**Create CSR and KEY**

**openssl req -new -newkey rsa:2048 -nodes -keyout somnath.com.key -out somnath.com.csr -subj "/C=in/ST=jharkhand/L=pakur/O=somnath inc./OU=it department/CN=somnath.com"**

**Create Self-Signed CRT (Certificate)**

**openssl x509 -req -days 365 -in somnath.com.csr -signkey somnath.com.key -out somnath.com.crt**

**Create PEM file from CRT**

**openssl x509 -in somnath.com.crt -out somnath.com.pem -outform PEM**

**Create PFX File from PEM and KEY**

**openssl pkcs12 -export -out somnath.com.pfx -inkey somnath.com.key -in somnath.com.pem**

**Create Public KEy**

**openssl x509 -in somnath.com.pem -pubkey -noout > somnath.com.publickey.pem**

**TroubleShhot ( Check private key, If opening then valid else create again)**

**openssl rsa -in somnath.com.key -noout -text**

**To Create New Key**

**openssl genrsa -out somnath.com.key 2048**

**CSR (Certificate Signing Request) and key:**

CSR stands for "Certificate Signing Request" and is a message that contains information about the entity requesting a digital certificate, including the public key that will be used for encryption or digital signatures. **A CSR is required to create a digital certificate.**

A key, on the other hand, is a piece of data used in encryption and decryption algorithms. In the context of digital certificates, **a key refers to a private key**, which is kept secret by the entity to whom the certificate belongs, and a public key, which is freely distributed to anyone who wants to communicate securely with that entity.

Create a CSR (Certificate Signing Request) and key in a single command using OpenSSL. Here is the command:

\*Command is case-sensitive

**openssl req**

**-new**

**-newkey rsa:2048**

**-nodes**

**-keyout SOMNATH.COM.KEY**

**-out SOMNATH.COM.CSR**

**-subj "/C=IN/ST=JHARKHAND/L=PAKUR/O=SOMNATH INC./OU=IT DEPARTMENT/CN=SOMNATH.COM"**

Let's break down this command:

**openssl req**: This tells OpenSSL to create a CSR.

-**new**: This specifies that a new CSR is being created.

-**newkey rsa:2048**: This generates a new RSA private key with a key size of 2048 bits.

-**nodes**: This tells OpenSSL not to encrypt the private key with a passphrase, which is useful for automated processes but may be less secure.

-**keyout example.com.key**: This specifies the filename and location to save the RSA private key.

-**out example.com.csr**: This specifies the filename and location to save the CSR.

Here's what each field in the example **-subj** string means:

**/C=IN**: The two-letter country code for the United States.

**/ST=Jharkhand**: The state or province of California.

**/L=Pakur**: The city or locality of San Francisco.

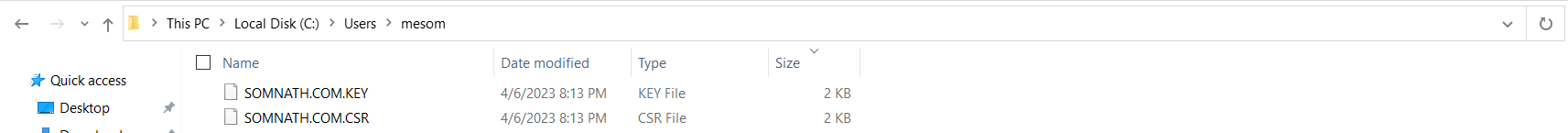
**/O=SOMNATH Inc**.: The legal name of the organization requesting the certificate.

**/OU=IT Department**: An optional organizational unit that further identifies a specific department within the organization.

**/CN=SOMNATH.com**: The common name (domain name) for which the certificate is requested.

**After running the command in command prompt it will generate CSR and KEY file.**





**To create a certificate from the CSR and key, you will need to submit the CSR to a Certificate Authority (CA) that can issue the certificate. The CA will verify your identity and domain ownership and then issue the certificate.**

Here are the general steps involved in getting a certificate from a CA:

1.Submit the CSR: Log in to the CA's website or portal, navigate to the certificate request page, and paste in the contents of your CSR. You may need to provide additional information, such as contact details or proof of identity.

2. Verify your identity: Depending on the type of certificate you are requesting; the CA may require additional information or documentation to verify your identity and domain ownership. For example, they may ask for a copy of your business license or articles of incorporation.

3. Wait for validation: Once you have submitted your CSR and any required documentation, you will need to wait for the CA to validate your information. This can take anywhere from a few hours to several days, depending on the CA's policies and procedures.

4.Receive and install the certificate: Once the CA has validated your information, they will issue the certificate and provide you with a certificate file in either **PEM, PFX or another format**. You can then install the certificate on your web server and configure it to use the private key you generated earlier.

