LAB-9 : OPENSSL

Welcome to the Introduction to OpenSSL Lab, OpenSSL is a powerful open-source software library that provides cryptographic functions and tools to secure communications over computer networks. It is widely used for implementing Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols, which are essential for ensuring privacy, data integrity, and authentication in internet communications.

For this lab we will be covering the following topics:

- ➔ Data integrity
- ➔ Encryption Decryption
- ➔ Digital Signature
- ➔ Digital Certificate
- ➔ Certification Authority

- **DATA INTEGRITY :**

  OpenSSL includes commands for generating and verifying checksums (such as MD5, SHA-1, SHA-256) to ensure data integrity. Checksums are used to detect and prevent data corruption during transmission or storage.

  Command to calculate the checksum :

  - ➔ `openssl sha1 <file_path>` (this calculate the sha1 checksum)
  - ➔ Similarly you can calculate sha256 checksum by `openssl sha256 <file_path>`

MINI TASK: To understand data integrity using checksums with OpenSSL, start by creating a text file with some content and calculate its checksum. The checksum serves as a unique identifier for the file's content. Make changes to the file, like adding or removing text, and then recalculate the checksum.

- **ENCRYPTION AND DECRYPTION :**

  OpenSSL offers a comprehensive suite of cryptographic algorithms designed to encrypt and decrypt data securely, ensuring the protection of sensitive information. One of the key strengths of OpenSSL lies in its support for asymmetric encryption using RSA keys, which are fundamental in modern cryptographic operations.

1. **Symmetric encryption** is a cryptographic technique where the same key is used for both encryption and decryption of data.
   STEP-1 : Generate a random key
   COMMAND: `openssl rand -hex 32 > key.pem`
   It generates a random key and saves to the file key.pem

   STEP-2 : To encrypt the file
   COMMAND: `openssl aes-256-cbc -in input.txt -out encrypted.txt -e -kfile key.pem -pbkdf2`
   In this command:
   - in input.txt specifies the input file.
   - out encrypted.txt specifies the output file.
   - e indicates encryption mode
   - kfile key.pem specifies the key obtained from key.pem
   - pbkdf2: Specifies the use of the PBKDF2 key derivation function, which is considered more secure.

   STEP-3 : To decrypt the file
   COMMAND: `openssl aes-256-cbc -in encrypted.txt -out decrypted.txt -d -kfile key.pem -pbkdf2`
   In this command:
   - in encrypted.txt specifies the input file.
   - out decrypted.txt specifies the output file.
   - d indicates decryption mode.
   - kfile key.pem specifies the key obtained from key.pem
   - pbkdf2: Specifies the use of the PBKDF2 key derivation function, which is considered more secure.

2. **Asymmetric encryption** involves the use of a pair of keys: a public key and a private key. The public key is used for encryption, while the private key is used for decryption. This asymmetric nature allows users to securely transmit data without the need to share the private key, enhancing data confidentiality and security.

   In this lab, we will leverage OpenSSL's capabilities to generate RSA key pairs, encrypt data using the public key, and decrypt it using the corresponding private key. This

approach ensures that sensitive information remains protected during transmission and can only be decrypted by authorized recipients possessing the private key.

STEP-1 : To generate RSA private key

COMMAND: openssl genrsa -out private_key.pem 2048

In this command:

- openssl genrsa generates an RSA private key.
- out private_key.pem specifies the output file name (private_key.pem in this example) where the private key will be stored.
- 2048 specifies the key size (in bits), in this case, a 2048-bit RSA key.

After running this command, you will have a private key stored in the private_key.pem file in PEM format.

STEP-2 : To generate RSA public key

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

In this command:

- openssl rsa is used to process RSA keys.
- in private_key.pem specifies the input file containing the private key.
- pubout indicates that the output should be the public key.
- out public_key.pem specifies the output file name (public_key.pem in this example) where the public key will be stored.

After running the second command, you will have the public key stored in the public_key.pem file in PEM format.

