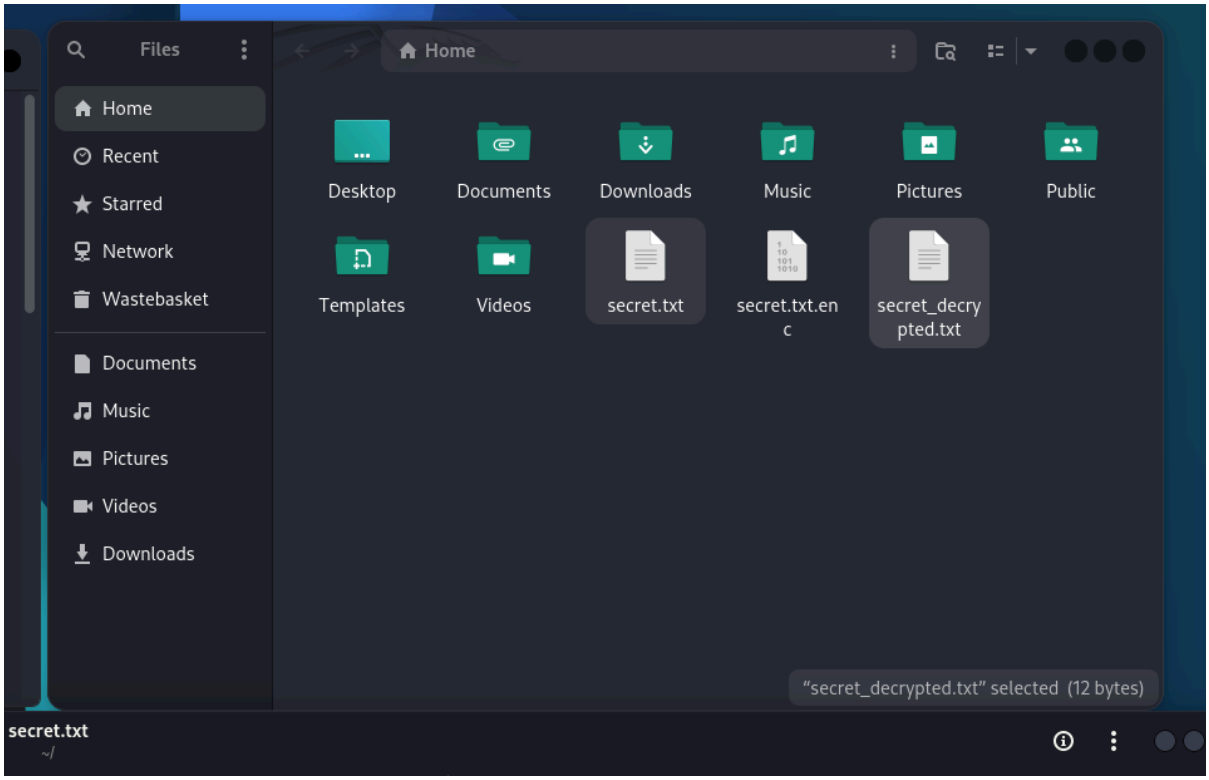
(Applying Cryptography in Real-World Scenarios)

1. **DATA PROTECTION WITH SYMMETRIC ENCRYPTION (AES) USING OPENSSL TOOL.**

SYMMETRIC ENCRYPTION is the method of encryption where the same key is used for both encryption and decryption of data. **AES** or **Advanced Encryption Standard**, is a widely trusted encryption algorithm used to secure data by converting it into an unreadable format without the proper key.

Firstly, a text file named secret.txt was created using the command Touch secret.txt, and a message was created inside the file using the echo “Hello World” > secret.txt command as shown in the image.





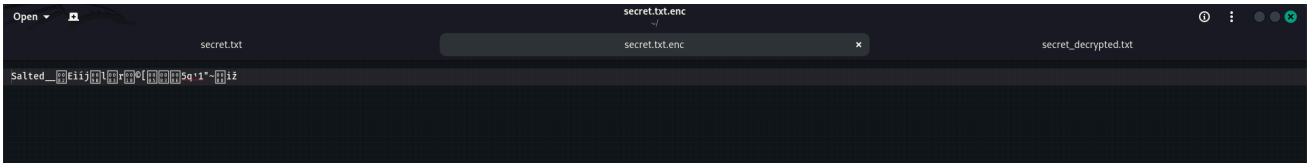
STEP 1: ENCRYPTING THE FILE USING AES-256

Entering the command below prompted me to enter an AES-256-CBC encryption password, which encrypted the secret.txt file and made it unreadable without the key, as shown in the image below.

openssl enc -aes-256-cbc -salt -in secret.txt -out secret.txt.enc

Explanation:

- enc: encryption mode
- -aes-256-cbc: AES algorithm with 256-bit key in the CBC mode
- -salt: adds randomness to the encryption
- -in: input file
- -out: output encrypted file
  I was prompted to enter a password which was used as the key.



