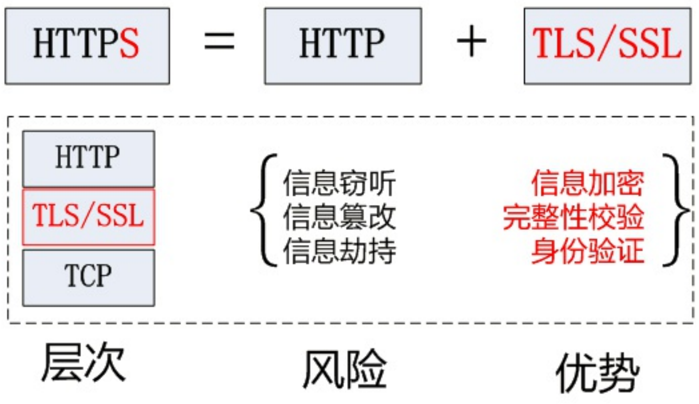
link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=147 sentences=398, words=2582

0.HTTPS简介

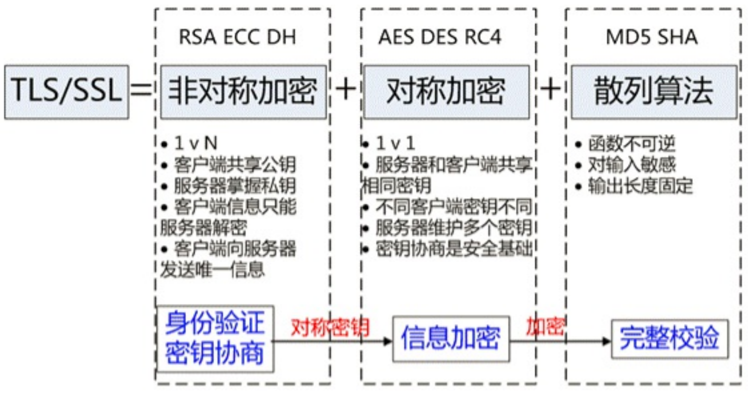
0.1. SSL和TLS

- 传输层安全性协议(Transport Layer Security，缩写TLS），及其前身安全套接层(Secure Sockets Layer,缩写SSL）是一种安全协议，目的是为互联网通信，提供安全及数据完整性保障



0.2. HTTPS

- HTTPS 协议的主要功能基本都依赖于 TLS/SSL 协议，TLS/SSL 的功能实现主要依赖于三类基本算法
  - 散列函数 散列函数验证信息的完整性
  - 对称加密 对称加密算法采用协商的密钥对数据加密
  - 非对称加密 非对称加密实现身份认证和密钥协商



1. 加密

- 加密就是研究如何安全通信的
- 保证数据在传输过程中不会被窃听
- crypto (https://nodejs.org/dist/latest-v13.x/docs/api/crypto.html)

1.1 对称加密

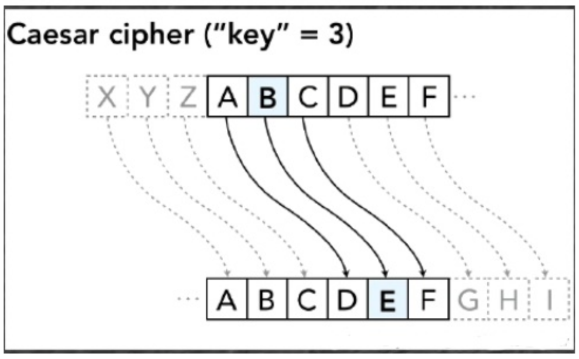
- 对称加密是最快速、最简单的一种加密方式,加密(encryption)与解密(decryption)用的是同样的密钥(secret key)
- 主流的有 AES和 DES

1.1.1 描述

1.1.2 简单实现

- 消息 abc
- 密钥 3
- 密文 def

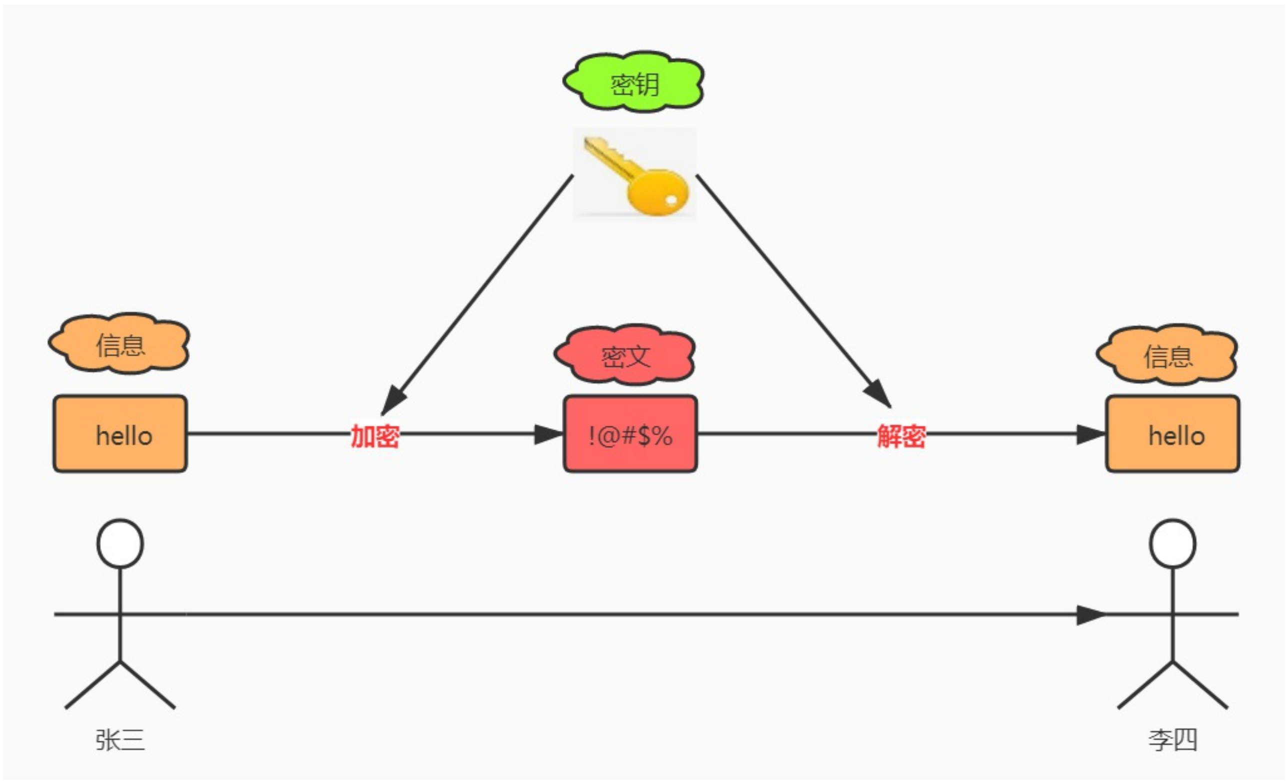
➢古罗马皇帝凯撒在打仗时曾经使用过以下方法加密军事情报：



```javascript
let secretKey = 3;
function encrypt(str) {
    let buffer = Buffer.from(str);
    for (let i = 0; i < buffer.length; i++) {
        buffer[i] = buffer[i] + secretKey;
    }
    return buffer.toString();
}
function decrypt(str) {
    let buffer = Buffer.from(str);
    for (let i = 0; i < buffer.length; i++) {
        buffer[i] = buffer[i] - secretKey;
    }
    return buffer.toString();
}
let message = 'abc';
let secret = encrypt(message);
console.log(secret);
let value = decrypt(secret);
console.log(value);
```

