Security - Confidentiality

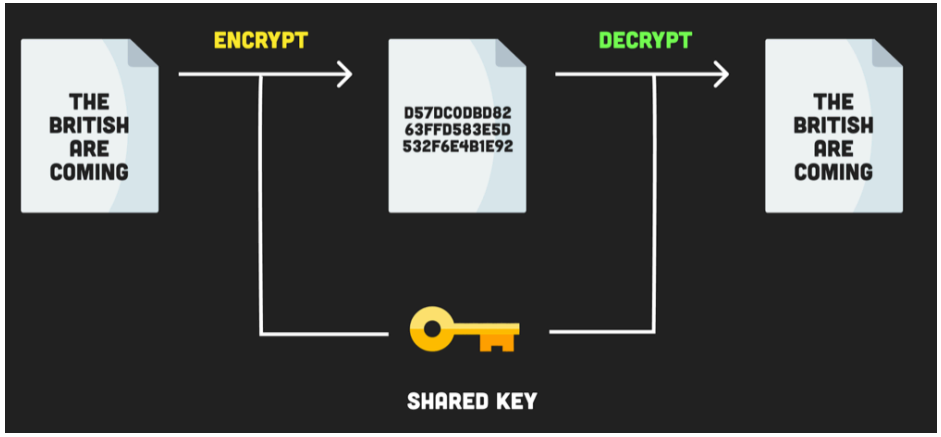
Confidentiality Tools

1. Symmetric Encryption & Decryption
2. DH (Diffie-Hellman) Key Exchange
3. Asymmetry Encryption & Decryption
4. Hybrid Encryption & Decryption

1. Symmetric Encryption & Decryption

Encryption is the process of making a message confidential, while allowing it to be reversible (decrypted) with the proper key.

In **symmetric encryption**, the same key is used to encrypt and decrypt the message.



Creating Keys and Initialization Vector

The first step is to create a key and an initialization vector (iv).

This information becomes the key that is shared.

```
// These key/iv should be shared
const key = randomBytes(32);
const iv = randomBytes(16);

const infoToShare = {
    key: key.toString('hex'),
    iv: iv.toString('hex')
};

const cipher = createCipheriv('aes256', key, iv);
```

Why IV?

It might seem like just a **password (key)** should be enough — but the **IV (Initialization Vector)** solves a hidden problem: **repetition**.

Problem: Same Message, Same Result

- If you encrypt the same message twice with the same key, you'll get **the same encrypted output** every time.
- Hackers can notice patterns — for example, they might see that two people sent the *same* secret message.

Solution: Add Randomness

The **IV** adds randomness to the start of encryption, so even if you use the same password and message, you should get **different encrypted results** each time.

In short:

> The **IV** adds randomness to make encryption unpredictable even when the **key** (password) stays the same — protecting **Confidentiality**.

Encryption Code

We share key and iv:

```
const infoToShare = {
    key: key.toString('hex'),
    iv: iv.toString('hex')
};
```

- We can use the cipher object to encrypt the message.
- We should add the final code to make it harder to decrypt.

```
const cipher = createCipheriv('aes256', key, iv);
// output is hex
const encryptedMessage = cipher.update(message, 'utf8', 'hex')
    + cipher.final('hex');
```

Decryption Code

Once we have the shared key and iv, we can decrypt the message.

```
const key2 = Buffer.from(infoToShare.key, 'hex');
const iv2 = Buffer.from(infoToShare.iv, 'hex');

const decipher = createDecipheriv('aes256', key2, iv2);

// output is utf-8
const decryptedMessage = decipher.update(encryptedMessage, 'hex', 'utf-8')
    + decipher.final('utf8');
```

AES 256 Algorithm

We use **AES-256 (Advanced Encryption Standard)**, a trusted algorithm maintained by **NIST**.

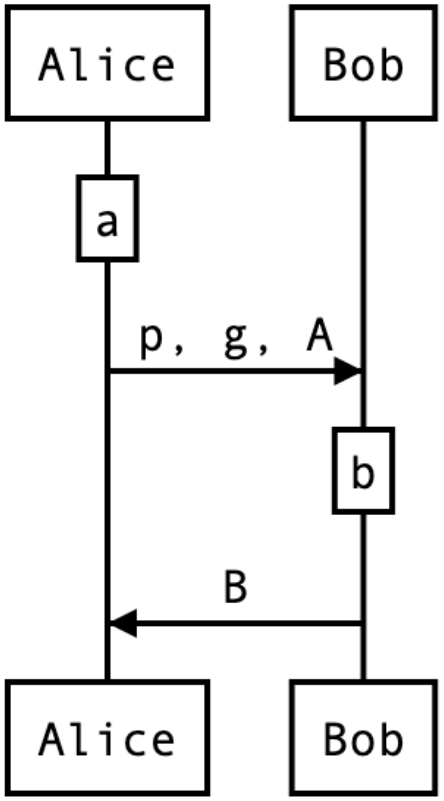
- Never create your own secret encryption — always use proven, standard algorithms like **AES-256** for security and reliability.
- AES is considered unbreakable with today’s technology.

