APPLYING CRYPTOGRAPHY IN REAL-WORLD SCENARIOS

Objective: The objective of this project is to demonstrate the practical application of cryptographic techniques in securing digital communication and data integrity.. It focuses on using OpenSSL for symmetric encryption, SHA-256 for integrity verification and public/private key pairs with digital signatures to simulate secure communication and explore real-world applications of cryptography.

A) DATA PROTECTION WITH ENCRYPTION

1. Introduction

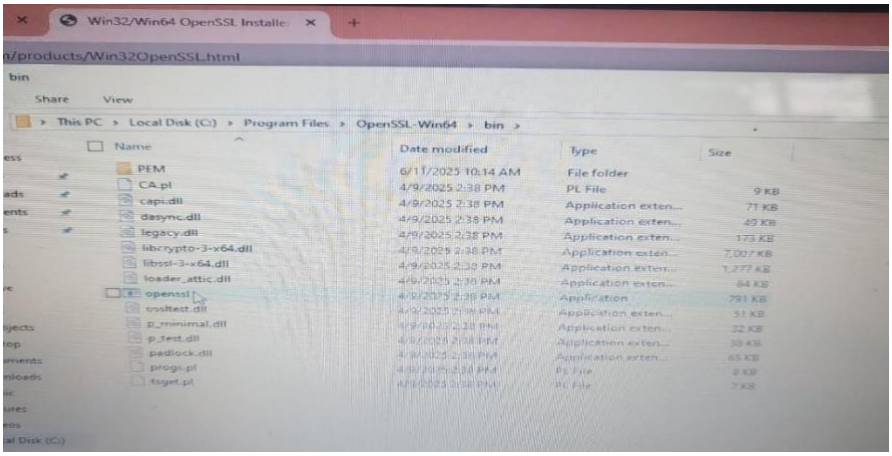
This report documents the process of encrypting and decrypting a file using OpenSSL on a Windows operating system. The purpose of this exercise is to demonstrate how symmetric encryption can protect sensitive information from unauthorized access.

2. Tools used

- Windows 10 Laptop
- OpenSSL for Windows
- Command Prompt
- Notepad

3. Step-by-step procedure

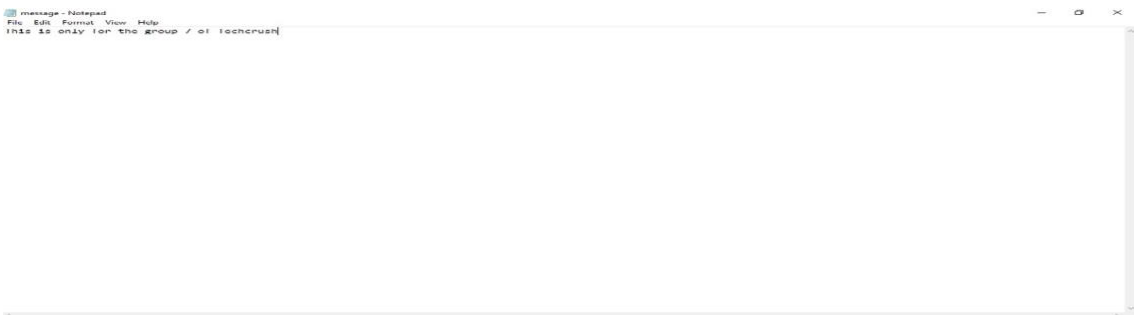
- **Step 1 - Installation and Setup:** OpenSSL was downloaded and installed from a trusted source. After installation, the bin directory path was noted.



- **Step 2 - Verifying OpenSSL Installation:** Command prompt was used to navigate to the OpenSSL installation path and installation was verified using the OpenSSL command.

