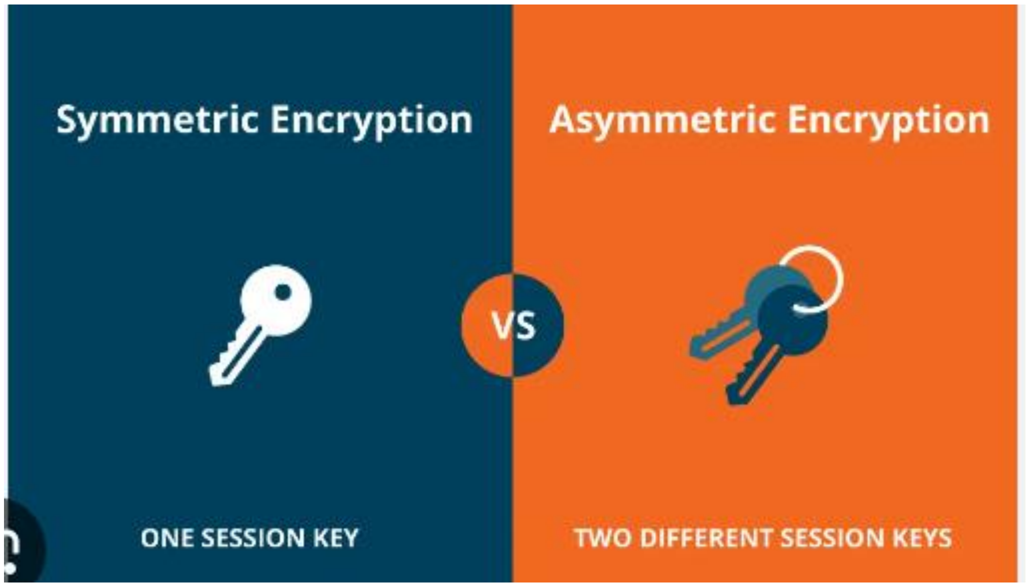
Understanding Public Key Infrastructure (PKI) and Certificate Management for Secure Web Transactions



Public Key Infrastructure (PKI) and Certificate Management

Public Key Infrastructure (PKI) is a system that facilitates secure communications and digital signatures using pairs of cryptographic keys — public and private. PKI includes key components such as certificate authorities (CAs), which issue digital certificates that link a user's identity to their public key, establishing trust in online transactions.

Key Components of PKI:

- **Public and Private Keys**: Asymmetric keys used for secure communication and authentication.
- **Digital Certificates**: Electronic documents issued by CAs that confirm a user's identity and contain the public key.
- **Certificate Authorities (CAs)**: Trusted entities responsible for issuing and managing certificates.

Certificate Management involves overseeing the entire lifecycle of a digital certificate, including:

1. **Issuance**: Creating and validating certificates through CAs.
2. **Deployment**: Installing certificates on servers to enable secure connections.
3. **Renewal and Expiration**: Tracking and renewing certificates before they expire.
4. **Revocation**: Invalidating compromised certificates and updating users through Certificate Revocation Lists (CRLs).

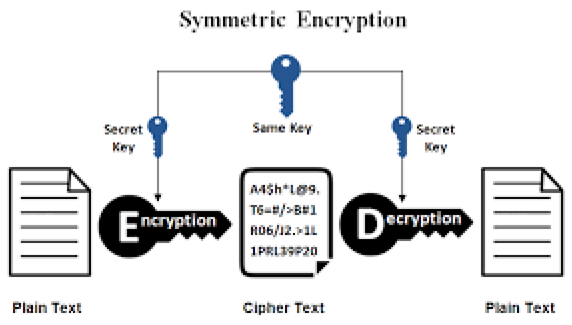
5. **Auditing**: Regularly reviewing certificate usage to ensure compliance with security policies.**symmetric and asymmetric encryption**

1. Symmetric Encryption

In symmetric encryption, the **same key** is used for both encryption and decryption. This type of encryption is typically fast and suitable for large data sets but requires secure key distribution.

Process:

1. The sender encrypts the message using a **shared secret key**.
2. The encrypted message is sent to the receiver.
3. The receiver decrypts the message using the **same key**.



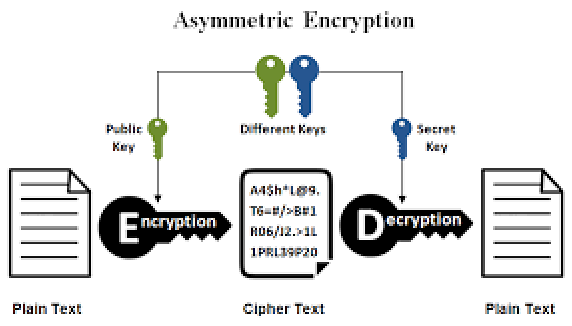
Key Challenge: Securely sharing the symmetric key between sender and receiver.

2. Asymmetric Encryption

In asymmetric encryption, two different but mathematically related keys are used: a **public key** for encryption and a **private key** for decryption. The public key is shared openly, while the private key is kept secret.

Process:

1. The sender encrypts the message using the **recipient's public key**.
2. The encrypted message is sent to the receiver.
3. The receiver decrypts the message using their **private key**.



Key Advantage: No need to share a secret key; only the private key must remain secure.

