**OpenSSL [ genpkey, pkey, pkeyutl, dgst, req ]**

**Introduction**

OpenSSL is a powerful toolkit widely used for implementing cryptographic functions and securing communications. It provides command-line tools and libraries for cryptographic operations such as generating keys, certificates, encrypting/decrypting data, and creating secure network connections

**Generating Private Key**

The genpkey command in OpenSSL is used to generate private keys for various cryptographic algorithms. It's a versatile command that supports multiple algorithms such as RSA, EC, DH, and Ed25519.

*openssl genpkey -algorithm <algorithm> -out <output\_file\_name> -pkeyopt <key\_size>*

*(or)*

*openssl genpkey -algorithm RSA -out rsa\_private\_key.pem -pkeyopt rsa\_keygen\_bits:2048*

*(or)*

*openssl genpkey -algorithm EC -out ec\_private\_key.pem -pkeyopt ec\_paramgen\_curve:P-256*

*(or)*

*openssl genpkey -algorithm DH -out dh\_private\_key.pem*

**Note**

To generate private key for DH algorithm we need to first create dhparam file which can be done using

*openssl dhparam -out dhparams.pem 2048*

**Generating Public Key**

The pkey command in OpenSSL is used for manipulating and processing public and private keys. It allows you to perform various cryptographic operations with keys, such as generating, converting, or performing operations like encryption, signing, and verifying signatures.

*openssl pkey -in <private\_key> -pubout -out <output\_public\_key\_file\_name>*

*(or)*

*openssl pkey -in rsa\_private\_key.pem -pubout -out rsa\_public\_key.pem*

**Encrypting Data**

The openssl pkeyutl command is used to perform cryptographic operations such as encryption, decryption, signing, and verification using public and private keys. It supports a variety of algorithms, including RSA, EC (Elliptic Curve), and others. Encryption is done using the public key of the receiver.

*openssl pkeyutl -encrypt -in <file\_to\_encrypt> -pubin -inkey <input\_public\_key> -out <output\_file\_name>*

*(or)*

*openssl pkeyutl -encrypt -in message.txt -pubin -inkey rsa\_public\_key.pem -out encrypted\_message.bin*

**Note**

-pubin option indicates encryption using the public key.

**Decrypting Data**

To decrypt the data the pkeyutl is used with the following options. Decryption is done using the private key of recipient.

*openssl pkeyutl -decrypt -in <file\_to\_decrypt> -inkey <input\_private\_key> -out <output\_file\_name>*

*(or)*

*openssl pkeyutl -decrypt -in encrypted\_message.bin -inkey rsa\_private\_key.pem -out decrypted\_message.txt*

**Signing a Software (or) a file**

* Generate a private and public key pair to use for signing and verification. The private key is kept secure, and the public key is shared with others.
* Calculate a cryptographic hash of the software or file to ensure data integrity. The hash represents the unique contents of the file.
* Use the private key to sign the hash. This creates a digital signature that can be used to prove the authenticity of the file.
* Share the software or file along with the digital signature. These are necessary for verifying the file's authenticity.

*openssl dgst -<hashing\_algorithm> -sign <private\_key> -out <output\_signature\_file\_name> <file\_to\_sign>*

*(or)*

*openssl dgst -sha256 -sign rsa\_private\_key.pem -out signature.bin software.zip*

**Verifying a software (or) a file**

* Obtain the file, its corresponding digital signature, and the sender's public key from official documentation or in the form of certificate.
* Compute a cryptographic hash of the received file using the same algorithm as used during signing.
* Use the sender's public key to decrypt the digital signature.
* Extract the hash value contained within the decrypted signature.
* Compare the hash value from the signature with the hash value computed from the file.
* If the two hash values match, the file is verified as authentic and untampered.
* If the hash values do not match, it indicates either the file has been altered, or the signature does not correspond to the file.

*openssl dgst -<hashing\_algorithm> -verify <public\_key> -signature <signature\_of\_the\_file> <file\_to\_verify>*

**Generating Certificates**

A digital certificate typically contains the following information:

* **Subject:** The entity to which the certificate is issued (e.g., a website or organization).
* **Issuer:** The Certificate Authority (CA) that issued the certificate.
* **Public Key:** The public key associated with the server (this is used in asymmetric encryption).
* **Serial Number:** A unique identifier for the certificate.
* **Validity Period:** The start and end dates when the certificate is valid.
* **Signature Algorithm:** The algorithm used by the CA to sign the certificate.
* **Signature:** The digital signature of the CA that verifies the authenticity of the certificate.
* **Version:** The version of the certificate (e.g., X.509 v3).
* **Extensions:** Additional information or functionality (such as key usage, domain names, etc.).
* First, generate a private key. This key is essential for signing the certificate or encrypting data. The private key is kept securely on the server, and it must never be shared.