Note that the specific steps and requirements for getting a certificate may vary depending on the CA and the type of certificate you are requesting. Be sure to read the CA's documentation and follow their instructions carefully to ensure that you receive a valid and trusted certificate for your domain.

**PEM, PFX, and .crt are all different formats for storing SSL/TLS certificates and keys.**

* PEM (Privacy-Enhanced Mail) format is a base64-encoded text format **that contains the certificate and private key information**. It is commonly used for Apache and other web servers, as well as for some load balancers and network devices.
* **PFX (Personal Information Exchange) format is a binary format that contains both the certificate and private key in an encrypted format. It is commonly used for Microsoft IIS and other Windows-based systems.**
* .crt (Certificate) format is a binary format that **contains only the certificate information**. It is commonly used for web servers and other SSL/TLS-enabled applications.

The reason we use a single command to create the .crt and .key files is that the key file is needed to create the certificate, and it is best practice to keep the key file secure and not distribute it. By creating the key and certificate in a single command, we can ensure that the key file is not distributed or exposed.

The .crt file contains the SSL/TLS certificate that is used to authenticate the identity of a website or server to clients. It contains information about the certificate issuer, the subject of the certificate (i.e. the website or server name), the public key, and other details such as the expiration date and signature algorithm. The certificate is signed by a trusted certificate authority (CA) to verify its authenticity, and clients can use the certificate to establish a secure connection with the server.

**For development or testing purposes, you can create a self-signed certificate using your CSR and key. A self-signed certificate is a certificate that is signed by its own private key rather than a trusted third-party CA. While self-signed certificates can be useful for testing and development, they are not trusted by default by most browsers and operating systems, so they should not be used in production environments.**

**Here are the steps to create a self-signed certificate using OpenSSL:**

**Generate a self-signed certificate using the following command:**

**openssl x509 -req -days 365 -in example.com.csr -signkey example.com.key -out example.com.crt**

**Replace example.com.csr and example.com.key with the filenames of your CSR and key, respectively. This command generates a self-signed certificate (example.com.crt) that is valid for 365 days.**

**If you have a certificate file in a format other than PEM**

You can use OpenSSL to convert it to PEM format. Here are the steps to do so:

Convert the certificate to PEM format using the following command:

**openssl**

**x509**

**-in example.com.crt**

**-out example.com.pem**

**-outform PEM**

Replace example.com.crt and example.com.pem with the file names you want to use for your certificate in its original format and in PEM format, respectively. This command converts the certificate from its original format to PEM format and saves it to the specified output file.

Verify the PEM format by opening the PEM file in a text editor. The file should contain a block of text that starts with -----BEGIN CERTIFICATE----- and ends with ---END CERTIFICATE-----.

That's it! You now have a certificate file in PEM format that can be used in a variety of applications, such as web servers, load balancers, and other network devices.

**Certificate Authority (CA) provided PEM file.**

Your Certificate Authority (CA) might have provided you with a PEM file because it is a common format for certificates and public keys. The PEM format is a base64-encoded format that is commonly used in Linux and Unix-based systems.

To use the certificate and private key in a Windows-based system, you need to convert the PEM file to a PFX file, which is a PKCS #12 format that contains the certificate and private key. You can use OpenSSL to perform this conversion.

The public key is typically included in the certificate file, so you don't need to generate it separately. You can extract the public key from the certificate using OpenSSL.

Here are the general steps to convert a PEM file to a PFX file:

1. **Open a command prompt or terminal window.**
2. **Navigate to the directory where the PEM file is located.**
3. **Run the following OpenSSL command to create a PFX file:**

**openssl**

**pkcs12**

**-export**

**-out certificate.pfx**

**-inkey private.key**

**-in certificate.pem**

Replace certificate.pem with the name of your PEM file, and private.key with the name of your private key file.

1. **Enter a password when prompted. This password is used to protect the PFX file.**
2. **Verify that the PFX file was created successfully.**

To extract the public key from the certificate, run the following command:

**openssl x509**

**-in certificate.pem**

**-pubkey**

**-noout**

**> publickey.pem**

Replace certificate.pem with the name of your PEM file, and publickey.pem with the name of the file you want to save the public key to.

The publickey.pem file will contain the public key

With these steps, you should have a PFX file containing the certificate and private key, and a PEM file containing the public key.

Trouble Shoot

**The error message "Could not read private key from -inkey file" suggests that OpenSSL is unable to read the private key from the specified file. There could be several reasons for this error, such as:**

* The file does not exist in the specified location.
* The file exists but is not accessible due to file permissions.
* The file is not in the correct format.