2. DH (Diffie-Hellman) Key Exchange

Sending secret keys directly can be risky — someone might steal them in transit: So people invented a clever way to **create the same secret key without ever sending it**.

- That's what the **Diffie–Hellman (DH) Key Exchange** does.
- DH (Diffie-Hellman) is a method of securely exchanging cryptographic keys over a public channel.

How It Works



1. Alice and Bob have random integers `a` and `b` that they do not share
2. Alice creates `A = g^a mod p` and shares `p`, `g`, and `A` with Bob
3. Bob creates `B = g^b mod p` and sends `B` to Alice

Key Points:

- Hackers can steal `p, g, A, B`, but not `a` (private only to Alice) and `b` (private only to Bob)
- Both Bob and Alice can generate a key `g^(ab) mod p` from:
    - `(A)^b mod p` = `(B)^a mod p` = `g^(ab) mod p`
- The hacker cannot make `g^(ab) mod p` from `p, g, A, B`
- Both Bob and Alice can share the key with this simple integer arithmetic

3. Asymmetry Encryption & Decryption

The Problems of Symmetric Encryption

- We should share keys to use symmetric encryption
- What if we should share keys with millions of people?

The Solution: Public Key Cryptography

The idea: Instead of sharing the key, share the lock!

- Everyone can use the public lock to **encrypt** their message, but only the person with the **private key** can **unlock** it.
- This way, the key is never shared — only the lock is.

RSA Algorithm

This algorithm, called the **RSA algorithm**, is a revolutionary idea that changed the encryption/decryption system.

- In RSA, we call the lock a **public key** and the key a **private key**
- The acronym "RSA" comes from the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman, who publicly described the algorithm in 1977
- They got the Turing award in 2002
- This RSA algorithm (or public key algorithm) is used widely when sharing information on the Internet

Generating Key Pairs

We need to generate a public and private key pair using the keypair.js module.

```
const { generateKeyPairSync } = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('rsa', {
    modulusLength: 2048, // the length of your key in bits
    publicKeyEncoding: {
        type: 'spki', // recommended by the Node.js docs
        format: 'pem',
    },
    privateKeyEncoding: {
        type: 'pkcs8', // recommended by the Node.js docs
        format: 'pem',
    },
});

export { privateKey, publicKey };
```

PEM Format

PEM stands for **Privacy Enhanced Mail** —
but today, it's mainly used as a **standard text format** for storing and sharing cryptographic data such as keys, certificates, and signatures.

PEM contains three parts:

1. **Header** → `-----BEGIN PUBLIC KEY-----`
2. **Base64-encoded data** → the actual binary key, turned into text
3. **Footer** → `-----END PUBLIC KEY-----`

When decoded from Base64, it becomes binary data structured using **ASN.1 / DER** (a standard data format).

That binary structure includes **mathematical components** like:

- `n` → large modulus number
- `e` → public exponent
- (and for private keys: `d`, `p`, `q`, etc.)

In short, those large numbers are stored as **binary data**, then **Base64-encoded** into a **human-readable text format** (PEM).

Why We Use PEM?

- Easy to store in text files (`.pem` , `.crt` , `.key`)
- Easy to send in email or copy between systems
- Works across many platforms (OpenSSL, Node.js, Python, etc.)

PEM Example: Public & Private Keys

This is an example of generated public key in the PEM format; this can be shared by anyone.

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2gC6TNPGXRGsOgTZrAFG
HcsPCNLtmIHy/swW80CKU1j1a36sw/ohF0RANYTIy535kzokuQlCjFRmSpfYtS+f
YVn9Jb53owNuIF3jm3lmN9JEiL+wWnUMjTJ0VXpVb+kX8oAx78ijhk8FHuD7Sg9k
w479qDZJHX+5pHfA2xajEfdR77I2SK5Ro/0m7UjaJnQDjrNPDwXapRZ0rj9HkzKY
dx4LAwG90+Z6kRDXsfVmMFo3h8370fsTncCYO4zb8w1yGuJMGXUVE78pceZWYqtd
dgZCtbZYJjkQWVYgXfb1DuZgI0UOIh1bVEe5htI3Iql45bf3pSANGAkeLb5A7mwS
ewIDAQAB
-----END PUBLIC KEY-----
```

This is an example of generated private key in the PEM format: this should be kept as secret.