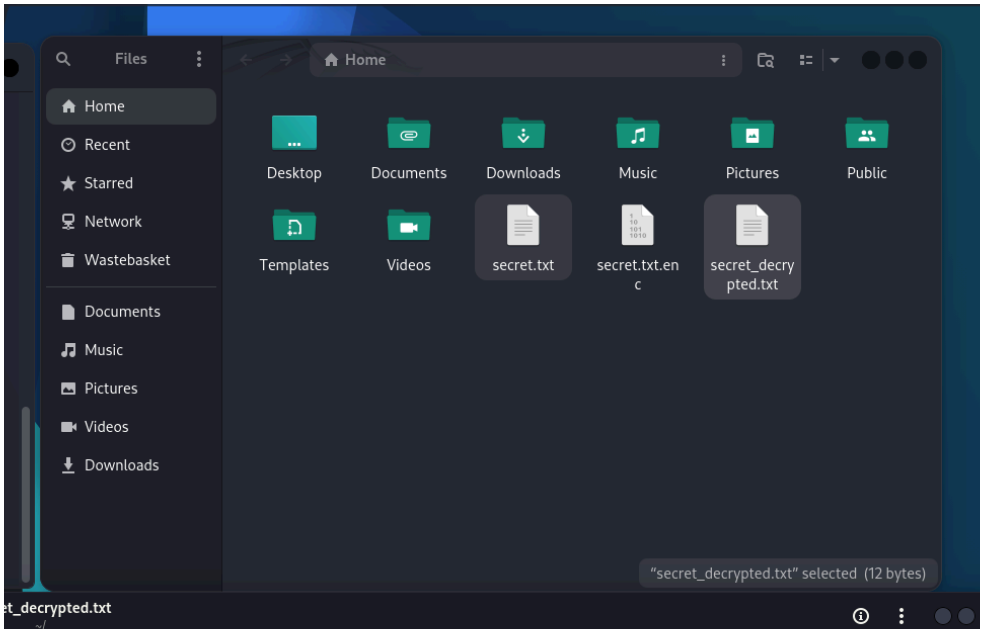
STEP 2: DECRYPTING THE FILE

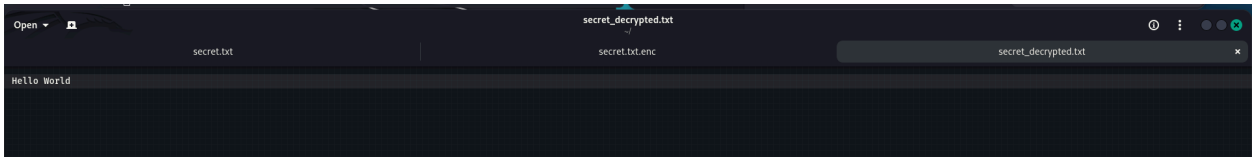
To decrypt the file, I entered the command below and the file became readable after entering the right password which I used in encrypting the file, as seen in the image below.

openssl enc -d -aes-256-cbc -in secret.txt.enc -out secret_decrypted.txt

Explanation:

- -d: decryption mode
- The same password which I used during encryption as the key is what I used to decrypt the file.





Now in total we have three files:

- Secret.txt (original)

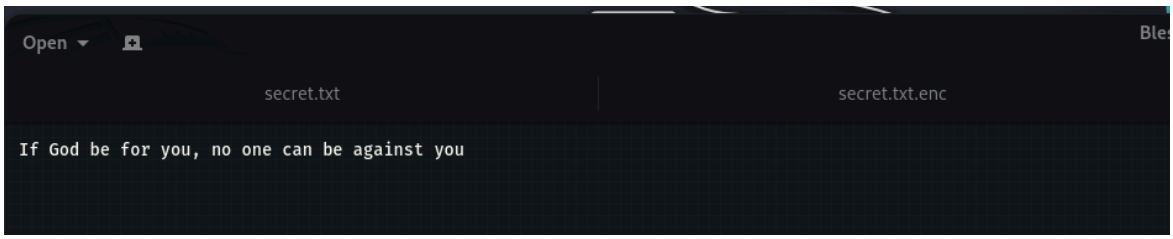
- Secret.txt.enc (encrypted, unreadable without password)
- Secret_decrypted.text (This should match the original)

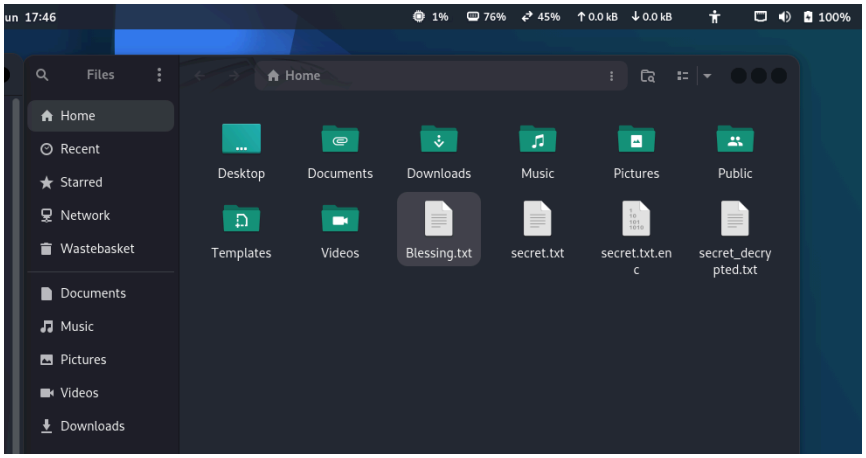
2. HASHING AND INTEGRITY CHECKING

- **GENERATING A HASH (SHA-256) FOR A FILE.**

SHA-256, known as Secure Hash Algorithm 256-bit, is used for cryptographic security, which produces irreversible and unique hashes.
Before generating the hash for a file, a file named **Blessing.txt** was created with content in it, which reads **"If God be for you, no one can be against you."** Then I proceeded to hash the file with the command below.

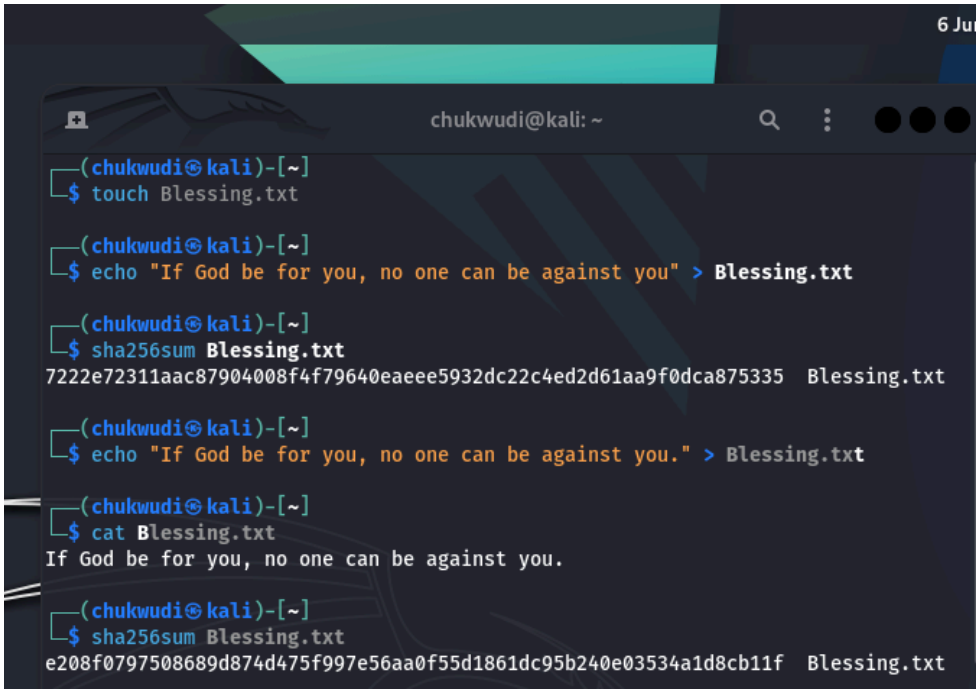
```
─(chukwudi㉿kali)-[~]
└─$ touch Blessing.txt

┌──(chukwudi㉿kali)-[~]
└─$ echo "If God be for you, no one can be against you" > Blessing.txt

┌──(chukwudi㉿kali)-[~]
└─$ sha256sum Blessing.txt
7222e72311aac87904008f4f79640eaeee5932dc22c4ed2d61aa9f0dca875335  Blessing.txt
```