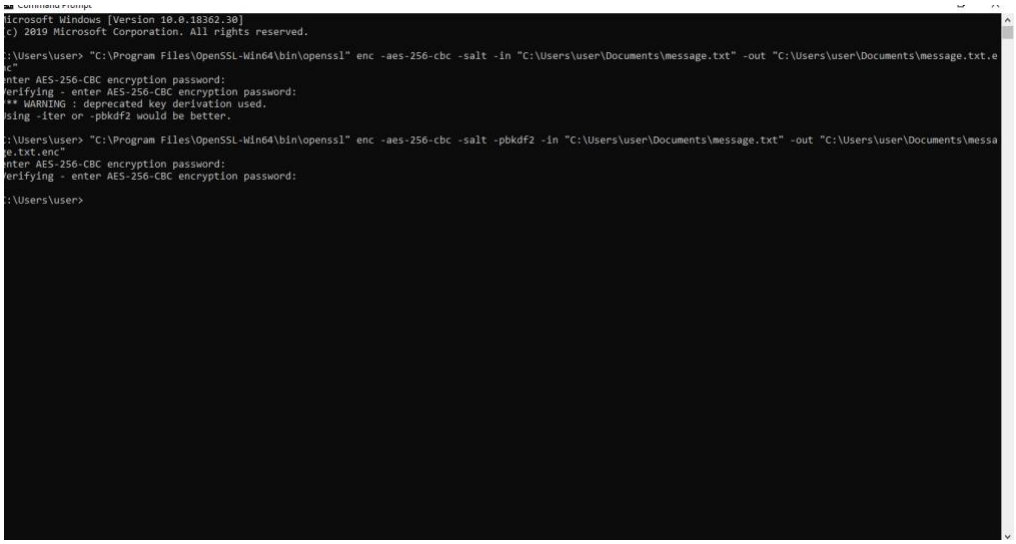
```
"C:\Program
Files\OpenSSL-Win64\bin\openssl"
```

- Step 3 - Creating a Plain Text File: I used Notepad to create a file named message.txt with some sample text, saved in the Documents folder.



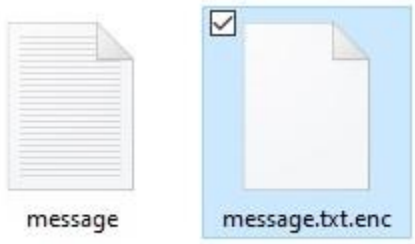
- Step 4 - Encrypting the File: I used the AES-256-CBC encryption algorithm with a password. Initially I used the default key derivation method, which raised a warning about deprecated security.

- Step 5 - Encrypting Again Using pbkdf2 (Secure Method)



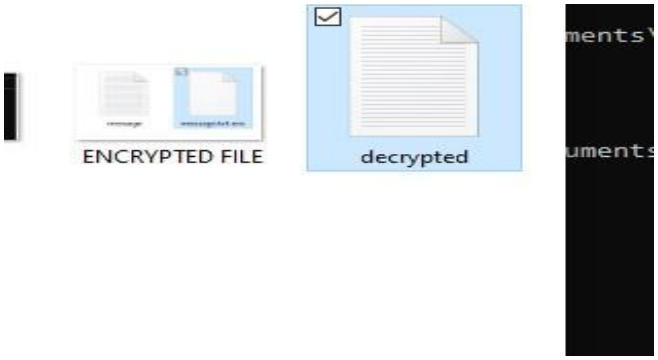
Re-ran the encryption command with the -pbkdf2 flag to use modern password based key derivation function 2.

- Step 6 - Attempting to Open Encrypted File: Attempted to open message.txt.enc with a text editor, which failed as expected. This confirmed that the file was successfully encrypted.



- Step 7 - Decrypting the File: Decrypted the file using the same password and confirmed the contents matched the original message.

```
"C:\Program Files\OpenSSL-Win64\bin\openssl"
enc -aes-256-cbc -d -pbkdf2 -in "C:
\Users\user\Documents\message.txt.enc" -out
"C:\Users\user\Documents\decrypted.txt"
```



4. **What the Encryption Protects:** Encryption protects the confidentiality and integrity of information. In this scenario, it prevents unauthorized individuals from reading the contents of message.txt unless they have the password used for encryption. This is crucial when storing or transmitting sensitive information such as personal data, legal documents, business strategies, or login credentials.

5. Challenges Faced

- Path issues with OneDrive documents folder causing OpenSSL to fail in locating the file.
- Confusion due to case sensitivity and spaces in the command line paths.
- Use of deprecated key derivation method initially, which required re-encryption with a more secure option.
- Difficulty with pasting into Command Prompt and confirming the right syntax.
- Ensuring OpenSSL was properly installed and that the system recognized its path.