1.1.3 AES

- crypto.html (https://nodejs.org/api/crypto.html#crypto_crypto_createcipheriv_algorithm_key_iv_options)
    - algorithm用于指定加密算法，如aes-128-ecb、aes-128-cbc等
    - key是用于加密的密钥
    - iv参数用于指定加密时所用的向量
- 如果加密算法是128，则对应的密钥是16位，加密算法是256，则对应的密钥是32位



```
const crypto = require('crypto');
function encrypt(data, key, iv) {
    let decipher = crypto.createCipheriv('aes-128-cbc', key, iv);
    decipher.update(data);
    return decipher.final('hex');
}

function decrypt(data, key, iv) {
    let decipher = crypto.createDecipheriv('aes-128-cbc', key, iv);
    decipher.update(data, 'hex');
    return decipher.final('utf8');
}

let key = '1234567890123456';
let iv = '1234567890123456';
let data = "hello";
let encrypted = encrypt(data, key, iv);
console.log("数据加密后:", encrypted);
let decrypted = decrypt(encrypted, key, iv);
console.log("数据解密后:", decrypted);
```

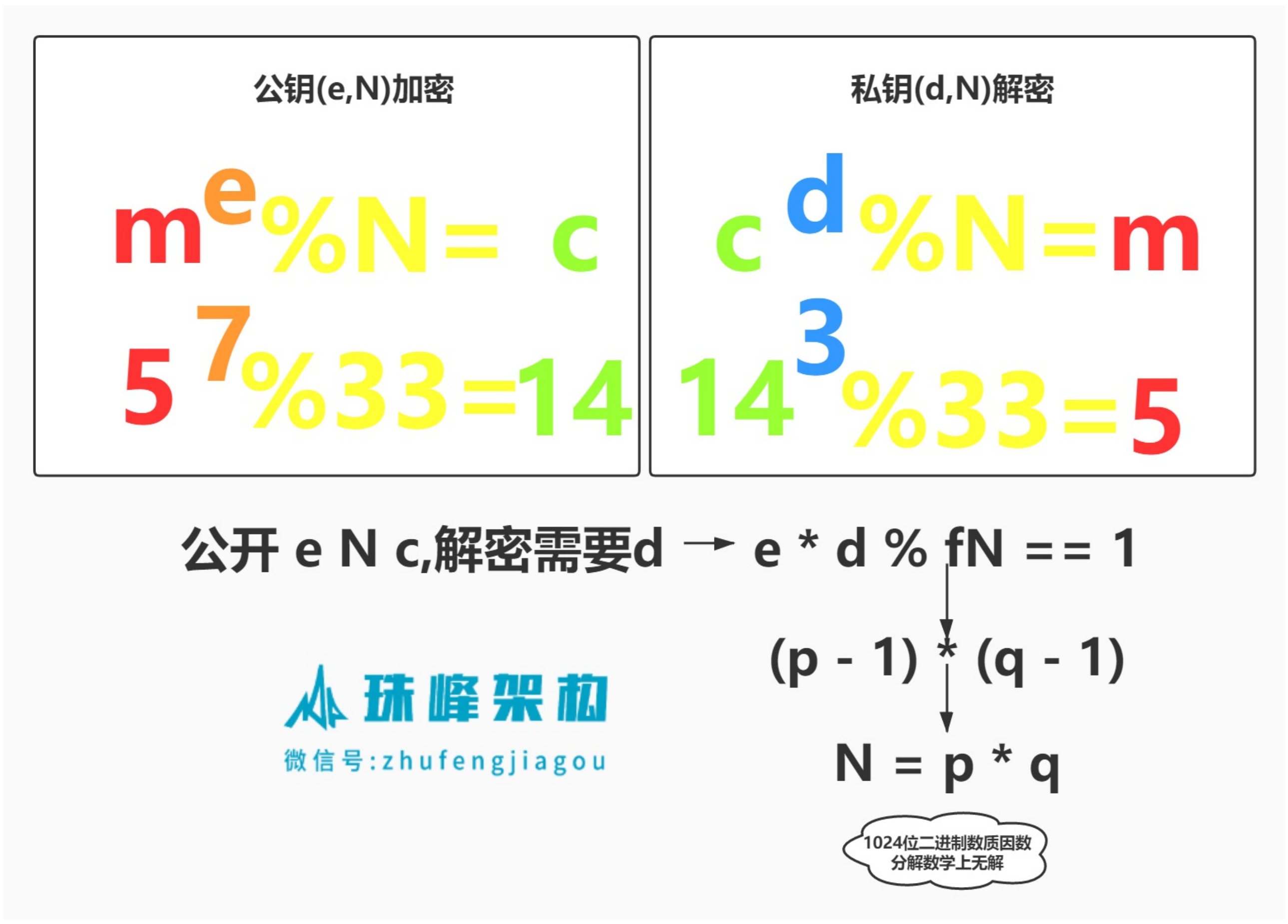
1.2 非对称加密

- 互联网上没有办法安全的交换密钥

1.2.1 单向函数

- 单向函数顺向计算起来非常的容易，但求逆却非常的困难。也就是说，已知x，我们很容易计算出f(x)。但已知f(x)，却很难计算出x
- 整数分解又称素因数分解,是将一个正整数写成几个约数的乘积
- 给出 45这个数，它可以分解成 9&#xD7;5,这样的分解结果应该是独一无二的

1.2.2 RSA加密算法



```
let p = 3, q = 11;
let N = p * q;
let fN = (p - 1) * (q - 1);
let e = 7;
for (var d = 1; e * d % fN !== 1; d++) {
    d++;
}

let publicKey = { e, N };
let privateKey = { d, N };

function encrypt(data) {
    return Math.pow(data, publicKey.e) % publicKey.N;
}
function decrypt(data) {
    return Math.pow(data, privateKey.d) % privateKey.N;
}
let data = 5;
let secret = encrypt(data);
console.log(secret);

let _data = decrypt(secret);
console.log(_data);
```