*openssl genpkey -algorithm <algorithm> -out <output\_file\_name> -pkeyopt <key\_size>*

*(or)*

*openssl genpkey -algorithm RSA -out rsa\_private\_key.pem -pkeyopt rsa\_keygen\_bits:2048*

* Create a Certificate Signing Request (CSR) using the private key. This request includes the public key that pairs with the private key, along with identifying information about the entity requesting the certificate, such as the domain name, organization name, location, and contact details.

*openssl req -new -key <private\_key> -out <file\_output>*

*(or)*

*openssl req -new -key rsa\_private\_key.pem -out certificate.csr*

* If you choose to create a self-signed certificate, you will sign the CSR with the private key. This creates a certificate that can be used for encrypting communication and establishing the identity of the server. A self-signed certificate doesn't require a third-party authority for verification.

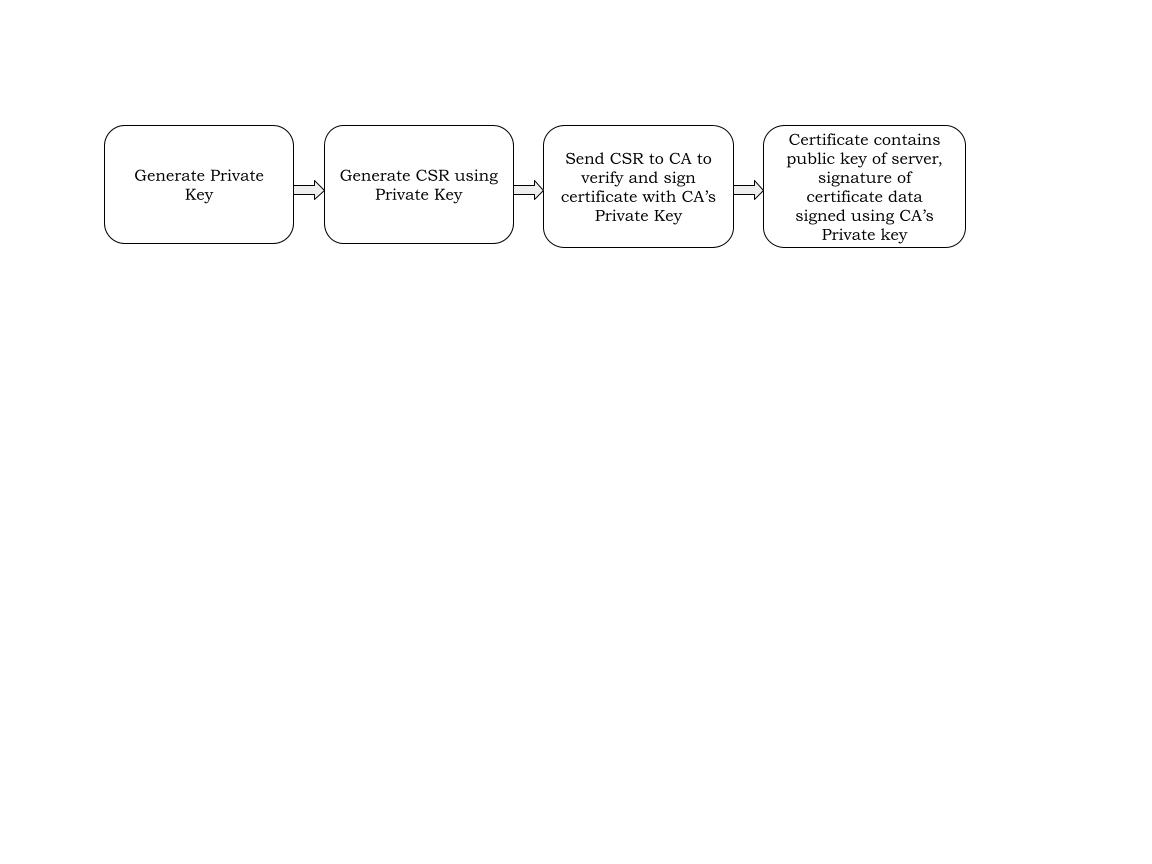
*openssl <certificate\_standard> -req -days <validity\_of\_certificate> -in <csr\_certificate> -signkey <private\_key> -out <output\_file>*

