SYMMETRIC ENCRYPTION METHOD

```
# openssl enc -aes-256-cbc -in msg.txt -out enc.txt
enter aes-256-cbc encryption password:
verify -enter aes-256-cbc encryption password:
#ls
msg.txt   enc.txt
```

(NOTE: msg.txt having some plaintext data, enc.txt having encryption data)

```
# openssl enc -aes-256-cbc -d -in enc.txt -out dec.txt
enter aes-256-cbc decryption password:
# ls
  msg.txt   enc.txt   dec.txt
```

(NOTE: dec.txt having decryption data)

WE WRITE PASSWORD IN COMMAND LINE DIRECTLY

```
# openssl enc -aes-256-cbc -base64 -in msg.txt -out enc.txt -salt -pass pass:<enter your password>
# openssl enc -aes-256-cbc -d -base64 -in enc.txt -out dec.txt -salt -pass pass:<enter your password>
```

ASYMMETRIC ENCRYPTION METHOD

One solution here is to use asymmetric encryption. we can generate a pair of keys- a private key and public key and a public key that is freely available to anyone who wants it.

If someone encrypts some data by using public key, it can be decrypted by using private key.

```
# openssl genrsa -out keypair.pem 2048
<generating RSA private key 2048 bit modules>
# cat keypair.pem
# openssl rsa -in keypair.pem -text
# openssl rsa -in keypair.pem -text -noout
# openssl rsa -in keypair.pem -pubout -out public.pem
# ls
 keypair.pem public.pem
# openssl rsautl -encrypt -in msg.txt -out enc.txt -inkey public.pem
# openssl rsautl -decrypt -in enc.txt -out dec.txt -inkey keypair.pem
```

FOR MORE INFORMATION, PLEASE REFER BELOW STEPS

Asymmetric Encryption

Let's say we have two parties (Alice and Bob) communicating over an insecure network, one on which an attacker (Eve) could intercept all communications. They want to communicate with confidentiality, meaning Eve shouldn't be able to understand what they're talking about, even if she can listen to everything they are sending each other. They want authenticity- Eve shouldn't be able to defraud Alice by sending messages to her claiming to be from Bob, and vice versa.

OpenSSL Demo

Here we'll implement all the steps of that protocol, using openssl terminal commands. In practice you're more likely to use openssl in the form of an API in another language- but learning the terminal commands is still valuable as a transferable skill. Each command is displayed with some explanations of its flags below.

If you want to follow along, you can make 3 folders, 1 for Alice, Bob and the CA respectively. You need to repeat steps 1.1 and 1.2 for Bob and CA so they can have their own pair of keys. And you need to generate a self-signed certificate for the CA (shown below).

Step 1.1 - Alice generates a private key

```
# openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -pkeyopt
rsa_keygen_pubexp:3 -out privkey-A.pem
```

● genpkey → generate a private key

● -algorithm RSA ➝ use the RSA algorithm (can also take “EC” for elliptic-curve)

● -pkeyopt opt:value ➝ set opt to value (see items below)

● rsa_keygen_bits:2048 ➝ sets the size of the key to 2048 bits (the default is 1024)

● rsa_keygen_pubexp:3 ➝ sets the public exponent e to 3 (default is 65, 537)

● -out privkey-A.pem ➝ outputs to the file privkey-A.pem

Step 1.2 - Alice generates a public key

```
# openssl pkey -in privkey-A.pem -pubout -out pubkey-A.pem
```

● pkey ➝ processes public or private keys

● -in privkey-A.pem ➝ read the key from filename privkey-A.pem

● -pubout ➝ output a public key (by default, a private key is output)

-> Viewing the keys in plain text

The keys are saved in base64, and aren’t human readable if you open them in a text editor or the terminal. Luckily, openssl provides us with a handy set of commands to convert them to text. The (-noout) flag suppresses the command from printing out the base64

Step 1.3 - Alice generates a certificate signing request

```
# openssl req -new -key privkey-A.pem -out A-req.csr
```

● req ➝ creates and processes signing requests

● -new ➝ generates a new certificate request, will prompt Alice for some information

● -key privkey-A.pem ➝ signs the request with Alice’s private key

-> Generating a self-signed certificate for the CA

```
# openssl req -x509 -new -nodes -key rootkey.pem -sha256 -days 1024 -out
root.crt
```