1.2.3 RSA加密

```
let { generateKeyPairSync, privateEncrypt, publicDecrypt } = require('crypto');
let rsa = generateKeyPairSync('rsa', {
    modulusLength: 1024,
    publicKeyEncoding: {
        type: 'spki',
        format: 'pem'
    },
    privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem',
        cipher: 'aes-256-cbc',
        passphrase: 'server_passphrase'
    }
});
let message = 'hello';
let enc_by_prv = privateEncrypt({
    key: rsa.privateKey, passphrase: 'server_passphrase'
}, Buffer.from(message, 'utf8'));
console.log('encrypted by private key: ' + enc_by_prv.toString('hex'));

let dec_by_pub = publicDecrypt(rsa.publicKey, enc_by_prv);
console.log('decrypted by public key: ' + dec_by_pub.toString('utf8'));
```

1.3 哈希

- hash 切碎的食物

1.3.1 哈希函数

- 哈希函数的作用是给一个任意长度的数据生成出一个固定长度的数据
- 安全性 可以从给定的数据X计算出哈希值Y，但不能从哈希值Y计算机数据X
- 独一无二 不同的数据一定会产出不同的哈希值
- 长度固定 不管输入多大的数据,输出长度都是固定的

1.3.2 哈希碰撞

- 所谓哈希(hash),就是将不同的输入映射成独一无二的、固定长度的值（又称"哈希值"）。它是最常见的软件运算之一

- 如果不同的输入得到了同一个哈希值,就发生了哈希碰撞(collision)
- 防止哈希碰撞的最有效方法，就是扩大哈希值的取值空间
- 16个二进制位的哈希值，产生碰撞的可能性是 65536 分之一。也就是说，如果有65537个用户，就一定会产生碰撞。哈希值的长度扩大到32个二进制位，碰撞的可能性就会下降到 `4,294,967,296` 分之一

```
console.log(Math.pow(2, 16));
console.log(Math.pow(2, 32));
```

1.3.3 哈希分类

- 哈希还可以叫摘要(digest)、校验值(chunkSum)和指纹(fingerPrint)
- 如果两段数据完全一样,就可以证明数据是一样的
- 哈希有二种
  - 普通哈希用来做完整性校验，流行的是MD5
  - 加密哈希用来做加密,目前最流行的加密算法是 SHA256(Secure Hash Algorithm) 系列

1.3.4 hash使用

1.3.4.1 简单哈希

```
function hash(input) {
    return input % 1024;
}
let r1 = hash(100);
let r2 = hash(1124);
console.log(r1, r2);
```

1.3.4.2 md5

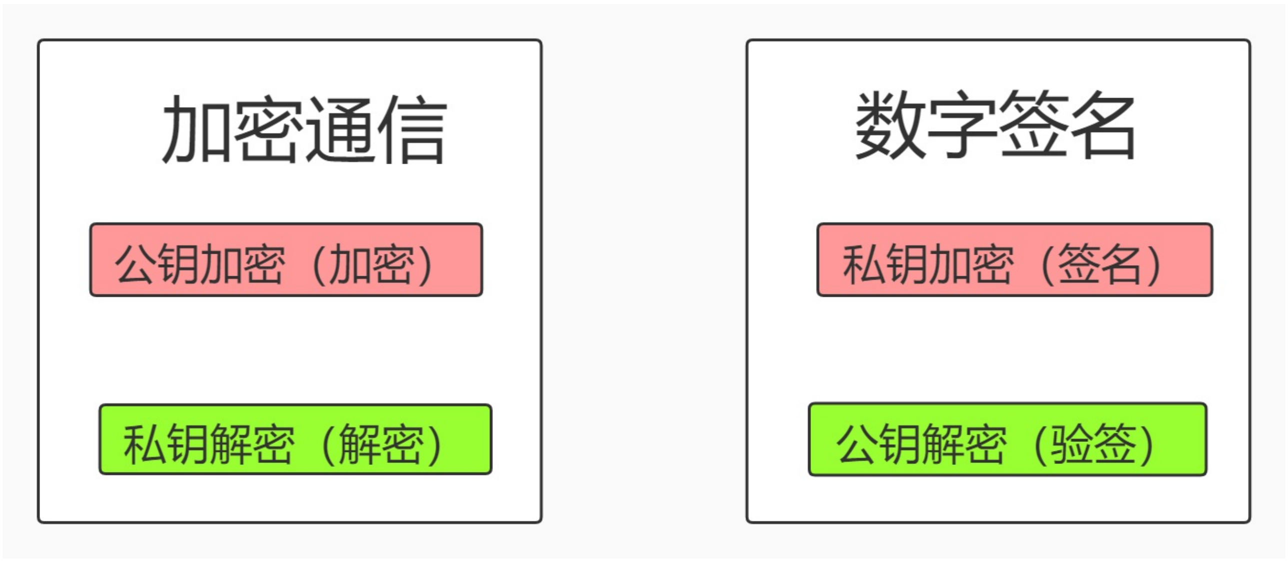
```
var crypto = require('crypto');
var content = '123456';
var result = crypto.createHash('md5').update(content).digest("hex")
console.log(result);
```

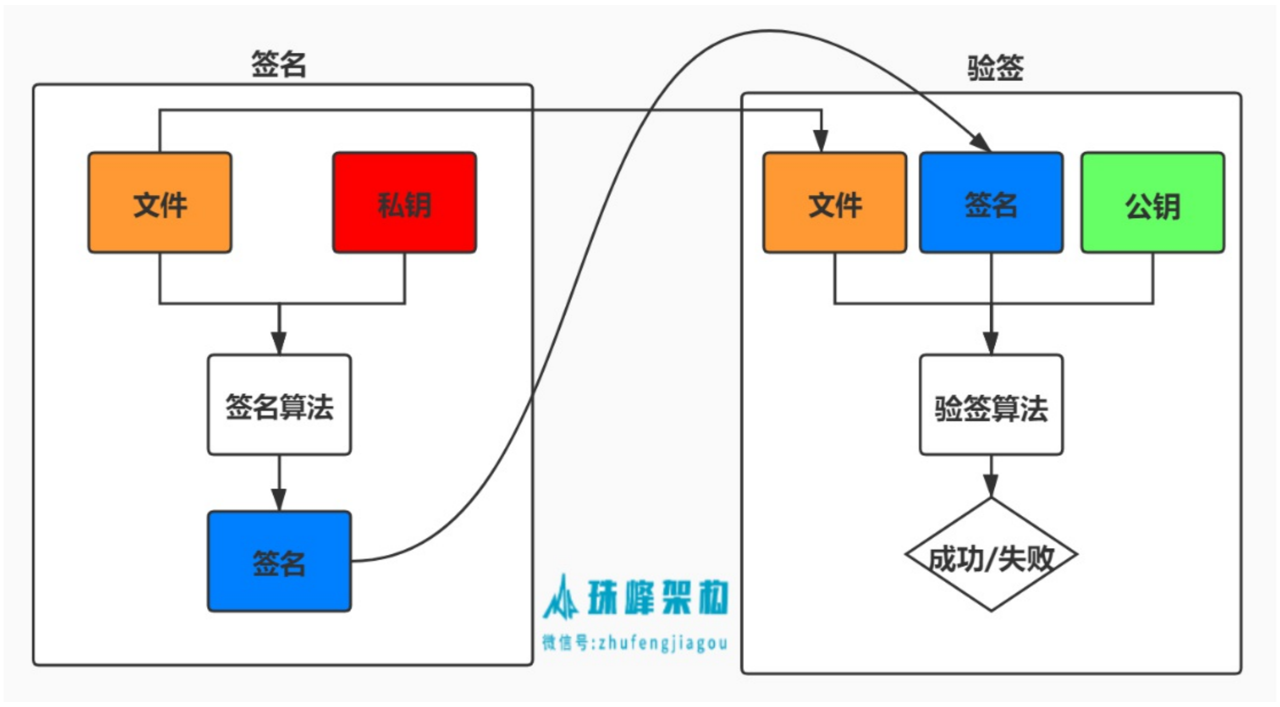
1.3.4.3 sha256

```
const salt = '123456';
const sha256 = str => crypto.createHmac('sha256', salt)
    .update(str, 'utf8')
    .digest('hex')

let ret = sha256(content);
console.log(ret);
```

