Introduction to Cryptography

OpenSSL is an open-source cryptography toolkit used to:

- Implement SSL/TLS (secure communication over networks)
- Perform encryption & decryption
- Generate and manage keys, hashes, and digital certificates

It is widely used in web servers (HTTPS), VPNs, email security, IoT, and cybersecurity labs.

Why OpenSSL Is Important

Without OpenSSL (or similar libraries):

- Websites couldn't use **HTTPS**
- Data like **passwords, credit cards, and logins** would be exposed
- Secure client–server communication wouldn't exist

Core Components of OpenSSL

1. SSL/TLS Protocol Implementation

- Handles **secure handshakes**
- Encrypts data in transit
- Authenticates servers (and optionally clients)

2. Cryptographic Library

Provides algorithms such as:

- **Symmetric encryption**: AES, DES
- **Asymmetric encryption**: RSA, ECC
- **Hashing**: SHA-256, SHA-512
- **Digital signatures**

3. Command-Line Tool

Used for:

- Creating certificates
- Testing HTTPS connections
- Encrypting/decrypting files
- Hashing passwords

How SSL/TLS Works

1. Client connects to server
2. Server sends digital certificate
3. Client verifies certificate
4. Secure keys are exchanged
5. Encrypted communication begins

Encryption is the process of converting **plain text → cipher text** to protect data from unauthorized access. There are two fundamental encryption models:

1. **Symmetric Encryption**
2. **Asymmetric Encryption**

What is AES

AES (Advanced Encryption Standard) is a **symmetric encryption algorithm**:

- Uses **one secret key**
- Same key is used for **encryption & decryption**
- Extremely fast and secure
- Used in **HTTPS, VPNs, disk encryption, cloud storage**

We will use **AES-256-CBC**, a common learning and lab standard.

File using AES 256 , one of the toughest encryption algorithms

Prerequisites

✓ Linux / Kali / Ubuntu / macOS
✓ OpenSSL installed
Check with:

Command : `openssl version`

Prepare a File to Encrypt

Create a sample file:

nano secret.txt

Add content:

My bank password is 1234

Save the file.

This is **plaintext** (readable data).

Understand the AES Command to Encrypt the File

We will use this command:

```
openssl enc -aes-256-cbc -salt -in secret.txt -out secret.enc
```

- **openssl**

Invokes OpenSSL, a cryptographic toolkit that provides encryption, hashing, certificates, and SSL/TLS utilities.

This is the main program.

- **enc**

Short for encode/encrypt.

- Tells OpenSSL you want to:
  - Encrypt (default)
  - Or decrypt (if -d is used)

Since -d is not specified, OpenSSL performs encryption.

- **-aes-256-cbc**

Specifies the encryption algorithm.

Let's split it:

- AES → Advanced Encryption Standard
- 256 → 256-bit key length (very strong)
- CBC → Cipher Block Chaining mode

Why this matters:

- 256-bit key → extremely hard to brute-force
- CBC → prevents identical plaintext blocks from producing identical ciphertext

- **-salt**

Adds random salt to the encryption process.

What does salt do?

- Makes encryption unique each time
- Prevents:
  - Rainbow-table attacks
  - Same password → same ciphertext

Without -salt:

- Two files encrypted with same password look identical
- -salt is critical for security.

➢ **-in secret.txt**

Specifies the input file.

- secret.txt = plaintext file
- Contains readable data before encryption

Example:

My password is 1234

-out secret.enc

Specifies the output file.

- secret.enc = encrypted file
- Contains unreadable binary data (ciphertext)

Original file remains unchanged.

What Happens When You Run This Command

Step 1: Password Prompt

Enter encryption password:

Verify encryption password:

- This password is not used directly
- It is input to a Key Derivation Function (KDF)

Step 2: Key & IV Generation

From your password + salt:

- AES-256 encryption key is derived
- IV (Initialization Vector) is generated

Key → encrypts data

IV → ensures randomness for first block

Step 3: File Block Processing

- secret.txt is split into 128-bit blocks
- AES encrypts each block
- CBC mode chains blocks together

Step 4: Ciphertext Output

- Encrypted data written to secret.enc
- Appears as random data

Verify Encryption Worked

Try reading encrypted file:

Command : `cat secret.enc`

output

`Salted__...`

This confirms:

- Original data is unreadable
- File is securely encrypted

Decryption Command

```
openssl enc -aes-256-cbc -d -in secret.enc -out secret.txt
```

openssl	Runs the OpenSSL tool
enc	Encryption/decryption utility
-aes-256-cbc	Same algorithm used during encryption
-d	Decrypt mode
-in secret.enc	Encrypted input file
-out secret.txt	Decrypted output file

Important:

- The algorithm and mode must match encryption exactly.
- You must enter the same password used during encryption.