6. Real World Applications

Symmetric encryption like AES is widely used in real-world scenarios including:

- Protecting files on personal computers and corporate servers
- Encrypting data before storing in cloud storage services
- Securing email attachments and messaging apps
- Ensuring privacy in financial records, health documents, and legal files
- Used by backup software, VPNs, and secure file transfer tools (e.g., SFTP, SCP)

7. Conclusion

This practical session demonstrated a full encryption-decryption cycle using OpenSSL. It reinforced the importance of data protection, highlighted common command line issues, and emphasized the move toward stronger cryptographic practices like using pbkdf2. Successfully completing this task provides foundational skills relevant to cybersecurity and compliance fields.

B) HASHING AND INTEGRITY CHECKING

1. Tools used

- SHA-256 hash function
- Command-line tool: sha256sum (on Kali Linux)
- Sample file: message.txt

2. Procedure

- **Select a file:** The file selected for the has text was named ‘message.txt’

  Original content: This is a confidential message meant for integrity testing

- **Generate initial hash:** The initial SHA-256 hash was generated using the command: sha256sum message.txt > hash_original.txt
    - Sample output: e99a18c428cb38d5f260853678922e03abd8334b... message.txt

```
$ sha256sum message.txt
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd7d23f3c4a  message.txt
```

- **Modify the file:** The file 'message.txt' was edited slightly by changing the word “testing” to “check”. This simulates a small, unauthorized alteration.
    - Modified content: This is a confidential message meant for integrity check.

- **Generate new hash:** The SHA-256 hash of the modified file was generated using the same command: sha256sum message.txt > hash_modified.txt

```
$ sha256sum message.txt
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd7d23f3c4a  message.txt
```

- **Compare the hashes:** The newly generated hash differed completely from the original, despite the small change in the file. This illustrates the avalanche effect in cryptographic hash functions.

```
$ sha256sum message.txt
c8fed00eb2e87f1cee8e90ebbe870c190ac3848c5ac3cce7c1a7ba3e  message.txt
```

3. Explanation

What Hashing Proves: Hash functions such as SHA-256 generate a fixed-size output (digest) that uniquely represents the data in a file. Any change to the file, even a single character, results in a completely different hash output. This property is known as the avalanche effect.

Why It Matters in Real Life:

- Verifying file downloads from trusted sources using checksums
- Ensuring backups have not been altered or corrupted
- Supporting forensic analysis and software integrity validation

4. Conclusion

This exercise demonstrated how hashing can be used as a reliable and efficient method to detect changes in file content. It highlights the importance of integrity checks in cybersecurity operations. By using hashing algorithms such as SHA-256, systems can quickly detect unauthorized modifications and ensure that digital assets remain intact and unaltered.

C) PUBLIC KEY ENCRYPTION / SECURE MESSAGING

1. Methodology

- Generation of a public/private key pair
- Usage of the keys: To encrypt a message to someone (using their public key). Decrypt a message (using your private key).
- Demonstrated a digital signature (signing a file or message to prove identity).
- How it protects the aspects of CIA triad as well as the challenges and the real-world application.