1.4 数字签名

- 数字签名的基本原理是用私钥去签名，而用公钥去验证签名

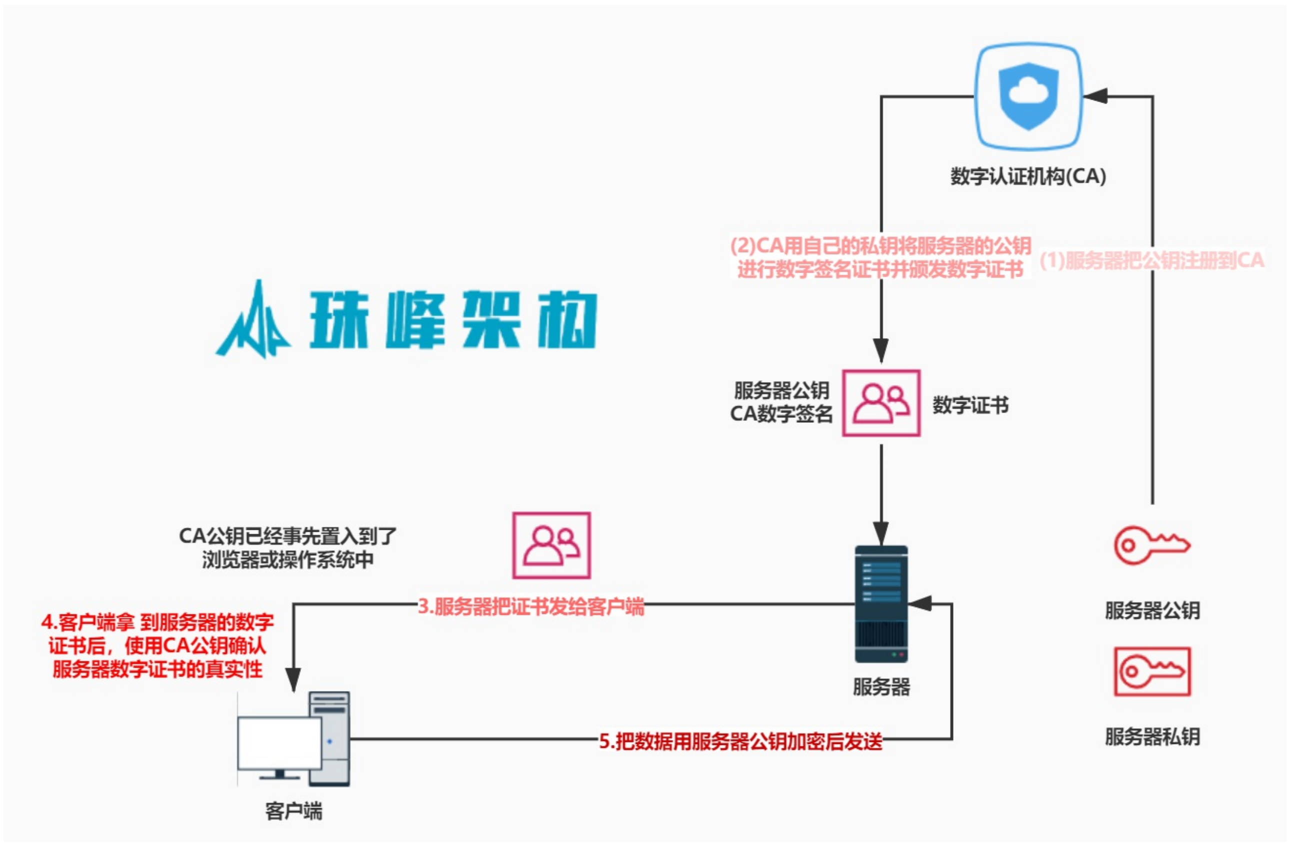




```
let { generateKeyPairSync, createSign, createVerify } = require('crypto');
let passphrase = 'zhufeng';
let rsa = generateKeyPairSync('rsa', {
    modulusLength: 1024,
    publicKeyEncoding: {
        type: 'spki',
        format: 'pem'
    },
    privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem',
        cipher: 'aes-256-cbc',
        passphrase
    }
});
let content = 'hello';
const sign = getSign(content, rsa.privateKey, passphrase);
let serverCertIsValid = verifySign(content, sign, rsa.publicKey);
console.log('serverCertIsValid', serverCertIsValid);
function getSign(content, privateKey, passphrase) {
    var sign = createSign('RSA-SHA256');
    sign.update(content);
    return sign.sign({ key: privateKey, format: 'pem', passphrase }, 'hex');
}
function verifySign(content, sign, publicKey) {
    var verify = createVerify('RSA-SHA256');
    verify.update(content);
    return verify.verify(publicKey, sign, 'hex');
}
```