```
-----BEGIN PRIVATE KEY-----
MIIEvAIBADANBgkqhkiG9w0BAQEFAASCBKYwggSiAgEAAoIBAQClDJnAQDzkHjkP
vXCnK/VU5PWoiPcLU6SA0MgxrJ05sX3IPpWZD1wHoosauMPDKzthIq0loyxkjZux
Zc6gbKBVUkaVTXTITYJk3eABHYftLxuqgYZC07ghCb23AcVeHyqtTLRrvhGjA/OO
BrCM8ULgBV+2xeJ8WGTgGtm+vCCgaun6eyxCtsVeZsVDMxA1N+JHeh5d391tg7yl
/J9xfUKkACGF8fbtarK59H1hkHdBzvnYzpdngl3YtJAe99N3MMNVOXadDScyYfWO
L5hNOguVC3BD1+GhqShTtOWCnDkdeo8g937E+dGF3dnz5NOLOiIC/DDniKX06591
1IC0mDvbAgMBAAECggEAFEbphrx9npr18LCZ6/fTEqR0Z8m37BzRC4Ba7/tgb4uW
Q+kocN75s/hstZI4frxLdcIQWpzi4dGh7Jfw3DWsNv8+K66R59/Z4rF9iTWLEwEk
GX0DNSeUqf5r39VoXYlfsV3IlqoJuajNHchFnXMwf5w5Z5eFHTiw3/V8hdnP+lUl
jokdfnV3r4lIH5lV6eEfPkjeCHk+Xv4gphUpW5cmux3BspH1IyOcfGDWabd+9964
T7zdRldhscJDXemZgqmVBfl+5PBIztVK2KmzeOp3P0PfVjpjCjaaJNxKb8uCJFWl
0lm3V8oJpVuKEaPV6z6u1aCTM2FZFoo88ytcA/HAAQKBgQDfTTzZf3d05V/s7rhb
jZFRQXX984cOmmknsvCx/DgVUVQI2Rd+8Ur8nhhMqyERDn2JFirScnxVLSDTcmYN
+YtZPs9aJQXpmDUmCn3dyLwKdVE5Mq0DRfuexztpE1CYtVinvS47cNPz50dG/43/
yXzm16WbWyWGZmKx1vnMlXRAAQKBgQC9N7DROeierFhtP0Xs0IGqBICYXkmlf8Sy
vCRhS9tFw9kby/G8fBunSuQZGlbEkr7sABQeIu5YYGoQtBshMjFjK5buTdysvTHD
1MFBnLTQeEH2Q9eSfcYajW40huOobJco97qFDKjxp2O8VJIbkF+rvyRB9b1LuVpR
DCQQv2V72wKBgGXGYEuJSzreKIBmtWQU+/ya51pdmlrPicnsRsP4ft9IeBKoT4su
Es1ciRq4jJOnlRPcJJxSTdTMgr+czmLkL9z3Qa6GXVGAbAjwkzl4eiwSGHkHXc+2
H97YU32qrftL/CJHmUDCRfhOmzAC3geNH/5enz5Vpp7BH46KusPCAoABAoGAXJwB
QqYhIGgh73Zkdv+mNtx1RB4B1aqvTvueorBJ5d9xR5WN/z6PGlYhRdgKV7OrFj8D
maxnE4I852f/T0QNwAf5mzEorLL8xGhBHifgp5f0UN/rhptgdmZ6ZeMPUcO0SCzX
cX88wOm4eti5mhwTdy/aQx5PQ9y4Q6q5jCjhNrkCgYAvJWEypGmdmHwpb3wPceUI
IbfTxkCG92gl1wAPaR0gWLAlHIUy4P5+PV8hs12izfIg2ARuBXO7e/I3uoLYfx1i
9PkokI75M+c1fA8T8ipJ3sUVv/RUPRAGPjwwYflVR2zLelPqbcZjHeGrCBc/CF/V
9JyVojsCk3pEfIa/gkEwkA==
-----END PRIVATE KEY-----
```

Encryption with Public Key

Using the public and private key, we can encrypt any information.

```
const { publicEncrypt, privateDecrypt } = require('crypto');
const { publicKey, privateKey } = require('./keypair');

const secretMessage = 'ASE 285 students are superb!';

const encryptedData = publicEncrypt(
    publicKey,
    Buffer.from(secretMessage)
);

console.log(encryptedData.toString('hex'));
```

Decryption with Private Key

We can decrypt the information using the private key.

```
const { publicEncrypt, privateDecrypt } = require('crypto');
const { publicKey, privateKey } = require('./keypair');

const decryptedData = privateDecrypt(
    privateKey,
    encryptedData
);

console.log(decryptedData.toString('utf-8'));
```

4. Hybrid Encryption & Decryption

The Problem

- **Symmetric encryption** is fast but hard to share keys securely.
- **Public-key encryption** is slower but easy to share keys.
- **Solution:** Use both — this is called **Hybrid Encryption**.

How Hybrid Encryption Works

The idea: "Use asymetric key to share symmetric key"

1. We first encrypt the **shared (symmetric) key** with the **public key** and send it securely.
2. After that, both sides use the shared key for fast, secure communication.

Benefits:

- Combines the security of asymmetric encryption for key exchange
- Uses the speed of symmetric encryption for data transmission
- Best of both worlds!