Both methods are crucial in cybersecurity, with **symmetric encryption** being efficient for bulk data encryption and **asymmetric encryption** providing secure key exchanges and digital signatures.

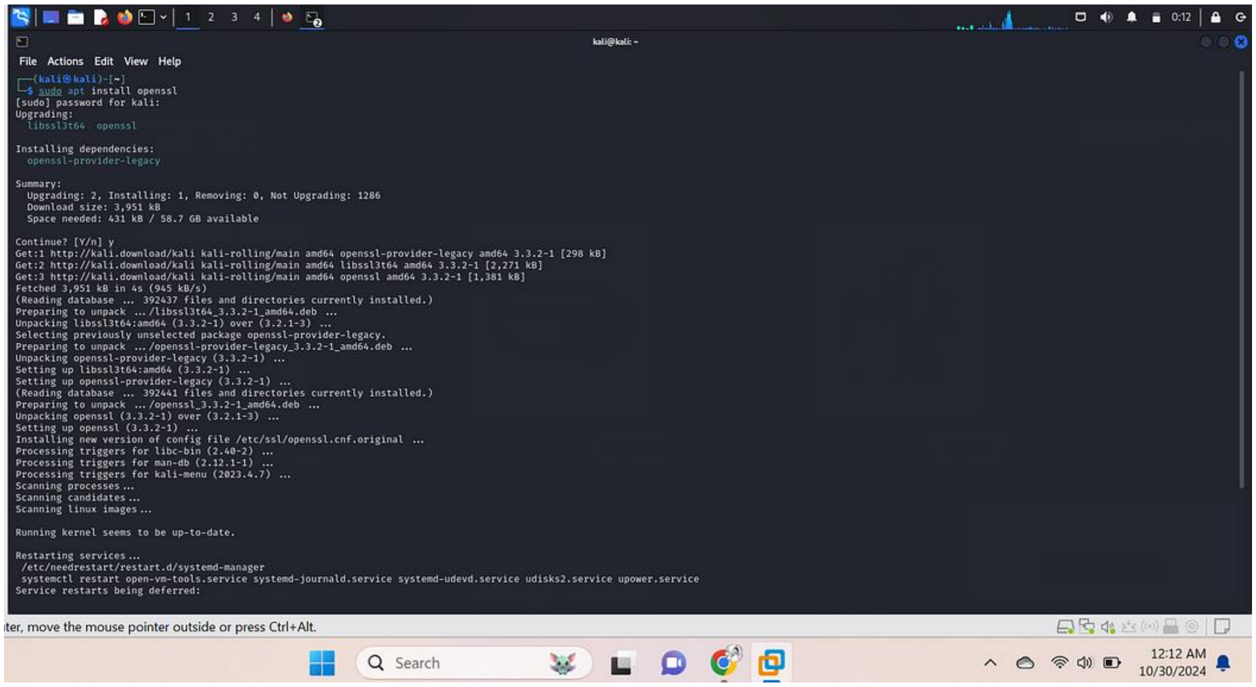
Proof of concepts:

symmetrical Encryption using openSSL

Step 1: Generate a Symmetric Key

For symmetric encryption with OpenSSL, we can use a passphrase as a key instead of generating a separate key file.

But we must install openssl or check if it is already install:



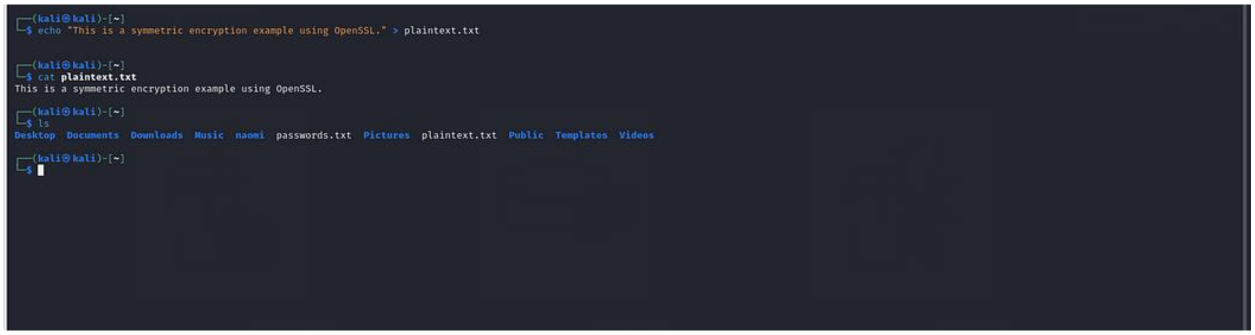
```
┌──(kali㉿kali)-[~]
└─$ sudo apt install openssl
[sudo] password for kali:
Upgrading:
  libssl3t64  openssl

Installing dependencies:
  openssl-provider-legacy

Summary:
  Upgrading: 2, Installing: 1, Removing: 0, Not Upgrading: 1286
  Download size: 3,951 kB
  Space needed: 431 kB / 58.7 GB available

Continue? [Y/n] y
Get:1 http://kali.download/kali kali-rolling/main amd64 openssl-provider-legacy amd64 3.3.2-1 [298 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 libssl3t64 amd64 3.3.2-1 [2,271 kB]
Get:3 http://kali.download/kali kali-rolling/main amd64 openssl amd64 3.3.2-1 [1,381 kB]
Fetched 3,951 kB in 4s (945 kB/s)
(Reading database ... 392437 files and directories currently installed.)
Preparing to unpack .../libssl3t64_3.3.2-1_amd64.deb ...
Unpacking libssl3t64:amd64 (3.3.2-1) over (3.2.1-3) ...
Selecting previously unselected package openssl-provider-legacy.
Preparing to unpack .../openssl-provider-legacy_3.3.2-1_amd64.deb ...
Unpacking openssl-provider-legacy (3.3.2-1) ...
Setting up libssl3t64:amd64 (3.3.2-1) ...
Setting up openssl-provider-legacy (3.3.2-1) ...
(Reading database ... 392441 files and directories currently installed.)
Preparing to unpack .../openssl_3.3.2-1_amd64.deb ...
Unpacking openssl (3.3.2-1) over (3.2.1-3) ...
Setting up openssl (3.3.2-1) ...
Installing new version of config file /etc/ssl/openssl.cnf.original ...
Processing triggers for libc-bin (2.40-2) ...
Processing triggers for man-db (2.12.1-1) ...
Processing triggers for kali-menu (2023.4.7) ...
Scanning processes...
Scanning candidates...
Scanning linux images...

Running kernel seems to be up-to-date.

Restarting services...
 /etc/needrestart/restart.d/systemd-manager
 systemctl restart open-vm-tools.service systemd-journald.service systemd-udevd.service udisks2.service upower.service
Service restarts being deferred:
```