1.5 数字证书

- 数字证书是一个由可信的第三方发出的，用来证明所有人身份以及所有人拥有某个公钥的电子文件

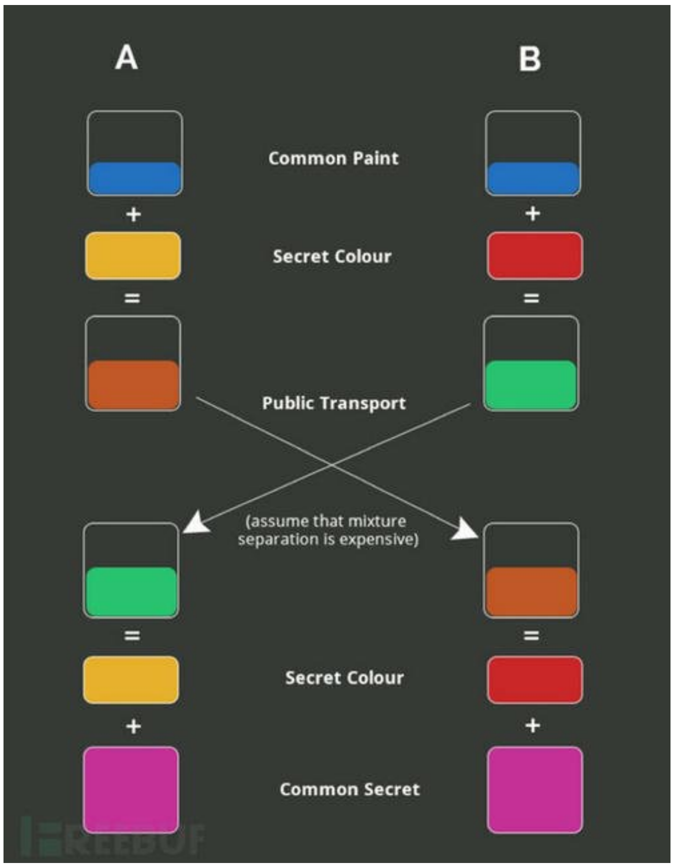

数字认证机构(CA)
(2)CA用自己的私钥将服务器的公钥进行数字签名证书并颁发数字证书
(1)服务器把公钥注册到CA
珠峰架构
服务器公钥
CA数字签名
数字证书
CA公钥已经事先置入到了浏览器或操作系统中
3.服务器把证书发给客户端
4.客户端拿 到服务器的数字证书后，使用CA公钥确认服务器数字证书的真实性
服务器
服务器公钥
服务器私钥
5.把数据用服务器公钥加密后发送
客户端

1.5.1 数字证书原理

```
let { generateKeyPairSync, createSign, createVerify, createHash } = require('crypto');
let passphrase = 'zhufeng';
let rsa = generateKeyPairSync('rsa', {
    modulusLength: 1024,
    publicKeyEncoding: {
        type: 'spki',
        format: 'pem'
    },
    privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem',
        cipher: 'aes-256-cbc',
        passphrase
    }
});
const info = {
    domain: "http://127.0.0.1:8080",
    publicKey: rsa.publicKey
};
const hash = createHash('sha256').update(JSON.stringify(info)).digest('hex');
const sign = getSign(hash, rsa.privateKey, passphrase);
const cert = { info, sign };

let certIsValid = verifySign(hash, cert.sign, rsa.publicKey);
console.log('certIsValid', certIsValid);

function getSign(content, privateKey, passphrase) {
    var sign = createSign('RSA-SHA256');
    sign.update(content);
    return sign.sign({ key: privateKey, format: 'pem', passphrase }, 'hex');
}
function verifySign(content, sign, publicKey) {
    var verify = createVerify('RSA-SHA256');
    verify.update(content);
    return verify.verify(publicKey, sign, 'hex');
}
```

1.6 Diffie-Hellman算法

- Diffie-Hellman算法是一种密钥交换协议，它可以让双方在不泄漏密钥的情况下协商出一个密钥来



1.6.1 Diffie-Hellman实现

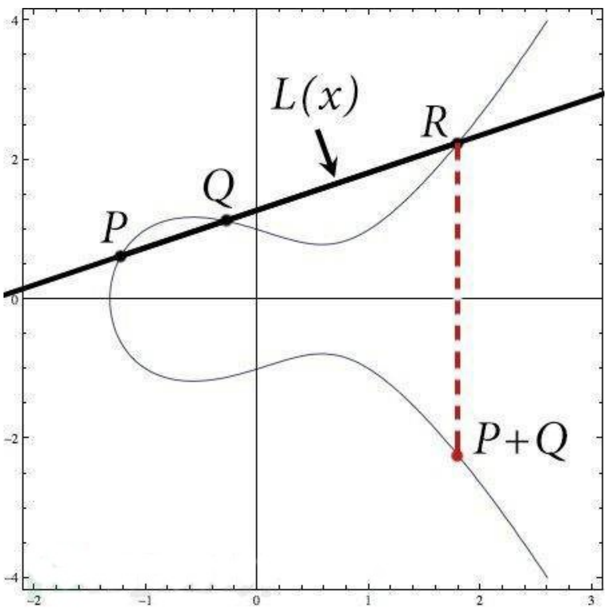
```
let N = 23;
let p = 5;
let secret1 = 6;
let A = Math.pow(p, secret1) % N;
console.log('p=', p, 'N=', N, 'A=', A);

let secret2 = 15;
let B = Math.pow(p, secret2) % N;
console.log('p=', p, 'N=', N, 'B=', B);

console.log(Math.pow(B, secret1) % N);
console.log(Math.pow(A, secret2) % N);
```

1.6.2 Diffie-Hellman算法

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

var client = createDiffieHellman(512);
var client_keys = client.generateKeys();

var prime = client.getPrime();
var generator = client.getGenerator();

var server = createDiffieHellman(prime, generator);
var server_keys = server.generateKeys();

var client_secret = client.computeSecret(server_keys);
var server_secret = server.computeSecret(client_keys);

console.log('client_secret: ' + client_secret.toString('hex'));
console.log('server_secret: ' + server_secret.toString('hex'));
```

1.7 ECC

- 椭圆曲线加密算法(ECC) 是基于椭圆曲线数学的一种公钥加密的算法



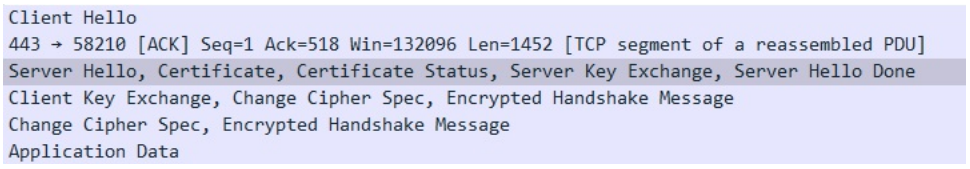
1.7.1 ECC原理