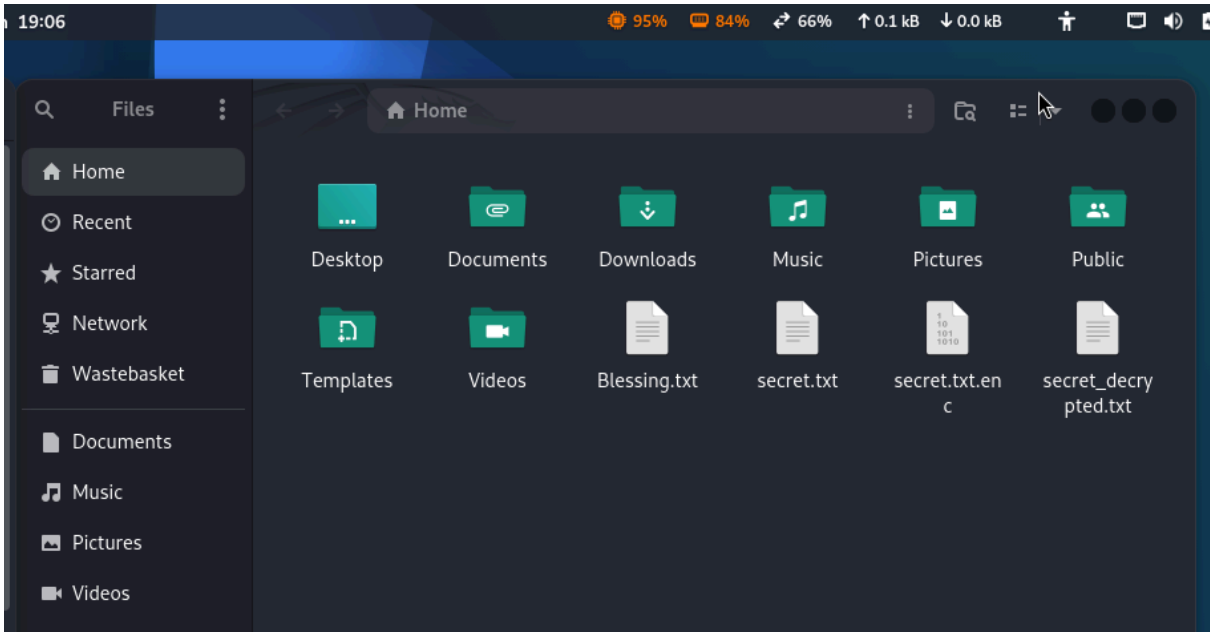




- **MODIFY THE FILE SLIGHTLY AND SHOW HOW THE HASH CHANGES**

Modifying the file Blessing.txt will indicate the file has been tampered with, which will automatically change the hash. So a full stop was added to the original text “If God be for you, no one can be against you” and a different hash was generated as shown in the image below.