- **Generation of the key pair**

Key Length

1024 Generate key pair

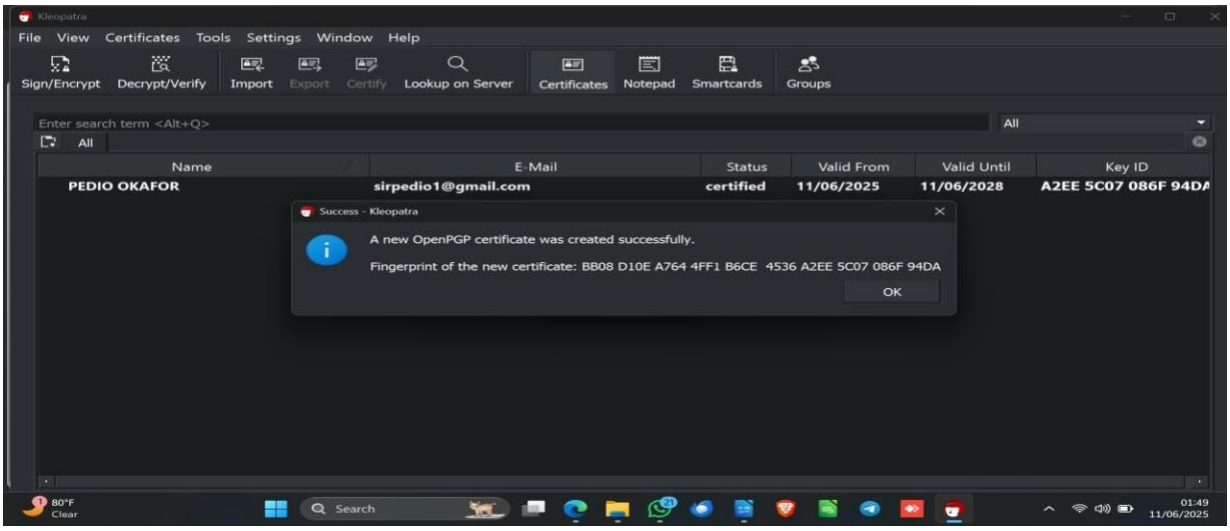
Private key

```
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDEIXxjHJZTzPzSPl2uC+6AmPUl4p3hOUKHo8vEJ2L//bJd+fbw
5as2OFYvF8T6quXDtevS/eSksW1KmDHohPaLfN94sMYYbrt4h1dIrgRPxmIVBHVr
oAzGHzZeT3+tKNFhIMfY6ZtZa6fK57tqBT6PjiNcxCFqJb48U8z+tcJCkQIDAQAB
AoGASl1r/WRsd0/FGK5+9KwffXEBmqDHPgto92carsBEYE3AILPTdtmORlusCeUY
UfcUcVV8Gh8cX3nKt7rnSYOIuwbRuagLcRD2qyBalI4sbAUBDZhwhNXI70mfYLNd
vXxn7xx9ARcdG2mFYIr5zTceA8r7LzvKowvXrjZPldcPb/ECQQDoGnfmq6HO+Pgz
YpsUvNWd7TZR1swql3QBNQ7GXcaG3L9KBR9/zVtH6fXPhml9C4aP1OAeoRwvHr3/
XGzTDqHNAkEA2FLkByyIRySL8DZwK0r6WsL0YPvNvlv0uMdmBy+YFkNttzGEzHJJ
hltKfBqq6TvQRPWgxC67sp7vxIpS6WUv1QJAS1xkt+ncVNZzyENo9RfHCEsACxyC
GAYq7UVr8a57Ckjcd00oiBeGXtgxLtuGnspCO6szxd4rx4TkzegVkYW7tQJAFaZe
GfW0AeK/eqAII90SJ0lBqf8pfTa4zFFwfaGeyAOhlGcJUmjRwPecAcWte0A4PN6U
Y24F7P+JIZNybtdorQJBANjsgF4RRnK7FSLdwmdgZwvnrMt7NcANnQU9fQmYa/8p
hOM6xdgC1mM0ka+Jj4zElAgdlSQ1xI55tbwUD73j4lo=
-----END RSA PRIVATE KEY-----
```