```
let G = 3;
let a = 5;
let A = G * a;
let b = 7;
let B = G * b;
console.log(a * B);
console.log(b * A);
```

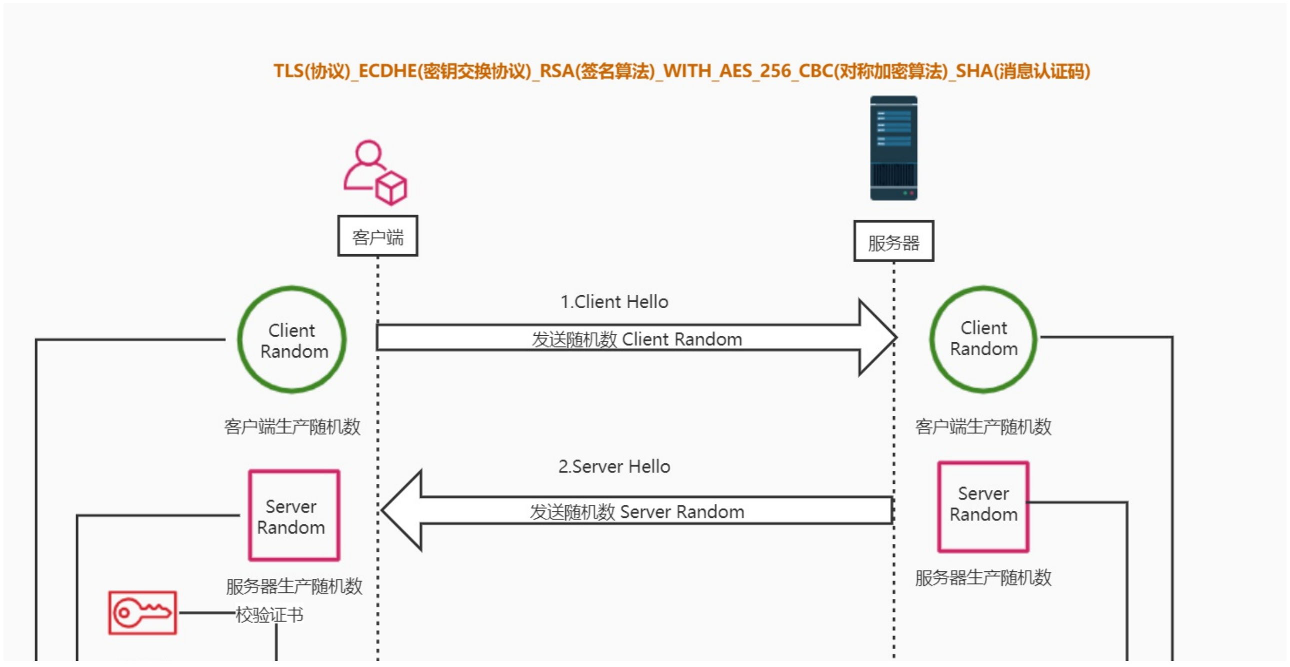
1.7.2 ECC使用

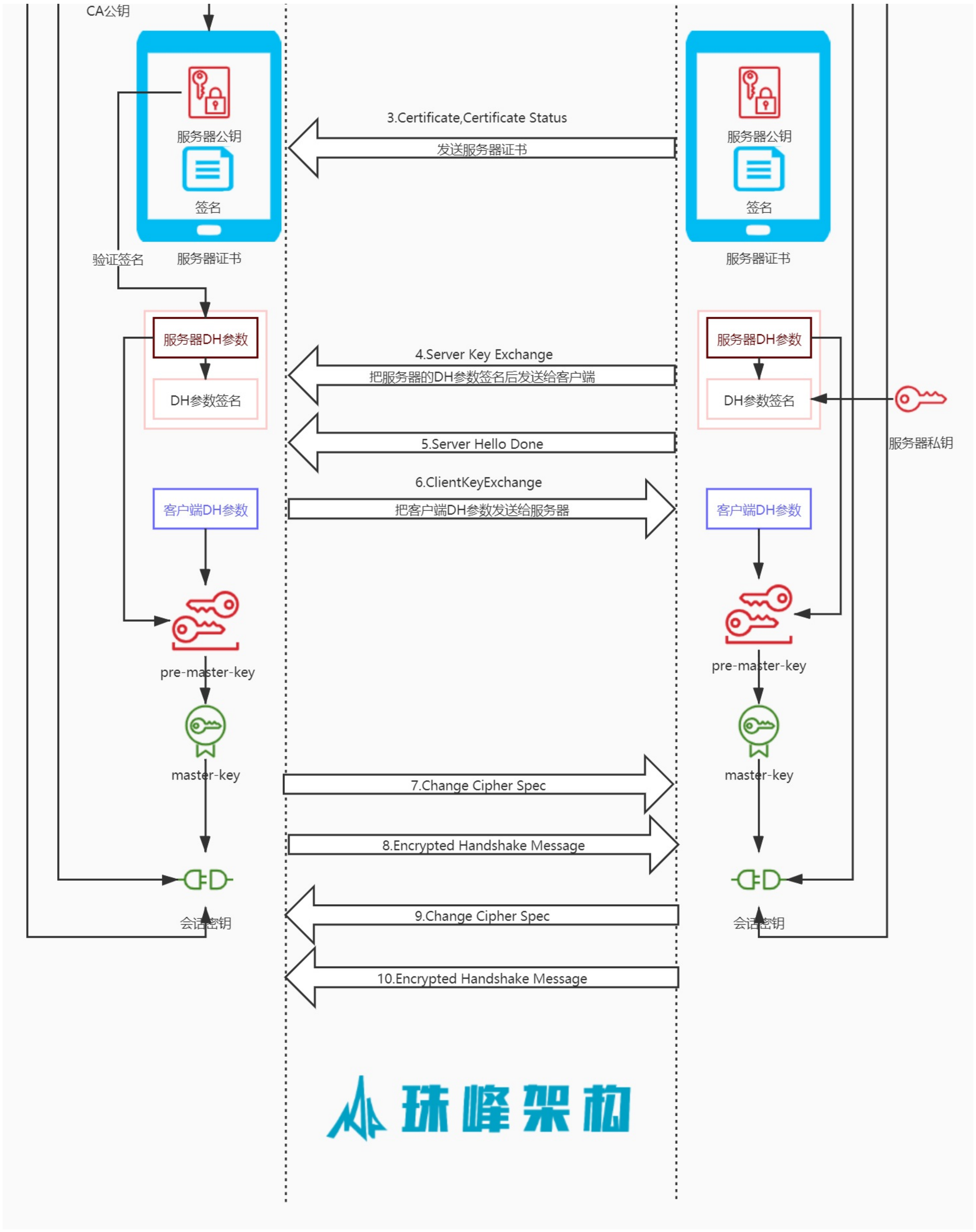
```
let { createECDH } = require('crypto');
const clientDH = createECDH('secp521r1');
const clientDHParams = clientDH.generateKeys();

const serverDH = createECDH('secp521r1');
const serverDHParams = serverDH.generateKeys();