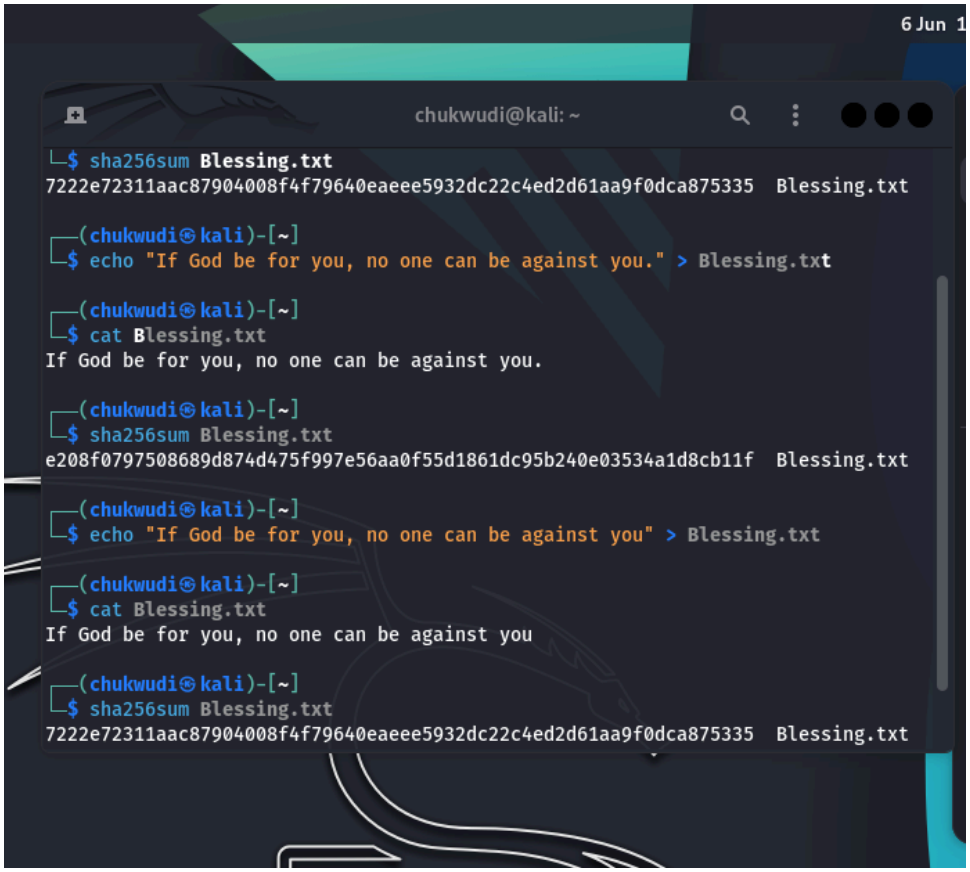


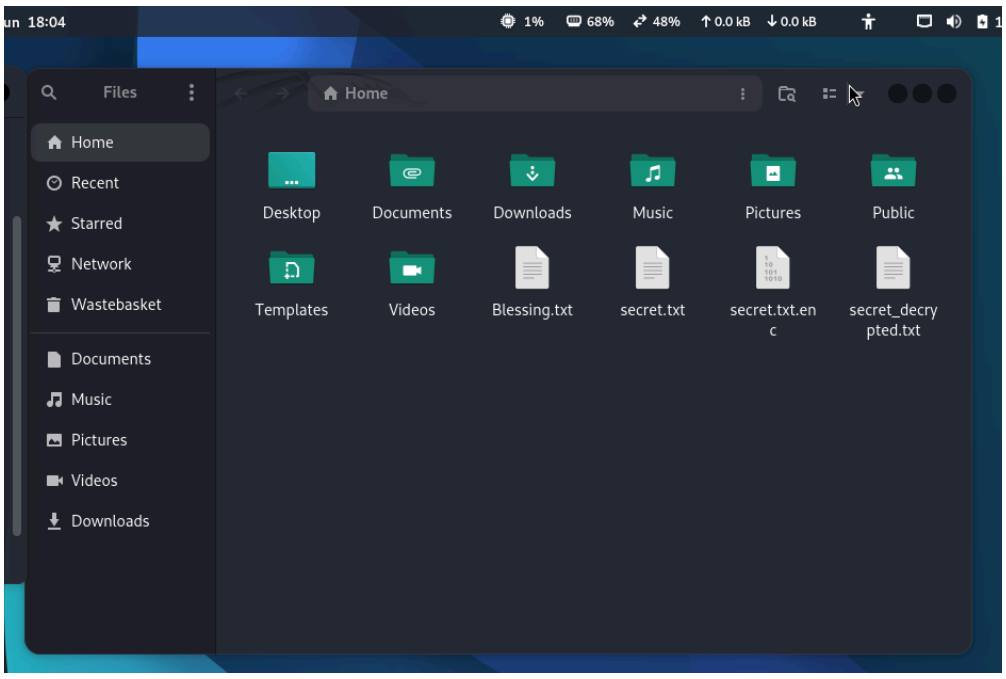

```
(chukwudi㉿kali)-[~]
└─$ echo "If God be for you, no one can be against you." > Blessing.txt

(chukwudi㉿kali)-[~]
└─$ sha256sum Blessing.txt
e208f0797508689d874d475f997e56aa0f55d1861dc95b240e03534a1d8cb11f  Blessing.txt
```

So tampering was detected when the original hash was compared to the modified hash, and they were totally different.

As shown in the image below, when I removed the full stop which I added to the original text to modify it, the hash changed back to the original hash of the Blessing.txt file.

```
(chukwudi㉿kali)-[~]
└─$ echo "If God be for you, no one can be against you" > Blessing.txt

┌──(chukwudi㉿kali)-[~]
└─$ cat Blessing.txt
If God be for you, no one can be against you

┌──(chukwudi㉿kali)-[~]
└─$ sha256sum Blessing.txt
7222e72311aac87904008f4f79640eaeee5932dc22c4ed2d61aa9f0dca875335  Blessing.txt
```


6 Jun 1
chukwudi@kali: ~
└─$ sha256sum Blessing.txt
7222e72311aac87904008f4f79640eaeee5932dc22c4ed2d61aa9f0dca875335 Blessing.txt
┌──(chukwudi㉿kali)-[~]
└─$ echo "If God be for you, no one can be against you." > Blessing.txt
┌──(chukwudi㉿kali)-[~]
└─$ cat Blessing.txt
If God be for you, no one can be against you.
┌──(chukwudi㉿kali)-[~]
└─$ sha256sum Blessing.txt
e208f0797508689d874d475f997e56aa0f55d1861dc95b240e03534a1d8cb11f Blessing.txt
┌──(chukwudi㉿kali)-[~]
└─$ echo "If God be for you, no one can be against you" > Blessing.txt
┌──(chukwudi㉿kali)-[~]
└─$ cat Blessing.txt
If God be for you, no one can be against you
┌──(chukwudi㉿kali)-[~]
└─$ sha256sum Blessing.txt
7222e72311aac87904008f4f79640eaeee5932dc22c4ed2d61aa9f0dca875335 Blessing.txt


un 18:04
1%
68%
48%
0.0 kB
0.0 kB
Files
Home
Recent
Starred
Network
Wastebasket
Documents
Music
Pictures
Videos
Downloads
Home
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
Videos
Blessing.txt
secret.txt
secret.txt.enc
secret_decrypted.txt

3. **PUBLIC KEY ENCRYPTION / SECURE MESSAGING.**

- **GENERATE A PUBLIC/PRIVATE KEY PAIR**

This method of encryption is known as **Asymmetric encryption** because it is a cryptographic technique that uses two different keys - a public key and a private key to encrypt and decrypt data. A public key is used to encrypt the data, while the private key is used to decrypt the data. This enables **confidentiality** and **Authenticity**

- **Generate a public/private key pair**

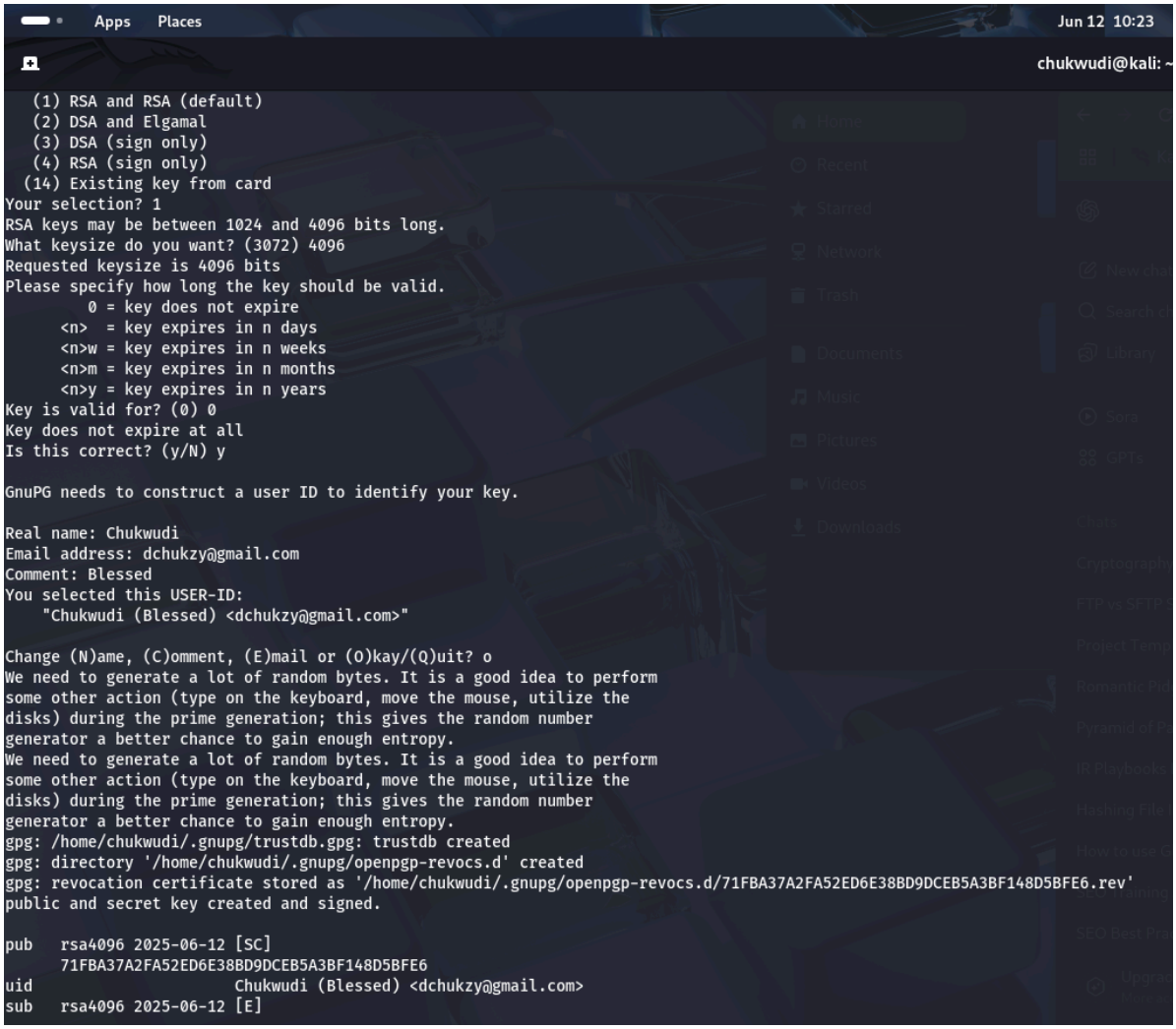
To generate a public/private key, the following command was initiated :