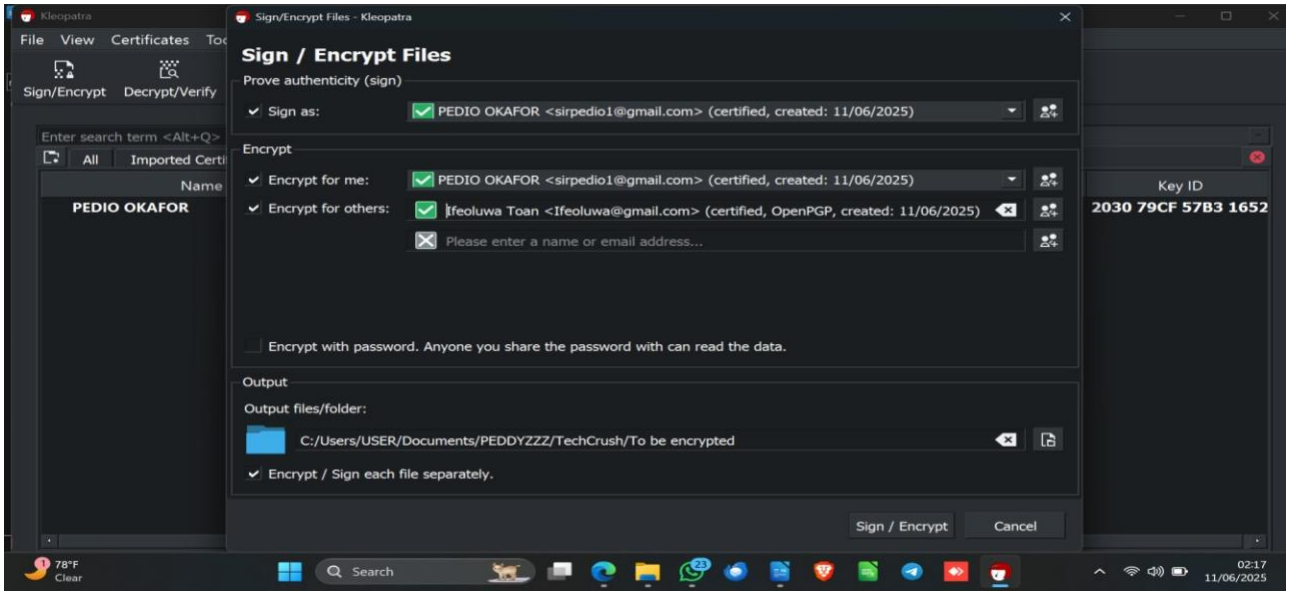
Public key

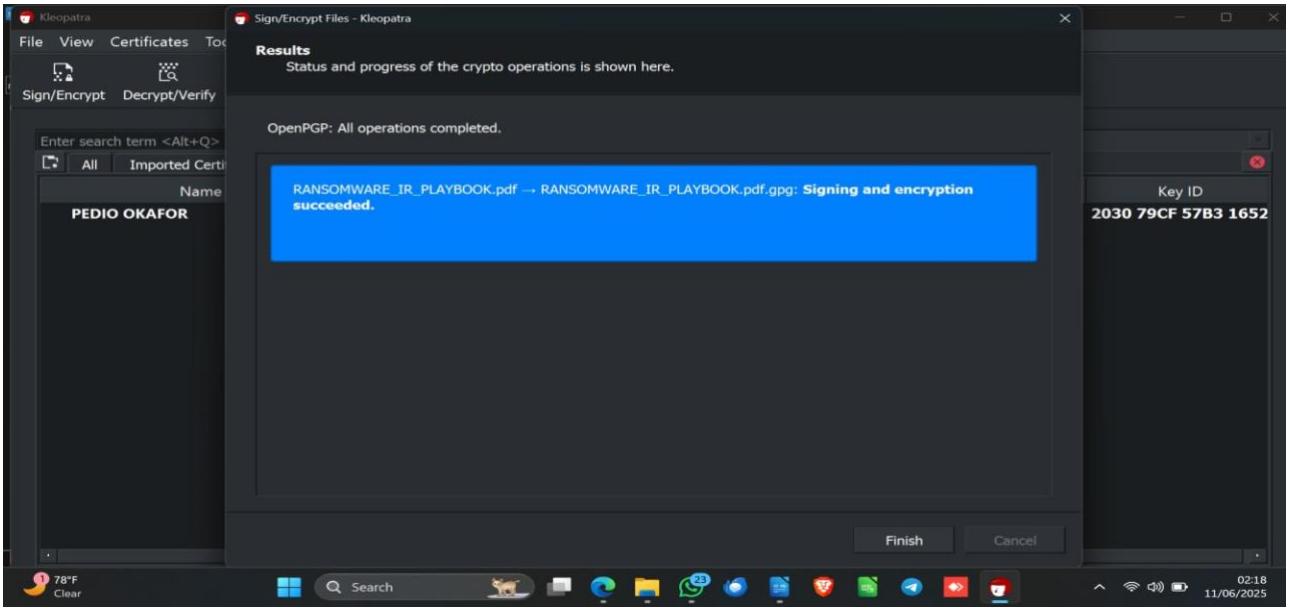
```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDEIXxjHJZTzPzSPl2uC+6AmPUl
4p3hOUKHo8vEJ2L//bJd+fbw5as2OFYvF8T6quXDtevS/eSksW1KmDHohPaLfN94
sMYYbrt4h1dIrgRPxmIVBHVroAzGHzZeT3+tKNFhIMfY6ZtZa6fK57tqBT6PjiNc
xCFqJb48U8z+tcJCkQIDAQAB
-----END PUBLIC KEY-----
```