What Happens Internally During Decryption

1. Password Verification

- You are prompted for the password
- OpenSSL applies the same Key Derivation Function (KDF) using:
    - Entered password
    - Salt stored inside secret.enc

2. Key & IV Reconstruction

- AES-256 key and Initialization Vector (IV) are regenerated
- If password is wrong → key is wrong → output is unreadable

3. AES-CBC Decryption

- Ciphertext blocks are decrypted
- CBC chaining is reversed block-by-block
- Original plaintext blocks are restored

4. Plaintext Output

- Decrypted content is written to:

secret.txt

File is now readable again

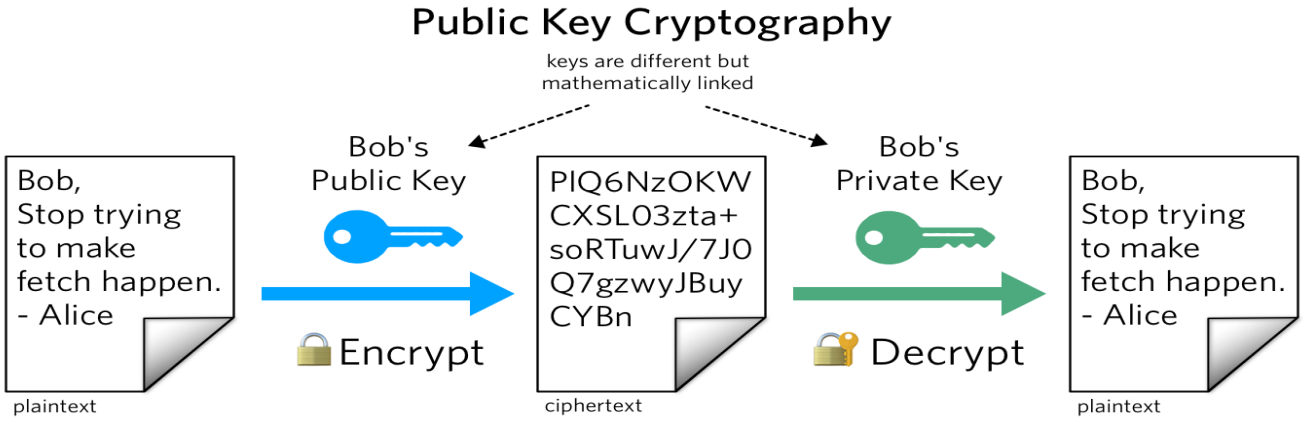
What Are RSA Keys?

RSA is an asymmetric encryption system that uses two keys:

- Private key → kept secret
- Public key → shared openly

They are used for:

- Encryption / decryption
- Digital signatures
- HTTPS certificates
- SSH authentication



Explaining the RSA Key-Pair Generation Command

```
openssl genrsa -out wikipedia.key 2048
```

What This Command Does

This command uses OpenSSL to:

- Generate a 2048-bit RSA private key
- Store it in a file named wikipedia.key
- From this private key, the public key can be derived (forming the RSA key pair)

Even though only one file is created, both keys already exist mathematically.

- **openssl**

Runs the OpenSSL cryptographic toolkit.

- **genrsa**

Means Generate RSA:

- Creates an RSA private key
- Uses secure random number generation
- Implements RSA key-generation mathematics

- **-out wikipedia.key**

- Specifies the output file
- wikipedia.key contains the RSA private key
- This file must be kept secret

- **2048**

- Specifies the key size in bits
- 2048-bit RSA:
    - Industry standard
    - Secure for HTTPS, SSH, certificates
    - Resistant to current brute-force attacks

View Private Key Details

```
openssl rsa -in private.key -text -noout
```

What Is Inside wikipedia.key?