Step 2: Encrypt a Message with a Symmetric Key

1. **Create a Plaintext File**:

Let's start by creating a sample plaintext file. Run the following command to create it:

echo "This is a symmetric encryption example using OpenSSL." > plaintext.txt

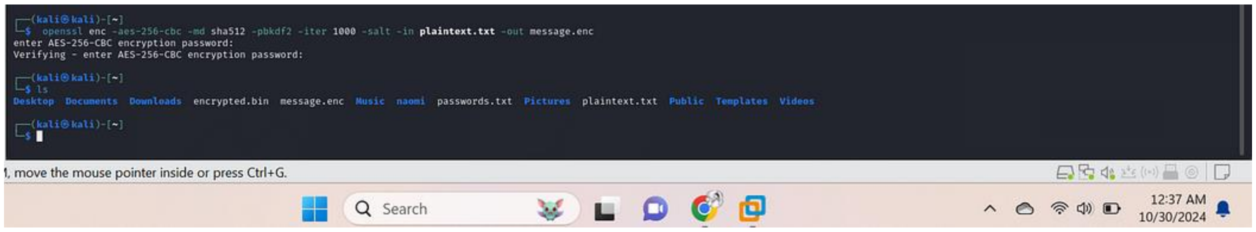


```
┌──(kali㉿kali)-[~]
└─$ echo "This is a symmetric encryption example using OpenSSL." > plaintext.txt


┌──(kali㉿kali)-[~]
└─$ cat plaintext.txt
This is a symmetric encryption example using OpenSSL.

┌──(kali㉿kali)-[~]
└─$ ls
Desktop  Documents  Downloads  Music  naomi  passwords.txt  Pictures  plaintext.txt  Public  Templates  Videos

┌──(kali㉿kali)-[~]
└─$ █
```

2. Encrypt the File:

Use the openssl enc command to encrypt the file with a passphrase. Here's an example with AES-256-CBC, a common symmetric encryption algorith

openssl enc -aes-256-cbc -md sha512 -pbkdf2 -iter 1000 -salt -in plaintext.txt -out message.enc.



```
┌──(kali㉿kali)-[~]
└─$  openssl enc -aes-256-cbc -md sha512 -pbkdf2 -iter 1000 -salt -in plaintext.txt -out message.enc
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:

┌──(kali㉿kali)-[~]
└─$ ls
Desktop  Documents  Downloads  encrypted.bin  message.enc  Music  naomi  passwords.txt  Pictures  plaintext.txt  Public  Templates  Videos

┌──(kali㉿kali)-[~]
└─$ █
```

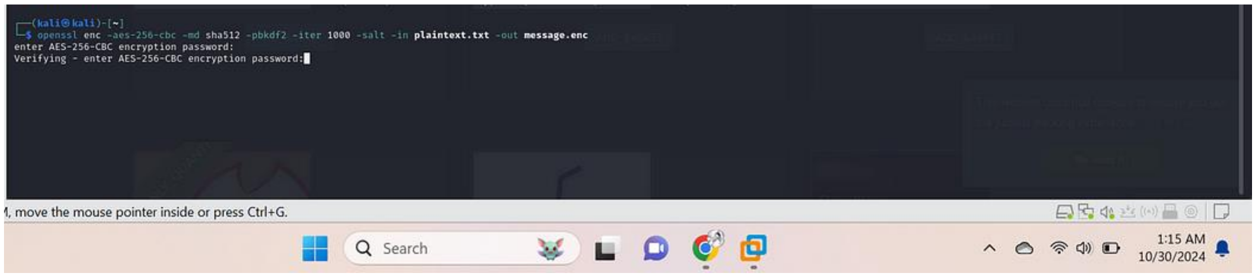
Explanation:

This command encrypts a file named plaintext.txt using the Advanced Encryption Standard (AES) with a 256-bit key size in Cipher Block Chaining (CBC) mode.