const clientKey = clientDH.computeSecret(serverDHParams);
const serverKey = serverDH.computeSecret(clientDHParams);
console.log('clientKey', clientKey.toString('hex'));
console.log('serverKey', serverKey.toString('hex'));
```

2. UDP服务器

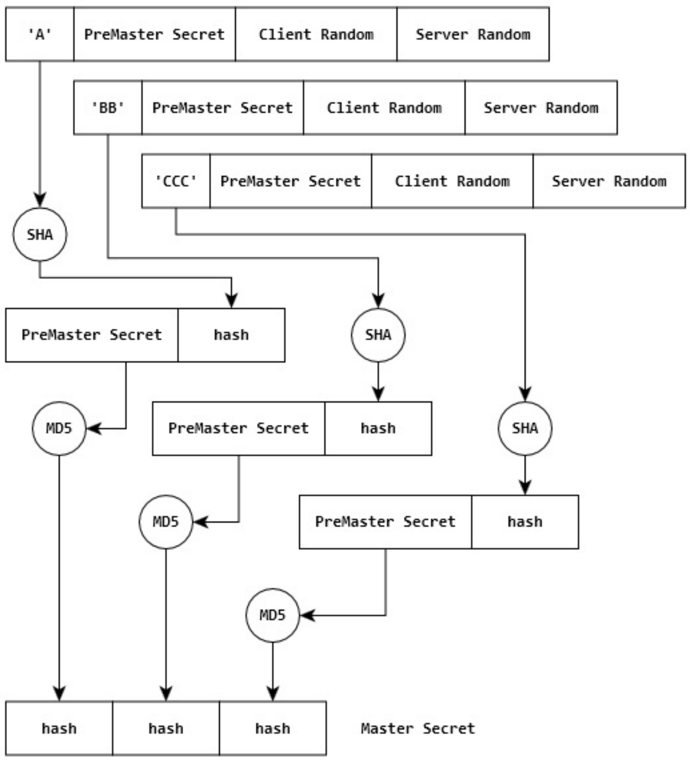


```
Client Hello
443 → 58210 [ACK] Seq=1 Ack=518 Win=132096 Len=1452 [TCP segment of a reassembled PDU]
Server Hello, Certificate, Certificate Status, Server Key Exchange, Server Hello Done
Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
Change Cipher Spec, Encrypted Handshake Message
Application Data
```





CA公钥
服务器公钥
签名
服务器证书
验证签名
3.Certificate,Certificate Status
发送服务器证书
服务器公钥
签名
服务器证书
服务器DH参数
DH参数签名
4.Server Key Exchange
把服务器的DH参数签名后发送给客户端
服务器DH参数
DH参数签名
服务器私钥
5.Server Hello Done
6.ClientKeyExchange
把客户端DH参数发送给服务器
客户端DH参数
客户端DH参数
pre-master-key
pre-master-key
master-key
master-key
7.Change Cipher Spec
8.Encrypted Handshake Message
9.Change Cipher Spec
10.Encrypted Handshake Message
会话密钥
会话密钥
珠峰架构



wireshark

```
tls and (ip.src == 47.111.100.159 or ip.dst == 47.111.100.159)
```

2.1 createCA.js

```
const ca_passphrase = 'ca';
let CA = generateKeyPairSync('rsa', {
    modulusLength: 1024,
    publicKeyEncoding: {
        type: 'spki',
        format: 'pem'
    },
    privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem',
        cipher: 'aes-256-cbc',
        passphrase: ca_passphrase
    }
});
let fs = require('fs');
let path = require('path');
fs.writeFileSync(path.resolve(__dirname, 'CA.publicKey'), CA.publicKey);
fs.writeFileSync(path.resolve(__dirname, 'CA.privateKey'), CA.privateKey);
```

2.2 ca.js

```
const { createHash } = require('crypto');
const { getSign } = require('./utils');
const ca_passphrase = 'ca';
const fs = require('fs');
const path = require('path');
let cAPrivateKey = fs.readFileSync(path.resolve(__dirname, 'CA.privateKey'), 'utf8');
function requestCert(info) {
    const infoHash = createHash('sha256').update(JSON.stringify(info)).digest('hex');
    const sign = getSign(infoHash, cAPrivateKey, ca_passphrase);
    return { info, sign };
}
exports.requestCert = requestCert;
```

2.3 utils.js

```
const { createCipheriv, createDecipheriv, createSign, createVerify } = require('crypto');
function encrypt(data, key) {
    let decipher = createCipheriv('aes-256-cbc', key, '1234567890123456');
    decipher.update(data);
    return decipher.final('hex');
}

function decrypt(data, key) {
    let decipher = createDecipheriv('aes-256-cbc', key, '1234567890123456');
    decipher.update(data, 'hex');
    return decipher.final('utf8');
}
function getSign(content, privateKey, passphrase) {
    var sign = createSign('RSA-SHA256');
    sign.update(content);
    return sign.sign({ key: privateKey, format: 'pem', passphrase }, 'hex');
}
function verifySign(content, sign, publicKey) {
    var verify = createVerify('RSA-SHA256');
    verify.update(content);
    return verify.verify(publicKey, sign, 'hex');
}
module.exports = {
    encrypt, decrypt, getSign, verifySign
}
```

2.4 udp_server.js

```
const dgram = require('dgram')
const udp_server = dgram.createSocket('udp4')
const protocol = require('./protocol');
const { generateKeyPairSync, randomBytes, createHash, createECDH } = require('crypto');
const server_passphrase = 'server';
const { getSign, decrypt, encrypt } = require('./utils');
const { requestCert } = require('./ca');

let serverRSA = generateKeyPairSync('rsa', {
    modulusLength: 1024,
    publicKeyEncoding: {
        type: 'spki',
        format: 'pem'
    },
    privateKeyEncoding: {
        type: 'pkcs8',
        format: 'pem',
        cipher: 'aes-256-cbc',
        passphrase: server_passphrase
    }
});
let serverRandom = randomBytes(8).toString('hex');
const serverInfo = {
    domain: "http://127.0.0.1:20000",
    publicKey: serverRSA.publicKey
};
let serverCert = requestCert(serverInfo);
let clientRandom;
const serverDH = createECDH('secp521r1');
const ecDHServerParams = serverDH.generateKeys().toString('hex');
const ecDHServerParamsSign = getSign(ecDHServerParams, serverRSA.privateKey, server_passphrase);
let masterKey;
let sessionKey;
udp_server.on('listening', () => {
    const address = udp_server.address();
    console.log(`client running ${address.address}: ${address.port}`)
})
udp_server.on('message', (data, remote) => {
    let message = JSON.parse(data);
    switch (message.type) {
        case protocol.ClientHello:

            clientRandom = message.clientRandom;
            udp_server.send(JSON.stringify({
                type: protocol.ServerHello,
                serverRandom,
                cipherSuite: 'TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA'
            }), remote.port, remote.address);

            udp_server.send(JSON.stringify({
                type: protocol.Certificate,
                serverCert,
            }), remote.port, remote.address);

            udp_server.send(JSON.stringify({
                type: protocol.ServerKeyExchange,
                ecDHServerParams,
                ecDHServerParamsSign
            }), remote.port, remote.address);

            udp_server.send(JSON.stringify({
                type: protocol.ServerHelloDone
            }), remote.port, remote.address);
            break;
        case protocol.ClientKeyExchange:

            let { ecDHClientParams } = message;
            preMasterKey = serverDH.computeSecret(Buffer.from(ecDHClientParams, 'hex')).toString('hex');
            masterKey = createHash('md5').update(preMasterKey + clientRandom + serverRandom).digest('hex');
            sessionKey = createHash('md5').update(masterKey + clientRandom + serverRandom).digest('hex');
            break;
        case protocol.ChangeCipherSpec:

            udp_server.send(JSON.stringify({
                type: protocol.ChangeCipherSpec
            }), remote.port, remote.address);
            break;
        case protocol.EncryptedHandshakeMessage:
            console.log("服务器收到解密后的数据:", decrypt(message.data, sessionKey));

            udp_server.send(JSON.stringify({
                type: protocol.EncryptedHandshakeMessage,
                data: encrypt("i am server", sessionKey)
            }), remote.port, remote.address);
            break;
        default:
            break;
    }