- **Creation of a passphrase**

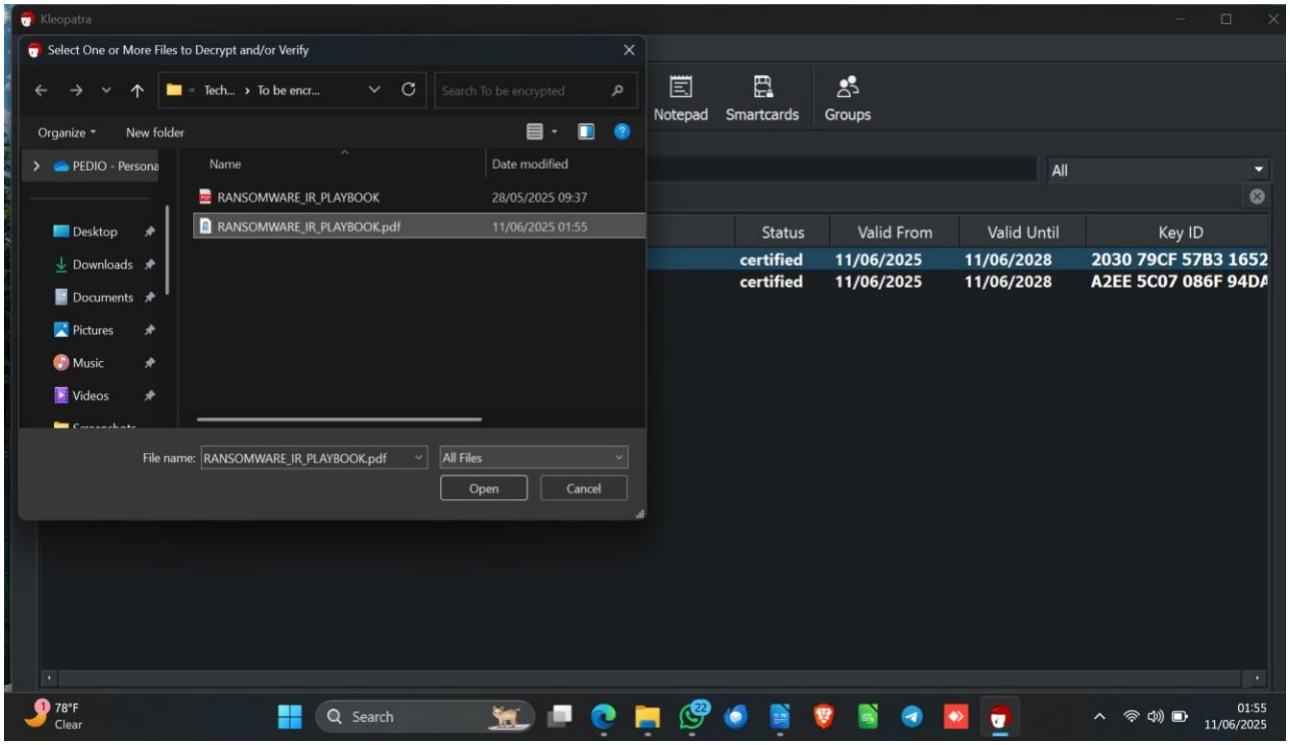


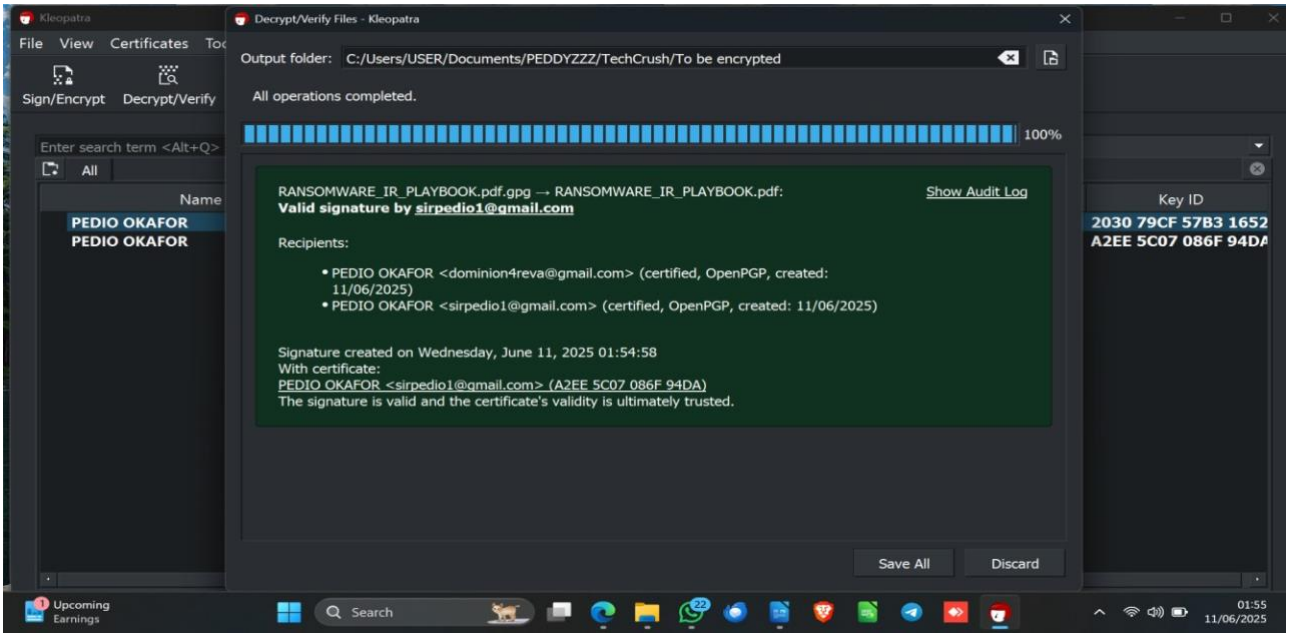
- **Encrypting of the Files as well as signing to prove identity.**