Here's a breakdown of the options used:

- **-aes-256-cbc**: Specifies the encryption algorithm (AES-256-CBC).
- **-md sha512**: Uses SHA-512 as the message digest algorithm for key derivation.
- **-pbkdf2**: Employs the Password-Based Key Derivation Function 2 (PBKDF2) to derive a cryptographic key from the user-provided passphrase.
- **-iter 1000**: Sets the number of iterations for the PBKDF2 algorithm. A higher iteration count increases the security of the derived key.
- **-salt**: Adds a random salt to the encryption process, making it more resistant to dictionary attacks.
- **-in plaintext.txt**: Specifies the input file to be encrypted.
- **-out message.enc**: Specifies the output file containing the encrypted data.

The encrypted output is saved as message.enc. You will be prompted to enter a password, which is required to decrypt the file later.

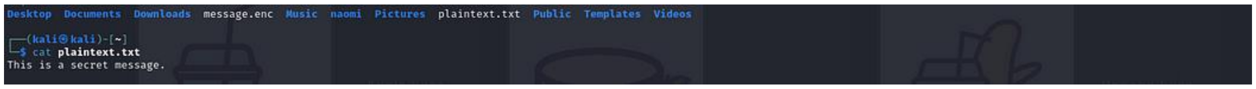


```
┌──(kali㉿kali)-[~]
└─$ openssl enc -aes-256-cbc -md sha512 -pbkdf2 -iter 1000 -salt -in plaintext.txt -out message.enc
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
```

Step 3: Decrypt the Message with the Same Symmetric Key

To decrypt the message, we use the same command but with the -d (decrypt) option:

openssl enc -aes-256-cbc -md sha512 -pbkdf2 -in message.enc -out plaintext.txt -d

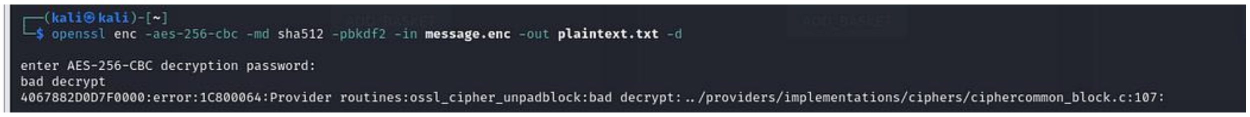


```
Desktop  Documents  Downloads  message.enc  Music  naomi  Pictures  plaintext.txt  Public  Templates  Videos

┌──(kali㉿kali)-[~]
└─$ cat plaintext.txt
This is a secret message.
```

Explanation:

- -d tells OpenSSL to decrypt.
- The rest of the parameters are similar to encryption but with the output going to *plaintext.txt*

When you run this command, OpenSSL prompts you to enter the password that was used during encryption. Upon entering the correct password, *message.enc* will be decrypted, and its original content will be saved to *plaintext.txt* If an incorrect password is entered, decryption will fail, and the content will remain inaccessible.



```
┌──(kali㉿kali)-[~]
└─$ openssl enc -aes-256-cbc -md sha512 -pbkdf2 -in message.enc -out plaintext.txt -d

enter AES-256-CBC decryption password:
bad decrypt
4067882D0D7F0000:error:1C800064:Provider routines:ossl_cipher_unpadblock:bad decrypt:../providers/implementations/ciphers/ciphercommon_block.c:107:
```

Asymmetrical encryption using openSSL

Asymmetric encryption example using **OpenSSL**. In asymmetric encryption, we use a pair of keys: a **public key** to encrypt data and a **private key** to decrypt it. This is useful for scenarios like secure message exchanges, SSL/TLS, and digital signatures.

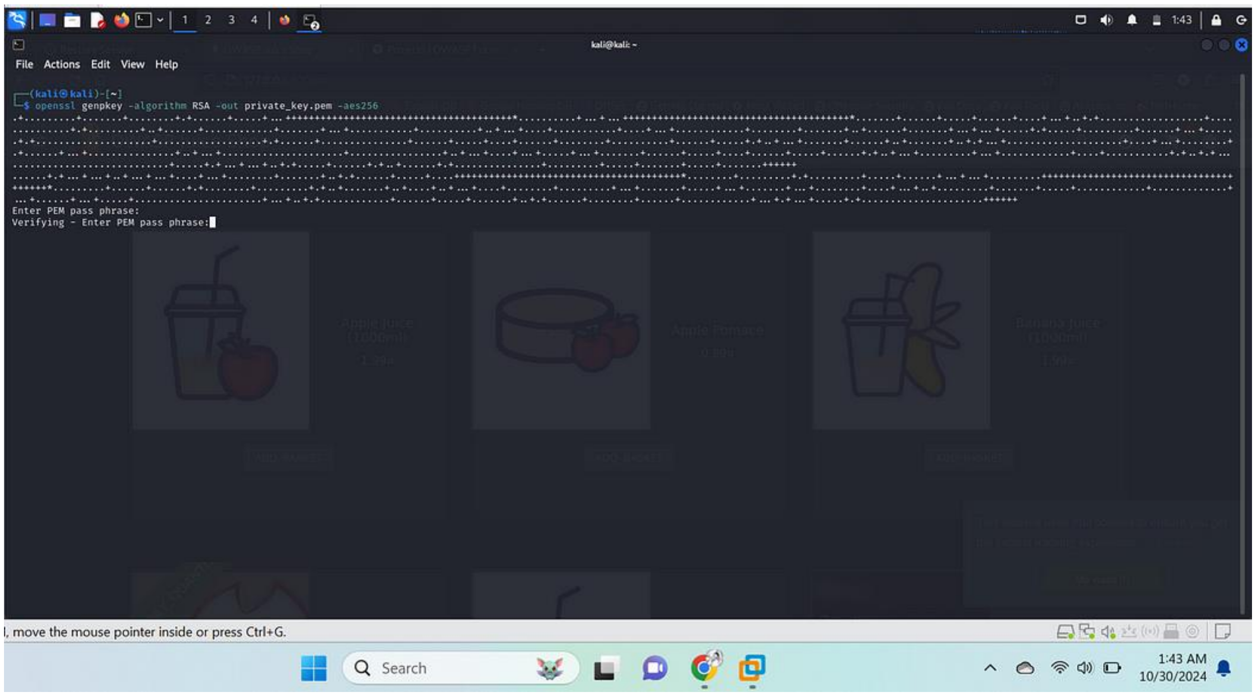
Steps for Asymmetric Encryption Using OpenSSL