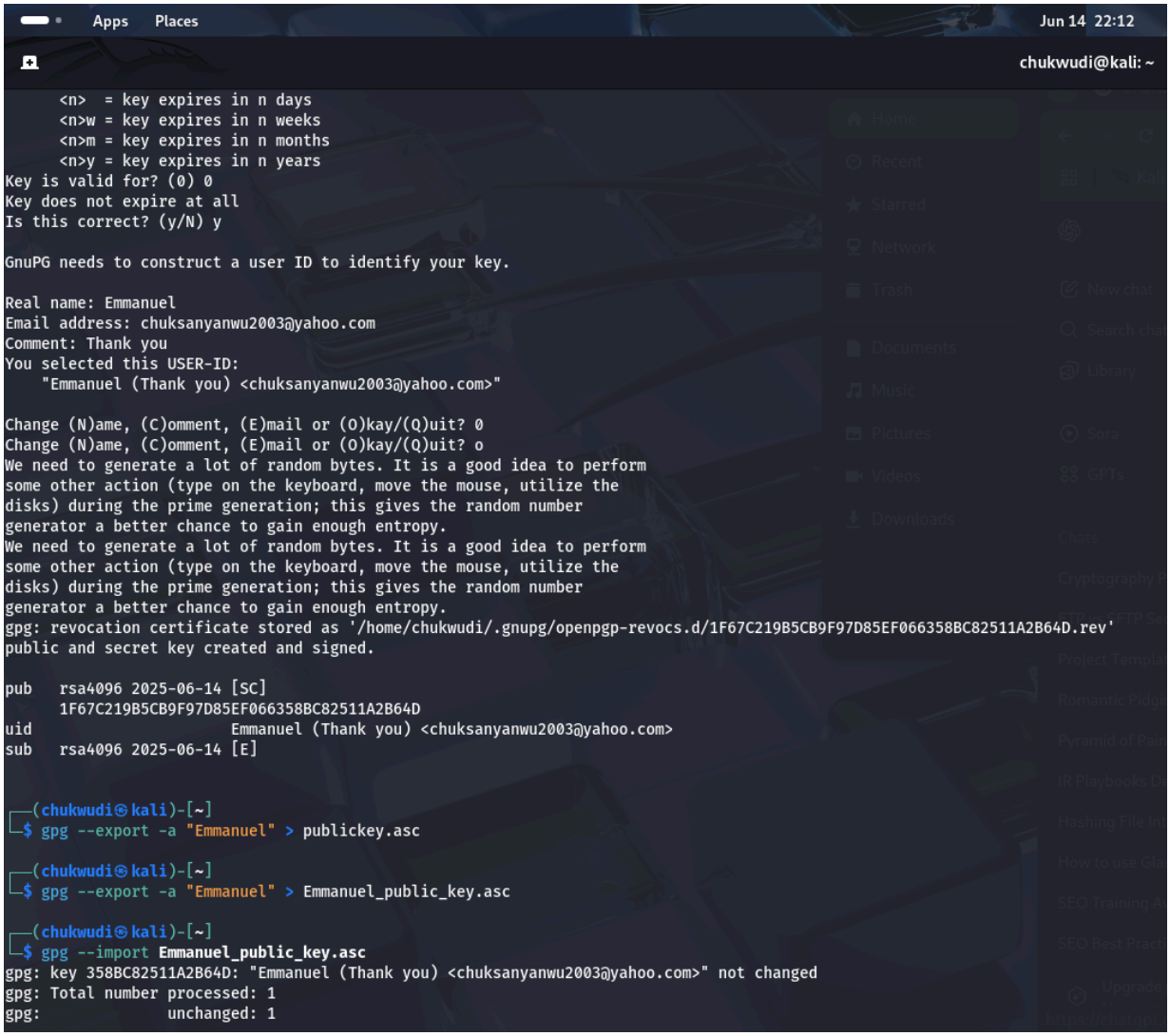
```
┌──(chukwudiⓀkali)-[~]
└─$ gpg --full-generate-key
```

Then the following selections were made as follows when creating the public/private key