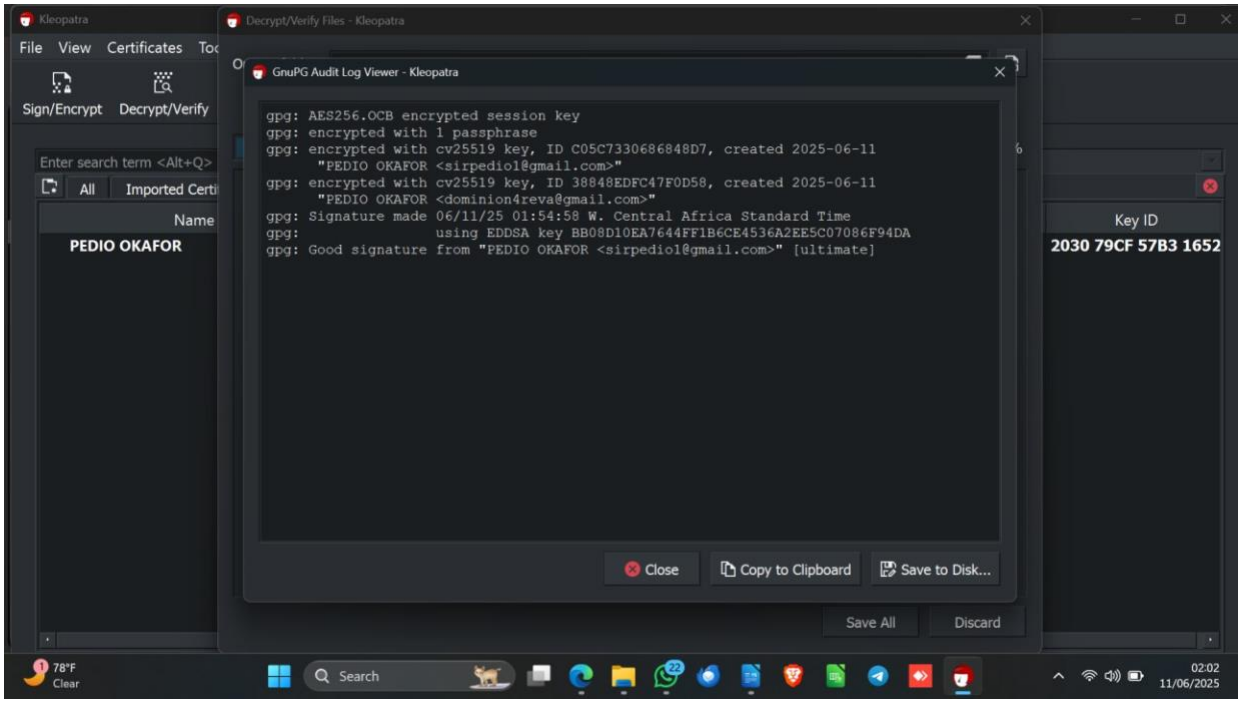


- **Decryption of the Files as well as signing to prove identity.**





• **Audit logs of action done**



2, Public Key Encryption/Secure Messaging protects the following aspects of the CIA triad:

• **Confidentiality:** It ensures that sensitive information remains private and accessible only to authorized parties.

• **Integrity:** It verifies that the message/data/file has not been altered or tampered with during transmission.

• **Authentication:** It confirms the identity of the sender and ensures the information is coming from a trusted source.

2. **Challenges:** Despite its benefits, Public Key Encryption/Secure Messaging faces different challenges:

• **Key Management:** Securely generating, distributing and managing public and private keys can be complex.

• **Scalability:** As the number of users and messages increases, the complexity of key management an encryption/decryption processes can be quite tasking and become a bottleneck.

• **Quantum Computing Threats:** The advent of quantum computing poses a potential threat to the security of public-key encryption algorithms.

3. **The Real-World Application includes:**

• **Secure Email Communication:** Protecting sensitive information exchanged via email.

• **Virtual Private Networks (VPNs**): Establishing secure connections for remote access to networks.

• **Secure Messaging Apps:** Protecting user conversations and data in messaging applications.

D) SCENARIO SIMULATION

To simulate a secure email exchange using OpenSSL, we can leverage its capabilities to generate cryptographic keys, certificates, and encrypt/decrypt messages. This setup mimics secure email communication by using S/MIME (Secure/Multipurpose Internet Mail Extensions), which is commonly used to sign and encrypt emails.

Below, I will provide a step-by-step guide to simulate a secure email exchange between two parties (Alice and Bob) on a Kali Linux system.

Linux commands used:

- Show directory: ls
- Show current directory: pwd
- Enter directory: cd dirname
- Ex: cd Desktop
- Exit a directory: cd …
- Create file: nano
- View file contents: cat filename