})
udp_server.on('error', (error) => {
    console.log(error);
});
udp_server.bind(20000, '127.0.0.1');
```

2.5 udp_client.js

```
const dgram = require('dgram')
const udp_client = dgram.createSocket('udp4')
const { randomBytes, createHash, createECDH } = require('crypto');
const { verifySign, encrypt, decrypt } = require('./utils');
const url = require('url');
const protocol = require('./protocol');
const fs = require('fs');
const path = require('path');
const cAPublicKey = fs.readFileSync(path.resolve(__dirname, 'CA.publicKey'), 'utf8');
const clientRandom = randomBytes(8).toString('hex');
let serverRandom;
let serverPublicKey;
let ecDHServerParams;
let clientDH = createECDH('secp521r1');
let ecDHClientParams = clientDH.generateKeys();
let masterKey;
let sessionKey;
udp_client.on('listening', () => {
    const address = udp_client.address();
    console.log(`client running ${address.address}: ${address.port}`)
})
udp_client.on('message', (data, remote) => {
    let message = JSON.parse(data.toString('utf8'));
    switch (message.type) {
        case protocol.ServerHello:
            serverRandom = message.serverRandom;
            break;
        case protocol.Certificate:

            let { serverCert } = message;
            let { info, sign } = serverCert;
            serverPublicKey = info.publicKey;
            const serverInfoHash = createHash('sha256').update(JSON.stringify(info)).digest('hex');
            let serverCertIsValid = verifySign(serverInfoHash, sign, cAPublicKey);
            console.log('验证服务器端证书是否正确?', serverCertIsValid);
            let urlObj = url.parse(info.domain);
            let serverDomainIsValid = urlObj.hostname === remote.address && urlObj.port == remote.port;
            console.log('验证服务器端域名正确?', serverDomainIsValid);
            break;
        case protocol.ServerKeyExchange:

            ecDHServerParams = message.ecDHServerParams;
            ecDHServerParamsSign = message.ecDHServerParamsSign;
            let serverDHParamIsValid = verifySign(ecDHServerParams, ecDHServerParamsSign, serverPublicKey);
            console.log('验证服务器端证书DH参数是否正确?', serverDHParamIsValid);
            break;
        case protocol.ServerHelloDone:

            udp_client.send(JSON.stringify({
                type: protocol.ClientKeyExchange,
                ecDHClientParams
            }), remote.port, remote.address);

            preMasterKey = clientDH.computeSecret(Buffer.from(ecDHServerParams, 'hex')).toString('hex');
            masterKey = createHash('md5').update(preMasterKey + clientRandom + serverRandom).digest('hex');
            sessionKey = createHash('md5').update(masterKey + clientRandom + serverRandom).digest('hex');

            udp_client.send(JSON.stringify({
                type: protocol.ChangeCipherSpec
            }), remote.port, remote.address);

            udp_client.send(JSON.stringify({
                type: protocol.EncryptedHandshakeMessage,
                data: encrypt("i am client", sessionKey)
            }), remote.port, remote.address);
            break;
        case protocol.EncryptedHandshakeMessage:

            console.log("客户端收到解密后的数据:", decrypt(message.data, sessionKey));
            break;
        default:
            break;
    }
})
udp_client.on('error', (error) => {
    console.log(error);
});

udp_client.send(JSON.stringify({
    type: protocol.ClientHello,
    clientRandom
}), 20000, '127.0.0.1');
```

3. 数字证书实战

- OpenSSL (https://www.openssl.org/source/)
- WinOpenSSL (http://slproweb.com/products/Win32OpenSSL.html)
- 安装后要添加环境变量 `C:\Program Files\OpenSSL-Win64\bin`

3.1 自建CA#

```
openssl genrsa -des3 -out ca.private.pem 1024

openssl req -new -key ca.private.pem -out ca.csr

openssl x509 -req -in ca.csr -extensions v3_ca -signkey ca.private.pem -out ca.crt
```

3.2 生成服务器CA证书

```
openssl genrsa -out server.private.pem 1024

openssl req -new -key server.private.pem -out server.csr

openssl x509 -days 365 -req -in server.csr -extensions  v3_req -CAkey  ca.private.pem -CA ca.crt -CAcreateserial -out server.crt  -extfile openssl.cnf
```

openssl.cnf

```
[req]
    distinguished_name = req_distinguished_name
    req_extensions = v3_req
    [req_distinguished_name]
    countryName = CN
    countryName_default = CN
    stateOrProvinceName = Beijing
    stateOrProvinceName_default = Beijing
    localityName = Beijing
    localityName_default = Beijing
    organizationalUnitName  = HD
    organizationalUnitName_default  = HD
    commonName = localhost
    commonName_max  = 64

    [ v3_req ]
    # Extensions to add to a certificate request
    basicConstraints = CA:FALSE
    keyUsage = nonRepudiation, digitalSignature, keyEncipherment
    subjectAltName = @alt_names

    [alt_names]

#&#x6CE8;&#x610F;&#x8FD9;&#x4E2A;IP.1&#x7684;&#x8BBE;&#x7F6E;&#xFF0C;IP&#x5730;&#x5740;&#x9700;&#x8981;&#x548C;&#x4F60;&#x7684;&#x670D;&#x52A1;&#x5668;&#x7684;&#x76D1;&#x542C;&#x5730;&#x5740;&#x4E00;&#x6837; DNS&#x4E3A;server&#x7F51;&#x5740;&#xFF0C;&#x53EF;&#x8BBE;&#x7F6E;&#x591A;&#x4E2A;ip&#x548C;dns
    IP.1 = 127.0.0.1
    DNS.1 = localhost
```

3.3 服务端

```
const https = require('https');
const fs = require('fs');
const path = require('path');

const options = {
    key: fs.readFileSync(path.resolve(__dirname, 'ssl/server.private.pem')),
    cert: fs.readFileSync(path.resolve(__dirname, 'ssl/server.crt'))
};

https.createServer(options, (req, res) => {
    res.end('hello world\n');
}).listen(9000);

console.log("server https is running 9000");
```

3.4 客户端

```
const https = require('https');
const options = {
    hostname: '127.0.0.1',
    port: 9000,
    path: '/',
    method: 'GET',
    requestCert: true,
    rejectUnauthorized: false,
};

const req = https.request(options, (res) => {
    let buffers = [];
    res.on('data', (chunk) => {
        buffers.push(chunk);
    });
    res.on('end', () => {
        console.log(buffers.toString());
    });
});
req.end();
```