To troubleshoot the issue, you can try the following steps:

1. Verify that the private key file exists in the specified location. You can use the ls command (on Linux/Mac) or the dir command (on Windows) to list the files in the directory.
2. Check the file permissions to ensure that you have read access to the private key file. On Linux/Mac, you can use the ls -l command to view the file permissions. On Windows, you can right-click the file, select "Properties", and go to the "Security" tab.
3. Verify that the private key file is in the correct format. You can use the following OpenSSL command to view the contents of the private key file:

**openssl rsa -in example.com.key -noout -text**

If the command displays the private key information, then the file is in the correct format. If the command displays an error message, then the file is not in the correct format.

If none of the above steps help to resolve the issue, then you may need to recreate the private key using OpenSSL. Here is an example command to create a new private key:

**openssl genrsa -out example.com.key 2048**

Replace example.com.key with the desired name for the private key file, and 2048 with the desired key size in bits (e.g., 4096). Once you have recreated the private key, try running the openssl pkcs12 command again.

**If you have uploaded the PFX file to your Azure App Service, you can use it in your Asp.NET Core Web API middleware to get the private key to decrypt the encrypted data. Here are the steps to implement this:**

1. First, you need to add the PFX file as a certificate in your Azure App Service. To do this, go to your App Service in the Azure portal, select "TLS/SSL settings" from the menu, and then upload your PFX file.
2. Once the certificate is uploaded, you can use it in your middleware by specifying the thumbprint of the certificate in your code. You can retrieve the thumbprint of the certificate from the Azure portal by going to your App Service, selecting "TLS/SSL settings" from the menu, and then selecting the uploaded certificate. The thumbprint will be listed under "Certificate details".
3. In your middleware, you can load the certificate from the certificate store using the thumbprint. Here is an example of how to load the certificate and get the private key:

**var thumbprint = "your-certificate-thumbprint";**

**var store = new X509Store(StoreName.My, StoreLocation.CurrentUser);**

**store.Open(OpenFlags.ReadOnly);**

**var certificate = store.Certificates.Find(X509FindType.FindByThumbprint, thumbprint, false)[0];**

**var privateKey = certificate.GetRSAPrivateKey();**

This code will load the certificate with the specified thumbprint from the certificate store, and then get the private key from the certificate.

1. Once you have the private key, you can use it to decrypt the encrypted data in your middleware.

Regarding the PFX password, it is not necessary to use it in your middleware if you have uploaded the PFX file to your Azure App Service. When you upload the PFX file to your App Service, you have the option to specify a password for the file. If you do not specify a password, the PFX file will be uploaded without a password, and you can access the private key without providing a password in your middleware. However, if you have specified a password for the PFX file, you will need to provide the password in your code to access the private key.

**in local development, you can use the thumbprint and X509Store to get the PFX certificate and its corresponding private key to decrypt the encrypted data in the middleware of your ASP.NET Core web API. Here are the steps to do it:**

1. Export the PFX certificate with its private key from the Azure App Service to your local machine.
2. Store the PFX certificate in the Local Machine certificate store using the following command in the command prompt with admin privileges:

**certutil -importpfx -user LocalMachine -p <pfx-password> <pfx-file-path>**

**<pfx-password> is the password that was used to encrypt the PFX file.**

**<pfx-file-path> is the file path of the PFX file.**

This will import the PFX certificate to the LocalMachine certificate store with the given password.

1. In your middleware class, you can retrieve the PFX certificate from the LocalMachine certificate store using its thumbprint value. You can use the following code to retrieve the certificate:

**var thumbprint = "your-certificate-thumbprint";**

**var store = new X509Store(StoreName.My, StoreLocation.LocalMachine);**

**store.Open(OpenFlags.ReadOnly);**

**var certCollection = store.Certificates.Find(X509FindType.FindByThumbprint, thumbprint, false);**

**store.Close();**

**var certificate = certCollection[0];**

This code retrieves the PFX certificate from the Local Machine certificate store using its thumbprint value.

1. Once you have the certificate, you can get its corresponding private key using the GetRSAPrivateKey() method. Here's an example code snippet:

**var rsaPrivateKey = certificate.GetRSAPrivateKey();**

This code retrieves the RSA private key from the PFX certificate.

1. Finally, you can use the RSA private key to decrypt the encrypted data in your middleware using the following code:

**var decryptedData = rsaPrivateKey.Decrypt(encryptedData, RSAEncryptionPadding.Pkcs1);**

This code decrypts the encrypted data using the RSA private key.

Note: In production, it is recommended to store the PFX certificate securely in a Key Vault or Azure App Configuration and retrieve it securely using Azure Key Vault client or Azure App Configuration client respectively.