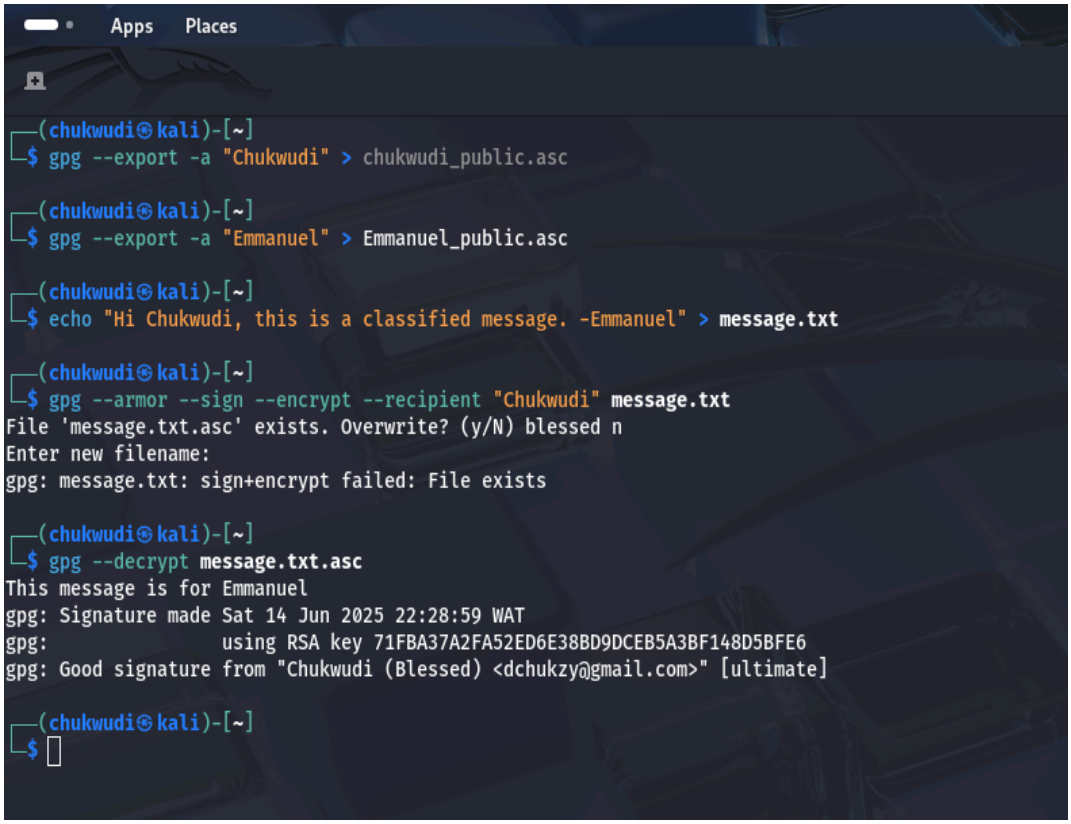
- Choose RSA and RSA (option 1)
- Use key size 4096 bits (stronger security)
- Choose a key to never expire or set a time
- Enter your name/email (e.g., Alice <alice@example.com>)
- Set a passphrase for your private key
- After creation, a private and public key was generated

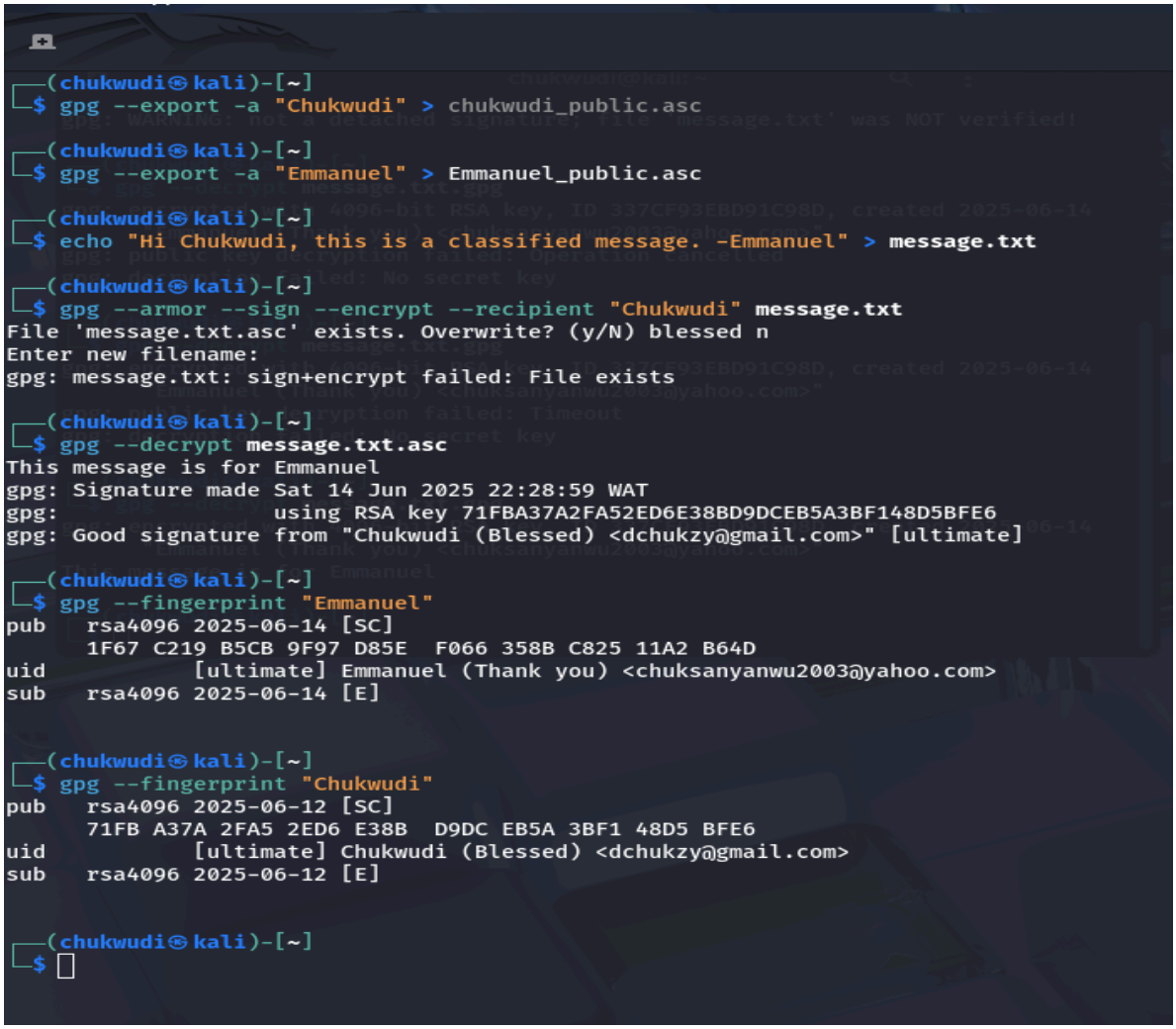
1. **A private key stored locally**
2. **A public key you can export and share**


Apps
Places
Jun 12 10:23
chukwudi@kali: ~
(1) RSA and RSA (default)
(2) DSA and Elgamal
(3) DSA (sign only)
(4) RSA (sign only)
(14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
0 = key does not expire
<n> = key expires in n days
<n>w = key expires in n weeks
<n>m = key expires in n months
<n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
GnuPG needs to construct a user ID to identify your key.
Real name: Chukwudi
Email address: dchukzy@gmail.com
Comment: Blessed
You selected this USER-ID:
"Chukwudi (Blessed) <dchukzy@gmail.com>"
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /home/chukwudi/.gnupg/trustdb.gpg: trustdb created
gpg: directory '/home/chukwudi/.gnupg/openpgp-revocs.d' created
gpg: revocation certificate stored as '/home/chukwudi/.gnupg/openpgp-revocs.d/71FBA37A2FA52ED6E38BD9DCEB5A3BF148D5BFE6.rev'
public and secret key created and signed.
pub rsa4096 2025-06-12 [SC]
71FBA37A2FA52ED6E38BD9DCEB5A3BF148D5BFE6
uid Chukwudi (Blessed) <dchukzy@gmail.com>
sub rsa4096 2025-06-12 [E]