We'll go through generating a key pair, encrypting a message with the public key, and decrypting it with the private key.

Step 1: Generate a Private Key

1. **Generate an RSA Private Key**:
2. Run the following command to create a 2048-bit RSA private key (saved as private_key.pem):

openssl genpkey -algorithm RSA -out private_key.pem -aes256



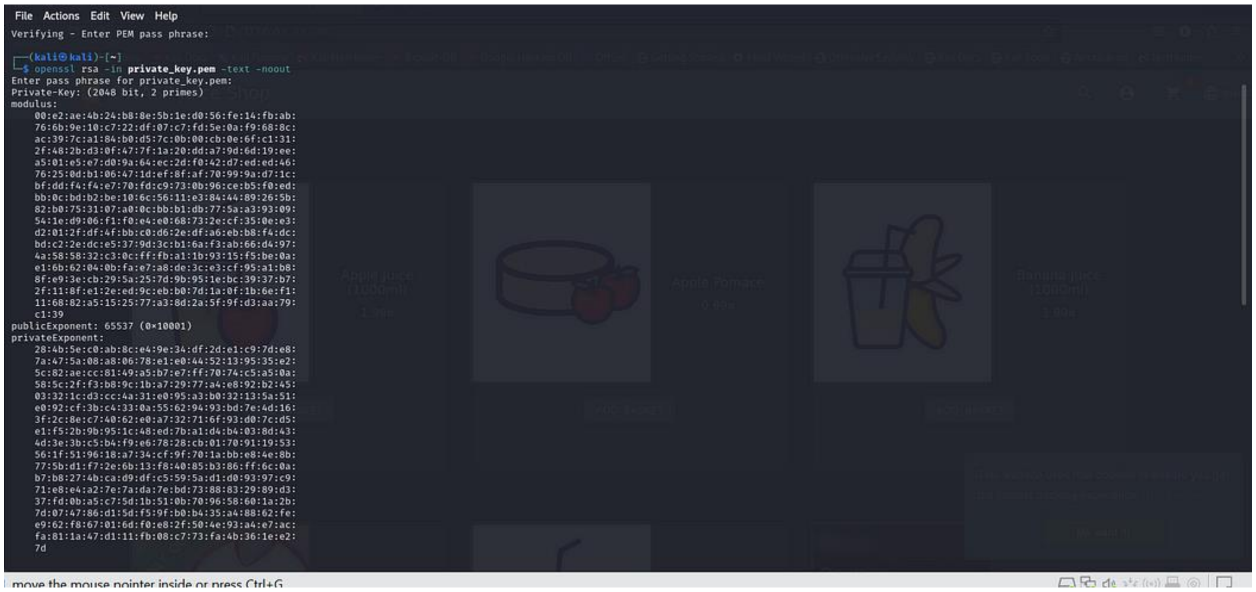
```
┌──(kali㉿kali)-[~]
└─$ openssl genpkey -algorithm RSA -out private_key.pem -aes256
[illegible]
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

You'll be prompted to enter a passphrase to protect the private key file.

Check the Key (Optional):

To view the private key details, you can use:

openssl rsa -in private_key.pem -text -noout



```
Verifying - Enter PEM pass phrase:

┌──(kali㉿kali)-[~]
└─$ openssl rsa -in private_key.pem -text -noout
Enter pass phrase for private_key.pem:
Private-Key: (2048 bit, 2 primes)
modulus:
    00:e2:ae:4b:24:b8:8e:5b:1e:d0:56:fe:14:fb:ab:
    76:6b:9e:10:c7:22:df:07:c7:fd:5e:0a:f9:68:8c:
    ac:39:7c:a1:84:b0:d5:7c:0b:00:cb:0e:6f:c1:31:
    2f:48:2b:d3:0f:47:7f:1a:20:dd:a7:9d:6d:19:ee:
    a5:01:e5:e7:d0:9a:64:ec:2d:f0:42:d7:ed:ed:46:
    76:25:0d:b1:06:47:1d:ef:8f:af:70:99:9a:d7:1c:
    bf:dd:f4:f4:e7:70:fd:c9:73:0b:96:ce:b5:f0:ed:
    bb:0c:bd:b2:be:10:6c:56:11:e3:84:44:89:26:5b:
    82:b0:75:31:07:a0:0c:bb:b1:db:77:5a:a3:93:09:
    54:1e:d9:06:f1:f0:e4:e0:68:73:2e:cf:35:0e:e3:
    d2:01:2f:df:4f:bb:c0:d6:2e:df:a6:eb:b8:f4:dc:
    bd:c2:2e:dc:e5:37:9d:3c:b1:6a:f3:ab:66:d4:97:
    4a:58:58:32:c3:0c:ff:fb:a1:1b:93:15:f5:be:0a:
    e1:6b:62:04:0b:fa:e7:a8:de:3c:e3:cf:95:a1:b8:
    8f:e9:3e:cb:29:5a:25:7d:9b:95:1e:bc:39:37:b7:
    2f:11:8f:e1:2e:ed:9c:eb:b0:7d:1a:0f:1b:6e:f1:
    11:68:82:a5:15:25:77:a3:8d:2a:5f:9f:d3:aa:79:
    c1:39
publicExponent: 65537 (0×10001)
privateExponent:
    28:4b:5e:c0:ab:8c:e4:9e:34:df:2d:e1:c9:7d:e8:
    7a:47:5a:08:a8:06:78:e1:e0:44:52:13:95:35:e2:
    5c:82:ae:cc:81:49:a5:b7:e7:ff:70:74:c5:a5:0a:
    58:5c:2f:f3:b8:9c:1b:a7:29:77:a4:e8:92:b2:45:
    03:32:1c:d3:cc:4a:31:e0:95:a3:b0:32:13:5a:51:
    e0:92:cf:3b:c4:33:0a:55:62:94:93:bd:7e:4d:16:
    3f:2c:8e:c7:40:62:e0:a7:32:71:6f:93:d0:7c:d5:
    e1:f5:2b:9b:95:1c:48:ed:7b:a1:d4:b4:03:8d:43:
    4d:3e:3b:c5:b4:f9:e6:78:28:cb:01:70:91:19:53:
    56:1f:51:96:18:a7:34:cf:9f:70:1a:bb:e8:4e:8b:
    77:5b:d1:f7:2e:6b:13:f8:40:85:b3:86:ff:6c:0a:
    b7:b8:27:4b:ca:d9:df:c5:59:5a:d1:d0:93:97:c9:
    71:e8:e4:a2:7e:7a:da:7e:bd:73:88:83:29:89:d3:
    37:fd:0b:a5:c7:5d:1b:51:0b:70:96:58:60:1a:2b:
    7d:07:47:86:d1:5d:f5:9f:b0:b4:35:a4:88:62:fe:
    e9:62:f8:67:01:6d:f0:e8:2f:50:4e:93:a4:e7:ac:
    fa:81:1a:47:d1:11:fb:08:c7:73:fa:4b:36:1e:e2:
    7d
```

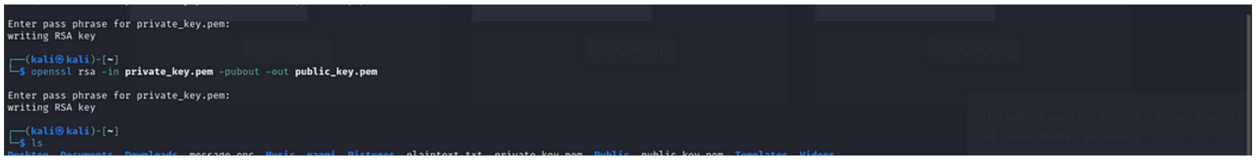
Step 2: Extract the Public Key from the Private Key

1. **Generate a Public Key**:

Using the private key, extract the public key

This will create a file named public_key.pem containing the public key.

openssl rsa -in private_key.pem -pubout -out public_key.pem

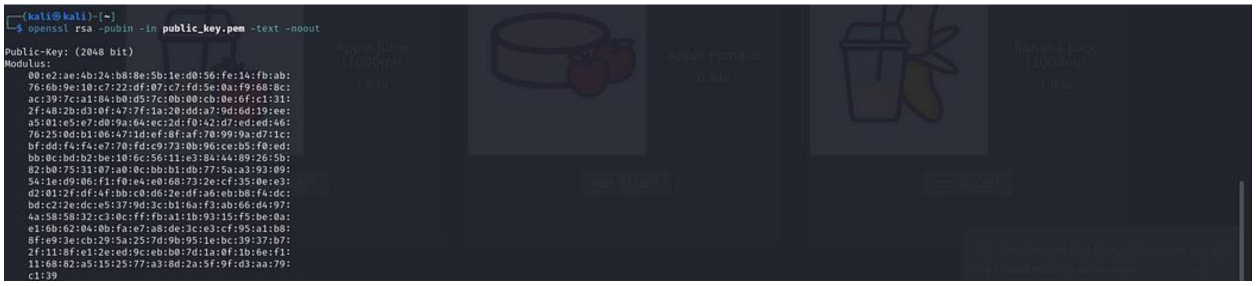


```
Enter pass phrase for private_key.pem:
writing RSA key

┌──(kali㉿kali)-[~]
└─$ openssl rsa -in private_key.pem -pubout -out public_key.pem