The file contains:

- RSA modulus (n)
- Public exponent (e)
- Private exponent (d)
- Two large prime numbers (p and q)
- All values encoded in PEM format

How the RSA Key Pair Works

Encryption

- Sender encrypts data using **public key**
- Only the matching **private key** can decrypt

Decryption

- Private key reverses the encryption
- Public key **cannot decrypt**

Public Key Extraction Command

```
openssl rsa -in wikipedia.key -pubout -out wikipedia_public.key
```

What This Command Does

- OpenSSL, this command:
- Reads an existing RSA private key (wikipedia.key)
- Derives the corresponding public key
- Saves it as wikipedia_public.key

No new keys are generated.
The public key already exists mathematically inside the private key.

- -pubout
- Instructs OpenSSL to output only the public key
- Removes all private components

View Public Key Details

```
openssl rsa -pubin -in public.key -text -noout
```

How the Public Key Is Used

Encryption

- Data encrypted with the public key
- Only the private key can decrypt

Signature Verification

- Public key verifies digital signatures
- Private key creates signatures

CSR (Certificate Signing Request) Generation Command

```
openssl req -new -key wikipedia.key -out wikipedia.csr
```

What This Command Does

Using OpenSSL, this command:

- Takes an existing RSA private key (wikipedia.key)
- Creates a Certificate Signing Request (CSR)
- Saves it as wikipedia.csr

The CSR is later sent to a Certificate Authority (CA) to get a signed SSL/TLS certificate.

- openssl

Runs the OpenSSL cryptographic toolkit.

- req

Short for request.

- Used to create and manage X.509 certificate requests
- Handles:
    - CSRs
    - Self-signed certificates
- -new
- Tells OpenSSL to create a new request
- Without this, OpenSSL would expect an existing CSR

- -key wikipedia.key
- Specifies the private key to use
- wikipedia.key is your RSA private key

Use this private key:

- Proves *you are the owner*
- The private key is never shared
- Used to:
    - Digitally sign the CSR
    - Prove ownership of the key
- -out wikipedia.csr

Save the request in this file:

- wikipedia.csr
- This file is safe to share

The private key is never shared.

- -out wikipedia.csr
- Specifies the output file
- wikipedia.csr contains the certificate request
- This file is safe to share with a CA

What Is Inside wikipedia.csr?

The CSR contains:

- Public key (derived from wikipedia.key)
- Subject information:
    - Country (C)
    - State (ST)
    - City (L)
    - Organization (O)
    - Organizational Unit (OU)
    - Common Name (CN – domain name)
- A digital signature created using the private key

Private key is NOT included.

Verify the CSR

```
openssl req -in wikipedia.csr -text -noout
```

Confirms domain name, public key, signature

Example **The Situation (Real Life)**

You want to open a **locker in a bank**.

The bank says:

"We can give you a locker, but first **prove who you are**."

Step-by-Step Mapping (Real Life → OpenSSL)

Step 1: Your Private Key = Your Personal Signature

wikipedia.key

Real life:

- Your **handwritten signature**
- Your **fingerprint**
- Your **ID proof**

You NEVER give this to anyone.

Step 2: CSR = Application Form

wikipedia.csr

Real life:

- Locker application form
- Contains:
    - Your name
    - Address
    - ID number
- **Signed by you**

This is what you submit to the bank.

What This Command Does (Using the Bank Example)

```
openssl req -new -key wikipedia.key -out wikipedia.csr
```

OpenSSL is doing this:

1. Takes your **private key** (your signature)
2. Asks for your **details** (name, organization, domain)
3. Creates an **application form (CSR)**
4. **Signs the form** using your private key

So the bank knows:

"Yes, this request really came from YOU."

If you created a CSR like wikipedia.csr:

```
openssl req -in wikipedia.csr -text -noout
```

What Is Inside the CSR? (Locker Form)

CSR contains:

- ✅ Your **public information**
- ✅ Proof you signed it
- ❌ NOT your private key

Just like:

- Application form has your name
- It does NOT contain your fingerprint

Step 3: Bank (Certificate Authority)

The bank checks:

- Is your form valid?
- Is your signature correct?

If yes → bank approves.

Step 4: Bank Gives You the Locker Key (Certificate)

Real life:

- Bank gives you **locker access**

Computer world:

Certificate (.crt)

Now:

- Your identity is trusted
- Secure communication is allowed

🔄 **Simple Flow (One Look)**

Private Key → Your signature

CSR → Application form

Certificate Authority → Bank

Certificate → Approval / Locker access

What If You Don't Create CSR?

- No application
- No verification
- No approval
- No HTTPS / No trust

Self-signed SSL Command

```
openssl x509 -in wikipedia.csr -out wikipedia.crt -req -signkey wikipedia.key -days 365
```

It creates a self-signed SSL certificate using your own private key, without involving any Certificate Authority (CA).