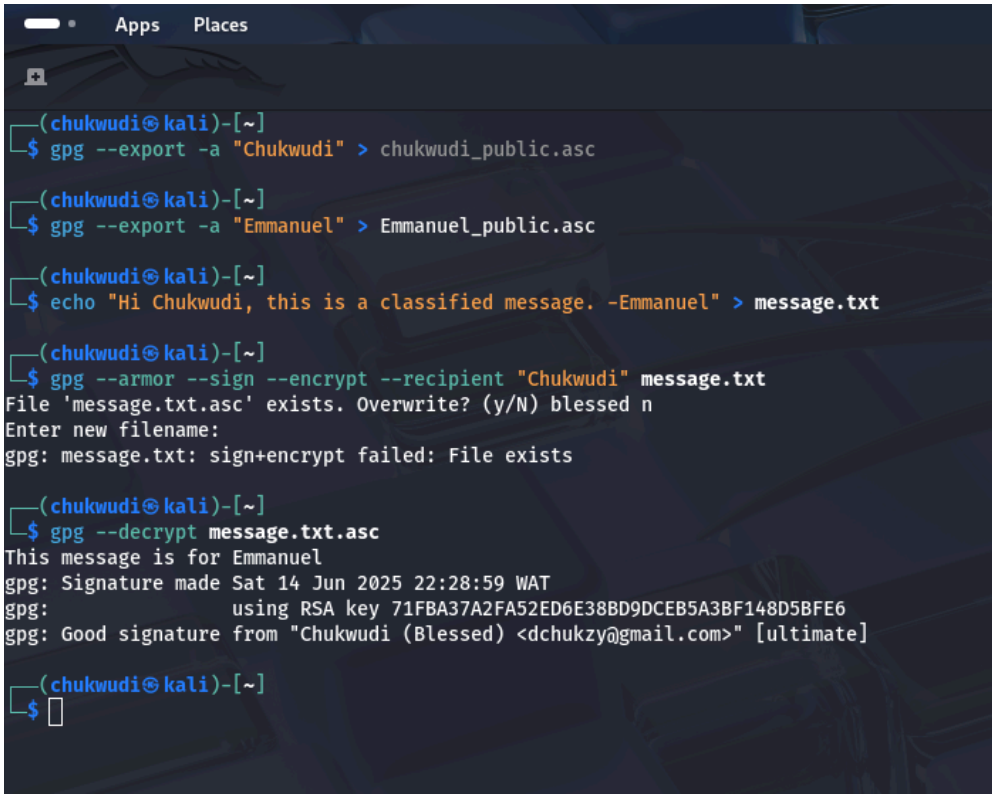


Two GPG pairs were created for the sender and the recipient
Sender : <dchukzy@gmail.com> (Chukwudi)
Receiver: <chuksanyanwu2003@yahoo.com> (Emmanuel)