Enter pass phrase for private_key.pem:
writing RSA key

┌──(kali㉿kali)-[~]
└─$ ls
```

2. Check the Public Key (Optional):

To view the public key details, you can use:

openssl rsa -pubin -in public_key.pem -text -noout



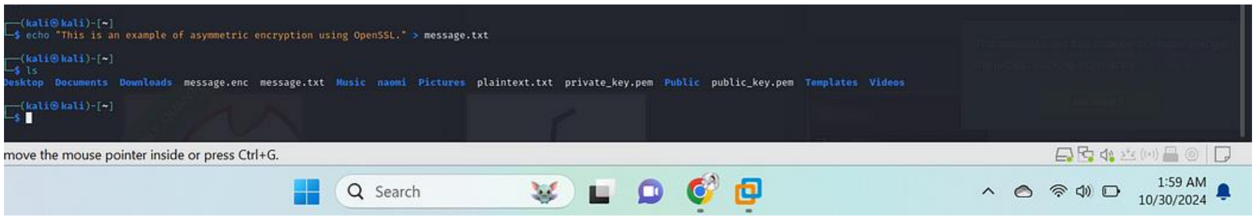
```
┌──(kali㉿kali)-[~]
└─$ openssl rsa -pubin -in public_key.pem -text -noout

Public-Key: (2048 bit)
Modulus:
    00:e2:ae:4b:24:b8:8e:5b:1e:d0:56:fe:14:fb:ab:
    76:6b:9e:10:c7:22:df:07:c7:fd:5e:0a:f9:68:8c:
    ac:39:7c:a1:84:b0:d5:7c:0b:00:cb:0e:6f:c1:31:
    2f:48:2b:d3:0f:47:7f:1a:20:dd:a7:9d:6d:19:ee:
    a5:01:e5:e7:d0:9a:64:ec:2d:f0:42:d7:ed:ed:46:
    76:25:0d:b1:06:47:1d:ef:8f:af:70:99:9a:d7:1c:
    bf:dd:f4:f4:e7:70:fd:c9:73:0b:96:ce:b5:f0:ed:
    bb:0c:bd:b2:be:10:6c:56:11:e3:84:44:89:26:5b:
    82:b0:75:31:07:a0:0c:bb:b1:db:77:5a:a3:93:09:
    54:1e:d9:06:f1:f0:e4:e0:68:73:2e:cf:35:0e:e3:
    d2:01:2f:df:4f:bb:c0:d6:2e:df:a6:eb:b8:f4:dc:
    bd:c2:2e:dc:e5:37:9d:3c:b1:6a:f3:ab:66:d4:97:
    4a:58:58:32:c3:0c:ff:fb:a1:1b:93:15:f5:be:0a:
    e1:6b:62:04:0b:fa:e7:a8:de:3c:e3:cf:95:a1:b8:
    8f:e9:3e:cb:29:5a:25:7d:9b:95:1e:bc:39:37:b7:
    2f:11:8f:e1:2e:ed:9c:eb:b0:7d:1a:0f:1b:6e:f1:
    11:68:82:a5:15:25:77:a3:8d:2a:5f:9f:d3:aa:79:
    c1:39
```

Step 3: Encrypt a Message Using the Public Key

1. **Create a Plaintext File**:

Let’s create a plaintext file to encrypt:

echo "This is an example of asymmetric encryption using OpenSSL." > message.txt



```
┌──(kali㉿kali)-[~]
└─$ echo "This is an example of asymmetric encryption using OpenSSL." > message.txt

┌──(kali㉿kali)-[~]
└─$ ls
Desktop  Documents  Downloads  message.enc  message.txt  Music  naomi  Pictures  plaintext.txt  private_key.pem  Public  public_key.pem  Templates  Videos

┌──(kali㉿kali)-[~]
└─$ █
```

move the mouse pointer inside or press Ctrl+G.

Search

1:59 AM
10/30/2024

2. Encrypt the Message:

Use the public key to encrypt the message

openssl pkeyutl -encrypt -pubin -inkey public_key.pem -in message.txt -out encrypted_message.bin

Explanation:

- **openssl**: Command-line tool for OpenSSL, used for cryptographic operations.
- **pkeyutl**: Utility for performing operations using public/private keys.
- **-encrypt**: Indicates that you are encrypting data.
- **-pubin**: Specifies that the input key is a public key.

- **-inkey public_key.pem**: The file containing the public key used for encryption.
- **-in message.txt**: The plaintext file containing the message you want to encrypt.
- **-out encrypted_message.bin**: The file where the encrypted message will be saved.

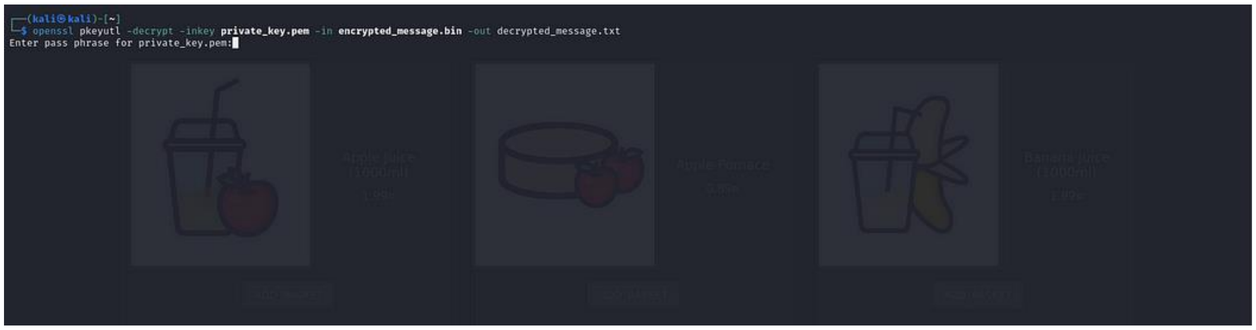
Step 4: Decrypt the Message Using the Private Key

1. **Decrypt the Encrypted Message**:

Use the private key to decrypt the message:

openssl pkeyutl -decrypt -inkey private_key.pem -in encrypted_message.bin -out decrypted_message.txt

it will prompt for a password

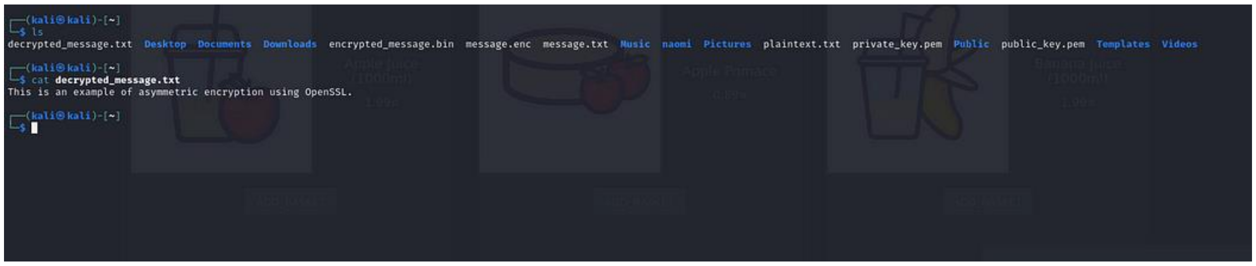