STEP-3 : ENCRYPTING A FILE

COMMAND: openssl pkeyutl -encrypt -inkey public_key.pem -pubin -in input.txt -out encrypted.txt

In this command:

- -encrypt: Specifies that we want to perform encryption using RSA.
- -inkey public_key.pem: Specifies the input file containing the public key (public_key.pem).
- -pubin: Indicates that the input file specified with -inkey is a public key file. This option is necessary to correctly interpret the input file as a public key.
- -in input.txt: Specifies the input file (input.txt) containing the data that you want to encrypt.
- -out encrypted.txt: Specifies the output file (encrypted.txt) where the encrypted data will be written.

STEP-4 : DECRYPTING A FILE

COMMAND: openssl pkeyutl -decrypt -inkey private_key.pem -in encrypted.txt -out decrypted.txt

In this command:

- -decrypt: This flag specifies that the operation to be performed is decryption.
- -inkey private_key.pem: This option specifies the private key file (private_key.pem) to be used for decryption.
- -in encrypted.txt specifies the input file containing the encrypted data.
- -out decrypted.txt specifies the output file where the decrypted data will be saved.

MINI TASK :

1.) Begin by generating an RSA private key with a key size of 2048 bits using OpenSSL. Then, use the private key to generate the corresponding RSA public key. Next, encrypt a text file named "input.txt" using the public key and save the encrypted data in a file named "encrypted.txt" Finally, decrypt the "encrypted.txt" file using the private key and save the decrypted data in a file named "decrypted.txt" This task will provide hands-on experience with OpenSSL's RSA key generation, asymmetric encryption, and decryption capabilities.

2.) Explore symmetric key encryption and consider its limitations compared to asymmetric encryption for data security.

- **GENERATING DIGITAL SIGNATURES**

  A digital signature is a cryptographic technique used to ensure the authenticity, integrity, and non-repudiation of digital messages or documents. It involves using a private key to create a unique digital signature for a message or document, which can then be verified using the corresponding public key.

  Follow the below steps to generate a digital signature

STEP-1: The first step is to generate an RSA key pair, consisting of a public key and a private key. The public key is used for verifying signatures, while the private key is used for generating signatures. Use the above steps to generate the same.

STEP-2: After generating the key pair, you can use the private key to generate a digital signature for a specific input file. The openssl sha256 -sign command computes the SHA-256 hash of the encrypted file and signs it using the private key, producing the signature file.

COMMAND: openssl sha256 -sign private_key.pem -out signature.sig encrypted.txt

STEP-3: To verify the authenticity of the signature and ensure data integrity, you can use the public key and the signature file. The openssl sha256 -verify command verifies the signature against the encrypted file using the public key.

COMMAND: openssl sha256 -verify public_key.pem -signature signature.sig encrypted.txt

MINI TASK: Begin by generating an RSA key pair (public and private keys) using OpenSSL. Next, assume that you need to send data to your client, so you encrypt the data using your client's public key and save it in a file called encrypted.txt and then generate a signature for it and send the encrypted file as well as the signature to the client. The client can now digitally verify the signature using your public key and if it's valid then it can decrypt it using his/her private key.

- **GENERATING DIGITAL CERTIFICATE :**

  A digital certificate, also known as a public key certificate, is a digital document that binds a public key to an entity (such as an individual, organization, or website) and provides information about the entity's identity.
  It contains the entity's public key, information about the entity (e.g., name, email address), the digital signature of a Certificate Authority (CA) to verify its authenticity, and the certificate's expiration date.Digital certificates are used in public key infrastructure (PKI) systems to establish trust and enable secure communication over insecure networks like the internet.

Follow the steps to generate a digital certificate:
STEP-1: Generate the RSA keys using the above method.

STEP-2: Generate a Self-Signed Certificate:

COMMAND: openssl req -x509 -key private_key.pem -out certificate.pem -days 365

This command generates a self-signed certificate using the private key specified by -key private_key.pem, and it's valid for 365 days (-days 365). The resulting certificate is saved to a file named certificate.pem.