Step 1.4 - CA generates and signs a certificate for Alice

```
# openssl x509 -req -in A-req.csr -CA root.crt -CAkey rootkey.pem
-CAcreateserial -out A.crt -days 500 -sha256
```

● x509 ➝ an x509 certificate utility (displays, converts, edits and signs x509 certificates)

● -req → a certificate request is taken as input (default is a certificate)

● -CA root.crt → specifies the CA certificate to be used as the issuer of Alice's certificate

● -CAkey rootkey.pem → specifies the private key used in signing (rootkey.pem)

● -CAcreateserial → creates a serial number file which contains a counter for how many certificates were signed by this CA

● -days 500 → sets Alice's certificate to expire in 500 days

● -sha256 → specifies the hashing algorithm to be used for the certificate's signature

→ Viewing the certificate as text:

```
# openssl x509 -in Alice.crt -text -noout
```

Step 2.1 - Alice verifies Bob's public certificate

```
# openssl verify -CAfile root.crt Bob.crt
```

● verify → a utility that verifies certificate chains

● -CAfile root.crt → specified the trusted certificate (root.crt)

● Bob.crt → the certificate to verify

● If you get an OK, you know the certificate can be trusted

Step 2.2 - Alice extracts Bob's public key

```
# openssl x509 -pubkey -in Bob.crt -noout > pubkey-B.pem
```

● -pubkey → outputs the certificate's public key (in PEM format)

Step 2.3 - Alice tries to encrypt her largefile.txt with Bob's public key

```
# openssl pkeyutl -encrypt -in largefile.txt -pubin -inkey pubkey-B.pem -out
ciphertext.bin
```

● pkeyutl → utility to perform public key operations

● -encrypt → encrypt the input data

● error! (recall: RSA is not meant for encrypting arbitrary large files- Alice needs to use symmetric key encryption for that)

Step 3.1 - Alice generates a symmetric key

# openssl rand -base64 32 -out symkey.pem

● rand → generates pseudo-random bytes (seeded by default by $HOME/.rnd)

● -base64 32 → outputs 32 random bytes and encodes it in base64

Step 3.2 - Alice encrypts symkey.pem using Bob's public key

# openssl pkeyutl -encrypt -in symkey.pem -pubin -inkey pubkey-B.pem -out

symkey.enc

Step 3.3 - Alice hashes symkey.pem and encrypts it using her private key

# openssl dgst -sha1 -sign privkey-A.pem -out signature.bin symkey.pem

● dgst -sha1 → hash the input file using the sha1 algorithm

● -sign privkey-A.pem → sign the hash with the specified private key

● symkey.pem → the input file to be hashed

Step 4.1 - Bob decrypts symkey.enc using his private key

# openssl pkeyutl -decrypt -in symkey.enc -inkey privkey-B.pem -out

symkey.pem

● -decrypt → decrypt the input file

Step 4.2 - Bob gets and verifies Alice's certificate and extracts her public key

(This is simply a retread of what Alice did in step 2)

Step 4.3 - Bob verifies the message is from Alice

Steps 4.3 and 4.4 in the protocol are combined in this step. Bob hashes symkey.pem, decrypts signature.bin, and compares the two results in one command:

# openssl dgst -sha1 -verify pubkey-A.pem -signature signature.bin symkey.pem

● -verify pubkey-A.pem → verify the signature using the specified filename

● -signature signature.bin → specifies the signature to be verified

● symkey.pem → the file to be hashed

Step 5.1 - Alice encrypts her largefile.txt with the symmetric key

# openssl enc -aes-256-cbc -pass file:symkey.pem -p -md sha256 -in

```
largefile.txt -out ciphertext.bin
```

● enc -aes-256-cbc → encrypt a file using the aes-256-cbc symmetric key algorithm

● -pass file:symkey.pem → specified the file to get the symmetric key from

● -p → prints the key, salt, initialization vector to the screen

● -md sha256 → uses sha256 as part of the key derivation function (a function that derives one or more secondary secret keys from a primary secret key)

Step 5.2 - Bob decrypts ciphertext.bin with the same symmetric key

```
# openssl enc -aes-256-cbc -d -pass file:symkey.pem -p -md sha256 -in
ciphertext.bin -out largefile.txt
```

● -d → decryption flag