```
┌──(kali㉿kali)-[~]
└─$ openssl pkeyutl -decrypt -inkey private_key.pem -in encrypted_message.bin -out decrypted_message.txt
Enter pass phrase for private_key.pem:
```

2.View the Decrypted Message:

- To check if decryption was successful, view the content of decrypted_message

cat decrypted_message.txt



```
┌──(kali㉿kali)-[~]
└─$ ls
decrypted_message.txt  Desktop  Documents  Downloads  encrypted_message.bin  message.enc  message.txt  Music  naomi  Pictures  plaintext.txt  private_key.pem  Public  public_key.pem  Templates  Videos

┌──(kali㉿kali)-[~]
└─$ cat decrypted_message.txt
This is an example of asymmetric encryption using OpenSSL.

┌──(kali㉿kali)-[~]
└─$ 
```

- Ensure that you have generated the private_key.pem and public_key.pem files correctly and that they are accessible in the current directory.
- The data encrypted with RSA has limitations on size. For larger messages, consider using hybrid encryption (encrypting a message with a symmetric key and then encrypting that key with RSA).

Scenario: Secure Communication Between John and Jane

Here's the revised scenario using John and Jane for secure communication:

Scenario: Secure Communication Between John and Jane

Background
John and Jane want to communicate securely over the internet to exchange sensitive information, such as personal data or confidential business information. To achieve this, they will use asymmetric encryption to ensure that only the intended recipient can read the messages.

Step-by-Step Process

1.Key Generation:
John and Jane each generate a pair of cryptographic keys:
John generates his own public and private keys. He keeps his private key secure and shares his public key with Jane.
Jane does the same, generating her own public and private keys.

2. Public Key Exchange:
John sends his public key to Jane, and Jane sends her public key to John. This exchange can occur over an insecure channel since the public keys do not need to be kept secret.

3. Message Encryption:
When John wants to send a secure message to Jane, he follows these steps:
John writes his message(e.g., “Hello Jane, this is a secret message.”).
He encrypts the message using Jane’s public key with the following OpenSSL command:

```
openssl pkeyutl -encrypt -pubin -inkey jane_public_key.pem -in message.txt -out encrypted_message.bin
```

The encrypted message is now secure, as it can only be decrypted with Jane’s private key.

4. Sending the Encrypted Message:
John sends the encrypted_message.bin file to Jane through an email or messaging service.

5. Message Decryption:
Upon receiving the encrypted message, Jane uses her private key to decrypt it:

```
openssl pkeyutl -decrypt -inkey jane_private_key.pem -in encrypted_message.bin -out
decrypted_message.txt
```

Jane can now read the original message, which confirms that it came from John and remains confidential.

6. Acknowledgment and Response:
Jane can respond securely by encrypting her message with John’s public key, following the same steps as above. This ensures that only John can decrypt and read Jane’s response.

7. Securing the Connection with SSL/TLS:
For ongoing secure communication, John and Jane can also establish an SSL/TLS connection:
They can use an SSL/TLS protocol to create a secure channel over the internet.
This involves obtaining a digital certificate from a trusted Certificate Authority (CA) and using it to encrypt their communication session.
With this setup, any data sent during the session is encrypted, providing confidentiality and integrity.

Conclusion

Through the use of asymmetric encryption and SSL/TLS protocols, John and Jane ensure their communication remains private and secure from eavesdroppers. By leveraging public and private key pairs, they can confidently exchange sensitive information, knowing that only the intended recipient can access the messages. This scenario highlights the importance of cryptographic techniques in securing communications in today’s digital landscape.