*(or)*

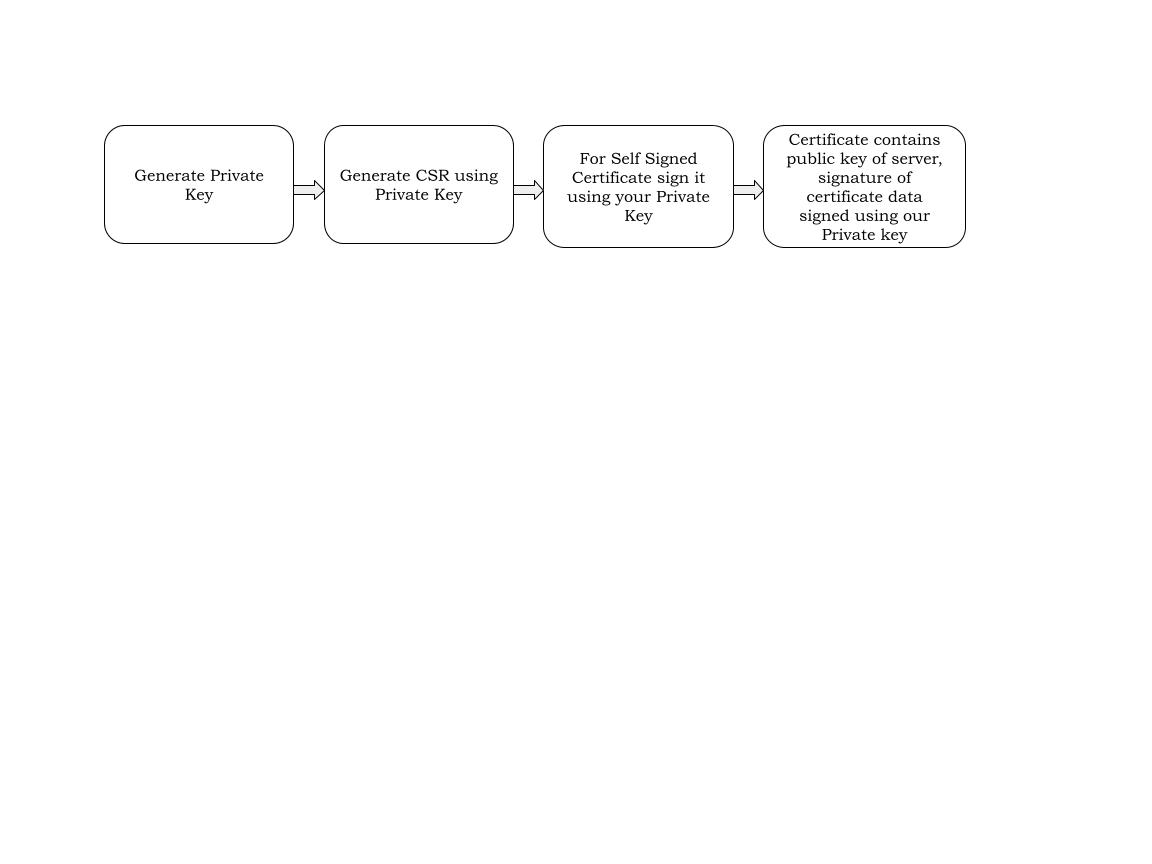
*openssl x509 -req -days 365 -in certificate.csr -signkey private\_key.pem -out certificate.crt*

* For a CA-signed certificate, you submit the CSR to a trusted Certificate Authority (CA). The CA verifies the details of the CSR, such as domain ownership, organization information, and whether the requestor is legitimate.
* After the CA verifies the information in the CSR, the CA signs the certificate using its own private key. This issued certificate includes the public key of the server and the CA’s digital signature, which validates the authenticity of the certificate.
* Once the certificate is issued (either self-signed or CA-signed), install the certificate along with the private key on the server. The certificate enables encrypted communication between the server and its clients.
* A self-signed certificate will not be trusted by clients unless they manually add it to their trusted certificate store. In contrast, a CA-signed certificate is automatically trusted by web browsers and other clients, as the CA is already included in their list of trusted certificate authorities.

**CA Signed Certificate Generation**

****

**Self Signed Certificate Verification**

****