STEP-3: View Certificate Information:

COMMAND: openssl x509 -in certificate.pem -noout -text

This command displays detailed information about a certificate (-in certificate.pem) without printing the encoded certificate itself (-noout -text).

Notice that the certificate contains the public key as well. You can extract this public key using the command `openssl x509 -pubkey -noout -in certificate.pem > public_key.pem`

MINI TASK: Begin by generating an RSA key pair (public and private keys) using OpenSSL. Then, generate a self-signed certificate using the private key, valid for 365 days, and save it as "certificate.pem" Finally, view detailed information about the generated certificate using OpenSSL's command to understand its contents and attributes.
NOTE that while generating the CSR or certificate you may be prompted with details like country,org etc… You can enter any value.

- **CERTIFICATE AUTHORITY**

A Certificate Authority (CA) is a trusted entity responsible for issuing digital certificates that validate the ownership of public keys used in secure communication.

Certificate Authority Hierarchy:

- ❖ Root CA: At the top of the hierarchy is the Root CA, which is a **self-signed certificate** that represents the highest level of trust. Root CAs are used to sign intermediate CAs or issue end-entity certificates directly.
- ❖ Intermediate CA: Intermediate CAs are subordinate to Root CAs and are used to issue certificates on behalf of the Root CA. They help manage the workload of the Root CA and can be used to create a hierarchical trust model.
- ❖ End-entity Certificates: These are the certificates issued to individual users, servers, or devices. They are signed either by the Root CA or an Intermediate CA, depending on the certificate chain.

STEP-1: Generate the RSA keys using the above method.

STEP-2: Generate a Certificate Signing Request (CSR):

COMMAND: openssl req -new -key private_key.pem -out csr.pem

This command generates a CSR using the private key specified by -key private_key.pem and saves the CSR to a file named csr.pem.

STEP-3: Requesting for the digital certificate from the upper hierarchical level.

```
openssl x509 -req -in <csr_file> -CA <ca_cert_file> -CAkey <ca_key_file> -CAcreateserial
-out <issued_cert_file> -days <validity_days>
```

- req: Indicates that a certificate signing request (CSR) is being processed.
- in <csr_file>: Specifies the input CSR file (generated by the intermediate CA requesting from Root CA or generated by the entity requesting from intermediate CA).
- CA <ca_cert_file>: Specifies the CA certificate file (e.g., Root CA or Intermediate CA certificate) that will sign the certificate.
- CAkey <ca_key_file>: Specifies the private key file of the CA certificate specified with -CA.
- CAcreateserial: Creates a serial number file if it does not exist, which is used to uniquely identify the issued certificate.
- out <issued_cert_file>: Specifies the output file where the issued certificate will be saved.
- days <validity_days>: Sets the validity period of the issued certificate in days.

MINI TASK: Firstly create a ROOT CA (you can create a self signed certificate for this). Then create 1 intermediate CA which is signed by ROOT CA and then finally create an entity certificate which will be signed by the intermediate CA. (Generate all necessary keys you require).

EXAMPLE: Putting Everything Together (Real World Scenario):

As a client, you possess a pair of asymmetric keys, with your public key accessible to all. When you initiate communication with a server, you request specific data. Upon receiving your request, the server processes the data and encrypts it using your public key. Additionally, the server utilizes its own private key to digitally sign the encrypted data. To ensure trustworthiness, the server obtains a digital certificate from a trusted third-party source.

The server then transmits the encrypted data, along with the digital signature and digital certificate, to you. Upon receiving the data, you first verify the digital signature using the server's digital certificate. This involves confirming the digital signature using the public key (of the server) obtained from the digital certificate. Successful validation indicates that the data has been signed by the server and hasn't been altered during transmission.

Next, you decrypt the encrypted data using your private key. This ensures that only you, possessing the corresponding private key, can access the original information.