- Clear: clear (ctrl + l)

1. ENCRYPTION USING OPEN SSL

- Open terminal
- Type “OpenSSL version”: This command helps you check if you have OpenSSL installed in your OS and the version and date available.
- Check list of cipher (encryption) commands. “OpenSSL list -cipher-commands”

  This command helps you check the list of encryption commands available, and which you can choose from.

- **Create a file and enter any text (using explorer), using nano (text editor)**

Type “nano’ and press enter

Enter any message

Press “ctrl + X” to close

Press “Y” to save

And enter the name of the file (Capstone)

Type “cat Bob” to view the message.

- Type “OpenSSL enc cipher type format -in filename -out destination file”: This command is used on Linux to encrypt or decrypt a file using a specified cipher algorithm. It’s part of the OpenSSL toolkit, commonly used for cryptographic operations.

2. Function of the Command

The command encrypts or decrypts the input file (filename) using the specified cipher (cipher type) and writes the result to the output file (destination file). The format option specifies how the input/output is handled (e.g., binary or base64). It’s typically used for:

- **Encryption**: Securing a file so only those with the key/password can access its contents.

- **Decryption**: Recovering the original file from an encrypted version using the correct key/password.
- **File Protection**: Ensuring confidentiality for sensitive data, such as emails, documents, or backups.

- Enter the password: Here you are to choose a private key, for the encryption this key you are also to share with whoever would be decrypt the content of the message.

3. DECRYPTION USING OPEN SSL

- Type “OpenSSL enc ciphertype -d format -in filename”: This command is used to decrypt a file that was previously encrypted using the specified cipher algorithm (ciphertype). It’s part of the OpenSSL toolkit, which provides cryptographic functions. This command reads an encrypted input file (filename), decrypts it, and outputs the decrypted data, typically to the terminal (stdout) unless redirected.
- Enter the password used in encryption