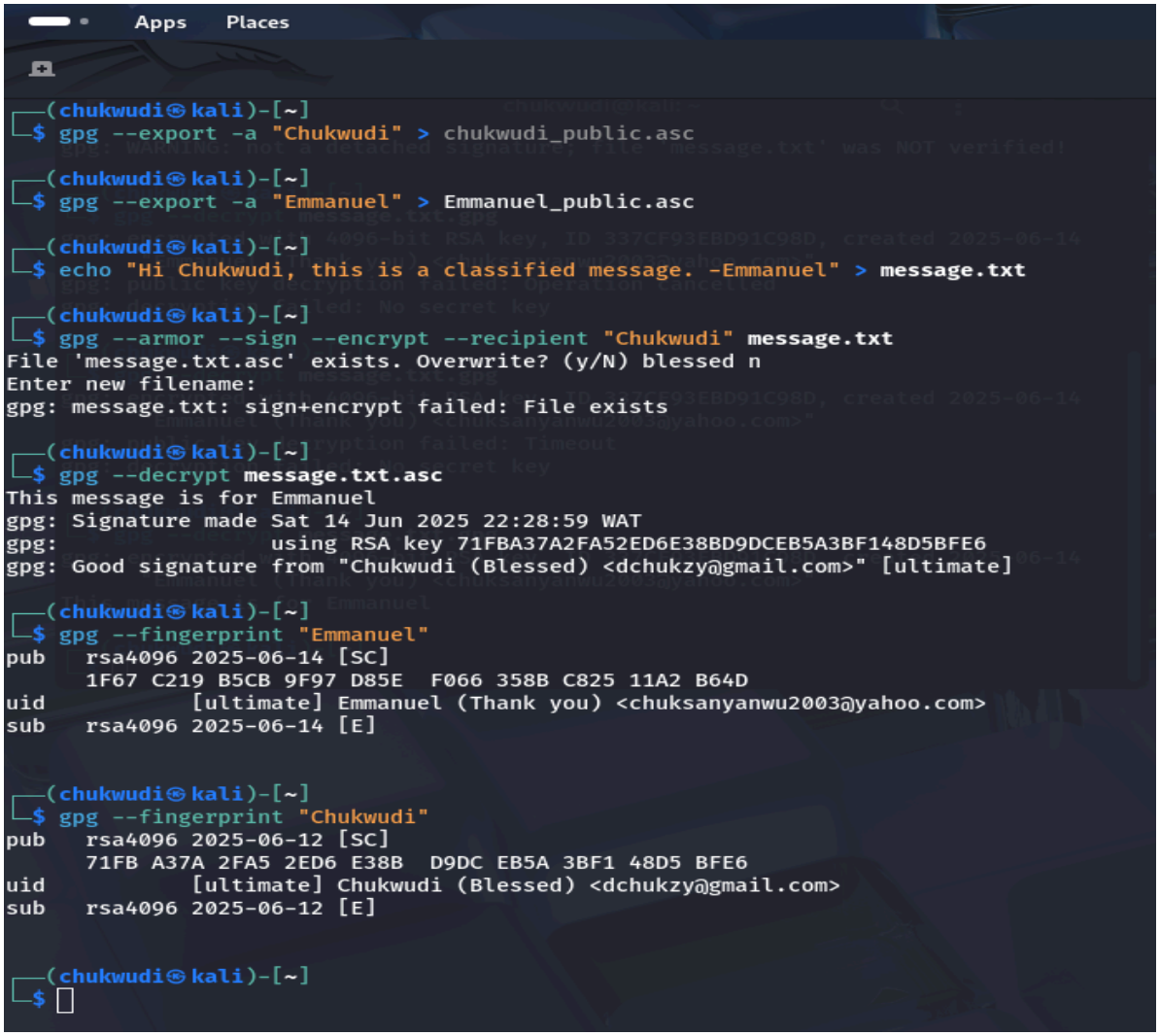

Apps
Places
(chukwudi㉿kali)-[~]
$ gpg --export -a "Chukwudi" > chukwudi_public.asc
(chukwudi㉿kali)-[~]
$ gpg --export -a "Emmanuel" > Emmanuel_public.asc
(chukwudi㉿kali)-[~]
$ echo "Hi Chukwudi, this is a classified message. -Emmanuel" > message.txt
(chukwudi㉿kali)-[~]
$ gpg --armor --sign --encrypt --recipient "Chukwudi" message.txt
File 'message.txt.asc' exists. Overwrite? (y/N) blessed n
Enter new filename:
gpg: message.txt: sign+encrypt failed: File exists
(chukwudi㉿kali)-[~]
$ gpg --decrypt message.txt.asc
This message is for Emmanuel
gpg: Signature made Sat 14 Jun 2025 22:28:59 WAT
gpg: using RSA key 71FBA37A2FA52ED6E38BD9DCEB5A3BF148D5BFE6
gpg: Good signature from "Chukwudi (Blessed) <dchukzy@gmail.com>" [ultimate]
(chukwudi㉿kali)-[~]
$



As shown in the image, a message was encrypted and sent from Chukwudi to Emmanuel and was not altered.



Digital signature confirmed as shown in the image below...



Action	Tool Command Example
Generate keys	`gpg --full-generate-key`
Export public key	`gpg --export -a you@example.com > key.asc`
Import public key	`gpg --import key.asc`
Encrypt message	`gpg --encrypt --sign -r recipient@example.com message.txt`

Decrypt message `gpg --decrypt message.txt.asc`

```
┌──(chukwudi㉿kali)-[~]
└─$ gpg --full-generate-key

gpg: directory '/home/chukwudi/.gnupg' created
gpg: keybox '/home/chukwudi/.gnupg/pubring.kbx' created
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
         0 = key does not expire
       <n>  = key expires in n days
       <n>w = key expires in n weeks
       <n>m = key expires in n months
       <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: Chukwudi
Email address: dchukzy@gmail.com
Comment: Blessed
You selected this USER-ID:
    "Chukwudi (Blessed) <dchukzy@gmail.com>"

gpg: /home/chukwudi/.gnupg/trustdb.gpg: trustdb created
gpg: directory '/home/chukwudi/.gnupg/openpgp-revocs.d' created
gpg: revocation certificate stored as
'/home/chukwudi/.gnupg/openpgp-revocs.d/71FBA37A2FA52ED6E38BD9DCEB5A3BF148D5B
FE6.rev'
public and secret key created and signed.
```

```
pub   rsa4096 2025-06-12 [SC]
      71FBA37A2FA52ED6E38BD9DCEB5A3BF148D5BFE6
uid                 Chukwudi (Blessed) <dchukzy@gmail.com>
sub   rsa4096 2025-06-12 [E]


┌──(chukwudi㉿kali)-[~]
└─$
```

SECOND KEY GENERATED.

```
──(chukwudi㉿kali)-[~]
└─$ gpg --full-generate-key

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072) 4096
Requested keysize is 4096 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: Emmanuel
Email address: chuksanyanwu2003@yahoo.com
Comment: Thank you
You selected this USER-ID:
    "Emmanuel (Thank you) <chuksanyanwu2003@yahoo.com>"
```

```
gpg: revocation certificate stored as
'/home/chukwudi/.gnupg/openpgp-revocs.d/1F67C219B5CB9F97D85EF066358BC82511A2B64
D.rev'
public and secret key created and signed.

pub   rsa4096 2025-06-14 [SC]
      1F67C219B5CB9F97D85EF066358BC82511A2B64D
uid                    Emmanuel (Thank you) <chuksanyanwu2003@yahoo.com>
sub   rsa4096 2025-06-14 [E]


──(chukwudi㉿kali)-[~]
└─$ gpg --export -a "Emmanuel" > publickey.asc

┌──(chukwudi㉿kali)-[~]
└─$ gpg --export -a "Emmanuel" > Emmanuel_public_key.asc

┌──(chukwudi㉿kali)-[~]
└─$ gpg --import Emmanuel_public_key.asc
gpg: key 358BC82511A2B64D: "Emmanuel (Thank you) <chuksanyanwu2003@yahoo.com>"
not changed
gpg: Total number processed: 1
gpg:              unchanged: 1

┌──(chukwudi㉿kali)-[~]
└─$ touch message.txt

┌──(chukwudi㉿kali)-[~]
└─$ echo "This message is for Emmanuel" > message.txt

┌──(chukwudi㉿kali)-[~]
└─$ gpg --encrypt --recipient "Emmanuel" message.txt
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 2u
```

4. **Scenario Simulation**

Choose one of the following real-life scenarios and apply cryptographic techniques to solve it

What Is a File Verification System?

A file verification system ensures that a downloaded file hasn't been tampered with, corrupted, or replaced. This is done using cryptographic hash function like:

- SHA-256

Cryptographic Hash Functions: How They Work

A hash function takes an input (file contents) and generates a fixed-length string called a hash. Even the smallest change in the file produces a totally different hash.

Real-World Example: Verifying a Software Download

Scenario:

You downloaded software.zip from a website. The website also provides a file called software.zip.sha256 containing the correct hash.

Step-by-Step Process:

- Step 1: **Download the File and the Hash**

You usually download two files from the software provider's website:

1. Main File – e.g., `software.zip`
2. Hash File – e.g., `software.zip.sha256`

This contains the official SHA-256 hash value from the developer.

wget https://example.com/downloads/software.zip

wget https://example.com/downloads/software.zip.sha256

- Step 2: **Compute the Hash Locally**

Using a tool like sha256sum:

sha256sum software.zip

Returns:

d2f6e1d1e9b66e8a6d3c2e3ab6d91427a9d0d50bfa9d728db2c68e09d05f3f2e software.zip

🔹 Step 3: **Compare with Provided Hash**

If the computed hash matches the one in software.zip.sha256, the file is safe and untampered.

🔹 Step 4: **If They Don't Match**

This means the file might be:

- Corrupted during download
- Tampered with by a malicious actor
- Different version than expected

Why Use Hashes?

Benefit Description

- Integrity Ensures file has not changed
- Tamper Detection Alerts you to unauthorized changes
- Trust Verification Confirms file came from a trusted source (when hash is signed)