Think of a Real-Life Example (College Context)

- **CSR** → Admission application form
- **Private key** → Your signature
- **Self-signed certificate** → *You approve your own admission*

There is no college authority checking it — you approve yourself.

- **openssl**

Starts the OpenSSL security tool.

- **x509**

Tells OpenSSL:

"I am working with an **X.509 certificate** (SSL certificate format)."

- **-in wikipedia.csr**

Input file:

- wikipedia.csr = your **certificate request**
- Contains:
  - Your public key
  - Your identity details

- **-req**

Means:

"This input file is a **CSR**, not a certificate."

- **-signkey wikipedia.key**

This is the **most important part**.

- Uses your **private key**
- Signs the certificate **by yourself**
- No CA involved

This is why it's called **self-signed**.

- **-out wikipedia.crt**

Output file:

- wikipedia.crt = the **certificate**
- This file is shared with servers/browsers

- **-days 365**

Validity period:

- Certificate is valid for **365 days (1 year)**

After that → it **expires**.

What Happens Internally (Step by Step)

1. OpenSSL reads your **CSR**
2. Extracts:
    - Public key
    - Identity information
3. Creates a certificate structure
4. Uses your **private key** to sign it
5. Writes the result to wikipedia.crt

Where Self-Signed Certificates Are Used

- Localhost testing
- Development servers
- Internal company apps
- Learning OpenSSL & HTTPS

Hash files and verify integrity

A hash is a fixed-length fingerprint of a file.

- Same file → same hash
- File changes (even 1 bit) → completely different hash

Common algorithms:

- SHA256 (recommended)

- SHA1 (weak)
- MD5 (broken , only for learning)

Why Verify File Integrity?

"Has this file been changed, corrupted, or tampered with?"

Used in:

- Software downloads
- Forensics
- Malware analysis
- Cybersecurity audits
- Secure file transfers

Step 1: Create a Test File (Kali Linux)

```
nano file.txt
```

Add:

This is my important file

Save and exit.

Step 2: Generate a Hash (SHA256)

Hash the file

```
sha256sum file.txt
```

Example output:

3f79bb7b435b05321651daefd374cdc681dc06faa65e374e38337b88ca046dea file.txt

This long string is the hash value.

Step 3: Save Hash to a File (Best Practice)

```
sha256sum file.txt > file.txt.sha256
```

Now you have:

file.txt

file.txt.sha256

Step 4: Verify File Integrity (IMPORTANT)

- **Verification command**

```
sha256sum -c file.txt.sha256
```

Correct output

file.txt: OK

This means:
File is unchanged
Integrity is verified

Step 5: Tamper the File (Demo Attack)

```
echo "hacked" >> file.txt
```

Now verify again:

```
sha256sum -c file.txt.sha256
```

Output

file.txt: FAILED

sha256sum: WARNING: 1 computed checksum did NOT match

Even a **tiny change** breaks integrity.

What Happened Internally?

1. Original file → hash calculated
2. Hash stored securely
3. Later, file is re-hashed
4. Hashes compared
5. Match = OK | Mismatch = Tampered

Group Algorithms by Type

1) Symmetric Encryption

- One secret key
- Very fast
- Used for bulk data

Examples: AES, ChaCha20

---

2) Asymmetric Encryption

- Public key + Private key
- Slower
- Used for key exchange & identity

Examples: RSA, ECC

---

3) Hashing (Not Encryption)

- One-way
- Integrity & passwords

Examples: SHA-256, SHA-512

Never compare hashing directly with encryption—they solve different problems.

Choose Comparison Criteria (Very Important)

Always compare algorithms using **the same criteria**:

Criteria	**What it Means**
Security strength	Resistance to attacks
Speed	Performance impact
Key size	Larger = stronger (usually)
Use case	Where it fits best
Scalability	Works well at large scale

Compare Common Algorithms (With Examples)

AES vs RSA

Feature	**AES**	**RSA**
Type	Symmetric	Asymmetric
Speed	Very fast ⚡	Slow 🐢
Key Size	128 / 256-bit	2048 / 4096-bit
Data Size	Large files	Small data only
Main Use	Encrypt data	Exchange keys

Example:

- Encrypting a **10 MB file → AES**
- Exchanging a secret key → **RSA**

RSA is **not used** to encrypt large data directly.